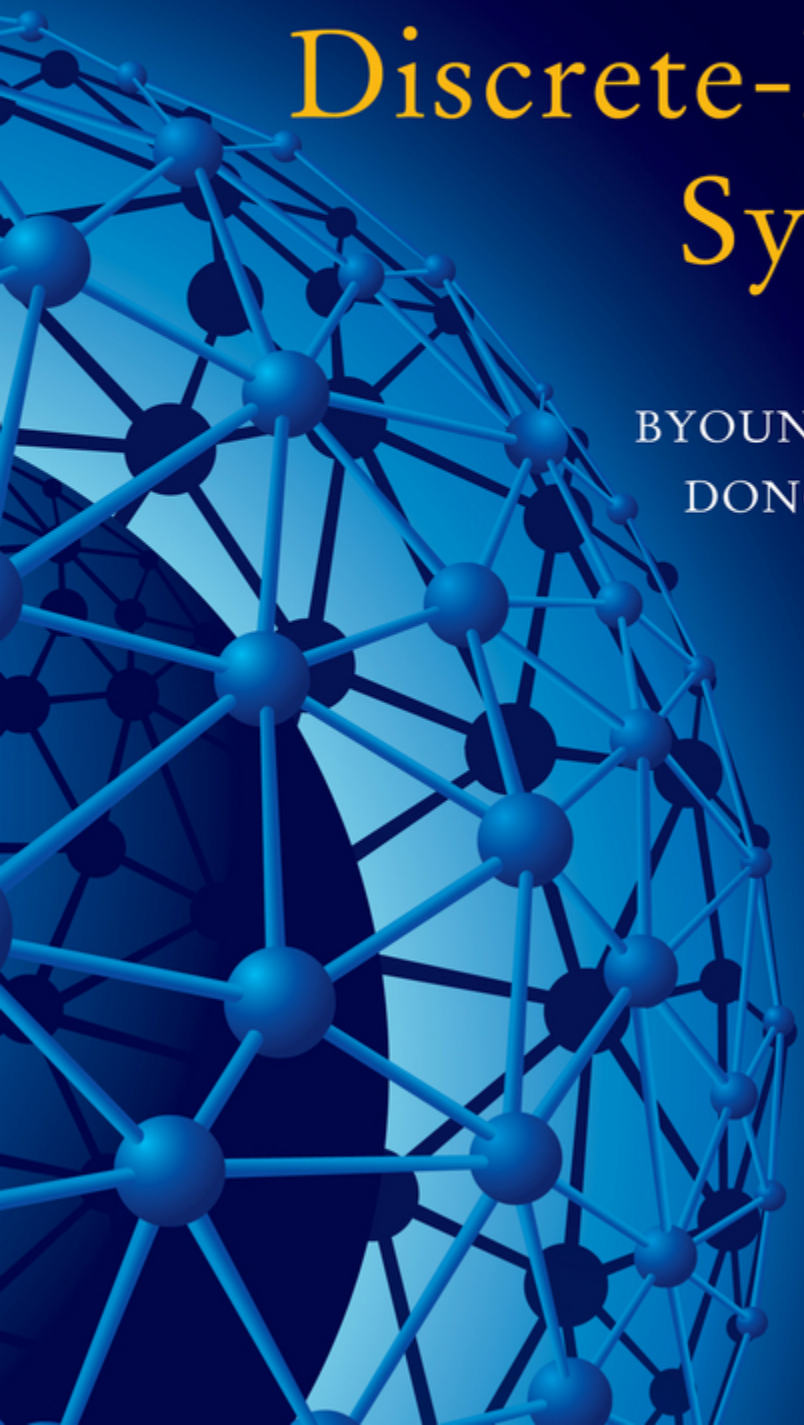
Modeling *and* Simulation *of* Discrete-Event Systems

BYOUNG KYU CHOI
DONGHUN KANG



WILEY

MODELING AND SIMULATION OF DISCRETE-EVENT SYSTEMS

MODELING AND SIMULATION OF DISCRETE-EVENT SYSTEMS

Byoung Kyu Choi
Department of Industrial and Systems Engineering, KAIST, South Korea
Department of Computer Science, KAU, Saudi Arabia

Donghun Kang
Department of Industrial and Systems Engineering, KAIST, South Korea

WILEY

Published by John Wiley & Sons, Inc., Hoboken, New Jersey
Published simultaneously in Canada

Library of Congress Cataloging-in-Publication Data:
Choi, Byoung Kyu, 1949–
Modeling and simulation of discrete-event systems / Byoung Kyu Choi, Donghun Kang.
pages cm
Includes index.
ISBN 978-1-118-38699-6 (cloth)
1. Discrete-time systems–Simulation methods. I. Kang, Donghun, 1981– II. Title.
T57.62.C377 2013
003'.83–dc23
2013013970

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

CONTENTS

Online Supplements

Numerous supplemental materials including software downloads are provided on the official website of the book at http://VMS-technology.com/book. The supplemental materials are grouped into (1) M&S practices with commercial simulators, (2) developing your own dedicated simulators, and (3) integrated simulation of electronics Fabs. The commercials simulators covered are an event-based simulator SIGMA®, an activity-based simulator ACE®, a state-based simulator SGS®, and an entity-based simulator Arena®.

PREFACE

This book provides comprehensive, in-depth coverage of modeling and simulation (M&S) of discrete-event systems (DESs). Here, the term *M&S* refers to computer simulation, with an emphasis on modeling real-life DESs and executing the models. The current state-of-the-art in DES M&S is a result of the breakthroughs in the following areas: (1) activity-based modeling formalism pioneered by K.D. Tocher in late 1950s; (2) the advent of process-oriented simulation languages, such as GPSS and SLAM, in the early 1970s; (3) state-based modeling formalism, or DEVS, founded by Bernard Zeigler in the mid-1970s; and (4) event-based modeling formalism as matured by Lee Schruben since the early 1980s.

There exists at least one classic textbook in each area—a textbook on activity-based modeling by Carrie, a few books on state-based (DEVS) modeling by Zeigler, a textbook on event-based modeling by Schruben, and a few books on process-oriented languages such as Arena® and ProModel®. In addition, there are quite a few books focusing on statistical notions of computer simulation. The researchers in each area advocate their own views as central to DES M&S. Only a couple of books (e.g., Fishwick) propose an integrated model engineering framework.

This book presents an integrated M&S framework covering all four DES M&S breakthrough areas. It is a product of 30 years of teaching at KAIST, as well as sponsored research and development projects at the authors' lab at KAIST, VMS (virtual manufacturing system) Lab, which has been a government-endowed National Research Lab since 1999. In particular, the practice-oriented theme of this book is a result of the authors' decade-long experience in developing simulation-based scheduling (SBS) solutions for Samsung Electronics and other companies in Korea. Virtually all the Samsung's semiconductor fabrication plants (Fabs) and flat panel display (FPD) Fabs are run utilizing solutions originated by the authors' lab, and upgraded and supported by a spin-off venture company.

This book is divided into three parts: Part I, *Basics of System Modeling and Simulation*; Part II, *Fundamentals of Discrete-Event System Modeling and Simulation*; and Part III, *Advances in Discrete-Event System Modeling and Simulation*. Parts I and II are designed as a primary textbook for an undergraduate level M&S course in Industrial Engineering, Computer Science, and Management Science. With Part III, it is designed as a graduate-level course. This book comprehensively covers the state-of-the art modeling formalisms

and execution algorithms in DES M&S thereby serving as a main reference for M&S researchers in academia. This book provides an easy-to-understand guide for simulation practitioners in industry using off-the-shelf simulators such as SIGMA® and Arena®. Finally, this book reveals a number of "secrets" for developing your own simulators: event graph simulator, ACD simulator, state graph simulator, and integrated Fab simulator—making it a valuable resource for M&S solution developers.

The book is largely self-contained, and few prerequisites are needed for understanding its main contents. However, some prior knowledge will help readers understand specific sections:

(1) Basic knowledge of statistics and probability (Chapter 3, Input Modeling);
(2) Basic knowledge of linear algebra (Chapter 8, Output Analysis and Optimization)
(3) Experience with computer programming (Sections on developing your own simulators, e.g. Sections 5.7 and 10.2).

Perhaps the most critical prerequisite for mastering this book is enthusiasm and commitment toward M&S. This book is about the art of M&S, and like other art forms, can only be mastered through persistent practice.

The authors wish to express their special thanks to Prof. K.H. Han of Gyeongsang National University for using part of this book in his class and providing valuable comments that led to its improvement; to Prof. I.K. Moon of Seoul National University and Prof. S.C. Park of Ajou University for their input during the early stage of writing this book; and to Prof. Lee Schruben of Berkeley for his encouragement and support. For developing sample models and exercise problems, and for executing "prototype" simulation models appearing in the book, we would like to thank our graduate students in the VMS Lab at KAIST, especially H.S. Kim, T.J. Choi, and E.H. Song.

Finally, Byoung Choi thanks his wife, Yong, and his son and best friend Samuel, for their support and encouragement. Donghun Kang thanks his parents for their loving care and support.

Byoung Kyu Choi
Donghun Kang

Daejeon, Korea, June 2013

ABBREVIATIONS

ACD	activity cycle diagram
AGV	automated guided vehicle
AMHS	automated material handling system
AON	activity-on-node
AQL	average queue length
AST	average sojourn time
ATM	automatic teller machine
ATT	activity transition table
AWT	average waiting time
BTO	bound-to-occur
CAL	candidate activity list
DES	discrete-event system
EFD	entity flow diagram
EG	event graph
EO	event object
EOS	end of simulation
Fab	Fabrication line (or plant)
FED	federation execution data
FEL	future event list
FIFO	first-in-first-out
FMS	flexible manufacturing system
FOM	federation object model
FPD	flat panel display
FSA	finite state automata
FSM	finite state machine
HLA	high level architecture
IFS	integrated factory simulator
LCD	liquid crystal display
LEL	local event list
LP	logical process
M&S	modeling and simulation
MCS	material control system
MDP	message delivery packet
MEL	move-type event list
MLE	maximum likelihood estimator
MS	mean square
MSR	message send request
OOEG	object-oriented event graph
P-ACD	parameterized ACD
PDM	process definition model
PEG	parameterized event graph
RSM	response surface methodology
RTD	real-time dispatcher
RTI	run time infrastructure
SG	state graph
SOM	simulation object model
SS	sum of square
TAG	time advance grant
TAR	time advance request
VE	virtual environment
WfMS	workflow management system

PART I

BASICS OF SYSTEM MODELING AND SIMULATION

We think by "constructing mental models and then simulating them in order to draw conclusions or make decisions." Thus, modeling and simulation (M&S) constitutes the central part of our thinking process. "I think, therefore I am" is a philosophical statement used by the French philosopher Descartes, which became a foundational element of Western philosophy. Therefore, if we combine the philosophical notion of thinking with the engineering definition of M&S, we may say that "we are engineers and scientists because we can model systems and simulate them." Furthermore, if our brain is not powerful enough to simulate a given complex system, we rely on computers to perform a computer simulation.

A dictionary definition of *simulation* is the technique of imitating the behavior of some situation by means of an analogous situation or apparatus to gain information more conveniently or to train personnel, while an academic definition of *computer simulation* is the discipline of designing a model of a system, simulating the model on a digital computer, and analyzing the execution output. In recent years, the term *modeling and simulation* (M&S) seems to be preferred to the term for computer simulation, with an emphasis on modeling. Part I of this book has two chapters, and it aims to provide the readers with a basic but comprehensive treatment of computer simulation.

Chapter 1, "Overview of Computer Simulation," will provide answers to the following basic questions in computer simulation:

1. What is a system?
2. What is computer simulation?
3. What is discrete-event simulation?
4. What is continuous simulation?
5. What is Monte Carlo simulation?
6. What are simulation experimentation and optimization?

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.

Chapter 2, "Basics of Discrete-Event System Modeling & Simulation," aims to provide answers to the following basic questions in discrete-event system (DES) M&S:

1. How is a discrete-event simulation carried out?
2. What are modeling components and a reference model?
3. What is a discrete-event system modeling formalism?
4. What is a formal model and how it is specified?
5. What is the integrated framework of discrete-event system modeling?
6. How do we build and simulate an event graph or activity cycle diagram (ACD) model of a DES?
7. How is the M&S life cycle managed?

CHAPTER 1

Overview of Computer Simulation

The wise man is one who knows what he does not know.

—Tao Te Ching

1.1 INTRODUCTION

Richmond [2003] defines thinking as "constructing mental models and then simulating them in order to draw conclusions or make decisions." Namely, he defines thinking as mental simulation. When the situation is too complex to be analyzed by mental simulation alone, we rely on computer simulation. According to Schruben [2012], simulation models provide unlimited virtual power: "If you can think of something, you can simulate it. Experimenting in a simulated world, you can change anything, in any way, at any time—even change time itself."

Fishwick [1995] defines *computer simulation* as the discipline of designing a model of a system, simulating the model on a digital computer, and analyzing the execution output. In the military, where computer simulation is extensively used in training personnel (e.g., war game simulation) and acquiring weapon systems (e.g., simulation-based acquisition), the term *modeling and simulation* (M&S) is used in place of *computer simulation*. In this book, these two terms are used interchangeably.

The purpose of this chapter is to provide the reader with a basic understanding of computer simulation. After studying this chapter, you should be able to answer the following questions:

1. What are the common characteristics that lead to a conceptual definition of system?
2. What are the three types of systems?
3. What are the three subsystems in a feedback control system?
4. What are the three types of virtual environment simulation?

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.

5. What are the three types of computer simulation?
6. What is the simulation model trajectory of a discrete-event system?
7. What is Monte Carlo simulation?
8. What is sensitivity analysis in simulation experimentation?

This chapter is organized as follows: Definitions and structures of systems are given in Section 1.2. Section 1.3 provides definitions and applications of simulation. The subsequent three sections introduce the three simulation types: discrete-event simulation in Section 1.4, continuous simulation in Section 1.5, and Monte Carlo simulation in Section 1.6. Finally, a basic framework of simulation experimentation is presented in Section 1.7.

1.2 WHAT IS A SYSTEM?

1.2.1 Definitions of Systems

Systems are encountered everywhere in the world. While those systems differ in their specifics, they share common characteristics that lead to a conceptual definition of a system. In Wu [1992], a *system* is defined as "a collection of components which are interrelated in an organized way and work together towards the accomplishment of certain logical and purposeful end." Thus, any portion of the real world may be defined as a system if it has the following characteristics: (1) it has a purpose or purposes, (2) its components are connected in an organized manner, and (3) they work together to achieve common objectives. A system consisting of people is often called a *team*. Needless to say, a mere crowd of people sharing no common objectives is not a team.

When defining a system, the concept of state variable plays a key role. A *state variable* is a particular measurable property of an object or system. Examples of state variables are the number of jobs in a buffer, status of a machine, temperature of an oven, etc. A system in which the state variables change instantaneously at discrete points in time is called a *discrete-event system*, whereas a system in which state variables change continuously over time is called a *continuous system*.

1.2.2 Three Types of Systems

Our universe, which is full of systems everywhere, may be viewed from the five levels of detail (Fig. 1.1): from the subatomic level to cosmological level. In the subatomic level, interactions among the components of a system are described using quantum mechanics, which is a physical science dealing with the behavior of matter and energy on the scale of atoms and subatomic particles. It is interesting to find that quantum mechanics is also used in modeling a system at the cosmological level [Mostafazadeh 2004]. Thus, a system in the subatomic level or cosmological level may be called a *quantum system*.

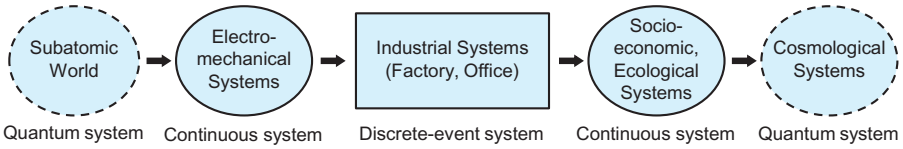


Fig. 1.1. Five levels of details of system definitions in the universe.

A system in the electromechanical level usually has components whose physical dynamics are described using differential equations of effort, such as force and voltage, and flow, such as velocity and current [Karnopp et al. 2000]. The behaviors of ecological systems and socioeconomic systems are usually described using differential equations of flow [Hannon and Ruth 2001]. As a result, these systems are called a *continuous system* or a *differential equation system*.

Systems in the middle level are industrial systems which are more conveniently described in terms of discrete events, and they are discrete-event systems. An event is an instance of changes in state variables. A special type of this system is a digital system such as a computer whose states are defined by a finite number of 0s and 1s.

1.2.3 System Boundaries and Hierarchical Structure

Everything in our world is connected to everything else in some way, which is known as the small world phenomenon [Kleinberg 2000]. Thus, in order to define a system, it is first necessary to isolate the components of the system from the remaining world and to enclose them within a system boundary.

A set of isolated components of primary interest is called a *target system*. The target system may have a number of subsystems, and it may be a subsystem of a higher-level system called a *wider system*. The wider system is separated from the external environment by a boundary [Wu 1992]. In summary, a typical system consists of a target system (composed of its subsystems) and a wider system (in which the target system is included). The system of interest consisting of a target system and its wider system is often referred to as a *source system*.

Most dynamic systems in engineering and management are feedback control systems. Key subsystems in a feedback control system are operational, monitoring, and decision-making subsystems. The operational subsystem carries out the system's tasks, and the monitoring subsystem monitors system performances and reports to the decision-making subsystem. The decision-making subsystem is responsible for making decisions and taking corrective actions. The relationships among the target feedback control system, its subsystems, wider system, and external environment are shown in Fig. 1.2 [Wu 1992]. For example, if your simulation study is focused on an emergency room of a hospital, the emergency room would become the target system and the hospital the wider system.

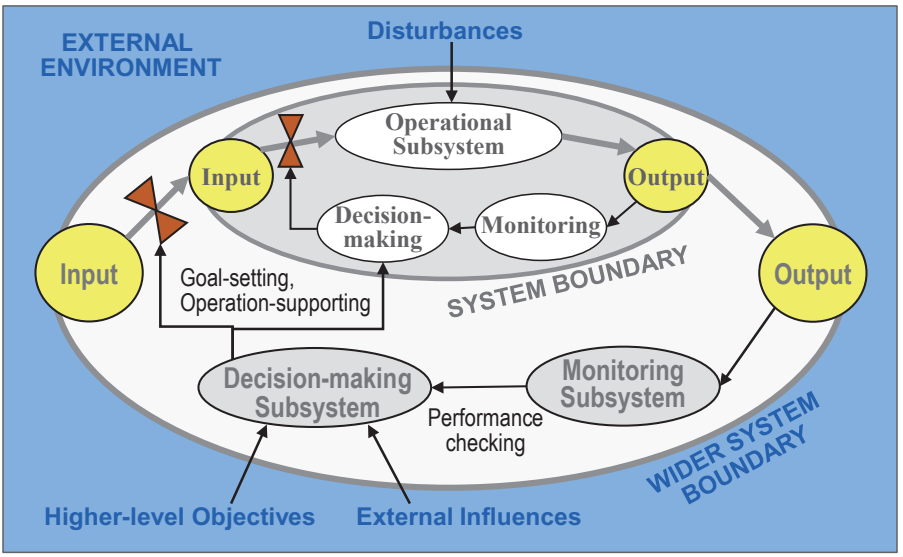


Fig. 1.2. Hierarchical structure of feedback control system.

The wider system influences the target system by setting goals, supporting operations, and checking performances. The target system is subject to disturbances from the external environment. In addition, the external environment provides the wider system with higher-level objectives and other external influences.

Exercise 1.1. Give an example of a feedback control system involving people and identify all the components of the system.

1.3 WHAT IS COMPUTER SIMULATION?

1.3.1 What Is Simulation?

A dictionary definition of *simulation* is "the technique of imitating the behavior of some situation by means of an analogous situation or apparatus to gain information more conveniently or to train (or entertain) personnel." "Some situation" in the definition corresponds to a source system, and an *apparatus* is a simulator. As elaborated in the definition, there are two types of simulation objectives: one is to gain information and the other is to train or entertain personnel. The former is often called an *analytic simulation* and the latter a *virtual environment simulation* [Fujimoto 2000].

The main purpose of an analytic simulation is the quantitative analysis of the source system based on "exact" data. Thus, the simulation should be executed in an as-fast-as-possible manner and be able to precisely reproduce the event sequence of the source system. An analytic simulation is often referred

Fig. 1.3. Examples of virtual environment simulation.

to as a *time-stamp simulation*. A virtual environment simulation is executed in a scaled real-time while creating virtual environments, and it is often referred to as a *time-delay simulation*. Shown in Fig. 1.3 are scenes from a war-game simulation and from a computer game.

An analytic simulation with human interaction is called a *constructive simulation*, and one without human interaction an *autonomous simulation.* If humans interact with the simulation as a participant, it is referred to as human-in-the-loop (HIL) simulation; if machines or software agents interact with the simulation, it is called a machine-in-the-loop (MIL) simulation. A virtual environment simulation without HIL/MIL is often called a *virtual simulation*; one with HIL only a *constructive simulation*; one with both HIL and MIL a *live simulation*. Figure 1.4 shows the classification of computer simulation.

1.3.2 Why Simulate?

Modeling and simulation is the central part of our thinking process. When the situation is too complex to be analyzed by mental simulation alone, we use a computer for simulating the situation. Let's consider the following situations:

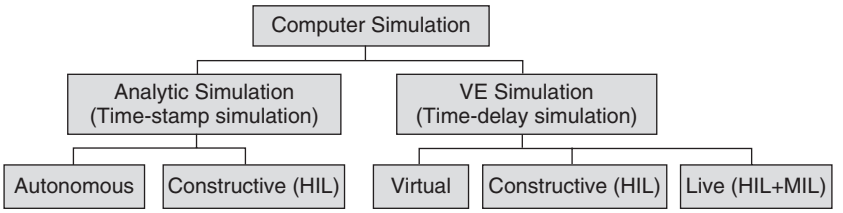


Fig. 1.4. Classification of computer simulation.

1. Finding optimal dispatching rules at a modern 300-mm semiconductor Fab
2. Evaluating alternative designs for hospitals, post offices, call centers, etc.
3. Designing the material handling system of a 3 billion dollar thin film transistor–liquid crystal display (TFT-LCD) Fab
4. Planning a wireless network for a telecommunication company
5. Evaluating high-tech weapons systems for a simulation-based acquisition
6. Designing or upgrading the urban traffic system of a big city
7. Evaluating anti-pollution policies to control pollutions in river systems
8. Evaluating risks in project schedules and financial derivatives

For the above real-life situations, simulation may be the only means to tackle the problems. In practice, simulation may be needed because experimenting with the real-life system is not feasible; your budget does not allow you to acquire an expensive prototype; a real test is risky; your customer wants it "yesterday"; your team wants to test several solutions and to compare them; you would like to keep a way to reproduce its performances later.

The simulation of a discrete-event system is called a *discrete-event simulation*, and that of a continuous system a *continuous simulation*. A class of computational schemes that rely on repeated random sampling to compute their results is referred to as *Monte Carlo simulation*. Among the above situations, Situations 1–6 are concerned with a discrete-event simulation. Situation 7 is concerned with a continuous simulation and Situation 8 with a Monte Carlo simulation.

1.3.3 Types of Computer Simulation

As depicted earlier in Fig. 1.1, the dynamic systems in the universe can be classified into five levels and three types. The three types of dynamic systems are: (1) discrete-event systems, (2) continuous systems, and (3) quantum systems. Thus, it is conceivable that there is one type of computer simulation for each system type. Discrete-event simulation and continuous simulation are widely performed on computers, but the direct simulation of quantum systems

on classical computers is very difficult because of the huge amount of memory required to store the explicit state of the system [Buluta and Nori 2009].

Continuous simulation is a numerical evaluation of a computer model of a physical dynamic system that continuously tracks system responses over time according to a set of equations typically involving differential equations. Let $\mathbf{Q}(t)$ and $\mathbf{X}(t)$ denote the system state and input trajectory vectors, respectively. Then, a linear continuous simulation is a numerical evaluation of the linear state transition function $d\mathbf{Q}(t)/dt = \mathbf{AQ}(t) + \mathbf{BX}(t)$, where $\mathbf{A}$ and $\mathbf{B}$ are coefficient matrices.

Discrete-event simulation is a computer evaluation of a discrete-event dynamic system model where the operation of the system is represented as a chronological sequence of events. In state-based modeling (see Chapter 9), the system dynamics is described by an internal state-transition function (δ_{int}: $Q \rightarrow Q$) and an external state-transition function (δ_{ext}: $Q \times X \rightarrow Q$), where Q is a set of system states and X is a set of input events. Thus, discrete-event simulation can be regarded as a computer evaluation of the internal and external transition functions.

Another type of popular computer simulation is the Monte Carlo simulation, which is not a dynamic system simulation. It is a class of computational algorithms that rely on repeated random sampling to compute the numerical integration of functions arising in engineering and science that are impossible to evaluate with direct analytical methods. In recent years, Monte Carlo simulation has also been used as a technique to understand the impact of risk and uncertainty in financial, project management, and other forecasting models.

1.4 WHAT IS DISCRETE-EVENT SIMULATION?

Figure 1.5 depicts a single server system consisting of a machine and a buffer in a factory. The dynamics of the system may be described as follows: (1) a job arrives at the system with an inter-arrival time of t_a, and the job is loaded on the machine if it is idle; otherwise, the job is put into the buffer; (2) the loaded job is processed for a service time of t_s and unloaded; (3) when a job is unloaded, the next job is loaded if the buffer is not empty. In Fig. 1.5, the state variables of the system are q and m, where q is the number of jobs in the buffer

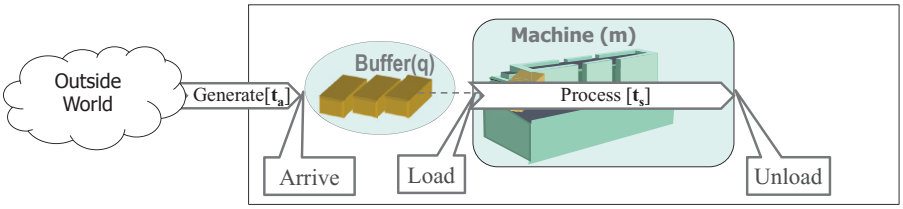


Fig. 1.5. A single server system model.

and m denotes the status (Idle or Busy) of the machine, and the events are Arrive, Load, and Unload.

1.4.1 Description of System Dynamics

Using the state variables and events, the system dynamics of the single server system may be described more rigorously as follows: (1) when an Arrive event occurs, q is increased by one, the next Arrive event is scheduled to occur after t_a time units, and a Load event is scheduled to occur immediately if m ≡ Idle(=0); (2) when a Load event occurs, q is decreased by one, m is set to Busy(=1), and an Unload event is scheduled to occur after t_s time units; (3) when an Unload event occurs, m is set to Idle and a Load event is scheduled to occur immediately if q > 0. The dynamics of the single server system may be described as a graph as given in Fig. 1.6, which is called an *event graph*.

1.4.2 Simulation Model Trajectory

An executable model of a system is called a *simulation model*, and the trajectory of the state variables of the model is called the *simulation model trajectory*. Let $\{a_k\}$ and $\{s_k\}$ denote the sequences of inter-arrival times (t_a) and service times (t_s), respectively. Then, the simulation model trajectory of the single server system would look like Fig. 1.7, where $\{t_i\}$ are event times, X(t) is input trajectory, and Q(t) = {q(t), m(t)} denotes the trajectory of the system

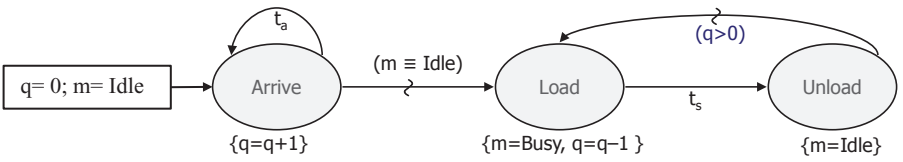


Fig. 1.6. Event graph describing the system dynamics of the single server system.

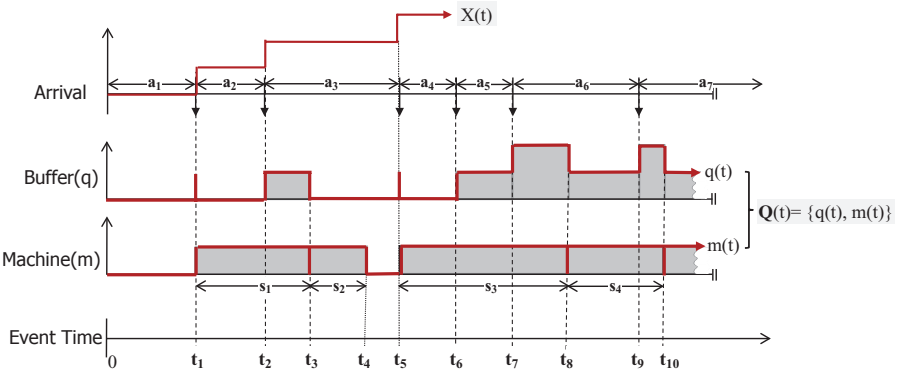


Fig. 1.7. Simulation model trajectory of the single server system.

state variables. The "time" here means a simulation time, which is a logical time used by the simulation model to represent physical time of the target system to be simulated.

At time t_1 (=a_1), a job J_1 arrives at an empty system and is loaded on the idle machine to be processed for a time period of s_1. In the meantime, another job J_2 arrives at time t_2 (=$a_1 + a_2$), which will be put into the buffer since the machine is busy. Thus, the buffer will have one job during the time period [t_2, t_3], which is denoted as a shaded bar in the buffer graph q(t) of Fig. 1.7. At t_3 (=$t_1 + s_1$), the first job J_1 is unloaded and the job J_2 in the buffer is loaded on the machine. At t_4 (=$t_3 + s_2$), J_2 is finished and unloaded, which will make the system empty again. Thus, the machine is busy during the time period [t_1, t_4]. At time t_5 (=$a_1 + a_2 + a_3$), another job J_3 arrives at the system and is loaded on the machine, and so on.

1.4.3 Collecting Statistics from the Model Trajectory

When simulating a service system, one may be interested in such items as (1) queue length, (2) waiting time distribution, (3) sojourn time, (4) server utilization, etc. In the case of the single server system, the following statistics can be collected from the model trajectory.

1. Queue length q(t) statistics during $t \in [t_0, t_{10}]$: AQL (average queue length)
   - AQL = $\{(t_3 - t_2) + (t_7 - t_6) + 2(t_8 - t_7) + (t_9 - t_8) + 2(t_{10} - t_9)\}/t_{10}$
2. Waiting time $\{W_j\}$ statistics for the first four jobs: AWT (average waiting time)
   - AWT = $\{W_1 + W_2 + W_3 + W_4\}/4 = \{0 + (t_3 - t_2) + 0 + (t_8 - t_6)\}/4 = (t_3 - t_2 + t_8 - t_6)/4$
3. Sojourn time $\{S_j\}$ statistics for the first four jobs: AST (average sojourn time)
   - AST = AWT + Average service time = AWT + $(s_1 + s_2 + s_3 + s_4)/4$
4. Server utilization during $t \in [t_0, t_{10}]$: U (utilization)
   - U = $\{(t_4 - t_1) + (t_{10} - t_5)\}/t_{10}$

1.5 WHAT IS CONTINUOUS SIMULATION?

As mentioned in Section 1.3.3, continuous simulation is a numerical evaluation of a computer model of a physical system that continuously tracks system responses over time, $\mathbf{Q}(t)$, according to a set of equations typically involving differential equations like $d\mathbf{Q}(t)/dt = f[\mathbf{Q}(t), \mathbf{X}(t)]$, where $\mathbf{X}(t)$ represents controls or input trajectory.

As an example, consider a Newtonian cooling model [Hannon and Ruth 2001]. Let $\sigma(t)$ be the cooling rate, then the temperature T(t) changes as

$dT(t)/dt = -\sigma(t)$. The cooling rate is expressed as $\sigma(t) = \kappa^*[T(t) - T_a]$, where κ is cooling constant and T_a is ambient temperature.

1.5.1 Manual Simulation of the Newtonian Cooling Model

The governing differential equation may be approximated by the following difference equation:

$$T(t + \Delta t) = T(t) - \sigma(t)^*\Delta t = T(t) - \kappa^*[T(t) - T_a]^*\Delta t, \text{ for } t = 0, \Delta t, 2\Delta t, 3\Delta t$$

Let's assume $T(0) = 37°C, T_a = 10°C, \kappa = 0.06$, and $\Delta t = 0.1$, then the temperature curve T(t) may be evaluated as follows:

$$T(0.1) = T(0) - 0.06^*[T(0) - 10]^*0.1 = 37 - 0.06^*(37 - 10)^*0.1 = 37 - 0.162$$
$$= 36.838$$
$$T(0.2) = T(0.1) - 0.06^*[T(0.1) - 10]^*0.1 = 36.838 - 0.06^*(36.838 - 10)^*0.1$$
$$= 36.677$$
…

1.5.2 Simulation of the Newtonian Cooling Model Using a Simulator

The cooling model may be simulated by using a commercial simulator such as STELLA®, as depicted in Fig. 1.8. In STELLA®, the level of state variable is regarded as a *stock* and the change in state variable as *flow*. In Fig. 1.8, TEMPERATURE is a stock and COOLING-RATE is a flow. COOLING CONSTANT and AMBIENT TEMPERATURE are parameters. These and other data are provided to the simulator via dialog boxes.

1.6 WHAT IS MONTE CARLO SIMULATION?

Monte Carlo simulation methods are a class of computational algorithms that rely on repeated random sampling to compute their results. They were developed for performing numerical integration of functions arising in engineering and science that were difficult to evaluate with direct analytical methods. In recent years, Monte Carlo simulation has also been used as a technique to understand the impact of risk and uncertainty in financial, project management, and other forecasting models.

1.6.1 Numerical Integration via Monte Carlo Simulation

As an example of numerical integration, consider the problem of finding the value of π via simulation. I am sure you have memorized the value of π as 3.14159. . .: but, for the moment, assume that you do not remember the value.

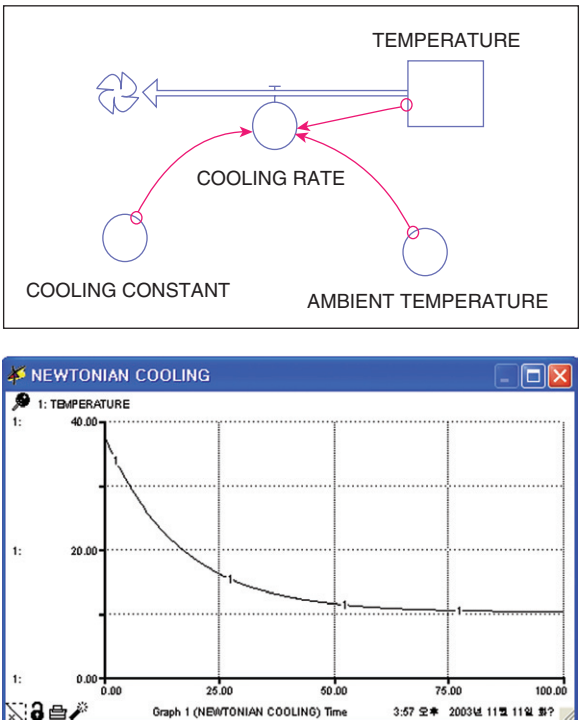


Fig. 1.8. STELLA® block-diagram modeling and output plot of the cooling system.

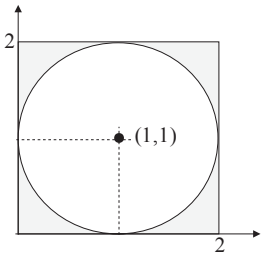


Fig. 1.9. A circle of unit radius to compute the value of π via Monte Carlo simulation.

In order to obtain the value of π via a Monte Carlo simulation, let's consider the circle shown in Fig. 1.9. It is a circle with a unit radius ($r = 1$) and its center is located at (1, 1). Uniform random variables with a range of [0, 2] are generated in pairs and are used as coordinates of points inside the square. Let n = total number of points generated (i.e., inside the square) and m = number of points inside the circle, and let A_c and A_s denote the areas of the circle and square, respectively. Then, the value of m/n approaches to the ratio A_c/A_s for

a large n. Since we know that $A_c = \pi r^2 = \pi$ and $A_s = 4$, we can compute π from the following relation: $m/n = A_c/A_s = \pi/4 \rightarrow \pi = 4\ m/n$ [Pidd 2004].

For the reader who may be curious about the execution of the simple Monte Carlo simulation, Java codes for (1) generating uniform random numbers and (2) computing the value of π are given below.

(1) Java code for generating uniform random number U ~ Uniform[0, 1]

```
double U = Math.random(); // Java function //
```

(2) Java code for finding the value of pi:

```
double m = 0, n = 0;
double max = 10000; // total number of sampling
while (n < max) {
   double u1 = Math.random();
   double u2 = Math.random();
   double x = 2.0 * u1;
   double y = 2.0 * u2;
   if ( ((x - 1) * (x - 1) + (y - 1) * (y - 1)) <= 1) m++;
   n++;
} // end of while
double phi = 4.0*m/n;
```

Exercise 1.2. Modify the above Monte Carlo simulation program (Java code) to compute the shaded area under the piece-wise linear function in Fig. 1.10.

1.6.2 Risk Analysis via Monte Carlo Simulation

Consider a project consisting of three tasks[1]: Task1, Task2, and Task3. Estimates of the time durations for the individual tasks are given in Table 1.1. We are interested in estimating the risk (or chance) of failing to meet a given project duration, say 15 months.

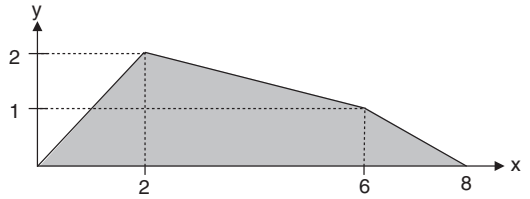


Fig. 1.10. Area under a piece-wise linear function.

[1]This example was taken from www.riskamp.com.

TABLE 1.1. Range Estimates for Individual Tasks

Task	Min (most optimistic)	Most likely	Max (most pessimistic)
Task1	4 months	5 months	7 months
Task2	3 months	4 months	6 months
Task3	4 months	5 months	6 months
Total	11 months	14 months	19 months

TABLE 1.2. Results of 500 Simulation Runs

Time duration (months)	12	13	14	15	16	17	18
# of on-time finishes	1	31	171	394	482	499	500
% of on-time finishes	0%	6%	34%	79%	96%	100%	100%

It is well accepted that the duration times are assumed to follow beta distribution (see Chapter 3). In the Monte Carlo simulation, values for the task duration times are randomly generated from respective beta distributions. The results of 500 simulation runs are summarized in Table 1.2, from which one may conclude that the risk of failing to finish the project within 15 months is about 20%. In recent years, Monte Carlo methods are quite popular in financial derivatives and option pricing evaluations.

1.7 WHAT ARE SIMULATION EXPERIMENTATION AND OPTIMIZATION?

The rules that govern the behavior of the system are called *laws*, while the rules under our control are called *policies*. When we experiment to determine the effects of changing the parameters of laws, we are doing a sensitivity analysis. When we experiment with changes in the control factors of policies, we are doing optimization [Schruben and Schruben 2001]. Both the parameters of laws and control factors of policies become handles of simulation experimentation. Both the optimization and sensitivity analysis may be performed in a simulation study. A simulation study should be carried out with

1. clear objectives of the study together with a set of performance measures;
2. output variables that can be mapped into the performance measures;
3. well-defined handles with which the simulation runs are to be controlled.

An experimental frame is a specification of the conditions under which the simulator is experimented with [Zeigler et al. 2000], and it is concerned with simulation optimization. As shown in Fig. 1.11, an experimental frame for simulation optimization consists of five steps: (1) an initial value of each

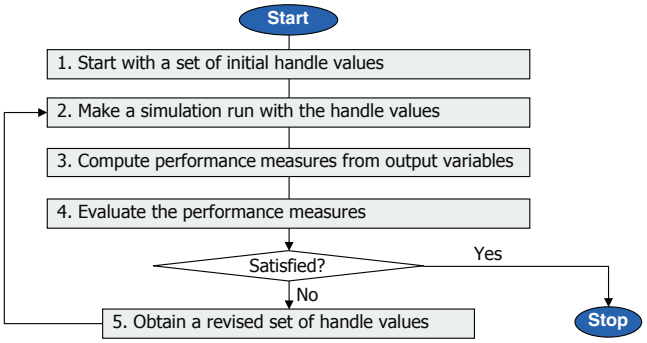


Fig. 1.11. Experimental frame for simulation optimization.

handle is generated; (2) a simulation run is made to compute values of the output variables; (3) performance measures are computed from the output variables; (4) the performance measures are evaluated to see if the results are acceptable; (5) if the results are not acceptable, go back to Step 2 with a revised set of handle values. Steps 3, 4, and 5 are often called *transducer*, *acceptor*, and *generator*, respectively.

1.8 REVIEW QUESTIONS

1.1. What are the common characteristics that lead to a conceptual definition of system?

1.2. Give a definition of a team based on the concept of system.

1.3. What is the difference between a source system and a target system?

1.4. What are the three key subsystems in a feedback control system?

1.5. What is an analytic simulation?

1.6. What is time-stamp simulation?

1.7. What would be the two popular areas where virtual environment simulation is used?

1.8. What is constructive simulation?

1.9. What is the main output from a continuous simulation?

1.10. In simulation, a rule under our control is called a policy. What is a law?

1.11. What is sensitivity analysis in simulation experimentation?

1.12. What is simulation optimization?

1.13. What is the role of the acceptor in an experimental frame?

CHAPTER 2

Basics of Discrete-Event System Modeling and Simulation

All models are wrong, some are useful.

—George E.P. Box

2.1 INTRODUCTION

A discrete-event system (DES) is a discrete-state and event-driven system in which the state changes depend entirely on the occurrence of discrete events over time. Examples of discrete-event systems include manufacturing systems, transportation systems such as urban traffic networks, service systems such as hospitals, and communication systems such as wireless networks, etc. This chapter aims to cover all the key subjects of and important issues in autonomous simulation of such discrete-event systems.

This chapter is organized as follows. Section 2.2 describes a step-by-step procedure for performing a discrete-event simulation. Section 2.3 deals with the fundamentals of DES modeling and introduces the concepts of reference model, modeling formalisms, and integrated framework of DES modeling. Illustrative examples are given in Section 2.4. Section 2.5 presents modeling and simulation (M&S) applications frameworks, and the last section addresses the issue of what to cover in a simulation class.

2.2 HOW IS A DISCRETE-EVENT SIMULATION CARRIED OUT?

Reproduced in Fig. 2.1 are the reference model and event graph of the single server system introduced in Chapter 1 (Section 1.4). There are two state variables (Q and M) and three event types (Arrive, Load, and Unload) in the system. Q is the job count, the number of jobs in the buffer; M denotes the

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.

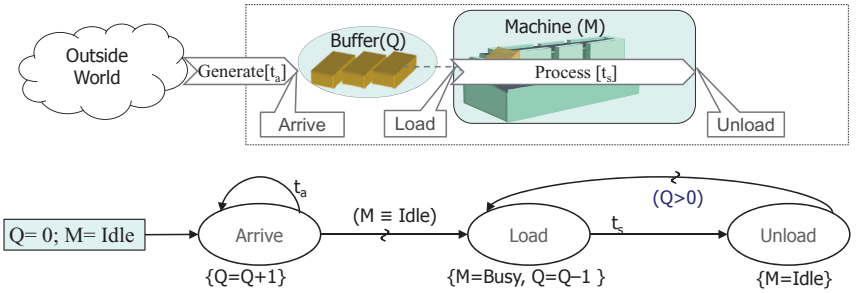
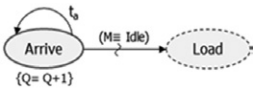
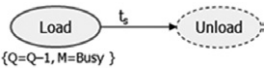


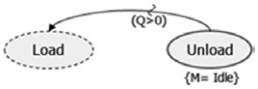
Fig. 2.1. Reference model and event graph of the single server system.

TABLE 2.1. Event Routines for the Event Nodes of the Event Graph in Fig. 2.1

Event Name	Event Node in the Event Graph	Event Routine

1. Arrive		Q = Q + 1; // increase the job-count by one
		If (M ≡ Idle), schedule an event <Load, Now>;
		Schedule an event <Arrive, Now + t_a>

2. Load		Q = Q − 1; //decrease the job-count by one
		M = Busy; // set the machine to busy.
		Schedule an event <Unload, Now + t_s>

3. Unload		M = Idle; // set the machine to idle
		If (Q > 0), schedule an event <Load, Now>

status (Idle or Busy) of the machine; and t_a and t_s denote interarrival times and service times, respectively.

2.2.1 Event Routines

A set of actions invoked by the occurrence of an originating event is called an *event routine*. Event routines systematically describe the dynamic behavior of the single server system. Event routines for the three event nodes in the event graph of Fig. 2.1 are listed in Table 2.1 and a brief description for each event routine is given below:

1. When an Arrive event occurs at time t, (1) the job count in the buffer is increased by one, Q = Q + 1; (2) a Load event is scheduled to occur immediately if the Machine is idle, M ≡ Idle; (3) a next Arrive event is scheduled to occur at time $t + t_a$.
2. When a Load event occurs at time t, (1) the job count in the buffer is decreased by one, Q = Q – 1; (2) the machine is set to busy (M = Busy); (3) an Unload event is scheduled to occur at time $t + t_s$.

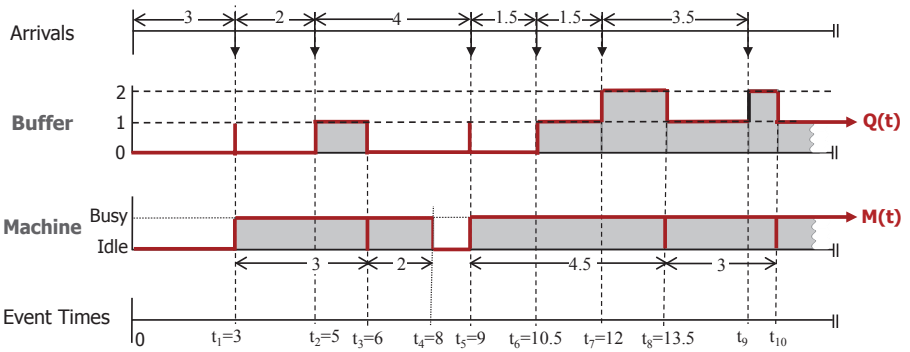


Fig. 2.2. Simulation model trajectory of the single server system.

3. When an Unload event occurs, (1) the machine is set to idle, M = Idle; (2) a Load event is scheduled to occur now if Q > 0.

2.2.2 Simulation Model Trajectory

An executable model of a system is called a *simulation model*. Shown in Fig. 2.2 is the simulation model trajectory of the single server system. Jobs have inter-arrival times $\{a_i\} = \{3, 2, 4, 1.5, 1.5, 3.5 \ldots\}$ and service times $\{s_i\} = \{3, 2, 4.5, 3 \cdots\}$, where a_i and s_i are the values for the i^{th} job J_i.

Initially, the system is empty and the machine is ready, thus Q = 0 and M = Idle. The event times $\{t_k\}$ are determined as follows:

1. At time $t_1 = 3$, J_1 arrives at the system and is loaded on the machine.
2. At time $t_2 = 5$, J_2 arrives at the system and is stored in the buffer.
3. At time $t_3 = 6$, J_1 is unloaded from the machine, and J_2 is loaded on the machine.
4. At time $t_4 = 8$, J_2 is unloaded from the machine.
5. . . .

In order to simulate the system dynamics, we need a mechanism for processing future events. A future event is an event that has been scheduled to occur in the future, and a future event that has the smallest, i.e., earliest, event time is called the *next event*. The event-processing mechanism consists of event cards and the *future event list* (FEL): the event-name and event time of a future event are recorded in an event card and it is stored in the FEL in an ascending order of event time.

2.2.3 Manual Simulation Execution

2.2.3.1 Initialization At the beginning, (1) the simulation time is set to zero and the state variables are initialized as Q = 0 and M = Idle, and (2) an

Arrive event is scheduled to occur at time $t_1 = 3$. The task of the "scheduling of an Arrive event to occur at time 3" is carried out by creating an event card with "Event-name = Arrive & Event-time = 3" and storing it in the FEL. Depicted in Fig. 2.3 are the state variables and FEL after initialization.

2.2.3.2 Next Event Processing The next event processing step starts with retrieving the next event from the FEL. First, the next event <Arrive, 3> is retrieved as depicted in Fig. 2.4(a) and the simulation time is set to 3. Then, the retrieved event is executed to update the state variables and FEL as

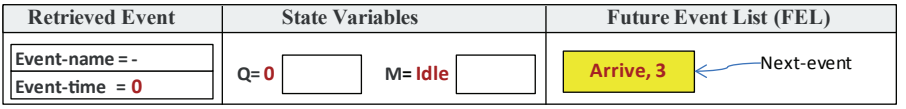


Fig. 2.3. State variables and FEL after initialization.

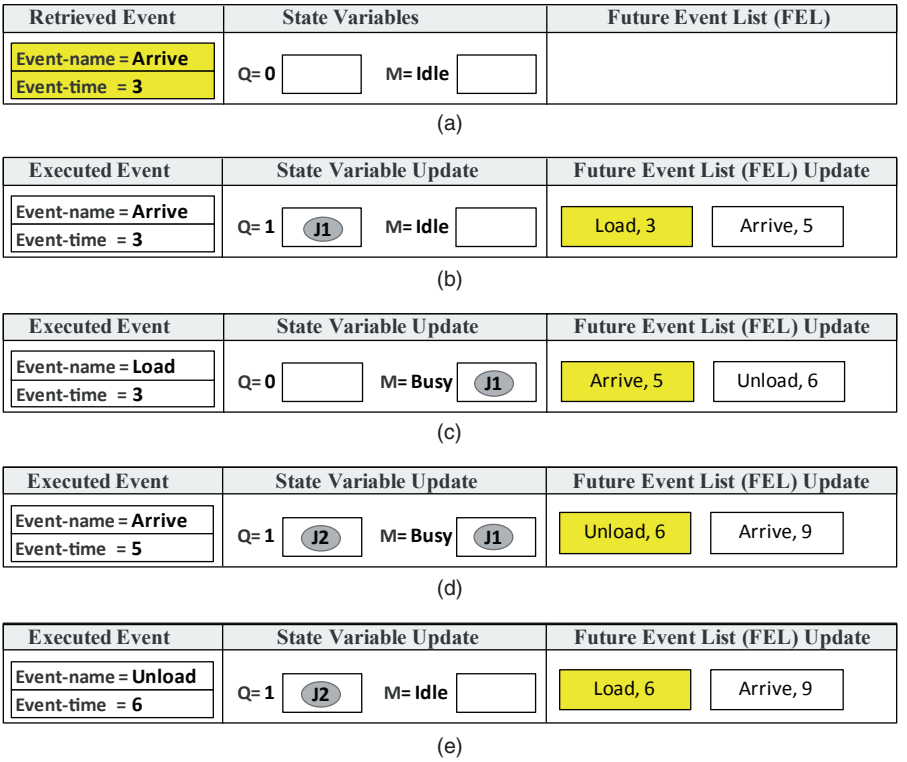


Fig. 2.4. (a) State variables and FEL after the retrieval of <Arrive, 3>; (b) state variables and FEL after the execution of <Arrive, 3>; (c) state variables and FEL after the execution of <Load, 3>; (d) state variables and FEL after the execution of <Arrive, 5>; (e) state variables and FEL after the execution of <Unload, 6>.

follows: (1) the new job J_1 is stored in the buffer and its job count is increased by one (Q = 0 + 1); (2) since the machine is Idle, a Load event is scheduled to occur now, i.e., at time 3, which is carried out by storing the event card <Load, 3> in the FEL; (3) an Arrive event is scheduled to occur at time 5 (= $3 + a_2 = 3 + 2$), which is carried out by storing the event card <Arrive, 5> in the FEL. The state variables and FEL after the execution of the next event <Arrive, 3> are shown in Fig. 2.4(b).

At the second step, the next event <Load, 3> is retrieved from the FEL of Fig. 2.4(b) and it is executed so that: (1) the job J_1 is loaded on the machine, the job count Q is decreased by 1, and the machine is set to busy (Q = 0 & M = Busy); (2) an Unload event is scheduled to occur at time 6 ($= 3 + s_1 = 3 + 3$), namely, <Unload, 6> is stored in the FEL. The state variables and FEL after the execution of the next event <Load, 3> are shown in Fig. 2.4(c), where <Arrive, 5> becomes the next event.

At the third step, <Arrive, 5> is retrieved from the FEL of Fig. 2.4(c) and is executed so that: (1) the new job J_2 is stored in the buffer and Q is increased by 1; (2) an Arrive event is scheduled to occur at time 9 ($= 5 + a_3 = 5 + 4$). The state variables and FEL after the execution of <Arrive, 5> are shown in Fig. 2.4(d), where <Unload, 6> become the next event.

At the fourth step, <Unload, 6> is retrieved and executed to set the machine to idle (M = Idle) and to schedule a Load event to occur immediately because Q > 0. The state variables and FEL after the execution of <Unload, 6> are as shown in Fig. 2.4(e), where the next event is <Load, 6>.

The trajectory of the state variables and FEL during the manual simulation up to the event time t_3 is given in Table 2.2. The first row contains the

TABLE 2.2. Trajectory of the State Variables and FEL during Manual Simulation

Event occurrence			State update				Future events list (FEL)		
Time	Name	Job	Q	M	Time delay	Scheduled event	Next event	Future event	Remark
0	Initialize	—	0	Idle	$a_1 = 3$	<Arrive, a_1>	<Arrive, 3>	—	Fig.2.3
$t_1 = 3$	Arrive	J_1	1	Idle	$a_2 = 2$	<Load, 3>, <Arrive, $3 + a_2$>	<Load, 3>	<Arrive, 5>	Fig.2.4(b)
$t_1 = 3$	Load	J_1	0	Busy	$s_1 = 3$	<Unload, $3 + s_1$>	<Arrive, 5>	<Unload, 6>	Fig.2.4(c)
$t_2 = 5$	Arrive	J_2	1	Busy	$a_3 = 4$	<Arrive, $5 + a_3$>	<Unload, 6>	<Arrive, 9>	Fig.2.4(d)
$t_3 = 6$	Unload	J_1	1	Idle	0	<Load, 6>	<Load, 6>	<Arrive, 9>	Fig.2.4(e)
$t_3 = 6$	Load	J_2	0	Busy	$s_2 = 2$	<Unload, $6 + s_2$>	<Unload, 8>	<Arrive, 9>	

information given in Fig. 2.3: event time is 0; Q = 0 and M = Idle; the scheduled event is <Arrive, a1> with $a_1 = 3$; the next event is <Arrive, 3>. The second row contains the information given in Fig. 2.4(b), and so on.

Exercise 2.1. Retrieve the next event <Load, 6> from the FEL of Fig. 2.4(e), execute it, and update the state variables and FEL. Continue the manual simulation a few more steps.

2.2.4 Flow Chart of Manual Simulation Procedure

The manual simulation procedure described in Section 2.2.3 can be described in the form of flow chart as depicted in Fig. 2.5. The simulation procedure consists of three steps: The initialization step, next event retrieval step, and event execution step:

1. At the initialization step, the state variables are initialized as Q = 0 & M = Idle, and the initial event <Arrive, 3> is stored in the FEL as depicted in Fig. 2.3.
2. At the next event retrieval step, the next event in the FEL is retrieved and the clock variable Now is set to the event time of the retrieved next event.
3. At the event execution step, the event routine (see Table 2.1) of the retrieved next event is executed.
4. Go back to step 2 (next event retrieval step) if the termination condition is not met.

In the flow chart, NOW is the current simulation time.

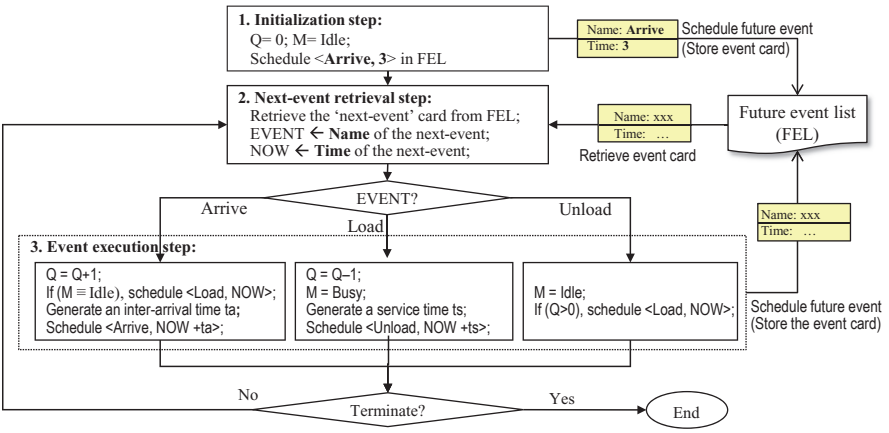


Fig. 2.5. Flow chart of the manual simulation procedure.

2.3 FRAMEWORK OF DISCRETE-EVENT SYSTEM MODELING

In this section, the framework of discrete-event system modeling will be analyzed in terms of (1) modeling components and reference model, (2) modeling formalism and formal model, and (3) formal modeling tools and model specification.

2.3.1 What Are Modeling Components and Reference Model?

2.3.1.1 Modeling Components From a practitioner's point of view, the aim of a discrete-event simulation is to learn about the behavior and performance potential of the system, and it is accomplished by the activities in which the resources and entities in the DES engage. From a system theoretic point of view, a DES can be viewed as a state machine consisting of a set of states, a set of events, a partial state transition function, and the initial and final states.

DES modeling components are a set of basis components of a DES used for describing the system dynamics. From the above observation, the modeling components of DES are resources, entities, activities, events, and states. Among them, resources and entities are referred to as *physical modeling components*, while activities, states, and events are *logical modeling components*. A resource such as a machine that is engaged in an activity is called an *active resource* and a resource such as a buffer used in storing entities is called a *passive resource*.

2.3.1.2 Reference Model An informal description of system dynamics using modeling components is referred to as a *reference model* of a DES. Let's consider the single server system shown in Fig. 2.1(a). It is an open system where entities are created from and disposed to the outside world. For an autonomous simulation, it is convenient to make it a closed system by treating the outside world as a job creator responsible for creating jobs. Then, the closed single server system may look like Fig. 2.6.

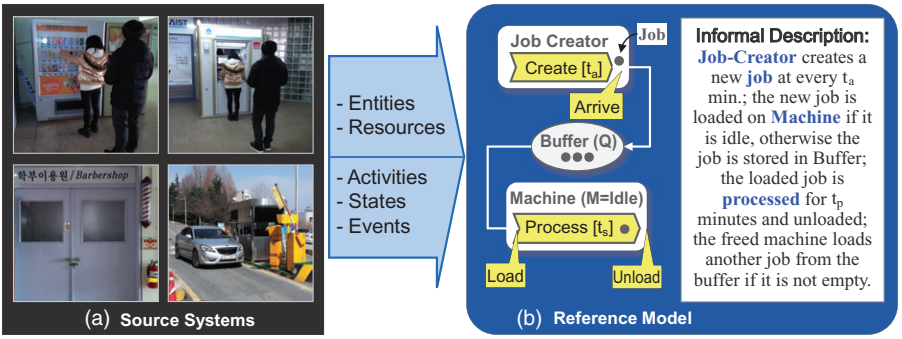


Fig. 2.6. (a) Source systems and (b) reference model of a single server system.

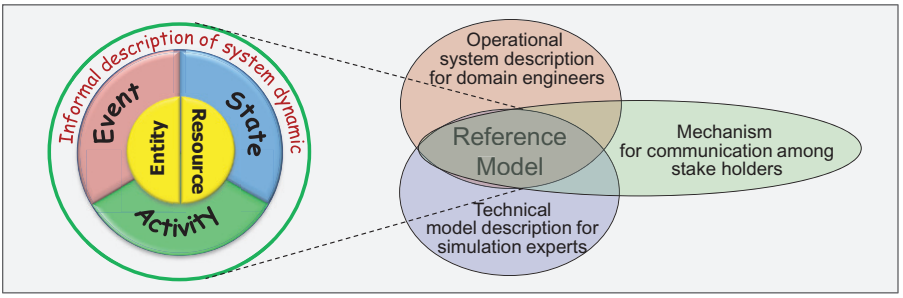


Fig. 2.7. The structure and role of a reference model.

The modeling components of the single server system of Fig. 2.6 are: (1) Entities are jobs; (2) Resources are Machine, Job-creator, and Buffer; (3) Activities are job creation and job processing; (4) State variables are the Machine status (M = Idle/Busy) and Buffer status (Q = number of jobs); (5) Events are Arrive, Load, and Unload. System dynamics is described using the modeling components as follows: Job-creator creates a new job at every t_a minutes; the new job is loaded on the Machine if it is idle, otherwise the job is stored in Buffer; the loaded job is processed by the Machine for a time period of t_s and then unloaded; the freed Machine loads another job from the Buffer if it is not empty.

The reference model structure (left-hand side of Fig. 2.7) has three layers: (1) At the core of the structure are physical modeling components—Entity and Resource—that constitute the static model of a DES; (2) at the next layer are logical modeling components—Activity, Event, and State—corresponding to the functional model of the DES; and (3) at the outer layer is an informal description of the system dynamics that corresponds to the control model of the DES. This view of the reference model structure is similar to the classical *object-oriented modeling* paradigm [Rumbaugh et al. 1991].

The roles of a reference model are depicted in the right-hand part of Fig, 2.7. System modeling needs a team effort involving domain engineers who have a working knowledge of the system, simulation experts who are responsible for building simulation models, and other stakeholders. A reference model should serve as an official system description for domain engineers and at the same time as a systematic model description for simulation experts. In addition, it should serve as a mechanism for communication among the stakeholders of the simulation project.

2.3.2 What Is a Discrete-Event System (DES) Modeling Formalism?

In this book, a *DES modeling formalism* is defined as a well-defined set of graphical conventions for specifying a DES. It has a formal syntax and can be executed by a simulation algorithm. There are three types of DES modeling

TABLE 2.3. Relationships Among Modeling Formalisms, Modeling Tools, and Worldviews

Modeling Formalism	Graphical Modeling Tools	Related World Views or System Specification
Activity-based formalism	ACD	Activity scanning world view
Event-based formalism	Event Graph	Event scheduling world view
State-based formalism	State Graph	DEVS (Discrete event system spec)

formalisms, one for each logical modeling component: activity-based, event-based, and state-based modeling formalisms. The modeling formalisms had been developed from different origins: (1) activity-based modeling formalism from the flow diagram method of Tocher [Hollocks 2008]; (2) event-based modeling formalism from event-scheduling languages such as SIMSCRIPT; (3) state-based modeling formalism from the state transition diagram method of finite state machine (FSM) modeling [Mealy 1955] and the DEVS (Discrete-EVent-system Specification) [Zeigler et al. 2000]. More details on FSM and DEVS can be found in Chapter 9 of this book.

Each of the modeling formalisms employs a graphical modeling tool: the activity cycle diagram (ACD) is used in the activity-based modeling formalism; the event graph in the event-based modeling formalism; and the state graph in the state-based modeling formalism. Among the three modeling formalisms, the first two are often referred to as *worldviews*: *activity-scanning worldview* and *event-scheduling worldview*. Summarized in Table 2.3 are the relationships among modeling formalisms, modeling tools, and worldviews.

2.3.2.1 Activity-Based Modeling Formalism and Activity Cycle Diagram

In the activity-based modeling formalism, the dynamics of system is described in terms of the activities of the active resources and entities in the system. It uses the activity cycle diagram (ACD) that was invented by Tocher [Tocher 1960]. Tocher's original work was followed by some further development work. An activity-based simulation language named ECSL® was developed [Clementson 1986] and a classical ACD was formally defined [Carrie 1988]. The classical ACD had some inherent limitations in handling complex systems [Hlupic and Paul 1994], and hierarchical ACD [Kienbaum and Paul 1994] and extended ACD [Martinez 2001] were proposed in order to enhance its modeling power. More recently, a formal specification of extended ACD was given and its generality was established [Kang and Choi 2011].

2.3.2.2 Process-Oriented Simulation Languages and Entity-Flow Diagram

When only the activity cycles of the entities in the system are considered, the activity-based modeling formalism becomes an entity-based modeling formalism or process interaction worldview [Carson 1993]. The sequence of activities

of an entity is often referred to as an *entity-flow*, and a diagram depicting the entity-flow is called an *entity-flow diagram* (EFD) [Harrell et al. 2012]. A flow of entities can be regarded as a time-ordered sequence of events that is often referred to as a *process*. Thus, entity-based modeling is often referred to as *process-oriented modeling* [Pritsker and Pegden 1979]. The process-oriented modeling approach (or process interaction worldview) is adopted in many of the modern simulation languages including Arena [Kelton et al. 2007] and ProModel [Harrell et al. 2012]. These simulation languages are often referred to as a *process-oriented simulation language*.

2.3.2.3 Event-Based Modeling Formalism and Event Graph In the event-based modeling formalism, a system is modeled by defining the changes that occur at event times and the system dynamics is described using an event graph. The event-based modeling concept was realized in the SIMSCRIPT language in 1960s [Kiviat et al. 1969] and an event graph is formally defined in 1980s [Schruben 1983]. In the event graph, events are represented as vertices and the relationships between events are represented as directed arcs. Event graph models are very compact. Yet, event graph models are capable of describing any system that can be implemented on a modern computer [Savage et al. 2005]. An event-based simulation language SIGMA® has been developed and widely used for modeling various types of DES [Schruben and Schruben 2006].

2.3.2.4 State-Based Modeling Formalism and State Graph In the state-based modeling formalism, the dynamics of a system is described in terms of the states of the resources in the system. The state-based modeling method is originated from the classical *finite state machine* (FSM) that was used for modeling the behavior of sequential circuits [Mealy 1955], where the concept of the state transition diagram was introduced. In 1970s, the classical FSM evolved to the classical DEVS in which internal transitions are also allowed [Zeigler 1976]. DEVS can be regarded as a special form of *timed automata* [Alur 1999] or timer-embedded FSM [Lee et al. 2010]. An in-depth treatment on the subject of the state-based modeling is provided in Part III (Chapter 9) of this book.

2.3.3 What Is a Formal Model and How Is It Specified?

A formalism-based modeling tool is referred to as a *formal modeling tool*. Among the DES modeling tools mentioned in the previous subsection (Section 2.3.2), ACD, event graph, and state graph are formal modeling tools. The EFD [Harrell et al. 2012] is a subset of the ACD, and it is not a formal modeling tool (we may call it a semi-formal modeling tool).

A DES model described with a formal modeling tool is referred to as a *formal model* if it provides a complete description of the system in a concise

and clear manner. A well-known algorithm is available for each of the formal models: the activity-scanning algorithm for ACD models [Carrie 1988]; the next event scheduling algorithm for event graph models [Schruben and Schruben 2006]; and the time-synchronization algorithm for state graph models [Lee et al. 2010]. In the following, we give an ACD model as a formal model of the single server system in Fig. 2.6. An event graph model was presented earlier in this chapter (Fig. 2.1), and is briefly reiterated here. The state graph model will be discussed in Part III (Chapter 9) of this book.

2.3.3.1 Event Graph Model of the Single Server System The event graph model of the single server system in Fig. 2.1 is a formal model that provides a complete and unambiguous description of the system dynamics of the DES from the event's point of view. An Arrive event increases the job count by 1 (Q++), always schedules the next Arrive event to occur after a time period of t_a, and triggers a Load event if the machine is idle (M ≡ Idle). The Load event sets the machine to busy (M = Busy), decreases the job count by 1 (Q––), and schedules an Unload event to occur after a time period of t_s. The Unload event resets the machine to idle (M = Idle) and triggers a Load event if the job count is positive (Q > 0). The event graph model can be executed with the next scheduling event algorithm (Chapter 4).

2.3.3.2 ACD Model of the Single Server System ACD consists of individual activity cycles. There is one activity cycle for each active resource, and one for each entity type. ACD is a bipartite directed graph having activity nodes denoted by rectangles and queue nodes by circles. Referring back to the single server system in Fig. 2.6, there are (1) two activities called Create and Process, (2) two active resources called Job-creator and Machine, and (3) one entity-type Job. Shown in Fig. 2.8 is an ACD model of the single server system. There is a resource activity cycle for each resource (Job-creator and Machine) indicated by dashed lines, and one entity activity cycle (Job cycle) indicated by solid lines. The ACD model is a complete and unambiguous description of the system dynamics of the single server system, and it provides a natural and intuitive view of the system dynamics.

The ACD model in Fig. 2.8 can be directly executed with a formal ACD simulator (see Chapter 6) or converted to a Petri-net model to be executed

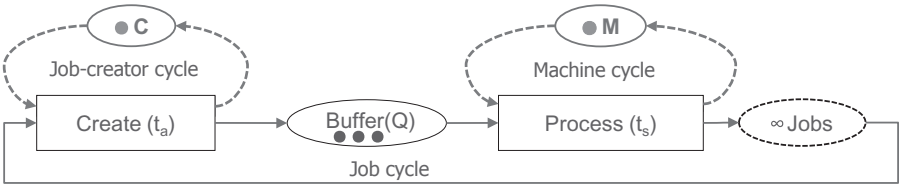


Fig. 2.8. ACD model of the single server system in Fig. 2.6.

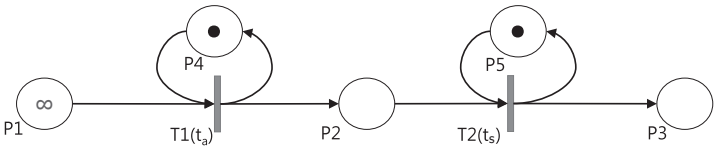


Fig. 2.9. Petri-net model corresponding to the ACD model of Fig. 2.8.

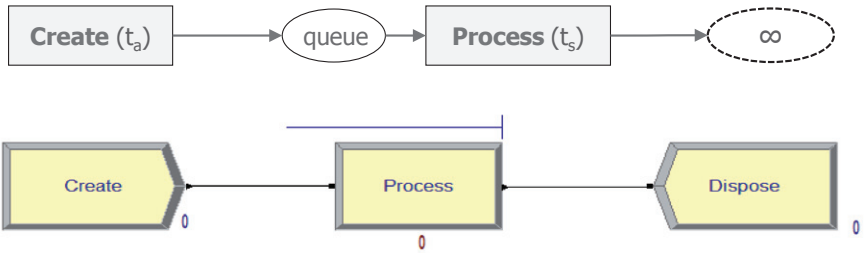


Fig. 2.10. Entity-flow diagram and Arena flowchart of the single server system.

with a Petri-net executor. A Petri-net model corresponding to this ACD model is given in Fig. 2.9. A Petri net is a bipartite directed graph consisting of places, transitions, and arcs, where an arc runs from a place to a transition or vice versa. A place is called an *input place* if an arc runs from the place to a transition. Likewise, it is called an *output place* if an arc from a transition runs into the place. Places in a Petri net may contain a number of tokens. Any distribution of tokens over the places will represent a state of the net called a *marking*. A transition of a Petri net may fire whenever there are sufficient tokens at its input places; when it fires, it consumes these tokens, and places the tokens at its output places. If time delay is allowed for a transition to fire, it is called a *timed transition*. A Petri net with timed transitions is called a *timed Petri net* (TPN), which is equivalent to an ACD. More details of Petri net are provided in Appendix of Chapter 10.

An ACD model may easily be converted to an EFD and executed with a process-oriented simulation language. An EFD model of the single server system is given in Fig. 2.10 along with its Arena flowchart. The EFD is the same as the entity activity cycle of the ACD in Fig. 2.8, and it does not provide a complete description of the system: Information about the resource-activity cycles of the ACD model has to be provided separately.

2.3.3.3 Specification of a Formal Model It is always possible to specify a formal model in an algebraic form. For example, a classical ACD model that is a bipartite directed graph consisting of a set of activity nodes and a set of queue nodes can be specified as follows [Kang and Choi 2011]:

$\boldsymbol{M}_{ACD} = <A, Q, I, O, \tau, \mu_0>$, where

$A = \{a_1, a_2 \cdots a_n\}$: finite set of activities,

$Q = \{q_1, q_2 \cdots q_m\}$: finite set of queues,

$I = \{i_a \subseteq Q \mid a \in A\}$: input function, a mapping from a set of queues to an activity,

$O = \{o_a \subseteq Q \mid a \in A\}$: output function, a mapping from an activity to a set of queues,

$\tau = \{t_a \in \mathrm{R}_0^+ \mid a \in A\}$: time delay function,

$\mu_0 = \{\mu_q \in \mathrm{N}_0^+ \mid q \in Q\}$: finite set of initial token values for each queue.

As an example, the ACD model given in Fig. 2.8 of the single server system may be specified as follows:

$\boldsymbol{M}_{ACD\ (\text{Fig. 2.8})} = <A, Q, I, O, \tau, \mu_0>$, where

A = {a_1: CREATE, a_2: PROCESS}

Q = {q_1: Jobs, q_2: C, q_3: Buffer, q_4: M}

$I(a_1) = \{q_1, q_2\}$; $I(a_2) = \{q_3, q_4\}$

$O(a_1) = \{q_2, q_3\}$, $O(a_2) = \{q_4, q_1\}$

$\tau(a_1) = \mathrm{t_a}$; $\tau(a_2) = \mathrm{t_s}$

$\mu_0(q_1) = \infty$; $\mu_0(q_2) = 1$; $\mu_0(q_3) = 3$; $\mu_0(q_4) = 1$

A formal model can also be specified in a tabular form. For practical purposes, specifying a formal model as an algebraic form is both tedious and hard to read, thus a tabular structure may be preferred for describing a formal model. As will be seen in Section 2.4, there is a well-defined tabular representation scheme for each of the three types of formal model: activity transition table for the ACD model; event transition table for the event graph model; object interaction table and state transition table for the state graph model.

2.3.4 Integrated Framework of DES Modeling

The current state-of-the-art in DES modeling is a result of major breakthroughs in the four areas mentioned in Section 2.3.2: (1) the activity-based modeling formalism; (2) the advent of process-oriented simulation languages; (3) the event-based modeling formalism; (4) the state-based modeling formalism. The modeling formalisms have been developed largely independently, and each of them was treated more or less as a separate framework of DES modeling. Among the four, the state-based modeling formalism is often referred to as a system specification and the remaining three as worldviews.

Based on the observations in Sections 2.3.1 through 2.3.3, this section presents an integrated framework of DES modeling. The proposed integrated framework consists of an integrated structure and an integrated procedure for DES modeling.

2.3.4.1 An Integrated Structure of the DES Model It was shown in Section 2.3.1 that all discrete-event systems can be described by a reference model consisting of physical and logical modeling components. Physical modeling components are resources and entities in the system, and logical modeling components are activities, events, and states. As there are three kinds of formal modeling tools (i.e., ACD, event graph, and state graph), one corresponding to each type of logical modeling component (i.e., activity, event, and state) three types of formal models can be constructed for a given reference model: ACD model, event graph model, and state graph model.

Figure 2.11(a) shows an integrated structure of a DES model consisting of three layers: (1) At the core of the integrated structure is the static model layer consisting of the physical modeling components—Entity and Resource; (2) at the next layer is the functional model layer constituted with the logical modeling components—Activity, Event, and State; (3) at the outer layer is the dynamic model layer defined by the three types of primary formal models—ACD model, EG (event graph) model, and SG (state graph) model. Also indicated in the integrated structure is that the EFD (entity-flow diagram) or Petri-net model can be obtained from an ACD model. Figure 2.11(b) depicts the conversion relations among the formal models. As mentioned in Section 2.3.3, an ACD model can be automatically converted into an EFD model or a Petri-net model. As will be seen in Chapter 10, Section 10.6, an ACD model can also be converted into an EG model or SG model as well.

2.3.4.2 Integrated Procedure for DES Modeling There exist various means to execute a simulation model of a given source system. Examples include: (1) entity-based simulation languages such as Arena [Kelton et al. 2007] and ProModel [Harrell et al. 2012], (2) Petri-net executors [Camurri and Coglio 1997], (3) ACD tool kits [Kang and Choi 2011] and (4) event-based simulation languages such as SIGMA [Schruben and Schruben 2006].

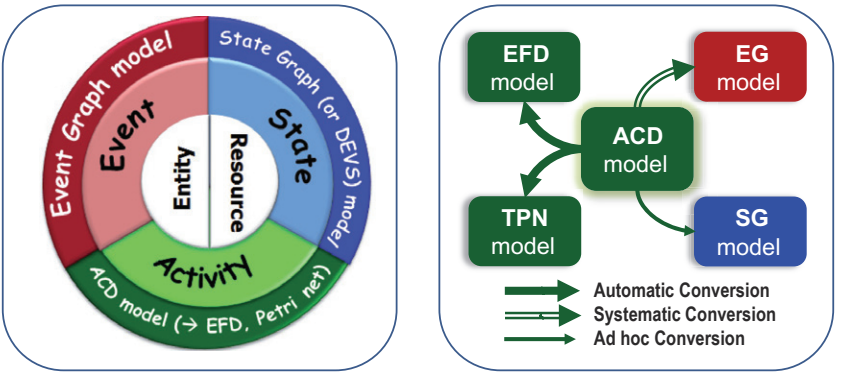


Fig. 2.11. (a) Integrated structure and (b) conversion relations of DES models.

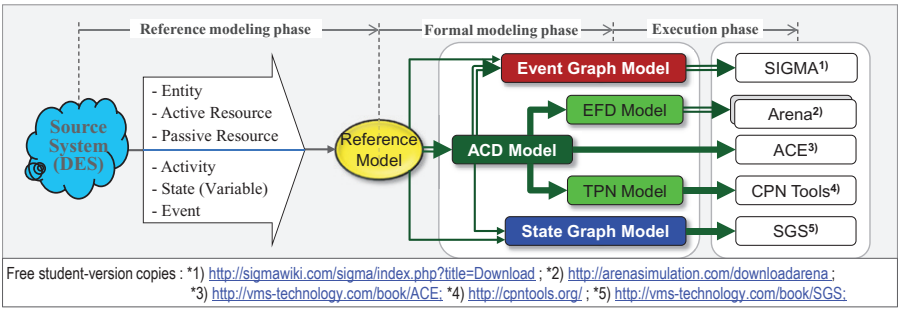


Fig. 2.12. An integrated procedure for DES modeling.

Regardless of the means used in executing a simulation model, construction of an executable simulation model from a given source system should follow a well-defined procedure. Figure 2.12 shows an integrated procedure for DES modeling consisting of three phases: the reference modeling phase, where a reference model of the DES is constructed from the source system; the formal modeling phase, where a formal model is obtained from the reference model; and the model execution phase, where the formal model is executed using a simulator. There exists at least one simulator in each of the five modeling formalisms. Moreover, free student-version copies are available as listed at the bottom of Fig. 2.12.

The *reference modeling phase* consists of four steps: (1) identify the physical modeling components—Entity, Active Resource, and Passive Resource; (2) define the logical modeling components—Activity, Event, and State; (3) describe the system dynamics in terms of the identified modeling components; (4) qualify the reference model against the source system. The referenced model qualification is a rigorous, systematic analysis of model relevance and consistency with the source system to ensure the reference model is fit for purpose.

The *formal modeling phase* consists of (1) selecting a modeling tool that is most compatible with the reference model; (2) building a formal model; (3) converting the formal model into another model if necessary; and (4) validating the formal model against the reference model. The formal model validation is a systematic analysis of model fidelity and sensitivity against the reference model to ensure the formal model is an accurate representation of the reference model. Selecting a best modeling tool for a given reference model is an open problem that deserves a further in-depth research. Model conversion among the formal models is another research area that deserves further investigation.

The *model execution phase* consists of (1) selecting a simulator suitable for executing the formal model; (2) preparing input data for making simulation runs; and (3) verifying the correctness of the simulation program against the formal model. The simulator verification is the process of making sure that the

written computer program corresponds precisely to the formal model [Fishwick 1995].

2.3.4.3 Criteria for Evaluating Models and for Selecting Modeling Tools

One may look for a model that is correct and perfect, but is it possible to have a *correct model*? George E.P. Box, who was an English chemist and statistician, is credited with the quote: "All models are wrong, some are useful." This quote may be an answer to the above question: We should look for a *good model*, not the correct one, and a good model is a useful one that serves its purposes. Is it possible to have a *perfect model*? Perhaps an answer to this question may be found from Antoine de Saint Exupery, a French writer and aviator, who is credited with the quote: "Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away." A perfect model is the one that contains just enough elements to make it useful.

A DES simulation model may be specified by using one of the formal modeling tools or programmed by employing a simulation package. Formal modeling tools are event graph, ACD, and state graph. Popular simulation packages include Arena®, AutoMod®, EXTEND®, SIGMA®, etc. Guidelines for selecting a formal modeling tool or a simulation language that is suitable for a given reference model are:

1. Choose one that has clear semantics and exact syntax.
   (a) Choose ACD if the system is described in terms of the activities of resources
   (b) Choose state graph if the system is described in terms of interacting objects
   (c) Choose event graph if the system is described in terms of interrelated events
   (d) Choose Arena, etc., if the system is described in terms of flows of entities
2. Choose one that has high modeling power.
3. Choose one that supports easy model building, communication, and validation.
4. Choose one that is amenable to easy implementation.

2.4 ILLUSTRATIVE EXAMPLES OF DES MODELING AND SIMULATION

This section aims to help the readers become acquainted with the modeling formalisms and show how to build and simulate formal models of a simple DES. For this purpose, a two-stage tandem line shown in Fig. 2.13 will be used as an example. The two-stage tandem line is obtained by concatenating two single server systems. The modeling components of the two-stage tandem line are as follows:

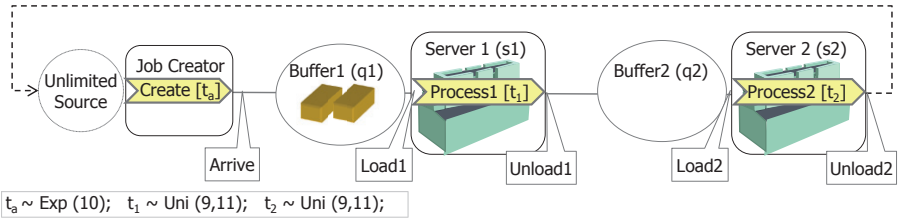


Fig. 2.13. Reference model of a two-stage tandem line.

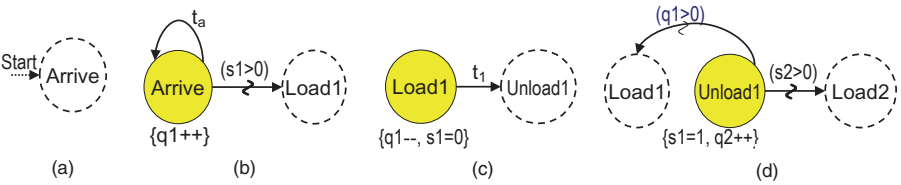


Fig. 2.14. Event routine graphs for (a) Start, (b) Arrive, (c) Load1, and (d) Unload1 events.

1. Entity: Job
2. Resources: Creator; Server1; Server2; Buffer1; Buffer2//unlimited-capacity buffer
3. Activity (time delay): Create (t_a); Process1 (t_1); Process2 (t_2)
4. State variables: number of idle servers (s1, s2); number of jobs in Buffer (q1, q2)
5. Event: Arrive {q1++}; Load1 {q1—, s1 = 0}; Unload1 {s1 = 1, q2++}; Load2 {q2—, s2 = 0}; Unload2 {s2 = 1}

2.4.1 How to Build and Simulate an Event Graph Model of a DES

An event graph model is a network of event nodes describing the events that take place in the system and the relationships among these events [Schruben 1983]. The event types found in the system of Fig. 2.13 are Arrive, Load1, Unload1, Load2, and Unload2. At the beginning, an Arrive event is scheduled, and then a sequence of events follows the flow of entities in the system.

2.4.1.1 Event Graph Modeling Shown in Fig. 2.14(a) is the initial state of the event graph where an Arrive event is scheduled with s1 = s2 = Idle (=1). When an Arrive event occurs, the job count in Buffer1 is increased by 1 (q1++), another Arrive event is scheduled to occur after t_a minutes, and if Server1 is Idle (s1 > 0), a Load1 event is scheduled to occur immediately. The situation is depicted in Fig. 2.14(b), which we call an *event-routine graph*.

Actions taken at an occurrence of a Load1 event are: decrease the job count in Buffer1 (q1—); set Server1 to Busy (s1 = 0); schedule an Unload1 event to occur after t_1 minutes. They are depicted in Fig. 2.14(c) as an event-routine

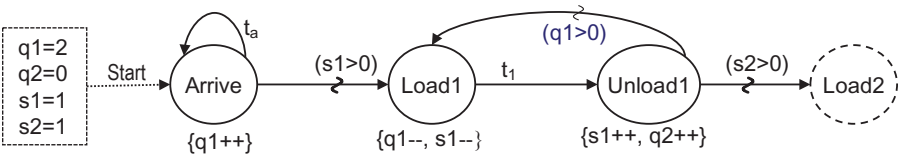


Fig. 2.15. Partial event graph model of the two-stage tandem line.

TABLE 2.4. Event Transition Table for the Event Graph of Two-Stage Tandem Line

Event Name	State Change	Edge Condition	Time Delay	Next Event
Start	q1 = 2, q2 = 0; s1 = s2 = 1;	Always	0	Arrive
Arrive	q1++;	Always	$t_a \sim$ Exp(10)	Arrive
		s1 > 0	0	Load1
Load1	q1—; s1—;	Always	$t_1 \sim$ Uni(9,11)	Unload1
Unload1	s1++; q2++;	q1 > 0	0	Load1
		s2 > 0	0	Load2

graph. The occurrence of an Unload1 event will result in the following actions as depicted in Fig. 2.14(d): Server1 is set to idle (s1 = 1); the job count in Buffer2 is increased (q2++); a Load1 event is scheduled to occur now if q1 > 0; a Load2 event is scheduled to occur now if s2 > 0. As mentioned earlier (Section 2.2.1), a set of actions taken at an occurrence of an event is called an event routine.

If we assemble the individual event-routine graphs of Fig. 2.14 into a "composite" event graph, we obtain the event graph shown in Fig. 2.15. The information specified in the event graph model can be summarized in a table called an *event transition table*. Table 2.4 is an event transition table for the event graph model shown in Fig. 2.15.

Exercise 2.2. Complete the event graph model of the two-stage tandem line by adding the event-routine graphs for Load2 and Unload2 events. Construct an event transition table for the full event graph of the two-stage tandem line.

2.4.1.2 Simulation of the Two-Stage Tandem Line Model with SIGMA®

Figure 2.16 shows a SIGMA event graph model of the two-stage tandem line that was generated by clicking and dragging the mouse. As will be shown in Chapter 4, it is very straightforward to build an event graph using SIGMA®.

If you double click the event vertex Arrive, a vertex dialog box like the one in Fig. 2.17(a) will show up where you provide the state change information (e.g., Q1 = Q1 + 1). If you double click the self-loop edge of the Arrive event,

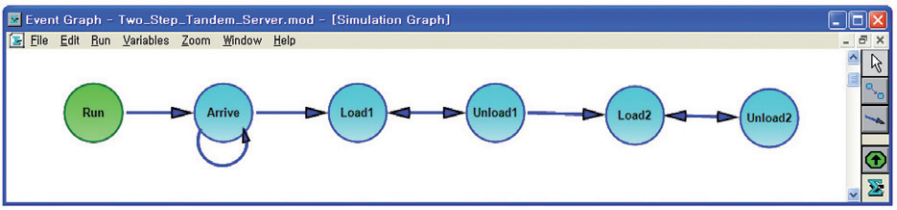
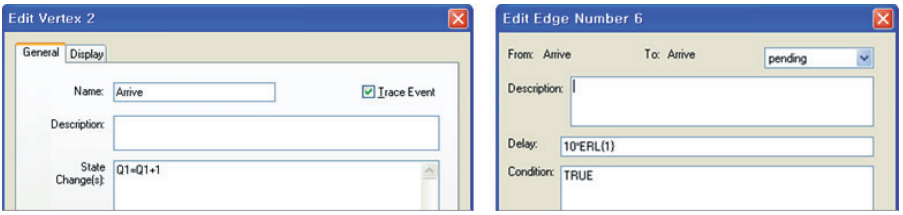


Fig. 2.16. SIGMA® event graph model.



(a) (b)

Fig. 2.17. SIGMA (a) Vertex dialog box and (b) Edge dialog box.

an edge dialog box of Fig. 2.17(b) will show up where you specify the inter-arrival time distribution (e.g., 10*ERL{1}).

2.4.2 How to Build and Simulate an ACD Model of a DES

2.4.2.1 ACD Modeling An activity cycle diagram (ACD) model consists of activity cycles, one for each entity and one for each active resource in the system. An activity cycle is a closed and alternating sequence of an active state (activity) denoted by a rectangle and a passive state (queue) denoted by a circle [Carrie 1998].

Referring back to Fig. 2.13, the two-stage tandem line system has three active resources: Creator, Server1, and Server2. On a close examination, all the resources are in an idle state. Buffer1 has two jobs, and Buffer2 has no jobs. Depicted in Fig. 2.18(a) are the activity cycles of the three resources in the system. A job created by Creator goes into Buffer1 from which it is loaded on Server1 for processing. Then it moves to Buffer2 to be processed by Server2. This "job flow" is modeled as an activity cycle of the job as shown in Fig. 2.18(b).

By combining the activity cycles in Figs. 2.18(a) and 2.18(b) together, an ACD model of the two-stage tandem line is obtained as shown in Fig. 2.19. Also shown in the ACD are the distribution functions, Exp(10) and Uni(9,11), for the inter-arrival times and processing times.

The information specified in an ACD model can be summarized in a table called an *activity transition table*. Table 2.5 is an activity transition table for the ACD model shown in Fig. 2.19. For each activity, (1) its firing condition is

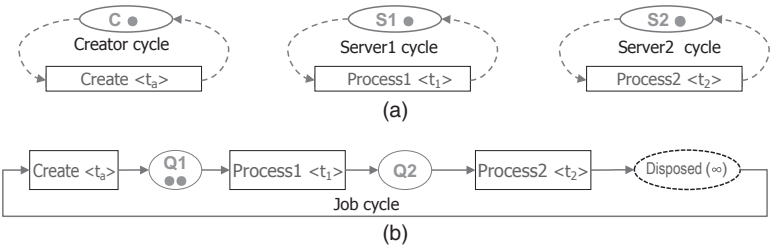


Fig. 2.18. (a) Activity cycles of the resources in the two-stage tandem line system; (b) activity cycle of the job in the two-stage tandem line system.

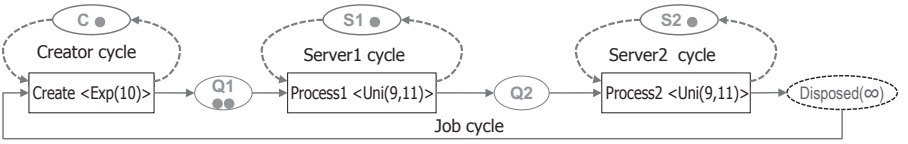


Fig. 2.19. ACD model of the two-stage tandem line.

TABLE 2.5. Activity Transition Table for the Event Graph of Two-Stage Tandem Line

		At Begin		BTO Event		At End	
No	Activity Name	Condition	Action	Time	Name	Action	Influenced Activities
1	Create	(C ≡ 1)	C—	Exp(10)	Arrived	Q1++, C++	Create, Process1
2	Process1	(Q1 > 0) & (S1 > 0)	Q1—, S1—	Uni(9,11)	Processed1	S1++, Q2++	Process1, Process2
3	Process2	(Q2 > 0) & (S2 > 0)	Q2—, S2—	Uni(9,11)	Processed2	S2++	Process2
Initial State		C = 1, S1 = 1, S2 = 1, Q1 = 2, Q2 = 0					

specified in the At-begin Condition field and the resulting state changes are given in the At-begin Action field; (2) the time-delay and name of the bound-to-occur event (BTO event) are specified in the BTO-event Time and BTO-event Name fields, respectively; (3) the state changes at the BTO event are specified in the At-end Action field. The ACD model may be simulated by using an ACD executor (see Chapter 6) or converted into a process-interaction simulation program like Arena (see Chapter 7).

2.4.2.2 Simulating the ACD Model with Arena® It is fairly straightforward to prepare Arena simulation inputs from an ACD model. In order to perform simulation with the Arena software, (1) all the resources and entities are declared first, (2) a flowchart model denoting the entity flow is generated, and (3) the attributes of each block (or module) are entered in its dialog box.

Resource - Basic Process

	Name	Type	Capacity
1	S1	Fixed Capacity	1
2	S2	Fixed Capacity	1

Entity - Basic Process

	Entity Type	Initial Picture	Holding Cost / Hour
1	Job	Picture. Box	0.0

Double-click here to add a new row.

Fig. 2.20. Declarations of Resource (S1 and S2) and Entity (Job).

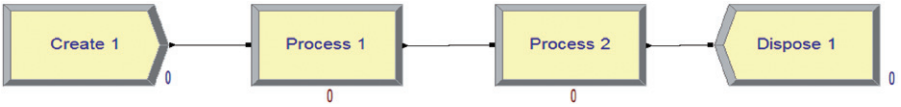


Fig. 2.21. Arena® flowchart model of the two-stage tandem line.

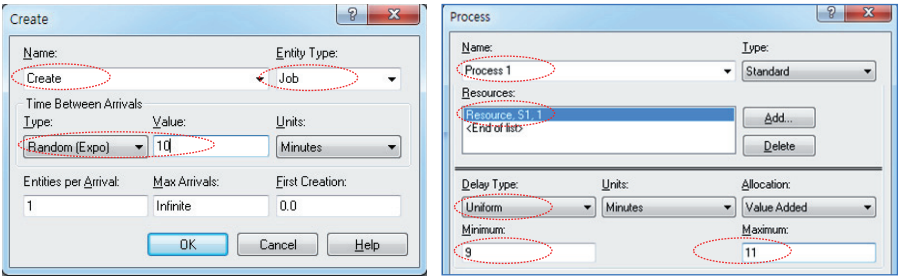


Fig. 2.22. Arena dialog boxes for defining Create module and Process module.

Shown in Fig. 2.20 are a resource data model and an entity data model in the Basic Process Template of Arena, where the resources (S1 & S2) and entity (Job) of the two-stage tandem line are declared.

Shown in Fig. 2.21 is an Arena flowchart model of the two-stage tandem line. The Create activity in the ACD model of Fig. 2.19 is mapped to the Create1 block in Arena flowchart model of Fig. 2.21, the Disposed queue is mapped to Dispose1 block, and the activity nodes are mapped to Process blocks.

Shown in Fig. 2.22 are dialog boxes for inputting data needed to define the Create block and Process1 block. In the Create block dialog box, the inter-arrival time distribution is defined as Type = Random (Expo) and Value = 10. In the Process block dialog box, the service time distribution is specified as Delay Type = Uniform, Minimum = 9, and Maximum = 11.

2.4.3 How to Build and Simulate a State Graph Model of a DES

The first step in state-based modeling is to identify objects in the system and construct an object interaction diagram describing interactions among the objects in the system. There are five objects (Creator, Buffer1, Server1, Buffer2, and Server2) in the two-stage tandem line, and they interact with each other

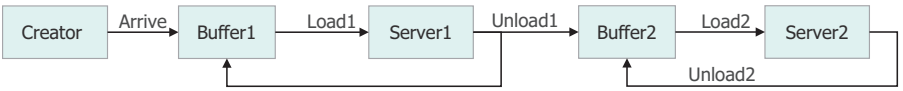


Fig. 2.23. Object interaction diagram of the two-stage tandem line.

via event messages: Creator sends an Arrive message to Buffer1; Buffer1 sends a Load1 message to Server1, which sends back an Unload1 message to Buffer1; Server1 sends an Unload1 message to Buffer2; Buffer2 sends a Load2 message to Server2, which sends back an Unload2 message to Buffer2. By combining all the individual interactions, the object interaction diagram of the two-stage tandem line system is obtained as shown in Fig. 2.23.

The second step in state-based modeling is to draw a state transition diagram for each object in the system. The object interaction diagram and the individual state-transition diagrams form a state graph of the system. In order to execute the state graph model of the system, the information described in a state transition diagram is specified in a state transition table. An extensive treatment on the subject is provided in Chapter 9 of this book.

2.5 APPLICATION FRAMEWORKS FOR DISCRETE-EVENT SYSTEM MODELING AND SIMULATION

2.5.1 How Is the M&S Life Cycle Managed?

Shown in Fig. 2.24 are various activities and entities involved in a real-life M&S project, which we call the *M&S life-cycle management* framework. The life-cycle management framework consists of four phases: problem definition phase, modeling phase, simulation phase, and implementation (or application) phase:

1. Phase 1 is the problem definition phase consisting of (1) diagnosis and analysis of a real-life situation from which a source system is identified and the objectives of the study are defined, (2) defining experimental frames, and (3) collecting data.
2. Phase 2 is the modeling phase consisting of (1) the descriptive modeling step for building a reference model and (2) the formal modeling step for building a formal model from the reference model by employing a modeling formalism. Also carried out in this phase are reference model qualification and formal model validation. A model qualification is a rigorous, systematic analysis and evaluation of the reference model for its relevance and consistency with observed behavioral data to ensure that the models are fit for purpose.
3. Phase 3 is the simulation phase where (1) a simulator is implemented from the formal model using a simulation software tool if necessary, (2) a series of experimentation is performed with the simulator according to

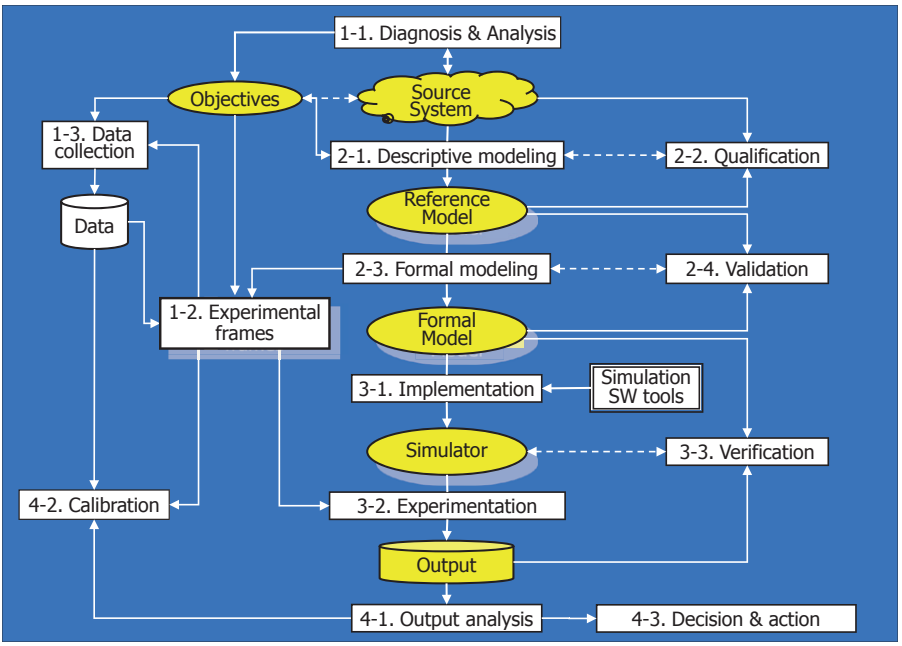


Fig. 2.24. M&S life-cycle management framework.

the specifications of the experimental frame, and (3) the simulator is verified against the formal model.

4. Phase 4 is the implementation phase, consisting of (1) output analysis, (2) simulator calibration, and (3) making decisions and taking actions. A simulator calibration is a systematic procedure for fine-tuning the simulator by adjusting model parameters so that the simulation outputs conform to actual trajectories of the target system.

Reference model qualification, formal model validation, and simulator verification and calibration are the key feedback functions in M&S life-cycle management.

2.5.2 Framework for Factory Life-Cycle Support

More than a 50 years ago, K.D. Tocher tried to solve the congestion control problem at United Steels in the U.K. [Tocher 1960]. He argued that "in more complex plants, in which there is a multiplicity of possible routes for the steel through the plant, it is possible to minimize congestion and maximize the rate of flow by a (simulation-based) scheduling procedure." It is truly remarkable that Tocher, who invented the ACD, tried to use simulation as an operation management tool in the 1960s. Congestion control is also a key issue in operation management for a modern electronics Fab (i.e., fabrication plant) such as a semiconductor Fab or a flat panel display (FPD) Fab.

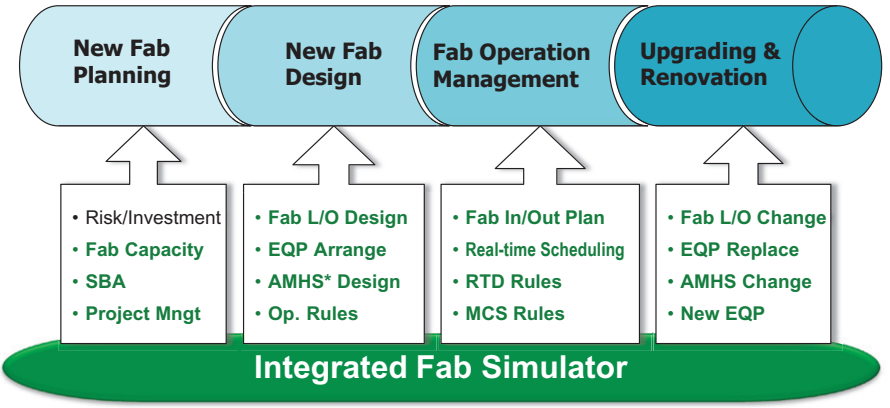


Fig. 2.25. Framework of simulation-based Fab life-cycle support. SBA, simulation-based acquisition; AMHS, automated material handling system; RTD, real-time dispatcher; MCS, material control system for AMHS.

The authors have been working with FPD makers to develop simulation-based Fab scheduling systems [Park et al. 2008] and a Fab simulator for an integrated simulation of production and AMHS (automated material handling system) [Song et al. 2011]. Shown in Fig. 2.25 are the four phases of Fab life cycle together with action items for Fab life-cycle management. The four phases are (1) planning phase for a new Fab, (2) the new Fab design phase, (3) Fab operation management phase, and (4) Fab upgrading and renovation phase. An integrated Fab simulator may be used as a decision-support tool covering the entire Fab life cycle. Issues in developing such an integrated Fab simulator will be addressed in Chapter 11.

2.6 WHAT TO COVER IN A SIMULATION CLASS

There exists a large volume of knowledge on modeling and simulation of discrete-event systems, and choosing the right topics to cover in a simulation class is not an easy task in simulation education. Key topics addressed so far in this chapter are as follows:

1. How to perform a manual simulation for executing an event graph model
2. How to develop a reference model of a DES
3. How to build an event-graph model and simulate it with a simulation package
4. How to build an ACD model and convert it to an EFD model
5. How to simulate an EFD model with a commercial simulation package
6. How to build a state-graph model and simulate it with a simulation package

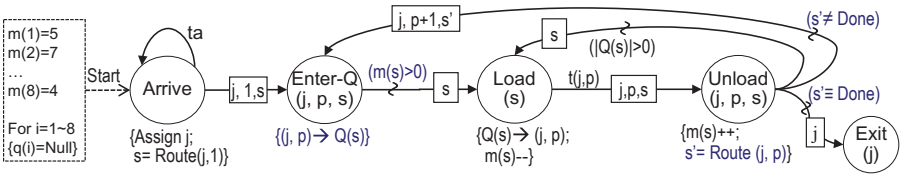


Fig. 2.26. Event graph model of a homogeneous job shop.

2.6.1 Event-Based M&S and Event-Graph Simulation with SIGMA®

At the heart of discrete-event system simulation is the concept of event-based modeling and simulation with an event graph. Thus, it is essential for an engineering simulation student to learn how to build and simulate an event-graph model of a DES. An event-graph model may be less intuitive than an entity-flow diagram model, but it is very flexible and powerful for describing complex discrete-event systems concisely.

Shown in Fig. 2.26 is an event-graph model of a homogeneous job shop that can be used as a template for modeling various types of job shops such as machine shops, electronics Fab, restaurants, hospitals, etc.

Basics of event-graph modeling and simulation are presented in Chapter 4. How to model a large system as a parameterized event graph with SIGMA® is discussed in Chapter 5, where the job shop model of Fig. 2.26 will be executed with SIGMA®.

2.6.2 Activity-Based M&S and Hands-On Modeling Practice with Arena®

It is essential for an undergraduate simulation class in an engineering school to give students hand-on experiences on modeling with a popular simulation package. There are quite a few simulation packages based on the entity-flow view (also known as the process-oriented or process-interaction view). Examples include Arena®, AutoMod®, and EXTEND®. Most of those packages are quite simple to learn and use, and student copies are readily available free of charge.

It would be enough for the students to get exposed to one package. Once the students get used to one package, they will be able to learn other packages by themselves. For this purpose, we chose to use Arena® in addition to the ACD simulator ACE® in this book. An approach to converting an ACD model to an Arena simulation program is elaborated in Chapter 7.

2.6.3 State-Based M&S

The subject of state-based modeling and other advanced topics in Part III may be skipped in an undergraduate simulation class in an ordinary engineering

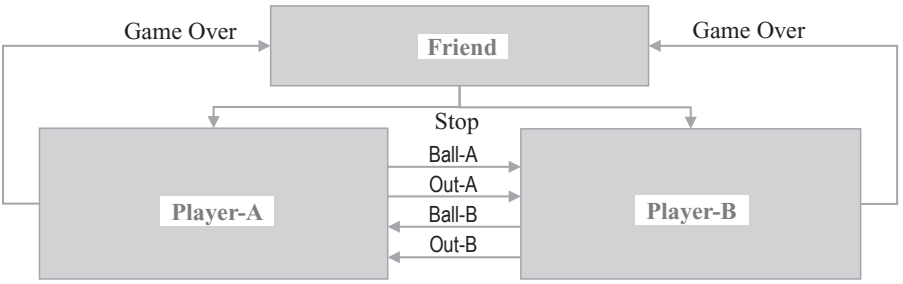


Fig. 2.27. Object interaction diagram model of a table tennis game.

school. This topic may be covered in a graduate-level class. State-based modeling is suitable for modeling a discrete-event system that is naturally described in terms of interacting objects in the system. As can be seen in the modeling example of the two-stage tandem line, the state graph may not be a suitable tool for modeling such a system. On the other hand, a complex urban traffic network may be properly modeled by a state graph.

Figure 2.27 shows an object interaction diagram model of a table tennis game played by two players—Player-A and Player-B—with their friend watching the game. The two players interact with each other by sending a Ball event message (meaning that the ball is sent to the opponent's table) or an Out event message (when the ball went out of bounds). The friend may send a Stop message to the players to interrupt in the middle of game, and the players send a Game-over message to the friend when the game is over. When modeling this kind of system, state-based modeling would be the choice. Detailed discussions on the subject may be found in Chapter 9 of this book.

2.7 REVIEW QUESTIONS

2.1. What is an event routine?

2.2. What is the next event?

2.3. What are the three logical modeling components?

2.4. What is a modeling formalism?

2.5. What are the three worldviews in discrete-event system modeling?

2.6. What is a reference model of a discrete-event system?

2.7. What are the requirements of a formal model?

2.8. What is model qualification?

2.9. What is simulator calibration?

PART II

FUNDAMENTALS OF DISCRETE-EVENT SYSTEM MODELING AND SIMULATION

A practical definition of a *discrete-event system* (DES) is given as "a system designed to process some sort of entities with some kind of resources." Examples of DESs are hospital emergency rooms and operating rooms, car repair shops, serial assembly lines, semiconductor fabrication lines, restaurants, urban traffic systems, etc. Part II, which is the main part of the book, is concerned with how to build simulation models of these DESs and perform simulation analyses. All the three *classical modeling formalisms*—event-based, activity-based, and entity-based formalisms—together with input modeling and output analyses are covered in Part II. The three classical modeling formalisms are also known as *event scheduling*, *activity scanning*, and *process-interaction worldviews*. There are six chapters, Chapters 3 to 8, in Part II.

Chapter 3 and Chapter 8, respectively, cover all the essential *input modeling and output analyses* topics that a simulation practitioner should know. After studying these two chapters, you should be able to do the following:

1. Generate inter-arrival times and service times from empirical data
2. Generate various theoretical random variates
3. Generate inter-arrival times for fluctuating arrival rates
4. Estimate the parameters of various distribution functions
5. Verify and calibrate the simulation logic
6. Compute confidence intervals of simulation outputs
7. Apply the response surface methodology to simulation optimization

Chapters 4 and 5 are devoted to *event-based modeling and simulation* (M&S). By studying these two chapters, you should be able to do the following:

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.

1. Provide formal specifications of (parameterized) event graph models
2. Build event graph models of various types of systems including job shops
3. Execute event graph models with the event-based simulator SIGMA®
4. Develop your own event graph simulator

Chapters 6 and 7 are devoted to *activity-based and entity-based M&S*, respectively. By studying these two chapters, you should be able to do the following:

1. Provide formal specifications of activity cycle diagram (ACD) models
2. Build ACD models of various types of systems including job shops
3. Execute ACD models with the activity-based simulator ACE®
4. Use the entity-based simulator Arena®
5. Convert ACD models into Arena® models and perform simulation runs.

CHAPTER 3

Input Modeling for Simulation

As far as the laws of mathematics refer to reality, they are not certain, and as far as they are certain, they do not refer to reality.

—Albert Einstein

3.1 INTRODUCTION

Discrete-event dynamic systems have some probabilistic elements, and a close match between the simulation input model and the true underlying probabilistic mechanism associated with the source system is required for successful simulation analyses. *Input modeling* defines mechanisms for generating random inputs of a simulation model. The general question is how to model a probabilistic element such as the arrival process or service times given a data set collected on the element of interest [Leemis 2001].

Let's assume that you have an automatic teller machine (ATM) in your building and that you have collected the data in Table 3.1 by observing the first 10 customers during a lunch hour. The ATM and the nearby floor space can be modeled as a single server system whose reference model and event graph model were given in Fig. 2.1 of the previous chapter. Then, how would you use the data listed in Table 3.1 to simulate your ATM system?

In general, if the actual data collected are available, there are three ways to model input: (1) *trace-driven simulation*, in which the collected data values are directly used in the simulation; (2) *empirical input modeling*, in which random variables for simulation are generated directly from the collected data; (3) *theoretical input modeling*, in which the parameters of a theoretical distribution function are estimated from the actual data and random variables are generated from the fitted distribution function.

This chapter is organized as follows. We start with the subject of empirical input modeling (for inter-arrival times as well as service times) in Section 3.2, and follow with a brief section (Section 3.3) on *theoretical distribution fitting*.

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.

TABLE 3.1. Collected Data for the First 10 Customers of an ATM during a Lunch Hour

Observation number (k)	1	2	3	4	5	6	7	8	9	10
Inter-arrival time in sec $\{A_k\}$	121	13	87	36	7	236	8	33	152	67
Service time in sec $\{X_k\}$	56	51	73	65	84	58	62	69	44	66

Section 3.4 deals with theoretical input modeling of inter-arrival times, and Section 3.5 deals with theoretical input modeling of service times. The last section (Section 3.6) covers input modeling for special applications.

3.2 EMPIRICAL INPUT MODELING

In empirical input modeling, random variables for simulation are generated directly from the collected data. There are three ways to generate random inputs from collected data $\{X_i\}$ without fitting a theoretical distribution function: the nonparametric modeling method, empirical modeling of individual data, and empirical modeling of grouped data.

3.2.1 Nonparametric Modeling

A simple approach to generating random inputs from collected data $\{X_i\}$ is to use a nonparametric model, in which the value of the random variable x is repeatedly sampled from collected data $\{X_k: k = 1 \sim n\}$ with probability $1/n$. The nonparametric input modeling method may be implemented as follows:

1. Generate a uniform random number $U \sim \text{Uni}(0,1)$.
2. Set $P = n \times U$ and compute the index $k = [P] + 1$; // $[P]$ is the integer part of P.
3. Return $x = X_k$.

Example 3.1, Generate a service time using the nonparametric method from the service time data $\{X_k\}$ given in Table 3.1. Let the value of the generated uniform random number U ~ Uni (0, 1) be 0.369881, then the index 'k' in the nonparametric modeling method is computed as $k = [n \times U] + 1 = [10 \times 0.369881] + 1 = 4$. Thus, $x = X_4 = 65$ is sampled as an empirical service-time to be used in a simulation.

3.2.2 Empirical Modeling of Individual Data

Let $\{X_{(k)}: k = 1 \sim n\}$ be the individual ordered sample data in an increasing order, then their empirical distribution $F(x)$ is a piecewise linear function with $F(X_{(k)}) = (k - 1) / (n - 1)$. Now, generate a uniform random number U and

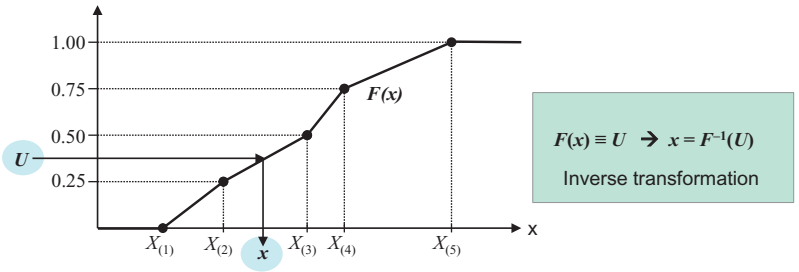


Fig. 3.1. Generation of random variable x from individual data ($n = 5$).

TABLE 3.2. Ordered Service-Time Data Obtained from the Collected Data in Table 3.1

Ascending order index (J)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Ordered service time data $\{X_{(J)}\}$	44	51	56	58	62	65	66	69	73	84

then sample the value of a random variable x from $F(x)$ as depicted in Fig. 3.1, which is called *inverse transformation*.

1. Generate a uniform random number $U \sim U(0,1)$.
2. Set $P = (n - 1) \times U$ and $J = [P] + 1$; // $[P]$ is the integer part of P.
3. Return $x = X_{(J)} + (P - J + 1) \times (X_{(J+1)} - X_{(J)})$.

Example 3.2. Table 3.2 shows the ordered service-time data $\{X_{(k)}\}$ that were obtained by rearranging the collected data in Table 3.1 in an ascending order. If the value of U is 0.369881 as before, we have $P = (n - 1) \times U = 9 \times 0.36988$ 1 = 3.328929 and $J = [P] + 1 = 4$. Then, an empirical service time is generated as:

$$x = X_{(4)} + (P - 4 + 1) \times (X_{(4+1)} - X_{(4)}) = 58 + (0.328929) \times (62 - 58) = 58 + 1.316 = 59.316.$$

3.2.3 Empirical Modeling of Grouped Data

When the data are grouped into m adjacent intervals $\{[a_0, a_1), [a_1, a_2) \ldots [a_{m-1}, a_m]\}$ and the j^{th} interval contains n_j observations, the grouped data distribution function $G(x)$ is also a piecewise linear function with $G(a_0) = 0$ and $G(a_j) = \sum_1^j n_i / n$ for $j = 1 \sim m$ (with $n = \Sigma\, n_j$).

Then, as depicted in Fig. 3.2, the random variable x can be sampled from the empirical distribution $G(x)$ using the following inverse transformation method:

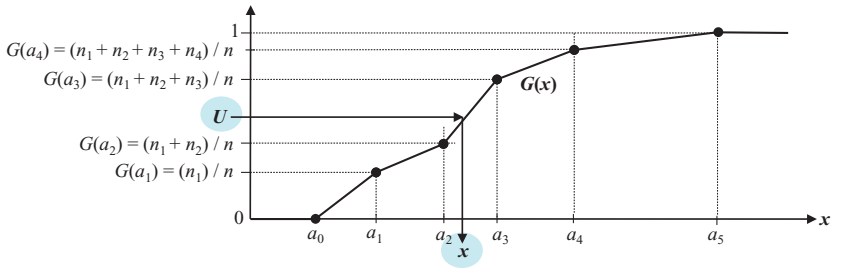


Fig. 3.2. Generation of random variable from grouped data ($m = 5$).

TABLE 3.3. Grouped Service-Time Data Obtained from the Ordered Data in Table 3.2

Group index (j)	**1**	**2**	**3**	**4**	**5**
Intervals of service times	$a_0 = 40 - 50$	$a_1 = 50 - 60$	$a_2 = 60 - 70$	$a_3 = 70 - 80$	$a_4 = 80 - 90 = a_5$
Frequency (n_j)	1	3	4	1	1
$G(a_j) = (\Sigma(n_j)/n)$	1/10	4/10	8/10	9/10	1.0

1. Generate $U \sim U(0,1)$.
2. Find integer J such that $G(a_J) \le U < G(a_{J+1})$.
3. Return $x = a_J + [U - G(a_J)] \times (a_{J+1} - a_J) / [G(a_{J+1}) - G(a_J)]$.

Exercise 3.1. Table 3.3 shows the grouped service-time data that were obtained by grouping the ordered data of Table 3.2 into five adjacent intervals {[40, 50), . . . , [80,90]}. Assuming the value of U is 0.369881 as before, generate an empirical service time.

3.3 OVERVIEW OF THEORETICAL DISTRIBUTION FITTING

Distribution fitting is a classical statistical estimation process consisting of data independence checking, distribution function selection, parameter estimation, and goodness-of-fit testing.

3.3.1 Data Independence Checking

The first step in theoretical input modeling is to check whether the obtained data are independent. A simple method of assessing data independence is to plot a scatter diagram. For the data $X_1, X_2 \ldots X_n$ listed in time-order of collection, pairs (X_i, X_{i+1}) for $i = 1 \sim n - 1$ are plotted on an x-, y-coordinate system

(X_i as x-value and X_{i+1} as y-value). If the plotted points are randomly scattered, one may conclude that the data are independent.

3.3.2 Distribution Function Selection

The second step is to select a suitable candidate distribution function based on some theoretical justification and/or by observing the shape of the histogram. For example, exponential distribution and Erlang distribution are commonly selected for inter-arrival times, while Weibull distribution is the choice for an interfailure time distribution. Service-time distributions that are widely used are beta distribution and lognormal distribution.

3.3.3 Parameter Estimation

The third step is to estimate the parameters of the selected distribution. Maximum likelihood estimator (MLE) is the preferred choice for parameter estimation, but other methods may be used when the MLE does not have a simple form. For example, the MLE is used for exponential, normal, and lognormal distributions; the method of moment for Erlang and beta distributions; the rank regression method for Weibull distribution. More details may be found in Appendix 3A of this chapter.

3.3.4 Goodness-of-Fit Test

The fourth step of theoretical distribution fitting assesses the model adequacy by using a goodness-of-fit test such as the chi-square test. Here, data are grouped into m adjacent intervals $\{[a_0, a_1), [a_1, a_2) \ldots [a_{m-1}, a_m]\}$ so that the j^{th} interval contains n_j observations (with $n = \Sigma n_j$), and a test statistic χ^2 is constructed using the expected proportion p_j computed from the fitted density function $\hat{f}(x)$ as follows:

$$\chi^2 = \sum_{j=1}^{m} \frac{(n_j - np_j)^2}{np_j}, \text{ where } p_j = \int_{a_{j-1}}^{aj} \hat{f}(x)dx. \tag{3.1}$$

Then, the test statistic is checked against the chi-square value with $(m - 1)$ degrees of freedom. An extensive treatment on the subject may be found in Law [2007].

3.3.5 Overview of Random Variate Generation

Having fitted a theoretical distribution for each type of input model, the final phase of input modeling generates random variates for simulation. When the distribution function has a closed-form inverse function, the inverse-transform method is the choice. Otherwise, special methods of generating random variables may be employed. More details may be found in Appendix 3B.

TABLE 3.4. Summary of Theoretical Distribution Fitting and Input Modeling

Input Variable Types	Distributions	Parameter Estimation	Generation Methods
Inter-arrival time	Exponential (θ)	Maximum likelihood method	Inverse-transform
	Erlang (k, θ)	Method of moment	Convolution of exponential
Service time (Repair time)	Triangular (a, b, c)		Composition method
	Beta (α, β)	Method of moment	Acceptance–rejection
	Normal (μ, σ)	Maximum likelihood method	Box & Muller method
	Lognormal (μ, σ)	Maximum likelihood method	Conversion of normal variate
Interfailure time	Weibull (α, β)	Rank regression method	Inverse-transform

Table 3.4 summarizes the distribution functions for different input model types, methods of parameter estimation, and methods of random variate generation. Erlang and exponential distributions are exclusively used in modeling inter-arrival times, whereas beta and uniform distributions are commonly used in modeling service times. Due to its flexibility, Weibull distribution is mostly used in modeling interfailure times.

There are three popular methods for parameter estimation: the maximum likelihood method for estimating the parameters of exponential and normal distributions; the method of moment for Erlang and beta distributions; and the rank regression method for Weibull distribution. The most popular method for random variate generation is the inverse-transform method, which is used in generating exponential, Erlang, and Weibull random variates. Details on the subjects are covered in the appendixes of this chapter.

3.4 THEORETICAL MODELING OF ARRIVAL PROCESSES

3.4.1 Theoretical Basis for Arrival Process Modeling

A Poisson process is a continuous stochastic process in which events occur independently of one another. The Poisson process is a collection $\{N(t): t \geq 0\}$ of random variables, where $N(t)$ is the number of events (arrivals) that have occurred up to time t (starting from time 0).

For a homogeneous Poisson process, the number of arrivals between time t and time $t + s$ is given as $N(t + s) - N(t)$ and has a Poisson distribution. Let λ be the arrival rate (expected number of arrivals in any interval of length 1), then the probability of k arrivals during $[t, t + s]$ is given by [Cinlar 1975]:

$$P[N(t+s)-N(t)=k]=e^{-\lambda s}(\lambda s)^k/k! \text{ for } k=0,1,2,\ldots$$

Consider the waiting time T_1 until the first arrival. Clearly T_1 is more than s if and only if the number of arrivals before time s is 0. Combining this property with the above probability distribution for the number of homogeneous Poisson process events in a fixed interval gives

$$\mathrm{P}[T_1>s]=P[N(s)=0]=P[N(s)-N(0)=0]=e^{-\lambda s}(\lambda s)^0/0!=e^{-\lambda s}. \quad (3.2)$$

Consequently, the waiting time until the first arrival T_1, which is equivalent to an inter-arrival time, has an exponential distribution with a density function given by $f(\mathrm{t}) = \lambda e^{-\lambda s}$ and its expected value given by $E(T_1) = \theta = 1/\lambda$.

The waiting times between k occurrences of the event in a homogeneous Poisson process follow an *Erlang distribution*, which was developed by A.K. Erlang to examine the number of telephone calls that might be made at the same time to the operators of the switching stations.

3.4.2 Generation of Inter-Arrival Times for a Constant Arrival Rate

When the arrival process is stationary with an arrival rate λ, the inter-arrival times follow *Erlang (k,θ)* with $\theta = 1/\lambda$. It becomes an exponential distribution when $k = 1$. The shape of the density function is dependent on the shape of parameter k and scale parameter θ as can be seen in Fig. 3.3, and the density function is defined as

$$f(x)=\frac{\theta^{-k}x^{k-1}e^{-x/\theta}}{(k-1)!}.$$

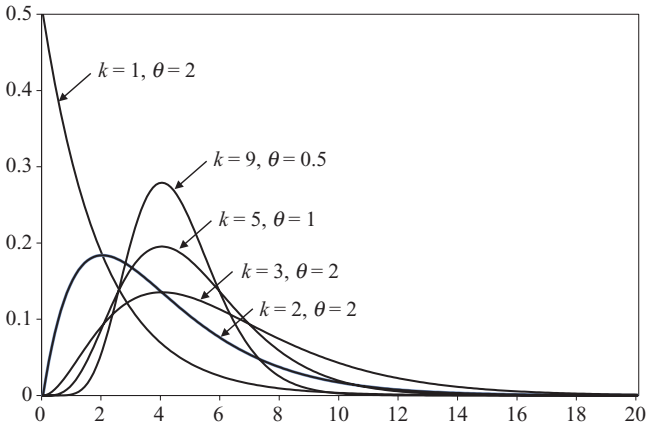


Fig. 3.3. Erlang-k density function.

3.4.2.1 Parameter Estimation If a sufficient amount of inter-arrival time data $\{X_i\}$ is available, estimators of the parameters are obtained from the sample moments as discussed in Appendix 3A (k is an integer):

$$\hat{k} \cong (m_1)^2/[m_2 - (m_1)^2]; \quad \hat{\theta} = [m_2 - (m_1)^2]/m_1, \tag{3.3}$$

where the first sample moment m_1 and second sample moment m_2 are given by

$$m_1 = \frac{1}{n}\sum_{i=1}^{n} x_i = \bar{x}; \quad m_2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2.$$

If $k = 1$, it becomes an exponential distribution and Eq. 3.3 reduces to

$$\hat{\theta} = m_1 = \bar{x}. \tag{3.4}$$

Example 3.3. The parameters of the Erlang distribution representing the inter-arrival distribution of the ATM system introduced in Section 3.1 can be estimated from the inter-arrival times data $\{A_k\}$ in Table 3.1 as follows. The first and second sample moments are calculated as $m_1 = 76$ and $m_2 = 10{,}816.6$. From Eq. 3.3, the Erlang parameters k and θ are computed as $k = 1.146$ and $\theta = 66.3$. Since k is close to 1, we have an exponential distribution. Thus, the inter-arrival times in the ATM system follow an exponential distribution with mean = 66.3.

3.4.2.2 Random Variate Generation As described in Appendix 3B, an exponential random variate x is generated via an inverse transformation as follows:

1. Generate $u \sim U(0,1)$.
2. Return $-\hat{\theta}\ln(u)$.

Utilizing the fact that the sum of independent exponential random variables is an Erlang random variable, the *Erlang-k* random variate x is generated as follows:

1. Generate independent $u_i \sim U(0.1)$ for $i = 1 \sim k$.
2. Return $-\hat{\theta}\ln\left(\prod_{i=1}^{k} u_i\right)$.

3.4.3 Generation of Inter-Arrival Times for Varying Arrival Rates

Let's assume the inter-arrival times are exponentially distributed, but the arrival rate $\lambda(t)$ is changing over time. This arrival process is called a *nonstationary Poisson process*, which is common in many service systems such as banks, cafeterias and barber shops.

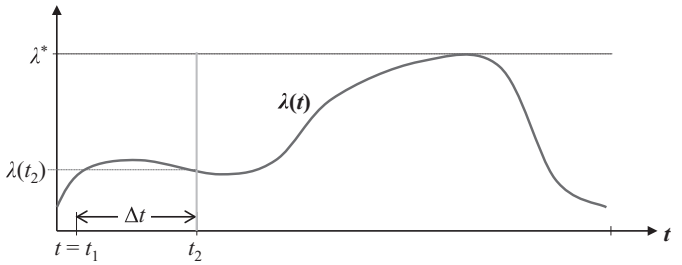


Fig. 3.4. Generation of exponential inter-arrival times for varying arrival rates.

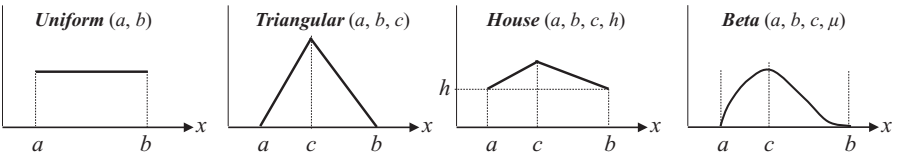


Fig. 3.5. Service-time distributions in the absence of data.

A widely used method called *thinning* for generating nonstationary inter-arrival times is shown in Fig. 3.4. It starts with a previous arrival time $t = t_1$ and generates an inter-arrival time Δt for the maximum arrival rate λ^* by using a uniform random number U_1. Thus, from the result of Section 3.4.1, we have $\Delta t = -(1/\lambda^*) \ln(U_1)$ and $t_2 = t + \Delta t$. Now generate another uniform random number U_2, and if $U_2 \leq \lambda(t_2) / \lambda^*$, then accept t_2 as the next arrival time, else set $t = t_2$ and start over. The *thinning method* of generating arrival times may be summarized as follows (start with $i = 1$):

1. Set: $t = t_{i-1}$.
2. Generate: $U_1 \sim U(0, 1)$ and $U_2 \sim U(0, 1)$.
3. $D = -(1/\lambda^*) \ln(U_1)$; //exponential random variable with $\theta = 1/\lambda^*$.
4. $t = t + D$.
5. If $U_2 \leq \lambda(t)/\lambda^*$, then return $t_i = t$, else go back to step 2.

3.5 THEORETICAL MODELING OF SERVICE TIMES

3.5.1 Generation of Service Time in the Absence of Data

In some simulation studies it may not be possible to collect the service-time data, but we have some knowledge or information about the service time distribution, such as its *range* [*a*, *b*] and *mode* c. The lower bound '*a*' is often referred to as the most *optimistic estimate* of service time, the upper bound '*b*' as the most *pessimistic estimate*, and the mode '*c*' as *most-probable estimate*. Figure 3.5 shows the density functions that are commonly used in modeling

service times and activity durations in the absence of collected data: uniform, triangular, house, and beta distribution.

***3.5.1.1 Uniform Random Variate:* X~ Uniform (a, b)** When only the range data $[a, b]$ is given, a simple but useful method is to generate the service time X from the uniform distribution $U(a, b)$. Namely, let $U \sim U(0,1)$ then the uniform random variate $X \sim U(a, b)$ is generated by

$$X = a + (b - a) \times U. \tag{3.5}$$

***3.5.1.2 Triangular Random Variate:* X~ Triangular (a, b, c)** If the mode c is also given in addition to the range data $[a, b]$, service times may be sampled from a triangular distribution. Service-time random variate X following the triangular distribution Triangular *(a, b, c)* may be generated by using a composition method (see Appendix 3B) given below:

1. Set $p = (c - a)/(b - a)$.
2. Generate $U_1 \sim U(0,1); U_2 \sim U(0,1)$.
3. If $U_1 \le p$, then $X = a + (c - a)\sqrt{U_2}$, else $X = c + (b - c)(1 - \sqrt{1 - U_2})$.

***3.5.1.3 House Distribution Random Variate:* X~ House (a, b, c, h)** When the height (h) the house is also specified (in addition to a, b, and c), service time may be sampled from the house distribution (See Fig. 3.5). As depicted in Fig. 3.6, a composition method is employed to generate house-distribution random variates X.

1. Set $r = h(b - a); p = (1 - r)(c - a)/(b - a); q = 1 - (p + r)$.
2. Generate $U_1 \sim U(0,1)$ *and* $U_2 \sim U(0,1)$.
3. If $(U_1 \le p)$, then $X = a + (c - a)\sqrt{U_2}$ (see Fig. 3B.3 in Appendix 3B), else if $(U_1 \le p + q)$, then $X = c + (b - c)(1 - \sqrt{1 - U_2})$, else $X = a + (b - a)(U_2)$.

***3.5.1.4 Beta Random Variate:* X~Beta(α,β)** A choice for the service-time distribution with a finite range is a beta distribution. The density function $f(x)$ of the standard beta distribution $Beta(\alpha,\beta)$ that has a unit range $[0, 1]$ is given by

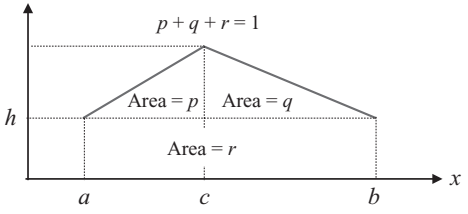


Fig. 3.6. Generation of "house" random variate via the composition method.

$$f(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)},$$

where $B(z_1, z_2) = \int_0^1 t^{z_1-1}(1-t)^{z_2-1}dt$ denotes the beta function.

Let c and μ, respectively, denote the *mode* and *mean* of the beta distribution with a finite range [*a, b*]. If $c \neq \mu$, the parameters α, β of the standard beta distribution $Beta(\alpha,\beta)$ may be estimated from the following relations (see Appendix 3A):

$$\hat{\alpha} = \frac{u(2m-1)}{m-u}; \hat{\beta} = \frac{(1-u)}{u}\hat{\alpha}; m = (c-a)/(b-a) \,\&\, u = (\mu-a)/(b-a), \quad (3.6)$$

where m and u are the *mode* and *mean* of the standard beta distribution.

There are quite a few methods for generating beta random variates [Law 2007, Cheng 1978]. A simple yet effective algorithm by Cheng [1978] for generating $Y \sim Beta(\hat{\alpha}, \hat{\beta})$ is given in Appendix 3B. Then the beta random variate X with a general range [*a, b*] can be obtained from Y as follows:

$$X = a + (b-a)Y.$$

3.5.2 Generation of Service Times from Collected Data

When data $\{X_i\}$ collected from the target system are available, the first step may be to construct the histogram of $\{X_i\}$ to identify the range and shape of the distribution. Popular candidates for service time distribution are beta and lognormal as shown in Fig. 3.7.

3.5.2.1 Beta Random Variate: X~Beta(α,β) If the service times have a finite range [*a*, *b*], beta distribution is the choice for generating them. The parameters of the standard beta distribution $Beta(\alpha,\beta)$ can be estimated using the following equation (see Appendix 3A):

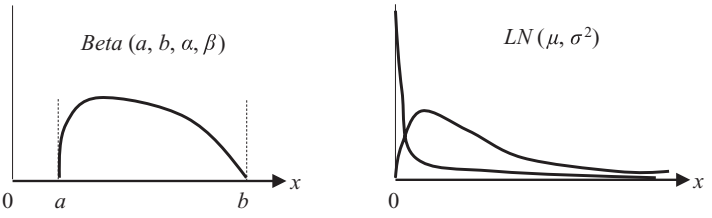


Fig. 3.7. Service-time distributions when collected data are available.

$$\tilde{\alpha} = u\left(\frac{u(1-u)}{v} - 1\right); \quad \hat{\beta} = (1-u)\left(\frac{u(1-u)}{v} - 1\right), \tag{3.7}$$

where the u and v are the *mean* and *variance* of the standard beta distribution, and they are given by

$$u = (\bar{x} - a)/(b - a); \quad v = s^2/(b - a)^2, \tag{3.8}$$

where the sample mean and sample variance are computed from the collected data:

$$\bar{x} = \tfrac{1}{n}\sum_{i=1}^{n} X_i; \quad s^2 = \tfrac{1}{n-1}\sum_{i=1}^{n} (X_i - \bar{x})^2.$$

Once the parameters are estimated from Eq. 3.7, the beta random variate is generated the same way as discussed in Section 3.5.1.

Example 3.4. The ATM service-time data shown earlier in Table 3.1 have a sample mean of 62.8 and sample variance of 129.96. The range of the ATM service time is [a = 40, b = 90]. Thus, from Eq. 3.8, the mean and variance of the standard beta distribution are obtained as u = (62.8 − 40)/50 = 0.456; v = 129.96/(50 × 50) = 0.052. Finally, from Eq. 3.7, the parameters of the standard beta distribution are computed as

$$\tilde{\alpha} = 0.456[0.456(1 - 0.456)/0.052 - 1] = 1.72; \hat{\beta} = 2.05.$$

3.5.2.2 Lognormal Random Variate: **X~LN(μ,σ²)** Finally, if the service-time distribution is skewed to the right, they are generated from the lognormal distribution $LN(\mu,\sigma^2)$ whose parameters can be estimated as

$$\hat{\mu} = \frac{1}{n}\sum In\, X_i; \hat{\sigma} = \left[\frac{1}{n}\sum (In\, X_i - \hat{\mu})^2\right]^{1/2}. \tag{3.9}$$

For given parameters μ and σ^2, the relationship between the lognormal random variate $X \sim LN(\mu,\sigma^2)$ and the normal random variate $Y \sim N(\mu,\sigma^2)$are as follows:

$$X = e^Y. \tag{3.10}$$

Thus, lognormal random variates $X \sim LN(\mu,\sigma^2)$ can be generated from normal random variates as follows:

1. Generate $Y \sim N(\mu,\sigma^2)$. (See Appendix 3B.5.)
2. Return $X = e^Y$.

3.6 INPUT MODELING FOR SPECIAL APPLICATIONS

3.6.1 Interfailure Time Modeling

Interfailure time is modeled by the Weibull distribution *Weibull* (α,β) mainly due to its flexibility as shown in Fig. 3.8. If the failure rate decreases over time, then use $\alpha < 1$; if the failure rate is constant over time, then use $\alpha = 1$ (i.e., exponential distribution); if the failure rate increases over time, then use $\alpha > 1$.

As presented in Appendix 3A, Weibull parameters can be estimated from the collected interfailure time data $\{X_i\}$ by using a rank regression method. In order to estimate the parameters, the collected data $\{X_i\}$ are rearranged in an increasing order to obtain a set of ordered sample data $\{X_{(i)}: i = 1 \sim n\}$ and the median rank R_i of the i-th sample data $X_{(i)}$ is computed using the following equation:

$$R_i = (i - 0.3)/(N + 0.4). \tag{3.11}$$

Then, the sequence of ordered rank data pairs $\{X_{(i)}, R_i\}$ are fitted to the Weibull distribution as depicted in Fig. 3.9.

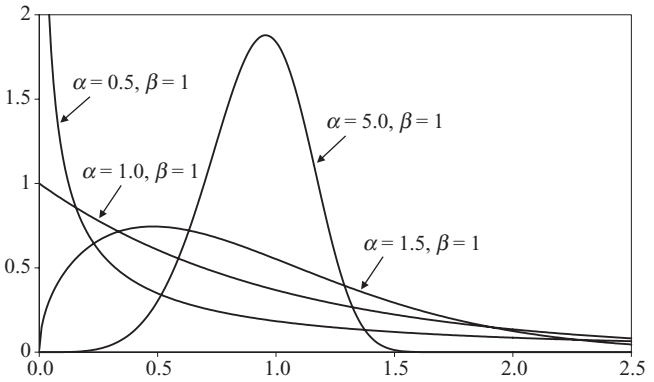


Fig. 3.8. Weibull density function.

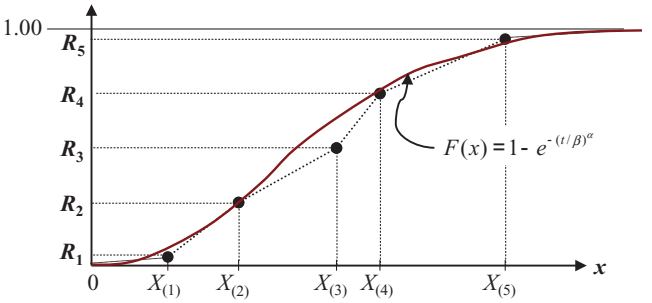


Fig. 3.9. Estimation of Weibull parameters via rank regression.

As described in Appendix 3A, the procedure for estimating Weibull parameters α, β from the ordered-rank pairs $\{X_{(i)}, R_i\}$ may be summarized as follows:

1. Transform data $\{X_{(i)}, R_i\}$ to form a linear equation $(y = a + bx)$

$$x_i = \ln(X_{(i)}); \quad y_i = \ln\{-\ln[1 - R_i]\}. \tag{3.12}$$

2. Compute the least-square estimators of a and b

$$\hat{b} = \left[\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}\right] / \left[\sum_{i=1}^{n} x_i^2 - n\bar{x}^2\right]; \quad \hat{a} = \bar{y} - \hat{b}\bar{x}.$$

3. Obtain the Weibull parameter estimators

$$\hat{\alpha} = \hat{b}; \quad \hat{\beta} = e^{-(\hat{a}/\hat{b})}. \tag{3.13}$$

Since the Weibull distribution function is easily inverted, Weibull random variates are generated employing the inverse-transform method. Namely, *Weibull* (α,β) random variate X is obtained from a uniform random variate U as follows:

$$X = \beta * (-\ln U)^{1/\alpha}. \tag{3.14}$$

3.6.2 Inspection Process Modeling

A Bernoulli process is a discrete-time stochastic process consisting of a sequence of independent random variables $\{X_i\}$ taking values over two symbols (0 or 1) such that $P[X_i = 1] = p$ for all i. Distribution functions associated with the Bernoulli process include *binomial bin*(t,p) and *negative* binomial *negbin (s,p)* distributions (geometric is a special case of negative binomial).

1. *bin (t,p)*: $\binom{t}{x} p^x (1-p)^{t-x}$ for $x = 0\sim$ t.
2. *negbin (s,p)*: $\binom{s+x-1}{x} p^s (1-p)^x$ for $x = 0, 1, 2 \cdots$

In quality control system simulations, input modeling of defective items is required. As an inspection process is a Bernoulli process, the number of defective items in a batch of size b can be sampled from the binomial distribution *bin (b,p)* and the number of inspections before encountering d defective items can be sampled from the negative binomial distribution *negbin(d,p)*. Here, p is the probability of an item is defective. What distribution is used for the number of inspections before encountering the first defective item?

3.6.3 Batch Size Modeling

In many service systems, customers may arrive in groups or batches. When frequency data collected from the source system is available and reliable, a simple yet effective method for generating batch sizes is the empirical modeling of the original frequency data. Namely, a batch size is sampled from the frequency data.

If the frequency data are not available (or unreliable) but the average batch size $\overline{B}$ is given, a theoretical distribution fitting method can be employed. When the maximum batch size b is given ($1 \le B \le b$), the binomial distribution $X \sim bin\ (t,p)$ is a choice. The parameters of binomial distribution are estimated as follows:

$$\hat{t} = b - 1; \quad \hat{p} = (\overline{B} - 1)/t. \tag{3.15}$$

Then, batch sizes following the binomial distribution *bin (t,p)* are generated as

1. Generate $\{U_i \sim \mathrm{U}(0,1)$ for $\mathrm{I} = 1 \sim t\}$.
2. For $i = 1 \sim t$ {If $U_i \le p$ then $Z_i = 1$, else $Z_i = 0$}.
3. $X = \Sigma Z_i$.
4. Return $B = X + 1$.

3.7 REVIEW QUESTIONS

3.1. What is trace-driven simulation?

3.2. What are the three ways to empirically generate random inputs from collected data?

3.3. What is the nonparametric input modeling method?

3.4. What is the inverse-transformation method of generating a random variable?

3.5. Where is a scatter diagram plotting used?

3.6. What test is widely used in the goodness-of-fit test?

3.7. How is the Erlang distribution defined in a homogeneous Poisson process?

3.8. What is a nonstationary Poisson process?

3.9. What is the thinning method of generating a next arrival time?

3.10. What is a Bernoulli process?

APPENDIX 3A: PARAMETER ESTIMATION

In this appendix, how to estimate parameters for the major continuous distributions listed in Table 3.4 will be explained in just enough detail for readers to implement their own input modeling functions. Distributions covered are exponential, Erlang, Beta, Weibull, normal, and lognormal. For a more comprehensive treatment on the subject, the reader is referred to Law [2007].

3A.1 Exponential Distribution

The exponential distribution $Expo(\theta)$ is defined by the scale parameter θ, and it is widely used in the field of queuing theory mainly due to its simplicity. For $x > 0$, the density function $f(x)$ and distribution function $F(x)$ are given by

$$f(x) = \frac{1}{\theta} e^{-x/\theta}; \; F(x) = 1 - e^{-x/\theta}. \tag{3A.1}$$

The mean and variance of an exponentially distributed random variable are θ and θ^2, respectively.

The parameter θ is estimated by using the maximum likelihood method, and the resulting estimator is called a *maximum likelihood estimator* (MLE). Let $\{x_i$: for $i = 1 \sim n\}$ denote n independent observations, then the likelihood function is given by the product of density functions as follows:

$$L(x_1, x_2 \cdots x_n \mid \theta) = \prod_{i=1}^{n} f(x_i \mid \theta) = (1/\theta)^n e^{-(\sum x_i)/\theta}. \tag{3A.2}$$

The natural logarithm of the likelihood function is expressed as $(\bar{x} = \left(\sum_{i=1}^{n} x_i\right)/n)$:

$$\Lambda = \ln L = n \ln(1/\theta) - \left(\sum_{i=1}^{n} x_i\right)/\theta = -n \ln \theta - n\bar{x}/\theta. \tag{3A.3}$$

Differentiating Λ, setting it equal to 0, and solving for θ, the MLE is obtained as:

$$\partial\Lambda/\partial\theta = -n[1/\theta + \bar{x}(-1/\theta^2)] = 0 \Rightarrow \hat{\theta} = \bar{x}. \tag{3A.4}$$

3A.2 Erlang Distribution

The Erlang distribution is defined by the shape parameter k and scale parameter θ and is widely used in modeling arrival processes. For $x > 0$ and positive integer k, the density function is given by (the distribution function does not have a simple form):

$$\text{Density function: } f(x) = \frac{\theta^{-k} x^{k-1} e^{-x/\theta}}{(k-1)!}. \tag{3A.5}$$

The mean (μ) and variance (σ^2) of an Erlang random variable are:

$$\mu = k\theta; \quad \sigma^2 = k\theta^2.$$

If $k = 1$, it becomes an exponential density function. If k is a real number, it becomes a gamma density function.

As there are no closed-form solutions for the MLE of the parameters, the *method of moment* is employed in estimating the parameters k and θ. Let $\{x_i$: for $i = 1 \sim n\}$ denote n independent observations, then the first sample moment m_1 and second sample moment m_2 are:

$$m_1 = \tfrac{1}{n}\sum_{i=1}^{n} x_i = \bar{x}; \quad m_2 = \tfrac{1}{n}\sum_{i=1}^{n} x_i^2. \tag{3A.6}$$

On the other hand, the first and second population moments $E(X)$ and $E(X^2)$ of the Erlang density function are:

$$E[X] = \int x f(x)dx = k\theta; \quad E[X^2] = \int x^2 f(x)dx = k(k+1)\theta^2. \tag{3A.7}$$

Equating the population moments in Eq. 3A.7 with the sample moments in Eq. 3A.6 and solving for the parameters k and θ, we obtain (k is an integer):

$$\hat{k} \cong (m_1)^2/[m_2 - (m_1)^2]; \quad \hat{\theta} = [m_2 - (m_1)^2]/m_1. \tag{3A.8}$$

3A.3 Beta Distribution

The beta distribution $Beta(\alpha,\beta)$ is defined by the shape parameter α and scale parameter β and is widely used in modeling service times, especially, in the field of project management. For $0 < x < 1$, the density function is given by (the distribution function does not have a simple form):

$$\text{Density function: } f(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}, \tag{3A.9}$$

where the beta function is $B(z_1, z_2) = \int_0^1 t^{z_1-1}(1-t)^{z_2-1}dt$. The mean μ and variance σ^2 of a beta random variable are:

$$\mu = \frac{\alpha}{\alpha+\beta}; \sigma^2 = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}. \tag{3A.10}$$

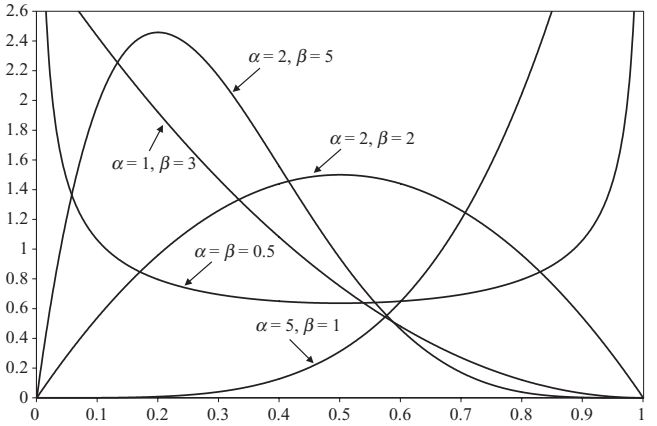


Fig. 3A.1. Beta density function.

Further, if $\alpha > 1$ and $\beta > 1$, the mode (m) of the density function is given by:

$$m = \frac{\alpha - 1}{\alpha + \beta - 2}. \tag{3A.11}$$

There are no closed-form solutions for the MLE of the parameters. Thus, a method of moment is employed in estimating the parameters. Namely, we solve Eq. 3A.10 for α and β, and then replace the population mean μ with sample mean $\bar{x}$ and population variance σ^2 with sample variance s^2 to have:

$$\hat{\alpha} = \bar{x}\left(\frac{\bar{x}(1-\bar{x})}{s^2} - 1\right); \hat{\beta} = (1-\bar{x})\left(\frac{\bar{x}(1-\bar{x})}{s^2} - 1\right), \tag{3A.12}$$

where $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$.

Alternatively, when $m \neq \mu$, the parameters may be expressed with respect to the mean μ and mode m as follows:

$$\hat{\alpha} = \frac{u(2m-1)}{m-u}; \hat{\beta} = \frac{(1-u)}{u}\hat{\alpha}. \tag{3A.13}$$

3A.4 Weibull Distribution

The Weibull distribution *Weibull (α,β)* is defined by the shape parameter α and scale parameter β and is widely used in the field of life data analysis due to its flexibility: If the failure rate decreases over time, then $\alpha < 1$; if it is constant, then $\alpha = 1$; if it increases, then $\alpha > 1$. For $x > 0$, the density function $f(x)$ and distribution function $F(x)$ are given by

$$f(t) = \alpha\beta^{-\alpha}t^{\alpha-1}e^{-(t/\beta)^{\alpha}};\ F(t) = 1 - e^{-(t/\beta)^{\alpha}}. \tag{3A.14}$$

As there are no closed-form solutions to estimating the parameters α and β, the estimates of the parameters are commonly obtained by using rank regression. A *rank regression method* for estimating α and β from a set of ordered sample data $\{X_{(i)}$ for $i = 1 \sim N\}$ will be explained in the following. An example of ordered data for a sample size of 6 ($N = 6$) is given below.

$$X_{(1)} = 16;\ X_{(2)} = 34;\ X_{(3)} = 53;\ X_{(4)} = 75;\ X_{(5)} = 93;\ X_{(6)} = 120. \tag{3A.15}$$

The median rank R_i of the i-th sample data $X_{(i)}$ of size N can be estimated using the following equation:

$$R_i = \frac{i - 0.3}{N + 0.4}. \tag{3A.16}$$

For the ordered data given in Eq. 3A.17, their median ranks may be estimated as:

$$R_1 = 0.11;\ R_2 = 0.26;\ R_3 = 0.42;\ R_4 = 0.58;\ R_5 = 0.73;\ R_6 = 0.89. \tag{3A.17}$$

In order to apply the rank regression method, the nonlinear distribution equation $F(t)$ in Eq. 3A.14 has to be linearized. Namely, rearranging $F(t)$ and taking the natural logarithm of both sides of the equation yields:

$$\ln[1 - F(t)] = -(t/\beta)^{\alpha} \rightarrow \ln\{-\ln[1 - F(t)]\} = -\alpha\ln(\beta) + \alpha\ln(t). \tag{3A.18}$$

The above equation is a linear equation of the form $y = a + bx$, where

$$y = \ln\{-\ln[1 - F(t)]\};\quad x = \ln(t) \tag{3A.19}$$

$$a = -\alpha\ln(\beta);\quad b = \alpha. \tag{3A.20}$$

As given by Eq. 8.16 in Chapter 8, the least-square estimators of the coefficients in the linear regression model ($y = a + bx$) are expressed as (n = sample size):

$$\hat{a} = \bar{y} - \hat{b}\,\bar{x};\quad \hat{b} = \left[\sum_{i=1}^{n} x_i y_i - n\bar{x}\,\bar{y}\right] \Big/ \left[\sum_{i=1}^{n} x_i^2 - n\bar{x}^2\right], \tag{3A.21}$$

where $\bar{x} = \sum x_i / n;\ \bar{y} = \sum y_i / n$. From the relations in Eq. 3A.21, x_i and y_i are expressed in terms of ordered sample data and median ranks as follows:

$$x_i = \ln(X_{(i)});\quad y_i = \ln\{-\ln[1 - R_i]\}. \tag{3A.22}$$

And, from Eq. 3A.22, the estimators of the parameters α and β are expressed as

$$\hat{\alpha} = \hat{b}; \quad \hat{\beta} = e^{-(\hat{a}/\hat{b})}. \tag{3A.23}$$

3A.5 Normal and Lognormal Distributions

Many measurements, ranging from psychological to physical phenomena can be approximated, to varying degrees, by the normal distribution $N(\mu,\sigma^2)$. If $\ln(X)$ follows a normal distribution $N(\mu, \sigma^2)$, then X follows a *lognormal distribution* $\text{LN}(\mu, \sigma^2)$. While the mechanisms underlying these phenomena are often unknown, the use of the normal model can be theoretically justified by assuming that many small, independent effects are additively contributing to each observation (for a real value of x). The density function is given by (distribution function has no closed form expression):

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2}. \tag{3A.24}$$

The mean and variance are μ and σ^2, respectively, and their MLEs are:

$$\hat{\mu} = \bar{x}; \quad \hat{\sigma} = \sqrt{s^2} = \sqrt{\tfrac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{x})^2}. \tag{3A.25}$$

APPENDIX 3B: RANDOM VARIATE GENERATION

In this appendix, methods of generating random variates for major continuous distributions are explained in some detail. Distributions covered are exponential, Erlang, Beta, Weibull, normal, lognormal, and triangular distributions. There are available free libraries at http://www.codeproject.com/KB/recipes/Random.aspx.

3B.1 Exponential Random Variate

An exponential random variate $x \sim Expo(\theta)$ is generated by the inverse transform method. This method is based on the observation that "If $X \sim F(x)$ and $U \sim U(0,1)$ then we have $U = F^{-1}(X)$," as depicted in Fig. 3B.1.

Since the exponential distribution is $F(x) = 1 - e^{-x/\theta}$, the distribution function is easily inverted to obtain the following inverse-transform relationship:

$$u = 1 - e^{-x/\theta} \Rightarrow \quad e^{-x/\theta} = (1-u) \Rightarrow \quad x = -\theta * \ln(1-u).$$

Utilizing the fact that "$(1 - U) \sim U(0,1)$ if $U \sim U(0, 1)$," an exponential random variate X is generated from a standard uniform random variable U as follows:

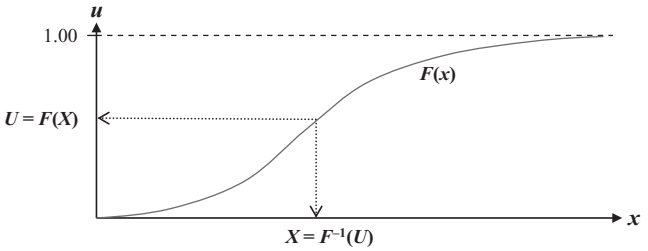


Fig. 3B.1. Inverse-transform method of random variate generation.

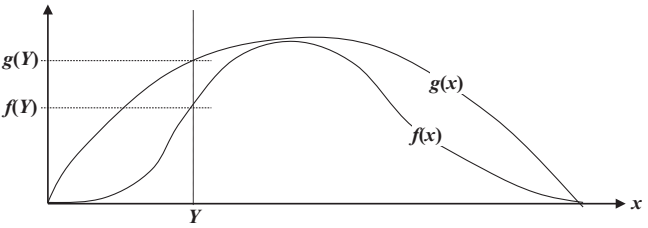


Fig. 3B.2. Acceptance–rejection method of random variate generation.

$$X = -\theta * \ln U. \tag{3B.1}$$

3B.2 Erlang Random Variate

An Erlang-k random variate $X \sim Erlang\ (k,\theta)$ with mean θ is defined as $X = \Sigma Y_i$ for $i = 1 \sim k$, where Y_i's are independent, identically distributed (IID) exponential random variables with mean θ / k. Thus, an Erlang-k random variate X can be generated as a sum of the k IID exponential random variates, which is known as the *convolution method* of random variate generation. Namely, an Erlang-k random variate is generated as a convolution of exponential random variates:

$$X = \sum_{i=1}^{k} Y_i = \sum_{i=1}^{k} \{-(\beta/k) * \ln(U_i)\} = -(\beta/k) * \ln\left(\prod_{i=1}^{k} U_i\right). \tag{3B.2}$$

3B.3 Beta Random Variate

As the direct methods (i.e., inverse-transform and convolution methods) are not applicable to Beta distribution $Beta(\alpha,\beta)$, the acceptance–rejection method is used in generating a beta random variate. In general, the acceptance–rejection method makes use of a majoring function $g(x)$ of the density function $f(x)$ for which we wish to generate random variates. The majoring function $g(x)$ is required to have the following properties: (1) $g(x) \geq f(x)$; (2) $m = \int g(x)\, dx < \infty$; (3) random variate $Y \sim g(x) / m$ is easily generated. Then the acceptance–rejection method of generating a random variate $X \sim f(x)$ may be summarized as follows (see Fig. 3B.2):

(1) Generate Y having density function $g(x) / m$;
(2) Generate $U \sim U(0,1)$, independent of Y ;
(3) If $U \leq f(Y) / g(Y)$ then return $X = Y$, else go back to step (1).

Applying the acceptance–rejection idea to generating a beta random variate is not a trivial problem, and there are quite a large number of beta random variate generation methods available in the literature (see for example, Cheng [1978], Schmeiser and Babu [1980]). Given below is the basic method of generating a standard beta random variate presented in Cheng [1978]. It is an acceptance–rejection method where the following functions are used as density function $f(y)$ and majorizing function $g(y)$:

Beta density function: $f(y) = y^{\alpha-1}/[B(\alpha,\beta)(1 + y^{\alpha+\beta})]$ *for* $y > 0$
Majorizing function: $g(y) = \lambda\mu\, y^{\lambda-1}(\mu + y^{\lambda})^{-2}$.

$Y \sim f(y)$ is known as the beta variate of the second kind $Beta2(\alpha,\beta)$. The following acceptance–rejection algorithm generates $Y \sim Beta2(\alpha,\beta)$ for $\alpha > 0$ and $\beta > 0$.

0. Initialization:
   – $A = \alpha + \beta$;
   – *If* $\min(\alpha, \beta) \leq 1$ *then* $B = \max(\alpha^{-1}, \beta^{-1})$ *else* $B = \sqrt{(A-2)/(2\alpha\beta - A)}$;
   – $C = \alpha + B^{-1;}$
1. Generate: $U_1 \sim U(0,1) \& U_2 \sim U(0,1)$.
2. Set: $V = B \log[U_1/(1 - U_1)]$; $W = \alpha \cdot e^{v}$.
3. If $\{A \cdot \log[A/(\beta + W)] + C \cdot V - \log 4\} < \log(U_1^2 U_2)$ then go to step (1); // rejection.
4. Return: $Y = W/(\beta + W)$.

Then, the standard beta variate $X \sim Beta(\alpha,\beta)$ with density function Eq. 3A.9 can be obtained from the beta random variate $Y \sim Beta2(\alpha,\beta)$ as follows [Cheng 1978]:

$$X = Y/(1+Y).$$

3B.4 Weibull Random Variate

As with the exponential distribution, the Weibull distribution $Weibull(\alpha,\beta)$ function is easily inverted to obtain the following inverse-transform relationship:

$$u = 1 - e^{-(t/\beta)^{\alpha}} \Rightarrow \quad x = \beta * \{-\ln(1-u)\}^{1/\alpha}.$$

Since $(1 - U) \sim U(0,1)$ if $U \sim U(0,1)$, a Weibull random variate X is generated from:

$$X = \beta * (-\ln U)^{1/\alpha}. \tag{3B.3}$$

3B.5 Normal and Lognormal Random Variates

Box and Muller [1958] developed a popular method for generating a normal random variate. It makes use of the relation that "X_1 and X_2 given by Eq. 3B.4 are IID *N(0,1)* if U_1 and U_2 are IID *U(0,1)*."

$$X_1 = (-2\ln U_1)^{1/2}\cos(2\pi U_2); \quad X_2 = (-2\ln U_1)^{1/2}\sin(2\pi U_2). \tag{3B.4}$$

Thus, once the parameters μ and σ^2 are estimated, normal random variates may be generated by using the method of Box and Muller [1958]:

1. Generate $U_1 \sim U(0,1)$ & $U_2 \sim U(0,1)$.
2. Compute $Z_1 = (-2 \ln U_1)^{1/2} \cos(2\pi U_2)$; $Z_2 = (-2\ln U_1)^{1/2} \sin(2\pi U_2)$.
3. Return $X_1 = \hat{\mu} + \hat{\sigma} \cdot Z_1$; $X_2 = \hat{\mu} + \hat{\sigma} \cdot Z_2$.

Since the normal random variates are generated in pairs, X_1 and X_2 are computed on each odd-numbered call to the generation function (but only X_1 is returned), and X_2 is returned on each subsequent even-numbered call. Let X be a normal random variate sampled from $N(m,s^2)$, then a lognormal random variate Y sampled from $LN(m,s^2)$ is obtained from $Y = e^X$.

3B.6 Triangular Random Variate

A double-triangle distribution *Triangular*(*a,b,c*)is defined by the lower bound value (*a*), upper bound value (*b*), and peak value (*c*). Depicted in Fig. 3B.3 are single-triangle density functions and random variates. A random variate of the "up-hill" triangle is generated as follows:

$$Y = a + (c - a)\sqrt{U}. \tag{3B-5}$$

And, the random variate of the "down-hill" triangle is generated as:

$$Y = c + (b - c)(1 - \sqrt{1 - U}). \tag{3B.6}$$

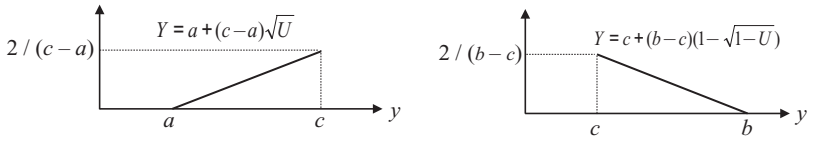


Fig. 3B.3. Single-triangle density functions and random variates.

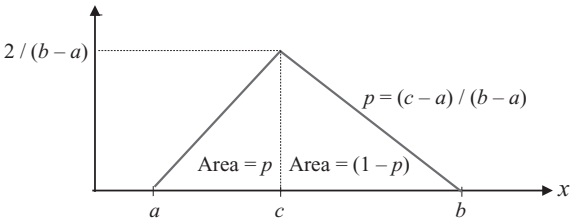


Fig. 3B.4. Double-triangle density function.

Then, the random variate of the double-triangle (see Fig. 3B.4) can be generated from the two single-triangle random variates by using the composition method. The composition method [Law 2007] calls for generating $U_1 \sim U(0,1)$ and checking whether $U_1 < p$. If so, generate an independent $U_2 \sim U(0,1)$ and return $X = a + (c - a) \times (U_2)^{1/2}$; Otherwise, return $X = c + (b - c) \times (1 - (1 - U_2)^{1/2})$.

CHAPTER 4

Introduction to Event-Based Modeling and Simulation

Little people discuss other people. Average people discuss events. Big people discuss ideas.

—R.E. Kalman

4.1 INTRODUCTION

This chapter is about the creative ideas for modeling and simulation of discrete-event systems using the concept of *event*. We often hear about events in the evening news, on the radio, and, more recently, through social media channels. If something that happened results in some meaningful changes, it is called an *event*. If we can identify the logical and temporal relationships between those events, we can understand our present situation better and may even be able to predict the future. Event-based modeling is a fundamental method of representing our knowledge about a discrete-event system, in which the dynamics of the system are represented by an event graph. An event graph is a network model of the logical and temporal relationships between the events. An event graph is a formal model that is easily implemented using the *next-event methodology* of simulation execution.

The purpose of this chapter is to provide a comprehensive coverage of event-based modeling and simulation (M&S) using the "ordinary" (i.e., non-parameterized) event graph. Event-based M&S involving a parameterized event graph is covered in the next chapter. After studying this chapter, you should be able to:

1. Provide an algebraic specification of an event graph model
2. Construct an event transition table for a given event graph model

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.

3. Build event graph models for single queue systems that have various features such as balking/reneging, batched service, assembly operation, and resource failure
4. Build event graph models of various types of tandem line systems such as time-constrained processing lines and conveyor-driven serial production lines
5. Build event graph models of special systems such as a flexible service shop, a car repair shop, a project management system, and an inline manufacturing cell
6. Simulate any event graph model using the commercial simulator SIGMA®
7. Develop your own computer simulation program for any event graph model

The remainder of the chapter is organized as follows. Section 4.2 reviews the integrated procedure for discrete-event simulation modeling using a single server system as an example. Then, the execution rules and formal specifications of an event graph model are presented in Section 4.3, followed by a section on event graph modeling templates. These modeling templates serve as building blocks for constructing larger models. Some real examples of event graph modeling are presented in Section 4.5, and the method of executing event graph models with SIGMA is explained in Section 4.6. Finally, a method for developing your own simulator is presented in the last section of this chapter.

4.2 MODELING AND SIMULATION OF A SINGLE SERVER SYSTEM

It was emphasized in Chapter 2 that the "integrated procedure for discrete-event system modeling" should be followed regardless of the modeling formalisms used. In this section, a method for applying the integrated procedure to the event-based modeling formalism is illustrated using the single server model presented in Chapter 2 (see Fig. 2.6). In event-based modeling formalism, a system is modeled by describing the changes that occur at different event times and the system dynamics are represented by an event graph.

4.2.1 Reference Modeling

The reference model of the single server system that was described in Chapter 2 is reproduced in Fig. 4.1(a). An informal description of the system dynamics in terms of the physical and logical modeling components is referred to as the *reference model*. The physical modeling components are classified into entities, active resources, and passive resources: Jobs are the entities; Job Creator and Machine are active resources; and Buffer is a passive resource. Among the

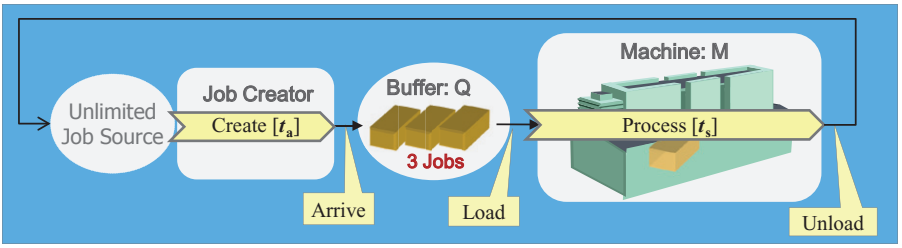


Fig. 4.1(a). Reference model of single server system (reproduced from Fig. 2.6).

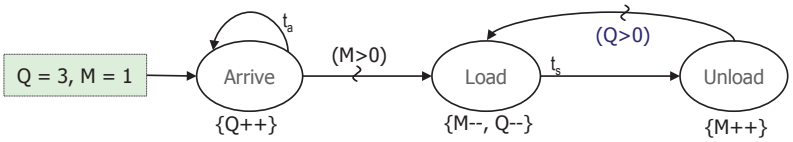


Fig. 4.1(b). Event graph of the single server system (reproduced from Fig. 2.1).

logical modeling components, (1) job creation and job processing are *activities*, (2) the machine status (M) and buffer status (Q) are *state variables*, and (3) Arrive, Load, and Unload are *events*.

The dynamics of the single server system can be described in terms of the modeling components (*Job Creator*, *Machine*, *Buffer*, *Job*, *create, process, arrive, load, unload*) as follows: the Job Creator creates a new Job for a duration of t_a minutes, which makes a new Job arrive every t_a minutes; the new Job is loaded on the Machine if it is idle, otherwise the Job is stored in the Buffer; the loaded Job is processed by the Machine for t_s minutes and then unloaded; the freed Machine loads another Job from the Buffer if it is not empty.

4.2.2 Formal Modeling

Reproduced in Fig. 4.1(b) is the event graph of the reference model in Fig. 4.1(a). A discrete-event system model described using a formal modeling tool is referred to as a *formal model*. It provides a complete description of the system in a concise and clear manner and can be executed with a well-defined simulation algorithm.

The event graph in Fig. 4.1(b) is interpreted as follows: (1) An Arrive event, which increases the job count by one (Q++), always schedules an Arrive event to occur after t_a and schedules a Load event if the machine is idle ($M \equiv 1$). (2) A Load event, which sets the machine to busy (M−−) and decreases the job count by one (Q−−), schedules an Unload event to occur after t_s. (3) The Unload event resets the machine to idle (M++) and schedules a Load event if the buffer is not empty ($Q > 0$).

TABLE 4.1. Event Transition Table for the Single Server System

No	Originating Event	State Change	Edge	Condition	Action	Delay	Destination Event
0	Initialize	Q = 3; M = 1;	1	True	schedule	0	Arrive
1	Arrive	Q++;	1	True	schedule	ta = Exp(5)	Arrive
			2	M > 0	schedule	0	Load
2	Load	M--; Q--;	1	True	schedule	ts = Exp(2)	Unload
3	Unload	M++;	1	Q > 0	schedule	0	Load

4.2.3 Model Execution

As mentioned in Chapter 2, the systematic method of specifying an event graph model is the use of an event transition table. This is a table in which the state change, edge condition, action-type (schedule/cancel), time delay, and destination events are specified for each event node. Table 4.1 shows an event transition table for the event graph in Fig. 4.1(b) with $t_a \sim \text{Exp}(5)$ and $t_s \sim \text{Exp}(2)$. Then, (as explained in Section 4.7.5), an event routine is obtained for each row of the event transition table so that the single server model is executed with your own event graph simulator. Alternatively, (as explained in Section 4.6.1), the event graph model in Fig. 4.1(b) is converted to a SIGMA-compatible event graph so that the single server model is executed with SIGMA.

4.3 EXECUTION RULES AND SPECIFICATIONS OF EVENT GRAPH MODELS

An event graph is a graphical formal model consisting of a set of event nodes and a set of directed edges. It provides a complete description of a discrete-event system (DES) in a concise and clear manner. Since the graphical model is to be interpreted by a human, its execution rules have to be described unambiguously. A graphical model can be specified in algebraic form (to be analyzed by a human logically) as well as in computer-readable form (to be executed on a computer).

4.3.1 Event Graph Execution Rules

There are two types of edges in an event graph: *scheduling edges* and *canceling edges*. In this subsection, the event vertex execution rules for these two types of edges are described [Schruben and Schruben 2001].

4.3.1.1 Execution of an Event with a Scheduling Edge Shown in Fig. 4.2 is an event vertex with a scheduling edge, which indicates that "whenever the

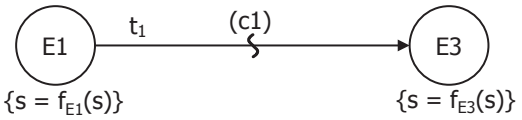


Fig. 4.2. Event graph with a scheduling edge.

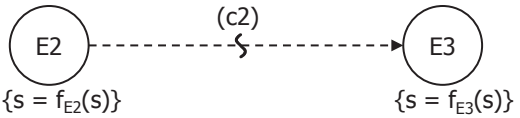


Fig. 4.3. Event graph with a canceling edge.

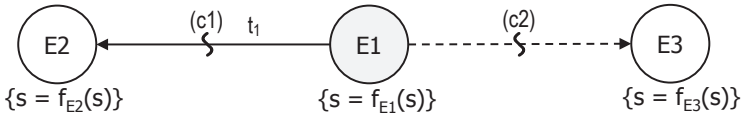


Fig. 4.4. Event graph with a scheduling edge and a canceling edge.

originating event E1 occurs, the state (s) changes according to $f_{E1}(s)$. Then, if the edge condition c1 is true, the destination event (E3) is scheduled to occur after a time delay of t_1." The scheduled execution time (τ) of E3 is obtained by adding the time delay (t_1) to the current execution time (*Clock*) of E1. Namely, $\tau = Clock + t_1$.

4.3.1.2 Execution of an Event Having a Canceling Edge Shown in Fig. 4.3 is an event graph with a canceling edge denoted by the dashed arrow. This event graph indicates that "whenever the originating event E2 occurs, the state 's' changes to $f_{E2}(s)$. Then, if the edge condition c2 is true, the scheduled event E3 is canceled immediately." If there is more than one event scheduled in the future event list, only the first one (i.e., the one with the smallest event time) may be canceled.

4.3.2 Tabular Specification of Event Graph Models

Let's consider an event vertex (E1) with a scheduling edge to event vertex E2 and a canceling edge to event vertex E3 as shown in Fig. 4.4. The event graph indicates that "whenever event E1 occurs, the state 's' changes to $f_{E1}(s)$. Then, if the edge condition c1 is true, event E2 is scheduled to occur after t_1; and if the edge condition c2 is true, event E3 is canceled immediately."

The event execution rules for the event graph of Fig. 4.4 can be specified in a tabular form called an *event transition table*. As shown in Table 4.2, each originating event is specified using five data fields: *state change*, *edge condition*, *action type* (schedule or cancel), *time delay*, and *destination event*.

TABLE 4.2. Event Transition Table for the Event Graph of Figure 4.4

Originating Event	State Change	Edge	Condition	Action	Delay	Destination Event
E1	$s = f_{E1}(s)$	1	c1	schedule	t_1	E2
		2	c2	cancel	0	E3

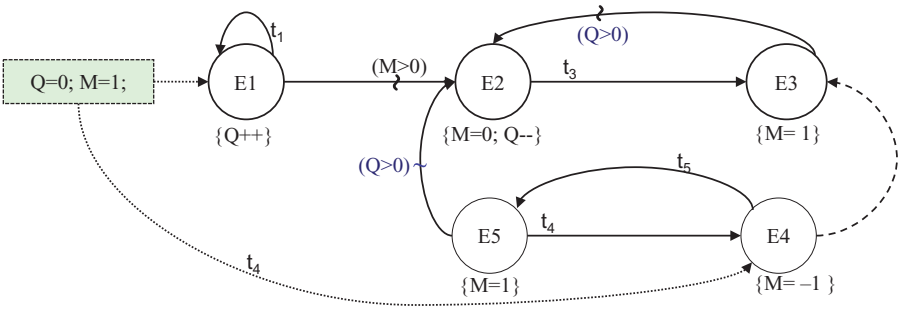


Fig. 4.5. Event graph model with an initialization box.

TABLE 4.3. Event Transition Table for the Event Graph in Figure 4.5

No	Originating Event	State Change	Edge	Condition	Action	Delay	Destination Event
0	Initialize	Q = 0;	1	True	schedule	0	E1
		M = 1;	2	True	schedule	t_4	E4
1	E1	Q++;	1	True	schedule	t_1	E1
			2	M > 0	schedule	0	E2
2	E2	M = 0;	1	True	schedule	t_3	E3
		Q—;					
3	E3	M = 1;	1	Q > 0	schedule	0	E2
4	E4	M = – 1;	1	True	schedule	t_5	E5
			2	True	cancel	0	E3
5	E5	M = 1;	1	Q > 0	schedule	0	E2
			2	True	schedule	t_4	E4

Figure 4.5 shows an event graph model with five event vertices. The rectangular box in the event graph denotes the initialize-event, where the state variables are initialized (Q = 0 and M = 1) and two events (E1 and E4) are scheduled. The first three events (E1–E3) constitute a single server system (compare with Fig. 4.1). An event transition table for the event graph model of Fig. 4.5 is given in Table 4.3.

4.3.3 Algebraic Specifications of an Event Graph Model

An event graph is a directed graph consisting of a set of event vertices (V), a set of directed edges (E), and a set of state variables (S). The edge set E represents the temporal and logical relationships between a pair of vertices. Associated with each vertex ($v \in V$) is a state transition function ($f_v \in F$); associated with each edge ($e \in E$) are edge conditions ($c_e \in C$), time delays ($d_e \in D$), and action types ($a_e \in A$; schedule or cancel). Thus, an event graph model M can be defined as a 7-tuple structure [Savage et al. 2005]:

M = <V, E, S, F, C, D, A>, where
- V = {v}: set of event vertices
- E = {e_{od}= (v_o, v_d)}: set of edges // v_o: originating event; v_d: destination event
- S = {s}: set of state variables
- F = {f_v: S → S ∀v ∈ V}: set of state transition functions associated with V
- C = {c_e: S→[0,1] ∀e ∈ E}: set of conditions associated with E
- D = {$d_e \in R_0^\infty$ ∀e ∈ E}: set of time delays associated with E
- A = {a_e ∈ [scheduling, canceling] ∀e ∈ E}: action type set

For example, the algebraic components of the event graph model in Fig. 4.1 (single server system) are as follows:

1. V = {v_1 = Arrive, v_2 = Load, v_3 = Unload}
2. E = {e_1 = (v_1, v_1), e_2 = (v_1, v_2), e_3 = (v_2, v_3), e_4 = (v_3, v_2)}
3. S = {Q, M}
4. F = {f_1: Q++; f_2: M––, Q––; f_3: M++}
5. C = (c_1: True; c_2: (M ≡ 1), c_3: True; c_4: (Q > 0)}
6. D = {d_1 = t_a; d_2 = 0; d_3 = t_s; t_4 = 0}
7. A = {a_1 = a_2 = a_3 = a_4 = scheduling}

Exercise 4.1. Specify the algebraic components of the event graph in Fig. 4.5.

4.4 EVENT GRAPH MODELING TEMPLATES

The modeling templates introduced here may be used as building blocks for modeling large systems. The single server event graph model shown in Fig. 4.1(b) is the baseline event graph model of a single queue system where a "table type" machine processes one job at a time, and the arriving jobs are stored in an infinite capacity buffer. This baseline model will be embellished and/or extended to cover more realistic and/or complex situations. Many of the modeling templates in the following are borrowed from Schruben and Schruben [2001].

4.4.1 Single Queue Models

By embellishing the baseline model of Fig. 4.1(b), a number of single queue models may be generated: a multi-server model, a limited waiting space model, a reneging queue model, a batched service model, an assembly-operation model, a resource priority model, and a resource failure model.

Exercise 4.2. Simplify the baseline model Fig. 4.1(b) by removing the Load event.

4.4.1.1 Flexible Multi-Server Model with Varying Number of Servers If a single queue system has more than one server, it is called a *multi-server system*. When the number of servers n is constant, the single server event graph model becomes a multi-server model if we set M = n (n > 1) in the *initialize box*.

Now, consider the case where the number of servers n(t) at time t varies over time, which we call a *flexible multi-server model*. Then, the event graph model of a flexible multi-server system can be represented as shown in Fig. 4.6, where M(t) denotes the number of idle machines at time t. It should be noted that the self-scheduling edge of the Load event vertex is introduced in order to manage the abrupt increase in the number of servers.

4.4.1.2 Limited Waiting Space Model (Balking Model) If the limited-capacity waiting space is full, an arriving job may not be able to enter the system and leave the system permanently, which is referred to as *balking*. Let c denote the capacity of the waiting space, and then the balking is modeled by introducing an Enter event as shown in Fig. 4.7.

4.4.1.3 Impatient Customer Model (Reneging Model) When customers arrive at a system that includes a queue and a server, they will enter the queue

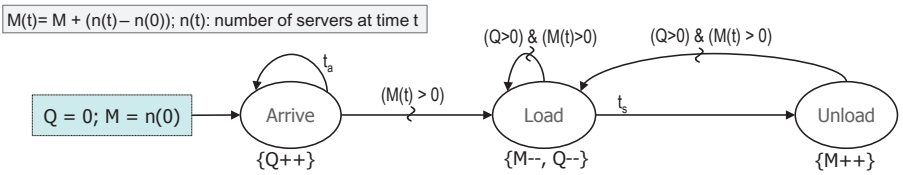


Fig. 4.6. Event graph model of a flexible multi-server system.

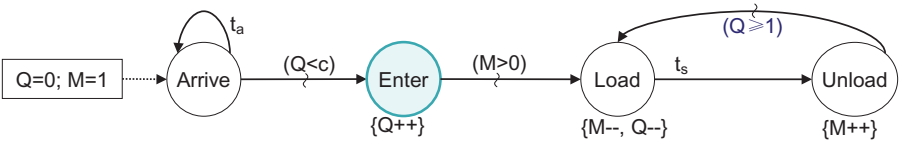


Fig. 4.7. Limited waiting space event graph model.

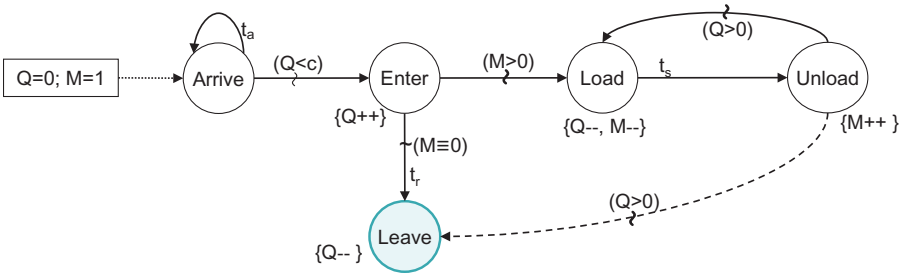


Fig. 4.8. Event graph model for reneging with balking.

if there is room. Once in the queue, they may choose to leave it if they have waited too long, which is often called *reneging* in queuing theory. As depicted in Fig. 4.8, a Leave event is introduced to manage the reneging situation: (1) every Enter event schedules a Leave event to occur after t_r minutes; (2) every Unload event cancels the oldest Leave event that has been scheduled, if there is one; and (3) every Leave event decreases Q by one (denoting the reneging customer).

Exercise 4.3. Construct a single server system event graph for reneging without balking.

4.4.1.4 Nonstationary (Fluctuating) Arrival Rates Model The method of generating exponential random variates with fluctuating arrival rates has been explained in Chapter 3 (refer to Fig. 3.4 of Section 3.4.3). Let $\lambda(t)$ denote the arrival rate at time t and it is bound by λ^*, then the thinning method of generating $X \sim \text{Exp}(1/\lambda(t))$ is as follows:

1. Set: $t = t_{i-1}$
2. Generate: $U_1 \sim U(0, 1)$ and $U_2 \sim U(0, 1)$
3. $D = -(1/\lambda^*) \ln(U_1)$; //exponential random variable with $\theta = 1/\lambda^*$
4. $t = t + D$
5. If $U_2 \leq \lambda(t)/\lambda^*$, then return $t_i = t$, else go back to step 2

Shown in Fig. 4.9 is an event graph model of a single server system subject to fluctuating arrival rates. The next Arrive event is scheduled to occur after an inter-arrival time t_a with a bounding arrival rate λ^*, and the Arrive event will schedule an Enter event only when the thinning test is passed (i.e., $U < \lambda(t)/\lambda^*$).

4.4.1.5 Batched Service Models Batched service occurs when a batch of jobs is processed simultaneously. In general, there is a maximum number (b) and a minimum number (a) of jobs that can be processed at one time, which is denoted as $a \leq J \leq b$, where J is the actual number of jobs in a batched

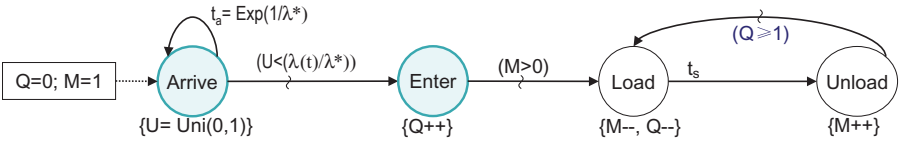


Fig. 4.9. Event graph model for fluctuating arrival rates (nonhomogeneous arrivals).

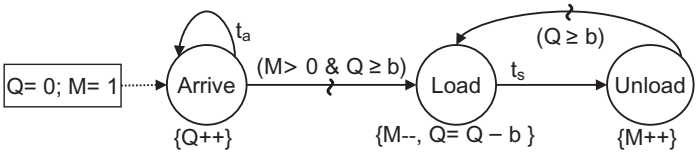


Fig. 4.10. Full batched service event graph model.

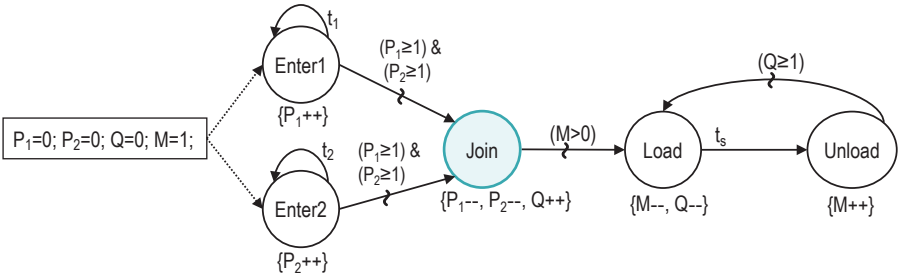


Fig. 4.11. Event graph model for an assembly operation.

service. If a = b, it is a full batched service; if a < b, then it is a partial batched service.

An event graph model for a full batched service using a single machine is given in Fig. 4.10. Notice that the baseline model of Fig. 4.1(b) is a special case of the full batched service model of Fig. 4.10, with b = 1.

Exercise 4.4. Revise the event graph presented in Fig. 4.10 to make it a partial batched service model.

4.4.1.6 Assembly Operation Model A type j part for j = 1, 2 arrives at the system every t_j minutes, and a pair of parts, one from each type, are assembled together using a machine. Let Q denote number of part pairs and P_j denote number of (unpaired) parts of type j; then, by introducing a Join event, the assembly operation is modeled as an event graph as shown in Fig. 4.11.

At this point, it is instructive to comment on the edge conditions in Fig. 4.11. Let C1 and C2 denote edge conditions of the Enter1→Join edge and Enter2→Join edge, respectively. It is specified in Fig. 4.11 that C1 = ($P_1 \geq 1$) & ($P_2 \geq 1$), and C2 = ($P_1 \geq 1$) & ($P_2 \geq 1$), which is valid. However, it can be found

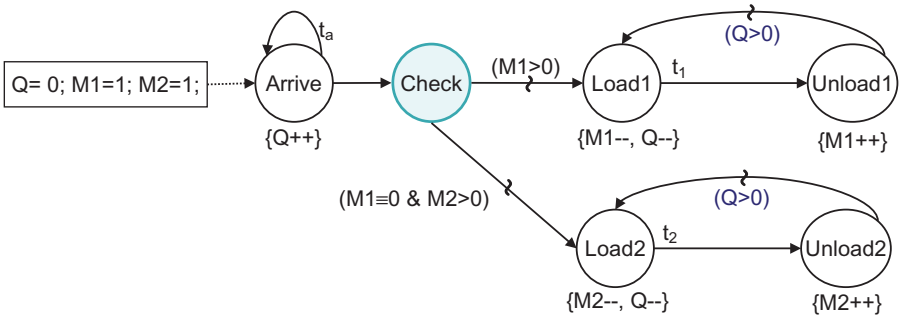


Fig. 4.12. Event graph model of a two-server system with resource priority.

that C1 = (P2 ≥ 1) and C2 = (P1 ≥ 1) are also valid. Similarly, the edge condition for the Join → Load edge is (M > 0) & (Q ≥ 1), but in Fig. 4.11, it is specified as (M > 0) because (Q ≥ 1) is always true at the Join event.

Exercise 4.5. Modify the event graph Fig. 4.11 to assemble three type 1 parts and four type 2 parts.

4.4.1.7 Resource Priority Model When there are two servers with different priorities, an arriving customer is served by the high priority server (M1) if it is free. The customer is directed to the low priority server (M2) only when M1 is busy, which is handled by a Check event. An event graph model for a "two server system with priority" is given in Fig. 4.12.

The above resource priority model has a common queue for both servers. There is a situation where each server has its own queue and the arriving customers join the smaller queue, which is often referred to as a queue length balanced line.

Exercise 4.6. Modify the event graph Fig. 4.12 to make it a queue length balanced line.

4.4.1.8 Resource Failure Models A single server system with resource failure may be modeled by introducing a Fail event with an interfailure time t_f and a Repair event with a repair time t_r. An event graph model of the single server system with failure is given in Fig. 4.13 (it is the same as Fig. 4.5). The Fail event will cancel a scheduled Unload event (if there is one) and schedule a Repair event to occur after t_r minutes. The active resources in the resource failure system are the machine and repairman, while the entities are the jobs and failures. This model assumes that a server may fail even when it is idle and that the job whose processing is interrupted by the failure is discarded without reprocessing.

Exercise 4.7. Modify the event graph of Fig. 4.13 so that the job that was interrupted by the failure is reprocessed.

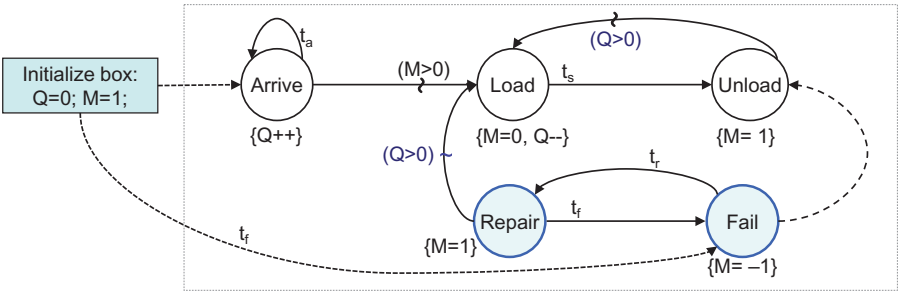


Fig. 4.13. Event graph of single server system with failure.

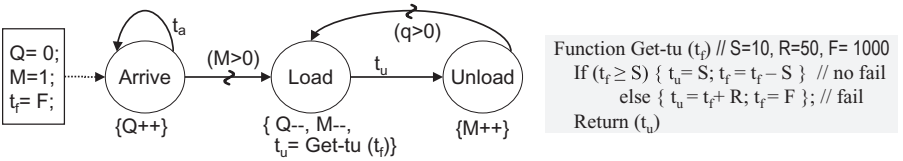


Fig. 4.14. Modeling of machine failure without an event cancellation.

Now consider the case where the interfailure time is effective only when the server is busy (i.e., idle periods have no effect on the failure) and the interrupted job is discarded. If all time data are deterministic such that service time S = 10, repair time R = 50, and interfailure time F = 1000, then the "single server system with failure" can be modeled without an event cancellation as shown in Fig. 4.14.

In the event graph model, t_f is the remaining time to failure; t_u is the actual time to unload. In general, an event graph model with an event cancellation can be transformed into a model without an event cancellation [Savage and Schruben 1995]. Notice in the model that the remaining time to failure (t_f) is also a state variable that is updated every time the machine completes a cycle. Issues related to modeling resource failures are discussed further in Schruben and Schruben [2001]. The actual time to unload (t_u) is computed in the Get-tu () function as follows: When the remaining time to failure is larger than the service time ($t_f \geq S$), the scheduled Unload event will be performed as scheduled. In this case, the actual time to unload equals to the service time ($t_u = S$), and the t_f is decreased by S. Otherwise ($t_f < S$), t_u becomes $t_f + R$, and t_f is set to the interfailure time F.

4.4.2 Tandem Line Models

The event graph of the two-stage tandem server defined previously in Chapter 2 (Fig. 2.15) is reproduced in Fig. 4.15, which serves as the baseline tandem line model in this chapter. The baseline model is obtained by appending a server model to the single server model of Fig. 4.1(b). From this baseline event graph model, a number of tandem line models may be generated: (1) limited

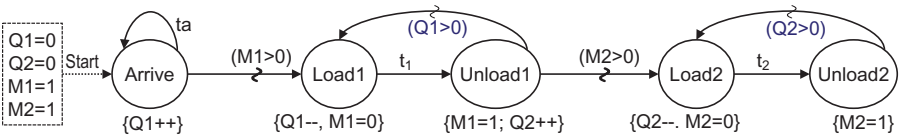


Fig. 4.15. Baseline event graph model of tandem line system.

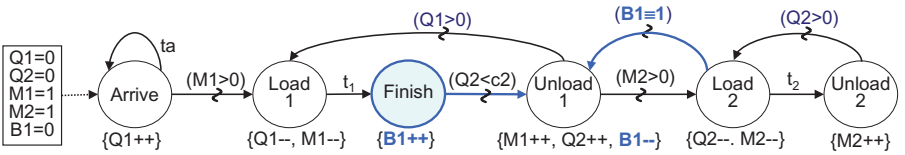


Fig. 4.16. Event graph model of a limited buffer tandem line (blocking).

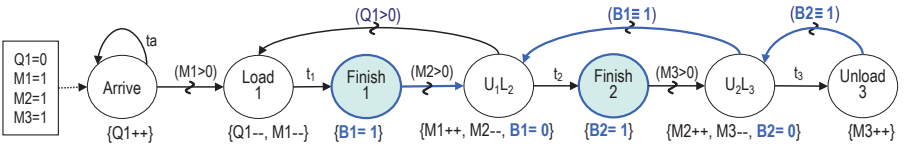


Fig. 4.17. Event graph model of three-stage buffer-less tandem line (blocking).

buffer tandem line model, (2) buffer-less tandem line model, and (3) time-constrained processing model.

4.4.2.1 Limited Buffer Tandem Line Model If the buffer after a machine has a limited capacity, a finished job may not be unloaded from the machine when the buffer is full. This situation is referred to as *blocking*. Figure 4.16 is an event graph model for a two-stage tandem line with a buffer of capacity c2. A Finish event and a blocking variable (B1) are introduced to control the blocking of M1: (1) the Finish event sets the blocking variable to true (B1 = 1) and schedules an Unload-1 event if the buffer is not full (Q2 < c2); (2) the Unload-1 event sets the blocking variable to false (B = 0); and (3) the Load-2 event schedules an Unload-1 event if the blocking variable is true (B1 ≡ 1).

4.4.2.2 Buffer-less Tandem Line Model If the buffer capacity is zero (c = 0) in the limited buffer tandem line model, the adjacent machines are tightly coupled such that unloading from machine-j becomes loading to machine-j+1, which is called an "U_jL_{j+1}" event. An event graph model of three-stage buffer-less tandem line is given in Fig. 4.17.

4.4.2.3 Time-Constrained Processing Tandem Line Model A processing situation where a job that had been processed on a machine (M1) must start the next processing step on the next machine (M2) within a time-out limit (t_o) is called a *time-constrained processing*. Otherwise, the time-out job is discarded

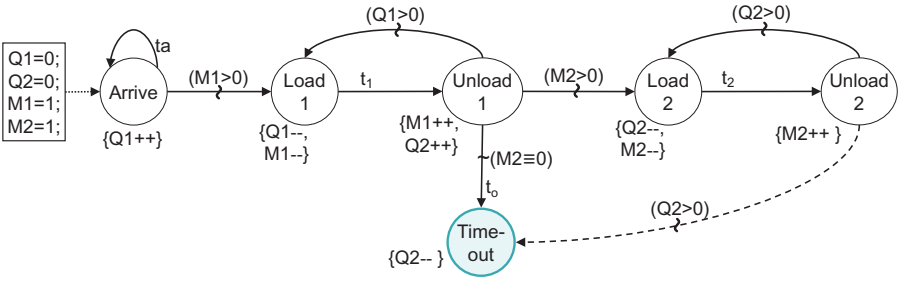


Fig. 4.18. Event graph model for time-constrained processing with a discarding policy.

(which is the same as the reneging situation of Fig. 4.8) or sent back to M1 for reprocessing. In Fig. 4.18, a Time-out event is introduced to manage the time constrained processing under a discarding policy: (1) every Unload-1 event schedules a Time-out event to occur after t_o minutes if M2 is 0; (2) every Unload-2 event cancels the oldest Time-out event that had been scheduled; (3) a Time-out event decrements Q2 (i.e., discard a time-out job).

Exercise 4.8. Modify the event graph Fig. 4.18 so that the time-out jobs are reprocessed.

4.5 EVENT GRAPH MODELING EXAMPLES

System modeling is an art that cannot be mastered without practice. In order for you to become familiar with event graph modeling, some examples of event graph modeling are provided in this section. The event graph application areas that will be covered are a simple service shop with fluctuating arrival rates, a car repair shop, a project management application, a conveyor-driven serial assembly line, and an inline manufacturing cell.

4.5.1 Flexible Multi-Server System with Fluctuating Arrival Rates

A salient feature of a service system is that the customer arrival rates fluctuate over time. In a flexible multi-server system, the resource levels change over time in order to cope with the changes in arrival rates. Let $\lambda(t)$ and n(t) denote arrival rates and the number of servers at time t, respectively; then, by combining the event graph templates in Figs. 4.6 and 4.9, the event graph model of a flexible multi-server system with fluctuating arrival rates can be constructed as in Fig. 4.19.

4.5.2 Car Repair Shop

The entities of a car repair shop are the cars brought in for repair and the resources are the technicians and repairmen. There are three types of activities:

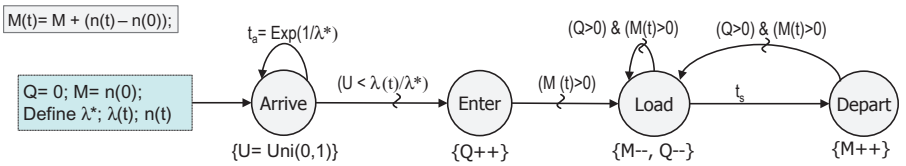


Fig. 4.19. Event graph of flexible multi-server system with fluctuating arrival rates.

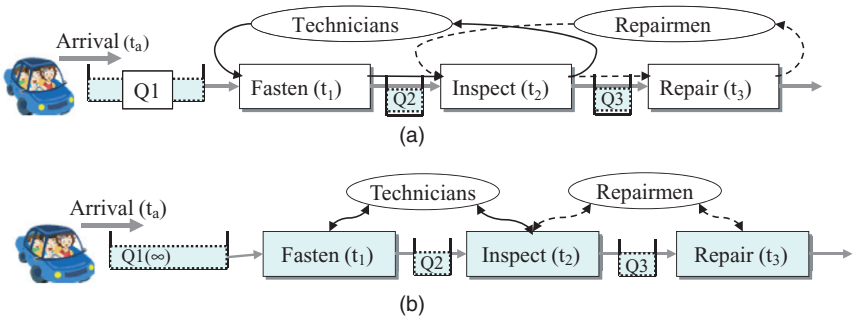


Fig. 4.20. (a) Reference model of car repair shop under a *same operator* policy; (b) reference model of car repair shop without a *same operator* policy.

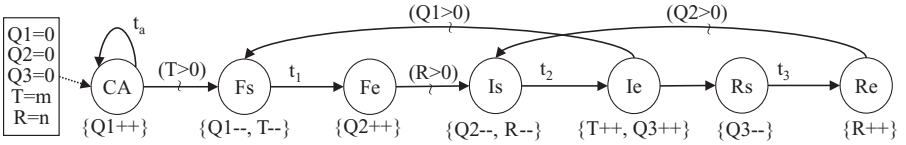


Fig. 4.21. Event graph of a car repair shop under the *same operator* policy.

Fasten, Inspect, and Repair, with processing times t_1, t_2, and t_3, respectively. The fasten operation is performed by a technician; the inspection operation requires both a technician and a repairman; and the repair operation is handled by a repairman.

A reference model of a simple car repair shop under a *same operator* policy is provided in Fig. 4.20(a) where a car is fastened and inspected by the same technician and is inspected and repaired by the same repairman. Thus, a technician stands by after fastening a car until a repairman is available. If the same operator restriction is removed, the reference model would change to that shown in Fig. 4.20(b).

Figure 4.21 is an event graph model of the car repair shop under the *same operator* policy. There are m free technicians and n free repairmen in the system. The state variables are the number of waiting cars (Q1, Q2, Q3), number of free technicians (T), and number of free repairmen (R). All start points and end points of the activities are regarded as events: car arrival (CA),

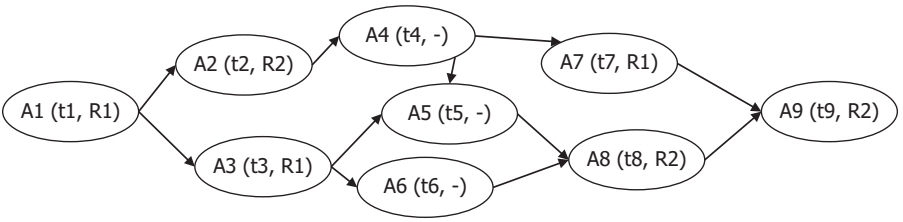


Fig. 4.22. Activity-on-node PERT diagram with finish–start precedence.

fastening start (F_s), fastening end (F_e), inspect start (I_s), inspect end (I_e), repair start (R_s), and repair end (R_e).

Exercise 4.9. Build an event graph for the reference model of Fig. 4.20(b).

4.5.3 Project Management Modeling

In project management, the precedence relationships among activities are represented as a directed graph of activities known as a PERT (program evaluation and review technique) diagram [Duncan 1996]. Shown in Fig. 4.22 is an *activity-on-node (AON) PERT diagram* involving nine activities (A1–A9) that serves as a reference model of the project management problem. A node denotes an activity, and an edge represents the *finish–start* precedence relationship between the two nodes (i.e., the first activity must be finished before starting the second activity). The activity ID (A_j), activity time (t_j), and the critical resource (R_k) required for each activity (noncritical resources are not explicitly identified) are indicated in each node. For example, the resource R_1 manages activities A1, A3, and A7.

Let's build an event graph model for the AON PERT diagram in Fig. 4.22 disregarding the resources. (This example was adopted from Schruben and Schruben [2001].) The first step is to identify the state variables of the PERT diagram. The state variables are $\{n_j \text{ for } j = 1\sim9\}$, where n_j denotes the number of unfinished precedent activities of activity A_j. Note that the activity A_j may be started only when n_j is 0. The start point and finish point of activity A_j are defined as the start event S_j and finish event F_j, respectively. An event graph model of the AON PERT diagram without resource constraints is given in Fig. 4.23.

Initially, the state variables have the following values: $n_1 = 0$, $n_2 = 1$, $n_3 = 1$, $n_4 = 1$, $n_5 = 2$, $n_6 = 1$, $n_7 = 1$, $n_8 = 2$, and $n_9 = 2$. The finish event F_1 will decrement its succeeding activity counts (n_2—, n_3—) and schedule the start events (S_2 and S_3) of its succeeding activities because $n_2 \equiv 0$ and $n_3 \equiv 0$. For example, the start event S_3 may start when $n_3 = 0$, and it will schedule the finish event F_3 to occur after t_3 time units. The succeeding activities of A3 are A5 and A6. Thus, n_5 and n_6 are decremented by F_3, and so on.

In general there are four types of precedence constraints: finish–start, finish–finish, start–start, and start–finish precedence constraints. For example,

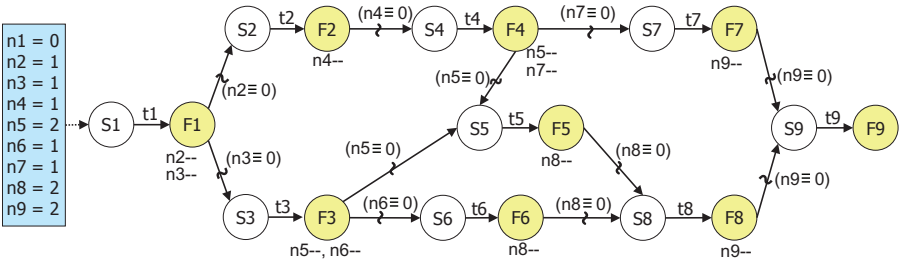


Fig. 4.23. Event graph of the PERT diagram of Fig. 4.22 without resource constraints.

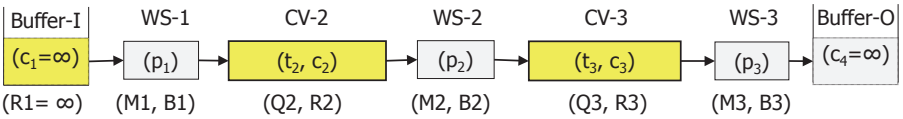


Fig. 4.24. Reference model of a three-stage conveyor-driven serial production line.

an event graph of a PERT diagram with a start-to-start precedence may be constructed similarly. However, the event graph of Fig. 4.23 may be simplified somewhat by eliminating all start events $\{S_j\}$. Methods of simplifying event graph models are elaborated in Schruben [1983].

In general, it is possible to construct an event graph model for a resource constrained PERT diagram by introducing a resource dispatch event for each resource together with additional state variables, but it may become quite complicated when the resource-activity relationships are not simple. Resource constrained PERT diagrams may be modeled more easily using an activity cycle diagram, as will be described further in Chapter 6.

Exercise 4.10. Simplify the PERT event graph Fig. 4.23 by removing the state variables $\{n_j\}$ whose initial value is 1.

4.5.4 Conveyor-Driven Serial Line

Consider a three-stage serial production line shown in Fig. 4.24 consisting of three *workstations (WS)* connected by *conveyors (CV)*. The base parts (jobs) stored in the input buffer (Buffer-I) are moved along the line, and the subparts are assembled into the base part at each workstation. The assembled base parts (i.e., products) are stored in the output buffer (Buffer-O). The entities in the system are the base parts, and the resources are the WS, CV, and Buffer. The activities are the production operations at the WS and the transport operations by the conveyors.

Each workstation WS_j for j = 1-n is specified by its production operation time p_j, while each conveyer CV_j for j = 2-n is specified by its capacity c_j and transport time t_j. The capacities of Buffer-I and Buffer-O, respectively

designated as c_1 and c_{n+1}, are assumed to be unlimited. Thus, the characteristics of the serial production line are defined using the following values:

p_j = processing time at WS_j;
t_j = transport time of the conveyor-j feeding WS_j; and
c_j = capacity of CV_j ($c_1 = c_{n+1} = \infty$).

The state of each workstation (WS_j) is specified by two state variables: M_j (free or busy) and B_j (blocked or not); that of CV_j is specified by Q_j (total number of jobs on a CV_j) and R_j (number of "ready" jobs that have been moved to the WS). Thus, the state variables of the serial production line are as follows:

M_j = workstation-j status (1: free, 0: busy);
B_j = blocking of workstation-j (1: blocked);
Q_j = total number of jobs at CV_j; and
R_j = number of jobs ready at CV_j (i.e., jobs that have been transported).

Since a conveyor acts as a limited buffer, each workstation in the serial line is modeled as a machine in the limited buffer tandem line of Fig. 4.16. Thus, there are three types of events associated with WS_j: Load (L_j), Finish (F_j), and Unload (U_j). Let T_j denote the Transport (to the end of conveyor) event of CV_j. Then, the operation cycle of CV_j is defined by U_{j-1} (unload from work station j–1), T_j, and L_j. Thus, the event graph model of the three-stage conveyor-driven serial line is as shown in Fig. 4.25, where T_j denotes the Transport event at CV_j.

4.5.5 Inline-Type Manufacturing Cell Modeling

An electronics fabrication factory (abbreviated as Fab) is a job shop in which a job goes through a number of processing steps according to its routing sequence. In a modern electronics Fab, unlike a mechanical job shop where a mechanical part is processed individually at table machines, the jobs are processed in batches mostly in inline cells.

Depicted in Fig. 4.26 is a photolithography cell commonly found in a modern TFT–LCD (thin film transistor–liquid crystal display) panel Fab. For brevity,

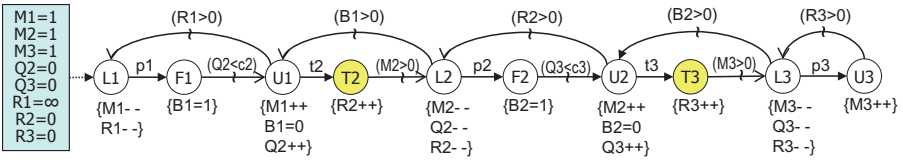


Fig. 4.25. Event graph model of a three-stage conveyor-driven serial line.

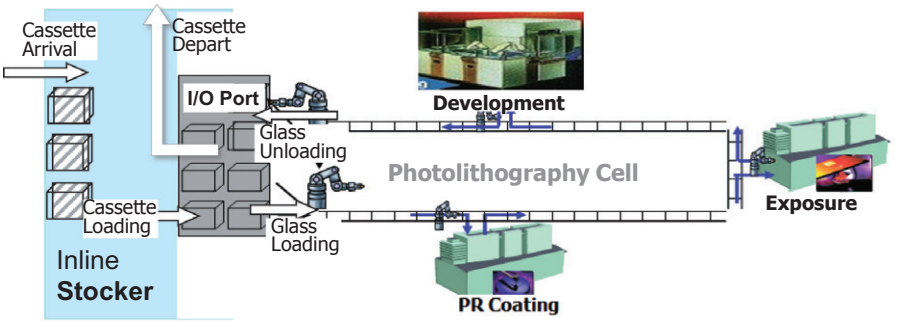


Fig. 4.26. An inline cell for photolithography process.

the inline-type manufacturing cell will be simply called an *inline cell*. This modeling case is quite significant as it is taken from a real-life simulation project in which a simulation-based planning and scheduling system was built for a TFT–LCD Fab [Park et al. 2008], and it considers important issues that arise in these Fabs. The issues addressed in the case study are the "divide and conquer" method of building event graph models and model simplification.

In the photolithography cell in Fig. 4.26, the jobs are glasses that go through photoresist (PR) coating, exposure, and development processes in the cell. The jobs are handled in batches with each batch (or lot) stored in a cassette. The arriving cassettes that are stored in the inline stocker are moved into the I/O port, which is called the cassette loading (CL) operation. The glasses are loaded inline using a loading robot, with one glass being loaded at every takt time (τ). It takes a flow time (π) for a loaded glass to reach the end of the cell where it is unloaded into the unloading cassette located at the I/O port. The unloading cassette departs when it is filled with finished glasses. In unloading the glasses, only one unloading cassette is used at a time.

4.5.5.1 Reference Model Figure 4.27 is the reference model of the cell given in Fig. 4.26. The physical components of the cell are the Stocker, I/O port, Robot, and Inline. The state variables in the model are Q (number of arriving cassettes in the Stocker), B (number of arriving cassettes in the I/O port), E (number of empty shelves in the I/O port), and R (status of Robot). The activities in the cell are (1) cassette arrival, (2) cassette loading, (3) glass loading, (4) glass unloading, and (5) cassette departure.

The capacity of the Stocker is assumed to be unlimited. The I/O port has a finite number of shelves for storing cassettes (arriving, unloading, and empty cassettes). Let N be the number of shelves of the I/O port. If all shelves are empty at the beginning, we have E = N and B = 0. If an arriving cassette is loaded onto I/O port, the state variables are updated to E = E–1 and B = B+1. If all glasses in an arriving cassette are loaded into the Inline, the cassette becomes empty. If the finished glasses are unloaded into an empty cassette, it

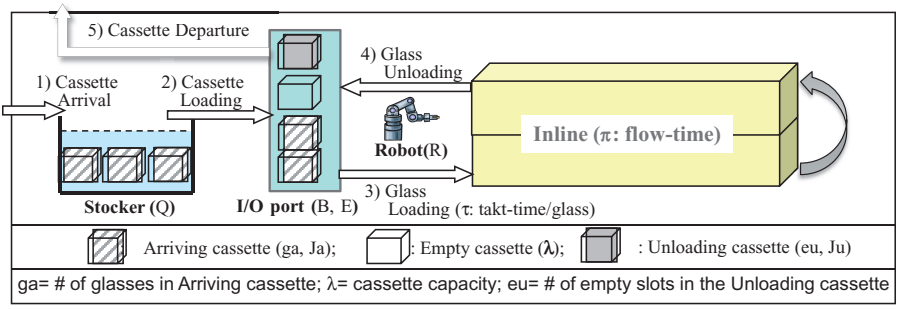


Fig. 4.27. Reference model of the inline-type manufacturing cell in Fig. 4.26.

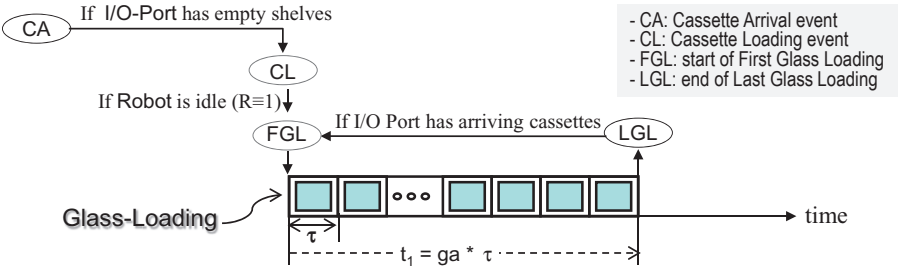


Fig. 4.28. Reference model of the loading region.

becomes an unloading cassette. When the unloading cassette departs from the I/O port, the number of empty shelves is increased by one (E = E+1).

Only the glasses with the same job type are stored in a cassette, which becomes the job type of the cassette. The job type of an arriving cassette is denoted by Ja and the number of glasses in the arriving cassette is ga $\leq \lambda$, where λ is the cassette capacity. The finished glasses are unloaded into the empty slots of the unloading cassette whose job type is denoted by Ju. The number of empty slots in the unloading cassette is denoted by eu $\leq \lambda$. Thus, the number of glasses in the unloading cassette is equal to λ – eu. The eu (number of empty slots) and Ju (job type) of the unloading cassette are also regarded as state variables. Recall that there is only one unloading cassette at a time.

In general, it is convenient to divide the reference model into regions, build a submodel for each region, and join the submodels in order to obtain the entire event graph model of the reference model. Now, we will divide our reference model into three regions: the Loading region, Processing region, and Unloading region.

4.5.5.2 Loading Region Modeling Figure 4.28 is a reference model of the loading region (Stocker + I/O port + Robot). The time required for processing all glasses in an arriving cassette is $t_1 = ga * \tau$, where τ is the takt time of a

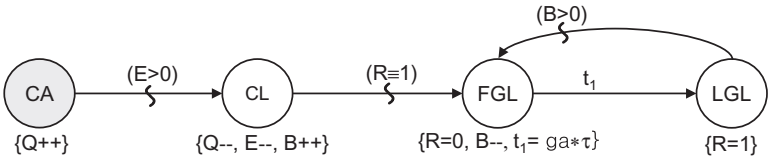


Fig. 4.29. Event graph of the loading region.

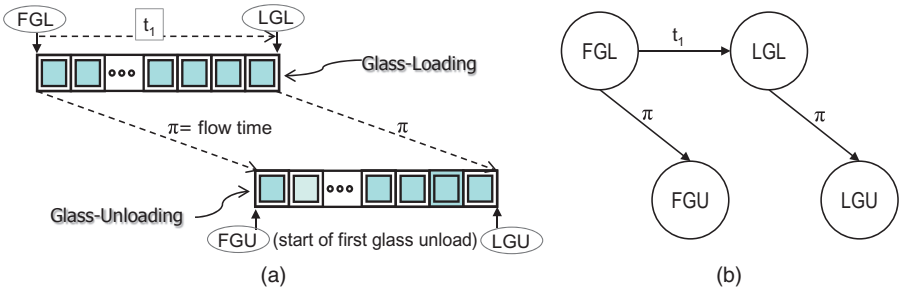


Fig. 4.30. (a) Reference model and (b) event graph model of the processing region.

glass loading. The events involved in the loading region model are CA (end of cassette arrival), CL (end of cassette loading), FGL (start of first glass loading), and LGL (end of last glass loading). The relationships among the events are as follows: (1) when a cassette arrives, it is loaded if the I/O port has space; (2) the first glass of the cassette is loaded if the Robot is idle; (3) the last glass is loaded after t_1 time units since the first glass is loaded; and (4) after the last glass loading, the first glass of the next cassette is loaded if there is an arriving cassette in the I/O port.

The system dynamics of the loading region described in the reference model may be formally specified as an event graph model in terms of the state variables. An event graph model of the loading region is given in Fig. 4.29, where the state variables are Q (number of arriving cassettes in Stocker), B (number of arriving cassettes in I/O port), E (number of empty shelves in I/O port), and R (status of Robot with R = 1 initially).

4.5.5.3 Processing Region Modeling A reference model of the processing region and its event graph model are shown in Fig. 4.30(a) and (b), respectively. The events at the start of the Inline are the FGL and LGL that were defined in the In-port region (see Fig. 4.29), and the events at the end of the Inline are FGU (start of the first glass unloading) and LGU (end of the last glass unloading). The FGU event is scheduled by the FGL event to occur after the flow time (π), and the LGU is scheduled by the LGL after π.

4.5.5.4 Unloading Region Modeling Figure 4.31 shows the reference model of the unloading region. The events involved are the FGU, LGU, and

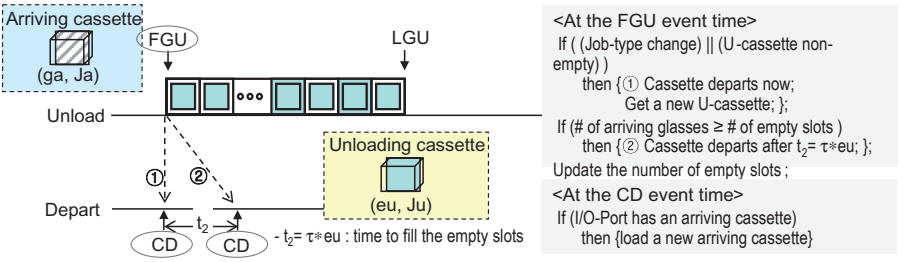


Fig. 4.31. Reference model of the unloading region.

CD (end of cassette departure). An arriving cassette is identified by its job type (Ja) and its number of glasses (ga), while the unloading cassette is specified by its job type (Ju) and the number of its empty slots (eu). An important restriction in glass unloading is that all glasses in the unloading cassette have the same job type. Namely, the unloading cassette departs either when it is full or when there is a job type change in the unloaded glasses.

At the time of the first glass unloading (FGU) event, a cassette departure (CD) event is scheduled based on the attribute values of the arriving cassette (Ja and ga) and state variables (Ju and eu). Depending on the values of these attributes and state variables, the following actions are taken at the FGU event time: (JTC and UCNE are Boolean variables denoting Job Type Change and Unloading Cassette Not Empty, respectively).

1. If there is a job type change (JTC = (Ja ≠ Ju)) or the unloading cassette is not empty [UCNE = (eu < λ)], then schedule a CD event to occur now and obtain a new unloading cassette (eu = λ; Ju = Ja).
2. If (ga ≥ eu), then schedule a CD event to occur after a time delay of $t_2 = \tau * eu$.
3. Update the state variables: Ju = Ja. If ga ≥ eu, then eu = λ − (ga − eu), else eu = eu − ga.

At the time of the cassette departure (CD) event, a cassette loading (CL) event is scheduled if there is an arriving cassette in the I/O port. Reflecting the above state transition relationships, the resulting event graph is as shown in Fig. 4.32.

In practice, the restriction that all glasses in an unloading cassette must have the same job type (requiring partially filled unloading cassettes to be removed when there is a job type change) may be relaxed in order to reduce the model complexity. Then, the cassettes are fully loaded during handling (ga = λ). The event graph of the unloading region may be simplified to that shown in Fig. 4.33.

4.5.5.5 Event Graph Model of Entire Cell By combining the three event graphs in Figs. 4.29, 4.30, and 4.33, we can obtain the event graph for the entire

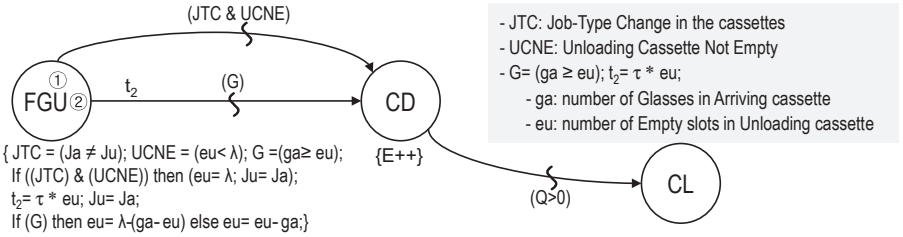


Fig. 4.32. Event graph model of the unloading region.

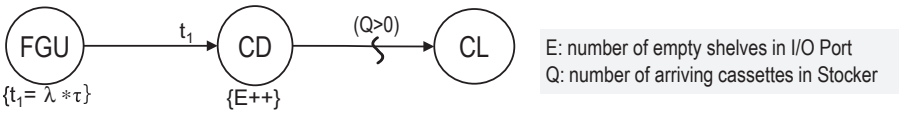


Fig. 4.33. Simplified event graph model of the unloading region.

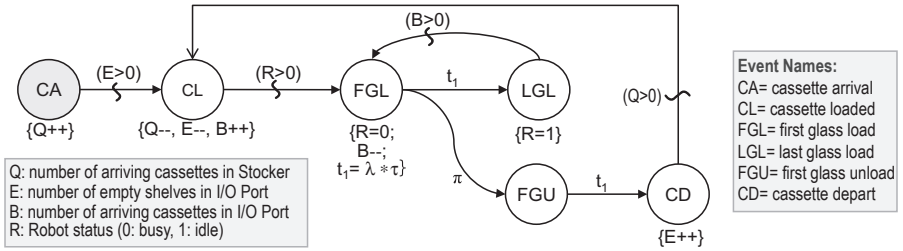


Fig. 4.34. Combined event graph model of the inline cell.

inline cell as shown in Fig. 4.34 if we assume that the arriving cassettes are fully loaded (ga ≡ λ). There are six events in the model: CA (cassette arrival), CL (cassette loading), FGL (first glass loading), LGL (last glass loading), FGU (first glass unloading), and CD (cassette departure). An event transition table for the event graph is given in Table 4.4.

The state variables in the inline cell model are Q (number of arriving cassettes in Stocker), B (number of arriving cassettes in I/O port), E (number of empty shelves in I/O port), and R (status of Robot). The design variables of the system are λ (cassette capacity), τ (takt time), and π (flow time).

Exercise 4.11. Simplify the inline cell event graph model in Fig. 4.34 by removing the FGU event vertex.

4.6 EXECUTION OF EVENT GRAPH MODELS WITH SIGMA

The purpose of this section is to introduce the SIGMA software. The overall procedure for building a SIGMA program for simulation is as follows. A brief

TABLE 4.4. Event Transition Table for the Event Graph of Figure 4.34

No	Originating Event	State Change	Edge	Condition	Delay	Destination Event
1	CA	Q = Q + 1;	1	E > 0	0	CL
2	CL	Q = Q − 1; E = E − 1; B = B + 1;	1	R > 0	0	FGL
3	FGL	R = 0; B = B − 1;	1	True	t_1	LGL
		$t_1 = \lambda * \tau$	2	True	π	FGU
4	LGL	R = 1;	1	B > 0	0	FGL
5	FGU		1	True	t_1	CD
6	CD	E = E + 1;	1	Q > 0	0	CL

SIGMA tutorial as well as the two SIGMA models discussed in this section may be found in the official website of this book (http://VMS-technology.com/Book/Sigma).

1. Create a SIGMA-generated event graph consisting of vertices and edges.
2. Declare variables: all variables are declared in a dialog box.
3. Define the Run vertex: state variables are listed as parameters in the dialog box.
4. Define the Event vertices: the state changes and parameter variables of each event vertex are described at each Edit vertex dialog box.
5. Define the Edges: the time delay, edge condition, and attribute (parameter value) of each edge are defined in each Edit edge dialog box.
6. Specify Run Options: various run options (end of simulation condition, trace variables, etc.) are specified and the state variables are initialized in the Run options dialog box.

4.6.1 Simulation of a Single Server System with SIGMA

The above six-step procedure will be illustrated using the single server system presented in Fig. 4.1. In order to become familiar with the basic functions of SIGMA, you are advised to follow the steps one by one.

4.6.1.1 Creating a SIGMA-Generated Event Graph SIGMA has a well-defined syntax system. For example, the initialize box of an ordinary event graph is treated as the #1 event vertex (named Run or Init); an exponential random variate with a mean of 1 is denoted as ERL{1}; and the operators ++/— are not allowed. Thus, using the SIGMA syntax, a SIGMA-compatible event graph may be obtained from the "neutral" event graph of Fig. 4.1(b) as shown in Fig. 4.35.

In this book, an event graph generated by SIGMA is called a SIGMA-generated event graph. The first step in building a SIGMA simulation program

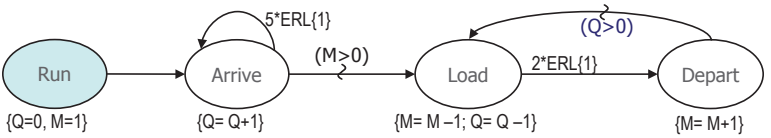


Fig. 4.35. SIGMA-compatible event graph of the single server system.

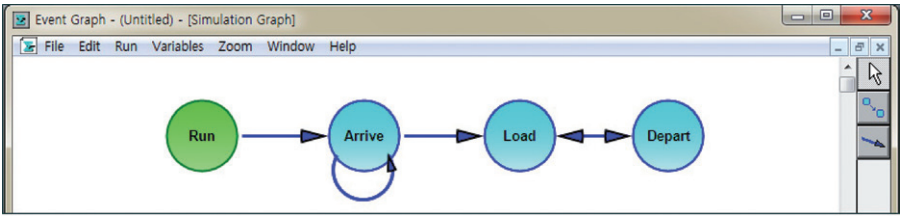


Fig. 4.36. SIGMA-generated event graph of the single server system.

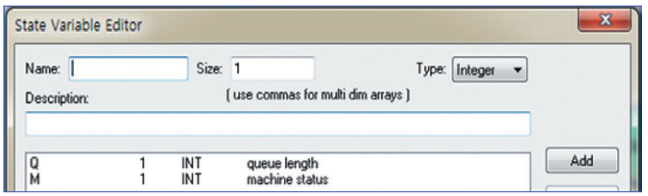


Fig. 4.37. Declaring Q and M as state variables.

is to obtain a SIGMA-generated event graph from the SIGMA-compatible event graph such as the one presented in Fig. 4.35. A SIGMA-generated event graph of a single server system is shown in Fig. 4.36 (see the SIGMA tutorial posted on the official website of this book) for further details. It is a graph consisting of four vertices and five directed edges. (There are two edges between the Load vertex and Depart vertex.) The vertices are named Run, Arrive, Load, and Depart.

4.6.1.2 Declaring State Variables All user-defined variables must be declared in the State Variable Editor window of SIGMA. As depicted in Fig. 4.37, the two variables, Q and M, are declared as integer variables. The number "1" in the row "Q 1 INT queue length" signifies that Q is an integer variable (or an array of size 1).

4.6.1.3 Defining the Run Vertex By clicking the first vertex of the SIGMA event graph (named Run), the dialog box Edit Vertex 1 is created, as shown in Fig. 4.38. Then, the variables (Q and M) that are initialized at the Run event vertex of Fig. 4.35 are entered in the parameters field of the Run vertex dialog box. (Q and M are initialized in the Run Options dialog box, as will be seen later in Fig. 4.42.)

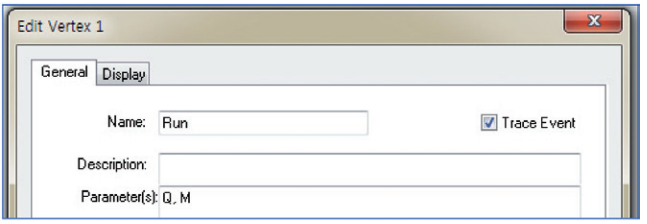


Fig. 4.38. Defining Q and M as parameters of the Run event vertex.

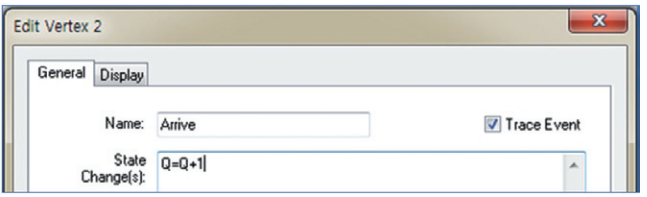


Fig. 4.39. Defining the state change (Q = Q+1) of the Arrive event.

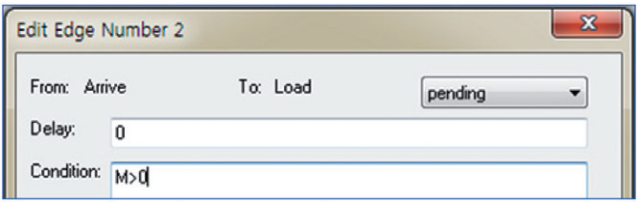


Fig. 4.40. Defining the time delay and edge condition of the Arrive→Load edge.

4.6.1.4 Defining Event Vertices (Arrive, Load, and Depart) The vertices are assigned numbers sequentially as they are created. There are three events in the single server system: Arrive, Load, and Depart. For example, by clicking the vertex Arrive of the event graph in Fig. 4.36, a dialog box Edit Vertex 2 is created as shown in Fig. 4.39. Then, the state change Q = Q+1 is entered in the State Change field of the dialog box. The state changes at other events are defined in the same way.

4.6.1.5 Defining Edges The edges are assigned numbers sequentially as they are created. For example, by clicking the edge Arrive→Load of the event graph in Fig. 4.36, a dialog box Edit Edge 2 is created, as shown in Fig. 4.40. Then, the time delay 0 and edge condition M > 0 are entered in the Delay field and Condition field of the dialog box, respectively.

For a double edge, each of the sub-edges is defined separately. By clicking the double edge Load↔Depart and then selecting its sub-edges, the time delay and edge condition of each sub-edge can be specified, as shown in Fig. 4.41.

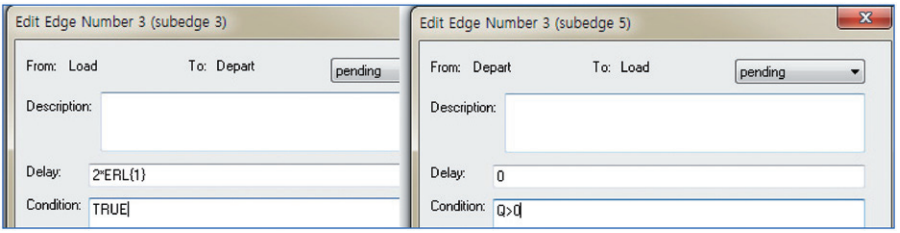


Fig. 4.41. Defining the time delay and edge condition of the sub-edges.

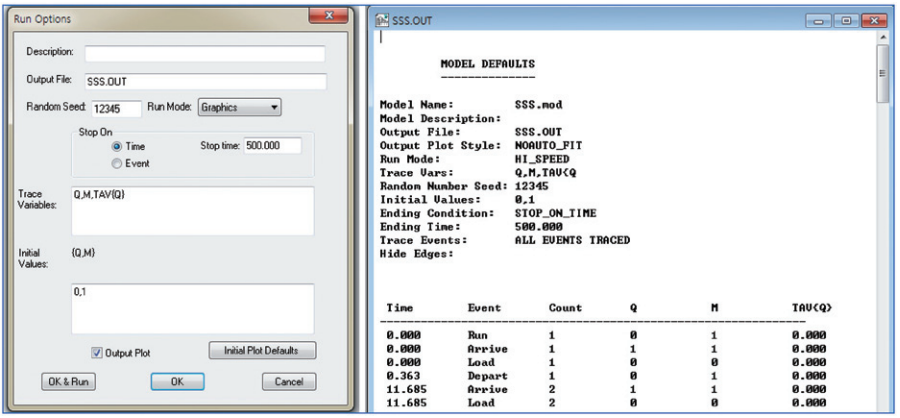


Fig. 4.42. Run Options dialog box and Model Defaults output.

4.6.1.6 Describing Run Options The experimental conditions and simulation output requirements are specified in the Run Options dialog box. The run options entered in the dialog box of Fig. 4.42 are:

- seed number for random variate generation: 12345;
- simulation run mode: graphic;
- EOS (end of simulation) time: 500 minutes;
- variables to be traced: Q, M, TAV(Q) // TAV stands for *time-average* //;
- initial values of the state variables Q and M: 0 and 1, respectively; and
- "Output Plot": enabled.

The model default output is shown in Fig. 4.42, and the output plots for the state variables Q and M are shown in Fig. 4.43.

4.6.2 Simulation of a Conveyor-Driven Serial Line with SIGMA

An event graph model of a two-stage conveyor-driven serial line with R1 = 500 and c2 = 10 is given in Fig. 4.44. The distributions of the processing times and

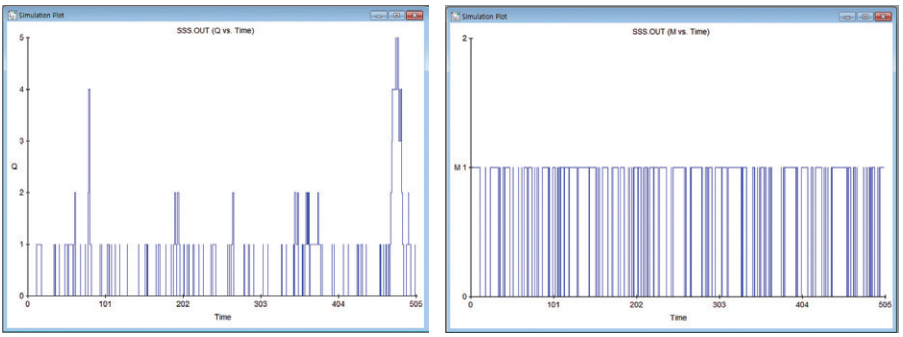


Fig. 4.43. Output plots of Q and M.

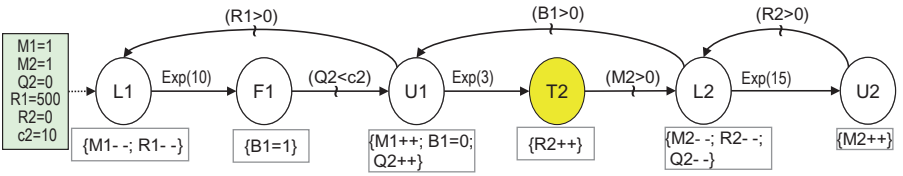


Fig. 4.44. Event graph model of a two-stage conveyor-driven serial line.

transport time are: p1 ~ Exp(10), p2 ~ Exp(15), and t2 ~ Exp(3). Assume that we are interested in the mean waiting time and mean queue length of the jobs in the internal conveyor.

4.6.2.1 Modifying the Event Graph to Collect the Waiting Time Statistics In this book, the variables introduced primarily for the purpose of collecting statistics are called *statistics variables*. The waiting time (WT) of a job is computed by subtracting the job's entering time from its leaving time, for which the queue is defined as a ranked list of job-entering times. In SIGMA, (1) the current simulation clock time is obtained from the function CLK, (2) the function PUT{O;L} is used for en-queuing a record into the ranked list L with option O (=FIF, LIF, INC, or DEC) and GET{O;L} for de-queuing, and (3) the record for en-queue/de-queue is stored in the built-in array ENT[]. A successful call to PUT{} or GET{} returns a value of 1. Thus, in order to collect the waiting time statistics, the following statistics variables must be specified as shown in Fig. 4.45.

At the U1 event vertex: ENT[0] = CLK; Q2 = Q2 + PUT{FIF;1}
At the L2 event vertex: Q2 = Q2 – GET{FST;1}; WT = CLK – ENT[0].

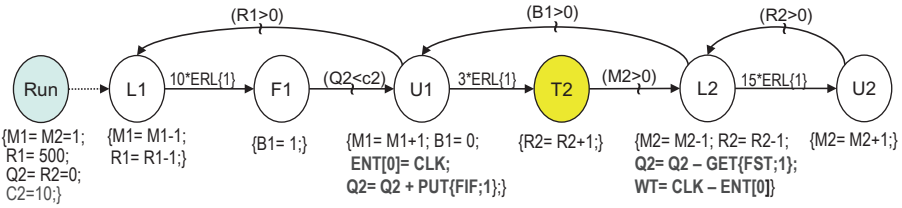


Fig. 4.45. SIGMA-compatible event graph of two-stage conveyor-driven serial line.

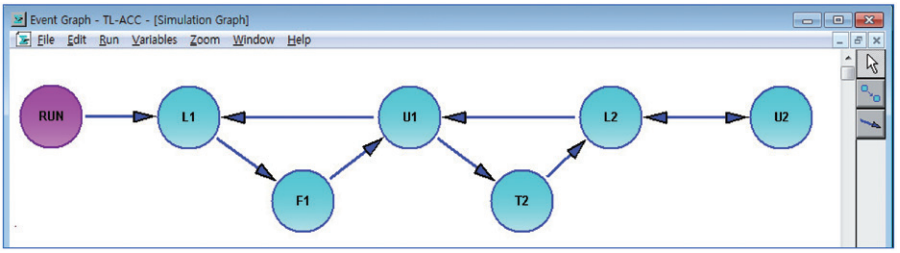


Fig. 4.46. Event graph of Fig. 4.45 constructed using the SIGMA GUI.

In the SIGMA model of Fig. 4.45, the number of jobs in the conveyor is stored in the integer variable Q2, and the waiting time of each job at the conveyor is stored in the real variable WT. The first entry of the ENT array, ENT[0], is used as a buffer for storing the data record into the built-in ranked list #1 with the PUT{FIF;1} function. The data stored in the FIFO (first-in, first-out) queue ranked list #1 is retrieved using the GET{FST;1} function.

4.6.2.2 Simulating the Conveyor-Driven Serial Line with SIGMA As described in Section 4.6.1, the procedure for executing an event graph model consists of six steps. The first step is to draw the event graph (Fig. 4.45) using the graphical user interface (GUI) functions of the SIGMA software as shown in Fig. 4.46.

The second step is to open the State Variable Editor dialog box and declare all variables appearing in the modified event graph of Fig. 4.45 as state variables. As shown in the left side of Fig. 4.47, M1, M2, Q2, R1, R2, B1, and C2 are integer variables; WT (waiting time) is a real variable; RNK is an integer array (with a size of 10,000); and ENT is a real array (with a size of 15). The third step is to bring in the Edit Vertex 1 dialog box and specify all user-defined variables in the RUN vertex as its parameter variables, as shown in the right side of Fig. 4.47.

The fourth step is to create an Edit Vertex dialog box for each event vertex in the SIGMA event graph of Fig. 4.46 and enter the state change expressions in the State Change field. An example of the Edit Vertex dialog box for the U1 event is shown in the left side of Fig. 4.48: M1 = M1 + 1, B1 = 0,

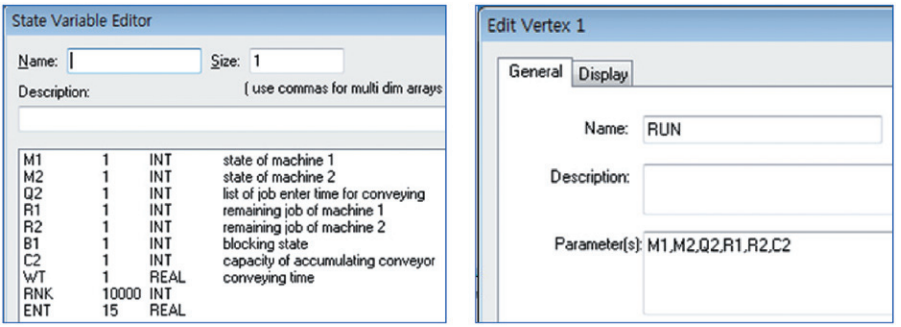


Fig. 4.47. Declaration of state variables and specifying parameter variables.

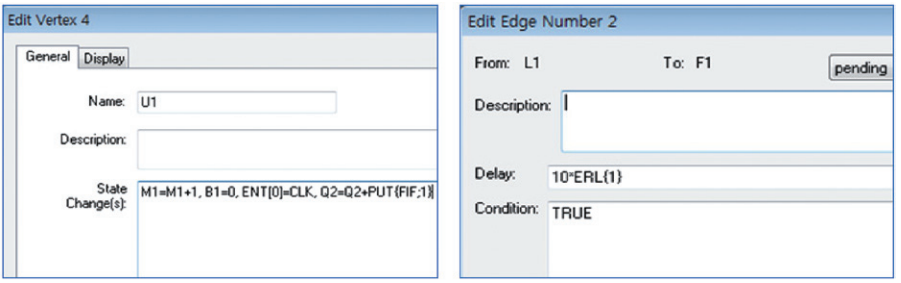


Fig. 4.48. Entering information for a vertex (U1) and an edge (L1→F1).

ENT[0] = CLK, Q2 = Q2 + PUT{FIF;1}. The fifth step is to describe the time delay and edge condition of each edge of the event graph. Shown in the right side of Fig. 4.48 is the Edit Edge dialog box for the L1→F1 edge, where the time delay is given as 10*ERL{1} and the edge condition is TRUE.

The sixth step is to create the Run Options dialog box and specify the experimental conditions and output requirements as depicted in the left side of Fig. 4.49, where the run options are specified as follows:

- seed number for random variate generation: 12345;
- simulation run mode: graphics;
- EOS (end of simulation) time: 5000;
- variables to be traced: Q2, M1, M2, TAV{Q2}, WT, AVE{WT};
- initial values of the state variables: M1 = 1, M2 = 1, Q2 = 0, R1 = 500, R2 = 0, C2 = 10; and
- "Output Plot": enabled.

Shown in the right side of Fig. 4.49 are the Run Option values (i.e., model default output) and a listing of the values of the traced variables at each event time. The output plots of Q2 and WT with respect to CLK are shown in Fig. 4.50.

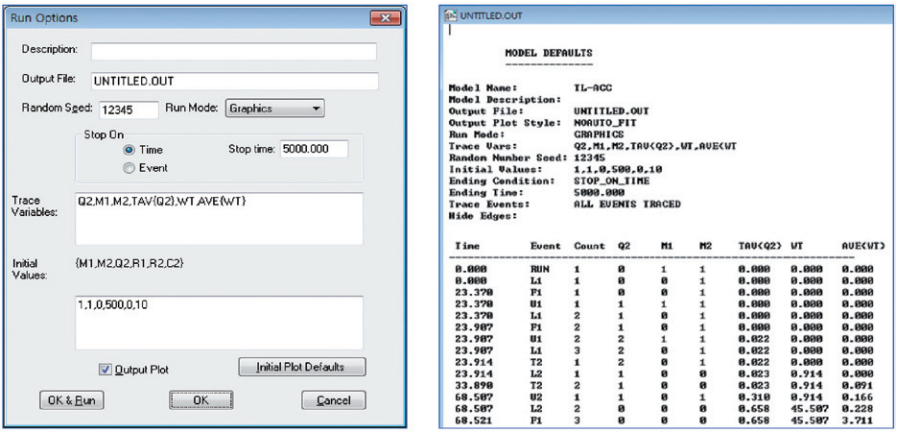


Fig. 4.49. Run Options dialog box and Model Defaults output.

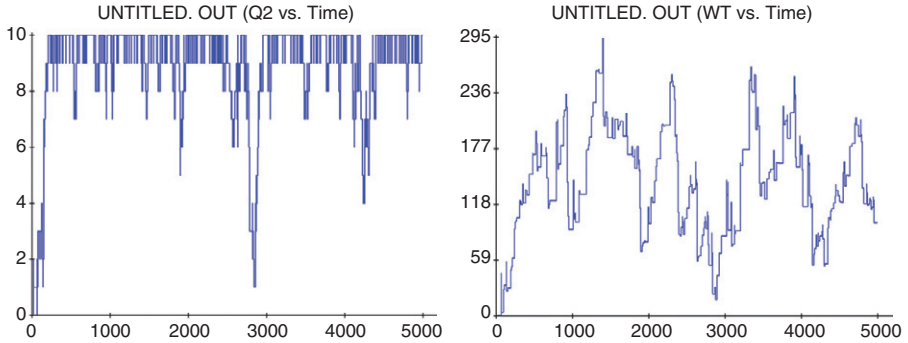


Fig. 4.50. Output plots of Q2 (queue size) and WT (waiting time).

4.7 DEVELOPING YOUR OWN EVENT GRAPH SIMULATOR

This section aims to help you become able to develop your own simulation program for executing a given event graph model. If you have foundational skills in computer programming and follow this section carefully, you should be able to write your own event graph simulator. The event graph simulator for simulating the single server model will be given in pseudocode form, which is language-independent and thus may be easier to understand. A complete list of C# codes for the single server simulator may be found on the official website of this book (http://VMS-technology.com/Book/EventGraphSimulator).

4.7.1 Functions for Handling Events and Managing Queues

The method of developing a dedicated simulator for a given event graph model will be described in a bottom-up manner, starting from the primitive functions for handling events and managing queues.

Figure 4.51 provides a schematic description of the three event-handling functions: Schedule-event (), Retrieve-event (), and Cancel-event (). Initially, there are three future events {<E1, 12.1>, <E2, 18.6>, <E3, 34.0>} stored in the *future event list (FEL)*. The management of these events using these functions will be explained with examples.

1. If the Schedule-event (E4, 22.7) function is invoked, the scheduled event <E4, 22.7> is inserted immediately after <E2, 18.6> in the FEL, which is a priority queue of event records, prioritized according to the increasing values of the event time. Now FEL has four future events: <E1, 12.1>, <E2, 18.6>, <E4, 22.7>, and <E3, 34.0>.
2. If the Retrieve-event (E, T) function is invoked, the next event <E = E1, T = 12.1> is retrieved (and deleted from the FEL).
3. If the Cancel-event (E4) function is invoked, the event node <E4, 22.7> is deleted from the FEL.

Figure 4.52 provides a schematic description of the basic queue handling functions for a FIFO (first-in, first-out) queue: (a) the New Q function will create a queue (as a variable array of records); (b) the en-queue function (j, x)→Q will append a record <10, 3.2>; (c) the en-queue function will append another record <20, 5.6>; and (d) the de-queue function Q→(j, p) will remove the first record and return j = 10 and p = 3.2.

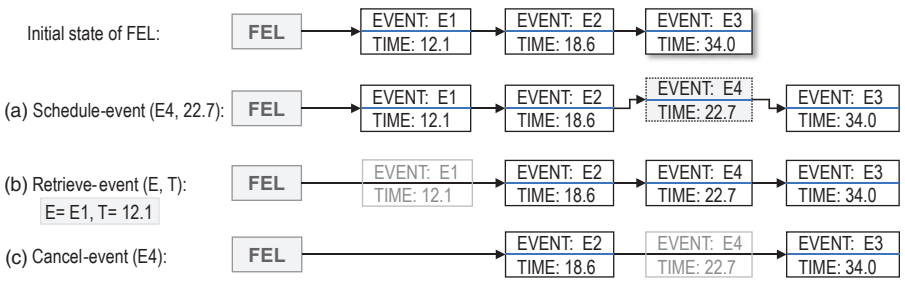


Fig. 4.51. Schematic descriptions of the event-handling functions.

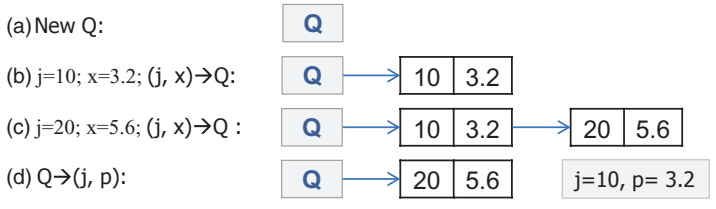


Fig. 4.52. Schematic descriptions of the queue-handling functions.

4.7.2 Functions for Generating Random Variates

Most programming languages support a built-in function for generating a standard uniform random number u~U[0,1]. In Java, the function u = Math. random() has the same function. Let x ~ U[a, b], then x is obtained from u as follows: x = a + (b − a) ∗ u.

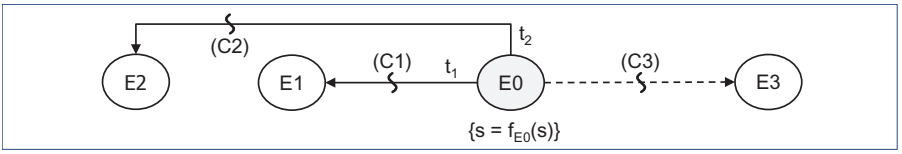
An exponential random variate X is generated from a uniform random number U as follows. Since the distribution function F(X) can be regarded as a uniform random number U, we have $U = F(X) = 1 - e^{-x/\theta}$, where θ is the mean. Upon solving this equation for X, we can obtain $X = -\theta \cdot \ln(1 - U)$, which is equivalent to $X = -\theta \cdot \ln(U)$ because (1 − U) is also a uniform random number. This method of generating a random variable is referred to as the *inverse-transformation* method. (See Appendix 3B of Chapter 3 for more details.)

In Java, the natural log ln(U) is implemented as Math.log (u). The random variable generation functions for the inter-arrival times and service times are listed below in a Java-like form. More details on this subject are provided in Chapter 3.

```
Exp (a):
{  If (a <= 0) then return False; u = Math.random ();
  Return (– a * Math.log (u)); }
Uni (a, b):
{  If (a >= b) then return False; u = Math.random ();
  Return (a + (b – a) * u); }
```

4.7.3 Event Routines

Figure 4.53 shows a portion of an event graph for an event vertex that has two scheduling edges and one canceling edge. The event graph indicates that "whenever E0 occurs, the state variable s changes to $f_{E0}(s)$. Then, if edge condition C1 is true, E1 is scheduled to occur after t_1; if edge condition C2 is true, E2 is scheduled to occur after t_2; and if edge condition C3 is true, E3 is canceled immediately."



Originating Event	State Change	Edge	Condition	Action	Delay	Destination Event
E0	$s = f_{E0}(s)$	1	C1	schedule	t_1	E1
		2	C2	schedule	t_2	E2
		3	C3	cancel	0	E3

Fig. 4.53. Event vertex with two scheduling edges and a canceling edge.

Also shown in Fig. 4.53 is an event transition table for E0. An event routine is a subprogram describing the changes in state variables and how the next events are scheduled and/or canceled for an originating event in the event transition table. One event routine is required for each event in an event graph. The event routine for the E0 event in Fig. 4.53 can be expressed as follows:

```
Execute-E0-event-routine (Now) // Fig. 4.53 //
{ s=f_E0(s); // state change
   If (C1) Schedule-event (E1, Now+ t_1);
   If (C2) Schedule-event (E2, Now+ t_2);
   If (C3) Cancel-event (E3);}.
```

4.7.4 Next Event Methodology of Simulation Execution

As described earlier in Chapter 2 (Section 2.2.4), the simulation maintains a simulation clock (CLK) and a future event list (FEL). The FEL is an ordered list of pairs $\{E_k, t_k\}$, where t_k is the scheduled execution time of the event E_k. The FEL is also a priority queue, ordered in increasing values of t_k. The overall procedure of the simulation execution, which is called the *next event methodology*, is as follows:

0. Reset the simulation clock CLK.
1. Initialize state variables and schedule initial events.
2. Time flow mechanism: get <E-type, E-time> from the FEL and set CLK to E-time.
3. Execute the event routine for the event E-type.
4. If a termination condition is not satisfied, go back to step 2.
5. Output statistics and stop.

The above next event methodology of the simulation execution, often called the *next event scheduling algorithm*, may be drawn as a flow chart as given in Fig. 4.54.

A template of an event graph model consisting of a set of event vertices $\{E_k: k = 1\text{-}n\}$ is depicted in Fig. 4.55. As shown in the figure, the given (pure) event graph must be augmented with a Statistics box as well as with the statistics variables.

Notice in Fig. 4.55 that that the simulation is stopped if an EOS (end of simulation) condition is met. Assuming that the Initialize box and Statistics box are implemented as an initialize routine and a statistics routine, respectively, the main program of the event graph simulator for executing the template event graph model of Fig. 4.55 will have the structure shown in Fig. 4.56. Listed in the Event-routine list are the event routines for E1~En.

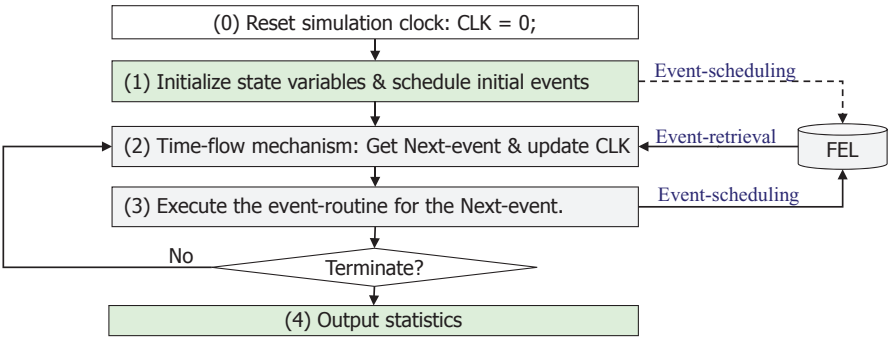


Fig. 4.54. Next event scheduling algorithm.

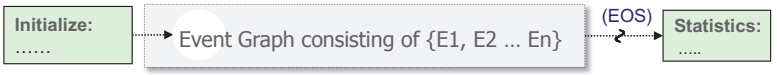


Fig. 4.55. Template of an augmented event graph model.

```
Main-Program // Fig.4.55 //
Begin
    CLK = 0;
    Execute-initialize-routine (CLK);                    // (1) Initialize
    While (CLK < 500) do {     // te = 500
        Retrieve-event (EVENT, TIME); CLK = TIME;  //   (2) Time-flow mechanism
        Case EVENT of {                                  // (3) Execute event-routine
              E1:  Execute-E1-event-routine (CLK);
              E2:  Execute-E2-event-routine (CLK);
               ….
              En:  Execute-En-event-routine (CLK);
        } // end-of-case
     }; // end-of-while
     Execute-statistics-routine (CLK);                   // (4) Output statistics
End
```

Event-routines list

Fig. 4.56. Main program of the template event graph simulator.

4.7.5 Single Server System Simulator

In this book, an event graph model only concerned with the dynamic behavior of the system without statistics variables is called a *pure event graph*. If the pure event graph is augmented with statistics variables for collecting statistics, it is called an *augmented event graph*.

Figure 4.57 presents a pure event graph model of the single server system introduced earlier in Fig. 4.1. In the figure, Q is the number of jobs in the buffer, M is the number of idle machines, and te is the EOS (end of simulation) time.

Shown in Fig. 4.58 is an augmented event graph for collecting the average queue length (AQL) statistics. Let $\{C_k\}$ denote the queue length change times,

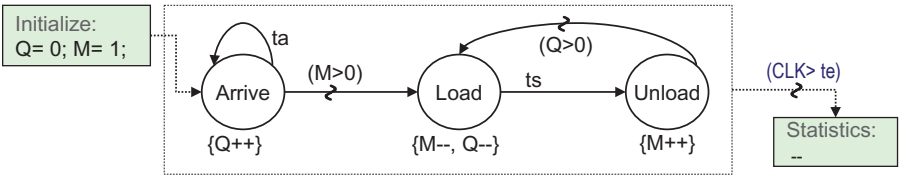


Fig. 4.57. Pure event graph model of single server system.

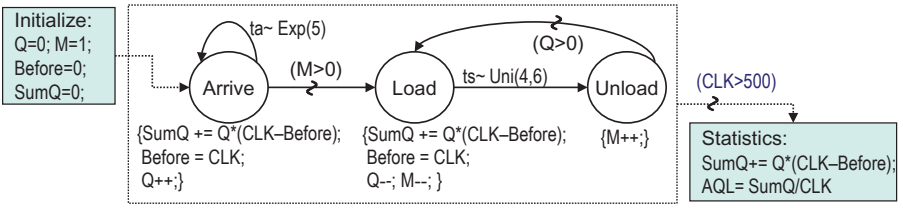


Fig. 4.58. Augmented event graph model for collecting AQL statistics.

TABLE 4.5. Event Transition Table for the Event Graph Model of Figure 4.58

No	Originating Event	State Change	Edge	Condition	Delay	Destination Event
0	Initialize	Q = 0; M = 1; Before = 0; SumQ = 0	1	True	—	Arrive
1	Arrive	SumQ += Q*(CLK– Before); Before = CLK; Q = Q + 1	1 2	True M > 0	Exp(5) 0	Arrive Load
2	Load	SumQ += Q*(CLK– Before); Before = CLK; M = M + 1; Q = Q − 1;	1	True	Uni(4,6)	Unload
3	Unload	M = M + 1;	1	Q > 0	0	Load
4	Statistics	SumQ += Q*(CLK – Before); AQL = SumQ/CLK				

then the k^{th} queue length change interval becomes $\Delta k = C_{k+1} - C_k$. Let Q_k be the queue size during Δk, then the AQL is expressed as $AQL = \Sigma(Q_k \times \Delta k)/\Sigma(\Delta k) \equiv SumQ/CLK$. An event transition table for this event graph model is given in Table 4.5.

In the event graph of Fig. 4.58, it is assumed that the inter-arrival times follow an exponential distribution with a mean of 5 and that the service times follow a uniform distribution with a range of 4.0–6.0. The initialize routine, event routines, and statistics routine of the augmented event graph model for collecting the AQL statistics are as follows:

```
Execute-Initialize-routine (Now) // Fig. 4.58 //
{  Q = 0;  M = 1;  Before = 0;  SumQ = 0;  Schedule-event
   (Arrive, Now); }
Execute-Arrive-event-routine (Now) // Fig. 4.58 //
{ SumQ = SumQ + Q*(Now—Before); Before = Now; Q = Q + 1;
   Schedule-event (Arrive, Now+ Exp (5)); If (M > 0)
   Schedule-event (Load, Now); }
Execute-statistics-routine (Now) // Fig. 4.58 //
{ SumQ = SumQ + Q*(Now—Before); AQL = SumQ/Now; }.
```

Then, from the template event graph simulator in Fig. 4.56, a single server system simulator is obtained as shown in Fig. 4.59.

Exercise 4.12. Write two event routines Execute-Load-event-routine () and Execute-Unload-event-routine ().

Another statistic that is commonly collected is the average waiting time (AWT) of the jobs in a queue. The waiting time (WT) of a job is computed by subtracting the arrival time (AT) from the load time at the Load event. Let N be the number of jobs loaded during a simulation, then the average waiting time is expressed as AWT = Σ(WT)/N. An augmented event graph for collecting the AWT statistics is given in Fig. 4.60, where the arrival time clock (CLK)

```
Main-Program of Single Server System Simulator
Begin
    CLK = 0;
    Execute-Initialize-routine (CLK);
    While (CLK < 500) do {    // te = 500
        Retrieve-event (EVENT, TIME); CLK = TIME;
        Case EVENT of {
                Arrive:        Execute-Arrive-event-routine (CLK);
                Load:          Execute-Load-event-routine (CLK);
                Unload:        Execute-Unload-event-routine (CLK);
        } // end-of-case
    }; // end-of-while
    Execute-statistics-routine (CLK);
End
```

Fig. 4.59. Main program of single server system simulator.

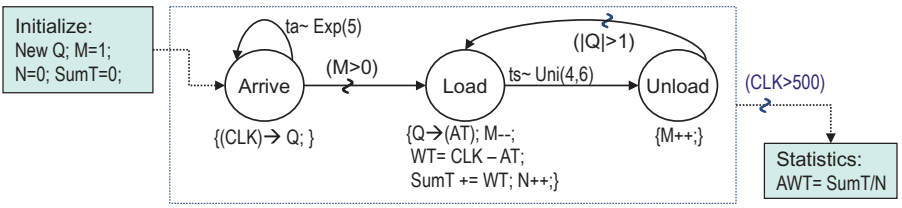


Fig. 4.60. Augmented event graph model for collecting AWT statistics.

is stored in the queue (Q) at the Arrive event, and it is retrieved from Q and assigned to the arrival time variable (AT) at the Load event.

Exercise 4.13. Modify the event graph in Fig. 4.60 to collect both AQL and AWT statistics.

4.8 REVIEW QUESTIONS

4.1. What are the three steps of the integrated simulation modeling (USM) procedure?

4.2. What are the logical modeling components of the single server system?

4.3. How are the state variables initialized in SIGMA?

4.4. How can an exponential random variate with a mean of 5 be generated in SIGMA?

4.5. How do you obtain a multi-server event graph model from a single server model?

4.6. What is balking? What is blocking?

4.7. What is the SIGMA function for en-queuing a record into the ranked-list L?

4.8. Where is the record for an en-queue/de-queue operation stored in SIGMA?

4.9. What does it mean to set RNK[5] = 1 in SIGMA?

4.10. How do you implement a FIFO queue in SIGMA?

4.11. What is the takt time of inline-type equipment?

4.12. What is the flow time of inline-type equipment?

4.13. How do you compute the average queue length of a time-dependent variable?

CHAPTER 5

Parameterized Event Graph Modeling and Simulation

Perfection is achieved not when there is nothing more to add, but when there is nothing left to take away.

—Antoine de Saint-Exupery

5.1 INTRODUCTION

The breakthrough improvement of the ordinary event graph framework is the parameterization of event vertices in which similar events are represented by a single vertex with different parameter values [Schruben 1995]. This enhancement enables the construction of a generalized model that represents a class of systems.

Parameterizing an event vertex is similar to defining an array variable for a large number of data items. The expression for computing the average of three data items (A, B, C) is given by:

$$\text{Mean} = (A + B + C)/3.$$

This expression for computing the sample mean can be represented by the event graph model shown in Fig. 5.1(a).

However, if you are asked to compute the sample mean of, say, 500 data points, you may define an array variable D[j] and formulate the following expression:

$$\text{Mean} = \frac{1}{500}\sum_{i=1}^{500} \text{D[j]}.$$

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.

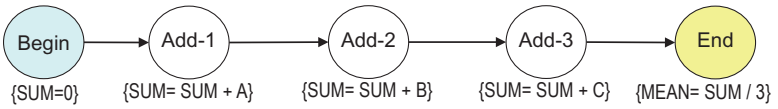


Fig. 5.1(a). Event graph model for computing the average of three data points.

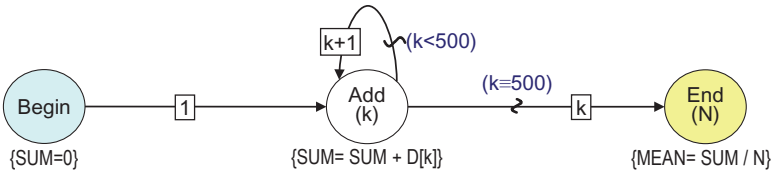


Fig. 5.1(b). Parameterized event graph for computing the sample mean.

Let D[j] be an array of data points. Then, the above sample mean expression can be modeled as a parameterized event graph, as shown in Fig. 5.1(b). In the figure, the summation operation is represented by a single parameterized event vertex.

This chapter is organized as follows. Examples of *parameterized event graphs* (PEGs) are introduced in Section 5.2, and the execution rules and specifications of the PEG models are presented in Section 5.3. The two sections that follow are devoted to the PEG modeling of tandem lines and job shops, respectively. Section 5.6 demonstrates the execution of PEG models with SIGMA. Finally, a method of developing your own PEG simulator is covered in the last section.

5.2 PARAMETERIZED EVENT GRAPH EXAMPLES

This section presents a number of parameterized event graph (PEG) examples. There are two common cases of parameterizing an ordinary event graph. The primary case is building a PEG model by introducing *indexing variables* to the repeating pattern of event vertices. The second case is defining a PEG model by passing *attribute values* of each entity along event vertices.

5.2.1 Introducing Index Variables to a Repeating Event-Vertex Pattern

Figure 5.2 presents an event graph model of the two-stage tandem line defined in Chapter 2 (see Fig. 2.15). There are six event vertices in the two-stage tandem line model. In general, an n-stage tandem line model has 3n event vertices. Figure 5.3 shows a PEG model of an n-stage tandem line, which is obtained by introducing an indexing variable (k) to the repeating pattern of event vertices "Enter-Load-Unload" in the event graph shown in Fig. 5.2.

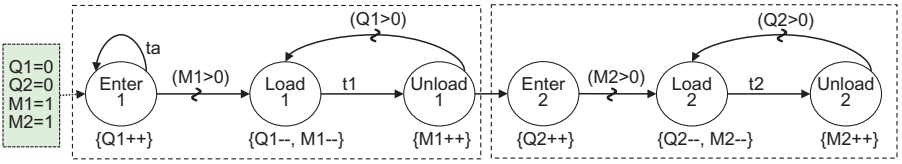


Fig. 5.2. Event graph model of a two-stage tandem line.

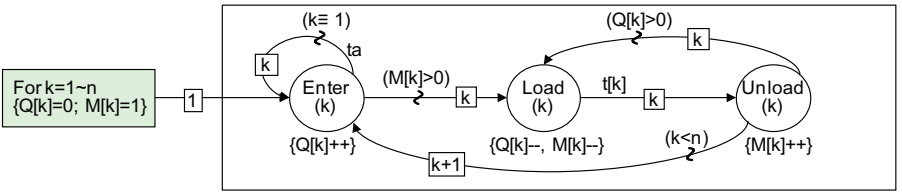


Fig. 5.3. Parameterized event graph model of the n-stage tandem line.

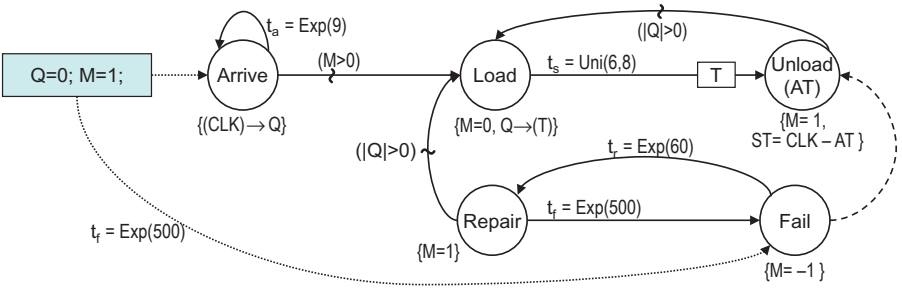


Fig. 5.4. Modified event graph model for collecting sojourn time statistics.

5.2.2 Passing Attribute Values of Each Entity along Event Vertices

Let us assume that we want to collect the sojourn time statistics when simulating the single server resource failure model introduced in Chapter 4 (see Fig. 4.13). In order to collect the sojourn time statistics, the arrival time of each job that was generated by the Arrival event must be passed through to the Unload event.

For this purpose, the original event graph model (see Fig. 4.13) is modified as shown in Fig. 5.4. Namely, (1) the clock time (CLK) of each job arrival is stored in a ranked list Q using an enqueue operation ((CLK)→Q) at the Arrive event; (2) the arrival time is retrieved from Q using a dequeue operation (Q→(T)) at the Load event; (3) the retrieved time (T) is passed to the Unload event as a parameter value. Then, the sojourn time (ST) is computed by subtracting the arrival time (AT) from the simulation clock (CLK) at the Unload event. In Fig. 5.4, the distributions of inter-arrival times (t_a), service times (t_s), interfailure times (t_f), and repair times (t_r) are given by $t_a \sim \text{Exp}(9)$, $t_s \sim \text{Uni}(6,8)$, $t_f \sim \text{Exp}(500)$, and $t_r \sim \text{Exp}(60)$.

5.3 EXECUTION RULES AND SPECIFICATIONS OF THE PARAMETERIZED EVENT GRAPH

A PEG model provides a compact description of a discrete-event system (DES) to be interpreted by a human. Its execution rules and specifications are similar to those of an ordinary event graph model described in Chapter 4.

5.3.1 Execution Rules of the PEG Model

In a PEG, a parameter value is a scheduling edge's attribute value that is passed to a destination event, whereas a parameter variable is a destination event's indexing or state variable whose value is set to the passed parameter value. The execution rules for a PEG are essentially the same as those for an ordinary event graph, with some minor differences as explained below.

PEGs are executed vertex by vertex. Figure 5.5 depicts a PEG with an originating event (E1) and a destination event (E2), where k is the parameter value associated with the scheduling edge, and j is the parameter variable associated with the destination event.

The PEG model shown in Fig. 5.5 may be interpreted as follows [Schruben 1995]:

> Whenever event E1 occurs, the state of the system changes to $f_{E1}(s)$. Then, if the edge condition (C) is true, destination event E2(j) is scheduled to occur after a time delay of t with the value of the parameter variable j equal to the edge parameter value k.

In general, each of the parameter value k and parameter variable j can be a vector. The scheduled execution time (τ) of E2 is obtained by adding the time delay (t) to the current execution time (*Clock*) of E1. Namely, $\tau = Clock + t$.

5.3.2 Tabular Specifications of the PEG Model

As with an ordinary event graph, the event execution rules for a PEG model can be specified in an event transition table. A parameterized event transition table is a table that describes (1) the state changes and outgoing edge numbers of each event and (2) the edge condition, delay time, parameter value, and destination event of each edge. It has one more column than an ordinary event transition table: the Parameter column for specifying the parameter value of each edge. Table 5.1 is an event transition table of the PEG model given in Fig. 5.3.

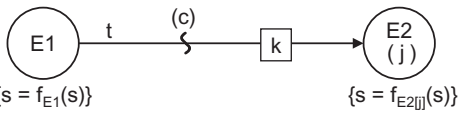


Fig. 5.5. Parameterized event graph with a scheduling edge.

TABLE 5.1. Event Transition Table for the Parameterized Event Graph Shown in Figure 5.3

No	Event	State Change	Edge	Condition	Delay	Parameter	Destination Event
0	Initialize	For k = 1~n {Q[k] = 0; M[k] = 1}	1	True	0	1	Enter (k)
1	Enter(k)	Q[k] ++;	1	k ≡ 1	ta	k	Enter (k)
			2	M[k] > 0	0	k	Load (k)
2	Load(k)	Q[k]—; M[k]—;	1	True	t[k]	k	Unload (k)
3	Unload(k)	M[k] ++;	1	Q[k] > 0	0	k	Load (k)
			2	k < n	0	k + 1	Enter (k)

5.3.3 Algebraic Specifications of the PEG Model

An ordinary event graph (without canceling edges) is a directed graph consisting of a set of event vertices (V), a set of directed edges (E), and a set of state variables (S). Associated with each vertex ($v \in V$) is a state transition function ($f_v \in F$), and associated with each edge ($e \in E$) are an edge condition ($c_e \in C$) and a time delay ($d_e \in D$).

In addition to the above-mentioned six elements (V, E, S, F, C, and D), a PEG has two more elements: a set of parameter value lists (K) associated with each edge and a set of parameter variable lists (J) associated with each vertex. Thus, a PEG model ($\mathbf{M_P}$) can be defined as an 8-tuple structure [Savage et al. 2005], as follows:

$\mathbf{M_P} = <V, E, S, F, C, D, K, J>$, where

$V = \{v\}$: set of event vertices

$E = \{e_{od} = (v_o, v_d)\}$: set of edges // v_o: originating event; v_d: destination event

$S = \{s\}$: set of state variables

$F = \{f_v: S \rightarrow S \ \forall v \in V\}$: set of state transition functions associated with each vertex (v)

$C = \{c_e: S \rightarrow [0,1] \ \forall e \in E\}$: set of conditions associated with each edge (e)

$D = \{d_e \in R_0^{\infty} \ \forall e \in E\}$: set of time delays associated with each edge (e)

$K = \{k_e \ \forall e \in E\}$: set of parameter value lists, if any, associated with each edge (e)

$J = \{j_v \ \forall v \in V\}$: set of parameter variable lists, if any, associated with each vertex (v)

For example, the components of the event graph in Fig. 5.3 (n-stage tandem line) are as follows:

1. $V = \{v_1 = \text{Enter}, v_2 = \text{Load}, v_3 = \text{Unload}\}$
2. $E = \{e_1 = (v_1, v_1), e_2 = (v_1, v_2), e_3 = (v_2, v_3), e_4 = (v_3, v_2), e_5 = (v_3, v_1)\}$
3. $S = \{\text{Q[k]}, \text{M[k]}\}$
4. $F = \{f_1: \text{Q[k]++}, f_2: \text{Q[k]—}, \text{M[k]—}, f_3: \text{M[k]++}\}$
5. $C = \{c_1: (\text{k} \equiv 1), c_2: (\text{M[k]} > 0), c_3: \text{True}, c_4: (\text{Q[k]} > 0), c_5: (\text{k} < \text{n})\}$
6. $D = \{d_1 = \text{ta}, d_2 = 0, d_3 = \text{t[k]}, d_4 = 0, d_5 = 0\}$
7. $K = \{k_1 = \text{“k”}, k_2 = \text{“k”}, k_3 = \text{“k”}, k_4 = \text{“k”}, k_5 = \text{“k + 1”}\}$
8. $J = \{j_1 = j_2 = j_3 = \text{“k”}\}$

5.4 PARAMETERIZED EVENT GRAPH MODELING OF TANDEM LINES

In the previous chapter, a number of tandem line models were introduced: an unlimited buffer tandem line model (Fig. 4.15), a limited buffer tandem line model (Fig. 4.16), a buffer-less tandem line model (Fig. 4.17), and a conveyor-driven serial line model (Fig. 4.25). In this section, methods of building PEG models for some tandem line event graph models are described.

5.4.1 PEG Modeling of an Unlimited Buffer Tandem Line

An event graph model of a two-stage unlimited buffer tandem line is given in Fig. 5.6; this model is obtained from the event graph shown in Fig. 5.2 by splitting the original Enter-1 event into Arrive and Enter-1 events. The event graph model shown in Fig. 5.6 has a repeating pattern of Enter(k)-Load(k)-Unload(k), which is identical for all {k}. Thus, the PEG model for the event graph model shown in Fig. 5.7, where N is the number of stages in the tandem line.

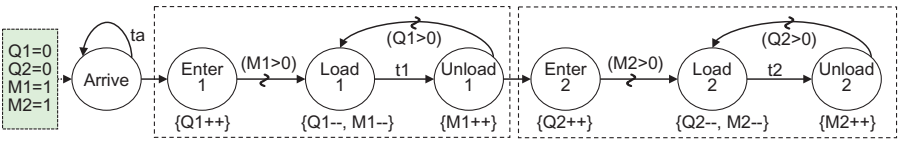


Fig. 5.6. Event graph model of a two-stage unlimited buffer tandem line.

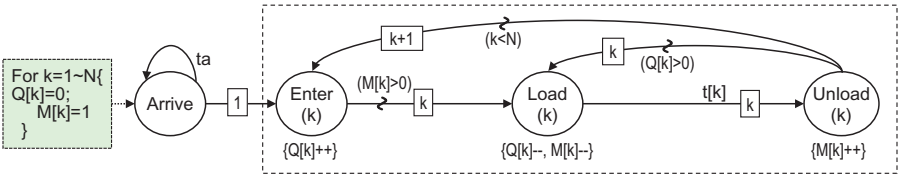


Fig. 5.7. PEG model of the unlimited buffer tandem line shown in Fig. 5.6.

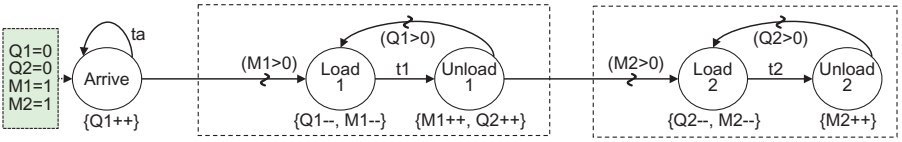


Fig. 5.8. Revised event graph of the unlimited buffer tandem line.

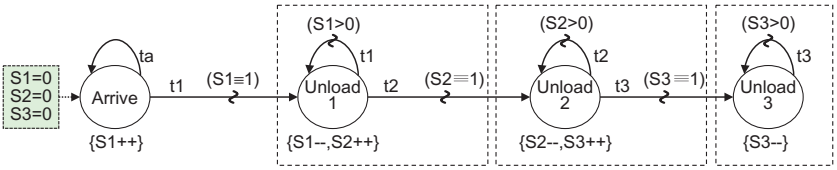


Fig. 5.9. Simplified event graph of the three-stage unlimited buffer tandem line.

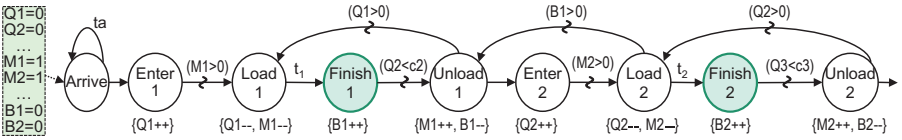


Fig. 5.10. Revised event graph model of the limited buffer tandem line.

In the ordinary event graph model shown in Fig. 5.6, the only operation performed by the Enter-k event is to increase Q_k by one. Thus, this event vertex may be deleted without loss of modeling power, as shown in Fig. 5.8. The revised event graph has a repeating pattern of Load(k)-Unload(k).

Exercise 5.1. Construct a PEG model for the event graph model shown in Fig. 5.8.

The event graph model of the unlimited buffer tandem line may be further simplified by eliminating the Load-k events. The resulting event graph model of a three-stage tandem line is given in Fig. 5.9. In this event graph model, the state variable (S_k) denotes the number of jobs in Stage-k.

5.4.2 PEG Modeling of a Limited Buffer Tandem Line

A limited buffer tandem line model is obtained from the unlimited buffer model shown in Fig. 5.2 (or Fig. 5.6) by inserting a Finish-k event between a Load-k event and an Unload-k event. Figure 5.10 shows an event graph model of the limited buffer tandem line with the repeating pattern of Enter-Load-Finish-Unload.

In Fig. 5.10, (1) the Finish-k event sets the blocking variable B_k to 1 (B_k++) and schedules the Unload-k event if the buffer is not full ($Q_{k+1} < c_{k+1}$) and (2) the Load-k+1 event schedules an Unload-k event if the blocking variable is true ($B_k > 0$). The PEG model of the limited buffer tandem line is as given in

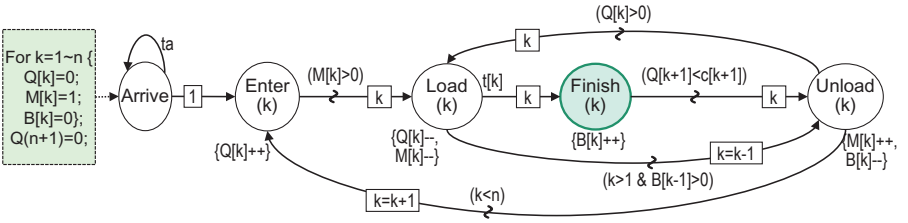


Fig. 5.11. PEG model of the limited buffer tandem line.

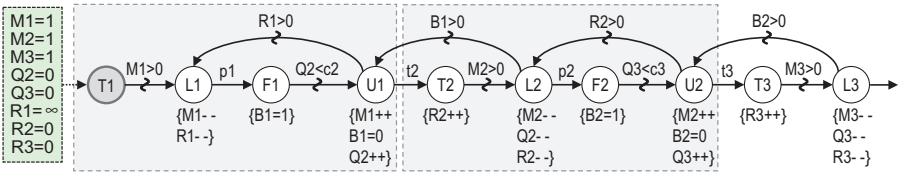


Fig. 5.12. Revised event graph model of the conveyor-driven serial line.

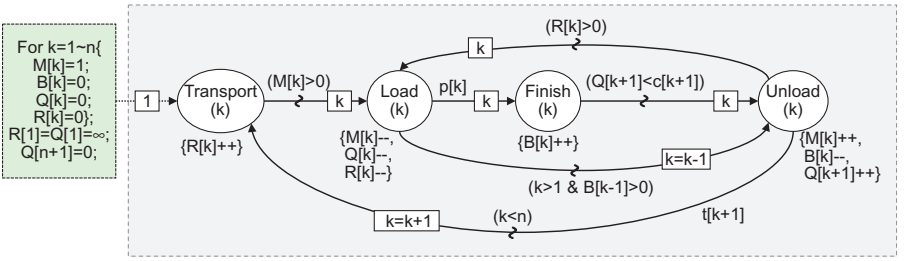


Fig. 5.13. PEG model of the conveyor driven serial line.

Fig. 5.11. It should be noted that the Arrive event schedules the Enter(1) event and the Unload(k) event schedules an Enter(k + 1) event when k < n.

5.4.3 PEG Modeling of a Conveyor-Driven Serial Line

Reproduced in Fig. 5.12 is the event graph model of the conveyor-driven serial line given in Fig. 4.25 of Chapter 4. (For a detailed description of the conveyor-driven serial line, please refer to Section 4.5.4.) In Fig. 5.12, a dummy transport event (T1) is added to the event graph model of Fig. 4.25 in order to form the repeating pattern of the Transport-Load-Finish-Unload events.

A PEG model of the conveyor-driven serial line is presented in Fig. 5.13. Since the input buffer Buffer-I is treated as a conveyor with an unlimited capacity and zero conveying time, c[1] = ∞ and t[1] = 0. Also, Q[n + 1] = 0 and c[n + 1] = ∞ need to be set as boundary conditions.

5.5 PARAMETERIZED EVENT GRAPH MODELING OF JOB SHOPS

A discrete-event system is called a *job shop* if (1) it consists of a number of stations (s) with each station having one or more identical machines, (2) there are a number of job types (j) with each job type having its own unique sequence of processing steps (p = 1, 2, . . .), and (3) the station number (s) for a processing step (p) of a given job type (j) is specified in the routing sequence of the job type. A job may visit a given station more than once.

A machine that processes one job at a time is referred to as a *table machine*. A simple job shop is a job shop consisting of table machines and unlimited buffers. Figure 5.14 depicts a reference model of such a job shop. It has six stations with the number of machines given by $\{m_1 = 3, m_2 = 5, m_3 = 4, m_4 = 7, m_5 = 2, m_6 = 5\}$. The routing sequence of a type-1 job is 1-3-2-5-Done: route[1,1] = 1, route[1,2] = 3, . . . , route[1,5] = Done. The processing time of a type-j job at processing step (p) is denoted by $t\,[j, p]$.

Any job shop that is not a simple job shop is called a *complex job shop*. Examples of complex job shops are inline job shops and mixed job shops. An inline job shop is a job shop consisting of the inline cells described in Chapter 4 (Section 4.5.5).

5.5.1 PEG Modeling of a Simple Job Shop without Transport

If we set $M[s] = m_s$ in the initialize event box of the unlimited buffer tandem line PEG model shown in Fig. 5.7, the model becomes a multi-server tandem line model, as depicted in Fig. 5.15. Another slight change in Fig. 5.15 (from Fig. 5.7) is that the next station (ns) is updated at the Unload(s) event vertex.

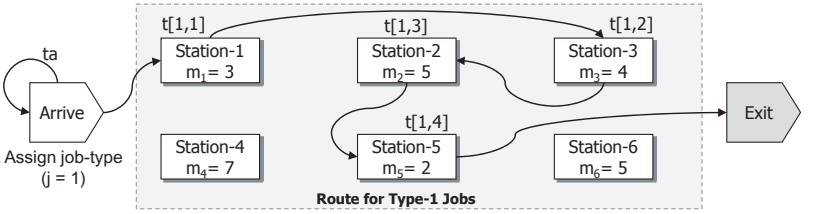


Fig. 5.14. Reference model of a simple job shop (for job type j = 1).

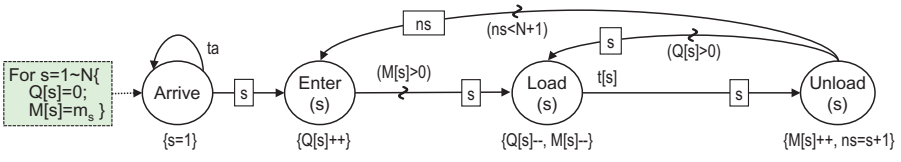


Fig. 5.15. PEG model of a multi-server tandem line obtained from Fig. 5.7.

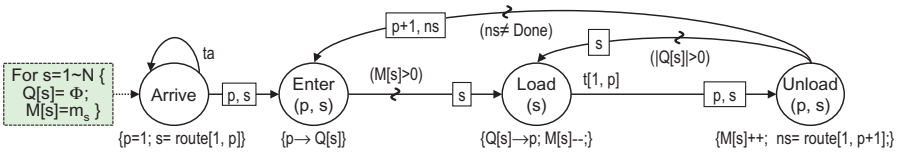


Fig. 5.16. PEG model of a simple job shop for processing a single job type (j = 1).

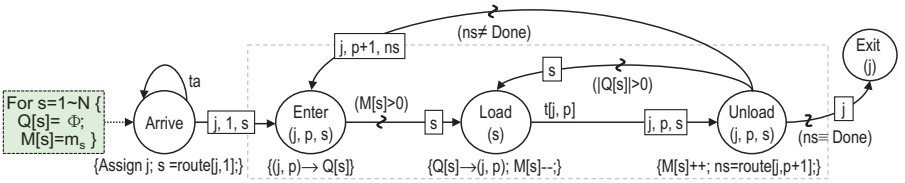


Fig. 5.17. PEG model of a simple job shop for processing multiple job types.

This tandem line is a special case of a simple job shop in which (1) there is only one job type and (2) the job processing step (p) is the same as the station number (s; for s = 1, 2, . . . , N). Thus, as depicted in Fig. 5.16, the tandem line PEG model shown in Fig. 5.15 may be converted to a PEG model describing the simple job shop shown in Fig. 5.14.

In the job shop PEG model shown in Fig. 5.16, a new job generated by the Arrive event is passed to the Enter event with parameter values {p = 1 and s = route [1, p]}. At the Enter (1, s) event, the new job is put into the queue Q[s], and a Load(s) event is scheduled if M[s] > 0. Since there is only one job type, a job is represented by its processing step (p). Q[s] is the queue of jobs identified by the current processing step.

When a Load(s) event is fired in the PEG model of Fig. 5.16, a job is retrieved from Q[s] for processing and an Unload(p,s) event is scheduled to occur after t[1, p] minutes. At the Unload(p,s) event, the processing step (p) is increased by one and the station number (ns) for the next processing step (p + 1) is determined by evaluating ns = route [1, p + 1]. Here, an Enter(p + 1, ns) event is scheduled if the next station number is not equal to Done and a Load(s) event is scheduled if Q[s] is not empty.

The PEG model of Fig. 5.16 may be generalized easily to a PEG model of a simple job shop by adding j (for j = 1~J) to the parameter list as shown in Fig. 5.17, where a job is represented by its job type and processing step. The state variables in the simple job shop model are:

j = job type;
p = processing step of a job;
s = station number for a job;
M[s] = number of idle machines in station s;
Q[s] = list of jobs {(j, p)} at station s.

Jobs of type j follow the routing sequence given by the table route [j, p] with the processing time t[j, p]. If a job finishes the last processing step, then the station number of the next processing step is set to Done so that the job exits the job shop. The initialization box shown in Fig. 5.17 is the same as that shown in Fig. 5.16.

At the Arrive event, a job type (j) is assigned to each new job and the station number (s) for the first processing step is determined as: s = route [j,1]. At the Enter (j, p, s) event, the job (j, p) is placed in the queue of station s and a Load(s) event is scheduled to occur immediately if the station (s) has an idle machine (M[s] > 0). At the Load(s) event, a job (j, p) is retrieved from the queue and an Unload(j, p, s) event is scheduled to occur after a time delay of t[j, p] minutes. At the Unload(j, p, s) event, the processing step is increased by one and the next station number is determined. Then (1) a Load(s) event is scheduled if the queue at station s is not empty, (2) an Enter(j, p, s) event is scheduled if the job needs another processing step, and (3) an Exit(j) event is scheduled if the job is done. These dynamic behaviors of the simple job shop model can be formally specified in an event transition table, as shown in Table 5.2.

Exercise 5.2. Revise the PEG model of the simple job shop shown in Fig. 5.17 by adding a new event Select-s (j, p) where the station number (s) for a job (j, p) is determined.

5.5.2 PEG Modeling of a Job Shop with Transport and Setup Times

In the simple job shop PEG model shown in Fig. 5.17, only the net processing times (t[j, p]) are reflected. In practice, considerable amounts of setup time (when the job type is changed) and transport delay (when the job is to be moved by a transporter) may be incurred in a job shop.

TABLE 5.2. Event Transition Table for the PEG Model of a Simple Job Shop (Figure 5.17)

No	Event	State Change	Edge	Condition	Delay	Parameter	Next Event
0	Initialize (Q, M)	For s = 1~N {Q[s] = Φ; M[s] = m_s}	1	True	0	—	Arrive
1	Arrive	Assign j; s = route[j, 1];	1	True	0	j, 1, s	Enter (j, p, s)
			2	True	ta	—	Arrive
2	Enter (j, p, s)	(j, p) → Q[s];	1	M[s] > 0	0	s	Load (s)
3	Load (s)	Q[s]→ (j, p); M[s]—;	1	True	t[j, p]	j, p, s	Unload (j, p, s)
4	Unload (j, p, s)	M[s]++;	1	|Q[s]| > 0	0	s	Load (s)
		ns = route[j, p + 1];	2	ns ≠ Done	0	j, p + 1, ns	Enter (j, p, s)
			3	ns ≡ Done	0	j	Exit (j)
5	Exit (j)		1				

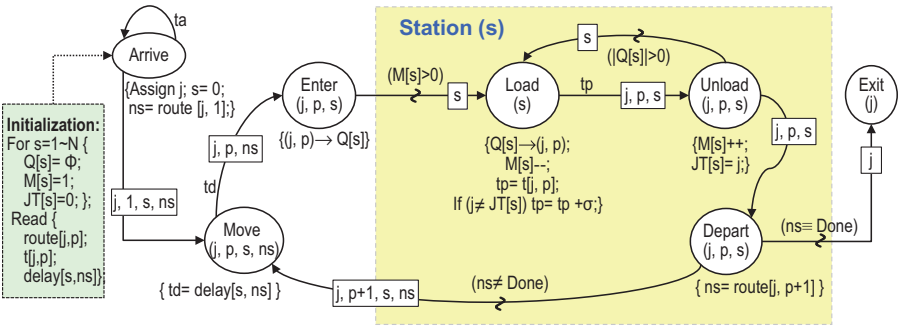


Fig. 5.18. Standard PEG model of a job shop reflecting setup time and transport time.

Figure 5.18 shows a PEG model of a simple job shop in which the setup times and the transport delay times are reflected. In order to reflect the setup time (σ) in the PEG model, a state variable denoting the current job type of each station (JT[s]) is introduced. (However, this is only valid when there is a single machine at each station.) It is assumed that all stations have identical setup times. Thus, the following additions are made to the PEG model shown in Fig. 5.17:

1. At initialization, the job type of each station is reset: JT[s] = 0 for s = 1-N.
2. At the Load event, a setup time (σ) is selectively added: If (j ≠ JT[s]) {tp = tp + σ}.
3. At the Unload event, the current job type of the station is updated: JT[s] = j.

The next station number (ns) and transport delay time (td) are modeled explicitly by the two event nodes Depart and Move: (1) ns is obtained from ns = route [j, p + 1] at the Depart event; (2) td (transport delay from station s to station ns) is evaluated from td = delay [s, ns] at the Move event.

The PEG model shown in Fig. 5.18 may be regarded as a standard template for defining a general job shop model from the PEG model of a station that starts with a Load event and ends at the Depart event. The event transition table for the PEG model shown in Fig. 5.18 is given in Table 5.3.

5.5.3 PEG Modeling of an Inline Job Shop

A job shop consisting of inline cells is called an *inline job shop*. Reproduced in Fig. 5.19 are the reference model (Fig. 4.27) and the event graph model (Fig. 4.34) of the inline-type cell introduced in Chapter 4. This is called a *uni-inline cell* because the input (Load) and output (Unload) operations are performed at the shared I/O-Port. It should be noted that the event graph (Fig. 5.19) without the redundant event FGU is the same as the one (Fig. 4.34) with FGU.

TABLE 5.3. Event Transition Table for the Revised PEG Model Shown in Figure 5.18

No	Event	State Change	Edge	Condition	Delay	Parameter	Next Event
0	Initialize	For s = 1~N {Q[s] = Φ; M[s] = 1; JT[s] = 0}; Read {route[j,p]; t[j,p]; delay[s, ns]}	1	True	0	—	Arrive
1	Arrive	Assign j; s = 0; ns = route[j, 1];	1	True	ta	—	Arrive
			2	True	0	j, 1, s, ns	Move (j, p, s, ns)
2	Move (j, p, s, ns)	td = delay[s, ns];	1	True	td	j, p, ns	Enter (j, p, s)
3	Enter (j, p, s)	(j, p) → Q[s];	1	M[s] > 0	0	s	Load (s)
4	Load (s)	Q[s]→ (j, p); M[s]—; tp = t[j, p]; If (j ≠ JT[s]) {tp = tp + σ};	1	True	tp	j, p, s	Unload (j, p, s)
5	Unload (j, p, s)	M[s]++; JT[s] = j;	1	|Q[s]| > 0	0	s	Load (s)
			2	True	0	j, p, s	Depart (j, p, s)
6	Depart (j, p, s)	ns = route[j, p + 1];	1	ns ≠ Done	0	j, p + 1, s, ns	Move (j, p, s, ns)
			2	ns ≡ Done	0	j	Exit (j)
7	Exit (j)		1				

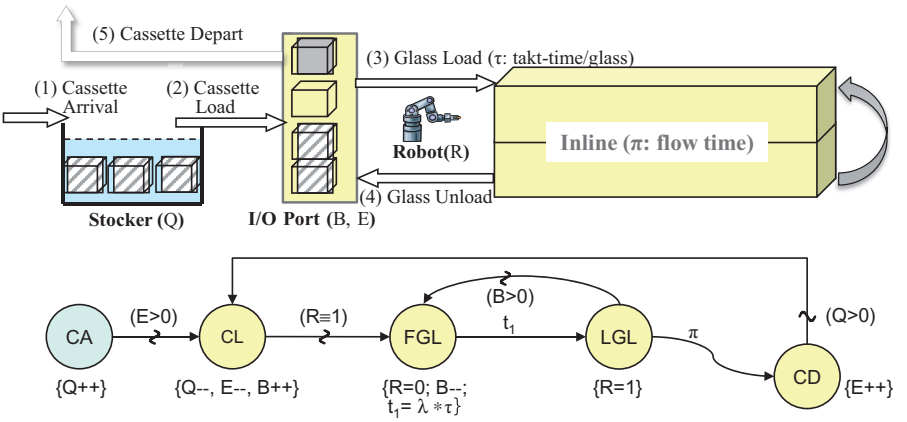


Fig. 5.19. Reference model and event graph of a uni-inline cell.

The event graph model in Fig. 5.19 has five event nodes, four state variables, and two time delay variables. The state variables are Q (number of arriving cassettes in the Stocker queue), B (number of arriving cassettes in the I/O-Port buffer), E (number of empty shelves in the I/O-Port), and R (status of Robot; 1 if Robot is idle, 0 if busy). The time delay variables are t_1 (cycle time for processing a cassette of glasses) and π (flow time). The state variables are changed by the events as follows:

- CA (Cassette Arrival) increases Q by one {Q++}
- CL (Cassette Load) decreases Q and E, and increases B by one {Q––, E––, B++}
- FGL (First Glass Load) sets Robot to busy and decreases B by one {R = 0, B––}
- LGL (Last Glass Load) sets Robot to idle {R = 1}
- CD (Cassette Departure) increases E by one {E++}

Let u denote the parameter variable for a uni-inline cell; then, all event vertices and state variables are parameterized in terms of u. In addition, the job type (j) and processing step (p) of a cassette may also be passed as parameter values. With these parameter variables, the state variables are defined as follows:

- Q[u]: Stocker queue of arriving cassettes {(j, p)} in the uni-inline cell (u)
- B[u]: I/O-Port queue of arriving cassettes {(j, p)} in the uni-inline cell (u)
- E[u]: number of empty ports (shelves) in the I/O-Port of a uni-inline cell (u)
- R[u]: status of the track-in Robot of a uni-inline cell (u)

The processing cycle time and flow time are parameterized as t_1[j, p] and π[j, p]. Thus, the event graph model of the uni-inline cell given in Fig. 5.19 may be parameterized as shown in Fig. 5.20. In the uni-inline cell PEG model shown in Fig. 5.20, the list handling operations are defined as follows.

- {(j, p) → Q[u]}: a job (j, p) is stored in the Stocker queue (Q[u])
- {(j, p) → B[u]}: a job (j, p) is stored in the I/O-Port queue (B[u])
- {Q[u] → (j, p)}: a job (j, p) is retrieved from the Stocker queue (Q[u])
- {B[u] → (j, p)}: a job (j, p) is retrieved from the I/O-Port queue (B[u])

In order to build a uni-inline job shop model, a state variable (JT[u]) denoting the current job type of a cell is introduced for modeling the setup time (σ) and the Move event is added to model the transport delay time (td) explicitly. Thus, as shown in Fig. 5.21, a standard PEG model of a uni-inline job shop may be constructed from the standard PEG model of a simple job shop shown

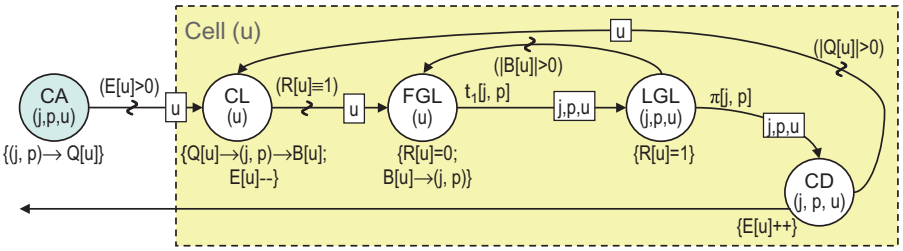


Fig. 5.20. PEG model of the uni-inline cell (u) given in Fig. 5.19.

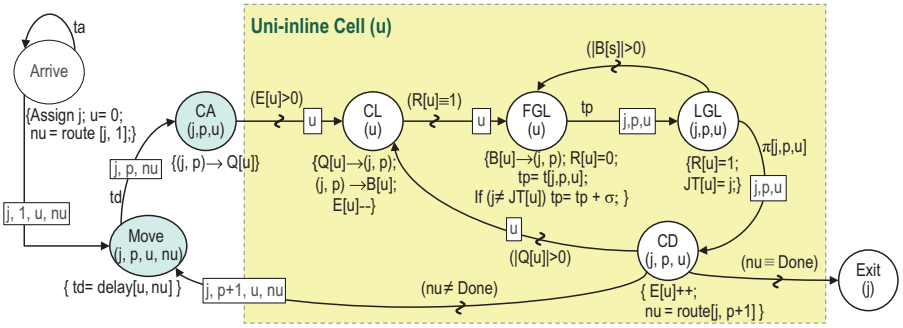


Fig. 5.21. Standard PEG model of a uni-inline job shop.

in Fig. 5.18 and the PEG model of the uni-inline cell shown in Fig. 5.20. As mentioned earlier, the state variables are the Stocker queue (Q[u]), I/O-Port queue (B[u]), number of empty shelves (E[u]), Robot status (R[u]), and job type (JT[u]). Assuming that the system is empty and the number of shelves in the I/O-Port of each cell (u) is 4, the state variables may be initialized as follows:

$$\text{For } u = 1\sim N \quad \{Q[u] = \Phi;\ B[u] = \Phi;\ E[u] = 4;\ R[u] = 1;\ JT[u] = 0\}.$$

5.5.4 PEG Modeling of a Mixed Job Shop

A mixed job shop may have different types of stations and/or cells. For example, by merging the two PEG models shown in Figs. 5.18 and 5.21, we can build a PEG model of a job shop consisting of table stations and uni-inline cells, as shown in Fig. 5.22. In the PEG model of the mixed job shop given in Fig. 5.22, **TM** denotes a set of table-type machines and **UC** a set of uni-inline cells in the job shop.

A job shop with different types of machines is often called a *heterogeneous job shop*, and one with one type of machine is called a *homogeneous job shop*.

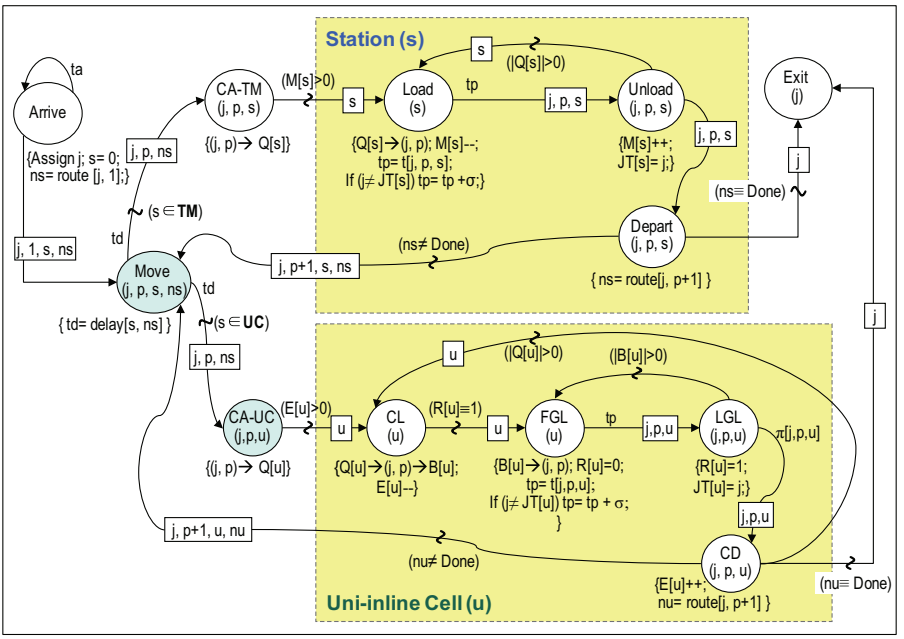


Fig. 5.22. Standard PEG model of a mixed job shop.

In a general job shop model, the material handling equipment may be modeled explicitly. More detailed discussions of these subjects may be found in Chapter 11 of this book.

5.6 EXECUTION OF PARAMETERIZED EVENT GRAPH MODELS USING SIGMA

The basic SIGMA functions were covered in the Chapter 4. This section aims to provide you with more experiences in and confidence with modeling with PEG and executing the PEG model with SIGMA. More specifically, this section demonstrates how to use certain advanced SIGMA functions in collecting the sojourn time statistics, reading array data, handling priority queues, and so on. All the SIGMA models discussed in this section, together with a brief SIGMA tutorial, may be found in the official website of this book (http://VMS-technology.com/Book/Sigma).

As mentioned in Chapter 4 (see Section 4.6), in order to execute a given event graph model using SIGMA, the event graph model is converted to a SIGMA-compatible event graph model. Figure 5.23 presents a schematic view of the SIGMA simulation program: all variables that appear in the SIGMA-compatible event graph model must be declared in the State Variable Editor; all state variables that are defined as parameters of the Run vertex are initial-

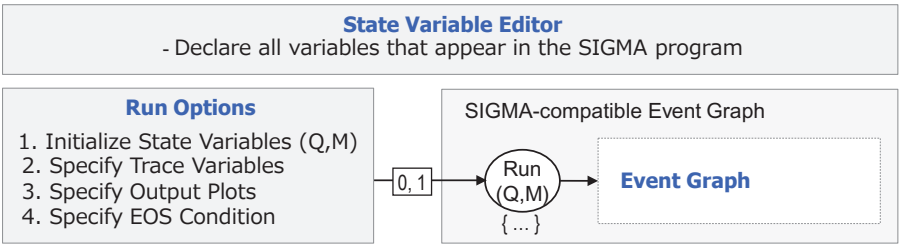


Fig. 5.23. Structure of the SIGMA simulation program.

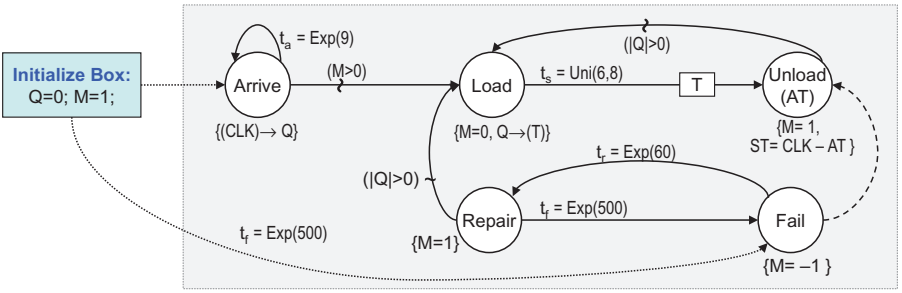


Fig. 5.24. Event graph model for collecting sojourn time statistics.

ized in the Run Options dialog box; the values of the state variables are passed to the Run vertex as parameter values.

5.6.1 Collecting Sojourn Time Statistics Using SIGMA Functions

Reproduced in Fig. 5.24 is the event graph model shown in Fig. 5.4 in Section 5.2.2. Recall from Section 5.2.2 that, in order to collect the sojourn time statistics, the arrival time of each job must be stored at the Arrival event and its sojourn time (ST) is computed by subtracting the arrival time (AT) from the departure time (CLK) at the Unload event.

5.6.1.1 Constructing a SIGMA-Compatible Event Graph In order to execute the event graph model using SIGMA, it must be converted to a SIGMA-compatible event graph in which (1) a Run event is defined to initialize the state variables and (2) the PUT{} and GET{} functions are used for storing and retrieving data in and from the built-in ranked list, respectively. A SIGMA-compatible event graph for collecting the sojourn time statistics is given in Fig. 5.25.

At the Run vertex shown in Fig. 5.25, RNK[1] = 0 is set so that ranked list 1 is ranked by the data field ENT[**0**]. At the Arrive vertex, the current arrival time (CLK) is assigned to ENT[0] and stored in ranked list **1** in an increasing order by invoking the PUT{INC;1} function. At the Load vertex, the arrival

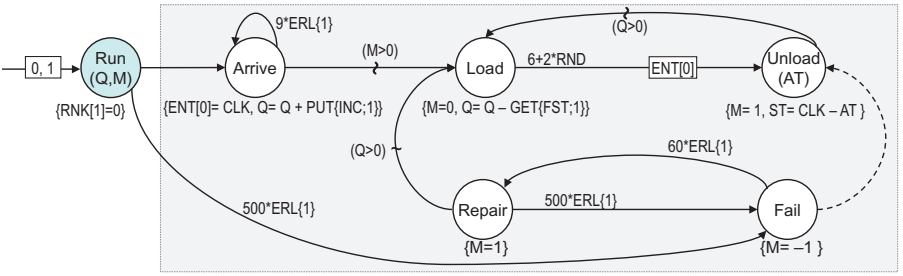


Fig. 5.25. SIGMA-compatible event graph for collecting sojourn time statistics.

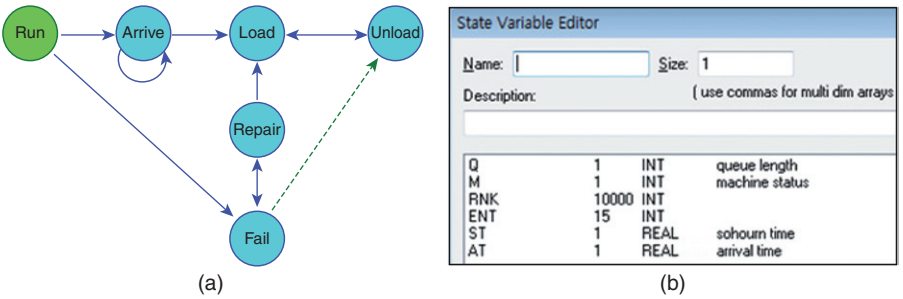


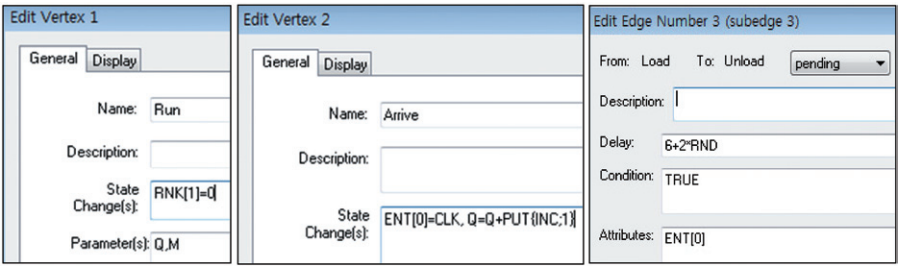
Fig. 5.26. (a) SIGMA-generated event graph and (b) declaration of state variables.

time is retrieved and stored in ENT[0] by the GET{FST;1} function; then, it is passed to the Unload vertex as an edge parameter value.

In Fig. 5.25, the number of jobs in the buffer is denoted by the integer variable (Q) and the sojourn time of a job is stored as the real variable (ST). The statistics that are collected are the time average queue length (TAV{Q}) and the average sojourn time (AVE{ST}). In this particular case, ranked list **1** is in effect a FIFO queue. Thus, we could use PUT{FIF,1} instead of PUT{INC,1}.

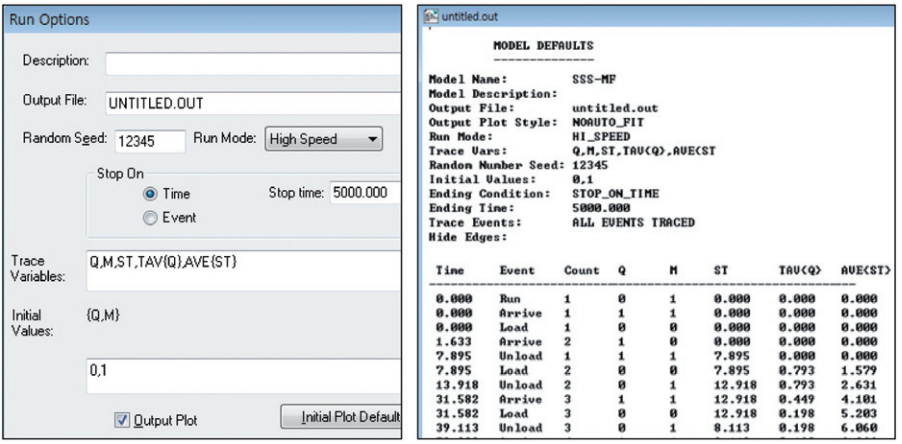
5.6.1.2 Building a SIGMA Program for Simulation As explained in Chapter 4, the procedure for building a SIGMA program consists of six steps. Step 1 is to create a graphical model of the SIGMA-compatible event graph on the main screen of SIGMA. As shown in Fig. 5.26(a), this graph consists of six vertices and eight edges. Step 2 is to bring in the State Variable Editor dialog box and declare all variables that appear in the SIGMA-compatible event graph. As shown in Fig. 5.26(b), the state variables are Q, M, RNK[10000], ENT[15], ST, and AT.

Step 3 is to define the Run vertex by creating the Edit Vertex 1 dialog box and specifying the state variables Q and M as its parameter variables. Step 4 is to create Edit Vertex dialog box for each event vertex in the SIGMA-generated event graph shown in Fig. 5.26(a) and to define its state changes and parameters where applicable. Figure 5.27(a) presents the Run vertex



(a) (b) (c)

Fig. 5.27. Defining the (a) Run vertex, (b) Arrive vertex, and (c) Load-Unload edge.



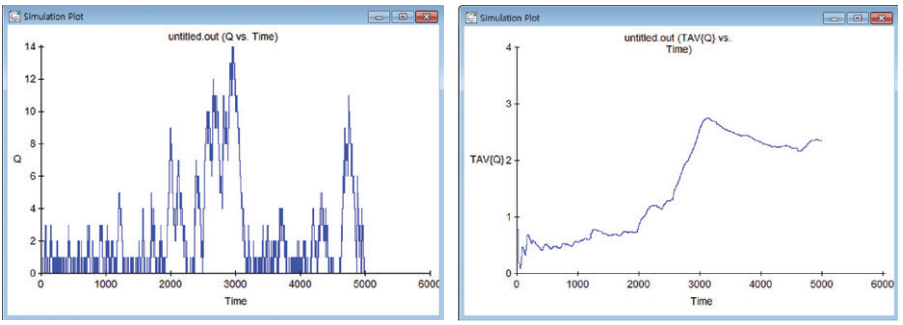
(a) (b)

Fig. 5.28. (a) Run Options dialog box and (b) Model Defaults output.

dialog box in which RNK[1] = 0 is entered in the State Change(s) field, and Q and M are entered in the Parameter(s) field (Q and M are initialized in the Run Options dialog box). Figure 5.27(b) shows the Arrive vertex dialog box.

The fifth step is to create an Edit Edge dialog box for each edge in the SIGMA-generated event graph shown in Fig. 5.26(a) and to specify the time delay value, edge condition, and parameter value. Figure 5.27(c) shows the dialog box of the edge Load→Unload whose time delay that is a Uniform(6,8) random variate is specified in the Delay field as "6+2*RND," edge condition TRUE is specified in the Condition field, and parameter value ENT[0] is specified in the Attributes field.

5.6.1.3 Running the SIGMA Program for Simulation The last step is to create the Run Options dialog box and to specify the experimental conditions and output requirements. Figure 5.28(a) shows the experimental conditions such as the random number seed (12345), end-of-simulation time (5,000 min), and the initial values of Q and M. Also specified in Fig. 5.28(a) are the variables



(a) (b)

Fig. 5.29. Simulation plots of the (a) queue sizes and (b) time average of the queue sizes.

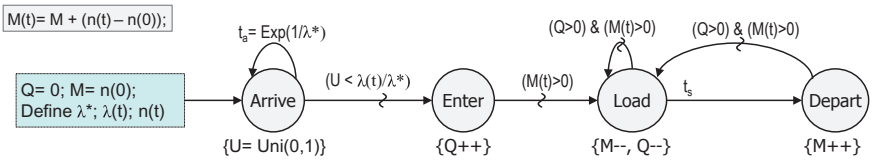


Fig. 5.30. Simple service shop with fluctuating arrival rates and varying resource levels.

(Q, M, ST) and statistics (time average of Q and average of ST) to be traced. Figure 5.28(b) shows the model default values and a list of the traced variable values at each event time.

Figure 5.29(a) and (b), respectively, present the simulation plots of the queue sizes (Q) and the time average of Q. The queue sizes fluctuate considerably as a result of the disturbances due to failures, but the time average of Q appears to converge to 2.

5.6.2 Simulating a Simple Service Shop with SIGMA

Figure 5.30 reproduces the event graph of a simple service shop that was given in Fig. 4.19 of Chapter 4 (Section 4.5.1). The simple service shop is subject to time-varying arrival rates (λ(t)). In order to manage the customer fluctuations, the number of servers (n(t)) is planned to change over time, which is often referred to as a *flexible multi-server system*.

Let us assume that customers arrive at the shop with arrival rates (customers per minute) of 0.0 during 0:00~5:59, 0.02 during 6:00~7:59, 0.10 during 8:00~9:59, and so on, as summarized in Table 5.4. That is, R[0] = R[1] = R[2] = 0, R[3] = 0.02, and so on. The maximum arrival rate is 0.5 during 14:00~15:59 (R[7] = 0.5). The base number of servers during the day hours (8:00~17:59) is three, with a peak of five during 2:00~3:59 p.m. (N[7] = 5). All servers are identical and their service times are exponentially distributed with a mean of 9.

TABLE 5.4. Arrival Rates and Number of Servers over a 24-Hour Period

Hours (120min)	0000-0159	0200-0359	0400-0559	0600-0759	0800-0959	1000-1159	1200-1359	1400-1559	1600-1759	1800-1959	2000-2159	2200-2359
k	0	1	2	3	4	5	6	7	8	9	10	11
R[k]	0.00	0.00	0.00	0.02	0.10	0.30	0.40	0.50	0.40	0.10	0.02	0.00
N[k]	0	0	0	0	3	3	3	5	3	1	1	0

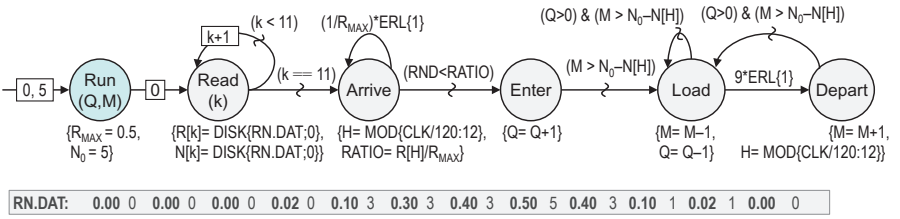


Fig. 5.31. SIGMA-compatible PEG model for managing fluctuations.

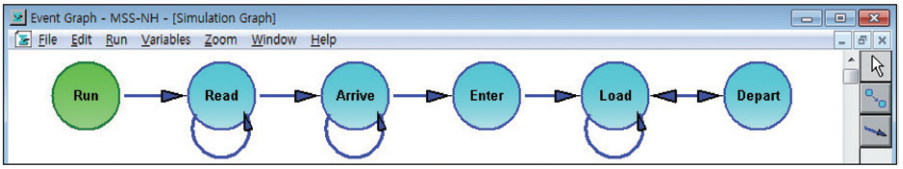
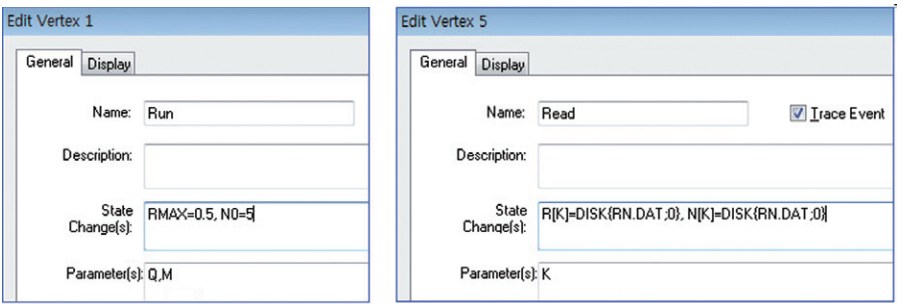


Fig. 5.32. SIGMA-generated event graph.

5.6.2.1 SIGMA-Compatible Event Graph for the Modified Event Graph Figure 5.31 presents a SIGMA-compatible event graph model of the modified event graph shown in Fig. 5.30. The figure also shows the input text file RN.DAT. The function DISK {RN.DAT; 0} reads the text file RN.DAT sequentially, and the for loop is implemented using the parameterized event Read(k). The simulation clock (CLK) is converted to the index value (h) shown in Fig. 5.31 using the function MOD{CLK/120; 12}. The default time unit of minutes is converted to 2-hour units by dividing CLK by 120. The RND function returns a standard uniform random variate.

5.6.2.2 Building a SIGMA Program for Simulation The first step of building a SIGMA program for simulation is to create a SIGMA-generated event graph on the main screen of SIGMA, as shown in Fig. 5.32. The second step is to bring in the State Variable Editor dialog box and declare all user-defined variables as state variables: Q, M, RMAX, R[12], N[12], N0, H, RATIO, and K (dialog box not shown).

The third step is to double click the Run vertex (shown in Fig. 5.32) to create the Edit Vertex 1 dialog box and enter the information {RMAX = 0.5; N0 = 5;



(a) (b)

Fig. 5.33. Defining the (a) Run vertex and (b) Read vertex.

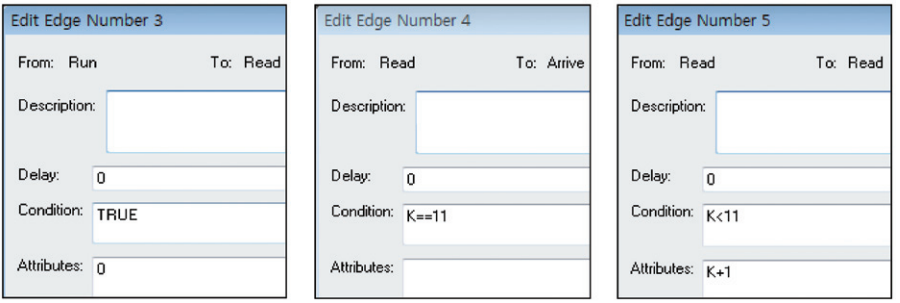


Fig. 5.34. Examples of edge dialog boxes.

Q = 0; M = N0} appearing in the Run vertex (shown in Fig. 5.31). As shown in Fig. 5.33(a), RMAX = 0.5 and N0 = 5 are entered in the State Change(s) field, and Q, M are specified in the Parameter(s) field. The fourth step is to create a dialog box for each event vertex. The dialog box of the Read vertex is shown in Fig. 5.33(b), in which the parameter value is K.

The fifth step is to create an Edit Edge dialog box for each of the edges in the SIGMA-generated event graph shown in Fig. 5.32. Figure 5.34 presents the dialog boxes of the three edges Run→Read (From: Run; To: Read), Read→Arrive, and Read→Read.

5.6.2.3 Running the SIGMA Program for Simulation The sixth step is to create the Run Options dialog box. Shown in Fig. 5.35(a) is the Run Options dialog box in which the experimental conditions such as the random number seed (12345), end-of-simulation time (5,000 min), and initial values of Q and M are specified. The dialog box also specifies the variables and statistics to be traced. Figure 5.35(b) shows a simulation plot of Q (number of customers in the queue) over the simulation time.

5.6.3 Simulation of a Three-Stage Tandem Line Using SIGMA

The PEG model introduced earlier in this chapter (Fig. 5.3 in Section 5.2.1) will be used as a vehicle for demonstrating the simulation of an n-stage tandem

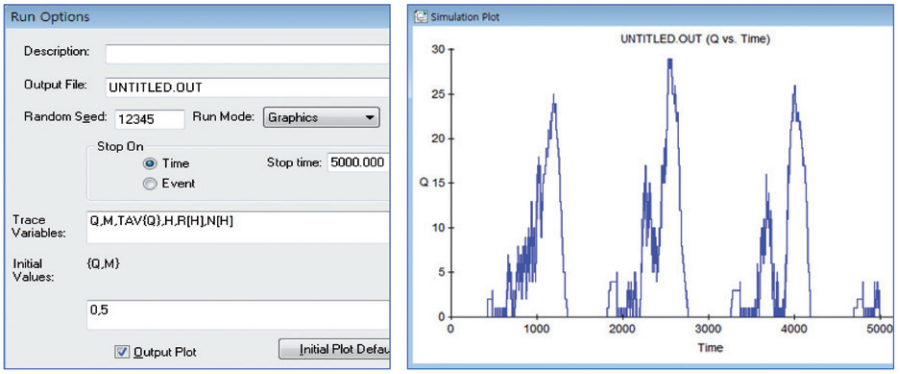


Fig. 5.35. Run Options dialog box and simulation plot of Q.

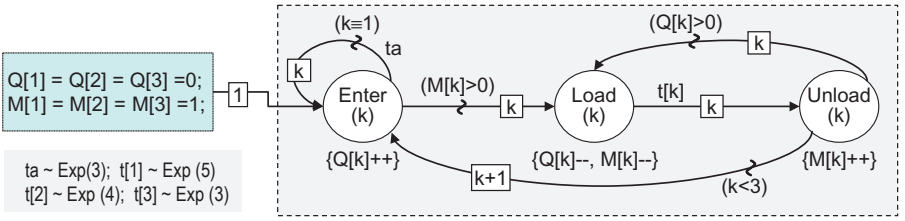


Fig. 5.36. PEG model of an n-stage tandem line (n = 3).

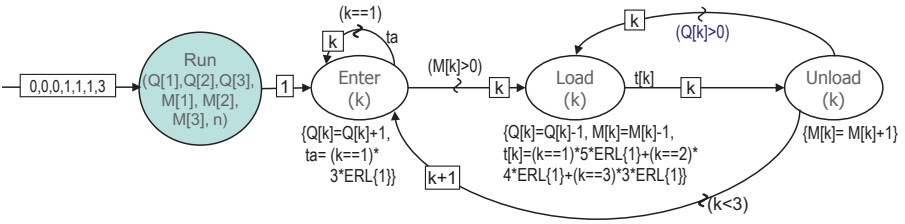


Fig. 5.37. SIGMA-compatible PEG model of the n-stage tandem line (n = 3).

line with SIGMA. The PEG model shown in Fig. 5.3 is reproduced in Fig. 5.36 with n = 3, where the distribution functions of the inter-arrival time and service times at the three stages are also specified.

5.6.3.1 Building a SIGMA-Compatible PEG Model In general, the first step in simulation with SIGMA is to modify the given event graph model in order to manage specific requirements. In this particular case, however, modifications are not required. Thus, the first step is to build a SIGMA-compatible PEG model from the neutral PEG model given in Fig. 5.36.

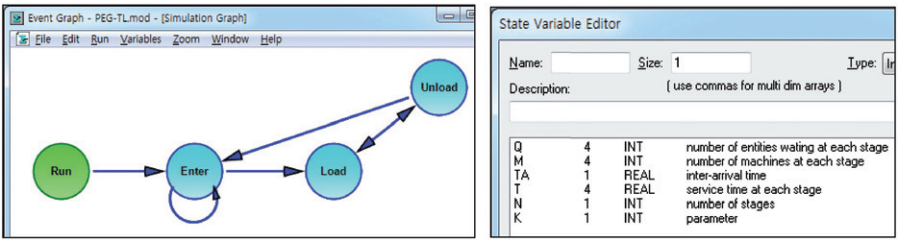
Figure 5.37 presents a SIGMA-compatible PEG model of the three-stage tandem line. Note in Fig. 5.37 that the number of stages "n" defined in the Run

vertex is never used in the model. A convenient feature of SIGMA is the use of Boolean variables. For example, the service time (t[k]) at stage k is given by Exp(5) if k ≡ 1, Exp(4) if k ≡ 2, and Exp(3) if k ≡ 3, which is expressed in SIGMA as follows:

$$t[k] = (k == 1)*5*ERL\{1\} + (k == 2)*4*ERL\{1\} + (k == 3)*3*ERL\{1\}.$$

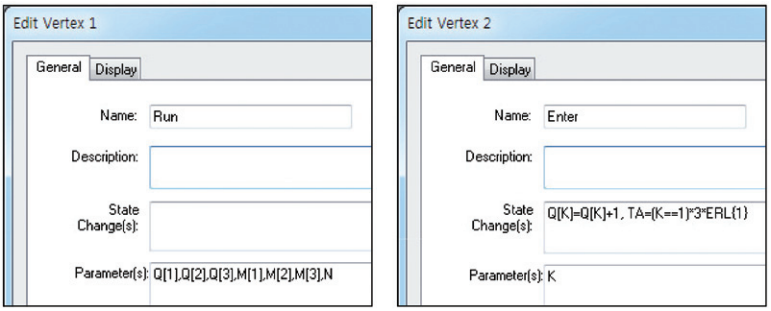
5.6.3.2 Building a SIGMA Program and Running the Simulation Figure 5.38(a) presents the SIGMA-generated event graph of the three-stage tandem line; Fig. 5.38(b) shows the State Variable Editor dialog box. The SIGMA-generated event graph for a PEG model is the same as that for an event graph model. The variables declared as the state variables are Q, M, TA, T, N, and K. The sizes of the arrays Q, M, and T are set to 4 in Fig. 5.38(b) because in Fig. 5.37 Q[k] and M[k] are defined for k=1, 2, 3 (Q[0] and M[0] are not used).

The dialog boxes for the Run vertex and the Enter vertex are given in Fig. 5.39(a) and (b), respectively. The variables to be initialized in the Run Options dialog box are declared as parameters in the Run vertex dialog box shown in Fig. 5.39(a). The State Change(s) (Q[K] = Q[K] + 1, TA=(K==1)*3*ERL{1}) and the parameter K are defined in the Enter vertex dialog box shown in Fig. 5.39(b). Figure 5.40 shows the dialog boxes defining the Run→Enter edge, Enter→Load edge, and Enter→Enter edge.



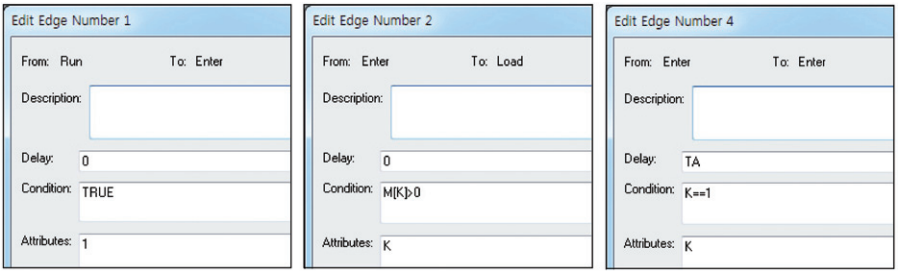
(a) (b)

Fig. 5.38. (a) SIGMA-generated event graph and (b) declaration of state variables.



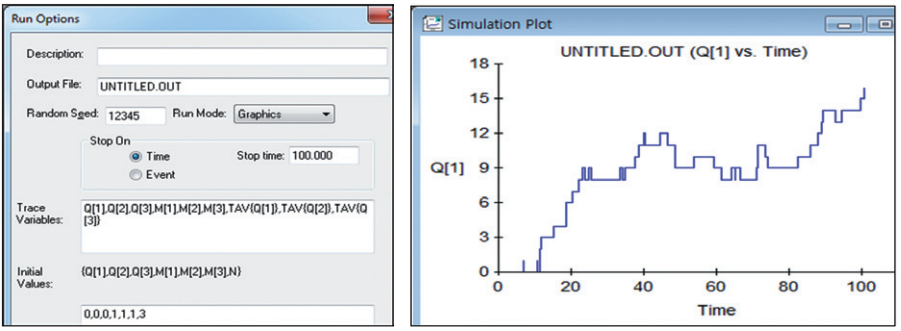
(a) (b)

Fig. 5.39. Defining (a) the Run vertex and (b) the Enter vertex.



(a) (b) (c)

Fig. 5.40. Defining the edges (a) Run→Enter, (b) Enter→Load, and (c) Enter →Enter.



(a) (b)

Fig. 5.41. (a) Run Options dialog box and (b) the simulation plot of Q[1].

Finally, the simulation experiment data are provided in the Run Options dialog box as depicted in Fig. 5.41(a). A simulation plot of Q[1] is given in Fig. 5.41(b).

5.6.4 Simulation of the Simple Job Shop with SIGMA

The PEG model of the job shop in Fig. 5.18 will be used as an example of a simulation using SIGMA. The example job shop has four single-machine stations (s = 0~3, i.e., M[s] = 1) and three job types (j = 0–2). A job arrives at every 12 minutes (=t_a) with job mix ratios of 26% for j = 0, 48% for j = 1, and 26% for j = 2. The routing sequences ({route(j,k)}) and processing times ({t_{jk}: j = job type, k = processing step}) are as given in Table 5.5. Note that the *processing step* is denoted by "k" (not "p") only when the routing sequence data and processing time data are as defined in Fig. 5.43 and Table 5.5.

Figure 5.42 depicts the routing sequence and processing times of the type-1 jobs. Let the transport delay time from station v to station w be denoted by d_{vw}; then, the net sojourn time of the type-1 job is expressed as $t_{10} + d_{01} + t_{11} + d_{13} + t_{12} + d_{31} + t_{13} + d_{12} + t_{14} + d_{24}$. The transport delay data are summarized in Table 5.6.

TABLE 5.5. Routing Sequence and Processing Times of the PEG Model Figure 5.18

	Step-0 (k = 0)		Step-1 (k = 1)		Step-2 (k = 2)		Step-3 (k = 3)		Step-4 (k = 4)	
Job (Ratio)	route (j,0)	t_{j0}	route (j,1)	t_{j1}	route (j,2)	t_{j2}	route (j,3)	t_{j3}	route (j,4)	t_{j4}
j = 0 (26%)	0	Exp (6)	1	Exp (5)	2	Exp (15)	3	Exp(8)	—	—
j = 1 (48%)	0	Exp (11)	1	Exp (4)	3	Exp (15)	1	Exp (6)	2	Exp (27)
j = 2 (26%)	1	Exp (7)	0	Exp (7)	2	Exp (18)		—		—

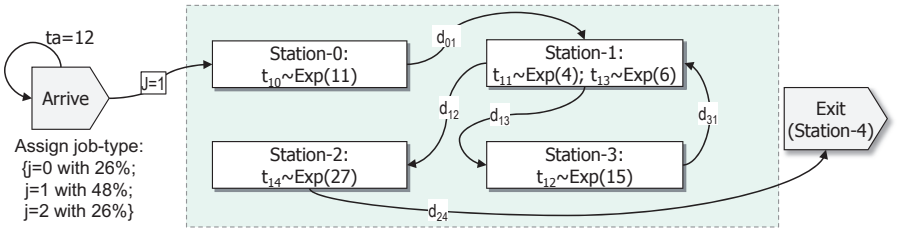


Fig. 5.42. Routing sequence and processing times of the type 1 job (j = 1).

TABLE 5.6. Transport Delay Data

		To Station (w)					Text file format of INPUTT.DAT				
delay[v, w] = d_{vw}		0	1	2	3	4					
From Station (v)	0	0	$d_{01} = 2$	$d_{02} = 4$	$d_{03} = 6$	$d_{04} = 2$	0	2	4	6	2
	1	$d_{10} = 6$	0	$d_{12} = 2$	$d_{13} = 4$	$d_{14} = 2$	6	0	2	4	2
	2	$d_{20} = 4$	$d_{21} = 6$	0	$d_{23} = 2$	$d_{24} = 2$	4	6	0	2	2
	3	$d_{30} = 2$	$d_{31} = 4$	$d_{32} = 6$	0	$d_{34} = 2$	2	4	6	0	2

5.6.4.1 Data Reading and Input Generation with SIGMA The configuration of a job shop is defined by a master data set. The important master data of the job shop are as follows:

1. Initial state of the queue in each station: Q[0] = 0, Q[1] = 0, Q[2] = 0, Q[3] = 0
2. Number of machines in each station: M[0] = 1, M[1] = 1, M[2] = 1, M[3] = 1
3. Initial job type of each station (machine): JT[0] = 0, JT[1] = 0, JT[2] = 0, JT[3] = 0
4. Routing sequence for each job type: route [J, K] as given in Table 5.5
5. Mean processing times for each job type: t[J, K] as given in Table 5.5

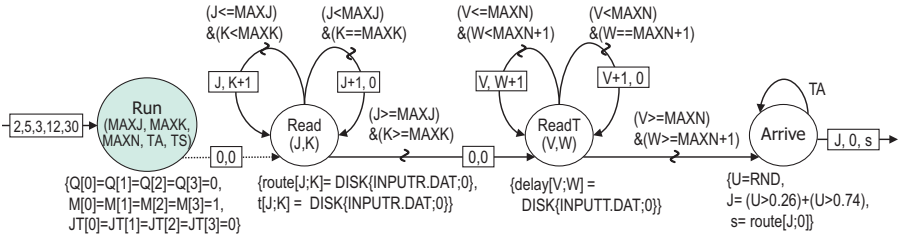


Fig. 5.43. SIGMA-compatible PEG model for data reading and input generation (K: processing step).

TABLE 5.7. INPUTR.DAT File of Routing and Mean Processing Time Data in Table 5.5

0	6	1	5	2	15	3	8	4	0	4	0
0	11	1	4	3	15	1	6	2	27	4	0
1	7	0	7	2	18	4	0	4	0	4	0

6. MAXJ = 2, MAXK = 5: maximum numbers of job types (J) and processing steps (K)
7. Transport delay times between stations: delay [V, W] as defined in Table 5.6
8. MAXN = 3: maximum number of stations (V, W)
9. Inter-arrival time and setup time: TA = 12, TS = 30

Figure 5.43 provides a SIGMA-compatible event graph for initializing the variables and reading the data, where (1) MAXJ = 2, MAXK = 5, MAXN = 3, TA = 12, and TS = 30 are received as parameter values; (2) Q[s], M[s], and JT[s] are initialized at the Run event; (3) route[J,K] and t[J,K] are read at the Read event; and (4) delay[V,W] is read at the ReadT event. The data reading function DISK {F; 0} is used to read the input data. Table 5.7 shows the input file (INPUTR.DAT) in which the routing data (route[J,K] = s) and mean processing time (t[J,K]) of Table 5.5 are provided. For example, the second line contains the following data: route[1,0] = 0, t[1,0] = 11, route[1,1] = 1, t[1,1] = 4, route[1,2] = 3, t[1,2] = 15, route[1,3] = 1, t[1,3] = 6, route[1,4] = 2, t[1,4] = 27, route[1,5] = 4, and t[1,5] = 0. Notice in Table 5.7 that s = 4 is used as a delimiter value indicating the end of a job process. The transport delay data of Table 5.6 are stored in the input file (INPUTT.DAT) in the same manner.

5.6.4.2 Building a SIGMA-Compatible PEG Model for a Job Shop Simulation The main part of the SIGMA-compatible PEG model for simulating the job shop operation is shown in Fig. 5.44. It is essentially the same as the PEG model shown in Fig. 5.18, but it has the following differences: (1) at the

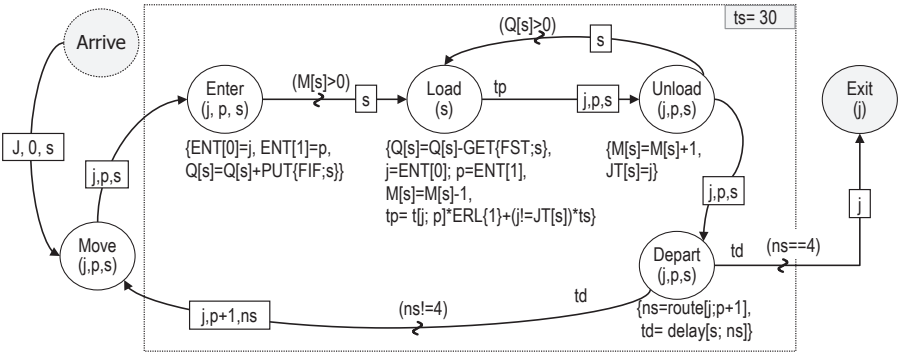


Fig. 5.44. SIGMA-compatible PEG model for the job shop operation (p: processing step).

Enter (j,p,s) event, the job type (j) and processing step (p) are assigned to each job entering the station (s); (2) the transport delay td = delay[s; ns] is defined at the Depart event; and (3) at the Load(s) event, the job (j, p) is retrieved and its processing time (tp) is computed as follows:

$$tp = t[j; p] * ERL\{1\} + (j! = JT[s])*TS.$$

The event transition table for the combined PEG model shown in Figs. 5.43 and 5.44 is given in Table 5.8. The constant data MAXJ (maximum number of job types), MAXK (maximum number of processing steps), MAXN (maximum number of stations), TA (deterministic inter-arrival time), and TS (deterministic setup time) are declared as parameters at the Run event. These values are specified in the Run Options dialog box. The job type (J) is obtained at the Arrive event using the RND function: U = RND, J = (U > 0.26) + (U > 0.74).

5.6.4.3 Building a SIGMA Program and Running the Simulation The first step of building a SIGMA program is to draw a SIGMA-generated event graph. Figure 5.45 shows the SIGMA-generated event graph model of the job shop in which there are 10 event vertices.

The second step is to declare all user-defined variables as state variables in the State Variable Editor dialog box, as shown in Fig. 5.46(a). There are 22 user-defined variables declared in the dialog box. The dimensions of the system variables ENT and RNK are also declared here. Among the declared variables are the three array variables (T, ROUTE, and DELAY) that constitute the master data of the job shop. Figure 5.46(b) shows the dialog box of the Run event, in which the job shop is initialized in the State Change(s) field and the constant variables are declared in the Parameter(s) field.

Finally, the simulation experiment data are provided in the Run Options dialog box as depicted in Fig. 5.47(a). A simulation plot of Q[2] is also shown in Fig. 5.47(b).

TABLE 5.8. Event Transition Table for the PEG Model Shown in Figures 5.43 and 5.44

No	Event	State Change	Edge	Condition	Delay	Parameter	Next Event
0	Run (MAXJ = 2, MAXK = 5, MAXN = 3, TA = 12, TS = 30)	Q[0] = 0, Q[1] = 0, Q[2] = 0, Q[3] = 0, M[0] = 1, M[1] = 1, M[2] = 1, M[3] = 1, JT[0] = 0, JT[1] = 0, JT[2] = 0, JT[3] = 0	1	True	0	0,0	Read(J,K)
1	Read(J,K)	ROUTE[J;K] = DISK{INPUTR.DAT;0}; T[J;K] = DISK{INPUTR.DAT;0}	1	(J<=MAXJ)&(K<MAXK)	0	J,K + 1	Read(J,K)
			2	(J<MAXJ)&(K==MAXK)	0	J + 1,0	Read(J,K)
			3	(J>=MAXJ)&(K>=MAXK)	0	0,0	ReadT(V,W)
2	ReadT(V,W)	DELAY[V;W] = DISK{INPUTT.DAT;0}	1	(V<=MAXN)&(W<MAXN+1)	0	V,W + 1	ReadT(V,W)
			2	(V<MAXN)&(W==MAXN+1)	0	V + 1,0	ReadT(V,W)
			3	(V>=MAXN)&(W>=MAXN+1)	0		Arrive
3	Arrive	U = RND, J = (U > 0.26) + (U > 0.74), S = ROUTE[J;0]	1	True	TA		Arrive
			2	True	0	J,0,S	Move(J,P,S)
4	Move(J,P,S)		1	True	0	J,P,S	Enter(J,P,S)
5	Enter(J,P,S)	ENT[0] = J, ENT[1] = P, Q[S] = Q[S] + PUT{FIF;S}	1	M[S] > 0	0	S	Load(S)
6	Load(S)	Q[S] = Q[S]-GET{FST;S}, J = ENT[0], P = ENT[1], M[S] = M[S] − 1, TP = T[J;P]*ERL{1} + (J! = JT[S])*TS	1	True	TP	J,P,S	Unload(J,P,S)
7	Unload(J,P,S)	M[S] = M[S] + 1, JT[S] = J	1	True	0	J,P,S	Depart(J,P,S)
			2	Q[S] > 0	0	S	Load(S)
8	Depart(J,P,S)	NS = ROUTE[J;P + 1]; TD = DELAY[S; NS]	1	NS==4	TD	J	Exit
			2	NS! = 4	TD	J,P + 1,NS	Move(J,P,S)

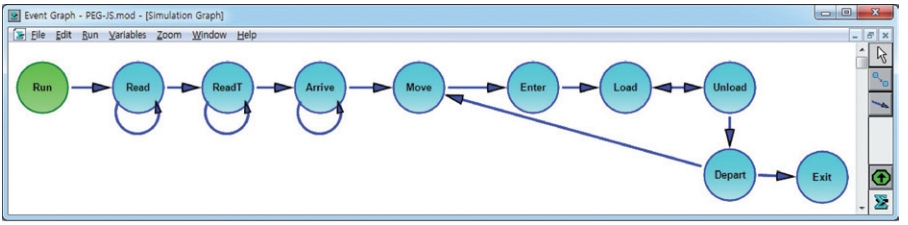
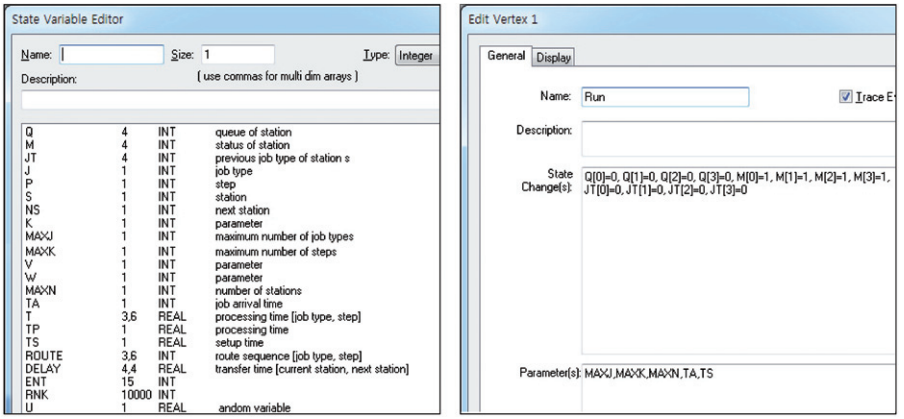
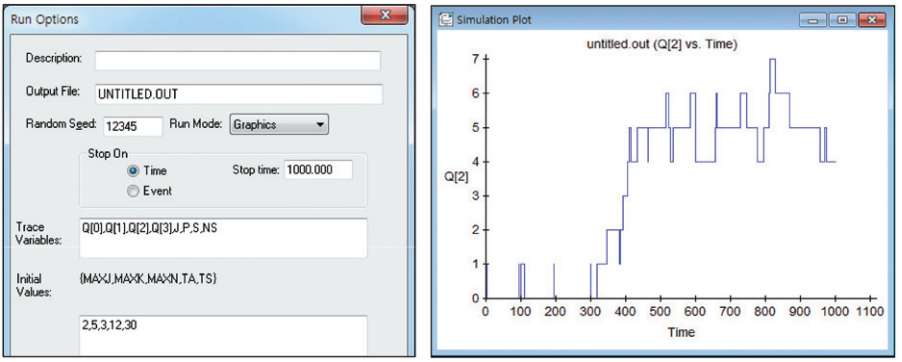


Fig. 5.45. SIGMA-generated event graph model of the job shop.



(a) (b)

Fig. 5.46. (a) Declaration of state variables and (b) defining the Run vertex.



(a) (b)

Fig. 5.47. (a) Run Options dialog box and (b) simulation plot of Q[2].

5.7 DEVELOPING YOUR OWN PARAMETERIZED EVENT GRAPH SIMULATOR

The process of developing your own simulator for a PEG model is the same as that for developing an (ordinary) event graph model, as described in Chapter 4. Namely, (1) the (pure) PEG model is converted to an augmented PEG model by adding the statistics variables and a statistics routine; (2) an event transition table is constructed from the augmented PEG model; (3) the initialize routine, event routines, and statistics routine are developed; and (4) the main program is obtained from the event graph simulator template shown in Fig. 4.56. The process of developing your own PEG simulator is described by using the three-stage tandem line model considered in Section 5.6.3 and the simple job shop model covered in Section 5.6.4. A complete list of C# codes for the tandem line simulator and the job shop simulator may be found in the official website of this book (http://VMS-technology.com/Book/EventGraphSimulator).

5.7.1 Tandem Line PEG Simulator

This section describes how to develop a PEG simulator for the tandem line shown in Fig. 5.36 in Section 5.6.3. Let us assume that we are interested in the average queue length at each stage of the tandem line.

Figure 5.48 shows an augmented PEG model of a three-stage tandem line for collecting the average queue length (AQL) statistics. The statistic variables introduced are the previous event time (Bef[k]) and the area under the queue-size curve (SumQ[k]) at stage k for k = 1–3. The average queue lengths (AQL[k]) for k = 1~3 are computed in the statistics routine. The event transition table of the PEG model is given in Table 5.9.

As in the case of the ordinary event graph simulator (see Chapter 4, Section 4.7), the initialize routine, event routines, and statistics routine of the augmented PEG model (Fig. 5.48 and Table 5.9) for collecting the AQL statistics are obtained easily, as follows:

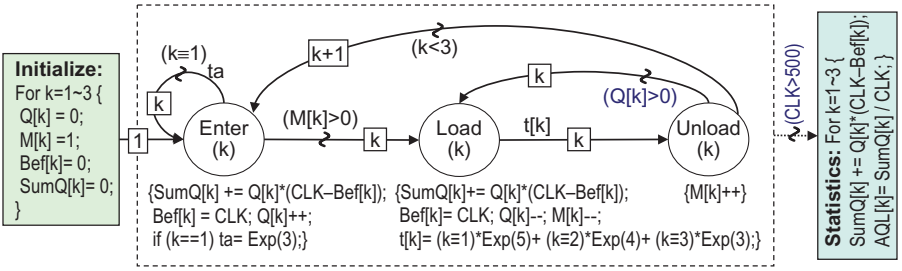


Fig. 5.48. Augmented PEG model of a three-stage tandem line for collecting AQL statistics.

TABLE 5.9. Event Transition Table for the PEG Model Shown in Figure 5.48

No	Event	State Change	Edge	Condition	Delay	Parameter	Next Event
0	Initialize	For k = 1~3 {Q[k] = 0; M[k] = 1; Bef[k] = 0; SumQ[k] = 0}	1	True	—	1	Enter(k)
1	Enter(k)	SumQ[k] += Q[k]* (CLK–Bef[k]); Bef[k] = CLK; Q[k] ++; if (k==1) ta = Exp(3);	1 2	k ≡ 1 M[k] > 0	ta 0	k k	Enter(k) Load(k)
2	Load(k)	SumQ[k] += Q[k]* (CLK–Bef[k]); Bef[k] = CLK; Q[k]––; M[k]––; t[k] = (k ≡ 1)* Exp(5) + (k ≡ 2)* Exp(4) + (k ≡ 3)* Exp(3);	1	True	t[k]	k	Unload(k)
3	Unload(k)	M[k]++;	1 2	Q[k] > 0 k < 3	0 0	k k + 1	Load(k) Enter(k)
4	Statistics	For k = 1~3 { SumQ[k] += Q[k]*(CLK–Bef[k]); AQL[k] = SumQ[k]/CLK; }					

```
Execute-Initialize-routine (Now) // Fig. 5.48 //
{  For k = 1 to 3 {Q[k] = 0; M[k] = 1; Bef[k] = 0; SumQ[k] = 0};
  Schedule-event (Enter, 1, Now);
}.

Execute-Enter-event-routine (k, Now) // Fig. 5.48 //
{  SumQ[k] += Q[k] * (Now – Bef[k]); Bef[k] = Now;
  Q[k] ++;
  If (k==1) Schedule-event (Enter, k, Now + Exp (3));
  If (M[k] > 0) Schedule-event (Load, k, Now);
}.

Execute-Load-event-routine (k, Now) // Fig. 5.48 //
{  SumQ[k] += Q[k] * (Now – Bef[k]); Bef[k] = Now;
   Q[k]––; M[k]––;
   t[k] = (k==1) * Exp(5) + (k==2) * Exp(4) + (k==3) * Exp(3);
  Schedule-event (Unload, k, Now + t[k]);
}.
```

```
Execute-Unload-event-routine (k, Now) // Fig. 5.48 //
{  M[k] ++;
   If (Q[k] > 0) Schedule-event (Load, k, Now);
   If (k < 3) Schedule-event (Enter, k + 1, Now);
}.

Execute-statistics-routine (Now) // Fig. 5.48 //
{  For k = 1~3 {SumQ[k] += Q[k] * (Now – Bef[k]); AQL[k]
   = SumQ[k] / Now;}
}.
```

With these event routines and initialize/statistics routines, the next event methodology algorithm for simulating the three-stage tandem line is realized as shown in Fig. 5.49.

If we are interested in the average sojourn time (AST) statistics, the augmented PEG model will be as shown in Fig. 5.50. The statistics variables required to collect the AST statistics are the job arrival time (AT), sum of sojourn times (SumT), and number of jobs passed through the third station (N). It should be noted that Q[k] is a list of real numbers (arrival times of jobs in the k^{th} stage).

```
Main-Program // PEG model in Figure 5.48 and Table 5.9 //
Begin
    CLK = 0;
    Execute-Initialize-routine (CLK);                    //  (1) Initialize
    While (CLK < 500) do {    // te = 500
        Retrieve-event (EVENT, k, TIME); CLK = TIME;  //  (2) Time-flow mechanism
        Case EVENT of {                                  //  (3) Execute event-routine
                Enter:      Execute-Enter-event-routine (k, CLK);
                Load:       Execute-Load-event-routine (k, CLK);
                Unload:     Execute-Unload-event-routine (k, CLK);
        } // end-of-case
    }; // end-of-while
    Execute-statistics-routine (CLK);                    //  (4) Output statistics
End
```

Fig. 5.49. Main program of the three-stage tandem line simulator for computing AQL.

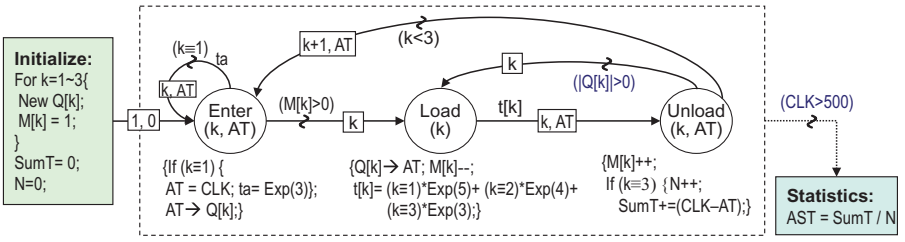


Fig. 5.50. Augmented event graph model for collecting AST statistics.

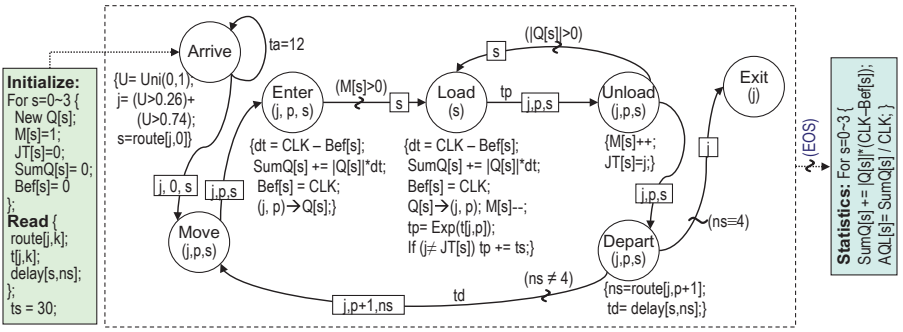


Fig. 5.51. Augmented PEG model of a simple job shop for collecting AQL statistics.

5.7.2 Simple Job Shop PEG Simulator

In this section, the simple job shop model covered in Section 5.5.2 (Fig. 5.18) and in Section 5.6.4 is adopted as a more general example of developing a PEG simulator. As described in Section 5.6.4, the simple job shop consists of four single-machine stations (s = 0, 1, 2, 3) and handles three job types (j = 0, 1, 2). The routing sequences and mean processing times are as given in Table 5.5; the transport time delays are as specified in Table 5.6. An augmented PEG model of this simple job shop is shown in Fig. 5.51, where EOS denotes the end-of-simulation condition.

In this example, the parameter variables are not equal among the events, which may cause difficulty in implementing the event routines. A simple method to avoid this difficulty is to use the same list of parameter variables for all event routines. In this case, the parameter list (j, p, s) is used for all event routines. For example, the event routines for the Arrive event and Enter event can be obtained as follows:

```
Execute-Arrive-event-routine (j, p, s, Now) // Fig. 5.51 //
{  U = Uni (0, 1); j = (U > 0.26) + (U > 0.74);
   s = route [j, 0];
   Schedule-event (Arrive, 0, 0, 0, Now +12);
   Schedule-event (Move, j, 0, s, Now);
}.

Execute-Enter-event-routine (j, p, s, Now) // Fig. 5.51 //
{  SumQ[s] += |Q[s]| * (Now – Bef[s]); Bef[s] = Now;
   (j, p) → Q[s];
   If (M[s] > 0) Schedule-event (Load, 0, 0, s, Now);
}.
```

TABLE 5.10. Event Transition Table for the PEG Model Shown in Figure 5.51

No	Event	State Change	Edge	Cond.	Delay	Parameter	Next Event
0	Initialize	For s = 0~3 {New Q[s]; M[s] = 1; JT[s] = SumQ[s] = Bef[s] = 0 }; Read { route[j, k], t[j, k]} for j = 0~2 and k = 0~5; Read {delay[j, k]} for j = 0~3 and k = 0~4; ts = 30;	1	True	0		Arrive
1	Arrive	U = Uni(0,1); j = (U > 0.26) + (U > 0.74); s = route[j,0];	1	True	12	—	Arrive
			2	True	0	j,0,s	Move(j,p,s)
2	Move(j,p,s)		1	True	0	j,p,s	Enter(j,p,s)
3	Enter(j,p,s)	SumQ[s] += |Q[s]|*(CLK–Bef[s]); Bef[s] = CLK; (j, p)→Q[s];	1	M[S] > 0	0	s	Load(s)
4	Load(s)	SumQ[s] += |Q[s]|*(CLK–Bef[s]); Bef[s] = CLK; Q[s]→(j, p); M[s]—; tp = Exp(t[j, p]); If (j ≠ JT[s]) tp += ts;	1	True	tp	j,p,s	Unload(j,p,s)
5	Unload(j,p,s)	M[s]++; JT[s] = j;	1	True	0	j,p,s	Depart(j,p,s)
			2	|Q[s]| > 0	0	s	Load(s)
6	Depart(j,p,s)	ns = route[j, p + 1]; td = delay[s, ns];	1	ns ! = 4	td	j,p + 1,ns	Move(j,p,s)
			2	ns ≡ 4	0	j	Exit(j)
7	Exit(j)		1				
8	Statistics	For s = 0~3 { SumQ[s] += |Q(s)|*(CLK-Bef[s]); AQL[s] = SumQ[s] / CLK;}					

Exercise 5.3. Write a main program (in a pseudocode form) for the PEG simulator that executes the augmented PEG model shown in Fig. 5.51.

5.8 REVIEW QUESTIONS

5.1. What are the two common cases for parameterizing an ordinary event graph?

5.2. What is the difference between a parameter variable and a parameter value?

5.3. What is the difference between a parameterized event transition table and an ordinary event transition table?

5.4. Compared to the ordinary event graph model, what are the additional elements in the algebraic structure of the PEG model?

5.5. What is the repeating pattern of event nodes in a limited buffer tandem line model?

5.6. What is a simple job shop?

5.7. What are the state variables in the event graph model of a simple job shop?

5.8. Where do you declare the state variables in SIGMA? Where do you initialize them?

CHAPTER 6

Introduction to Activity-Based Modeling and Simulation

In all things there is a law of cycles.

—Publius C. Tacitus

6.1 INTRODUCTION

Our lives are full of activities: It is through activities that something meaningful is achieved. An activity always involves at least one *actor*. In our definition of a discrete-event system, it is often the case that the actor is a resource and the target of an activity is a transient entity to be served or processed by the resource. In a machine shop, the resource is a *machine* and the transient entity is a *job*. The outcome of one activity may trigger other activities. If we can identify the relationships among the activities, we can better understand the present situation and may be able to predict future situations.

Thus, activity-based modeling is a natural way to represent our knowledge of a system. When we describe a real-life dynamic situation, we naturally follow a sequence of steps, i.e., activities that are involved in the situation. In activity-based modeling, the dynamics of the system are represented using an *activity cycle diagram* (ACD), which is a network model of the logical relationships between the activities. An ACD is a formal model that can be executed with a well-defined algorithm.

The single server system introduced in Chapter 2 (Figs. 2.6 and 2.8) is shown in Fig. 6.1: A job arrives at the system as a result of a Create activity performed by the Job Creator, and it is served by the Machine through the Process activity. Each job goes through the system in the following sequence: After being created for t_a minutes, it stays in the passive resource Buffer until it can be loaded onto the Machine, and then it is processed by the active resource

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.

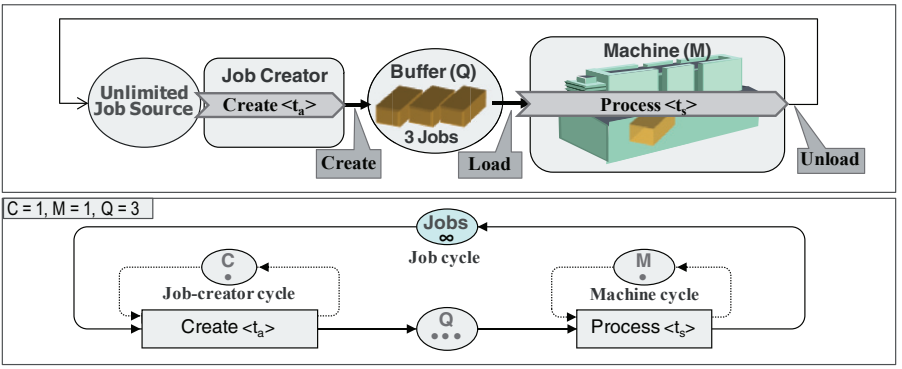


Fig. 6.1. Reference model and ACD model of a single server system.

Machine for t_s minutes. The system state is defined by the states of the resources: Job Creator (C = 1 if idle, C = 0 if busy), Machine (M = 1 if idle, M = 0 if busy), and Buffer (Q = number of jobs in the Buffer).

The purpose of this chapter is to explain the essential features of activity-based modeling and simulation (M&S) using an ACD. The advanced features and special applications of the activity-based M&S will be covered in Part III (Chapter 10) of this book. After studying this chapter, you should be able to do the following:

1. Construct an *activity transition table* for an ACD model
2. Build ACD models with various "template systems" such as a flexible multi-server system, limited buffer tandem line, nonstationary Poisson process, batched service multi-server line, and inspection-repair line
3. Build ACD models for simple service systems such as restaurants, gas stations, coffee shops, and car repair shops
4. Build an ACD model for a PERT (program evaluation and review technique) system
5. Build an ACD model for a conveyor-driven serial line
6. Construct parameterized ACD models of n-stage tandem lines and of a job shop
7. Simulate various types of ACD models using the formal ACD simulator ACE®

The remainder of this chapter is organized as follows. The execution rules and specifications of ACD are described in Section 6.2, and ACD modeling templates and examples are given in Section 6.3 and Section 6.4, respectively. The definition and execution rules of a P-ACD as well as its application to modeling tandem lines and job shops are given in Section 6.5. How to simulate

ACD models using the formal ACD simulator ACE® is explained in detail in the last section.

6.2 DEFINITIONS AND SPECIFICATIONS OF AN ACTIVITY CYCLE DIAGRAM

The core idea of activity-based M&S was conceived by Tocher in 1957 [Hollocks 2008] when he was investigating the *congestion control problem* at the United Steels in the United Kingdom. Tocher argued that "in more complex plants, in which there is a multiplicity of possible routes for the steel through the plant, it is possible to minimize congestion and maximize the rate of flow by a (simulation-based) scheduling procedure" [Tocher 1960, p.50].

Tocher used a flow diagram of activities in modeling the dynamic behavior of the steel plant (Fig. 6.2). "The plant is regarded as a set of machines, each with a set of states, progressing as time unfolds through states that change only at discrete events. At any moment of time, components are grouped together in activities, which endure for a sampled time, and then become free, after a possible change of state, to regroup with other components in further activities" [Tocher 1960, p.59].

The *activity flow diagram* shown in Fig. 6.2 later evolved into the ACD where an activity node is denoted by a rectangle and a queue node (or passive state node) by a circle [Carrie 1988]. This version of the ACD is often referred to as the *classical ACD*. However, it has transpired that the classical ACD has some inherent limitations in handling complex systems [Hlupic and Paul 1994]. In order to enhance the modeling power, concepts of a hierarchical ACD [Kienbaum and Paul 1994] and an extended ACD [Martinez 2001] have been proposed. More recently, a formal specification of an *extended ACD* has been developed by the authors of this book [Kang and Choi 2011].

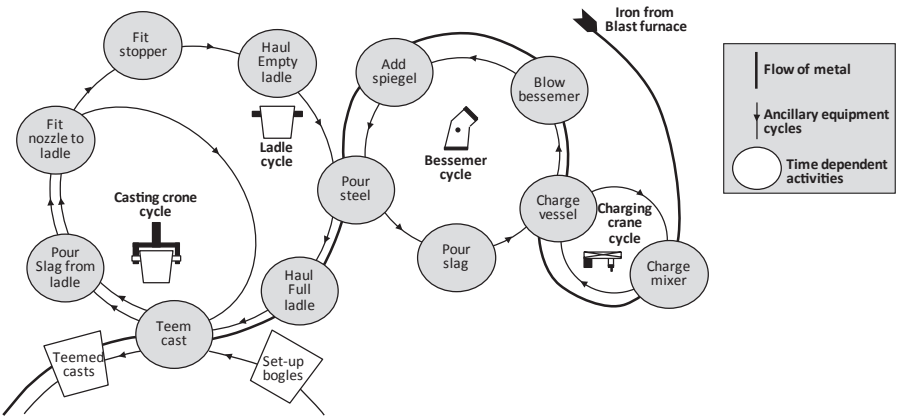


Fig. 6.2. Flow diagram of Acid Bessemer steel-making process [Tocher 1960].

6.2.1 Definitions of an ACD

In the classical ACD, an activity typically represents the interaction between an entity and active resources. (Note that in this book the term *entity* is used to denote a transient object that arrives at the system and eventually leaves the system, while in other ACD literature this term includes resident resources as well.) An entity or an active resource is either in a passive state called a *queue* or in an active state called an *activity*. Queue nodes and activity nodes are connected by arcs.

Figure 6.3 presents a classical ACD model for a single server system with a setup operator. There are two types of entities (Job and Break) and four types of resources (Job Creator, Machine, Operator, and Break Generator). The basic conventions for drawing an ACD are as follows:

1. Each entity and resource has an activity cycle.
2. Each cycle consists of activities and queues.
3. Activities and queues alternate in a cycle.
4. Activities are depicted by rectangles and queues by circles.
5. A cycle is closed.

The dynamics of an ACD model are described in terms of token variables. The value of a *token variable* represents either the state of an active resource or the number of entities in a passive resource like a buffer. In the ACD model shown in Fig. 6.3, the token (denoted by a dot "•") in the Machine Cycle indicates that the machine is in the Hold state, which is specified as Hold ≡ 1. The numeric value of a token variable in a queue is specified in a *pair of chevrons* <>. For example, the number of tokens in the Q queue is specified as <3>, indicating that there are three jobs in Q. When the value is zero, the number may be omitted.

For example, the number of tokens in the Ready queue is not specified because it is zero. A vector representing the numbers of tokens in the queues is called a *marking*. For the ACD model shown in Fig. 6.3, the marking $\boldsymbol{M}$ is a vector of token variables {C, Q, Hold, Ready, Wait, B, G, BG} and the initial marking, $\boldsymbol{M}_0 = \{1, 3, 1, 0, 1, 0, 1, 1\}$, defines the initial state of the system.

As can be seen in Fig. 6.3, the *duration of an activity* is also specified in a pair of chevrons <>. For example, the duration of the Process activity is speci-

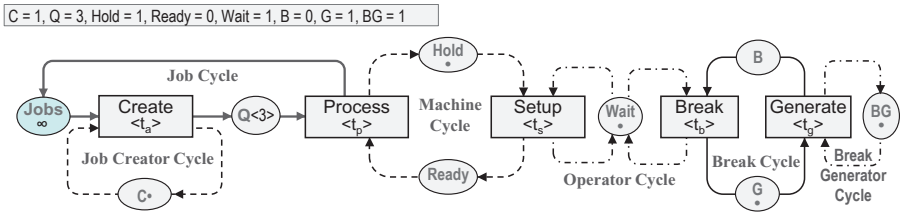


Fig. 6.3. Classical ACD for a single server system with a setup operator.

fied as $<t_p>$ in the activity node. There are two forms of activity cycles: (1) an entity activity cycle for an entity that has a definite sequence and (2) a resource activity cycle for an active resource that may perform one or more different activity cycles in any sequence. In Fig. 6.3, for example, the Operator Cycle, which is a resource activity cycle has two cycles to choose. In this case, it is implicitly assumed that a rule exists for choosing one of the two.

6.2.2 Execution Rules and Tabular Specifications of an ACD

An important extension to the classical ACD is the addition of arc conditions and arc multiplicities, which is often referred to as an *extended ACD*. An *arc condition* is a Boolean expression that must be true in order for the arc to be enabled, and an *arc multiplicity* represents the number of tokens passing through the arc when the enabled activity is executed. An arc condition and an arc multiplicity define the arc attributes of the arc. When arc attributes are not specified, by default, the arc condition is true and the arc multiplicity is 1.

Figure 6.4 shows a portion of an extended ACD in which (1) Q1 is an input queue, (2) c1 is an input arc condition, (3) m1 is an input arc multiplicity, (4) c2 is an output arc condition, (5) m2 is an output arc multiplicity, (6) Q2 is an output queue, and (7) A2 and A3 are *influenced activities* of activity A1 (because the execution of A1 directly influences the start of A2 and A3). Queues S1, S2, and S3 represent the numbers of idle resources required to perform activities A1, A2 and A3, respectively. A2 has a higher priority over A3 because A3 is only enabled when the A2 resource is busy ($S2 \equiv 0$).

In the following, the execution rules of an extended ACD are described for the A1 activity in Fig. 6.4. An activity is confined by two events: an activity-begin event and an activity-end event. Once an activity-begin event occurs, the activity-end event is bound to occur after the time delay of the activity duration. Thus, the activity-end event is called a *bound-to-occur event* (BTO event).

The At-begin execution rules of activity A1 in Fig. 6.4 are as follows: "If the input arc condition (c1) is true and the number of tokens in the input queue Q1 is at least its arc multiplicity ($Q1 \geq m1 > 0$ or $Q1 > m1 \equiv 0$) ***and*** if there is at least one token in the queue S1 ($S1 > 0$), then (1) the A1 activity will begin after de-queuing m1 tokens from Q1 ($Q1 = Q1 - m1$) and one token from S1 and (2) its BTO event is scheduled to occur after the activity duration (t_1)." Similarly, the At-end execution rules are expressed as "If the output arc

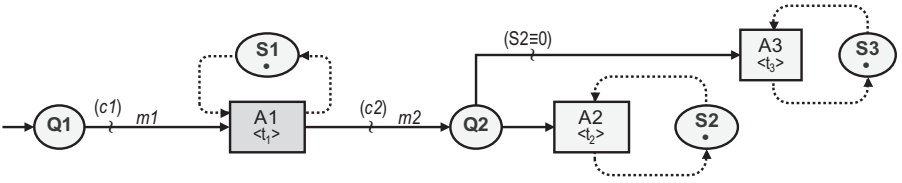


Fig. 6.4. Illustration of the arc attributes in an extended ACD.

condition (c2) is true, then (1) m2 tokens are created and en-queued into the output queue Q2 and (2) a token is returned to queue S1. Then, the influenced activity A2 is examined first for execution, and if A2 is not ready for execution (i.e. S2 ≡ 0), A3 is considered for execution."

An activity transition table is a formal specification of an ACD in a tabular form that defines the properties of the ACD. It specifies At-begin condition, At-begin action, BTO-event time, and BTO-event name of each activity. Here, updating state variables is referred to as an *action*. The table also specifies At-end condition, At-end action, and Influenced Activity for each output arc of the activity. Table 6.1 is an activity transition table of the ACD shown in Fig. 6.4. This table may also be regarded as an *execution rules table* because the execution rules are summarized concisely within it. In addition, the initial marking and enabled activities (i.e., activities whose At-begin conditions are true) are specified in the Initialize row of the table.

Table 6.2 gives another illustration of an activity transition table for the single server system ACD shown in Fig. 6.1. The single server system ACD consists of two activities (Create, Process) and four queues (C, Q, M, Jobs). Since the Jobs queue denotes the outside world in which there are an infinite number of jobs (i.e., the number of tokens is ∞), it is disregarded when interpreting the ACD.

Exercise 6.1. Construct an activity transition table of the classical ACD in Fig. 6.3.

6.2.3 Algebraic Specifications of an ACD

As mentioned in Chapter 2 (Section 2.3.3), a classical ACD is essentially a timed Petri net. A Petri net is a bipartite directed graph, so it is a classical ACD. An activity node in the ACD corresponds to a (timed) transition in a Petri net, and a queue node relates directly to a place. A Petri net consists of a finite set of places and a finite set of transitions, and an arc runs from a place to a transition, or vice versa. The places from which an arc runs to a transition are called *input places* of the transition; the places to which arcs run from a transition are *output places*. The places in a Petri net may contain a number of tokens. Any distribution of tokens over the places will represent a state of the net called a *marking* [Peterson 1981]. Due to their common structure, the algebraic specification of an ACD is derived from a Petri net. Further discussion on the Petri net and its relationship to the ACD is presented in Chapter 10.

An ACD is a bipartite directed graph consisting of a set of activity nodes (A) and a set of queue nodes (Q). The arcs connecting the activities from the input queues are defined in the input function (I), and those connecting the output queues from the activities are defined in the output function (O). Associated with each activity node ($a \in A$) is a time delay ($\tau_a \in$ T), and a number of tokens ($\mu_q \in \boldsymbol{\mu}$) is specified for each queue node ($q \in Q$) with their initial marking $\boldsymbol{\mu}_0$. Thus, a classical ACD model **M** can be defined as 7-tuple structure, as follows:

TABLE 6.1. Activity Transition Table of the ACD in Figure 6.4

		At-begin		BTO-event		At-end			
No	Activity	Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
1	A1	(c1) & (Q1 ≥ m1) & (S1 > 0)	Q1 = Q1 − m1; S1—;	t_1	EventA1	1 2	True (c2)	S1++; Q2 = Q2 + m2;	A1 A2, A3
2	A2	(Q2 > 0) & (S2 > 0)	Q2—; S2—;	t_2	EventA2	1	True	S2++;	A2
3	A3	(S2 ≡ 0) & (Q2 > 0) & (S3 > 0)	Q2—; S3—;	t_3	EventA3	1	True	S3++;	A3
Initialize		*Initial Marking* = {S1 = S2 = S3 = 1, Q1 = Q2 = 0, . . .}; *Enabled Activities* = { }							

TABLE 6.2. Activity Transition Table of the Single Server System ACD in Figure 6.1

		At-begin		BTO-event		At-end			
No	Activity	Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
1	Create	(C > 0)	C−−;	t_a	Created	1	True	C++;	Create
						2	True	Q++;	Process
2	Process	(Q > 0) & (M > 0)	Q−−; M−−;	t_s	Processed	1	True	M++;	Process
Initialize		*Initial Marking* = {C = 1, M = 1, Q = 3}; *Enabled Activities* = {Create}							

$\boldsymbol{M}$ = <A, Q, I, O, T, $\boldsymbol{\mu}$, $\boldsymbol{\mu_0}$>, where

$A = \{a_1, a_2 \cdots a_n\}$ is the finite set of activities;

$Q = \{q_1, q_2 \cdots q_m\}$ is the finite set of queues;

$I = \{i_a \subseteq Q \mid a \in A\}$ is the input function, which is mapped from a set of queues to an activity;

$O = \{o_a \subseteq Q \mid a \in A\}$ is the output function, which is mapped from an activity to a set of queues;

$T = \{\tau_a \in R_0^+ \mid a \in A\}$ is the time delay function;

$\boldsymbol{\mu} = \{\mu_q \in N_0^+ \mid q \in Q\}$ is the finite set of the number of tokens for each queue; and,

$\boldsymbol{\mu_0} = \{\mu_1, \mu_2 \cdots \mu_m\}$ is the finite set of initial number of tokens for each queue.

As an example, the single server system ACD in Fig. 6.1 may be specified as follows:

$\boldsymbol{M} = (A, Q, I, O, T, \boldsymbol{\mu}, \boldsymbol{\mu_0})$, where

$A = \{a_1, a_2\} = \{\text{Create, Process}\}$

$Q = \{q_1, q_2, q_3, q_4\} = \{\text{Jobs, C, Q, M}\}$

$I(a_1) = \{q_1, q_2\}$, $I(a_2) = \{q_3, q_4\}$

$O(a_1) = \{q_2, q_3\}$, $O(a_2) = \{q_1, q_4\}$

$T(a_1) = t_a$, $T(a_2) = t_s$

$\boldsymbol{\mu} = \{\mu_1, \mu_2, \mu_3, \mu_4\}$

$\boldsymbol{\mu_0} = \{\mu_1 = \infty, \mu_2 = 1, \mu_3 = 3, \mu_4 = 1\}$

An algebraic structure of an extended ACD is defined similarly [Kang and Choi 2011].

6.3 ACTIVITY CYCLE DIAGRAM MODELING TEMPLATES

The single server system ACD model shown in Fig. 6.1 is the baseline ACD model in which a resource processes one entity at a time and the buffer has

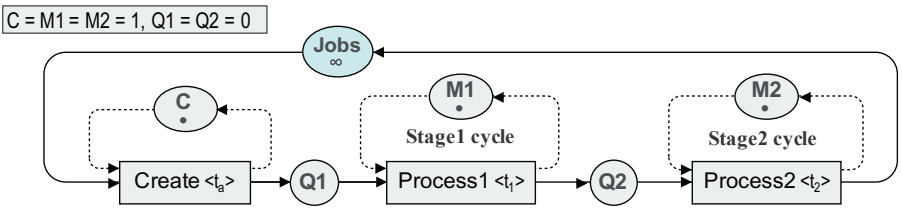


Fig. 6.5. ACD model of a two-stage tandem line.

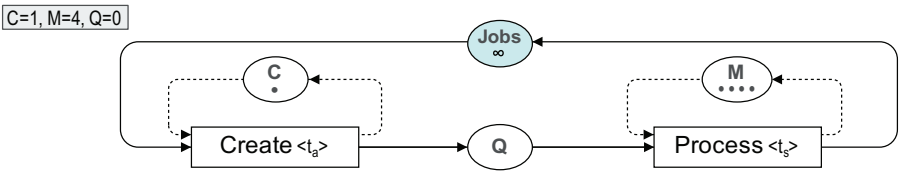


Fig. 6.6. ACD model of a fixed multi-server system with four identical servers.

an infinite capacity. This baseline model can be embellished to cover more complex situations. Most ACD modeling templates covered in this section are taken from the event graph modeling templates introduced in Section 4.4 of Chapter 4. The templates involving event canceling are covered in Chapter 10. The ACD modeling templates presented here can be used as building blocks for modeling large systems.

The single server ACD model of Fig. 6.1 is easily extended to a two-stage tandem line ACD model by adding one more stage (i.e., Process activity) as shown in Fig. 6.5. The two-stage tandem line ACD model has two Process activities (Process1 and Process2) with two time delays (t_1 and t_2, respectively).

Exercise 6.2. Construct an ACD model of a three-stage tandem line.

6.3.1 ACD Template for Flexible Multi-Server System Modeling

The single server system depicted in Fig. 6.1 consists of a single server and a buffer with an unlimited capacity. If there are two or more identical servers in the system, it is a multiple server system. Figure 6.6 shows the ACD model of a fixed multi-server system with four identical servers. The initial marking is $\mu_0 = \{C = 1, M = 4, Q = 0\}$.

Consider the case in which the number of servers varies over time, which is called a *flexible multi-server* system. Let N(t) denote the number of servers at time t, then the ACD model of the flexible multi-server system becomes the one shown in Fig. 6.7. CLK denotes the current simulation clock time.

An activity transition table for the flexible multi-server ACD in Fig. 6.7 is given in Table 6.3. At every At-begin execution time, the state variable D, which denotes the change in the number of servers, is updated and the Process activity is started if its At-begin Condition ((M > D) & (Q > 0)) is true. The

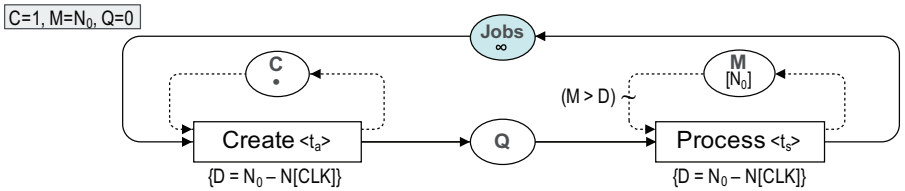


Fig. 6.7. ACD model of a flexible multi-server system.

TABLE 6.3. Activity Transition Table of the Flexible Multi-Server ACD Given in Figure 6.7

		At-begin		BTO-event		At-end			
No	Activity	Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
1	Create	(C > 0)	C—; D = N_0 – N[CLK];	t_a	Created	1 2	True True	C++; Q++;	Create Process
2	Process	(M > D) & (Q > 0)	M—; Q—; D = N_0 – N[CLK]	t_s	Processed	1	True	M++;	Process
Initialize		*Initial Marking* = {C = 1, M = N_0, Q = 0}; *Enabled Activities* = {Create}							

initial marking is {C = 1, M = N_0, Q = 0}. As a convention, the state variable updates are specified beneath the activity node in the ACD and are described in the At-begin Action entry of the activity transition table.

6.3.2 ACD Template for Limited Buffer Tandem Line Modeling

As discussed in Chapter 4 (Section 4.4), balking may occur if the waiting space for the arriving jobs becomes full, and blocking may occur if the unloading space of a machine is full. In activity-based modeling, the limited buffer problem is managed using Kanbans, which is the work-in-progress (WIP) control mechanism used in just-in-time (JIT) production or lean manufacturing. A Kanban is a kind of entrance ticket that is issued to an incoming entity and is collected when the entity leaves the system.

Consider the unlimited buffer two-stage tandem line model shown in Fig. 6.5. Let's assume that the buffer capacity in front of Stage1 is three (K1 = 3) and the buffer capacity between Stage1 and Stage2 is four (K2 = 4). The capacity of the waiting space or buffer is represented by the number of tokens (or Kanbans) in queues K1 and K2. The ACD model of the two-stage limited buffer tandem line is shown in Fig. 6.8. The Enter activity is allowed to start only when at least one token is available in K1. The Unload1 activity is prohibited (i.e., blocked) when there are no tokens in K2.

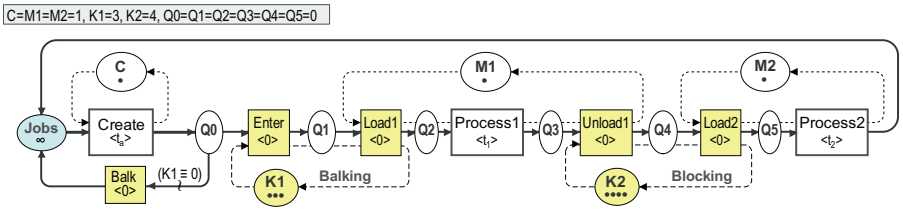


Fig. 6.8. Limited buffer tandem line modeling (balking and blocking).

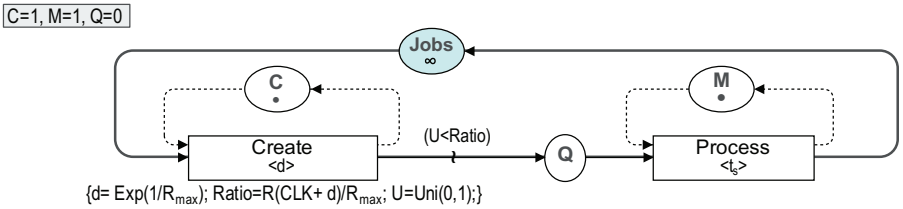


Fig. 6.9. ACD model of a single server system with nonstationary arrival rates.

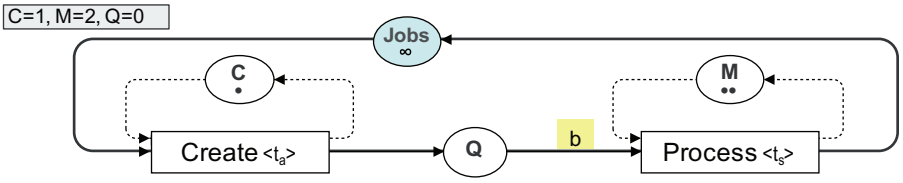


Fig. 6.10. ACD model of a batched service multi-server system.

6.3.3 ACD Template for Nonstationary Arrival Process

The thinning method of generating inter-arrival times under a nonstationary Poisson process was explained in Chapter 3 (see Fig. 3.4 in Section 3.4.3), and an event graph model of a single server system subject to fluctuating arrival rates was given in Chapter 4 (Fig. 4.9 in Section 4.4.1).

Figure 6.9 shows an ACD model of a single server system with fluctuating inter-arrival times sampled from a nonstationary Poisson process. The time delay (d) of the next Create activity is computed with the maximum arrival rate (R_{max}), and then the generated job is sent to queue Q only when it passes the thinning test (i.e., U < Ratio), where CLK is a function returning the current simulation clock, R() is the arrival rate function, Exp() is an exponential random variate generation function, and Uni(0,1) is a standard uniform random number generator.

6.3.4 ACD Template for Batched Service Modeling

An ACD of a batched service multi-server is shown in Fig. 6.10, where the multiplicity of the (directed) arc from queue Q to activity Process is set to

TABLE 6.4. Activity Transition Table for the ACD in Figure 6.10

		At-begin		BTO-event		At-end			
No	Activity	Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
1	Create	(C > 0)	C––;	t_a	Created	1	True	C++;	Create
						2	True	Q++;	Process
2	Process	(Q ≥ b) & (M > 0)	Q = Q–b; M––;	t_s	Processed	1	True	M++;	Process
Initialize		*Initial Marking* = {C = 1, M = 2, Q = 0}; *Enabled Activities* = {Create}							

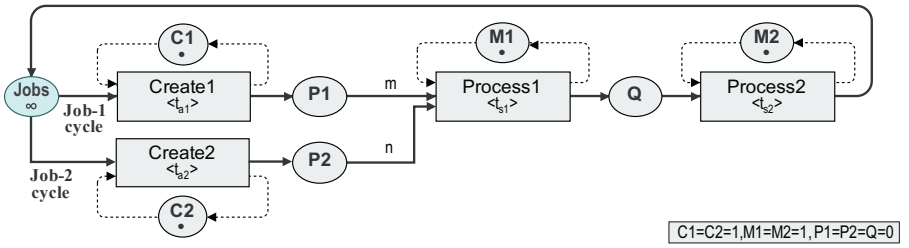


Fig. 6.11. ACD model of a joining operation line.

batch size b. In general, there are a maximum number (b) and a minimum number (a) of jobs that can be processed at one time, which is denoted as $a \le J \le b$, where J is the actual number of jobs in a batched service. If a = b, it is a full batched service; if a < b, it is a partial batched service. The activity transition table of the ACD model for a full batched service system is given in Table 6.4.

Exercise 6.3. Revise the ACD in Fig. 6.10 to create a partial batched service model.

6.3.5 ACD Template for Joining Operation Modeling

An ACD model for a production line that joins two parts (Job-1, Job-2) is shown in Fig. 6.11, where the Job-1 part is treated as the main entity. When m parts of Job-1 and n parts of Job-2 are joined, they are specified as an arc multiplicity.

6.3.6 ACD Template for Probabilistic Branching Modeling

Figure 6.12 presents an ACD model for probabilistic branching where 90% of the jobs pass inspection and go to queue P for the next processing. The remaining jobs are moved to queue S for the scrapping operation. A probabilistic branching is modeled as an arc condition involving a uniform random number (U). The activity transition table of this ACD model is given in Table 6.5.

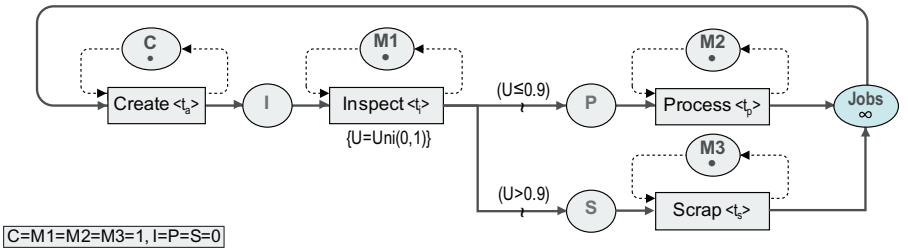


Fig. 6.12. ACD model for probabilistic branching.

TABLE 6.5. Activity Transition Table for the ACD in Figure 6.12

		At-begin		BTO-event		At-end			
No	Activity	Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
1	Create	(C > 0)	C−−;	t_a	Created	1	True	C++;	Create
						2	True	I++;	Inspect
2	Inspect	(I > 0) & (M1 > 0)	I−−; M1−−; U = Uni(0, 1);	t_i	Inspected	1	True	M1++;	Inspect
						2	(U ≤ 0.9)	P++;	Process
						3	(U > 0.9)	S++;	Scrap
3	Process	(P > 0) & (M2 > 0)	P−−; M2−−;	t_p	Processed	1	True	M2++;	Process
4	Scrap	(S > 0) & (M3 > 0)	S−−; M3−−;	t_s	Scraped	1	True	M3++;	Scrap
Initialize		*Initial Marking* = {C = M1 = M2 = M3 = 1, I = P = S = 0}; *Enabled Activities* = {Create}							

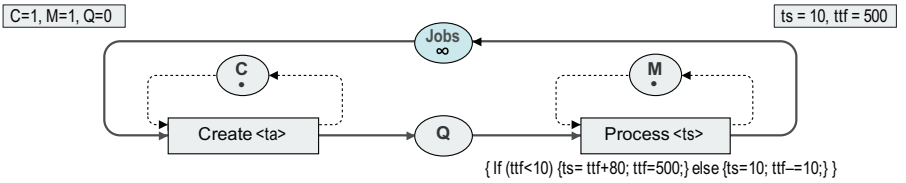


Fig. 6.13. ACD for a single server system with machine failure.

6.3.7 ACD Template for Resource Failure Modeling

In Chapter 4 (Section 4.4.1), two cases of resource failure event graph models were considered: a failure model where the server may fail even when it is idle, and a model where a failure is only allowed when the server is busy. In this section, only the second case of resource failure is modeled using an ACD. The ACD modeling of the first case will be discussed in Chapter 10 (for which we need a *canceling arc*).

Figure 6.13 presents an ACD model of the single server system with machine failure where a failure is only allowed when the server is busy and the

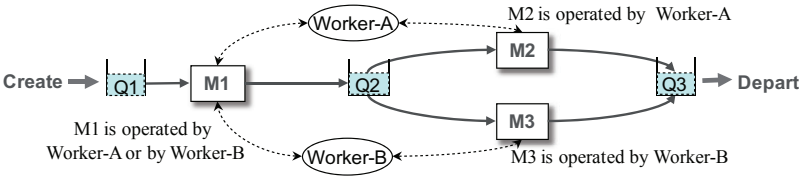


Fig. 6.14. Reference model of a worker-operated two-stage tandem line.

TABLE 6.6. Activity Transition Table of the ACD Model of Figure 6.13

		At-begin		BTO-event		At-end			
No	Activity	Condition	Action	Time	Name	Arc	Condition	Action	Infl. Act.
1	Create	(C > 0)	C—;	ta	Created	1	True	C++;	Create
						2	True	Q++;	Process
2	Process	(Q > 0) & (M > 0)	Q—; M—; If (ttf < 10) {ts = ttf + 80; ttf = 500;} else {ts = 10; ttf –= 10;}	ts	Processed	1	True	M++;	Process
Initialize		*Initial Marking* = {C = 1, M = 1, Q = 0}; *Variables* = {ts = 10, ttf = 500}; *Enabled Activities* = {Create};							

interrupted job is discarded. Its event graph model was given in Fig. 4.14 in Chapter 4, where the service time was 10 (ts = 10), repair time was 80 (tr = 80), and the value of remaining time-to-failure (ttf) was initially set to 500. As with the event graph model, the variable ttf is regarded as a state variable. The activity transition table of this ACD model is given in Table 6.6.

6.4 ACTIVITY-BASED MODELING EXAMPLES

System modeling is an art that may only be mastered by learning the best practices and internalizing them through relentless practices. This section presents basic ACD modeling examples including a worker-operated tandem line, an inspection-repair line, a restaurant, a simple service station, a car repair shop, a project management system, and a conveyor-driven serial line. We use the terms *serial* and *tandem* interchangeably. More advanced examples involving parameterized ACDs will be presented in Section 6.5 and in Chapter 10.

6.4.1 Activity-Based Modeling of a Worker-Operated Tandem Line

Figure 6.14 depicts a reference model of a worker-operated tandem line. The first operation is performed on machine M1, which is operated by Worker-A

or Worker-B; the second operation is performed on either of the two identical machines M2 and M3. M2 is operated by Worker-A and M3 by Worker-B.

If the machines are operated unattended, the ACD model of the line would be that of Fig. 6.5 with M2 = 2 (and the Depart activity is added). One rule for building an ACD model is that "operations involving different resources are treated as different activities." Thus, the process activity for M1 is divided into two activities: (1) Process1a performed by Worker-A and (2) Process1b performed by Worker-B. The resulting ACD model is given in Fig. 6.15. Worker-A is given a higher priority over Worker-B for processing a job on M1, and M2 has a higher priority over M3 for the second operation. However, the Worker-A (Worker-B) cycle does not have a specified sequence, and a dispatching rule may be required in order to choose between Process1a (Process1b) and Process2 (Process3).

6.4.2 Activity-Based Modeling of an Inspection-Repair Line

Figure 6.16 is a reference model of an inspection-repair line in which two types of jobs are inspected on a single inspection machine (I) and are repaired by a single repair machine (R). Another rule for building an ACD model is that "jobs that follow separate paths are treated as separate entities." Thus, by using this rule and employing the probabilistic branching template (Fig. 6.12), an ACD of the inspection-repair line is obtained as given in Fig. 6.17. In Fig. 6.17, note that the arcs from the final activity nodes (i.e., Delivery-1, Delivery-2, and Scrap) to the source queue nodes (J1 and J2) are omitted for brevity.

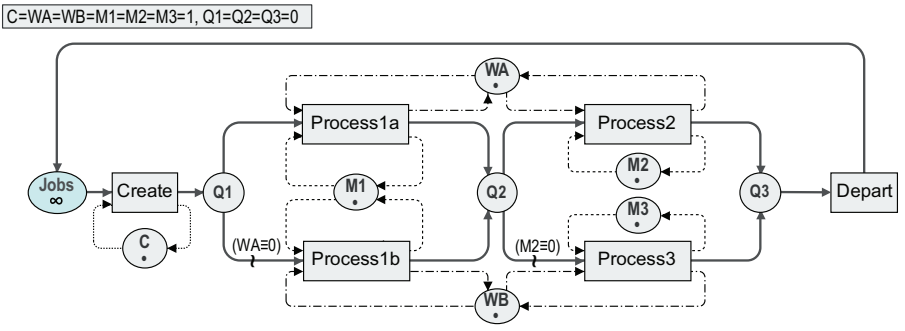


Fig. 6.15. ACD model of the worker-operated tandem line in Fig. 6.14.

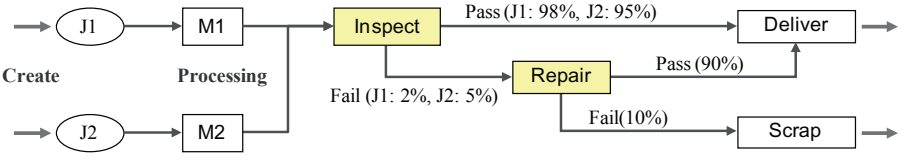


Fig. 6.16. Reference model of an inspection-repair line.

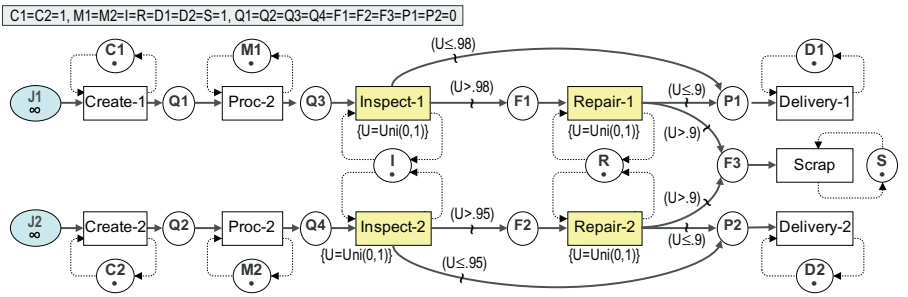


Fig. 6.17. ACD model of the inspection-repair line in Fig. 6.16.

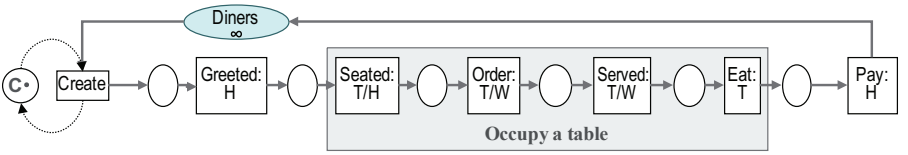


Fig. 6.18. Lifecycle of the entity in a restaurant model.

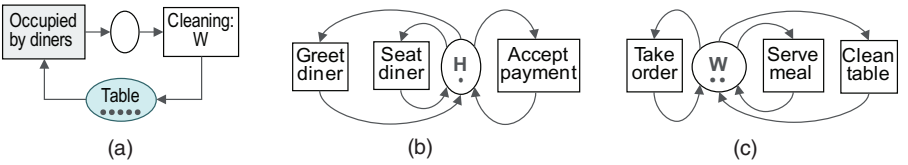


Fig. 6.19. Lifecycle of the (a) table, (b) head waiter, and (c) waiters.

6.4.3 Activity-Based Modeling of a Restaurant

Consider a restaurant served by a head waiter (H) and two waiters (W) [Activity cycle diagram 2012]. There are five tables (T) in the restaurant. A batch of diners coming together is regarded as an entity. The life cycle of an entity (customer batch) is Arrive → Greeted (by H) → Seated (at T by H) → Order (at T to W) → Served (at T from W) → Eat (at T) → Pay bill (to H), which may be represented as the ACD shown in Fig. 6.18.

The life cycle of a table (T) is "occupied by diners" and "cleaned," which can be modeled as in the ACD of Fig. 6.19(a). The head waiter (H) greets diners, seats them on a table, and accepts payment, which is may be modeled as in the ACD of Fig. 6.19(b). Waiters (W) take orders, serve meals, and clean the table after the diners leave, which is modeled as the ACD of Fig. 6.19(c). By combining these individual ACDs, an ACD model of the restaurant system can be obtained as shown in Fig. 6.20. Note that the two resource cycles, H and W, have an indefinite sequence.

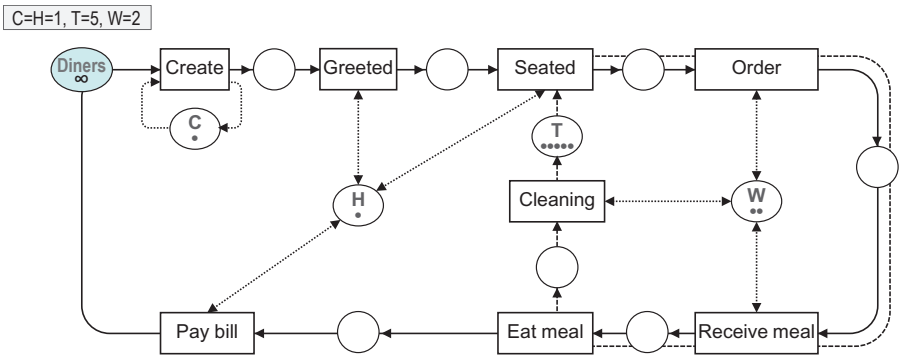


Fig. 6.20. ACD model of the entire restaurant.

Hours (120 min)	0000-0159	0200-0359	0400-0559	0600-0759	0800-0959	1000-1159	1200-1359	1400-1559	1600-1759	1800-1959	2000-2159	2200-2359
Index (k)	0	1	2	3	4	5	6	7	8	9	10	11
Arrival rate/min: R[k]	0.00	0.00	0.00	0.02	0.10	0.30	0.40	0.50	0.40	0.10	0.02	0.00
No. of servers: N[k]	0	0	0	0	3	3	3	5	3	1	1	0

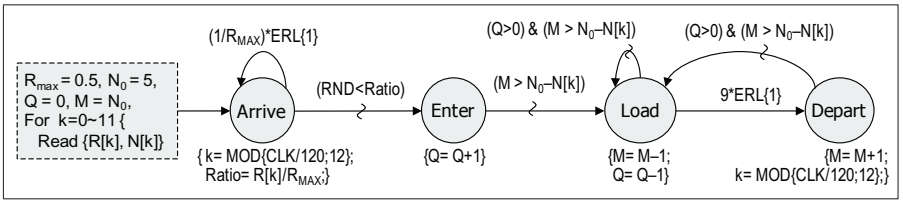


Fig. 6.21. Input data for a flexible multi-server system and its event graph model.

6.4.4 Activity-Based Modeling of a Simple Service Station

A simple service station, like a gas station or coffee shop, is a flexible multi-server system (see Section 6.3.1) with nonstationary arrival rates (see Section 6.3.3). Figure 6.21 presents the table showing the arrival rates and number of servers over a 24-hour period together with a SIGMA-compatible event graph model of the flexible multi-server system that is reproduced from Section 5.6.2 in Chapter 5 (Table 5.4 and Fig. 5.31). In the event graph model, CLK is a built-in function returning the simulation clock (in minutes), and the current simulation time CLK is converted to the index k using the modulus function k = MOD (CLK/120, 12).

The customer arrival rates (per minute) are 0.00 for 00:00–05:59, 0.02 for 06:00–07:59, 0.10 for 08:00–09:59, etc. Thus, we have R[0] = R[1] = R[2] = 0, R[3] = 0.02, R[4] = 0.10, etc. The maximum arrival rate is 0.5 during 14:00~15:59 (R[7] = 0.5). The number of servers during the day hours (8:00~17:59) is three, with a peak level of five during 2:00~3:59 p.m. (N[7] = 5). All servers are identical and their service times are exponentially distributed with a mean of 9. A

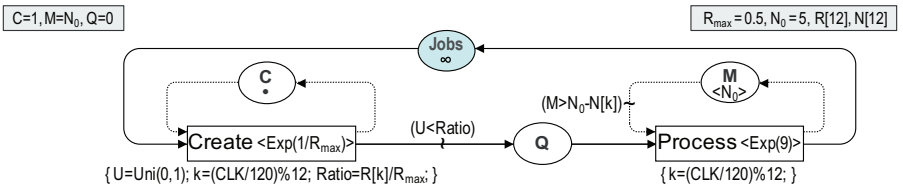


Fig. 6.22. ACD model of the flexible multi-server system given in Fig. 6.21.

TABLE 6.7. Activity Transition Table of the ACD Model of Figure 6.22

		At-begin		BTO-event		At-end			
No	Activity	Condition	Action	Time	Name	Arc	Condition	Action	Infl. Act.
1	Create	(C > 0)	C−−; U = Uni(0,1); k = (CLK/120)%12; Ratio = R[k]/R_{max};	Exp(1/R_{max})	Created	1 2	True (U < Ratio)	C++; Q++;	Create Process
2	Process	(Q > 0)& (M > N_0 − N[k])	Q−−; M−−; k = (CLK/120)%12;	Exp(9)	Processed	1 2	True True	M++; —	Process —
Initialize		*Initial Marking* = {C = 1, M = 5, Q = 0}; *Variables* = {R_{max} = 0.5, N_0 = 5, Read {R[k], N[k]} for k = 0~11}; *Enabled Activities* = {Create};							

formal ACD model of the flexible multi-server system is presented in Fig. 6.22, and its activity transition table is given in Table 6.7.

6.4.5 Activity-Based Modeling of a Car Repair Shop

Figure 6.23 presents a reference model and an event graph model of a car repair shop under the same operator policy that was presented in Section 4.5.2 of Chapter 4 (Fig. 4.20 and Fig. 4.21). The fasten operation is performed by a technician; an inspection operation needs both a technician and a repairman; a repair operation is handled by a repairman. The *same operator policy* refers to a policy where a car is fastened and inspected by the same technician and is inspected and repaired by the same repairman. Thus, a technician stands by after fastening a car until a repairman is available. There are three technicians (T = 3) and two repairmen (R = 2) in the car repair shop.

An ACD model and an activity transition table of the car repair shop are given in Fig. 6.24 and Table 6.8, respectively. Note that the ACD model is almost identical to the reference model. The initial state of the system is {C = 1, T = 3, R = 2, Q1 = Q2 = Q3 = 0}.

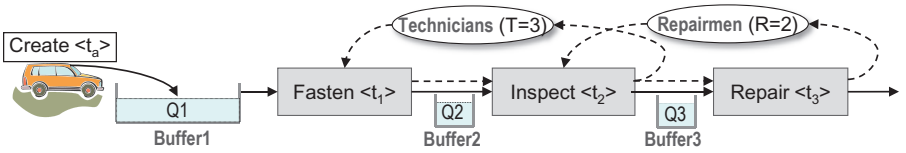


Fig. 6.23. Reference model of a car repair shop.

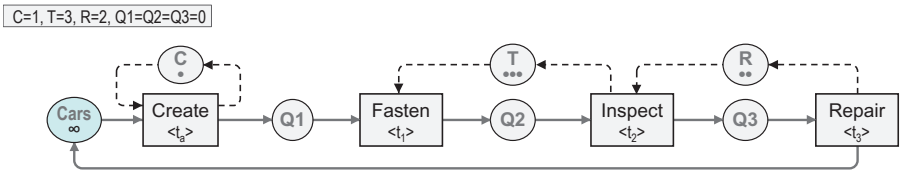


Fig. 6.24. ACD model of the car repair shop given in Fig. 6.23.

TABLE 6.8. Activity Transition Table of the ACD Model of Figure 6.24

		At-begin		BTO-event		At-end			
No	Activity	Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
1	Create	(C > 0)	C—;	t_a	Created	1	True	C++;	Create
						2	True	Q1++;	Fasten
2	Fasten	(Q1 > 0) & (T > 0)	Q1—; T—;	t_1	Fastened	1	True	Q2++;	Inspect
3	Inspect	(Q2 > 0) & (R > 0)	Q2—; R—;	t_2	Inspected	1	True	Q3++;	Repair
						2	True	T++;	Fasten
4	Repair	(Q3 > 0)	Q3—;	t_3	Repaired	1	True	R++;	Inspect
Initialize		*Initial Marking* = {C = 1, T = 3, R = 2, Q1 = Q2 = Q3 = 0}; *Enabled Activities* = {Create}							

6.4.6 Activity-Based Modeling of a Project Management System

Figure 6.25 presents the program evaluation and review technique (PERT) diagram introduced in Chapter 4 (Fig. 4.22). There are nine activities (A1~A9) and two active resources (R1, R2) involved in the model. Resource R1 is responsible for A1, A3, and A7, while resource R2 is responsible for A2, A8, and A9. Again, the non-bottleneck resources performing the remaining activities are excluded from the model. The PERT diagram is converted to an ACD by inserting a queue in the middle of each arc of the PERT diagram, as depicted in Fig. 6.26.

6.4.7 Activity-Based Modeling of a Conveyor-Driven Serial Line

Reproduced in Fig. 6.27 is the reference model of the conveyor-driven serial line introduced in Chapter 4. There are three work stations connected by

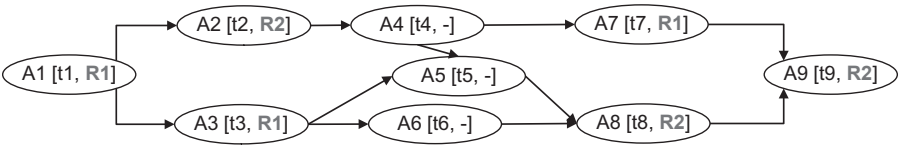


Fig. 6.25. Reference model of a project schedule (PERT diagram).

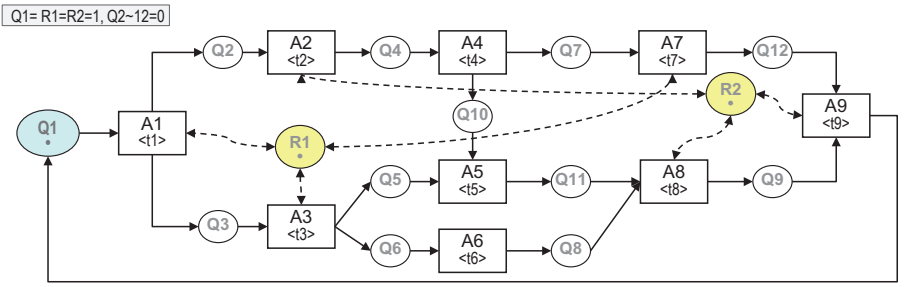


Fig. 6.26. ACD model of the PERT diagram in Fig. 6.25.

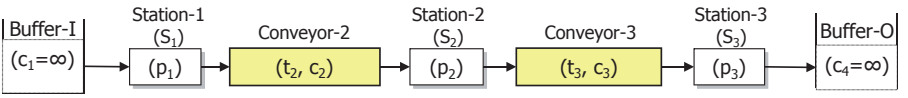


Fig. 6.27. Reference model a three-stage conveyor-driven serial line.

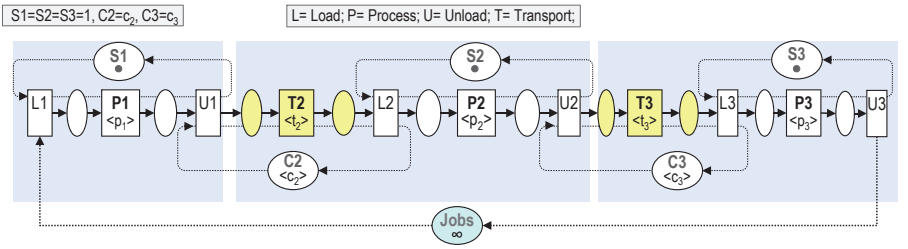


Fig. 6.28. ACD model of the three-stage conveyor driven serial line.

accumulating conveyors in serial. Entities are jobs that are stored in the input buffer (Buffer-I) and are moved along the line. Resources are the Stations and Conveyors. Activities are the production operations of the Stations and the transport operations of the Conveyors.

Each station-j for j = 1~3 is specified by its production operation time (p_j). Each accumulating conveyer-j for j = 2, 3 is specified by its transport time (t_j) and capacity (c_j). The activity cycle of a job at each station is Load (L) → Process (P) → Unload (U). Since an accumulating conveyor acts as a finite capacity buffer with a Transport (T), each station is modeled using the blocking template introduced in Section 6.3.2. The resulting ACD model is presented in Fig. 6.28.

6.5 PARAMETERIZED ACTIVITY CYCLE DIAGRAM AND ITS APPLICATION

In Chapter 5, we showed that a complex system with some repeating patterns could be concisely represented using a parameterized event graph, in which an event node is allowed to have parameter variables. The classical ACD can also be parameterized in the same way. Namely, the parameter values are passed along an arc as its attribute values so that the parameter variables of the destination node are set to the arc attribute values. Parameterization does not increase the modeling power, but it significantly reduces the modeling complexity, which is critical in the art of modeling and simulation.

6.5.1 Definition and Specifications of Parameterized ACD

Both the classical ACD and parameterized ACD (P-ACD) are bipartite directed graphs that consist of a set of activity nodes, a set of queue nodes, and a set of directed arcs. However, in a P-ACD, each node is allowed to have parameter variables, and the parameter values are assigned to the parameter variables through an arc. The role of a parameter variable of a node is the same as that of an index variable of an array. Thus, the P-ACD may be defined as a bipartite directed graph consisting of a set of activity array nodes, a set of queue array nodes, and a set of directed arcs with parameter values.

Figure 6.29 is a classical ACD model of a three-stage unlimited buffer tandem line: the activity nodes are Create and Process(k) for k = 1~3; the queue nodes are C, Jobs, and B(k) and M(k) for k = 1~3. On close examination, it can be seen that the three nodes B(k)-Process(k)-M(k) form a pattern; there is a directed arc from the Process(k) node to the B(k + 1) node if k < 3; and the Process(3) node is connected to the Jobs node. The activity times are defined as an array (t(k)).

Shown in Fig. 6.30 is a P-ACD model of the same three-stage tandem line, where the three nodes B(k)-Process(k)-M(k) form a pattern; there is a directed arc from the Process(k) node to the B(k + 1) node if k < 3; and the Process(3) node is connected to the Jobs node. Thus, the two ACD models in Figs. 6.29 and 6.30 are equivalent. As a convention, a parameter variable is enclosed using a pair of parentheses and a parameter value is put in a small rectangle on an arc. As in the classical ACD, the P-ACD can be formally specified using an activity transition table.

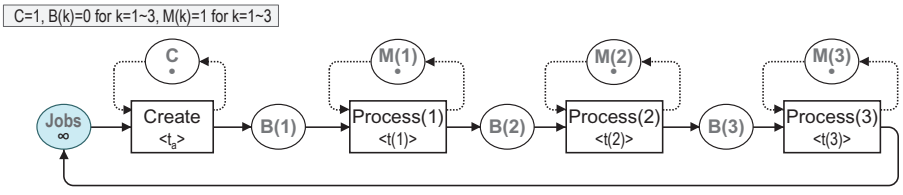


Fig. 6.29. Classical ACD model of a three-stage unlimited buffer tandem line.

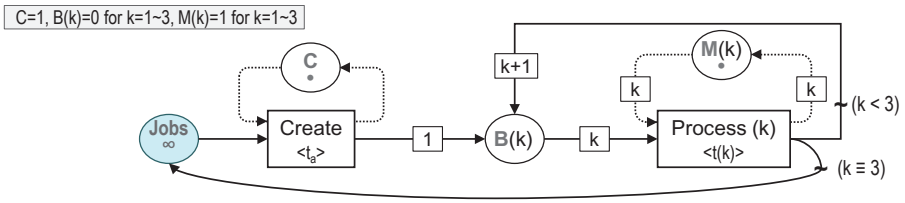


Fig. 6.30. P-ACD model of the three-stage tandem line of Fig. 6.29.

TABLE 6.9. Activity Transition Table of the P-ACD Model Given in Figure 6.30

		At-begin		BTO-event		At-end				
No	Activity	Condition	Action	Time	Name	Arc	Condition	Parameter	Action	Influenced Activity
1	Create	(C > 0)	C—;	t_a	Created	1	True	—	C++;	Create
						2	True	1	B(k)++;	Process(k)
2	Process(k)	(B(k) > 0) & (M(k) > 0)	B(k)—; M(k)—;	t(k)	Processed	1	True	k	M(k)++;	Process(k)
						2	(k < 3)	k + 1	B(k)++;	Process(k)
						3	(k ≡ 3)	—	—	—
Initialize		*Initial Marking* = {C = 1, B(k) = 0 for k = 1~3, M(k) = 1 for k = 1~3}; *Enabled Activities* = {Create}								

Table 6.9 presents the activity transition table of the P-ACD model of Fig. 6.30. The activity transition table contains all information from the P-ACD model in a structured way. As in the activity transition table of a classical ACD model, the initial values of the marking and a list of the enabled activities are defined at the bottom of the table. Specified for each activity are the At-begin Condition and Action, the BTO-event Time and Name, and the At-end arc Condition, Parameter, Action, and Influenced Activity. The activity transition table for the P-ACD is the same as that for a classical ACD except that it has one more column (the Parameter column).

6.5.2 Rules for Executing the P-ACD Model

In the P-ACD, the parameter values are associated with each arc, and the parameter variables are associated with each of the queues or activities. The execution rules for the P-ACD are essentially the same as those for the classical ACD.

Now consider the segment of a P-ACD shown in Fig. 6.31(a). The rule for executing activity A(j) is as follows: "If the input queue Q1 has at least one token and the input arc condition c1 is true, then the parameter variable (j) of activity A is set to the parameter value (k), a token is removed from the input queue Q1, and the BTO event is scheduled to occur at t time units later." Referring to Fig. 6.31(b), the rule for executing the BTO event is "If the output

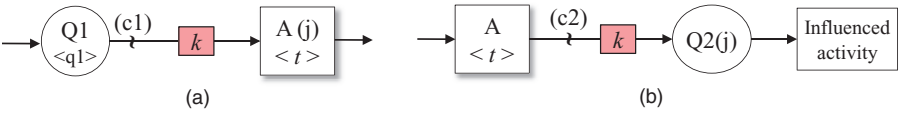


Fig. 6.31. Rules for executing (a) an activity and (b) its BTO event.

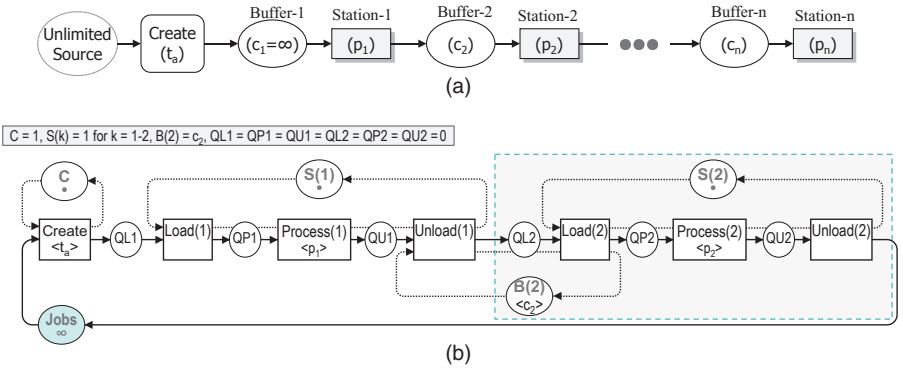


Fig. 6.32. (a) Reference model of an n-stage limited buffer tandem line and (b) ACD model of two-stage limited buffer tandem line.

arc condition c2 is true, then a token will be en-queued into output queue Q2 with parameter variable (j), equal to parameter value (k). The value of the output arc attribute (k) is computed when activity A is executed."

6.5.3 P-ACD Modeling of Tandem Lines

The unlimited buffer tandem line model given in Section 6.5.1 may easily be extended to cover limited buffer tandem lines and conveyor-driven tandem (or serial) lines.

6.5.3.1 P-ACD Modeling of a Limited Buffer Tandem Line Figure 6.32(a) is the reference model of an n-stage limited buffer tandem line, where t_a is the inter-arrival time, p_k is the processing time at Station-k, and c_k is the capacity of Buffer-k (with $c_1 = \infty$). Figure 6.32(b) is a classical ACD model of a two-stage limited buffer tandem line, where S(k) is the number of idle machines at Station-k and B(k) is the number of empty slots at Buffer-k.

As indicated in Fig. 6.32(b) by the shaded rectangle, the major repeating pattern of activities is Load(k) → Process(k) → Unload(k) → Load(k), and there are minor patterns of activities: Unload(k) → Load(k + 1) when k < n and Load(k) → Unload(k − 1) when k > 1. Thus, all (activity/queue) nodes covered by the repeating patterns are parameterized as shown in Fig. 6.33. The activity transition table for the P-ACD model of Fig. 6.33 is given in Table 6.10.

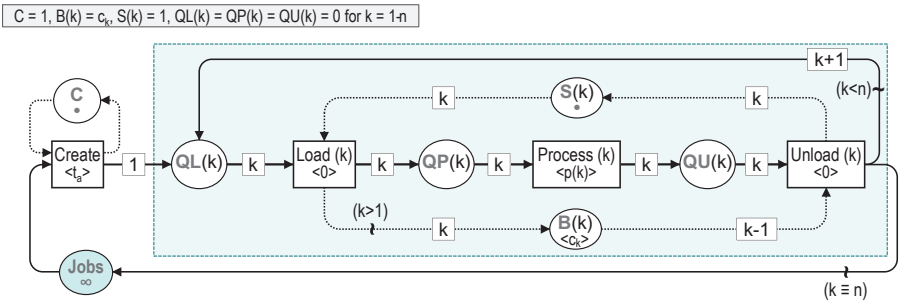


Fig. 6.33. P-ACD model of the limited buffer tandem line of Fig. 6.32(a).

6.5.3.2 P-ACD Modeling of a Conveyor-Driven Serial Line Figure 6.34 presents the reference model of an n-stage conveyor-driven serial line, where p_k is the processing time at Station-k and t_k and c_k, respectively, are the transport time and capacity of Conveyor-k (with $c_1 = \infty$). This system is a slight modification of the limited buffer tandem line given in Fig. 6.32(a): The Create activity of the limited buffer tandem line has been removed and a Transport activity has been added between a pair of adjacent Stations, replacing the buffer queue nodes QL_k.

A classical ACD model of a three-stage conveyor-driven serial (i.e., tandem) line was given earlier in Section 6.4.7 (Fig. 6.28). In Fig. 6.28, you can see that the major repeating pattern of activities is Transport(k) → Load(k) → Process(k) → Unload(k) → Transport(k + 1). The minor patterns of activities are Unload(k) → Transport(k + 1) when k < n, Unload(k) → Load(k), and Load(k) → Unload(k − 1) if k > 1. Reflecting these repeating patterns, a P-ACD model of an n-stage conveyor-driven serial line is obtained as shown in Fig. 6.35.

In the P-ACD of Fig. 6.35, all entity queues are initially empty except QT(1) and QL(1). QT(1) is set to one because the first activity to be executed is Transport(1), and it requires a begin condition of QT(1) ≡ 1; QL(1) is set to infinity because the activity Transport(1) is executed only once [increasing QL(1) by one] and the activity Load(1) is executed numerous times [decreasing QL(1) by one]. That is, QP(k) = QU(k) = 0 for k = 1 − n, and QT(k) = QL(k) = 0 for k = 2 − n, QT(1) = 1 and QL(1) = ∞. The capacities (c_k) and transport times (t_k) of the accumulating conveyors are given for k = 2 − n; for k=1, we set $t_1 = 0$ and $c_1 = \infty$ (or any value). The queue C(n + 1) is set to infinity [i.e., C(n + 1) = ∞] so that the final unloading activity Unload(n) is always executed if a job is available (i.e., QU(n) > 0). Thus, the initial marking ($\mathbf{M}_0$) of the P-ACD is given by:

$$\mathbf{M}_0 = \{S(k) = 1, QP(k) = QU(k) = 0 \text{ for } k = 1 \text{ to } n,$$
$$C(k) = c_k, QT(k) = QL(k) = 0 \text{ for } k = 2 \text{ to } n,$$
$$QT(1) = 1, QL(1) = C(1) = C(n+1) = \infty\}$$

TABLE 6.10. Activity Transition Table of the P-ACD Model Given in Figure 6.33

		At-begin		BTO-event		At-end				
No	Activity	Condition	Action	Time	Name	Arc	Condition	Parameter	Action	Influenced Activity
1	Create	(C > 0)	C—;	t_a	Created	1	True	—	C++;	Create
						2	True	1	QL(k)++;	Load(k)
2	Load(k)	(QL(k) > 0) &	QL(k)—;	0	Loaded	1	True	k	QP(k)++;	Process(k)
		(S(k) > 0)	S(k)—;			2	(k > 1)	k	B(k)++;	Unload(k-1)
3	Process(k)	(QP(k) > 0)	QP(k)—;	p(k)	Processed	1	True	k	QU(k)++;	Unload(k)
4	Unload(k)	(QU(k) > 0) &	QU(k)—;	0	Unloaded	1	True	k	S(k)++;	Load(k)
		(B(k + 1) > 0)	B(k + 1)—;			2	(k < n)	k + 1	QL(k)++;	Load(k)
						3	(k ≡ n)	—	—	—
Initialize		*Initial Marking* = {C = 1, B(k) = c_k, S(k) = 1, QL(k) = QP(k) = QU(k) = 0 for k = 1~n}; *Enabled Activities* = {Create}								

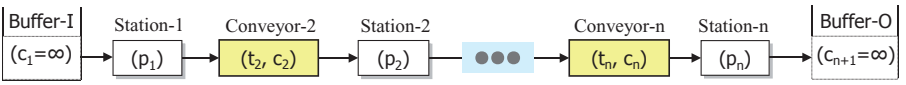


Fig. 6.34. Reference model of n-stage conveyor-driven serial line.

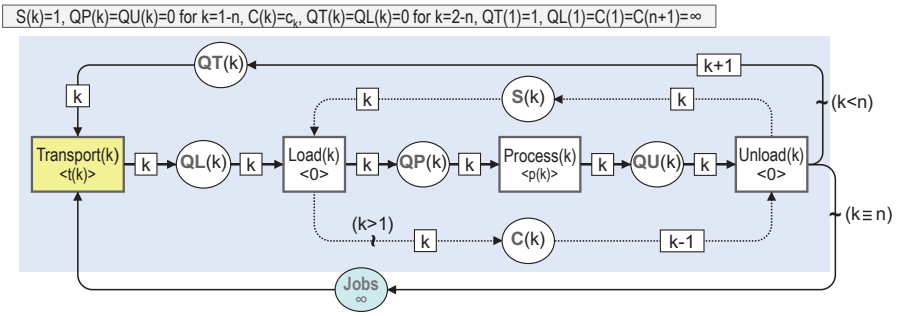


Fig. 6.35. P-ACD model of the conveyor-driven serial line of Fig. 6.34.

The activity transition table for the P-ACD model in Fig. 6.35 is given in Table 6.11.

6.5.4 P-ACD Modeling of Job Shops

As discussed in the previous chapter (Section 5.5), a *simple job shop* is characterized by a number of stations (s) with each station having one or more identical machines and multiple job types (j) with each job type having a unique routing sequence. The station number (s) for a processing step (p) of a given job type (j) is specified in the routing sequence of the job type. Each station has an unlimited buffer space, and job (j) may visit a given station more than once. Thus, an unlimited buffer tandem line may be regarded as a special case of a simple job shop where there is only one job type (j = 1) and the processing step is equal to the station number (i.e., p = s).

Reproduced in Fig. 6.36 is the reference model of a simple job shop described in the previous chapter (see Fig. 5.14). It has six stations with the number of machines given by $\{m_1 = 3, m_2 = 5, m_3 = 4, m_4 = 7, m_5 = 2, m_6 = 5\}$. The routing sequence of a type-1 job is $1 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow Done$: *sn(1,1) = 1, sn(1,2) = 3, sn(1,3) = 2, sn(1,4) = 5, sn(1,5) = Done*. The processing time of type-j job at the p-th step is denoted by t (j, p).

Figure 6.37(a) presents a classical ACD model of the simple job shop for a single job type, where M(k) denotes the number of idle machines in Station-k, and Q(k) is the buffer of Station-k. There are four processing activities: Process (1, 1), Process (2, 3), Process (3, 2), and Process (4, 5), where Process (p, s) is the p-th processing operation of the job of type-1 at Station-s. The routing sequence is given by $\{sn(1,1) = 1, sn(1,2) = 3, sn(1,3) = 2, sn(1,4) = 5\}$. A P-ACD model for the reference model of Fig. 6.36 is shown in Fig. 6.37(b).

TABLE 6.11. Activity Transition Table of the P-ACD Model Given in Figure 6.35

		At-begin		BTO-event		At-end				
No	Activity	Condition	Action	Time	Name	Arc	Condition	Parameter	Action	Influenced Activity
1	Transport(k)	QT(k) > 0	QT(k)−−;	t(k)	Transported	1	True	k	QL(k)++;	Load(k)
2	Load(k)	QL(k) > 0 &	QL(k)−−;	0	Loaded	1	True	k	QP(k)++;	Process(k)
		S(k) > 0	S(k)—;			2	(k > 1)	k	C(k)++;	Unload(k-1)
3	Process(k)	QP(k) > 0	QP(k)—;	p(k)	Processed	1	True	k	QU(k)++;	Unload(k)
4	Unload(k)	QU(k) > 0 &	QU(k)—;	0	Unloaded	1	True	k	S(k)++;	Load(k)
		C(k + 1) > 0	C(k + 1)−−;			2	(k < n)	k + 1	QT(k)++;	Transport(k)
						3	(k ≡ n)	—	—	—
Initialize		*Initial Marking* = {S(k) = 1, QP(k) = QU(k) = 0 for k = 1 − n, C(k) = c_k, QT(k) = QL(k) = 0 for k = 2 − n, QT(1) = 1, QL(1) = C(1) = C(n + 1) = ∞}; *Enabled Activities* = {Transport(1)}								

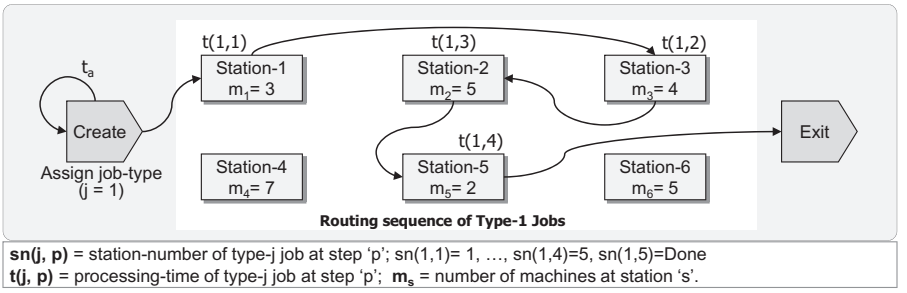


Fig. 6.36. Reference model of a simple job shop with one job type.

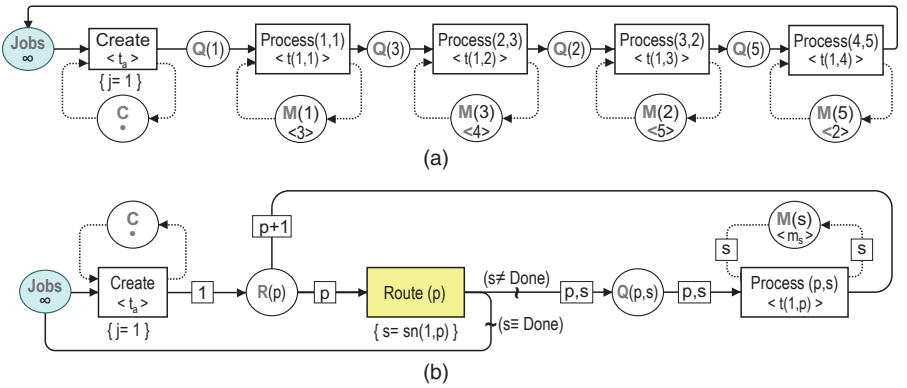


Fig. 6.37. (a) Classical ACD and (b) P-ACD model of the job shop in Fig. 6.36.

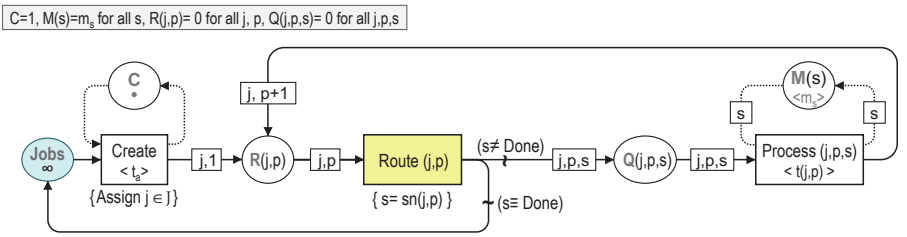


Fig. 6.38. P-ACD model of the simple job shop with multiple job types.

In Fig. 6.37(b), decision-making activity Route is introduced where the station number for the next processing step is determined and the job is sent to the Exit if the station number is Done. Both the Route activity and the Process activity are parameterized by processing step p and the station number s = sn(1,p).

Given in Fig. 6.38 is a P-ACD model of the simple job shop with a number of job types (j ∈ J). The activities Route and Process are parameterized using the three index variables of job type (j), processing step (p), and station

number (s). An activity transition table of the P-ACD model in Fig. 6.38 is given in Table 6.12.

6.6 EXECUTION OF ACTIVITY CYCLE DIAGRAM MODELS WITH A FORMAL SIMULATOR ACE®

There are three approaches to executing an ACD model: using a formal ACD simulator; using a process oriented simulation language such as Arena®; and developing a dedicated ACD simulator. The purpose of this section is to introduce a formal ACD model simulator ACE® and make you learn how to execute ACD models with ACE. The next chapter is devoted to the subject of executing ACD models using Arena®. How to develop your own dedicated ACD simulators are described in Chapter 10.

ACE, which stands for *activity cycle executor*, is a formal ACD model simulator developed by the authors of this book. It is a formal simulator in the sense that its input is a formal ACD model specified in the form of activity transition table. Figure 6.39 shows the main window of ACE that has three main regions: main menu, Activity Transition Table (ATT) window, and Spreadsheet window. Also provided in the main window are ATT tool bar and Queue tool bar. A brief ACE tutorial, as well as the four ACE models discussed in this section, may be found in the official website of this book (http://VMS-technology.com/Book/ACE).

The ACE main menu contains a few menus including File, Run, and Help. The ATT window is where the activity transition table of the ACD model is constructed. The Spreadsheet window is used for declaring queues and variables appearing in the ACD model. The procedure for executing an ACD model is as follows:

1. Queues and variables are declared in the Spreadsheet window by selecting the Queue spreadsheet and Variable spreadsheet, respectively.
2. All the activity transition data are described in the ATT window by clicking the subsequent menus of Model menu in the main menu.
3. The initially enabled activities are set at the ATT window by clicking the Set Enabled Activity button in the ATT tool bar.
4. Simulation run options are specified in the Run Options dialog box by clicking the Run > Run Options menu in the main menu.

6.6.1 Simulation of Single Server Model with ACE

The above four-step procedure will be illustrated using the single server system specified in Table 6.13, which was reproduced from Table 6.2 with interarrival times (t_a) and service times (t_s) specified as Exp(15) and Uni(10, 12), respectively.

TABLE 6.12. Activity Transition Table of the Job Shop P-ACD Model in Figure 6.38

		At-begin		BTO-event		At-end				
No	Activity	Condition	Action	Time	Name	Arc	Condition	Parameter	Action	Influenced Activity
1	Create	(C > 0)	C—; Assign j;	t_a	Created	1 2	True True	— j,1	C++; R(j,p)++;	Create Route(j,p)
2	Route(j,p)	(R(j,p) > 0)	R(j,p)—; s = sn(j,p);	0	Routed	1 2	(s ≠ Done) (s ≡ Done)	j,p,s —	Q(j,p,s)++; —	Process(j,p,s) —
3	Process(j,p,s)	(M(s) > 0) & (Q(j,p,s) > 0)	M(s)—; Q(j,p,s)−−;	t(j,p)	Processed	1 2	True True	s j, p + 1	M(s)++; R(j,p)++;	Process(j,p,s) Route(j,p)
Initialize		*Initial Marking* = {C = 1; M(s) = m_s for all s, R(j,p) = 0 for all j, p, Q(j,p,s) = 0 for all j,p,s}; *Enabled Activities* = {Create}								

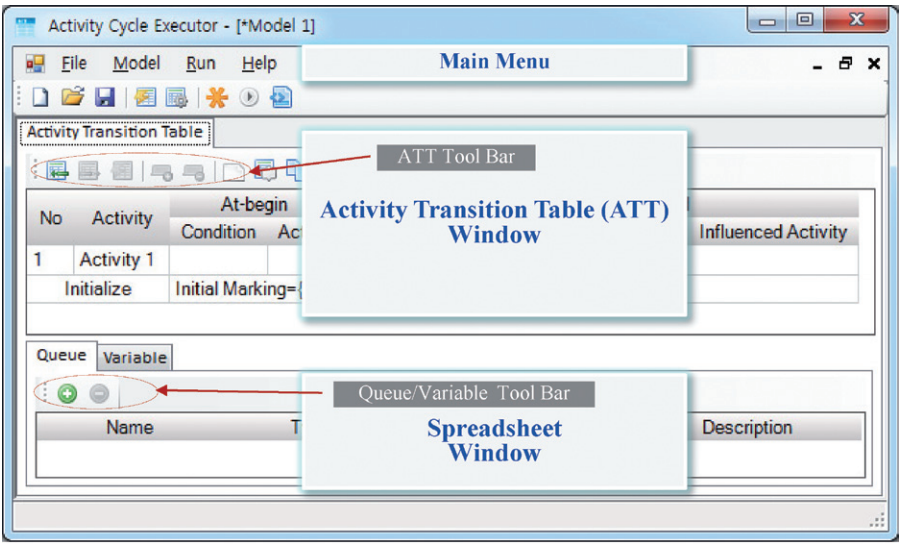


Fig. 6.39. The main window of ACE.

TABLE 6.13. Activity Transition Table of a Single Server System (from Table 6.2)

		At-begin		BTO-event		At-end			
No	Activity	Condition	Action	Time	Name	Arc	Condition	Action	Infl. Activity
1	Create	(C > 0)	C—;	Exp(15)	Created	1	True	C++;	Create
						2	True	Q++;	Process
2	Process	(M > 0) & (Q > 0)	M—; Q—;	Uni(10, 12)	Processed	1	True	M++;	Process
Initialize		*Initial Marking* = {C = 1, M = 1, Q = 3}; *Enabled Activities* = {Create}							

6.6.1.1 Declaring Queues and Variables in the Single Server Model The ACD model of the single server system has three queues (C, M and Q), but it has no variables. A queue is Creator Type, Resource Type, or Entity Type. Figure 6.40 shows the Queue spreadsheet window in which the three queues C, M, and Q are declared. The type of C is declared as Creator, those of M and Q as Resource and Entity, respectively. The initial values of C, M, and Q are declared as 1, 1, and 3, respectively. The declared queues are reflected in the ATT window as depicted in Fig. 6.41.

6.6.1.2 Defining Activity Transitions in the Single Server Model Figure 6.42 shows the graphical user interface (GUI) of the ATT window in which each entry of the activity transition table of the single server system (Table 6.13) is described "as it is" one by one: (1) the row numbers in the No column

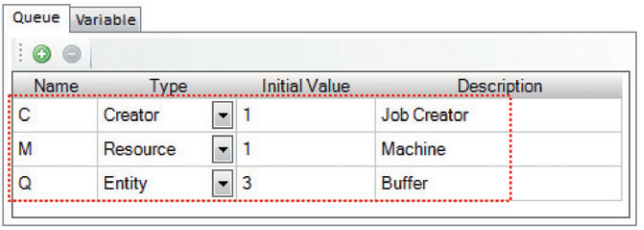


Fig. 6.40. Declaring queues in the Queue spreadsheet window.

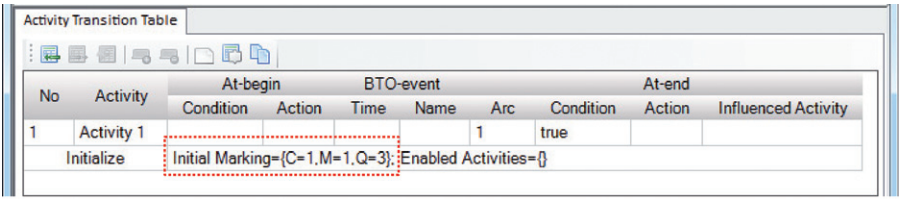


Fig. 6.41. The declared queues reflected in the ATT window.

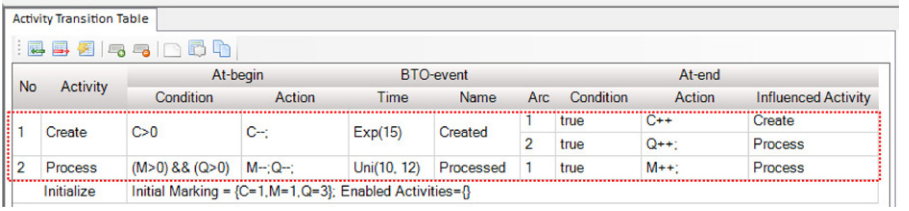


Fig. 6.42. The activity transition table of the single server system.

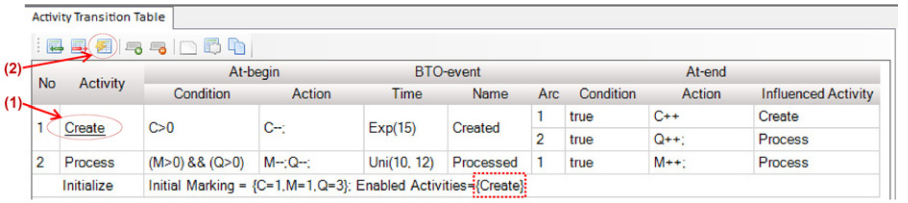
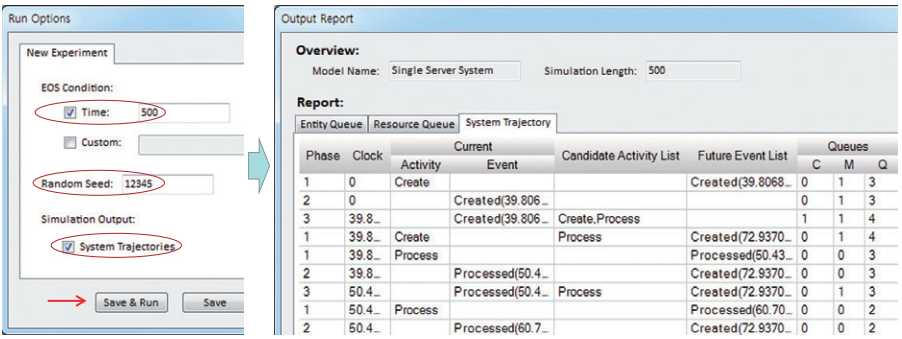


Fig. 6.43. Choosing Create as an enabled activity.

are automatically assigned in sequence; (2) AND operator "&&" and OR operator "||" are used in the Condition columns; (3) each statement in the Action columns should be ended with a semicolon; (4) a time in the Time column is either a constant or a random variate specified, for example, as Exp() or Uni().

6.6.1.3 Specifying the Enabled Activities in the Single Server Model As depicted in Fig. 6.43, the Create activity is specified as an enabled activity by



(a) (b)

Fig. 6.44. (a) Run Options dialog box and (b) Output Report sheet.

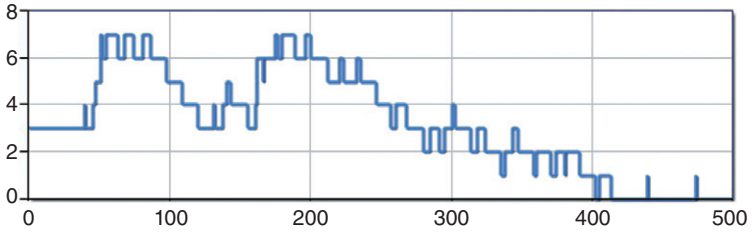


Fig. 6.45. Output plot for Q.

(1) selecting Create in the ATT window and (2) clicking the Set Enabled Activity ( ) button at the ATT tool bar. Now, the ATT is completely defined.

6.6.1.4 Running the Simulation In order to run the simulation, (1) the Run Options dialog box is opened by clicking the Run > Run Options menu at the main menu and (2) run options such as the EOS (end-of-simulation) time and the random number seed are provided in the dialog box. Figure 6.44 shows an example of Run Options (EOS time = 500; random number seed = 12345) and the resulting output report. Output plots may also be generated as shown in Fig. 6.45.

6.6.2 Simulation of Probabilistic Branching Model with ACE

Figure 6.46 is the ACD model for probabilistic branching reproduced from Fig. 6.12 in Section 6.3.6. In Fig. 6.46, however, counter variables (NumberIn, NumberOut, and NumberScrap) for collecting statistics are added to the original ACD model of Fig. 6.12. An activity transition table of the ACD model is given in Table 6.14.

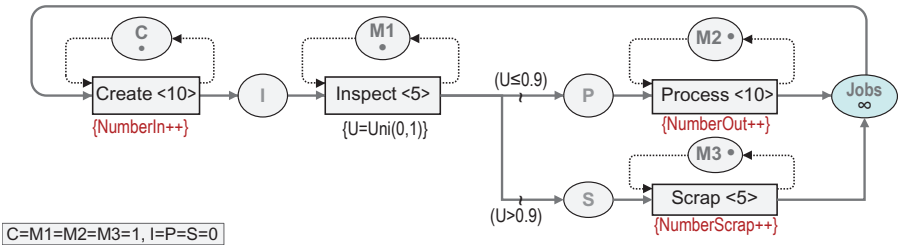


Fig. 6.46. ACD model for probabilistic branching with counter variables added to the ACD model of Fig. 6.12.

6.6.2.1 Declaring Queues and Variables in the Probabilistic Branching Model The probabilistic branching ACD model of Fig. 6.46 has three entity queues (I, P, and S), three resource queues (M1, M2, and M3), and four variables (U, NumberIn, NumberOut, and NumberScrap). The queues and variables are declared in the Spreadsheet window of ACE as shown in Figs. 6.47 and 6.48, respectively.

6.6.2.2 Defining Activity Transitions and Specifying the Enabled Activities Figure 6.49 shows the ATT window in which each entry of the activity transition table (Table 6.14) is described "as it is" one by one and the selected enabled activity is denoted at the bottom right of the table.

6.6.2.3 Running the Simulation Having defined all the entries in the activity transition table of Fig. 6.49, a simulation run is made by providing run options such as the EOS time and the random number seed (any integer value is acceptable) at the Run Options dialog box. Figure 6.50 shows an example of output plot (machine utilization plot).

6.6.3 Simulation of Resource Failure Model with ACE

Figure 6.51 shows the ACD model of the "single server system with machine failure" introduced in Section 6.3.7, together with its activity transition table. In this particular example, the inter-arrival times are sampled from Exp(10), the service time is 10 (ts = 10), repair time is 80, and the initial remaining time-to-failure is 500 (ttf = 500).

6.6.3.1 Declaring Queues and Variables in the Resource Failure Model Figure 6.52(a) shows the Queue spreadsheet window where the three queues in the resource failure model are declared: C is a Creator-type queue with an initial value of 1; M is a Resource-type queue with an initial value of 1; Q is an Entity-type queue with an initial value of 0. Figure 6.52(b) shows

TABLE 6.14. Activity Transition Table for the Probabilistic Branching Model of Figure 6.46

		At-begin		BTO-event		At-end			
No	Activity	Condition	Action	Time	Name	Arc	Condition	Action	Infl. Act.
1	Create	(C > 0)	C—; NumberIn++;	10	Created	1	True	C++;	Create
						2	True	I++;	Inspect
2	Inspect	(I > 0) & (M1 > 0)	I—; M1—;	5	Inspected	1	True	M1++;	Inspect
			U = Uni(0, 1);			2	(U ≤ 0.9)	P++;	Process
						3	(U > 0.9)	S++;	Scrap
3	Process	(P > 0) & (M2 > 0)	P—; M2—; NumberOut++;	10	Processed	1	True	M2++;	Process
4	Scrap	(S > 0) & (M3 > 0)	S—; M3—; NumberScrap++;	5	Scraped	1	True	M3++;	Scrap
Initialize		*Initial Marking* = {C = M1 = M2 = M3 = 1, I = P = S = 0}; *Variables* = {NumberIn = NumberOut = NumberScrap = U = 0}; *Enabled Activities* = {Create}							

Name	Type	Initial Value	Description
C	Creator	1	Job Creator
I	Entity	0	Inspect Queue
M1	Resource	1	Machine 1
M2	Resource	1	Machine 2
M3	Resource	1	Machine 3
P	Entity	0	Process Queue
S	Entity	0	Scrap Queue

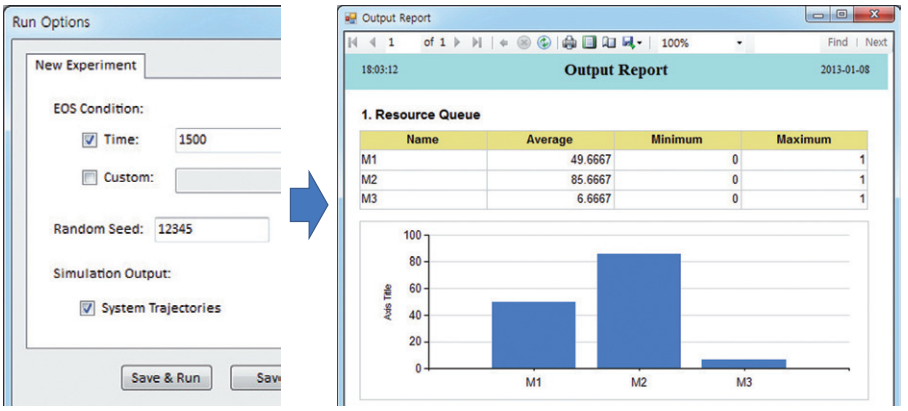
Fig. 6.47. Declaring queues in the Queue spreadsheet window.

Name	Rows	Columns	Type	Initial Value	Description
NumberIn	1	1	int	1 rows	
NumberOut	1	1	int	1 rows	
NumberScrap	1	1	int	1 rows	
U	1	1	double	1 rows	Uniform Random Number

Fig. 6.48. Declaring variables in the Variable spreadsheet window.

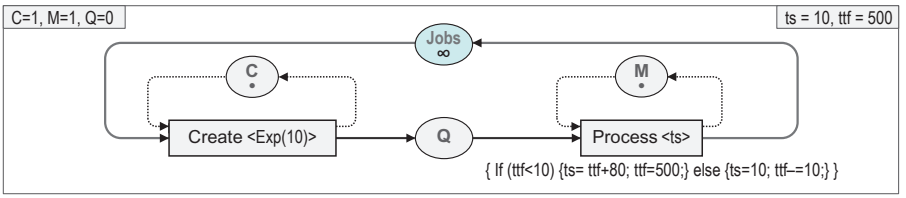
No	Activity	At-begin Condition	At-begin Action	BTO-event Time	BTO-event Name	BTO-event Arc	At-end Condition	At-end Action	At-end Influenced Activity
1	Create	(C>0)	C--;NumberIn++;	10	Created	1	true	C++;	Create
						2	true	I++;	Inspect
2	Inspect	(I>0) && (M1>0)	I--;M1--; U=Uni(0,1);	5	Inspected	1	true	M1++;	Inspect
						2	(U<=0.9)	P++;	Process
						3	(U>0.9)	S++;	Scrap
3	Process	(P>0) && (M2>0)	P--;M2--;NumberOut++;	10	Processed	1	true	M2++;	Process
4	Scrap	(S>0) && (M3>0)	S--;M3--;NumberScrap++;	5	Scraped	1	true	M3++;	Scrap
	Initialize	Initial Marking={C=1,I=0,M1=1,M2=1,M3=1,P=0,S=0};		Variables={NumberIn=0,NumberOut=0,NumberScrap=0,U=0};		Enabled Activities={Create}.			

Fig. 6.49. Description of activity transitions and selection of enabled activities.



(a) (b)

Fig. 6.50. (a) Run option and (b) resources utilization plot.



No	Activity	At-begin		BTO-event		At-end			
		Condition	Action	Time	Name	Arc	Condition	Action	Infl. Act.
1	Create	(C>0)	C--;	Exp(10)	Created	1	True	C++;	Create
						2	True	Q++;	Process
2	Process	(Q>0) & (M>0)	Q--; M--; If (ttf<10) {ts= ttf + 80; ttf = 500;} else {ts = 10; ttf –= 10;}	ts	Processed	1	True	M++;	Process
Initialize		*Initial Marking* = {C= 1, M= 1, Q= 0}; *Variables* = {ts= 10, ttf= 500}; *Enabled Activities* = {Create};							

Fig. 6.51. ACD model and activity transition table for the single server system with machine failure (from Fig. 6.13 and Table 6.6).

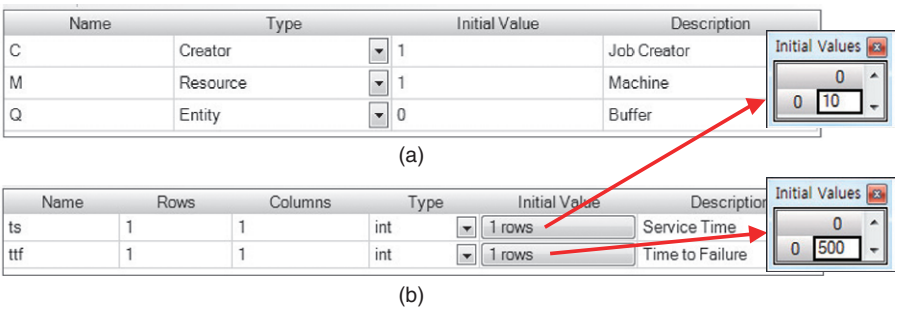


Fig. 6.52. Declaring (a) queues and (b) variables in the Spreadsheet window of ACE.

No	Activity	At-begin		BTO-event		At-end			
		Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
1	Create	(C>0)	C--;	Exp(10)	Created	1	true	C++;	Create
						2	true	Q++;	Process
2	Process	(Q>0) && (M>0)	Q--;M--; if (ttf<10) {ts=ttf+80;ttf=500;} else {ts=10;ttf-=10;}	ts	Processed	1	true	M++;	Process
Initialize		Initial Marking={C=1,M=1,Q=0}; Variables={ts=10,ttf=500}; Enabled Activities={Create}							

Fig. 6.53. Descriptions of activity transitions and enabled activities.

how the two variables ts and ttf are declared in the Variable spreadsheet window.

6.6.3.2 Defining Activity Transitions and Specifying the Enabled Activities Figure 6.53 shows the ATT window in which each entry of the activity transition table (at the bottom of Fig. 6.51) is described "as it is" one by

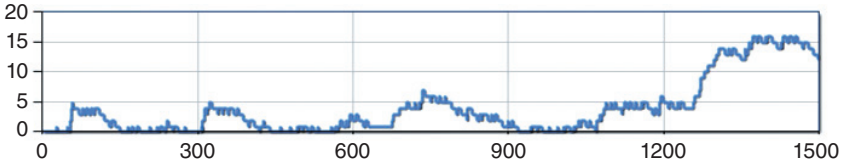


Fig. 6.54. Queue trajectory plot (for the entity queue Q).

Hours (120 min)	0000-0159	0200-0359	0400-0559	0600-0759	0800-0959	1000-1159	1200-1359	1400-1559	1600-1759	1800-1959	2000-2159	2200-2359
Index (k)	0	1	2	3	4	5	6	7	8	9	10	11
Arrival rate/min: R[k]	0.00	0.00	0.00	0.02	0.10	0.30	0.40	0.50	0.40	0.10	0.02	0.00
No. of servers: N[k]	0	0	0	0	3	3	3	5	3	2	2	0

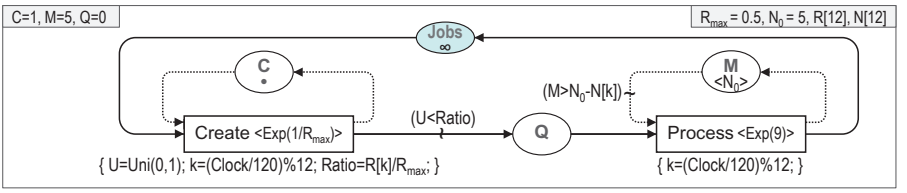


Fig. 6.55. Input data (Fig. 6.21) and ACD model (Fig. 6.22) of the simple service station.

one and the selected enabled activity is denoted at the bottom right of the table.

6.6.3.3 Running the Simulation Having defined all the entries in the activity transition table, a simulation run is made by providing run options at the Run Options dialog box. Figure 6.54 shows an example of output plot (queue trajectory plot).

6.6.4 Simulation of Simple Service Station Model with ACE

Reproduced in Fig. 6.55 are the input data of the simple service station and its ACD model from Section 6.4.4 (Fig. 6.21 and Fig. 6.22). The ACD model correctly reflects the changes in the arrival rates, but it does not exactly accommodate the changes in the resource levels.

In the ACD model of Fig. 6.55, the changes in the number of servers may not be promptly taken into account in the Process activity. For example, the number of servers is increased to 3 from 0 at time 0800 so that the customers waiting in Q could be served right away, but the system detects this change only at the next event time after 0800 (i.e., the arrival time of the first after 0800). As shown in Fig. 6.56, the problem is easily fixed by introducing the

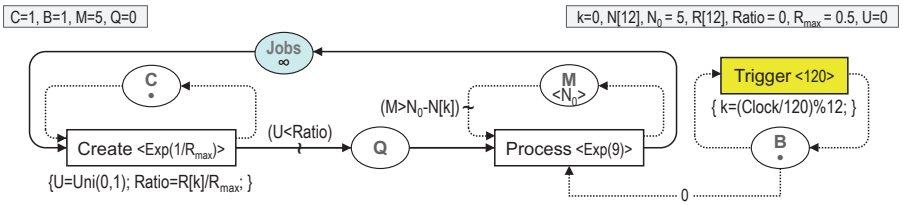


Fig. 6.56. Corrected ACD model of the simple service station.

Trigger activity whose role is to trigger the Process activity every 2 hours. Observe that the arc multiplicity of the arc from queue B to activity Process is 0, which means that Process is an influenced activity of Trigger but no token flows between the two activities, as can be seen in the activity transition table of Table 6.15. Another observation to make is that among the two influenced activities of the Trigger activity, Trigger should be executed before Process. Thus, in the activity transition table of Table 6.15, Trigger is listed first in the Influenced Activity column of the Trigger activity.

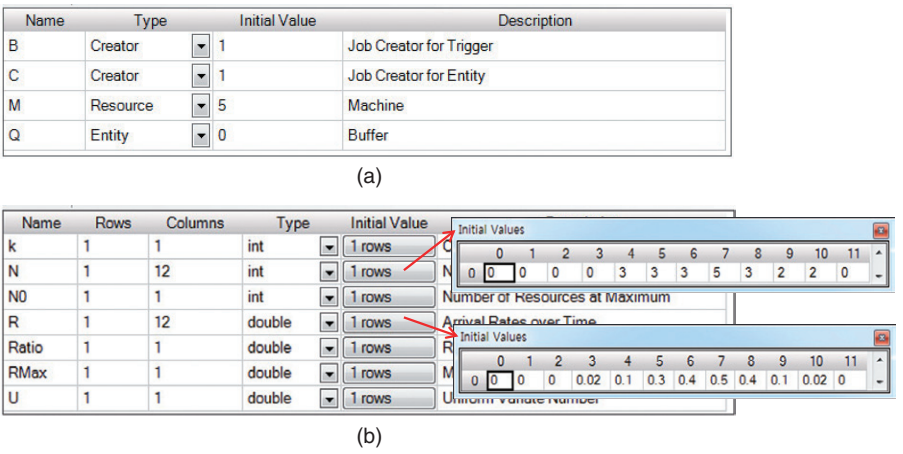
6.6.4.1 Declaring Queues and Variables in the Simple Service Station Model Figure 6.57(a) shows the Queue spreadsheet window of ACE where the queues in the simple service station model are declared: C and B are Creator-type queues with an initial value of 1; M is a Resource-type queue with an initial value of 5; Q is an Entity-type queue with an initial value of 0. Fig. 6.57(b) shows all the variables declared in the Variable spreadsheet window (a step-by-step procedure for declaring variables is given in ACE tutorial that can be found on authors' website: http://VMS-technology.com/Book/ACE).

6.6.4.2 Defining Activity Transitions and Specifying the Enabled Activities Figure 6.58 shows the ATT window in which each entry of the activity transition table (Table 6.15) is described "as it is" one by one, except the integer assignment expression k = (int)((Clock/120)%12), where (int) is inserted for an explicit type casting. The enabled activities are denoted at the bottom right of the window. A step-by-step procedure for defining the activity transitions is presented in ACE tutorial that can be found on authors' web site (http:// VMS-technology.com/Book/ACE).

6.6.4.3 Running the Simulation Having defined all the entries in the activity transition table, a simulation run is made by providing run options at the Run Options dialog box. Figure 6.59 shows an example of output plot (queue trajectory plot for queue Q).

TABLE 6.15. Activity Transition Table of the Corrected ACD Model of Figure 6.56

		At-begin		BTO-event		At-end			
No	Activity	Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
1	Create	(C > 0)	C—; U = Uni(0,1); Ratio = R[k]/R_{max};	Exp(1/ R_{max})	Created	1 2	True (U < Ratio)	C++; Q++;	Create Process
2	Process	(Q > 0) & (M > N_0 – N[k])	Q—; M—;	Exp(9)	Processed	1 2	True True	M++; —	Process —
3	Trigger	(B > 0)	B—; k = (Clock/120)%12;	120	Triggered	1	True	B++;	Trigger, Process
Initialize		*Initial Marking* = {C = B = 1, M = 5, Q = 0}; *Variables* = {k = 0, N[12], N_0 = 5, R[12], Ratio = 0, R_{max} = 0.5, U = 0}; *Enabled Activities* = {Create, Trigger};							



Name	Type	Initial Value	Description
B	Creator	1	Job Creator for Trigger
C	Creator	1	Job Creator for Entity
M	Resource	5	Machine
Q	Entity	0	Buffer

(a)

Name	Rows	Columns	Type	Initial Value
k	1	1	int	1 rows
N	1	12	int	1 rows
N0	1	1	int	1 rows
R	1	12	double	1 rows
Ratio	1	1	double	1 rows
RMax	1	1	double	1 rows
U	1	1	double	1 rows

(b)

Fig. 6.57. Declaring (a) queues and (b) variables in the Spreadsheet window of ACE.

No	Activity	At-begin		BTO-event		At-end		
		Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
1	Create	(C>0)	C--;U=Uni(0,1); Ratio=R[k]/RMax;	Exp(1/RMax)	Created	1	true	C++;	Create
						2	(U<Ratio)	Q++;	Process
2	Process	(Q>0) &(M>N0-N[k])	Q--;M--;	Exp(9)	Processed	1	true	M++;	Process
3	Trigger	(B>0)	B--;k=(int)((Clock/120)%12);	120	Triggered	1	true	B++;	Trigger, Process
Initialize		Initial Marking={B=1,C=1,M=5,Q=0}; Variables={k=0,N[12],N0=5,R[12],Ratio=0,RMax=0.5,U=0};				Enabled Activities={Create,Trigger}			

Fig. 6.58. Descriptions of activity transitions and enabled activities.

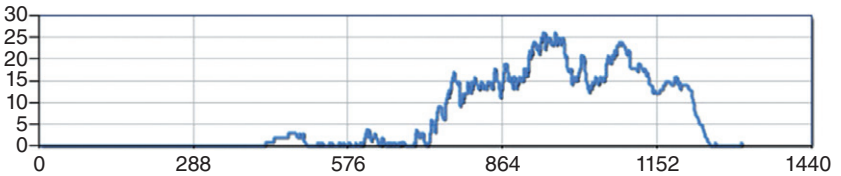


Fig. 6.59. Queue trajectory plot (for the entity queue Q).

6.7 REVIEW QUESTIONS

6.1. What are the basic conventions for drawing a classical ACD?

6.2. What is the initial marking of an ACD model?

6.3. What is an extended ACD?

6.4. What is a BTO event?

6.5. What is an activity transition table?

6.6. What is a flexible multi-server system?

CHAPTER 7

Simulation of ACD Models Using Arena®

Knowledge is a treasure, but practice is the key to it.

—English proverb

7.1 INTRODUCTION

This chapter introduces a systematic framework for executing activity cycle diagram (ACD) models using the process-oriented simulation language, Arena®. Process-oriented modeling is often referred to as *entity-based modeling* or *process interaction worldview*. Arena is one of the most popular simulation languages in academia. There are a few Arena-based simulation text books available for more detailed discussions on modeling with Arena, e.g., Altiok and Melamed [2007], Kelton et al. [2007], and Rossetti [2010]. Here we focus on how we use Arena for executing models developed with ACD.

In discrete-event system modeling, a process is defined as *a time-ordered sequence of events that may encompass several activities* [Pritsker and Pegden 1979]. When an entity, e.g., a customer, is involved in a process, the process may be described in terms of the *flow of the entity* being processed. In this respect, the terms *process* and *entity-flow* are often used interchangeably. Figure 7.1 depicts the relationships between the concepts of events, activities, and processes. The Arena model in Fig. 7.1 shows that it describes the entire experience of an entity as it flows through the system while interacting with the resources R1 and R2. An Arena model is thus basically an entity-flow diagram (EFD) for the system being modeled.

The purpose of this chapter is twofold: (1) introduce the basics of the simulation language Arena, and (2) present a structured guide to executing ACD models with Arena. After studying this chapter, the readers should be able to do the following:

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.

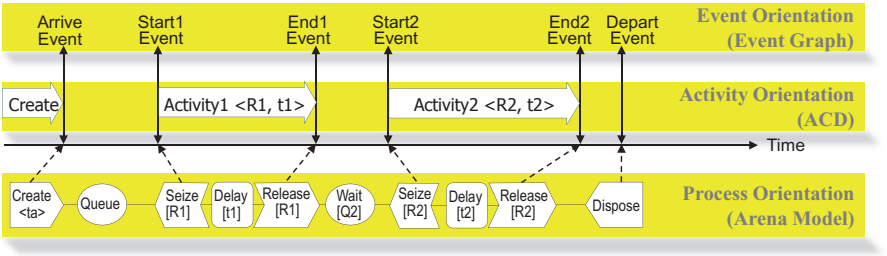


Fig. 7.1. Relationships between the activity orientation and process orientation.

1. Understand Arena models (i.e., flowchart models and static models) built by others
2. Build Arena models of various discrete-event systems
3. Convert various types of ACD models to Arena models
4. Run Arena simulations and analyze the outputs

The remainder of the chapter is organized as follows: Basic elements of Arena are described in Section 7.2. Those who are already familiar with Arena may skip this section and directly proceed to Section 7.3. Section 7.3 presents key Arena model templates for converting ACD models to Arena models. The last section introduces ACD-based Arena modeling examples in various application areas.

7.2 ARENA BASICS

Arena is a simulation environment consisting of module templates built on SIMAN® language constructs, and it is augmented by a visual front end [Altiok and Melamed 2007]. SIMAN [Pegden 1989], from which Arena has evolved, consists of two classes of objects: *blocks* and *elements*. Blocks are basic logic constructs representing operations (or a process), and elements represent facilities such as resources and queues and other components.

The fundamental building blocks of Arena models are *modules*. 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 the module consists of a few SIMAN blocks such as ASSIGN, QUEUE, SEIZE, DELAY, and RELEASE. When constructing an Arena flowchart model, modules are selected from template panels, e.g., Basic Process and Advanced Process panels, and placed on the model window. This section provides a brief overview of Arena. More details on modeling with Arena can be found in Arena User's Guide [Rockwell Automation 2010] and Kelton et al. [2007]. How to obtain a free copy of Arena is explained in http://VMS-technology.com/Book/Arena.

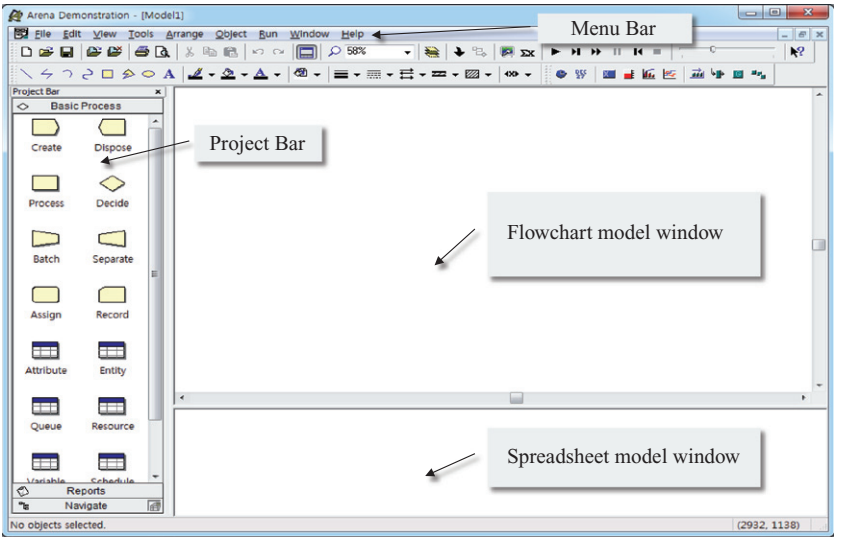


Fig. 7.2. The Arena main window.

7.2.1 Arena Modeling Environment

The Arena modeling environment will open with a main window (see Fig. 7.2). There are three main regions in the main window: Menu Bar, Project Bar, and Model Windows (flowchart model window and spreadsheet model window).

7.2.1.1 Menu Bar The Arena Menu Bar consists of a number of general purpose menus (File, Edit, View, Window, and Help) and Arena-specific menus (Tools, Arrange, Object, and Run). The functions of the Arena-specific menus are as follows: (1) The Tools menu provides access to Arena parameters and tools; (2) the Arrange menu is used for drawing and constructing flowchart models; (3) the Object menu is for creating submodels and connecting modules; (4) the Run menu allows controlling simulation runs.

7.2.1.2 Project Bar The Project Bar provides access to various Arena modules in Arena template panels. The core template panels that are essential for building an Arena model are the Basic Process panel, the Advanced Process panel, and the Advanced Transfer panel.

1. The Basic Process panel provides (1) a set of high-level flowchart modules such as Create, Assign, Batch, Process, and Decide modules; (2) a set of data modules for defining objects in the spreadsheet model such as Entity, Queue, Resource, and Schedule modules; and (3) calendar schedule information.

2. The Advanced Process panel provides (1) a set of low-level flowchart modules such as Pickup, Match, Seize, Delay, Release, Hold, Dropoff, and Signal modules; and (2) a set of advanced data modules such as Advanced Set, Failure, Expression, and File modules.
3. The Advanced Transfer panel provides (1) a set of general flowchart modules such as Enter, Route, Leave, and Station modules; (2) a set of conveyor flowchart modules such as Access, Convey, and Exit modules; (3) a set of transporter flowchart modules such as Activate, Allocate, Request, Move, and Free modules; (4) a set of data modules such as Sequence, Conveyor, Transporter, Network, and Segment modules.

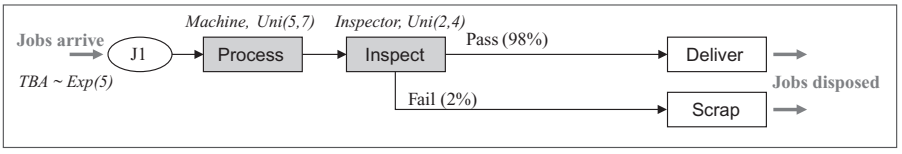
7.2.1.3 Model Window Arena model window is divided into two sections: flowchart section and spreadsheet section. The Flowchart Model window is where the graphical representation of your model is presented, including the process flowchart, animation, and other drawing elements. The Spreadsheet Model window displays model data, such as times, costs, and other parameters.

A flowchart model is built progressively by selecting necessary modules one at a time from the Project Bar and dragging-and-dropping them into the Flowchart Model window. When you select a module in the flowchart model, a spreadsheet containing relevant information on the module will be displayed in the Spreadsheet Model window so that you can edit the associated data.

7.2.2 Building a Flowchart Model of a Process-Inspect Line

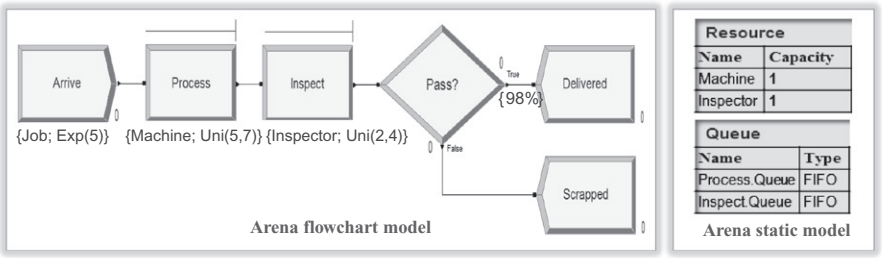
Figure 7.3(a) shows a reference model of a simple process-inspect line. The time between arrivals (TBA) of jobs follows an exponential distribution with mean 5. Jobs are processed by a Machine whose delay time (i.e., processing time) is uniformly distributed with a range between 5 and 7. They are then inspected by an Inspector whose delay time is uniformly distributed with a range of between 2 and 4. It is expected that 98% of the processed jobs would pass the inspection. As a font convention, Arial Narrow is used for Arena module names.

A flowchart model and static model of the above process-inspect line are shown in Fig. 7.3(b). The *flowchart model* consists of (1) a Create module by module name Arrive, (2) two Process modules Process and Inspect, (3) a Decide module Pass?, and (4) two Dispose modules Delivered and Scrapped. The *static*



(a)

Fig. 7.3(a). Reference model of a process-inspect line.



(b)

Fig. 7.3(b). Arena flowchart model and static model of the process-inspect line.

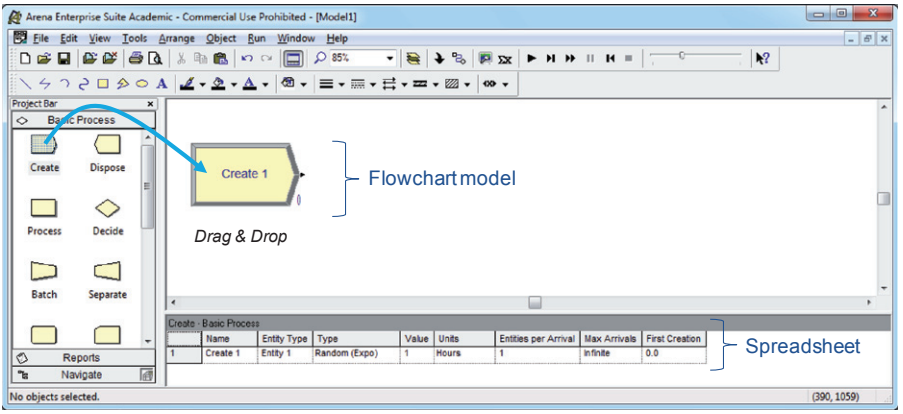


Fig. 7.4. Drag a Create module into the flowchart model window.

model specifies the resources and queues in the system. In the following, a step-by-step procedure for building the Arena flowchart model of Fig. 7.3(b) is explained.

7.2.2.1 Build a Create Module Arrive We'll start building the flowchart model with Create module in the Basic Process panel. A Create module is the starting point for the flow of entities through the model.

1. Figure 7.4: Drag the Create module in the Basic Process panel into the flowchart model window. A default name, Create 1, is given to the module when it is placed in the flowchart model window.
2. Figure 7.5(b): Edit the spreadsheet to define the Create module[1]: (1) Set Name to Arrive; (2) set Entity type to Job; (3) set Value to 5; and (4) set Unit to Minutes.

[1] In Arena, the attribute values for flowchart modules can also be entered in a dialog box by doubling-clicking the module in the flowchart model window.

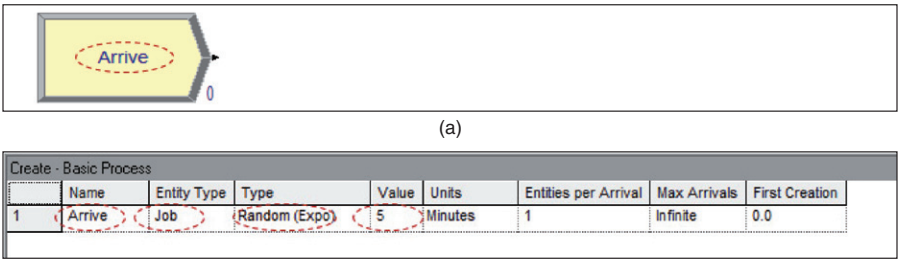


Fig. 7.5. (a) Flowchart model and (b) spreadsheet definition of the Arrive module.

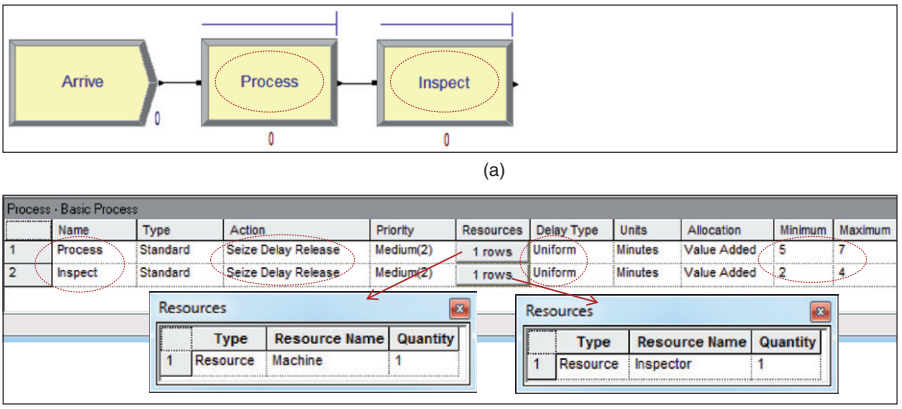


Fig. 7.6. (a) Flowchart model and (b) spreadsheet for the Process and Inspect modules.

7.2.2.2 Build Process Modules Process and Inspect The next step is to build two Process modules Process and Inspect to model job processing and job inspection, respectively.

1. Be sure that the Create module Arrive is selected (so that Arena automatically connects the Process module to the Create module).
2. Drag the Process module in the Basic Process panel into the flowchart model window. A default name, Process1, is given to the module when it is placed.
3. Be sure that the Process module Process1 is selected, and drag the Process module once more from the Basic Process panel into the flowchart model window. A default name, Process2, is given to the module when it is placed.
4. Edit the spreadsheet to define the two Process modules (see Fig. 7.6): (1) Set Names to Process and Inspect; (2) set Action to Seize Delay Release; (3) define Resources as Machine and Inspector; (4) set Delay Type to

Uniform, (5) set Units to Minutes, (6) set Minimum to 5 and 2, and (7) set Maximum to 7 and 4.

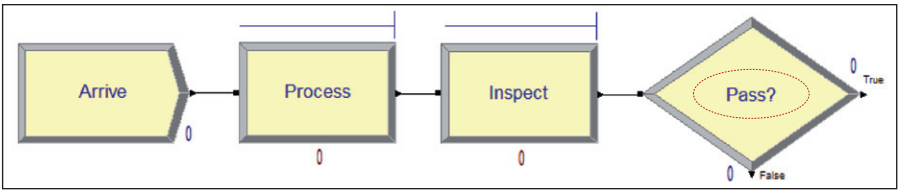
7.2.2.3 Build Decide Module Pass? The third step is to build the Decide module Pass? to model the probabilistic branching.

1. Be sure that Process module Inspect is selected.
2. Drag the Decide module from the Basic Process panel into the flowchart model window. A default name, Decide1, is given to the module when it is placed.
3. Edit the spreadsheet to define the Decide module (see Fig. 7.7) as: Set Name to Pass? and set Percent True to 98.

7.2.2.4 Build Dispose Modules Delivered and Scrapped The last step is to build two Dispose modules, Delivered and Scrapped.

1. Be sure that the Decide module Pass? is selected.
2. Drag the Dispose module into the model window (Name Dispose1 is given).
3. Drag the Dispose module into the model window (Name Dispose2 is given).
4. Edit the spreadsheet (see Fig. 7.8): Set Names to Delivered and Scrapped.

7.2.2.5 How to Manually Connect One Module to Another? If no connection is automatically made between a selected "from" module and a newly added "to" module, you can connect the two modules manually. Click the Object > Connect menu in the menu bar to draw a connection. Your cursor will change to a cross hair. Start the connection by clicking the exit point (▶) of the "from" module, then click the entry point (■) of the "to" module to



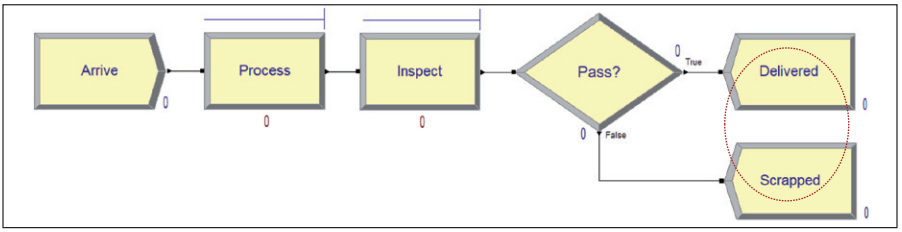
(a)

Decide - Basic Process

	Name	Type	Percent True
1	Pass?	2-way by chance	98

(b)

Fig. 7.7. (a) Flowchart model and (b) spreadsheet for the Decide module Pass?



(a)

Dispose - Basic Process

	Name	Record Entity Statistics
1	Delivered	☑
2	Scraped	☑

(b)

Fig. 7.8. (a) Flowchart model and (b) spreadsheet for Delivered and Scrapped modules.

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	☑
2	Inspector	Fixed Capacity	1	0.0	0.0	0.0		0 rows	☑

Queue - Basic Process

	Name	Type	Shared	Report Statistics
1	Process.Queue	First In First Out	☐	☑
2	Inspect.Queue	First In First Out	☐	☑

Fig. 7.9. Initial default static model of the process-inspect line.

complete the connection. If you need to make multiple connections, simply select Object > Connect twice (the Connect button will remain active, and it is in multi-connect mode). Then draw as many connections as desired. A valid connection target (e.g., entry point, exit point, or operand object) will be highlighted when the pointer hovers over the target. To end the multi-connection session, click again on the Connect option or press Esc.

7.2.3 Completing a Static Model of a Process-Inspect Line

As shown earlier in Fig. 7.3(b), the static model provides detailed descriptions of the resources and queues in the system. Figure 7.9 shows the default static model of a process-inspect line that is generated when the Arena flowchart model is constructed. The Queue data modules, Process.Queue and Inspect.Queue, are automatically generated when the Process modules are defined as in Fig. 7.6(b).

The static model consists of a Resource data module and a Queue data module from the Basic Process panel. Information in the initial default static model is provided automatically by the Arena system, and additional details can be entered by the user in the spreadsheets (or dialog boxes). In this

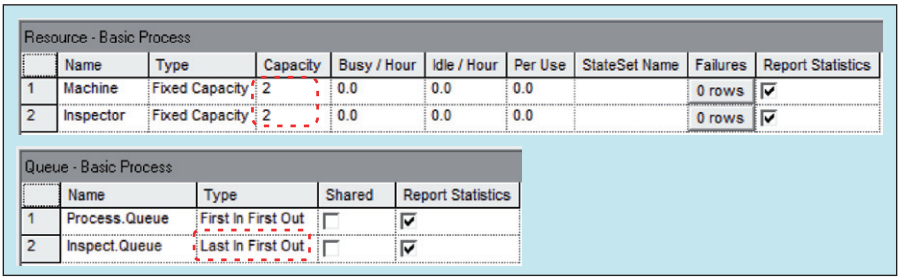


Fig. 7.10. Final static model of the process-inspect line.

example, the default static model is sufficient for the reference model of our process-inspect line [see Fig. 7.3(a)].

If the intended static model requires more information than the default static model, data modules for the resources and queues are brought in to the Spreadsheet Model window where the required information can be entered. Figure 7.10 shows an example where (1) the capacity of the resources is increased to two in the Resource data module and (2) the queue discipline of the Inspect queue is changed to Last-In-First-Out (in the Queue data module). Now we have a process-inspect line with two machines and two inspectors, and jobs are selected for inspection on a last-in-first-out basis.

7.2.4 Arena Simulation and Output Reports

7.2.4.1 Prepare for the Simulation of the Process-Repair Line Before executing a simulation run for the Arena model, we need to specify general project information including the duration of the simulation run. Here, we will perform a short, 24-hour run.

1. Figure 7.11(a): Open the Run Setup dialog box by using the Run > Setup menu and clicking the Project Parameters tab. In the Project Title field, type "Process-Inspect Line Analysis"; we will use the default values for the Statistics Collection check boxes, with Entities, Queues, Resources, and Processes checked.
2. Figure 7.11(b): Click the Replication Parameters tab. In the Replication Length field, type "24"; and set its Time Units to Hours from the drop-down list. Click OK to close the dialog box.

7.2.4.2 Save the Simulation Model The simulation model is now ready for a simulation run, and it is a good time to save the model. Select the File > Save menu item in the menu bar. Arena will prompt you for a destination folder and file name. Browse to the target folder in which you want to save the model and type a name in the File Name field.

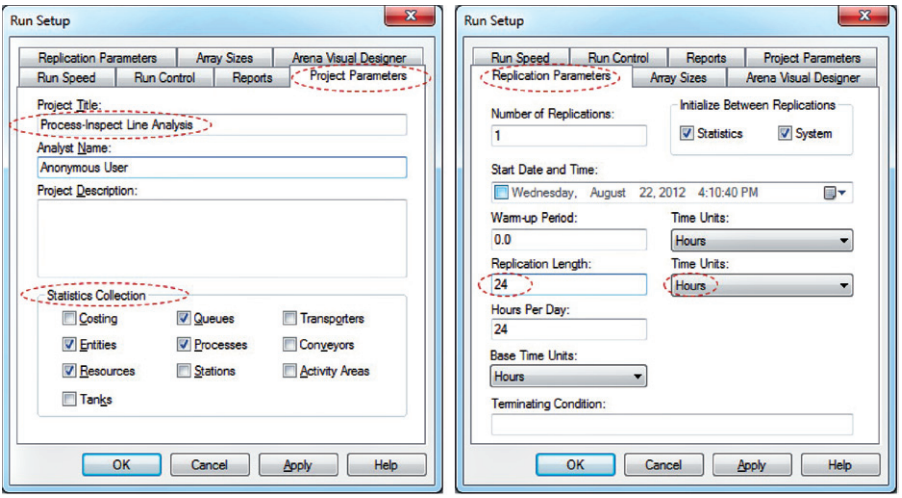


Fig. 7.11. Run Setup dialog box: (a) Project Parameters; (b) Replication Parameters.

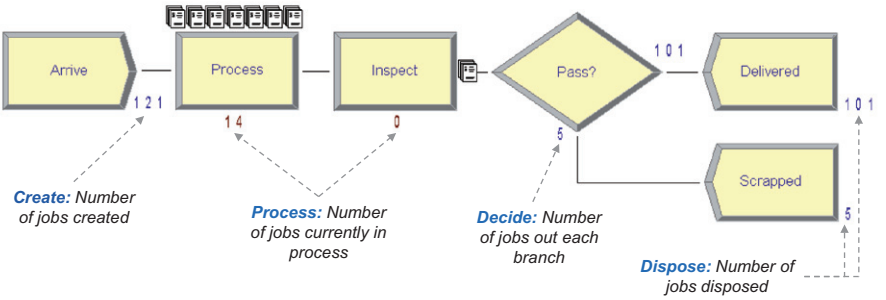


Fig. 7.12. Automatic flowchart animation during the simulation run.

Arena's model files store all of the model definition, including the flowchart, other graphics you have drawn, and the module data you entered into the spreadsheets. When you perform a simulation run, the results are stored in a database using the same name as the model file.

7.2.4.3 Run the Simulation Start a simulation run by clicking Run > Go menu item in the menu bar. Arena first will check validity of your model, and then launch the simulation. As the simulation progresses, you will see small entity pictures (resembling a document) moving along the flowchart. Also, several variables appear in the flowchart and change their values as entities are created and processed, as shown in Fig. 7.12.

If Arena displays an error message, you can use the Find button in the error window to locate the source of the error. You can change between the error window and model windows using the window menu in the menu bar.

If the animation is too fast, you can slow it down by adjusting the animation scale factor. Use the less-than (<) key during the run to decrease the scale factor by 20%. Pressing the "<" key repeatedly is an easy way to fine tune the animation speed. The greater-than (>) key speeds up animation by 20%. Be sure that the model window is active, not the Navigate panel, or the ">" and "<" keys will not take effect.

To pause the simulation run, press the Esc key. As shown in Fig. 7.12, you can see how many entities (jobs) have been created, are currently in the Process modules Process and Inspect, have left each branch of the Decide module Pass?, and have left the model at each of the terminating Dispose modules Delivered and Scrapped. These variables can be helpful in verifying the model.

7.2.4.4 View Simulation Reports You may skip the animation and run right to the end of the simulation to view the reports. Pause the simulation (i.e., flowchart animation), then click the Fast Forward button to run the simulation without updating the animation. At the end of the run, Arena will ask whether you want to view reports. Click Yes, and the default report (the Category Overview Report) will be displayed in a report window, as shown in Fig. 7.13

7.2.5 Arena Modules

Arena provides two types of modules—flowchart modules and data modules—and they define the process to be simulated. All information required to simulate a process is stored in modules. Flowchart modules—those that are placed

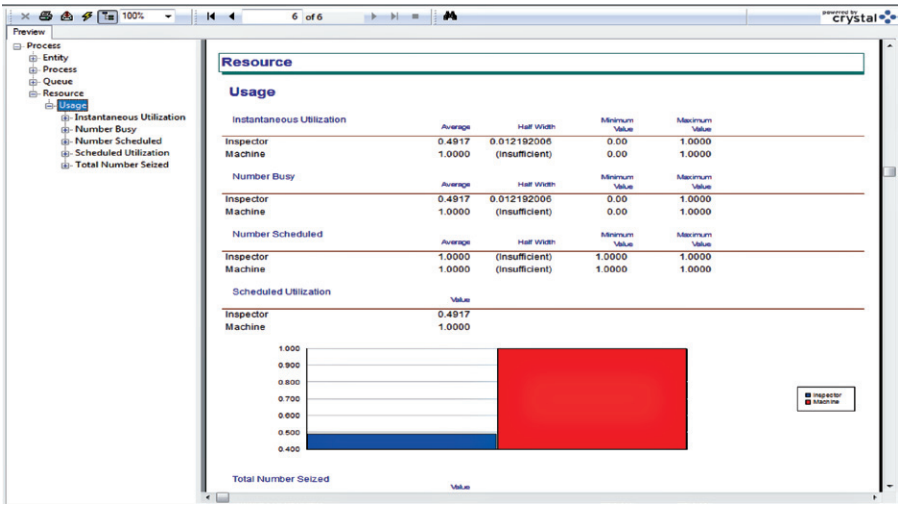


Fig. 7.13. Arena reports of the process-inspect line simulation run.

in the flowchart model window—describe the dynamics of the system. Data modules—those that are used in declaring objects (entities, resources, and queues) in the spreadsheet model—define the static model of the system. Arena modules are grouped into three panels: Basic Process, Advanced Process, and Advanced Transfer panels.

7.2.5.1 Flowchart Modules in the Basic Process Panel The flowchart modules in the Basic Process panel are high-level modules that are used in modeling a system at a higher level (i.e., at a lower resolution). There are eight flowchart modules in this group, as listed below.

1. The Create () module is a starting point of a process flow. Entities enter the simulation and their type is specified.
2. The Dispose () module is the end of a process flow. Entities are removed from the simulation.
3. The Process () module defines an activity, usually performed by one or more resources; processing activity requires some time to complete, i.e., processing time.
4. The Decide () module defines branching in a process flow. Only one branch is taken according to a decision rule.
5. The Batch () module collects a number of entities before they can continue in a process flow.
6. The Separate () module duplicates entities for concurrent or parallel processing, or separates a previously established batch of entities.
7. The Assign () module changes the value of the entity's attribute or model variable during simulation.
8. The Record () module collects statistics, such as an entity count or cycle time.

7.2.5.2 Flowchart Modules in the Advanced Process Panel The flowchart modules in the Advanced Process panel are used in modeling a system at a lower level (i.e., at a high resolution). There are 14 modules in this group, and six frequently used modules are listed below.

1. The Seize () module allocates units of one or more resources to an entity.
2. The Delay () module delays an entity by a specified amount of time.
3. The Release () module releases the specified units of a resource that an entity previously has seized.
4. The Hold () module holds an entity in a queue to either wait for a signal, wait for a specified condition to become true, or be held indefinitely.

5. The Match ( ) module brings together a specified number of entities waiting in different queues.
6. The Search ( ) module searches a queue or a group (batch) to find the rank of an entity or the value of the global variable J that satisfies the specified search condition.

7.2.5.3 Flowchart Modules in the Advanced Transfer Panel The flowchart modules in the Advanced Transfer panel are materials-handling modules that are used in modeling the movements of entities in the system. There are 17 modules in this group, and nine of them are listed below.

1. The Station ( ) module defines a station (or a set of stations) corresponding to a physical or logical location where processing occurs.
2. The Route ( ) module transfers an entity to a specified station, or the next station in the station visit sequence defined for the entity.
3. The Access ( ) module allocates one or more cells of a conveyor to an entity for movement from one station to another.
4. The Convey ( ) module moves an entity on a conveyor from its current station to a specified destination station.
5. The Exit ( ) module releases the cells on the specified conveyor that have been allocated to an entity.
6. The Allocate ( ) module assigns a transporter (i.e., vehicle) to an entity without moving it to the entity's station location.
7. The Move ( ) module advances a transporter from one location to another without moving the controlling entity to the destination station.
8. The Transport ( ) module transfers the entity to a destination station. After the transport time delay, the entity reappears in the model at the destination station module.
9. The Free ( ) module releases the entity's most recently allocated transporter unit.

7.2.5.4 Data Modules Data modules are used for defining the static Arena model. Major components are the Resource module and the Queue module (see Fig. 7.10). Data modules are grouped into Basic Process, Advanced Process, and Advanced Transfer types. There are seven data modules in the Basic Process panel as listed below:

1. The Entity data module defines entity types in a simulation.
2. The Resource data module defines the resources including resource availability.
3. The Queue data module defines the queues in the system.

4. The Schedule data module defines (1) an operating schedule of a resource (with the Resource module), or (2) defines an arrival schedule with the Create module.
5. The Set data module defines various types of sets, including resource, counter, tally, and entity type and entity picture.
6. The Attribute data module defines the entity's attributes, which consist of type, dimensions, and initial value.
7. The Variable data module defines the variables used across the modules in the model.

There are eight data modules in the Advanced Transfer panel, and six of them are listed below: (1) the first one is used for a transfer without a transporter, (2) the next two modules for conveyors, and (3) the remaining three for guided transporters.

1. The Sequence data module is used to define a sequence for entity flow through the model. A sequence consists of an ordered list of stations that an entity will visit.
2. The Conveyor data module allows the definition of a conveyor for entity movement between stations.
3. The Segment data module defines the distance between two stations in the segment set of a conveyor.
4. The Transporter data module allows the definition of a free-path for guided transporters traveling on a network defined in the Network and Network Link modules.
5. The Network data module defines a network that guided transporters will follow. A network encompasses a set of links specified in its Network Links repeat group.
6. The Network Link data module defines the characteristics of a guided transporter path. The Network module then references a set of network links to define a network that guides transporters follow.

7.3 ACTIVITY CYCLE DIAGRAM-TO-ARENA CONVERSION TEMPLATES

This section aims to establish a systematic procedure for converting an ACD model to an Arena model. We show you how to build Arena models for the ACD models of various "template systems" developed in Chapter 6, Section 6.3. The resulting Arena model templates can be used as building blocks for modeling larger systems with Arena. A complete list of the Arena model templates can be found in the official website of this book (http://VMS-technology.com/Book/Arena).

As before, our baseline system is a single server system with unlimited waiting space. Figure 7.14 is an ACD-Arena mapping diagram that shows (1) the ACD model of a single server system, (2) an Arena flowchart model, (3) an Arena static model, and (4) the mapping relationships from the ACD model components to the Arena model components. The ACD model has two activities {Arrive, Process} and four queues {A, B, M1, Job}. The Process activity denotes an actual processing operation with a time delay (i.e., a nonzero processing time). Among the queues, B denotes the number of entities in the buffer, M1 denotes the number of available resources, and Job is a source queue. The Arena flowchart model for this ACD model has three flowchart modules as follows:

1. The Create module is a starting point of process flow where entities enter the simulation and their type is specified.
2. The Dispose module is an end point of process flow where entities are removed from the simulation.
3. The Process module represents an activity, usually performed by one or more resources and requiring some time to complete. Uniform random variate is denoted by Uni(a,b) or U(a,b).

In an ACD model, an activity with a nonzero time duration is called a *timed activity*. A queue that represents the number of jobs in a buffer is called a *buffer queue*, and that which denotes the number of available resources is called a *resource queue*.

There are five ACD-to-Arena mapping relationships for this single server system example, which are indicated by the curved arrows in the figure. They are:

- The job creator cycle in the ACD model maps to a Create module (Arrive).
- The timed activity Process maps to a Process module (Process).
- The arc back to the source queue Job maps to the Dispose module (Dispose).
- The resource queue M1 maps to the Resource data module of the static model.
- The buffer queue B maps to the Queue data module of the static model.

The resulting Arena model in Fig. 7.14 has three flowchart modules (Create, Process, and Dispose) and two *data modules* (*Resource* and *Queue*). With an Arena model obtained, the *Arena simulation program* for the model is constructed by following the steps previously described in Sections 7.2.2 and 7.2.3. In the following, examples of building Arena models from ACD models for various template systems are presented.

7.3.1 Template for Fixed Multi-Server Modeling

A single server system is a single station system in which the station has only one server. A multi-server system is a single station system with multiple

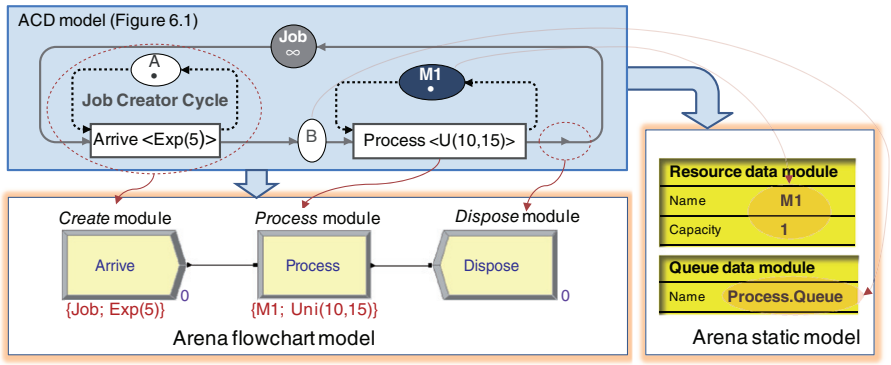


Fig. 7.14. ACD-Arena mapping diagram for a single server system.

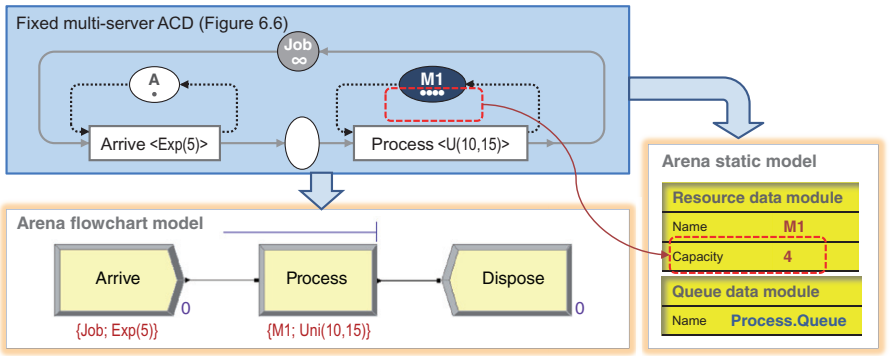


Fig. 7.15. ACD-Arena mapping diagram for a fixed multi-server system.

servers. If the number of servers in the station is fixed, we have a fixed multi-server system; if it varies over time, we have a flexible multi-server system.

Figure 7.15 shows an ACD-Arena mapping diagram for a fixed multi-server system. The Arena flowchart model and static model of the fixed multi-server system are exactly the same as those of the single server system except the Capacity field of the Resource data module in the Arena static model. The resource capacity in the fixed multi-server model is set to 4. In the following, a step-by-step procedure for building an Arena simulation program will be explained.

The first step of preparing an Arena program for simulating the fixed multi-server model is to (1) generate a flowchart model consisting of three flowcharts modules (Create, Process, and a Dispose) and (2) assign their names as Arrive, Process, and Dispose, as shown in the flowchart model window of Fig. 7.16.

The second step is to provide attribute values of each flowchart module. Shown in the spreadsheet model window (inside the dashed-line rectangle) of Fig. 7.16 is the spreadsheet for the Arrive flowchart module. In the spreadsheet, the inter-arrival time distribution is specified as Value = 5, and Units = Minutes.

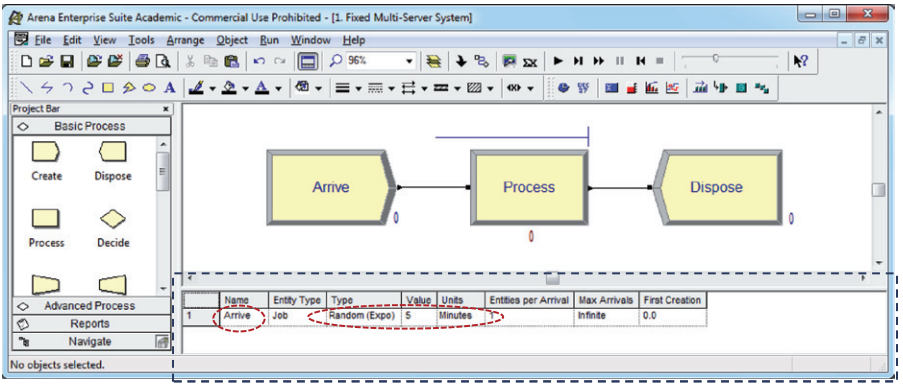


Fig. 7.16. Generation of the flowchart model of the fixed multi-server system.

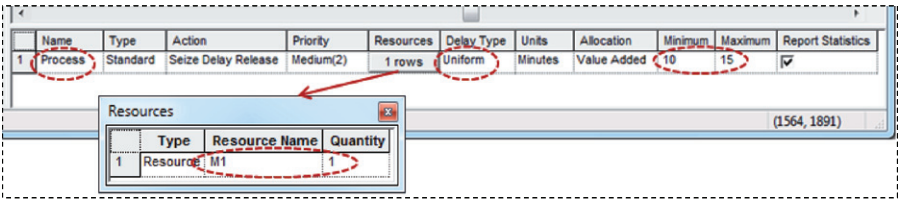


Fig. 7.17. Spreadsheet for providing attribute values of the Process module.

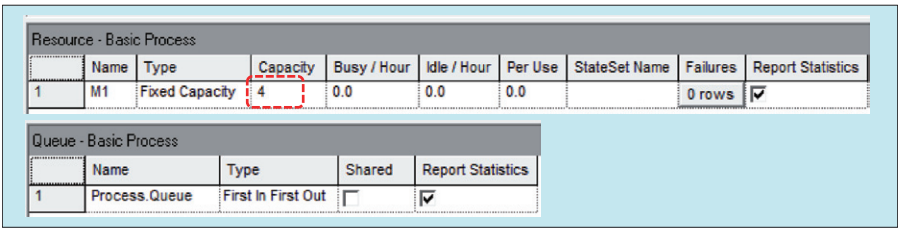


Fig. 7.18. Completed static model of the fixed multi-server system.

Likewise, the attribute value Process for module is specified. Figure 7.17 shows the spreadsheet for the flowchart module Process in which the service time distribution is specified as Delay Type = Uniform, Minimum = 10, and Maximum = 15. Also defined in the spreadsheet (in the popped up window Resources) are Resource Name = M1 and Quantity = 1.

The third and last step for building the Arena simulation program is to complete the Arena static model by providing additional information to the data modules. In this example, the number of servers needs to be provided. Figure 7.18 shows that the number of servers (=4) is specified in the Capacity field of the Resource data module.

TABLE 7.1. A Daily Resource Schedule with 2-Hour Intervals

Hours (120 min)	0000-0159	0200-0359	0400-0559	0600-0759	0800-0959	1000-1159	1200-1359	1400-1559	1600-1759	1800-1959	2000-2159	2200-2359
k	0	1	2	3	4	5	6	7	8	9	10	11
N[k]	0	0	0	0	3	3	3	5	3	1	1	0

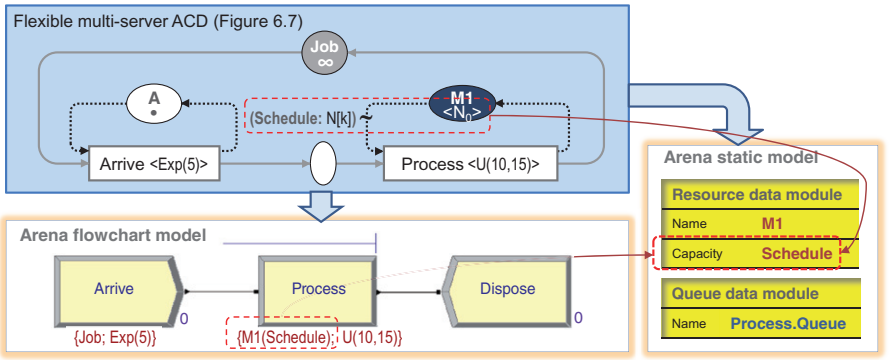


Fig. 7.19. ACD-Arena mapping diagram for a flexible multi-server system.

7.3.2 Template for Flexible Multi-Server Modeling

When the number of servers in a multi-server system changes over time, it is called a *flexible multi-server system*. The number of servers in a flexible multi-server system changes according a resource schedule. Table 7.1 shows a daily resource schedule (from Fig. 6.21) in which the number of servers changes every 2 hours.

Figure 7.19 shows the ACD model of a flexible multi-server system (Fig. 6.7 in Section 6.3.1), an Arena flowchart model, an Arena static model, and the mapping relationships from the ACD model components to the Arena model components. The Arena model of the flexible multi-server system is obtained by modifying the fixed multi-server system in the Capacity field of the Resource data module in the Arena static model. The resource capacity is now determined according the daily resource schedule given in Table 7.1. In the ACD model of Fig. 7.19, N_0 denotes the initial number of resources, and N[k] is the number of resources at the k-th time interval as defined in Table 7.1.

The procedure for constructing an Arena simulation program for the flexible multi-server system is similar to the fixed multi-server case described in Section 7.3.1. The only difference is the Capacity of the Resource M1 in the static model. Figure 7.20 shows how to define the resource schedule using the Arena data modules: (1) The Resource data module is selected from the Basic Process panel, and then its Type field is set to Based on Schedule and the Schedule Name field to MachineSchedule; (2) the Schedule data module is

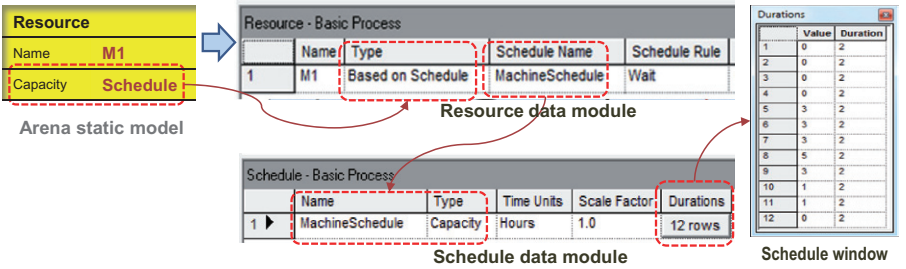


Fig. 7.20. Method of defining the resource schedule of a flexible multi-server system.

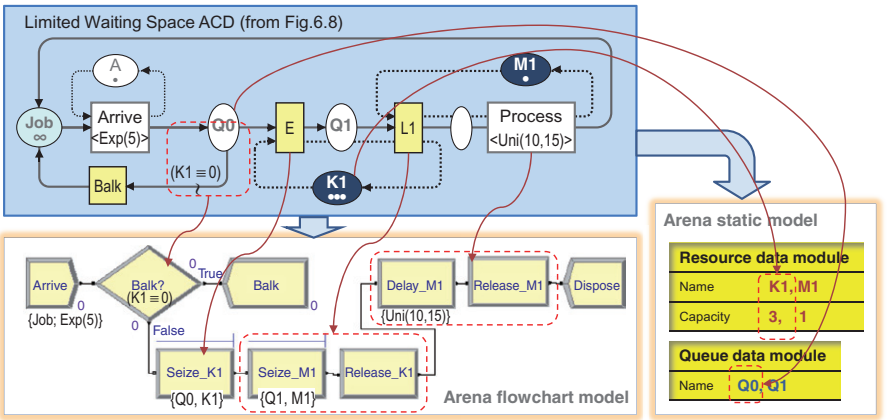


Fig. 7.21. ACD-Arena mapping diagram for a limited waiting space system (balking).

selected, and then its Name field and Type field are set to MachineSchedule and Capacity, respectively; (3) the Durations field of the Schedule data module is set to 12 rows to bring up a Schedule window having 12 rows; and (4) the resource schedule is defined as Value-Duration pairs as depicted in the Schedule window.

7.3.3 Template for Balking (Conditional Branching) Modeling

In section Chapter 6, Section 6.3.2, a single server system with balking was presented. As discussed in Chapter 4 (Section 4.4), balking occurs when the waiting space in front of a server station is full. Figure 7.21 shows an ACD-Arena mapping diagram for a limited waiting space system. In the ACD model in Fig. 7.21, (1) K1 denotes the number of the empty slots in the waiting space; (2) M1 denotes the number of idle machines; (3) Q0 denotes a virtual waiting space for the arriving jobs; and (4) Q1 denotes the real waiting space for the jobs that entered in the system. K1 and M1 are resource queues (K1 can be regarded as a capacity queue as well), and Q0 and Q1 are buffer queues.

In an ACD model, an activity that requires no time delays is called an *instant activity*. The ACD model in Fig. 7.21 has three instant activities {Balk, E, L1}, in addition to the timed activity Process and the create activity Arrive. An instant activity is used for modeling the state change (i.e., event) of an entity, such as entering or exiting a buffer, loaded on a machine, or accessing a conveyor. An instant activity may represent enter activity by which an entity enters a buffer, exit activity by which an entity exits a buffer, load activity by which an entity is loaded on a machine, etc. In the ACD model in Fig. 7.21, E is an enter activity and L1 is a load/exit activity.

The Arena flowchart model in Fig. 7.21 has four types of newly introduced modules: one Decide module {Balk?}, two Seize modules {Seize_K1, Seize_M1}, two Release modules {Release_K1, Release_M1}, and a Delay module {Delay_M1}.

- A Decide module is a branching point in process flow. Only one branch is taken according to a decision rule.
- A Seize module allocates units of one or more resources to an entity.
- A Delay module delays an entity by a specified amount of time.
- A Release module releases units of a resource that an entity previously has seized.

As indicated in Fig. 7.21 by the curved arrows, there are five newly introduced mapping relationships from the ACD model to the Arena model:

- The conditional branching at Q0 maps to a Decide module (Balk?).
- The enter activity E maps to a Seize module (Seize_K1).
- The load/exit activity L1 maps to a Seize-Release module pair.
- The timed activity Process connected to a load activity (L1) maps to a Delay-Release module pair {Delay_M1, Release_M1}.
- The buffer queues {Q0, Q1} map to the Queue data module in the static model.
- The resource queues {M1, K1} map to the Resource data module.

Having generated the Arena flowchart model shown in Fig. 7.21, the next step is to provide the attribute values of the flowchart modules. Figure 7.22(a) shows the spreadsheet of the "by condition" Decide module where the attribute values are provided as Name = Balk? and Value = NR(K1) == 3. The remaining fields are filled with default values by Arena. NR(K1) is a built-in Arena function: NR(K1) = the number of K1 resources currently being used.

Thus, NR(K1) == 3 means that all three units of K1 resources are being used (i.e., no room in Q1). Figure 7.22(b) shows the Seize module spreadsheet where the attribute values of the Seize modules are provided.

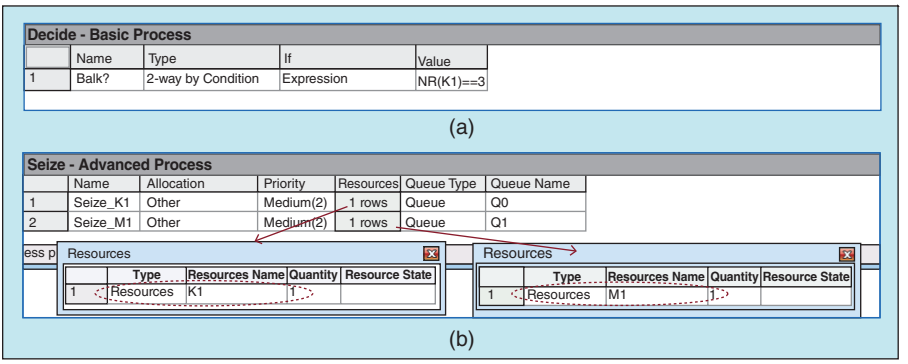


Fig. 7.22. Decide and Seize spreadsheets for providing the attribute values.

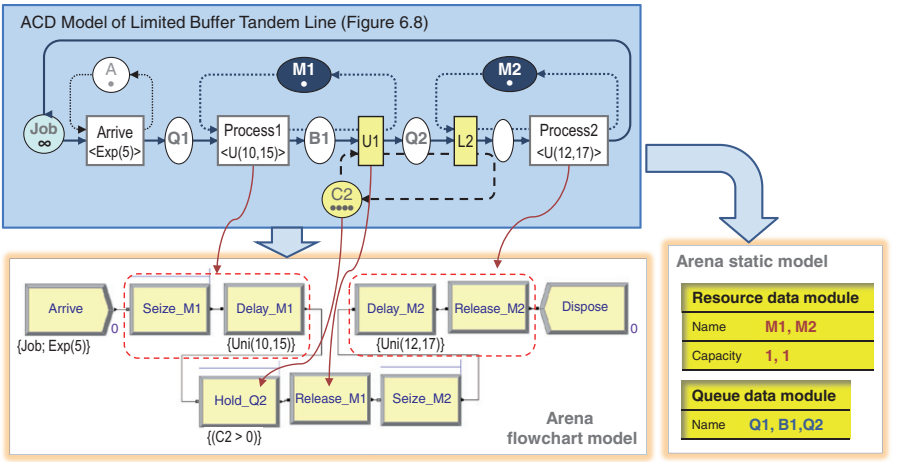
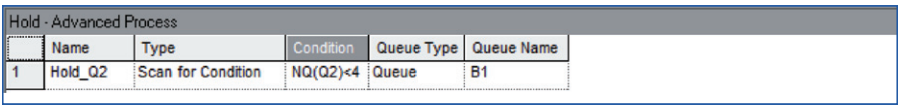


Fig. 7.23. ACD-Arena mapping diagram for a limited buffer tandem line (blocking).

7.3.4 Template for Limited Buffer Tandem Line Modeling

As discussed in Chapter 4 (Section 4.4.2.1), blocking may occur if the unloading space of a machine is full. Figure 7.23 shows an ACD-Arena mapping diagram for a limited buffer tandem line. The ACD model, which is a part of the limited buffer tandem line ACD model introduced in Chapter 6 Section 6.3.2 (Fig. 6.8), consists of five activity nodes and nine queue nodes. Among the nine queues in the ACD model, (1) M1 and M2 are resource queues; (2) Q1, B1, and Q2 are buffer queues; and (3) C2 is a capacity queue. This model has two instant activities {U1, L2} and two timed activities {Process1, Process2}. The instant activity U1 is an *unload* activity, and L2 is a *load* activity.


Hold - Advanced Process

	Name	Type	Condition	Queue Type	Queue Name
1	Hold_Q2	Scan for Condition	NQ(Q2)<4	Queue	B1

Fig. 7.24. Spreadsheet for inputting the attribute values of the Hold module.

The Arena flowchart model in Fig. 7.23 has two Seize modules {Seize_M1, Seize_M2}, two Delay modules {Delay_M1, Delay_M2}, two Release modules {Release_M1, Release_M2}, and a Hold module Hold_Q2. A brief description of the newly introduced Hold module: The Hold module holds an entity in a queue until a specified condition becomes true.

Curved arrows in Fig. 7.23 indicate key ACD-to-Arena mapping relationships as follows:

- The timed activity Process1 followed by an unload activity (U1) maps to the Seize-Delay module pair (Seize_M1 & Delay_M1).
- The capacity queue C2 maps to a Hold module (Hold_Q2).
- The unload activity U1 maps to a Release module (Release_M1).
- The timed activity Process2 following a load activity (L2) maps to the Delay-Release module pair {Delay_M2, Release_M2}.

Now, the next step is to provide the attribute values of each and every of the flowchart modules. Figure 7.24 shows the spreadsheet for the newly introduced Hold module where its attribute values are provided as Name = Hold_Q2, Condition = NQ (Q2) < 4, and Queue Name = B1. NQ(Q2) is a built-in Arena function returning: NQ(Q2) = the current number of entities in the buffer queue Q2.

Exercise 7.1. Revise the Arena model in Fig. 7.23 by treating C2 in the ACD model as a resource queue (instead of a capacity queue).

7.3.5 Template for Nonstationary Arrival Process Modeling

In Chapter 6, Section 6.3.3 (see Fig. 6.9), we presented the ACD model of a single server system having inter-arrival times sampled from a nonstationary Poisson process. In the ACD model, the thinning method (see Chapter 3, Section 3.4.3) was explicitly implemented to generate nonstationery inter-arrival times. The same thinning method is used in Arena.

Let's assume that the mean arrival rates over a 24-hour period are as given in Table 7.2. Figure 7.25 shows a Create module spreadsheet in which the job creation Type is set to Schedule, and the mean arrival rates of the nonhomogeneous arrival process in Table 7.2 are provided in the Durations window of the Schedule data module. For example, POIS (0.3) indicates that the mean of the Poisson distribution is 0.3.

TABLE 7.2. Mean Arrival Rates (Arrivals per Minute) over a 24-Hour Period

Hours (120 min)	0000-0159	0200-0359	0400-0559	0600-0759	0800-0959	1000-1159	1200-1359	1400-1559	1600-1759	1800-1959	2000-2159	2200-2359
k	0	1	2	3	4	5	6	7	8	9	10	11
R[k]	0.00	0.00	0.00	0.02	0.10	0.30	0.40	0.50	0.40	0.10	0.02	0.00

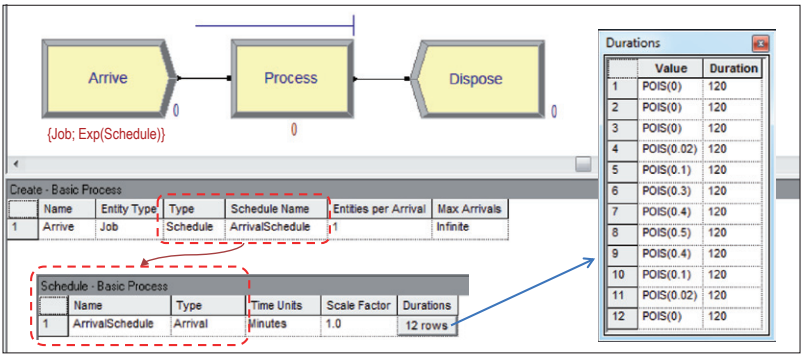


Fig. 7.25. Spreadsheet for inputting the nonstationery arrival rate data.

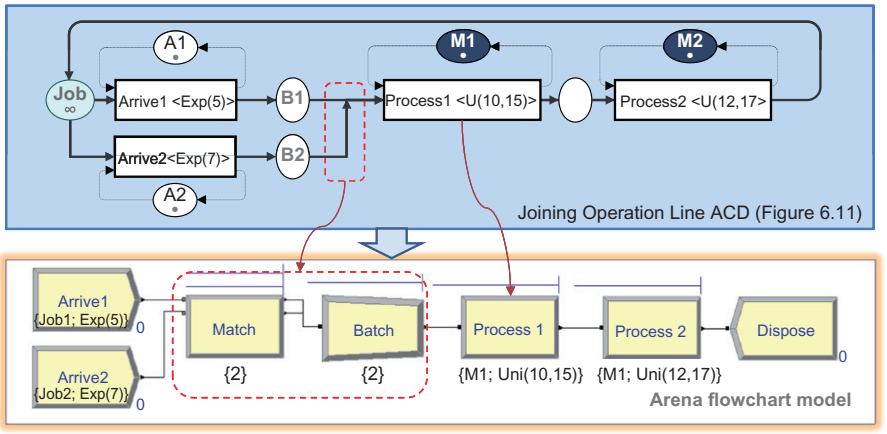


Fig. 7.26. ACD-Arena mapping diagram for a joining operation line.

7.3.6 Template for Joining Operation Modeling

Figure 7.26 shows an ACD-Arena mapping diagram for a joining operation line (Chapter 6, Section 6.3.5, Fig. 6.11). The ACD model has two job creator cycles and two timed activities {Process1, Process2}, together with a merge junction. The Arena model has a Match-Batch module pair, in addition to the

two Create modules and two Process modules. Brief descriptions of the newly introduced Match module and Batch module is given below:

- The Match module brings together a number of entities waiting in different queues.
- The Batch module collects a number of entities before they can continue processing.

In the ACD model of Fig. 7.26, one Type-1 job in the buffer queue B1 and one Type-2 job in the buffer queue B2 are matched together to form a batch of size 2, and then assembled by M1. In Arena, this matching operation is handled by the Match-Batch module pair. Thus, the ACD-to-Arena mapping relationship here can be expressed as the merging of arcs having no arc-multiplicity maps to the Match-Batch module pair in the Arena flowchart model. Figure 7.27 shows a Match module spreadsheet where the Number to Match is set to 2 and a Batch module spreadsheet where the Batch Size is set to 2.

Exercise 7.2. Revise the Arena flowchart model in Fig. 7.26 so as to model the case where two Type-1 jobs and four Type-2 jobs are assembled in the line.

7.3.7 Template for Inspection (Probabilistic Branching) Modeling

Figure 7.28 shows an ACD-Arena mapping diagram for an inspection line (Chapter 6, Section 6.3.6, Fig. 6.12). The ACD model has three timed activities {Inspect, Process, Scrap}, three resource queues {M1–M3}, and three buffer queues {B1–B3}, as well as a probabilistic branching junction. The Arena flowchart model has three Process modules {Inspect, Process, Scrap} and a "by chance" Decide module Decide.

In the ACD model, a conditional branching is made right after the Inspect activity to choose either the Process activity or the Scrap activity. In Arena, this probabilistic branching operation is handled by the "by chance" Decide module. Thus, the ACD-to-Arena mapping relationship here can be expressed as the probabilistic branching at an activity (Inspect) maps to a "by chance" Decide module Decide. Figure 7.29 shows the spreadsheet for inputting the attribute values of the "by chance" Decide module where the Percent True is set to 90.

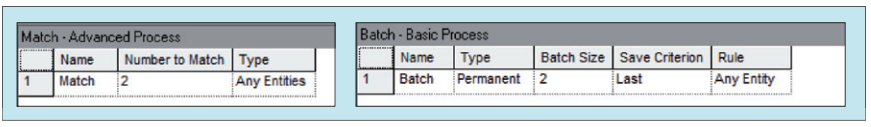


Fig. 7.27. Spreadsheets for defining the Match module and Batch module.

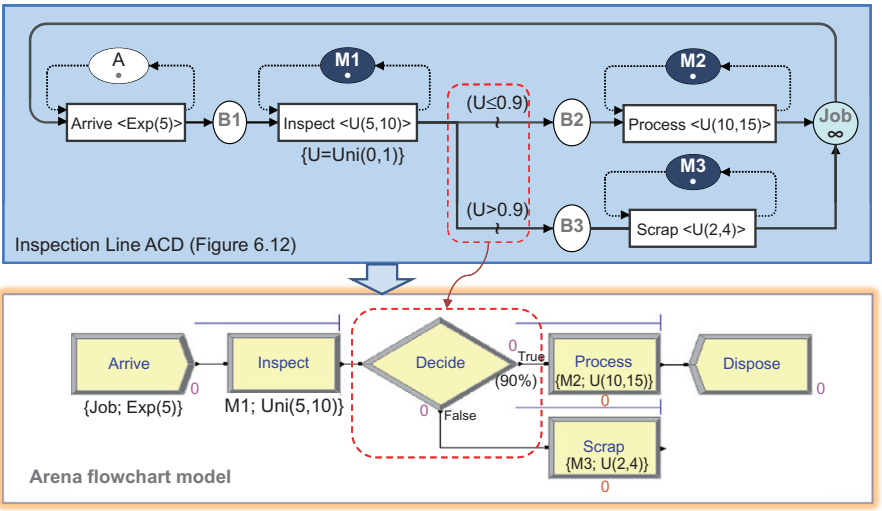


Fig. 7.28. ACD-Arena mapping diagram for probabilistic branching (inspection line).

Fig. 7.29. Spreadsheet for inputting the attribute values of the Decide module.

7.3.8 Template for Resource Failure Modeling

Figure 7.30 shows an ACD-Arena mapping diagram for a single server system with resource failure (Chapter 6, Section 6.3.7, Fig. 6.13). The ACD model assumes that a failure is allowed only when the server is busy (i.e., in use) and the interrupted job is discarded. The service time is 10 (ts = 10), repair time is 50 (tr = 50), and the value of remaining time-to-failure (ttf) is initially set to 1000. The ACD model consists of two activities (Arrive and Process), but its Arena model consists of eight modules: a Create modules, a Seize module, a Decide module, two Assign modules, a Delay module, a Release module, and a Dispose module. This is a brief description of the newly introduced Assign module: the Assign module changes the value of the entity's attribute or model variable during simulation.

At the beginning of the Process activity in the ACD model, the state variables ttf and ts are updated: If (ttf<10) {update-1} else {update-2}. In Arena, the operation "If () {} else {}" is handled by a Decide module, and the operation {update variables} is handled by an Assign module. The Process activity in which an update operation is performed is divided into three Arena

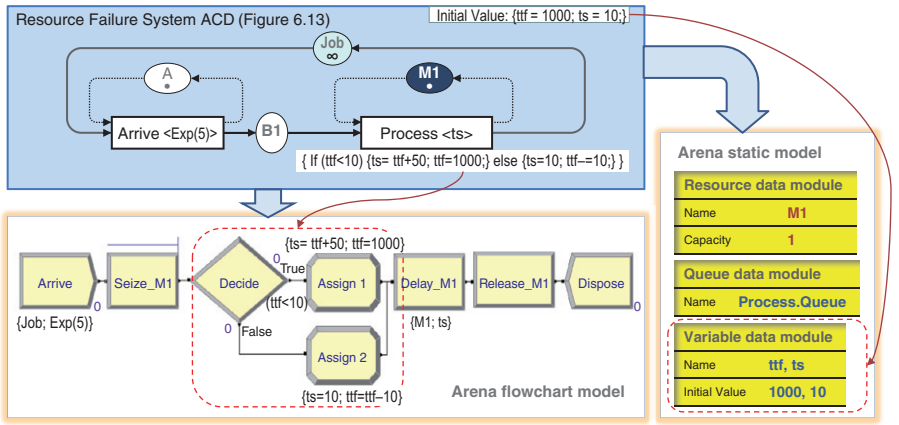


Fig. 7.30. ACD-Arena mapping diagram for a resource failure.

modules: Seize, Delay, and Release modules in the Advanced Process panel, instead of simply using the Process module in the Basic Process panel. In addition, the state variables and their initial values have to be declared in the Variable data module of the Arena static model. Namely:

- A conditional update expression in an activity (Process) of an ACD maps to a Decide-Assign module structure in the Arena flowchart model [Fig. 7.30].
  (1) "**If** () {} else {}" becomes a "2-way by condition" Decide module [Fig. 7.31(a)].
  (2) "{**Variable update expressions**}" becomes an Assign module [Fig. 7.31(b)].
- State variables and their initial values are declared in the Variable data module of the Arena static model [Fig. 7.31(c)].

7.4 ACTIVITY CYCLE DIAGRAM-BASED ARENA MODELING EXAMPLES

In this section, we will show you how to build Arena models from the ACD models of the "example systems." The example systems covered in this section include: a worker-operated tandem line, a restaurant, a simple service station involving flexible multi-servers and nonstationary customer arrivals, a project management system, a simple job shop, and a conveyor-driven serial line. ACD models for the example systems were presented in Chapter 6, Sections 6.4 and 6.5. System modeling is an art that can only be mastered by learning the best

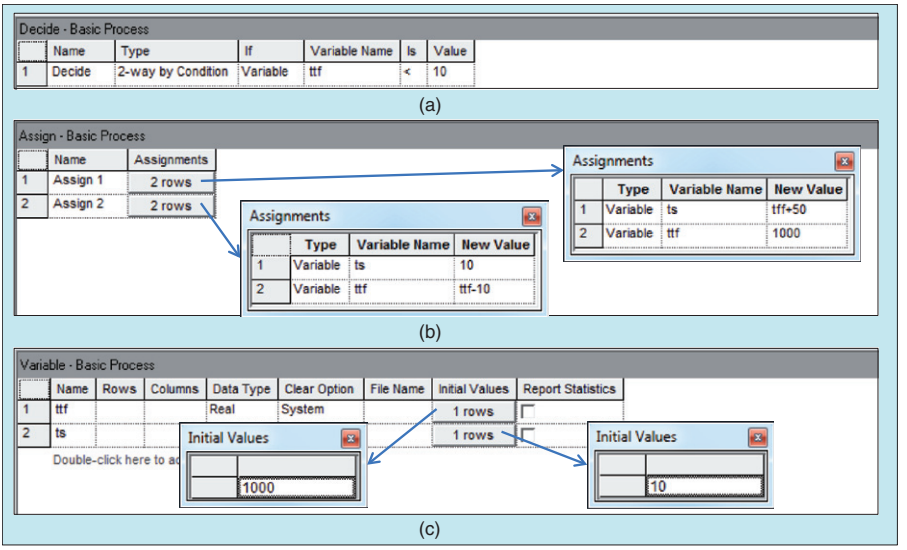


Fig. 7.31. Spreadsheet for defining (a) Decide, (b) Assign, and (c) Variable modules.

practices and internalizing them by relentless practices. If you study the examples carefully and practice with the ACD-based Arena modeling examples provided in the official website of this book (http://VMS-technology.com/Book/Arena), you will gain confidence on modeling real-life systems.

7.4.1 ACD-Based Arena Modeling of a Worker-Operated Tandem Line

Figure 7.32 shows an ACD-Arena mapping diagram for a worker-operated tandem line (the ACD model was reproduced from Fig. 6.15). The ACD model has five resource queues {M1, M2, M3, WA, WB}, three buffer queues {Q1–Q3}, and four timed activities. The fact that the priority of Worker-A is higher than that of Worker-B is denoted by the condition WA ≡ 0 on the arc from the Q1 queue to the Process1b activity. Similarly, the condition M2 ≡ 0 on the arc from the Q2 queue to the Process3 activity denotes that the priority of Machine-2 is higher than that of Machine-3. The Arena flowchart model has two Hold modules, two Decide modules, and four Process modules.

A conditional branching in which the probability of selecting one of the branches is less than one is called an *incomplete conditional branching*. In the ACD model of Fig. 7.32, the conditional branching at Q1 is incomplete and maps to a Hold-Decide module pair {Hold1, Decide1} in the Arena model. Similarly, the conditional branching at Q2 maps to the Hold2 and Decide2 modules. Recall from Section 7.3.3 that a complete conditional branching maps to a Decide module. Thus, the mapping relationship found in this

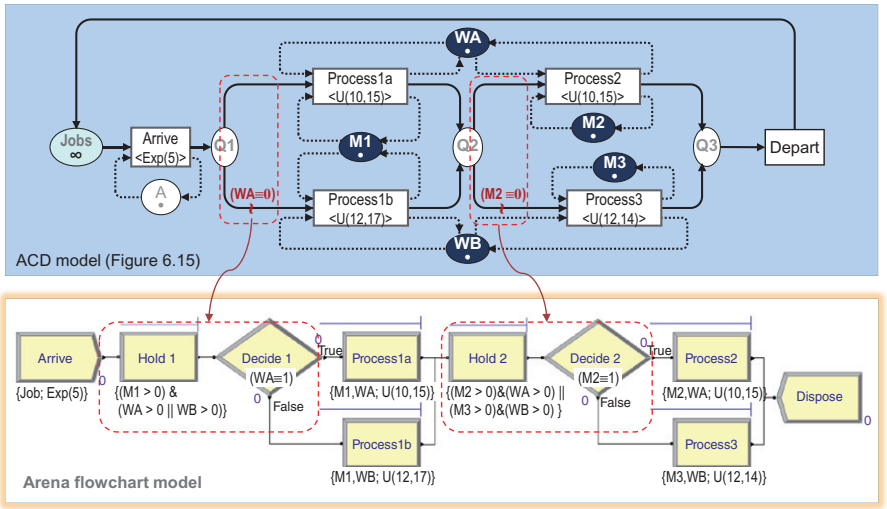


Fig. 7.32. ACD-Arena mapping diagram for a worker-operated tandem line.

Hold - Advanced Process

	Name	Type	Condition	Queue Type	Queue Name
1	Hold 1	Scan for Condition	(NR(M1)==0) && (NR(WA)==0 || NR(WB)==0)	Queue	Q1
2	Hold 2	Scan for Condition	(NR(M2)==0 && NR(WA)==0) || (NR(M3)==0 && NR(WB)==0)	Queue	Q2

Fig. 7.33. Spreadsheet for defining the Hold modules in Fig. 7.32.

particular example may be expressed as the incomplete conditional branching in an ACD model becomes a Hold-Decide module pair in the Arena model.

The condition of the Hold1 module is Machine-1 is available and at least one of Worker-A or Worker-B is available, which is expressed as: (M1 > 0) & (WA > 0 || WB > 0). Similarly, the condition of Hold2 is expressed as ((M2 > 0) & (WA > 0)) || ((M3 > 0) & (WB > 0)). As mentioned in Section 7.3.3, the above Boolean expressions are handled in Arena by using a built-in function NR(resource) that returns the number of busy units for the resource. For example, the Boolean expression M1 ≡ 1 is denoted as NR(M1) ≡ 0.

Having generated the Arena flowchart model, the next step is to provide attribute values of each flowchart module in Fig. 7.32. For example, Fig. 7.33 shows a Hold module spreadsheet where the conditions of the Hold1 and Hold2 modules are defined.

7.4.2 ACD-Based Arena Modeling of Restaurant

Figure 7.34 shows an ACD-Arena mapping diagram for the restaurant system introduced in Chapter 6, Section 6.4.3. The ACD model has three resource

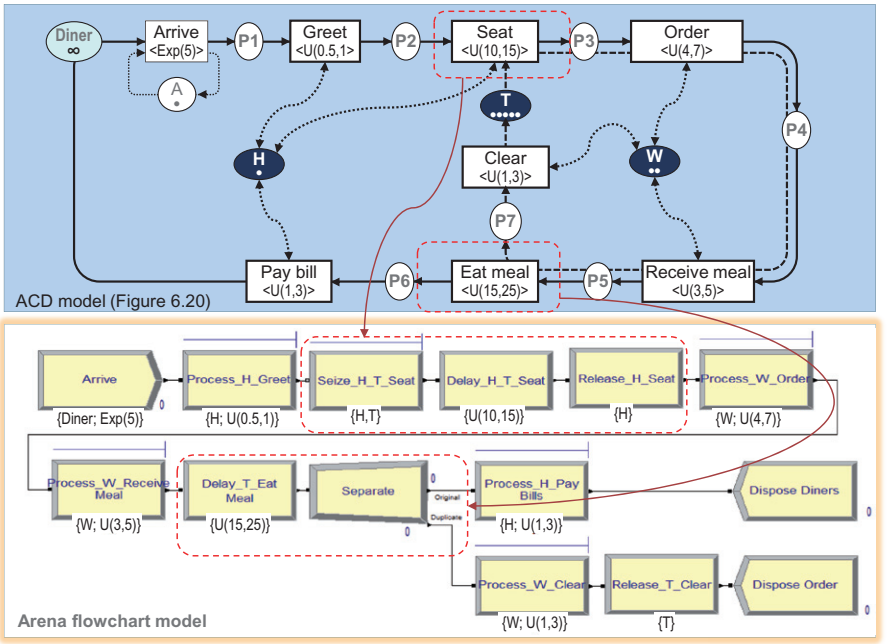


Fig. 7.34. ACD-Arena mapping diagram for a restaurant.

queues {H, T, W}; seven buffer queues {P1–P7}; and seven timed activities {Greet, Seat . . . Pay bill}. There are five tables (T = 5), two waiters (W = 2), and one head waiter (H = 1) in the system. The Arena flowchart model has five Process modules {_H_Greet, _W_Order, _W_Receive meal, _H_Pay bills, _W_Clear}; a Seize module {_H_T_Seat}, two Delay modules {_H_T_Seat, _T_Eat meal}, two Release modules {_H_Seat, _T_Clear}, and a Separate module {Separate}. A brief description of the newly introduced Separate module is that the Separate module duplicates entities for concurrent or parallel processing, or separating a previously established batch of entities.

The ACD-to-Arena mapping relationships for the restaurant model are fairly straightforward except the Seat activity and the Eat meal activity in the ACD model. Starting the Seat activity requires a table T as well as the head waiter H, but only the head waiter is released at the end of the activity. Thus, it maps to a sequence of three modules {Seize_H_T_Seat, Delay_H_T_Seat, and Release_H_Seat}.

The Eat meal activity requires neither a waiter nor a head waiter. At the end of this activity, the diner moves to the Pay bills activity and the table goes through the Clear activity. Thus, assuming that a copy of the diner entity goes with the table, the Eat meal activity maps to a Delay-Separate module pair.

For the Arena flowchart model in Fig. 7.34, we now need to specify attribute values of each flowchart module. As an example, Fig. 7.35 shows the

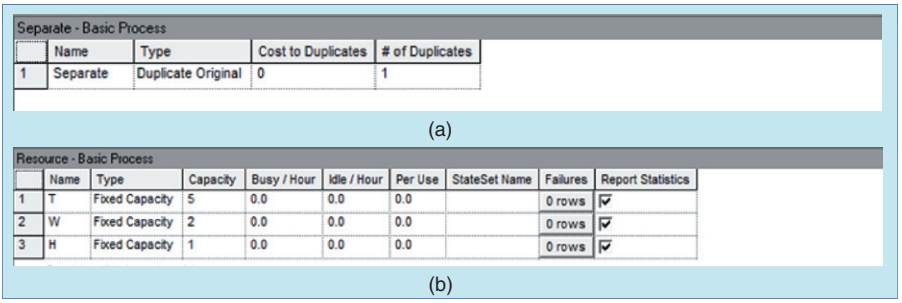


Fig. 7.35. Spreadsheets for defining (a) Separate and (b) Resource data modules.

TABLE 7.3. Schedules for the Number of Resources and Mean Arrival Rates

Hours (120 min)	0000-0159	0200-0359	0400-0559	0600-0759	0800-0959	1000-1159	1200-1359	1400-1559	1600-1759	1800-1959	2000-2159	2200-2359
k	0	1	2	3	4	5	6	7	8	9	10	11
R[k]	0.00	0.00	0.00	0.02	0.10	0.30	0.40	0.50	0.40	0.10	0.02	0.00
N[k]	0	0	0	0	3	3	3	5	3	1	1	0

spreadsheet for defining the Separate module Separate, and the Resource data module spreadsheet for defining the resource capacity values: T = 5, W = 2, and H = 1.

7.4.3 ACD-Based Arena Modeling of a Simple Service Station

As discussed in Chapter 6, Section 6.4.4, a simple service station like a gas station is a flexible multi-server system subject to nonstationery arrival rates. Table 7.3 shows a mean arrival rate schedule and a resource schedule over a 24-hour period.

Combining the results presented in Section 7.3.2 (see Fig. 7.19) and in Section 7.3.5 (see Fig. 7.25), we can easily obtain an Arena simulation model of a simple service station from its ACD model as depicted in Fig. 7.36. The Arena flowchart model consists of a Creator module Arrive, a Process module Process_M1, and a Dispose module Dispose. The Creator module generates jobs with inter-arrival times sampled from a nonstationery Poisson process whose mean arrival rates are given as the Arrival-schedule in Table 7.3. The Process module processes the jobs with varying number of machines as specified in the Machine-schedule.

Figure 7.37 shows attribute values for Create module and Process module. The Schedule Name entry of Create module Arrive is set to ArrivalSchedule, and in Process module Process_M1, the Resource Name entry is set to M1.

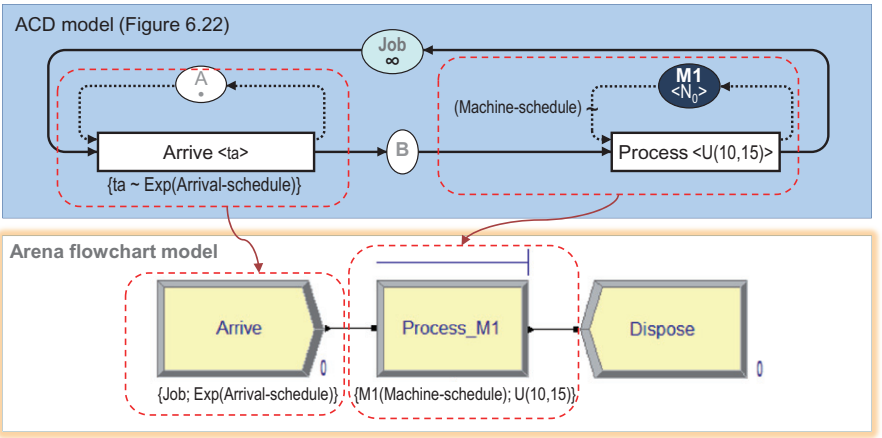


Fig. 7.36. ACD-Arena mapping diagram for a simple service station.

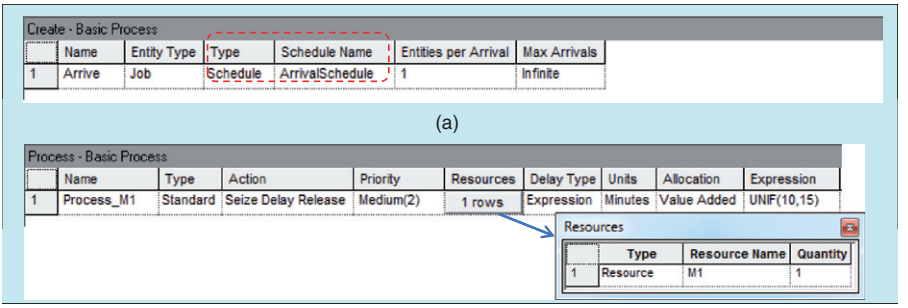


Fig. 7.37. Spreadsheets for defining the Create and Process modules in Fig. 7.36.

The Arena modeling is completed by specifying the schedule of arrival rates and available machines, shown in Table 7.3, using the Resource and Schedule data modules. Recall that these data modules are available in the Basic Process panel. The upper part of Fig. 7.38 shows the spreadsheet for defining the Resource data module in which the Schedule Name entry is set to Machine-Schedule, and the lower part for defining the Schedule data module where the ArrivalSchedule and MachineSchedule are provided in their Durations windows.

7.4.4 ACD-Based Arena Modeling of a Project Management System

Figure 7.39 shows an ACD-Arena mapping diagram for the project management system introduced in Chapter 6, Section 6.4.6. The ACD model consists of nine activities {A1–A9} and 14 queues. Among the queues, 11 are buffer

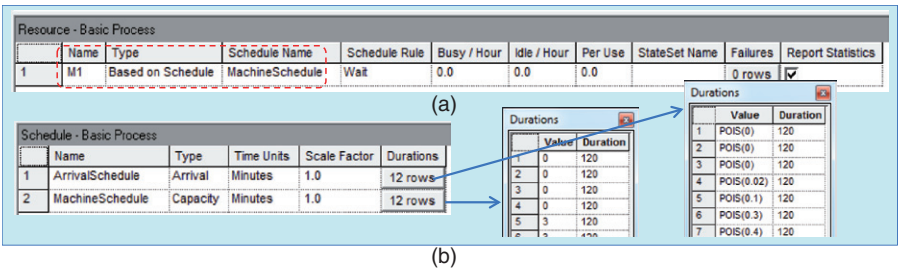


Fig. 7.38. Spreadsheets for defining the Resource and Schedule data modules.

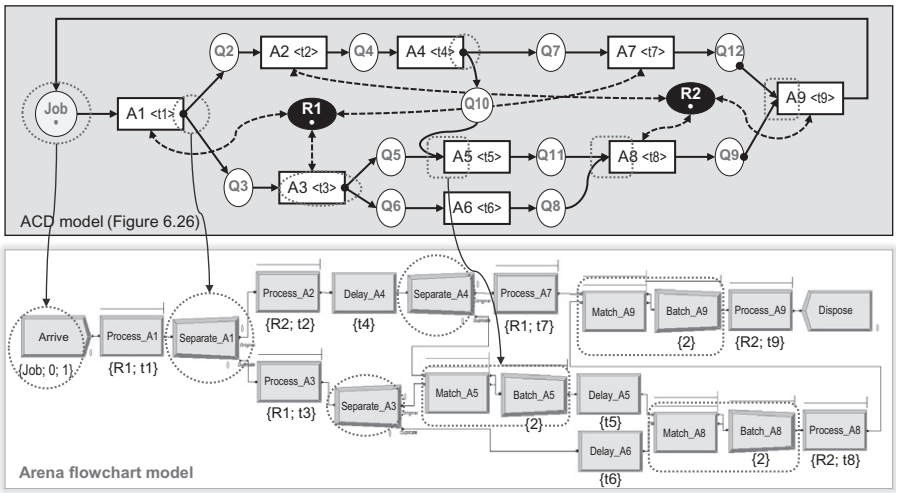
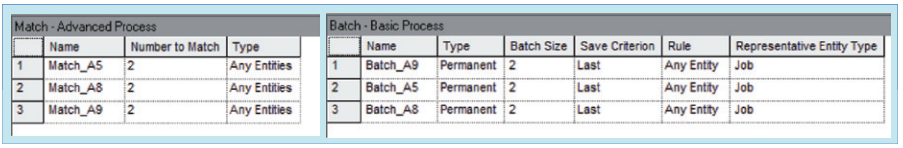


Fig. 7.39. ACD-Arena mapping diagram for a project management system.

queues {Q2–Q12}, two are resource queues {R1–R2}, and one is a source queue {Job}. The Arena flowchart model consists of six Process modules {Process_A1, A2, A3, A7, A8, A9}, three Delay modules {A4, A5, and A6}, three Separate modules {A1, A3, A4}, three Match modules {A5, A8, A9}, three Batch modules {A5, A8, A9}, a Create module, and a Dispose module.

An activity that requires a positive time delay but not executed by a resource is called a ***delay activity.*** The ACD model in Fig. 7.39 has six *timed activities* {A1–A3, A7–A9} and three delay activities {A4–A6}. The *source queue* Job in the ACD model maps to the Create module Arrive in which a single entity Job is created at time zero. A *timed activity* maps to a *Process* module, whereas a *delay activity* maps to a *Delay* module. The ACD model has three *activities with a split point* {A1, A3, A4} and three *activities with a merge point* {A5, A8, A9}, and the mapping relationships for the split/merge points are as follows:



Match - Advanced Process

	Name	Number to Match	Type
1	Match_A5	2	Any Entities
2	Match_A8	2	Any Entities
3	Match_A9	2	Any Entities

(a)

Batch - Basic Process

	Name	Type	Batch Size	Save Criterion	Rule	Representative Entity Type
1	Batch_A9	Permanent	2	Last	Any Entity	Job
2	Batch_A5	Permanent	2	Last	Any Entity	Job
3	Batch_A8	Permanent	2	Last	Any Entity	Job

(b)

Fig. 7.40. Spreadsheets for defining the (a) Match modules and (b) Batch modules.

TABLE 7.4. Station Numbers and Processing Times for Jobs Processed in a Job Shop

	Processing Step-1 (p = 1)		Processing Step-2 (p = 2)		Processing Step-3 (p = 3)		Processing Step-4 (p = 4)		Processing Step-5 (p = 5)	
Job (Ratio)	sn(j,1)	pt(j,1)	sn(j,2)	pt(j,2)	sn(j,3)	pt(j,3)	sn(j,4)	pt(j,4)	sn(j,5)	pt(j,5)
j = 1 (26%)	1	Exp(6)	2	Exp(5)	3	Exp(15)	4	Exp(8)	—	—
j = 2 (48%)	1	Exp(11)	2	Exp(4)	4	Exp(15)	2	Exp(6)	3	Exp(27)
j = 3 (26%)	2	Exp(7)	1	Exp(7)	3	Exp(18)		—		—

- A *split point* maps to a Separate module in the Arena flowchart model:
  - The timed activity A1 → Process_A1 – Separate_A1 modules
  - The delay activity A4 → Delay_A4 – Separate_A4 modules
- A *merge point* maps to a Match-Batch module pair in the flowchart model:
  - The delay activity A5 → Match_A5 – Batch_A5 – Delay_A5 modules
  - The timed activity A8 → Match_A8 – Batch_A8 – Process_A8 modules

Figure 7.40 shows spreadsheets for inputting the attribute values of the Match modules and Batch modules.

7.4.5 ACD-Based Arena Modeling of a Job Shop

As discussed in the previous chapter (see Section 6.5.4), a simple job shop is characterized by a number of stations {s} with each station having one or more identical machines. There are multiple job types {j} to be processed in a job shop, and each job type has a unique routing sequence. Table 7.4 shows a typical example of station numbers sn(j,p) and processing times pt(j,p) for three types of jobs processed in the job shop. For example, Type-2 jobs go through Station-1, Station-2, Station-4, Station-2, Station-3, and Exit (=Station-0). The product mixes are (1) 26% of the jobs are of Type-1, (2) 48% are Type-2 jobs, and (3) 26% Type-3 jobs.

7.4.5.1 Building the Arena Flowchart Model of the Job Shop Figure 7.41 shows an ACD-Arena mapping diagram for the simple job shop. The parame-

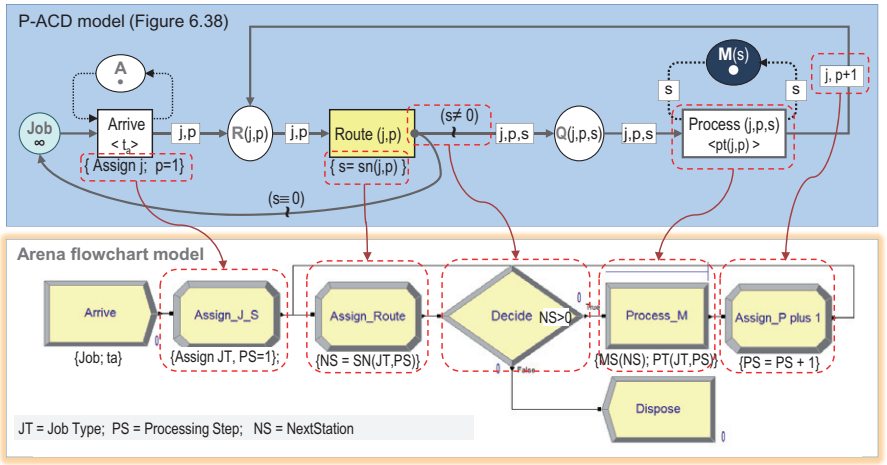


Fig. 7.41. ACD-Arena mapping diagram for the simple job shop.

Resource - Basic Process

	Name	Type	Capacity
1	M1	Fixed Capacity	1
2	M2	Fixed Capacity	1
3	M3	Fixed Capacity	1
4	M4	Fixed Capacity	1

Double-click here to add a new row.

Entity - Basic Process

	Entity Type	Initial Picture	Holding Cost
1	Job	Picture.Report	0.0
2	J1	Picture.Report	0.0
3	J2	Picture.Report	0.0
4	J3	Picture.Report	0.0

Double-click here to add a new row.

Attribute - Basic Process

	Name	Rows	Columns
1	JT		
2	NextStation		
3	PS		

Double-click here to add a new row

Fig. 7.42. Spreadsheets specifying the Resource, Entity, and Attribute data modules.

terized ACD (P-ACD) model consists of three activity nodes and five queue nodes. The three activity nodes are a create activity Arrive, an instant activity Route (j,p), and a timed activity Process (j,p,s). Among the five queue nodes are a resource queue M and two buffer queues R(j,p) and Q(j,p,s). The Arena flowchart model has one Process module Process_M, three Assign modules, and one Decide module. The ACD-to-Arena mapping relationships are: (1) Each assignment (or state update) operation in the ACD model maps to an Assign module in the Arena model, (2) the complete conditional branching point maps to the Decide module, and (3) the timed activity maps to the Process module.

7.4.5.2 Specifying the Arena Static Model of the Job Shop Basic components of Arena static model are resources, entities, and attributes of an entity. Figure 7.42 shows the spreadsheets for the Resource data module, Entity data module, and Attribute data module for the simple job shop example.

For this model, the resources {M1–M4} and the entity types {J1–J3} are defined as sets in the Set data module shown in Fig. 7.43. The resource set and the entity set are named as MS and ET, respectively, such that MS(1) = M1, MS(2) = M2 . . . ET(3) = J3.

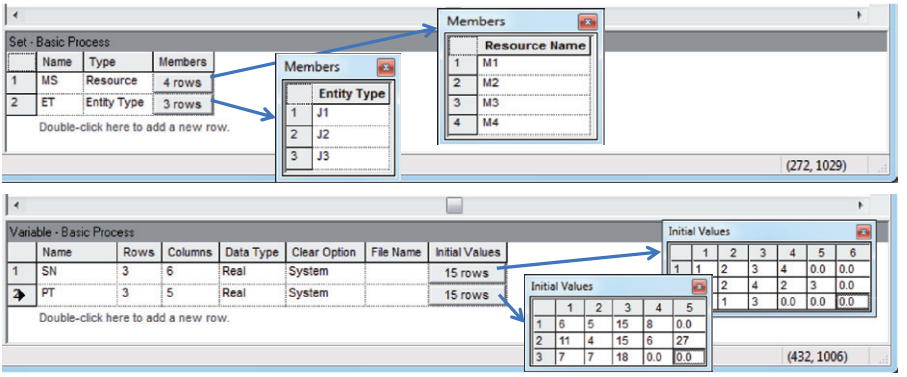


Fig. 7.43. Spreadsheets specifying the Set and Variable data modules .

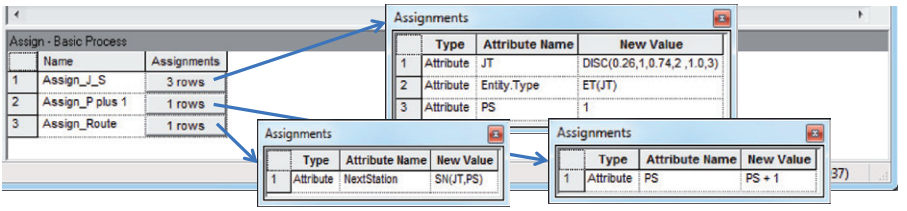


Fig. 7.44. Spreadsheet for inputting the attribute values of the Assign modules.

The station numbers and processing times for each job type, listed in Table 7.4, are declared in the Variable data module of the Arena static model. Figure 7.43 (bottom) shows the spreadsheets for the array variables SN (station number) and PT (processing time).

7.4.5.3 Inputting the Attribute Values of Flowchart Modules There are five types of modules—Create, Assign, Decide, Process, and Dispose—in the Arena flowchart model in Fig. 7.41. Attribute values for Create and Dispose modules were discussed in Section 7.2.2 (see Sections 7.2.2.1 and 7.2.2.4). In this section, we will describe the attribute values for the Assign, Decide, and Process modules.

Figure 7.44 shows a spreadsheet for specifying the attribute values for the three Assign modules in the Arena flowchart model of the simple job shop. In the Assign_J_S module, it is specified that JT = DISC (0.26, 1, 0.74, 2, 1.0, 3) and Entity.Type = ET (JT), which will return the value of Entity.Type as J1, J2, or J3 with probabilities 26%, 48%, and 26%, respectively. Assignments for finding the next processing step is done at the Assign_P plus 1 as PS = PS + 1. The next station number is assigned at the Assign_Route module as NextStation = SN (JT, PS).

Figure 7.45 shows spreadsheets defining the Decide module and the Process module. Note that these modules use the global variables, JT, NextStation, and

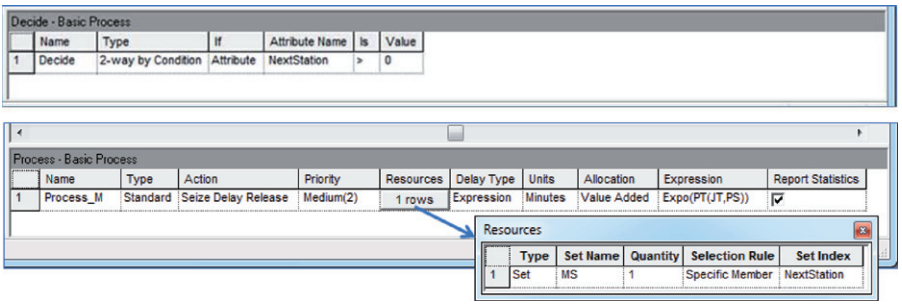


Fig. 7.45. Spreadsheet for the Decide module and the Process module.

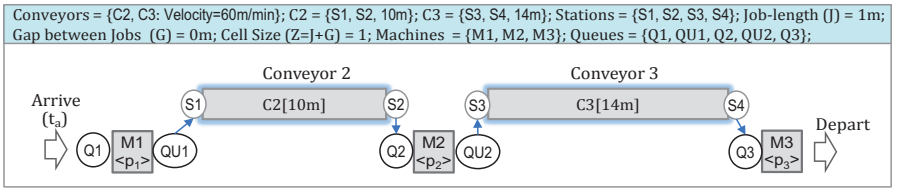


Fig. 7.46. Schematic layout of a conveyor driven serial line.

PS. Recall that the values of the global variables, declared in the Attribute data module (see Fig. 7.42), are updated at the Assign modules (see Fig. 7.44). As depicted in Fig. 7.45, (1) the branching condition of the Decided module is specified as NextStation > 0; and (2) the resource ID of the Process module is obtained as MS(NextStation) and the processing time is determined from the expression Expo(PT(JT,PS)).

7.4.6 ACD-Based Arena Modeling of a Conveyor-Driven Serial Line

Figure 7.46 shows a schematic layout of the conveyor-driven serial line described in the Chapter 6, Section 6.4.7. There are three machines {M1–M3} and two conveyor segments {C2–C3} in the line. In Arena, the start and end points of a conveyor segment are designated by a pair of stations. Jobs arrive with an inter-arrival time of t_a and move along the line in the following sequence: Q1→ M1 → QU1 → C2 → Q2→ M2 → QU2 → C3 → Q3 → M3.

In Arena, a conveyor segment is specified in terms of its physical attributes such as length, velocity, and cell size, whereas in an ACD, a conveyor is specified in terms of its logical attributes. A *cell size* is defined as the sum of the job length and the gap between jobs on the conveyor. For example, the physical attributes of the conveyor segment C2 in Fig. 7.46 are defined as Velocity = 60 m/min, Length = 10 m, and Cell size = 1 m. The logical attributes of a conveyor model in ACD are Conveying-time (t) and Capacity (C). Figure 7.47 shows the conversion relationships between the physical attributes and logical attributes of a conveyor segment.

Physical Attributes of Conveyors (Arena)

Resource Name	Velocity (V)	Length (L)	Cell Size (Z)
Conveyor-2	60 m/min	10 m	1 m
Conveyor-3	60 m/min	14 m	1 m

Logical Attributes of Conveyors (ACD)

Resource Name	Conveying Time (t = L / V)	Capacity (C = L / Z)
Conveyor-2	**t2** = 0.17 min	**C2** = 10
Conveyor-3	**t3**= 0.23 min	**C3** = 14

Fig. 7.47. Relationships between the physical and logical attributes of a conveyor.

7.4.6.1 Building the Arena Flowchart Model of the Conveyor-Driven Serial Line Figure 7.48 shows an ACD-Arena mapping diagram for the conveyor-driven serial line depicted in Fig. 7.46. The ACD model has three timed activities {P1–P3}, two transport activities {T2–T3}, six instant activities, three resource queues {M1–M3}, two conveyor queues {C2–C3}, and five buffer queues {Q1–Q3, QU1–QU2}. The six instant activities are a load activity {L1} and an unload activity {U3}, two unload/access activities {U1/A2, U2/A3}, and two exit/load activities {E2/L2, E3/L3}. The Arena flowchart model contains three Seize modules {M1–M3}, three Delay modules {M1–M3}, three Release modules {M1–M3}, four Station modules (Station 1-4), two Access modules {C2–C3}, two Convey modules {C2–C3}, and two Exit modules {C2–C3}. Brief descriptions of the newly introduced Arena modules are given below:

- The Station module defines a station for a location where processing occurs.
- The Access module allocates a conveyor cell(s) to an entity for movement.
- The Convey module moves an entity on a conveyor from a station to another.
- The Exit module releases the cells on the conveyor that have been allocated to an entity.

Depicted in Fig. 7.48 are six cases of mapping relationships from the ACD-to-Arena flowchart. Among the six cases, four are simple mapping cases:

1. Load activity L1 maps to the Seize module Seize_M1.
2. Unload activity U3 maps to the Release module Release_M3.
3. Each of the timed activities P1–P3 maps to a Delay module.
4. Each of the transport activities T2–T3 maps to a Convey module.

And, there are two cases of composite mapping:

- An unload/access activity maps to a Station-Access-Release module triplet:
  - The U1/A2 Activity → Station1 – Access_C2 – Release_M1 modules

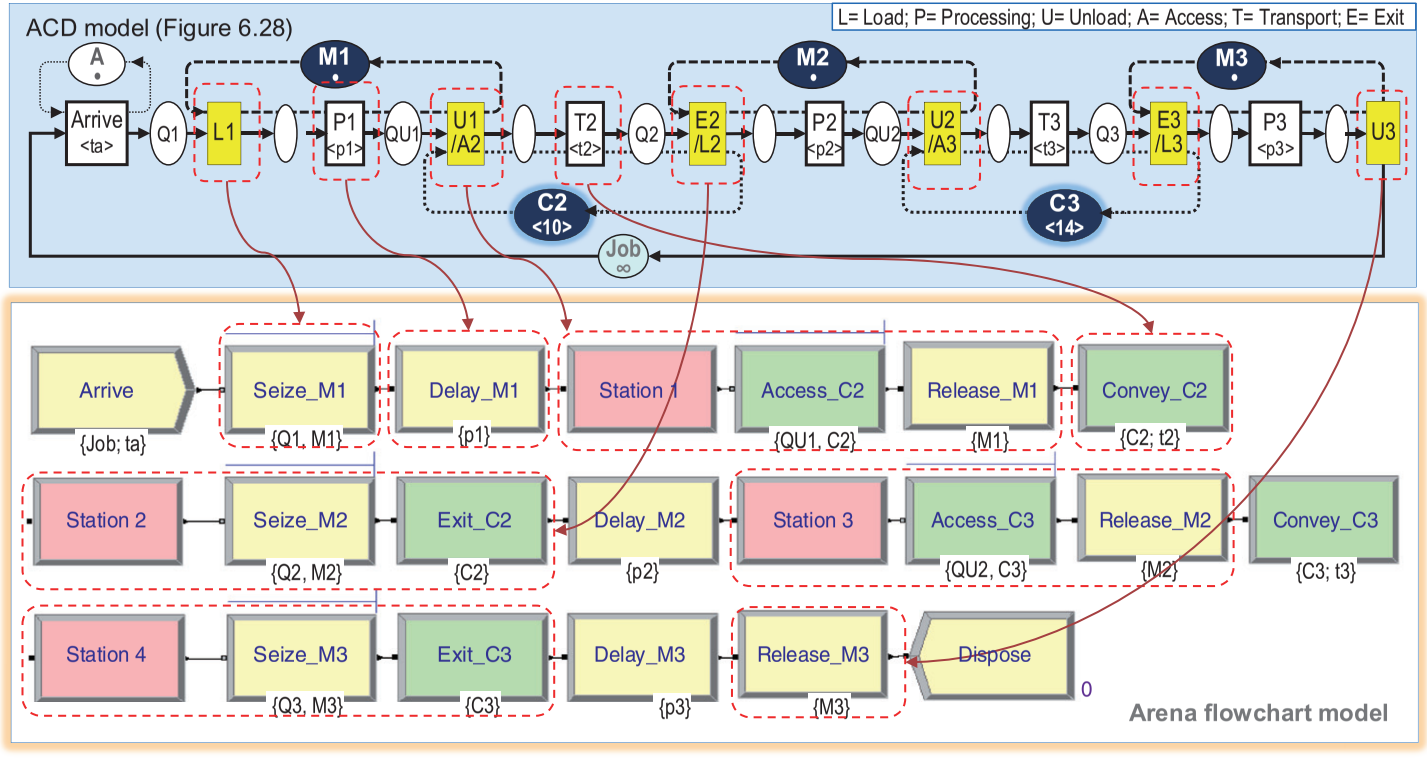


Fig. 7.48. ACD-Arena mapping diagram for the conveyor-driven serial line of Fig. 7.46.

- An exit/load activity maps to a Station-Seize-Exit module triplet:
  - The E2/L2 Activity → Station2 – Seize_M2 – Exit_C2 modules

7.4.6.2 Specifying the Arena Static Model of the Conveyor-Driven Serial Line Figure 7.49 shows spreadsheets specifying the contents of the Arena static model of the conveyor-driven serial line. The resource queues {M1, M2, M3} in the ACD model are specified in the Resource data module and buffer queues are specified in the Queue data module. The conveyor queues {C2, C3} are specified in the Conveyor data modules as well as in the Segment data modules where the physical attributes (see Fig. 7.47) of each conveyor are provided.

7.4.6.3 Inputting the Attribute Values of Flowchart Modules This section discusses attribute values for the newly introduced conveyor-related modules {Station, Access, Convey, Exit}. Figure 7.50 shows spreadsheets for specifying

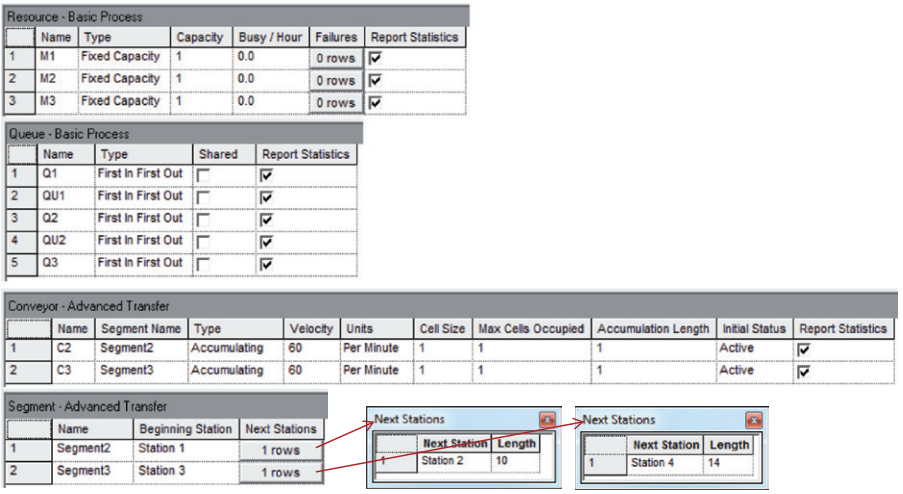


Fig. 7.49. Spreadsheets specifying the Arena static model of the conveyor-driven serial line.

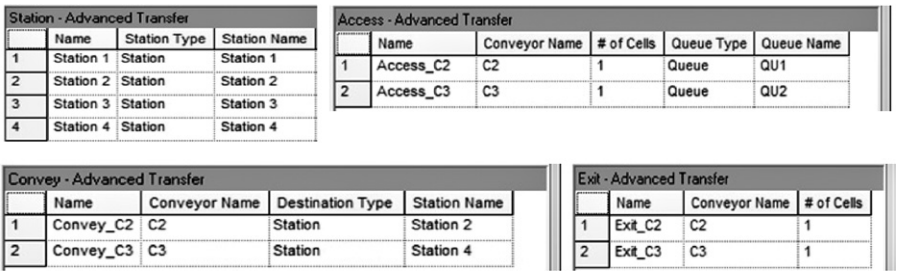


Fig. 7.50. Spreadsheets for inputting attribute values of conveyor-related modules.

the attribute values of the conveyor-related modules: (1) Station Name of each Station module is provided (top left); (2) Conveyor Name, # of Cells per access, and Queue Name of each Access module are provided (top right); (3) Conveyor Name and the destination Station Name of each Convey module are provided (bottom left); (4) Conveyor Name and # of Cells per exit of each Exit module are provided (bottom right).

7.5 REVIEW QUESTIONS

7.1. How is a process defined in discrete-event system modeling?

7.2. What does an entity flow diagram describe in discrete-event system modeling?

7.3. What are the names of the four Arena-specific menus?

7.4. What is the difference between a timed activity and a delay activity?

7.5. What is a resource queue in an ACD?

7.6. How is the resource schedule defined in Arena?

7.7. What is an instant activity in ACD?

7.8. What is the meaning of the Boolean expression NR(K1) == 3?

7.9. How is a nonhomogeneous arrival process modeled (or specified) in Arena?

7.10. What is the incomplete conditional branching?

7.11. How is the cell size of a conveyor defined in Arena?

CHAPTER 8

Output Analysis and Optimization

When you can measure what you are speaking about and express it in numbers, you know something about it. But when you cannot measure it, when you cannot express it in numbers, your knowledge is of the meager and unsatisfactory kind; you have scarcely in your thoughts advanced to the state of science whatever the matter may be.

—Lord Kelvin

8.1 INTRODUCTION

In a real-life simulation project, it is exciting to watch as your simulation program begins to generate some outputs. However, you may easily become overwhelmed by the large amount of data produced by the simulation program. The goal of output analysis and optimization is to draw conclusions and make decisions correctly and efficiently from the simulation outputs.

The purpose of this chapter is to provide basic coverage of simulation output analysis and optimization. The topics that are covered in this chapter are the framework of simulation output analyses, qualitative output analyses, statistical output analyses, linear regression analyses, and response surface methodology for simulation optimization. Also, after studying this chapter, you should be able to answer the following questions:

1. What is a simulator calibration? How does it differ from simulator verification?
2. What is a simulation sensitivity analysis? How does it differ from a simulation optimization?
3. Why is simulation optimization different to analytic optimization?
4. What are the commonly used output plots?

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.

5. What are R^2 statistics?
6. How are the design points spaced in a two-variable central composite design?

The remainder of the chapter is organized as follows. The overall framework of the simulation output analysis is given in Section 8.2, and guidelines for qualitative output analyses are presented in Section 8.3. Statistical output analysis methods for terminating simulations, nonterminating simulations, and comparing alternative systems are presented in Section 8.4, and linear regression modeling for output analyses is presented in Section 8.5. The last section is devoted to the response surface methodology for simulation optimization. Student's *t*-distribution and *t*-tests are reviewed in the Appendix.

8.2 FRAMEWORK OF SIMULATION OUTPUT ANALYSES

In order to draw conclusions and make decisions correctly and efficiently from the simulation outputs, we need to (1) verify the correctness of the simulation program and calibrate the simulation outputs against the actual data collected, (2) design simulation experiments carefully, (3) perform statistical analyses on the output values in order to draw conclusions with confidence, and (4) employ optimization techniques to find the optimal solutions. In addition, the simulation team should be able to sell the simulation results to the customers and users of the simulation.

Shown in Fig. 8.1 is the scope of simulation output analyses for discrete-event simulations. As depicted in the figure, the key activities related to the output analyses are verification and calibration, experimentation, drawing conclusions with confidence, and communication and presentation.

8.2.1 Verification and Calibration

Verification is defined as ensuring that the simulation program is correct with respect to the formal model, while *calibration*, which is often referred to as

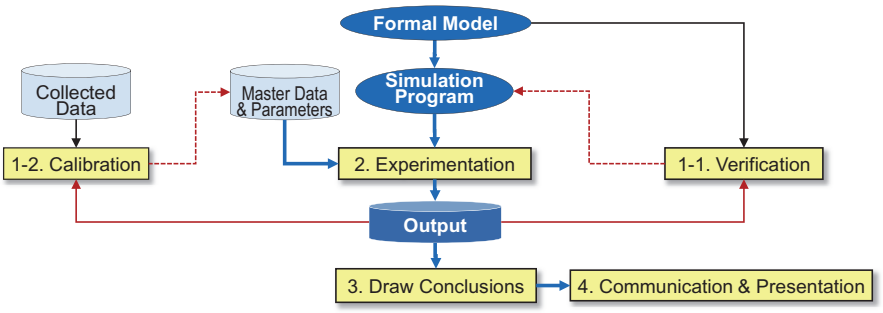


Fig. 8.1. Framework of a simulation output analysis in discrete-event simulation.

the *operational validation* [Sargent 2004], is defined as rendering the simulator's output behavior sufficiently accurate over the domain of its intended applicability by adjusting the values of the master data and the parameters of the target system model. Verification and calibration are undertaken during the initial development and testing; careful experimentation is required because a simulator may be valid for one set of experimental conditions and invalid in another.

In a real-life project, verification can be undertaken internally within the simulation team, whereas calibration must be performed jointly with the project stakeholders or users. Calibration is undertaken by defining and adjusting the handles in order to reduce the gap between the simulator output and the source system output. Sargent [2004] recommends the following calibration steps:

1. At the beginning, an agreement is made between the simulation team and the sponsor (or user) that specifies the basic calibration approach as well as the techniques to be used.
2. The amount of accuracy required is specified for the simulator's output variables of interest for their intended applications.
3. In each of the iterations, comparisons are made between the simulation output and the data collected from the source system.
4. Calibration documentation is developed for inclusion in the overall project documentation.

8.2.2 Simulation Experimentation

The main purpose of the experimentation in Fig. 8.1 is to optimize the simulation. This may have various forms, such as selecting the best alternative or finding the optimal parameter values. According to Schruben and Schruben [2001], the rules or factors that govern the interaction of entities in a system that can be controlled are called *parameters*, while those that cannot be controlled are called *laws*. A simulation experiment that determines the optimal values of the parameters is called a *simulation optimization* and that which determines the effects of the changes in the laws is called a *sensitivity analysis*. The controllable parameters are referred to as *handles* or *decision variables*.

A framework for the simulation optimization is called an *experimental frame*. As shown in Fig. 8.2, an experimental frame consists of a transducer that analyzes the output, an acceptor that evaluates the performance measures, and a generator that adjusts the handles. Optimization is usually undertaken iteratively: (1) perform a set of simulation runs, (2) analyze outputs, (3) evaluate key performances, (4) adjust the handles, and (5) perform another set of simulation runs. However, in some cases, sufficient numbers of simulation runs may be performed according to a predesigned plan and then optimization is undertaken. The handles can be quantitative variables such as the number of resources or qualitative variables such as dispatching rules.

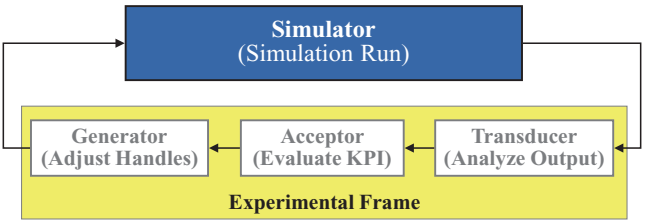


Fig. 8.2. Experimental frame for simulation optimization.

Simulation optimization is quite different from analytic optimization because (1) an analytic expression of the objective function does not exist, (2) the objective function is a stochastic function of deterministic decision variables, (3) the simulation executions are much more expensive than evaluating the analytic functions, and (4) interfacing simulators with generic optimization routines are not always a simple task [Azadivar 1999]. The most common form for a simulation optimization is to minimize an expected value as given by the following equation [Fu et al. 2005]:

$$\min_{\theta \in \Theta} E[L(\theta, \omega)],$$

where θ represents the vector of the decision variables, Θ is the constraint set, L is the sample performance measure, and ω represents a sample path (simulation replication).

In the literature, it has been proposed that the above optimization problem can be solved by employing methods such as the stochastic approximation, sample average approximation, and heuristic search methods [Azadivar 1999, Fu et al. 2005, Kim 2006]. Some simulation optimization software packages that primarily employ heuristic search methods are also available [Fu et al. 2005]. However, the above-mentioned optimization methods may be too theoretical for simulation practitioners.

Thus, a more practical, perhaps more promising, approach is to use process improvement techniques in designed experiments, such as the *response surface methodology* and Taguchi methods [Dellino et al. 2008]. The response surface methodology for simulation optimization will be covered in Section 8.6.

8.2.3 Communication and Presentation

Communication with the stakeholders and presentation to the customers are crucial for a successful modeling and simulation (M&S) project. Remember that "all models are wrong, but some are useful." In practice, it is the author's experience that M&S is a consensus-building process. Building a consensus among the stakeholders and acquiring model credibility via qualification and sensitivity analyses are prerequisites for a successful simulation project.

8.3 QUALITATIVE OUTPUT ANALYSES

Simulation output analyses may be classified into qualitative analyses and statistical analyses. Since discrete-event system simulations include some randomness, rigorous output analyses should be supported with statistical methods. However, in practice, the various output analyses activities in Fig. 8.1 are performed with qualitative simulation outputs.

Simulation outputs may be grouped into graphical and alphanumeric outputs. Graphical outputs are in the form of animations and output plots. An animation may be physical or logical: physical animations animate the physical behavior of the source system at a high fidelity employing computer graphics, whereas logical animations show the behavior of the logical elements in the system, such as entities, events, and state (token) changes. Figure 8.3 shows two examples of physical animation outputs: an automated manufacturing system and an urban traffic intersection. Figure 8.4 presents the logical animation screen of Arena® where it is shown that six jobs are waiting in the buffer (waiting to use the resource), one job is being processed, one job is about to be disposed, and 112 jobs have been disposed out of the 121 jobs created.

Commonly used output plots include scatter plots, line plots, and histograms. Figure 8.5 shows a scatter plot of the waiting time of each entity and the queue size when each entity departs (unloaded from the single server system shown in Fig. 4.1 in Chapter 4), and a behavior graph that compares the simulation model and the real system from Sargent [2004]. Line plots are used for time-dependent statistics such as queue sizes and work-in-process (WIP) levels over time, while histograms are used for sample statistics such as

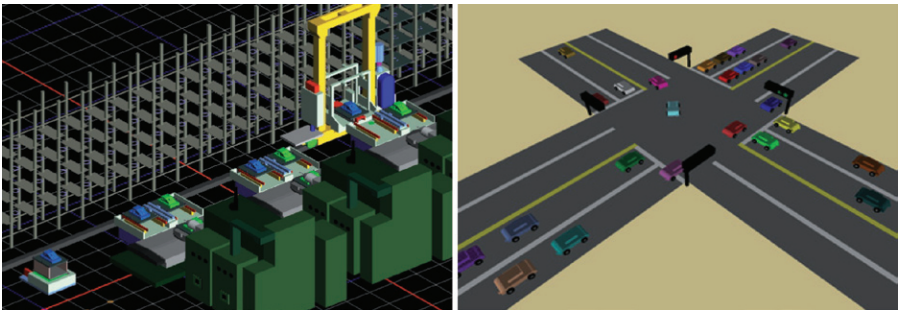


Fig. 8.3. Examples of physical animation outputs.

Fig. 8.4. Logical animation screen of Arena®.

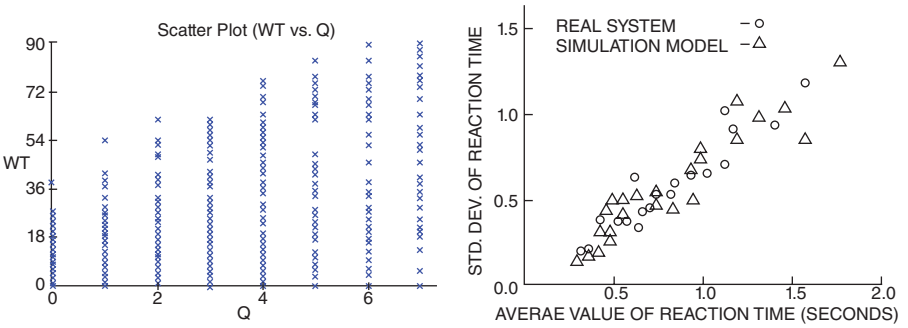


Fig. 8.5. Scatter plot of waiting time vs queue size, and a behavior graph.

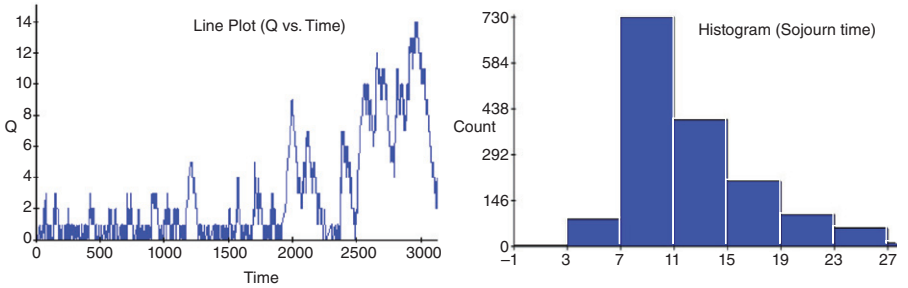


Fig. 8.6. Line plot of queue size and a histogram of sojourn times.

TABLE 8.1. Event Traces from a SIGMA® Simulation of a Single Server System with Failure

Time	Event	Count	Q	M	ST	TAV{Q}	AVE{ST}
0	Run	1	0	1	0	0	0
0	Arrive	1	1	1	0	0	0
0	Load	1	0	0	0	0	0
1.633	Arrive	2	1	0	0	0	0
7.895	Unload	1	1	1	7.895	0	0
7.895	Load	2	0	0	7.895	0.793	1.579
13.918	Unload	2	0	1	12.918	0.793	2.631
31.582	Arrive	3	1	1	12.918	0.449	4.101

sojourn times. Figure 8.6 presents a line plot of the queue size and a histogram of the sojourn times in the single server system given in Fig. 4.1.

Common types of alphanumeric outputs are traces and output statistics. Traces of events and/or state changes are essential for simulator verification. Output statistics are also useful in verification, but they are primarily used for simulator calibration. Table 8.1 shows the initial part of an event trace of a single server system simulation using SIGMA®.

In summary, Schruben and Schruben [2001] provide the following additional tips regarding the use of simulation outputs:

1. During the initial development/testing, logical animation is the most valuable simulation output.
2. For locating gross logic errors, physical animations are the most useful simulation outputs.
3. When evaluating alternative system designs at a high level, output plots are the most useful output.
4. Output statistics are the most appropriate for sensitivity analyses and optimizations.
5. In analyzing the overall performance/dynamics of a system, output plots are the most useful outputs.
6. In selling the simulation to prospective users, physical animations are the most useful simulation outputs.

8.4 STATISTICAL OUTPUT ANALYSES

Many simulation studies are concerned with estimating the performance measures of the source system. Because discrete-event system simulations include some randomness, simulation output data is effectively a random variable. Thus, in principle, the data analysis methods found in statistics books may be used in the simulation data analyses. However, in general, the simulation data are not independent, and extra efforts may be required in order to accommodate the dependency in the simulation output data.

Simulations may be terminating or nonterminating, depending on whether there is an obvious method for determining a simulation run. A *terminating simulation* is one for which there is a "natural" event that specifies the length of the simulation time of interest for the source system [Nakayama 2002]. Otherwise, it is a nonterminating simulation, which is often referred to as a *steady state simulation*.

8.4.1 Statistical Output Analyses for Terminating Simulations

The simulation of most service systems is a terminating simulation, because they have an obvious terminating event (i.e., closing time). Suppose X represents a performance measure of the system and suppose that we are interested in computing the mean (μ) and variance (σ^2) of X, as defined below:

$$\mu = E[X]; \sigma^2 = E[(X-\mu)^2] \equiv Var(X). \tag{8.1}$$

Let X_j be output data obtained from the j^{th} replication for $j = 1 \sim r$. Then, the point estimates of μ and σ^2 are computed from the sample mean and sample variance, as follows:

$$\hat{\mu} = \bar{X}(r) = \frac{1}{r}\sum X_i \text{ and } \hat{\sigma}^2 = S^2(r) = \frac{1}{r-1}\sum (X_i - \bar{X}(r))^2. \quad (8.2)$$

Further, the *100(1–α)%* confidence interval for the mean μ is given by (see Appendix):

$$\bar{X}(r) \pm t_{r-1,1-\alpha/2}\sqrt{S^2(r)/r}. \quad (8.3)$$

The second term in Eq. 8.3 is called a *confidence interval half-length* (β). For example, suppose that we have obtained a sample mean of 20 and a sample variance of 4 out of 9 replications. Then, a 90% confidence interval half-length is computed as follows ($r = 9$, $\alpha = 0.1$):

$$\beta = t_{8,0.95}\sqrt{4/9} = 1.860 \times 2/3 = 1.24.$$

Furthermore, the 90% confidence interval for the mean (μ) is 20 ± 1.24. How many additional replications are required if we want the value of β to be less than 1.0?

In theory, we can reduce the number of replications required for a given value of β using a random number (U_k) for a particular purpose (e.g., an inter-arrival time) in generating X_j, and using its complement ($1 - U_k$) for the same purpose in generating X_{j+1}. This technique of variance reduction is known as the use of *antithetic variates*.

8.4.2 Statistical Output Analyses for Nonterminating Simulations

A nonterminating simulation is primarily concerned with the steady state performance measures of the system. Let $\boldsymbol{Y} = \{Y_1, Y_2, Y_3 \ldots\}$ be an output sequence for a performance measure, where Y_j is the sojourn time of the j^{th} customer in a nonterminating system. Let's define the distribution of Y_j as follows:

$$F_j(y \mid C) = \text{Prob}(Y_j \le y \mid C) \quad \text{for } j = 1, 2, 3 \ldots,$$

where C denotes the initial conditions of the system at time 0. If the relation given in Eq. 8.4 holds for all y and for any initial condition C, then $F(y)$ is called the *steady state distribution* of the output sequence $\boldsymbol{Y}$ [Nakayama 2002]:

$$F_j(y \mid C) \to F(y) \quad \text{as } j \to \infty. \quad (8.4)$$

The above expression is read as "Y_j converges in distribution to $\boldsymbol{Y}$." Further, the expected value of the $E(Y)$ of Y is called the *steady state performance measure* and the density function ($F_j(y)$) of Y_j is called a *transient density* function. Figure 8.7 depicts the transient behavior of a nonterminating simulation.

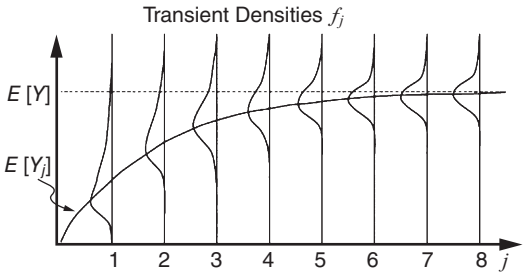


Fig. 8.7. Transient behavior of a nonterminating simulation [Nakayama 2002].

Suppose that we want to estimate the steady state mean, $\mu = E(Y)$. In practice, μ is estimated from the sample mean of the steady state observations $\{Y_j: j = s + 1 \sim n\}$ after deleting the warm-up period data ($Y_j: j = 1 \sim s$), as follows:

$$\hat{\mu} = \bar{Y}(n, s) = \left(\sum_{j=s+1}^{n} Y_j \right) \Big/ (n - s). \tag{8.5}$$

Now, the question is how to determine s (warm-up period length). A popular technique for determining s is a graphical procedure known as *Welch's procedure* [Law 2007, p. 509] in which the moving averages of $\{Y_j\}$ are plotted in order to visually detect the start point (Y_s) of the steady state.

Thus, how do we estimate the variance (σ^2) of Y? The sample variance of $\{Y_j: j = s + 1 \sim n\}$ cannot be used as an estimator of σ^2 because $\{Y_j\}$ is not independent. A practical method is the method of batch means, which is summarized as follows [Nakayama 2002]:

1. Choose a number (m) of batches so that the size (b) of each batch becomes $b = (n - s)/m$. (It has been suggested to choose $10 \le m \le 30$.)
2. Run a simulation to generate $(n - s)$ steady state observations: Y_j for $j = s + 1 \sim n$.
3. Group the $(n - s)$ observations into m batches of size b each, and calculate the k^{th} *batch mean* as follows:

$$\bar{Y}_k(b) = \left(\sum_{j=s+(k-1)b+1}^{s+kb} Y_j \right) \Big/ b. \tag{8.6}$$

4. Calculate the sample variance of the batch means using the results in Eqs. 8.5 and 8.6, as follows:

$$S^2(m, b) = \frac{1}{m-1} \sum_{k=1}^{m} [\bar{Y}_k(b) - \bar{Y}(n, s)]^2. \tag{8.7}$$

5. Finally, the 100(1 – α) % confidence interval for the mean μ is computed as follows:

$$\bar{Y}(n,s) \pm t_{m-1,1-\alpha/2}\sqrt{S^2(m,b)/m}. \tag{8.8}$$

The above batch mean method is based on the observation that when m is large, and Y_j and Y_{j+m} are almost independent. Alternatively, we may use the method of multiple replications in which nonterminating simulations are performed r times. In this case, Eq. 8.3 is used to compute the confidence interval.

8.4.3 Statistical Output Analyses for Comparing Alternative Systems

Let X_j and Y_j denote the output data obtained at the j^{th} replication from System A and System B, respectively. Define $Z_j = X_j - Y_j$ for $j = 1 \sim r$, where r is the number of replications. Then, the point estimates of $\mu = E[Z]$ and $\sigma^2 = Var[Z]$ are computed from the sample mean and sample variance, as follows:

$$\hat{\mu} = \bar{Z}(r) = \frac{1}{r}\sum_{j=1}^{r} Z_j \text{ and } \hat{\sigma}^2 = S^2(r) = \frac{1}{r-1}\sum_{j=1}^{r}(Z_j - \bar{Z}(r))^2. \tag{8.9}$$

Then, the 100(1 – α) % confidence interval for the mean (μ) is obtained using Eq. 8.3.

In a practical simulation study, the confidence interval may be used to decide whether or not to accept a proposed (improved) system that may require addition investments. Let System A be the proposed system and System B be the existing system. Suppose the stakeholders want to know if the proposed system will improve the performance measure by an amount of δ with a confidence level of 90%. Then, you can recommend the proposed system if the following holds:

$$(\bar{Z}(r) - t_{r-1,0.95}\sqrt{S^2(r)/r}) > \delta. \tag{8.10}$$

The above procedure for comparing two systems may be used to compare more than two systems as well via a number of pairwise comparisons. In the literature, a number of ranking and selection methods have been proposed for the comparison of multiple alternatives [Fu et al. 2005], but these theoretical schemes may be too complicated to be useful in real-life industrial simulation projects.

A useful and effective method of increasing the statistical efficiency of comparing alternative systems is to use common random numbers (CRN). The fundamental concept is based on reducing the variance of $Z_j = X_j - Y_j$ by inducing a positive covariance between X_j and Y_j. More precisely, the variance of the sample mean $\bar{Z}(r)$ is expressed as follows:

$$Var[\bar{Z}(r)] = Var(Z_j)/r = \{Var(X_j) + Var(Y_j) - 2Cov(X_j, Y_j)\}/r. \quad (8.11)$$

Thus, if we can induce a positive correlation between X_j and Y_j, the covariance term in Eq. 8.11 will become a positive number, resulting in a decrease in the variance of Z. In order for this to be effective, however, it is essential to synchronize the random numbers across the different systems on a particular replication. Namely, a specific random number used for a specific purpose in one configuration must be used for exactly the same purpose in the other configurations.

8.5 LINEAR REGRESSION MODELING FOR OUTPUT ANALYSES

Simulation (output) analyses are primarily performed in order to obtain behavioral knowledge of a system from its structural knowledge. By combining linear regression modeling with simulation analyses, we can also obtain some generative knowledge of the system. Namely, linear regression modeling enables us to obtain generative knowledge from simulation experiments. In addition, linear regression modeling is a key prerequisite for simulation optimization, as discussed in Section 8.6.

8.5.1 Linear Regression Models

Consider a second-order polynomial model with two variables of x_1 and x_2 as given below:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{11} x_1^2 + \beta_{22} x_2^2 + \beta_{12} x_1 x_2 + \varepsilon,$$

where y is the response variable and ε is an error or residual. Define $x_3 = x_1^2$, $x_4 = x_2^2$, $x_5 = x_1 x_2$, $\beta_3 = \beta_{11}$, $\beta_4 = \beta_{22}$, and $\beta_5 = \beta_{12}$, and then the above equation can be expressed as follows:

$$y = \beta_0 + \sum_{j=1}^{5} \beta_j x_j + \varepsilon,$$

where β_j are parameters and x_j is a regressor variable. In general, a linear regression model with k regressor variables is given by:

$$y = \beta_0 + \sum_{j=1}^{k} \beta_j x_j + \varepsilon. \quad (8.12)$$

Product form models such as $st^a d^\beta f^\gamma = C$ can also be converted into a linear regression model using logarithms and substituting variables.

TABLE 8.2. Data Arrangement for a Linear Regression

Variable	y	x_1	x_2	⋯	x_k
Data	y_1	x_{11}	x_{12}	⋯	x_{1k}
	.	.	.		.
	.	.	.		.
	y_n	x_{n1}	x_{n2}	⋯	x_{nk}

8.5.2 Regression Parameter Estimation

Suppose we have data obtained from n simulation experiments as shown in Table 8.2, where y_i is the response (e.g., performance measure) obtained from experiment i for $i = 1$ to n, and x_{ij} is the value (or level) of the regressor variable x_j for $j = 1$ to k used at experiment i. Then, the model equation (Eq. 8.12) may be written in terms of the data in Table 8.2, as follows:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_k x_{ik} + \varepsilon_i \quad for\ i = 1, 2, \ldots n. \tag{8.13}$$

In order to estimate the parameters in the observation equation (Eq. 8.13), we need to have $n > k$.

The observation equation (Eq. 8.13) may be expressed in a matrix form as follows:

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}, \tag{8.14}$$

where

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}; \quad \mathbf{X} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1k} \\ 1 & x_{21} & \cdots & x_{2k} \\ \vdots & \vdots & & \vdots \\ 1 & x_{n1} & \cdots & x_{nk} \end{bmatrix}; \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_k \end{bmatrix}; \quad \text{and } \boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}.$$

Note that $\boldsymbol{X}$ is a $(n \times p)$ matrix, where $p = k + 1$. In order to obtain the least square estimators ($\boldsymbol{b}$) of the parameters, the sum-of-squares of the residuals is defined as follows:

$$\mathrm{L} = \boldsymbol{\varepsilon}'\boldsymbol{\varepsilon} = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})'(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) = \mathbf{y}'\mathbf{y} - 2\boldsymbol{\beta}'\mathbf{X}'\mathbf{y} + \boldsymbol{\beta}'(\mathbf{X}'\mathbf{X})\boldsymbol{\beta}.$$

The least square estimator vector $\boldsymbol{b}$ must satisfy the following:

$$\partial \mathrm{L}/\partial \boldsymbol{\beta}|_{\mathbf{b}} = -2\mathbf{X}'\mathbf{y} + 2(\mathbf{X}'\mathbf{X})\mathbf{b} = \mathbf{0}.$$

Thus, the least square estimator $\boldsymbol{b}$ for $\boldsymbol{\beta}$ is obtained as follows:

$$\boldsymbol{b} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y} = (b_0, b_1, b_2 \ldots b_k). \tag{8.15}$$

Furthermore, the fitted regression model is expressed as:

$$\hat{y} = b_0 + b_1 x_1 + b_2 x_2 + \cdots + b_k x_k,$$

whose matrix form is given by $\hat{\mathbf{y}} = \boldsymbol{Xb}$. The predicted response for the i^{th} observation (i.e., simulation run) is expressed as:

$$\hat{y}_i = b_0 + b_1 x_{i1} + b_2 x_{i2} + \cdots + b_k x_{ik}, i = 1, 2, \ldots, n.$$

Now consider a linear regression model having one regressor variable, $y = a + bx + \varepsilon$, which is a special case of Eq. 8.12 with $k = 1$ ($a = \beta_0$, $b = \beta_1$, and $x = x_1$). The sum-of-squares of the residuals for this special case is expressed as:

$$L = \varepsilon'\varepsilon = \sum_{i=1}^{n} (a + bx_i - y_i)^2.$$

Then, the least square estimators of a and b must satisfy:

$$\partial L / \partial a = 2\sum_{i=1}^{n} (a + bx_i - y_i) = 0; \quad \partial L / \partial b = 2\sum_{i=1}^{n} (a + bx_i - y_i) x_i = 0.$$

Solving the above equations simultaneously yields the following:

$$\hat{a} = \bar{y} - \hat{b}\bar{x}; \quad \hat{b} = \left[\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}\right] \Big/ \left[\sum_{i=1}^{n} x_i^2 - n\bar{x}^2\right], \tag{8.16}$$

where $\bar{x} = \sum x_i / n; \bar{y} = \sum y_i / n.$

8.5.3 Test for Significance of Regression

With the fitted regression model in Eq. 8.16 obtained from the experiment data of Table 8.2, a question arises: "Can we trust the model?" The statistical procedure to answer this question is called an *analysis of variance* (ANOVA). In an ANOVA, the total sum-of-squares (SS_T) and error sum-of-square (SS_E) are computed as follows:

$$SS_T = \sum_{i=1}^{n} (y_i - \bar{y})^2 = \sum_{i=1}^{n} y_i^2 - n(\bar{y})^2 = \mathbf{y}'\mathbf{y} - \left(\sum y_i\right)^2 \Big/ n, \tag{8.17}$$

$$SS_E = \sum_{i=1}^{n} \varepsilon_i^2 = \boldsymbol{\varepsilon}'\boldsymbol{\varepsilon} = (\mathbf{y} - \mathbf{Xb})'(\mathbf{y} - \mathbf{Xb}) = \mathbf{y}'\mathbf{y} - \mathbf{b}'\mathbf{X}'\mathbf{y}, \tag{8.18}$$

where the column vector ***X′y*** is expressed in terms of the data as:

$$\mathbf{X'y} = [c_0, c_1, c_2, \ldots, c_k]^T = \left[\sum_{i=1}^{n} y_i, \sum_{i=1}^{n} x_{i1} y_i, \sum_{i=1}^{n} x_{i2} y_i, \ldots, \sum_{i=1}^{n} x_{ik} y_i\right]^T. \tag{8.19}$$

The quantity ***b′X′y*** is expressed as:

$$\mathbf{b'X'y} = \sum_{j=0}^{k} b_j c_j = b_0 \sum_{i=1}^{n} y_i + \sum_{j=1}^{k} \sum_{i=1}^{n} b_j x_{ij} y_i. \tag{8.20}$$

The regression sum-of-squares (SS_R) is obtained by subtracting SS_E from SS_T, as follows:

$$SS_R = SS_T - SS_E. \tag{8.21}$$

The degree of freedom (*d.f.*) of the total sum-of-squares is $n - 1$ and that of the regression sum-of-square is k. Thus, the degree of freedom of SS_E becomes $n - k - 1$. A mean square is obtained by dividing the sum-of-square by its degree of freedom. Thus, the regression mean square (MS_R) and error mean square (MS_E) are computed as follows:

$$MS_R = SS_R / k; MS_E = SS_E / (n - k - 1). \tag{8.22}$$

The total mean square (MS_T) is computed similarly.

Finally, F_0 is defined as the ratio of MS_R and MS_E. The statistics obtained so far are summarized in an ANOVA as shown in Table 8.3. When ε_i in the model equation (Eq. 8.13) are independent and normally distributed with a mean of 0 and a variance of σ^2, we reject the null hypothesis $\beta_1 = \beta_2 = \cdots = \beta_k = 0$ if $F_0 > F_{a,k,n-k-1}$.

The coefficient of multiple determination, commonly called the R^2 *statistic*, is defined as:

$$R^2 = SS_R / SS_T = 1 - (SS_E / SS_T). \tag{8.23}$$

R^2 is a measure of the amount of response variability explained by the fitted model. However, R^2 always increases as more terms are added to the regression model. Thus, the *adjusted* R^2 defined below may be a better measure:

TABLE 8.3. ANOVA Table

Source of Variation	Sum-of-Square	d. f.	Mean Square	F_0
Regression	SS_R	k	MS_R	MS_R / MS_E
Error	SS_E	$n - k - 1$	MS_E	
Total	SS_T	$n - 1$	MS_T	

$$R^2_{adj} = 1-(MS_E / MS_T) = 1-\left(\frac{n-1}{n-k-1}\right)(SS_E / SS_T) = 1-\left(\frac{n-1}{n-k-1}\right)(1-R^2). \tag{8.24}$$

The adjusted R^2 will often decrease if unnecessary terms are added.

8.5.4 Linear Regression Modeling Example[1]

Data obtained from 14 experiments involving two decision variables (v_1 and v_2) are summarized in Table 8.4. The decision variables are transformed into coded variables (x_1 and x_2) as follows: $x_1 = (v_1 - 225)/30$ and $x_2 = (v_2 - 4.36)$ / 0.36. In general, let μ and ρ, respectively, denote the mean and range of a decision variable (v). Then, its coded variable (x) is expressed as:

$$x = 2(v-\mu)/\rho\,.$$

Now, consider a first-order regression model containing the main effects of the decision variables.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon.$$

TABLE 8.4. Collected Data with Decision Variables and Coded Variables

		Decision Variable		Coded Variable	
Run	y	v_1	v_2	x_1	x_2
1	1004	195	4.00	–1	–1
2	1636	255	4.00	1	–1
3	852	195	4.60	–1	0.6667
4	1506	255	4.60	1	0.6667
5	1272	225	4.20	0	–0.4444
6	1270	225	4.10	0	–0.7222
7	1269	225	4.60	0	0.6667
8	903	195	4.30	–1	–0.1667
9	1555	255	4.30	1	–0.1667
10	1260	225	4.00	0	–1
11	1146	225	4.70	0	0.9444
12	1276	225	4.30	0	–0.1667
13	1225	225	4.72	0	1
14	1321	230	4.30	0.1667	–0.1667

[1] Myers and Montgomery 1995, Chapter 2.

Then, the $\boldsymbol{y}$-vector and $\boldsymbol{X}$-matrix are as follows:

$$\boldsymbol{y} = \begin{pmatrix} 1004 \\ 1636 \\ 852 \\ 1506 \\ 1272 \\ 1270 \\ 1269 \\ 903 \\ 1555 \\ 1260 \\ 1146 \\ 1276 \\ 1225 \\ 1321 \end{pmatrix}; \boldsymbol{X} = \begin{pmatrix} 1 & -1 & -1 \\ 1 & 1 & -1 \\ 1 & -1 & 0.6667 \\ 1 & 1 & 0.6667 \\ 1 & 0 & -0.4444 \\ 1 & 0 & -0.7222 \\ 1 & 0 & 0.667 \\ 1 & -1 & -0.1667 \\ 1 & 1 & -0.1667 \\ 1 & 0 & -1 \\ 1 & 0 & 0.9444 \\ 1 & 0 & -0.1667 \\ 1 & 0 & 1 \\ 1 & 0.1667 & -0.1667 \end{pmatrix}.$$

From the above data, the least square estimator $\boldsymbol{b}$ of $\boldsymbol{\beta}$ is obtained as follows:

$$\boldsymbol{b} = (\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{y} = [1242.3, 323.4, -54.8]'.$$

Thus, the fitted model in the coded variables x_1 and x_2 is:

$$\hat{y} = 1242.3 + 323.4x_1 - 54.8x_2.$$

Then, the fitted model in the natural decision variables v_1 and v_2 becomes:

$$\hat{y} = -520.1 + 10.8v_1 - 152.2v_2.$$

From the expressions given by Eqs. 8.17, 8.18, and 8.21, the sum-of-squares SS_T, SS_E, and SS_R are computed as follows:

$$\begin{aligned} SS_T &= \mathbf{y}'\mathbf{y} - (\textstyle\sum y_i)^2 / n = 22,527,889 - (17,495)^2/14 = 665,387; \\ SS_E &= \mathbf{y}'\mathbf{y} - \mathbf{b}'\mathbf{X}'\mathbf{y} = 22,527,889 - 22,514,468 = 13,421; \\ SS_R &= SS_T - SS_E = 665,387 - 13,421 = 651,996. \end{aligned}$$

Then, the mean squares and F-statistic are obtained as ($n = 14$, $k = 2$):

$$\begin{aligned} MS_E &= SS_E / (n - k - 1) = 13,421/11 = 1,220; \\ MS_R &= SS_R / k = 651,996/2 = 325,983; \\ F_0 &= MS_R / MS_E = 267.2. \end{aligned}$$

TABLE 8.5. ANOVA Table of Linear Regression Modeling Example

Source of Variation	Sum-of-Square	d.f.	Mean Square	F_0	p-value
Regression	651,996	2	325,983	267.2	4.74×10^{-10}
Error	13,421	11	1,220		
Total	665,387	13			

The above statistics are summarized in Table 8.5 (the ANOVA table).

At a significance level of 5% (i.e., $\alpha = 0.05$), the null hypothesis "$\beta_1 = \beta_2 = 0$" is rejected because $F_0 = 267.2 >> F_{0.05,2,11} = 3.98$ (from the F-distribution table), which means that the regression model is significant. Alternatively, the significance test may be performed using the *p-value*, which is the probability of obtaining a test statistic at least as extreme as the one that was actually observed, assuming that the null hypothesis is true. (There are a number of websites providing p-value calculators.) The R^2 and adjusted R^2 are computed as follows:

$$R^2 = SS_R / SS_T = 651,996 / 665,387 = 0.9798;$$

$$R^2_{adj} = 1 - \left(\frac{n-1}{n-k-1}\right)(1-R^2) = 1 - (13/11)(1-0.9798) = 0.9762.$$

8.5.5 Regression Model Fitting for Qualitative Variables

When some regressor variables are qualitative, the different levels (e.g., machine types) of a qualitative variable are represented as indicator variables taking on values of 0 or 1. If the qualitative variable has two levels, an indicator variable, e.g., x_2, is employed as follows:

$x_2 = 0$ if the observation is from level 1;
$x_2 = 1$ if the observation is from level 2.

Consider the case where y is the dependent variable, x_1 is a quantitative variable, x_2 is a qualitative variable taking on values of 0 or 1, and x_1x_2 is the interaction. The model takes the following form:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + \varepsilon.$$

If it has three levels, we use two indicator variables (x_2 and x_3) such that

$x_2 = 0, x_3 = 0$ if the observation is from level 1;
$x_2 = 1, x_3 = 0$ if the observation is from level 2;
$x_2 = 0, x_3 = 1$ if the observation is from level 3.

The remainder of the regression modeling and analysis may be undertaken in the same manner. More details on this subject may be found at this site: http://v8doc.sas.com/sashtml/stat/chap55/sect52.htm, as well as in Myers and Montgomery [1995, pp. 421–444].

8.6 RESPONSE SURFACE METHODOLOGY FOR SIMULATION OPTIMIZATION

As mentioned earlier, the simulation experimentation used to determine the best values of decision variables that provide optimum performance measures is called *simulation optimization*. Even though the cost of computer simulation experiments is much cheaper than that of the designed experiments in real life, the same response surface methodology (RSM) that was developed for real-life experiments can be used in the simulation optimization. Most subjects covered in this section are from the seminal text by Myers and Montgomery [1995].

8.6.1 Overview of RSM for Process Optimization

Response surface methodology (RSM) for process optimization via designed experiments is a sequential process consisting of three phases: a screening phase, a search phase, and an optimization phase. In the screening phase, a set of screening experiments is performed in order to identify the important variables by eliminating the unimportant ones from the initial candidates.

With the important decision variables identified, the search phase is begun in order to determine if the current levels of decision variables would result in a value of response that is near the optimum. If the current levels of decision variables are far from the optimum, a systematic search is undertaken to move the decision variables toward the optimum. In this phase, first-order models are fitted and the method of steepest ascent is employed for the systematic search. When the levels of the decision variables are near the optimum, the optimization phase is executed by fitting second-order models from the data in the near optimum region.

A decision variable used for the process optimization is often referred to as a *control variable*. If there are uncontrollable variables, which are often referred to as *noise variables*, they may also be considered in the RSM. An experimental design to locate the optimal values of the control variables considering the noise variables is called a *robust parameter design*.

8.6.2 Searching for Optimum Regions with the Steepest Ascent

The design of the experiment, regression model building, and sequential experimentation for locating a region of improved response comprise the method of steepest ascent. The search phase of the method of steepest ascent consists of the following six steps:

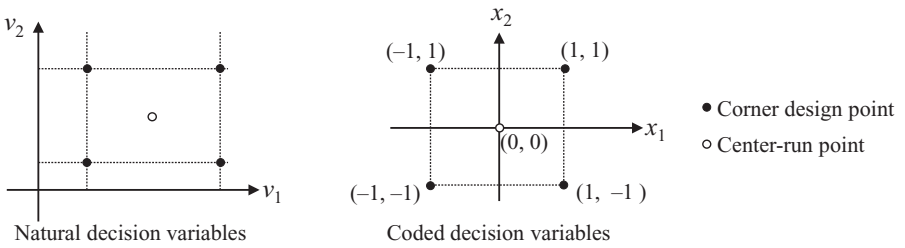


Fig. 8.8. Two-level designs for two-decision variables.

1. Fit a first-order model using a two-level design (with some center runs).
2. Compute a path of steepest ascent if a maximizing response is required. If minimizing, compute the path of steepest descent.
3. Conduct experimental runs along the path.
4. At a location where an approximation of the optimum response is detected, plan the next experiment.
5. Conduct the experiment and fit a first-order model. Then, create a lack-of-fit test.
6. If the lack-of-fit is not significant, compute a second path based on the new model and return to Step 3. Otherwise, terminate the search phase.

8.6.2.1 First-Order Model Fitting Depicted in Fig. 8.8 is a two-level design with center runs. The original values of the decision variables (v_1, v_2) may be transformed into coded variables (x_1, x_2) so that the design levels become +1 and –1 and the center run point is (0, 0). Let m and r, respectively, denote the mean and range of a decision variable (v); then, it's coded variable x is obtained from:

$$x = 2(v - m) / r. \tag{8.25}$$

The data obtained from the two-level design are fitted to a first-order regression model. Let's assume that we obtained the following fitted model:

$$\hat{y} = b_0 + b_1 x_1 + b_2 x_2 = 3 + 4x_1 + 2x_2. \tag{8.26}$$

8.6.2.2 Computation of the Steepest Ascent Path When there are k decision variables, the first-order regression equation will be expressed as:

$$\hat{y} = b_0 + \sum_{j=1}^{k} b_j x_j. \tag{8.27}$$

The movement in x_j along the path of steepest ascent is proportional to the magnitude of b_j with the direction being the sign of b_j. The movement direction

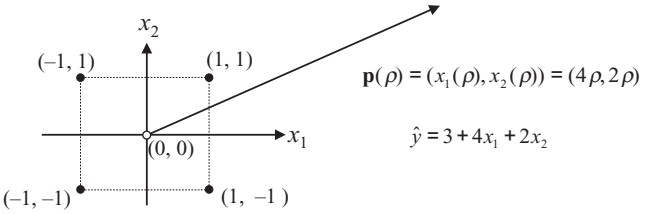


Fig. 8.9. Path of steepest ascent.

of the steepest descent is the opposite of the sign of the coefficient. For the fitted model in Eq. 8.26, the path ***p*** is expressed as a parametric equation of line and is given by:

$$\mathbf{p}(\rho) = (x_1(\rho), x_2(\rho)) = (4\rho, 2\rho).$$

The path of steepest ascent is depicted in Fig. 8.9.

In general, for the regression equation (Eq. 8.27), the coordinates of the decision variables along the steepest ascent path are expressed as follows:

$$\mathbf{p} = (x_1, x_2, \cdots x_k) = (\rho b_1, \rho b_2, \ldots, \rho b_k). \tag{8.28}$$

8.6.2.3 Conduct Experimental Runs along the Path A series of path points along the steepest ascent path can be defined for different values of ρ, and a number of experiments can be performed at each path point. A simple method of defining the path points is to obtain an increment value (Δ), as follows:

$$\Delta = 1 / \max(|b_j|). \tag{8.29}$$

If this rule is applied to the path shown in Fig. 8.9, we obtain $\Delta = 1/4$ and the path points may be defined as:

$$\rho = 2\Delta, 3\Delta, 4\Delta, \cdots = 2/4, 3/4, 4/4, 5/4 \cdots \tag{8.30}$$

For the coded variables, the path points are obtained as follows and as depicted in Fig. 8.10:

$$(x_1, x_2) = (4\rho, 2\rho) = (2, 1), (3, 3/2), (4, 2) \cdots$$

The values of the natural (decision) variables (v_1, v_2) at the path points are obtained from those of the coded variables (x_1, x_2) using the relation in Eq. 8.25. The hypothetical results of the experimental runs are presented in Table 8.6 in the y_p-column, and the response values of the fitted model in Eq. 8.26 are given at the $\hat{y}$-column. Because the deviation becomes quite large at 4Δ,

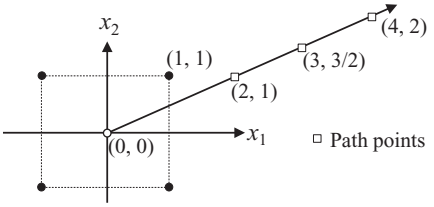


Fig. 8.10. Path points along the steepest ascent path.

TABLE 8.6. Results of the Experimental Runs along the Steepest Ascent Path

		x_1	x_2	$\hat{y}$	y_p	Deviation
0	Base	0	0			
1	Base + 2Δ	2	1	13	12.5	0.5
2	Base + 3Δ	3	1.5	18	16.4	1.6
3	Base + 4Δ	4	2	25	17.2	5.8
4	Base + 5Δ	5	2.5	28	—	

the experimental run is stopped at this point even though the response value is increasing.

8.6.2.4 Plan for the Next Experiment At the last path point where the experimental run was stopped, a two-level design with some center runs is planned such that the last path point becomes the center.

8.6.2.5 Conduct Experiment and Fit a First-Order Model This step is essentially the same as Step 1, but replicated experiments are performed in order to create a lack-of-fit test.

8.6.2.6 Testing for Lack-of-Fit A lack-of-fit test (as described below) is created, and if the lack-of-fit is not significant, a second path based on the new model is computed; the process then returns to Step 3. Otherwise, the search phase is terminated and the optimization phase is started. In the following, a procedure for a lack-of-fit (LOF) test is described briefly.

In general, a LOF test is performed when we want to determine if there is systematic curvature present in a first-order model of the form in Eq. 8.12, which is reproduced below:

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k + \varepsilon.$$

An experiment is planned such that n_i *replicate experiments* are performed at the i^{th} regressor level for $i = 1 \sim m$. Then, for the data from a total of

$n = \sum_{i=1}^{m} n_i$ experiments, the least square estimator is computed using Eq. 8.15 and the predicted response at the i^{th} level is expressed as:

$$\hat{y}_i = b_0 + b_1 x_{i1} + b_2 x_{i2} + \cdots + b_k x_{ik}.$$

Let y_{ij} denote the j^{th} response at the i^{th} regressor level; then, the total sample mean and the i^{th} level sample mean is computed as follows:

$$\bar{\bar{y}} = \frac{1}{n}\sum_{i=1}^{m}\sum_{j=1}^{n_i} y_{ij}; \quad \bar{y}_i = \frac{1}{n_i}\sum_{j=1}^{n_i} y_{ij}.$$

Now, the total sum-of-square and error sum-of-square are computed as follows:

$$\begin{aligned} SS_T &= \sum_{i=1}^{m}\sum_{j=1}^{n_i} (y_{ij} - \bar{\bar{y}})^2; \\ SS_E &= \sum_{i=1}^{n} \varepsilon_i^2 = \sum_{i=1}^{m}\sum_{j=1}^{n_i} (y_{ij} - \hat{y}_i)^2. \end{aligned} \tag{8.31}$$

The above error sum-of-square (SS_E) consists of an SS due to a LOF and an SS due to the pure error (PE), namely, $SS_E = SS_{LOF} + SS_{PE}$. The pure error sum-of-square is computed as follows:

$$SS_{PE} = \sum_{i=1}^{m}\sum_{j=1}^{n_i} (y_{ij} - \bar{y}_i)^2. \tag{8.32}$$

Note that the degrees of freedoms of SS_E and SS_{PE} are $n - k - 1$ and $n - m$, respectively, which indicates that the degree of freedom of SS_{LOF} is $m - k - 1$. Thus, the mean squares of the LOF and PE are computed as:

$$MS_{PE} = SS_{PE}/(n-m); \quad MS_{LOF} = (SS_E - SS_{PE})/(m-k-1). \tag{8.33}$$

Finally, the F-statistic (F_0) is computed from the mean squares and is compared against the F-value. Namely, the hypothesis that the regression model does not contain systematic curvature is rejected if the following condition holds ($p = k + 1$):

$$F_0 = MS_{LOF}/MS_{PE} > F_{\alpha,m-p,n-m}. \tag{8.34}$$

8.6.3 Second-Order Model Fitting for Optimization

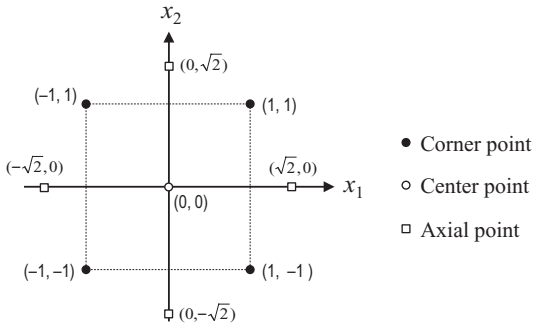
The last phase of RSM for simulation optimization is the optimization phase where a second-order regression model is fitted. Central composite designs are used widely for fitting second-order models. A central composite design

involves the use of a two-level design combined with a set of axial points and a number of center runs. Recall that we already have experiment results for a replicate two-level design with center runs at the end of the searching phase. Thus, it is sufficient to perform additional experiments with the added axial points. The optimization phase consists of the following four steps:

1. Prepare a central composite design.
2. Perform experiments at the axial points and fit second-order models.
3. Perform significance tests and refine the fitted regression model if necessary.
4. Evaluate the fitted model to determine the optimal values of the decision variables.

8.6.3.1 Central Composite Design for Second-Order Model Fitting A central composite design for two decision variables ($k = 2$) is depicted in Fig. 8.11(a), where the corner points and axial points are spaced equally on a circle with a radius of $\sqrt{2}$. As we already have the experiment data for the corner points of the two-level design and the center point (as a result of the search phase), only the axial points need to be determined. In general, we need $2k$ axial points as shown in Fig. 8.11(b), where the value of the axial distance (α) varies from 1.0 to $\sqrt{k}$. The choice of 1.0 places all axial points on the face of the hypercube, while the choice of $\sqrt{k}$ places them on a common sphere. Unless there are constraints imposed on the decision variables, we use $\alpha = \sqrt{k}$.

8.6.3.2 Experiments at Axial Points and Second-Order Model Fitting Recall that a coded variable (x) was obtained from a natural variable (v) using



(a)

x_1	x_2	···	x_k
$-\alpha$	0	···	0
α	0	···	0
0	$-\alpha$	···	0
0	α	···	0
.	.		.
.	.		.
0	0	···	$-\alpha$
0	0	···	α

(b)

Fig. 8.11. (a) Central composite design for $k = 2$ and (b) $2k$ axial points.

the relation given in Eq. 8.25, namely $x = 2(v - m) / r$, where m and r, respectively, denote the mean and range of the natural variable. The value of the natural variable (v) for each coded variable (x) is obtained from:

$$v = m + (r/2)x. \tag{8.35}$$

Now, experiments are performed for the axial points and a second-order model is fitted using all data in the central composite design. A full second-order model has the following form:

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k + \beta_{11} x_1^2 + \cdots + \beta_{kk} x_k^2 + \beta_{12} x_1 x_2 + \cdots + \beta_{k-1,k} x_{k-1} x_k + \varepsilon. \tag{8.36}$$

By applying the linear regression modeling method described in Section 8.5.2, the least square estimators of the regression coefficients are computed in order to obtain the following fitted response surface model:

$$\hat{y} = b_0 + b_1 x_1 + \cdots + b_k x_k + b_{11} x_1^2 + \cdots + b_{kk} x_k^2 + b_{12} x_1 x_2 + \cdots + b_{k-1,k} x_{k-1} x_k. \tag{8.37}$$

8.6.3.3 Significance Test and Model Refinement The significance of the fitted response surface in Eq. 8.37 is tested using the procedure given in Section 8.5.3. If the fitted response surface does not appear highly significant, we may perform tests on the individual regression coefficients or groups of coefficients [Myer and Montgomery 1995, pp. 31–36] in order to refine the fitted model.

8.6.3.4 Evaluation of the Fitted Regression Model The resulting response surface is a quadratic polynomial function that is easily evaluated. The evaluated values may be represented as a three-dimensional surface or a contour plot when the number of decision variables is two.

8.7 REVIEW QUESTIONS

8.1. What is a simulator calibration? How does it differ from a simulator verification?

8.2. What is a sensitivity analysis? How does it differ from a simulation optimization?

8.3. Why is simulation optimization different from analytic optimization?

8.4. What are the commonly used output plots?

8.5. What is the use of common random numbers in output analyses?

8.6. What are R^2 statistics?

8.7. How are the design points spaced in a two-variable central composite design?

APPENDIX 8A: STUDENT'S *t*-DISTRIBUTION

Student's *t*-distribution (published by William S. Gosset in 1908 under the pseudonym "Student") arises in the problem of estimating the mean of a normally distributed population when the sample size is small and the unknown standard deviation must be estimated from the data.

8A.1 Definition

Let Z ~ N (0,1) and $V \sim \chi_k^2$ (chi-square distribution) be independent of each other, then the statistic $T = Z / \sqrt{V / k}$ follows the *t*-distribution with k degrees of freedom.

8A.2 Derivation of the *t*-Statistic

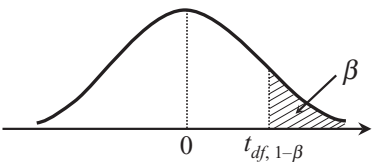
Suppose $\{X_i\}$ are independent random variables that are normally distributed with an expected value of μ. Compute the sample mean and sample variance as follows:

$$\bar{X}(n) = \frac{1}{n}\sum X_i; S^2(n) = \frac{1}{n-1}\sum (X_i - \bar{X}(n))^2.$$

Then, it can be shown that the statistic T follows the *t*-distribution with $n - 1$ degrees of freedom:

$$T = (\bar{X}(n) - \mu) / (S(n) / \sqrt{n}).$$

8A.3 Table of Critical *t*-Values with Degrees of Freedom (df)



df	$\beta = 0.4$	$\beta = 0.25$	$\beta = 0.1$	$\beta = 0.05$	$\beta = 0.025$	$\beta = 0.01$	$\beta = 0.005$	$\beta = 0.0005$
1	0.3249	1.0000	3.0777	6.3138	12.7062	31.8205	63.6567	636.6192
2	0.2887	0.8165	1.8856	2.9200	4.3027	6.9646	9.9248	31.5991
3	0.2767	0.7649	1.6377	2.3534	3.1824	4.5407	5.8409	12.9240
4	0.2707	0.7407	1.5332	2.1318	2.7764	3.7469	4.6041	8.6103
5	0.2672	0.7267	1.4759	2.0150	2.5706	3.3649	4.0321	6.8688
6	0.2648	0.7176	1.4398	1.9432	2.4469	3.1427	3.7074	5.9588
7	0.2632	0.7111	1.4149	1.8946	2.3646	2.9980	3.4995	5.4079
8	0.2619	0.7064	1.3968	1.8595	2.3060	2.8965	3.3554	5.0413
9	0.2610	0.7027	1.3830	1.8331	2.2622	2.8214	3.2498	4.7809
10	0.2602	0.6998	1.3722	1.8125	2.2281	2.7638	3.1693	4.5869
11	0.2596	0.6974	1.3634	1.7959	2.2010	2.7181	3.1058	4.4370
12	0.2590	0.6955	1.3562	1.7823	2.1788	2.6810	3.0545	4.3178
13	0.2586	0.6938	1.3502	1.7709	2.1604	2.6503	3.0123	4.2208
14	0.2582	0.6924	1.3450	1.7613	2.1448	2.6245	2.9768	4.1405
15	0.2579	0.6912	1.3406	1.7531	2.1314	2.6025	2.9467	4.0728
16	0.2576	0.6901	1.3368	1.7459	2.1199	2.5835	2.9208	4.0150
17	0.2573	0.6892	1.3334	1.7396	2.1098	2.5669	2.8982	3.9651
18	0.2571	0.6884	1.3304	1.7341	2.1009	2.5524	2.8784	3.9216
19	0.2569	0.6876	1.3277	1.7291	2.0930	2.5395	2.8609	3.8834
20	0.2567	0.6870	1.3253	1.7247	2.0860	2.5280	2.8453	3.8495
21	0.2566	0.6864	1.3232	1.7207	2.0796	2.5176	2.8314	3.8193
22	0.2564	0.6858	1.3212	1.7171	2.0739	2.5083	2.8188	3.7921
23	0.2563	0.6853	1.3195	1.7139	2.0687	2.4999	2.8073	3.7676
24	0.2562	0.6848	1.3178	1.7109	2.0639	2.4922	2.7969	3.7454
25	0.2561	0.6844	1.3163	1.7081	2.0595	2.4851	2.7874	3.7251
26	0.2560	0.6840	1.3150	1.7056	2.0555	2.4786	2.7787	3.7066
27	0.2559	0.6837	1.3137	1.7033	2.0518	2.4727	2.7707	3.6896
28	0.2558	0.6834	1.3125	1.7011	2.0484	2.4671	2.7633	3.6739
29	0.2557	0.6830	1.3114	1.6991	2.0452	2.4620	2.7564	3.6594
30	0.2556	0.6828	1.3104	1.6973	2.0423	2.4573	2.7500	3.6460
40	0.2550	0.6807	1.3031	1.6839	2.0211	2.4233	2.7045	3.5510
60	0.2545	0.6786	1.2958	1.6706	2.0003	2.3901	2.6603	3.4602
120	0.2539	0.6765	1.2886	1.6577	1.9799	2.3578	2.6174	3.3735
∞	0.2533	0.6745	1.2816	1.6449	1.9600	2.3263	2.5758	3.2905

APPENDIX 8B: STUDENT'S *t*-TESTS

8B.1 One Sample *t*-Test

The *t*-test statistic has the form of $T = Z/s$ with $Z = \sqrt{n}\bar{X}/\sigma$ and $s = S/\sigma$, where σ^2 is the population variance. The sample mean ($\bar{X}$) and sample standard deviation (S) are given by:

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i \quad \text{and} \quad S = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2}. \tag{8B.1}$$

The assumptions underlying a one-sample *t*-test are that Z follows a standard normal distribution and $(n-1)s^2$ follows a χ^2 distribution with $(n-1)$ degrees of freedom under the null hypothesis, and that Z and s are independent.

8B.2 Unpaired Two Sample *t*-Test

The assumptions in the *t*-test that compares the means of two samples are that each of the two populations being compared should follow a normal distribution and that the data used to undertake the test are sampled independently.

8B.2.1 Equal Variance Case Let $\bar{X}_j$ and S_j, respectively, denote the sample mean and sample standard deviation of group (j) with sample size n_j for $j = 1, 2$. Then, when the variances of the two groups are equal, the *t*-statistic (T) to test whether the group means are different is given by (degree of freedom: $d = n_1 + n_2 - 1$):

$$T = \frac{\bar{X}_1 - \bar{X}_2}{S_{12}\sqrt{1/n_1 + 1/n_2}}, \quad \text{where } S_{12} = \sqrt{\frac{(n_1-1)S_1^2 + (n_2-1)S_2^2}{n_1 + n_2 - 2}}. \tag{8B.2}$$

8B.2.2 Unequal Variance Case When the variances of the two groups are not equal, the *t*-statistic (T) and degree of freedom (d) are given by:

$$T = \frac{\bar{X}_1 - \bar{X}_2}{S_{12}}, \quad \text{where } S_{12} = \sqrt{S_1^2/n_1 + S_2^2/n_2}; \tag{8B.3}$$

$$d = \frac{\left(S_1^2/n_1 + S_2^2/n_2\right)^2}{\left(S_1^2/n_1\right)^2/(n_1-1) + \left(S_2^2/n_2\right)^2/(n_2-1)}. \tag{8B.4}$$

8B.2.3 Examples Let A_1 denote a set obtained by taking six random samples from a larger set $(n_1 = 6)$:

$$A_1 = \{30.02, 29.99, 30.11, 29.97, 30.01, 29.99\}.$$

Let A_2 denote a second set obtained similarly $(n_2 = 6)$:

$$A_2 = \{29.89, 29.93, 29.72, 29.98, 30.02, 29.98\}.$$

These could be, for example, the weights of screws that were chosen from a bucket of screws. We will perform tests of the null hypothesis that the means of the populations from which the two samples were taken are equal.

The difference between the two sample means is $\bar{X}_1 - \bar{X}_2 = 0.095$, and the sample standard deviations for the two samples are $S_1 = 0.05$ and $S_2 = 0.11$. For such small samples, a test of equality between the two population

variances would not be very powerful. Because the sample sizes are equal, the two forms of the two sample *t*-tests will perform similarly in this example.

If the approach for unequal variances (discussed above) is followed, the results are:

$$S_{12} = \sqrt{S_1^2/n_1 + S_2^2/n_2} = 0.0485 \quad \text{and} \quad T = \frac{\bar{X}_1 - \bar{X}_2}{S_{12}} = 1.959;$$

$$d = \frac{\left(S_1^2/n_1 + S_2^2/n_2\right)^2}{\left(S_1^2/n_1\right)^2/(n_1 - 1) + \left(S_2^2/n_2\right)^2/(n_2 - 1)} = 7.03.$$

The two-tailed test *p*-value[2] is approximately 0.091 and the one-tailed *p*-value is 0.045. For a case of equal variance, we have $S_{12} = 0.084$, $T = 1.959$, and $d = 10$. Here, the two-tailed *p*-value is approximately 0.078, and the one-tailed *p*-value is approximately 0.039. Thus, if there is good reason to believe that the population variances are equal, the results become somewhat more suggestive of a difference in the mean weights for the two populations.

[2] The *p-value* is the probability of obtaining a test statistic at least as extreme as the one that was actually observed, assuming that the null hypothesis is true.

PART III

ADVANCES IN DISCRETE-EVENT SYSTEM MODELING AND SIMULATION

Part III is design to be used in a graduate-level simulation course in industrial engineering, computer science, and management science. However, researchers and industrial practitioners may find the materials quite useful for embarking upon new modeling and simulation (M&S) researches and implementing new simulation-based solutions. The subject areas covered in Part III are (1) state-based M&S; (2) advanced activity-based M&S; (3) object-oriented event graph modeling for integrated Fab simulation; and (4) parallel simulation. One chapter is devoted to each subject area.

Chapter 9 covers fundamental topics in *state-based modeling and simulation*. After studying this chapter, you should be able to answer the following questions:

1. What are finite state machines?
2. What are timed automata? What is a state graph?
3. How do you build state graph models of various systems?
4. How do you execute state graph models?

Chapter 10 is devoted to *advanced topics in activity-based modeling*. By studying this chapter, you should be able to:

1. Develop your own ACD simulators
2. Make cycle time analyses of various types of work cells
3. Build ACD models of complex systems such as flexible manufacturing systems
4. Convert ACD models to event graph models and state graph models

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.

Chapter 11 is devoted to an *advanced event graph modeling for integrated Fab simulation*. By studying this chapter, you should be able to:

1. Understand electronics fabrication systems
2. Build object-oriented event graph models for integrated simulation of Fab operation
3. Execute object-oriented event graph models
4. Perform automated material handling system (AMHS)-embedded integrated simulation of electronics Fab

Chapter 12 is devoted to parallel simulation and high-level architecture (HLA)/ run-time infrastructure (RTI). By studying this chapter, you should be able to answer the following questions:

1. What is parallel simulation?
2. How do you apply the parallel simulation concept to *workflow simulation*?
3. What is HLA/RTI?
4. How do you perform a parallel simulation with HLA/RTI?

CHAPTER 9

State-Based Modeling and Simulation

Everything has its wonders, even darkness and silence, and I learn, whatever state I may be in, therein to be content.

—Helen Keller

As mentioned in Chapter 1, a comprehensive description of a discrete-event system can be given in terms of the two types of physical modeling components (*Entity* and *Resource*) and three types of logical modeling components (*Activity*, *Event*, and *State*). The existing methods of discrete-event system modeling and simulation (M&S) have been developed around the logical modeling components. Depending on the modeling component to be focused on, we can use (1) event-based M&S methods as covered in Chapters 4 and 5, (2) activity-based M&S methods as covered in Chapter 6, and (3) state-based M&S methods. This chapter is devoted to the state-based M&S of discrete-event systems (DESs).

9.1 INTRODUCTION

A finite state machine (FSM) is the oldest known formal model for modeling the sequential behavior of a DES [Wagner et al. 2006]. The FSM does not consider time, and it is a state-based M&S tool that is widely used in the design analysis of automated systems and embedded software systems. The term *finite state automata* (FSA) is preferred to FSM in computer science, where it is primarily used in language processing and text scanning applications.

In order to describe the dynamic behavior of a DES over time, the FSM (or FSA) formalism has been extended to *timed automata* [Alur and Dill 1994] and DEVS [Zeigler 1976]. The classic DEVS (Discrete-EVent system Specification) is essentially a restricted type of timed automaton, and it has been the *de facto* choice for the state-based M&S of DES. However, the classic DEVS method has some drawbacks: DEVS models are not easy to build, and DEVS

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.

simulators are not efficient to execute. Thus, this book introduces a new state-based M&S formalism, called the *state graph formalism*. A state graph is a special type of timed automata with which some of the difficulties in the classical DEVS are overcome.

The remainder of this chapter is organized as follows. The definitions and models of FSM are presented in Section 9.2, and the definitions and mechanisms of timed automata are given in Section 9.3. The structure and modeling examples of the state graph are given in Sections 9.4 and 9.5, respectively. A comprehensive description of simulation executions of state graph models is provided in the last section of this chapter. A brief description of classic DEVS is given in the Appendix of this chapter.

9.2 FINITE STATE MACHINE

If the present inputs in a system are sufficient to determine its outputs, it is called a *combinatorial system*. If the system needs additional information about the sequence of the previous inputs in order to determine the outputs, the system is a *sequential system*, which requires a mechanism to memorize the previous inputs. The mechanism to store the input history is the *state*, and a finite state machine is a sequential system.

9.2.1 Existing Definitions of Finite State Machines

A finite state machine (FSM) is a formal model for modeling the sequential behavior of a discrete-event system. An FSM is also called a *finite state automaton* (FSA), *finite state transducer*, *state machine*, and more. It is a computation model that consists of a set of states, a start state, an input alphabet, and a transition function that maps the inputs and current states to a next state. The computation begins in the start state with an input string and changes to new states depending on the transition function.

A number of algebraic definitions of FSM exist in the literature. In the classical definition [Peterson 1981], an FSM is defined as a quintuple structure $(S, X, Y, \delta, \lambda)$, where:

1. S is a finite nonempty set of states
2. X is a finite input alphabet
3. Y is a finite output alphabet
4. δ is the next state function (δ: $S \times X \rightarrow S$)
5. λ is the output function (λ: $S \times X \rightarrow Y$)

In computer science, where the term FSA is used instead of FSM, an FSM is defined as a quintuple structure (S, X, δ, s_0, F), where [Hopcroft et al. 2006]:

1. S is a finite set of states
2. X is a finite set of symbols (or events)
3. δ is the transition function
4. s_0 is the start state (or initial state)
5. F is the set of final states (a subset of S)

In this definition, the final states (F) are referred to as *accepting states* because once the FSM moves into a state in F, it accepts every sequence of further inputs.

An FSM is called *deterministic* if the transition function (δ) returns a member of S, and it is called *nondeterministic* if δ returns a subset of S. In engineering, a deterministic FSM that generates outputs is referred to as a *finite state transducer*, which is a sextuple structure (S, X, Y, δ, s_0, λ) where:

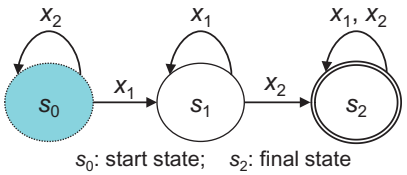
1. S is a finite nonempty set of states
2. X is the input alphabet (a finite nonempty set of symbols)
3. Y is the output alphabet (a finite nonempty set of symbols)
4. δ is the state transition function (δ: $S \times X \rightarrow S$)
5. s_0 is the start state (an element of S)
6. λ is the output function (λ: $S \times X \rightarrow Y$)

In the above definitions, the terms *alphabet* and *symbol* have the same meaning, *event* and *message* are used interchangeably, *input alphabet* and *input event* are used interchangeably, and *output alphabet* and *output message* are used interchangeably. It is worthwhile to compare the last two definitions. In computer science, where FSA is mostly used in language processing and text scanning applications, the concept of final states (F) has a specific function. Thus, F is included in the definition. In contrast, in engineering, where FSMs are commonly used in designing embedded software for reactive systems such as traffic systems and telecommunication systems, the concept of outputs (Y) is critical. An FSM with transition probabilities assigned to the transition function (δ) is referred to as a *probabilistic FSM*.

9.2.2 Finite State Machine Models

Specifying an FSM as an algebraic structure with a detailed description of the transition function (δ) is both tedious and difficult; thus, there are two preferred notations for describing FSM: *state transition diagram* and *state transition table* [Hopcroft et al. 2006]. A state transition diagram of an FSM, which is a graphical model of the FSM structure, is constructed as follows.

a. There is a node for each state in S.
b. Let $\delta(p, x) = q$, where $p, q \in S$ and $x \in X$. Then, there is an arc from node p to node q, which is labeled x. If there are several input symbols



No	State	Input	Output	Next State
1	s_0	x_1	-	s_1
		x_2	-	s_0
2	s_1	x_1	-	s_1
		x_2	-	s_2
3	s_2	x_1, x_2	-	s_2

Fig. 9.1. State transition diagram and state transition table of an FSM.

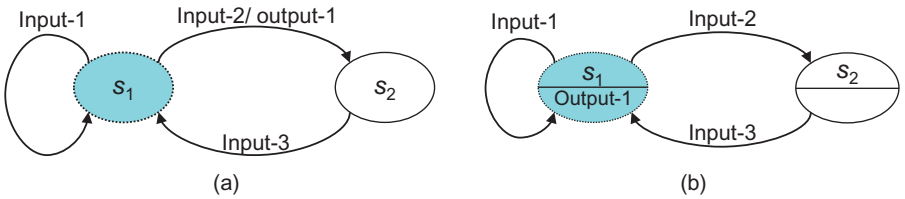


Fig. 9.2. State transition diagrams for the (a) Mealy model and (b) Moore model.

that cause transitions from p to q, the arc is labeled by listing these symbols.

c. The start state node is marked using a dashed circle with shading.
d. Nodes corresponding to the final states F are marked using a double circle.

A *state transition table* is a tabular specification of the transition function. Figure 9.1 presents a state transition diagram (often called an *FSM diagram*) and state transition table. Note in the figure that the start state (s_0) is marked using a dashed circle with shading and the final state (s_2) using a double circle.

In engineering applications, the basic models of FSM are the Mealy and Moore models [Wagner et al. 2006]. A Mealy model is an FSM that produces an output for each transition, which means that the FSM diagram will include both input and output signals for each transition arc. Thus, the finite state transducer (the third definition) in the previous section is a Mealy model. A Moore model is an FSM that produces an output for each state. The selected models will influence the design, but general indications as to which model is better do not exist yet. In practice, mixed models are often used with several output types [Wagner 2005]. Note that there may be no final states in the FSM of a reactive system. Figure 9.2 presents the FSM diagrams for the Mealy and Moore models where the start state is s_1.

9.2.3 Finite State Machine Modeling of Buffer Storage and Single Server Systems

Buffer storage is a passive resource whose state is defined by the number of jobs stored in the buffer. Figure 9.3(a) shows an unlimited capacity buffer in

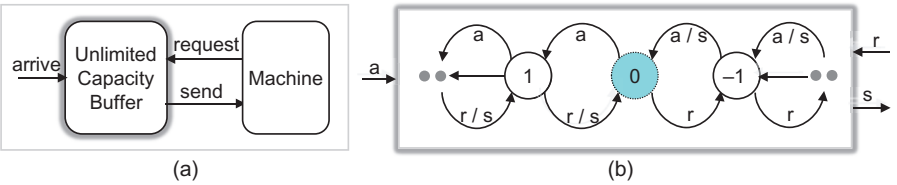


Fig. 9.3. Mealy model of an unlimited capacity buffer.

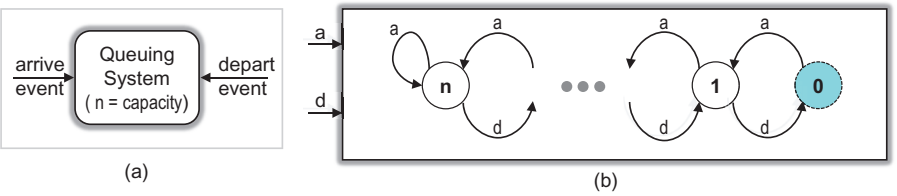


Fig. 9.4. State machine model of a finite capacity queuing system.

a single server system. Let J denote the number of jobs in the buffer, then $J = \{\cdots 2, 1, 0, -1 \cdots\}$ defines the state space of the buffer. A negative J value denotes the number of backlogs. Figure 9.3(b) shows the Mealy model of the Buffer in Fig. 9.3(a) where a (arrive) and r (request) are input events, and s (send) is an output message. However, because the state is not finite, it is an infinite state machine with a countable state set.

Figure 9.4(b) shows the Mealy model of a finite capacity queuing system where the buffer storage and the machine are merged. The queuing system depicted in Fig. 9.4(a) receives arrive event messages {a} and depart event messages {d} from outside. The number of jobs (J) in the queuing system defines the finite state space {n $\cdots$ 2, 1, 0}.

9.2.4 Execution of Finite State Machine Models

The outputs of an FSM are generated by actions. The action types associated with a state are (1) an *entry action* performed when entering the state, (2) an *exit action* performed when exiting the state, (3) an *input action* performed upon receiving an input event, and (4) a *transition action* associated with a transition [Wagner and Wolstenholme 2003].

Figure 9.5 shows the execution flow of an FSM. The FSM waits for an input in the current state; when an input is received, the *input action condition* is tested. If the condition is met, the input action is executed and the *transition condition* is verified. If the transition condition is met, the FSM exits from the current state after executing an exit action, moves to the next state while executing a transition action, and enters into the next state while executing an entry action.

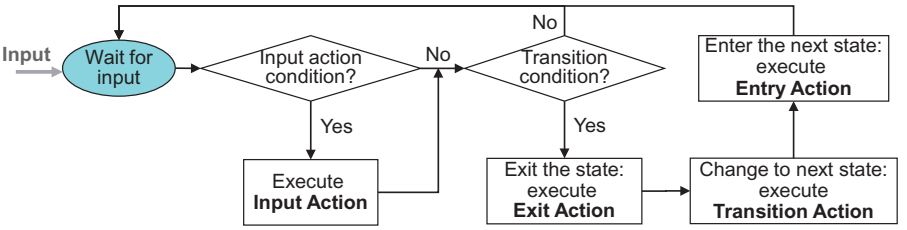


Fig. 9.5. Execution flow of an FSM with actions.

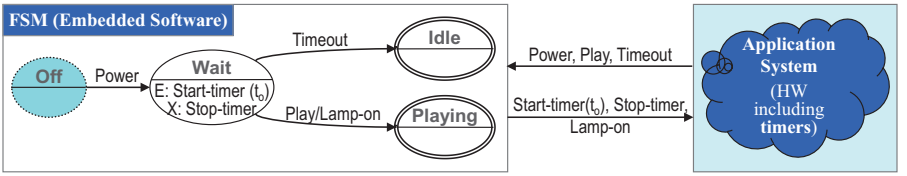


Fig. 9.6. FSM (control system) and its application system.

TABLE 9.1. State Transition Table of the FSM in Figure 9.6

No	State	Entry Action	Exit Action	Input	Transition Action	Next State
1	Off	—	—	Power	—	Wait
2	Wait	Start-timer (t_o)	Stop-timer	Timeout	—	Idle
				Play	Lamp-on	Playing
3	Idle	Stop()	—	—	—	—
4	Playing	Stop()	—	—	—	—

An FSM may be regarded as the *control system* of an application system. Figure 9.6 depicts a DES where the FSM acts as the control system of the application system. The FSM diagram is a mixed model. It is initially in the Off state and it moves to the Wait state upon receiving Power. Immediately after the transition, the entry action (Start-timer) is executed to start the timer. Then, the FSM moves to the Idle state if a Timeout is received or to the Playing state if Play is received before Timeout. In either case, the system executes the exit action Stop-timer before exiting the Wait state. A state transition table for the FSM in Fig. 9.6 is given in Table 9.1.

An FSM executor program may be written easily from the state transition table of the FSM. Let's assume that the communications between the FSM and its application system are made through two I/O functions: Get-Input (INPUT) and Send-Output (OUTPUT). A state transition routine is developed for each state. The execution starts from the start state OFF, and the program control is routed to a next state while executing the exit/entry actions if an input is received. Figure 9.7 shows an FSM executor program written in pseudocode form for the state transition table of Table 9.1.

```
FSM-Executor ( ) {
  State-Off(); } // start state

State-OFF ( ) {
  STATE = OFF;                        // state update;
  Repeat { Get-Input (INPUT); }
      Until (INPUT ≡ "Power");        // wait for input "Power"
   State-WAIT ( ); }                  // move to next state WAIT

State-WAIT ( ) {
  STATE = WAIT;                       // state update
  Send-Output ("Start-timer(t_o)");   // entry action
  Repeat { Get-Input (INPUT); }
     Until ((INPUT ≡ "Timeout") || (INPUT ≡ "Play"));  // wait for input
  Send-Output ("Stop-timer");         // exit action
  if (INPUT ≡ "Timeout") { State-IDLE (); }
  else {
    Send-Output ("Lamp-on");          // transition action
    State-PLAYING ( ); } }

State-PLAYING ( ) {
  STATE = PLAYING;
  Stop (); }

State-IDLE () {
  STATE = IDLE;
  Stop (); }
```

Fig. 9.7. FSM executor in pseudocode form for the FSM in Fig. 9.6.

9.3 TIMED AUTOMATA

A *timed automaton* is essentially a finite automaton (or finite state machine) extended with real valued variables modeling the logical clocks in the system [Bengtsson and Yi 2004]. This section briefly describes the concepts of timed automata. The discussions that follow are primarily from Cassandras and Lafortune [2010]. We begin with the definitions of language and deterministic automaton.

9.3.1 Language and Automata

The event set (E) of a DES is viewed as an alphabet. A sequence of events taken from this alphabet forms a *string*. An *empty string* is denoted by ε. A *language* in a DES is defined over an event set as follows: language (L) defined over an event set (E) is a set of finite length strings formed from the events in E. For example, let $E = \{a, b\}$ then the language may be defined as $L = \{\varepsilon, a, b, abb\}$.

An *automaton* is a device that is capable of representing a language according to well-defined rules, and it is classified into a deterministic, nondeterministic, or timed automaton. The automaton notion is best presented using a directed graph representation called a *state transition diagram* (which is the

same as that of an FSM). A *deterministic automaton* is defined as a quintuple $G_d = (S, E, f, \Gamma, s_0)$, where

1. S is a set of states (or state nodes)
2. E is a finite set of events
3. f: $S \times E \rightarrow S$ is the transition function with $f(s, e) \in S$
4. Γ: $S \rightarrow 2^E$ is the active event function
5. $s_0 \in S$ is the initial state

If S is a finite set, G_d is called a *deterministic* FSA or FSM. The functions f and Γ are specified in the state transition diagram. For example, in the state transition diagram of Fig. 9.1, $f(s_0, x_1) = s_1$ and $\Gamma(s_1) = \{x_1, x_2\}$. The transition function $f(s_0, x_1) = s_1$ indicates that there is a transition labeled by event x_1 from state s_0 to state s_1. The *active event set* $\Gamma(s)$ is a set of all events (e) for which $f(s, e)$ is defined.

9.3.2 Timed Automata

The concepts of the clock structure of an event set and the score of an event have key functions in timed automata.

- The *clock structure* ($\boldsymbol{V}$) associated with an event set (E) is a set $\boldsymbol{V} = \{\boldsymbol{v}_i: i \in E\}$ of clock sequences $\boldsymbol{v}_i = \{v_{i,1}, v_{i,2} \cdots\}$ with $v_{i,k} \in R^+$ for $i = 1, 2 \cdots m$ ($m = |E|$) and $k = 1, 2 \cdots$.
- The ***score*** ($N_{i,k}$) of event $i \in E$ after the k^{th} *state transition* on a given sample path is the number of times that event i has been activated in the interval $[t_0, t_k]$ where t_k is the k^{th} *state transition time*.

The score ($N_{i,k}$) serves as a pointer to $\boldsymbol{v}_i$, which specifies the next lifetime to be assigned to its clock when event i is activated. The clock structure ($\boldsymbol{V}$) is the input to the DES, and this information is translated into an actual event sequence ($\{e_1, e_2 \cdots\}$). Conceptually, the *next event* (e') is determined from the current state (s) and the clock structure ($\mathbf{V}$) as follows:

$$e' = h(s, \boldsymbol{V}).$$

Now, we are ready to define a timed automaton.

- A *timed automaton* (G_t) is a sextuple $G_t = (G_d, \boldsymbol{V})$ where $G_d = (S, E, f, \Gamma, s_0)$ is a deterministic automaton and $\boldsymbol{V} = \{\boldsymbol{v}_i: i \in E\}$ with $\boldsymbol{v}_i = \{v_{i,1}, v_{i,2} \cdots\}$ is a clock structure.

G_t is by definition a deterministic timed automaton. A nondeterministic timed automaton may be defined similarly as $G_{nt} = (G_{nd}, \boldsymbol{V})$. Starting from $s = s_0$ and $t = t_0$, the timed automaton (G_t) is evaluated as follows:

0. Set the initial clock values and initial scores to $\{y_i = v_{i,1}$ and $N_i = 1$ for $i \in \Gamma(s)\}$.
1. Evaluate the *feasible event set* (or active event set) $\Gamma(s)$.
2. Determine the smallest clock value (y^*) among the feasible event clock values $\{y_i\}$:

$$y^* = \min_{i \in \Gamma(s)} \{y_i\}. \tag{9.1}$$

3. Determine the triggering event (e') that defines y^* in Eq. 9.1:

$$e' = \arg\min_{i \in \Gamma(s)} \{y_i\}. \tag{9.2}$$

4. Determine the next state (s') and next event time (t'): $s' = f(s, e')$; and $t' = t + y^*$.
5. Determine the new clock values and new scores:

$$y_i' = \begin{cases} y_i - y^* & \text{if } i \neq e' \text{ and } i \in \Gamma(s) \\ v_{i,N_i+1} & \text{if } i = e' \text{ or } i \notin \Gamma(s) \end{cases} \quad i \in \Gamma(s'). \tag{9.3}$$

$$N_i' = \begin{cases} N_i + 1 & \text{if } i = e' \text{ or } i \notin \Gamma(s) \\ N_i & \text{otherwise} \end{cases} \quad i \in \Gamma(s'). \tag{9.4}$$

6. Iterates: set $t = t'$, $y_i = y_i'$, and $s = s'$, and return to step 1.

***Example 9.1.* (*Timed Automaton*).** The queuing system model depicted in Fig. 9.4 is a timed automaton $G_t = (S, E, f, \Gamma, s_0, \boldsymbol{V})$, where

1. $S = \{0, 1, 2 \cdots n\}$
2. $E = \{a, d\}$
3. $f(s, a) = s + 1$, $f(s, d) = s - 1$ for $s > 0$
4. $\Gamma(0) = \{a\}$, $\Gamma(s) = \{a, d\}$ for integer $s > 0$, $s_0 = 0$

The clock structure is $\boldsymbol{V} = \{\boldsymbol{v}_a, \boldsymbol{v}_d\}$, where $\boldsymbol{v}_a = \{v_{a,1}, v_{a,2}, v_{a,3} \cdots\}$ is a sequence of inter-arrival times and $\boldsymbol{v}_d = \{v_{d,1}, v_{d,2}, v_{d,3} \cdots\}$ is a sequence of service times. A sample clock structure and state trajectory of the queuing system are depicted in Fig. 9.8.

9.3.3 Timed Automata with Guards

If clock constraints, called *guards*, are used on transitions in a timed automaton to restrict its behavior, it becomes a *timed automaton with guards*. In this formalism, each transition has a clock constraint attached to it that specifies when the transition can occur. There is a single clock in timed automata, and the clock is reset to zero each time an event occurs. In a timed automaton with

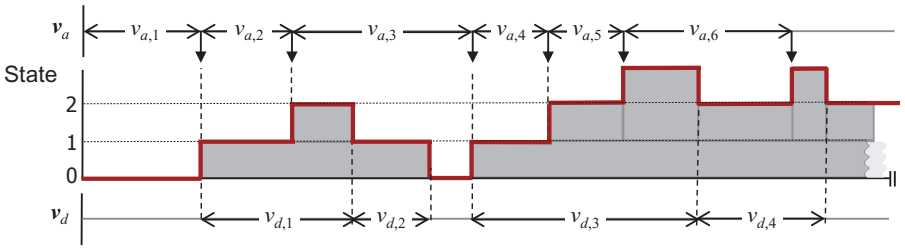


Fig. 9.8. Clock structure and state trajectory of the queuing system in Fig. 9.4.

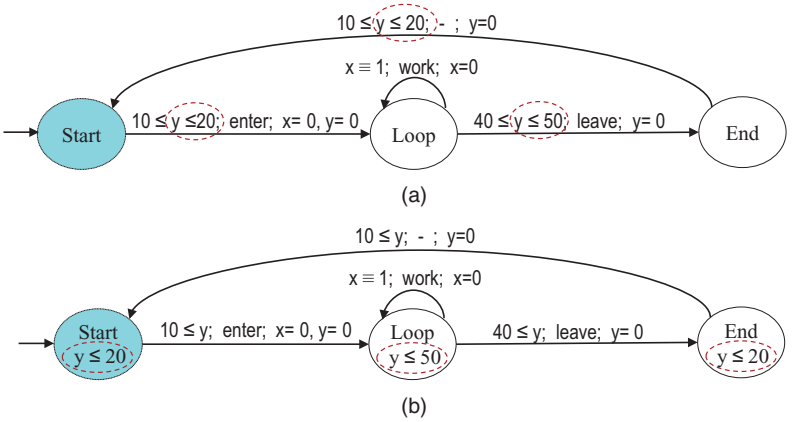


Fig. 9.9. Timed automata with (a) guards and (b) guards and invariants [Bengtsson and Yi 2004].

guards, a clock constraint may also be placed into a state node (often referred to as a *location*). A clock constraint in a state node is referred to as an *invariant condition* of the state node. Then, the automaton may remain in that state node as long as its clock value satisfies the invariant condition.

Figure 9.9(a) shows the timed automaton with guards introduced in Bengtsson and Yi [2004]. The system is initially at the Start node and it transitions to the Loop node if an enter event occurs when the value of clock y is between 10 and 20. During the transition, the two clocks x and y are set to zero. Then, the system transitions to the End node if a leave event occurs when the value of clock y is between 40 and 50, and so on. Figure 9.9(b) shows another timed automaton with guards and invariants.

A *transition edge* (t_e) in a timed automaton with guards is specified as a set of the triples of the form:

$$t_e = (\text{from node; } \textit{guard}; \textit{event}; \textit{reset}; \text{ to node}). \tag{9.5a}$$

Then, a formal specification of the timed automaton with guards (G_{tg}) is as given below [Cassandras and Lafortune 2010].

- A timed automaton with guards is a sextuple $G_{tg} = (S, E, C, Tra, Inv, s_0)$ where:
  1. S is a set of states
  2. E is a finite set of events
  3. $C = \{c_1 \cdots c_n\}$ is the finite set of clocks with $c_i(t) \in \mathrm{R}^+, t \in \mathrm{R}^+$
  4. $Tra \subseteq S \times \Psi(C) \times E \times 2^C \times S$ is the set of timed transition where $\Psi(C)$ is the set of admissible constraints for the clocks in C
  5. $Inv: S \rightarrow \Psi(C)$ is the set of state invariants
  6. $s_0 \in S$ is the initial state

Referring to the timed automaton with guards in Fig. 9.9(b), it is not clear how the system might leave a state when its invariant condition is about to be violated (e.g., when the clock y reaches to 20 in the Start node). According to Alur [1999], as a *requirement for the executability* of the timed automaton, some outgoing edges must be enabled when the invariant of the state node is violated. For this purpose, we use the *ε-transition edge* denoting an edge with an empty input string. An ε-transition edge, which is denoted by a dashed (dotted) edge, is automatically enabled when the invariant condition reaches its boundary.

Figure 9.10 depicts a timed automaton with an ε-transition edge that ensures the executability of the timed automaton in Fig. 9.9(b). The system initially residing in the Start state moves to the Loop state via a regular transition (solid arrow) when an enter event occurs during the clock time interval of $10 \leq y < 20$ or via an ε-transition (dashed arrow) when the value of the clock y reaches 20. As depicted in the figure, the ε-transition edge is specified as t_e = (Start; $y \equiv 20$; ε; $x = 0$ & $y = 0$; Loop), which is interpreted as "when clock y reaches 20, the system moves from the Start state to the Loop state after resetting x and y to zero" The system moves from the Loop state to the End state in the same way, and so on.

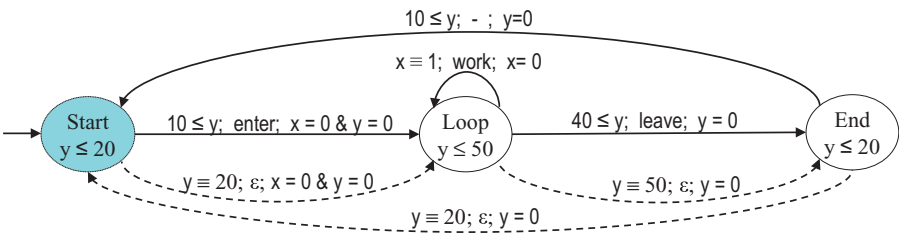


Fig. 9.10. A timed automaton with guards, invariants, and ε transitions.

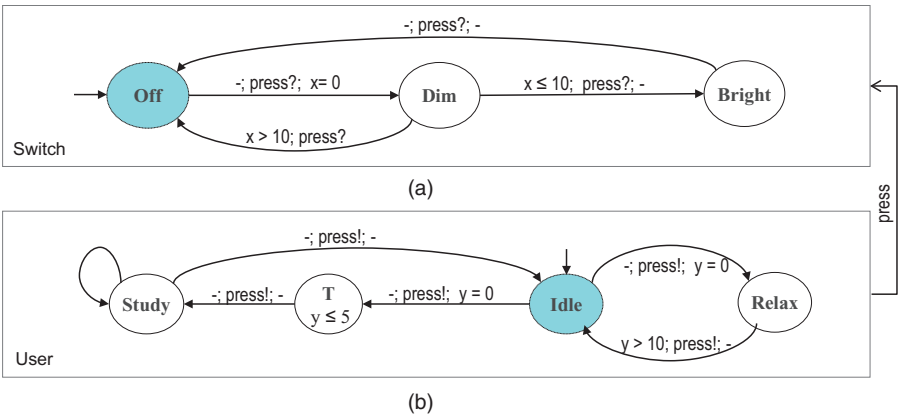


Fig. 9.11. Network of timed automata [Bengtsson and Yi 2004].

9.3.4 Networks of Timed Automata

According to Bengtsson and Yi [2004], a *network of timed automata* is a parallel composition of a set of timed automata called *processes*. Synchronous communication between the processes is via handshake synchronization using input and output actions (i.e., events): the alphabet "a" is assumed to consist of symbols for an input event denoted by "a?" and an output event denoted by "a!".

Figure 9.11 shows an example system composed of two timed automata: (a) a timed automaton for a time-dependent light switch and (b) another for its user. The user and the switch communicate using the message press. The user presses the switch (press!) and the light switch waits to be pressed (press?).

In general, each automaton in a timed automata network may need to be provided with a set of output messages. With the output message sets, the deterministic automaton in Section 9.3.1 and the timed automaton with guards in Section 9.3.3 are defined as follows.

- A deterministic automaton with an output message set O is defined a sextuple $\boldsymbol{G_{do}} = (S, E, f, \Gamma, s_0, O)$, where (1) S is a state set; (2) E is an event set; (3) f is a transition function; (4) Γ is active event function; (5) s_0 is the initial state; and (6) O is an output message set.
- A *timed automaton with guards and an output* message set O is defined a septuple $\boldsymbol{G_{tgo}} = (S, E, C, Tra, Inv, s_0, O)$, where (1) S is a state set; (2) E is an event set; (3) C is a finite set of clocks; (4) Tra is a set of timed transitions; (5) Inv is a set of state invariants; (6) s_0 is the initial state; and (7) O is an output message set.

Furthermore, a transition edge (t_e) in a timed automaton with guards and an output message set is specified as:

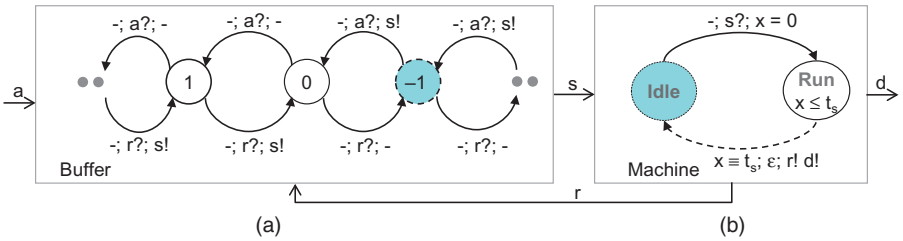


Fig. 9.12. Network of timed automata: (a) Buffer and (b) Machine.

$$t_e = (\text{from-node}; \textit{guard}; \textit{event}; \textit{action}; \text{to-node}). \tag{9.5b}$$

Figure 9.12 presents a network of timed automata representing the single server system in Fig. 9.3(a). The arrival of a job at the Buffer is signified by the input message arrive (a?) delivered from outside. If a request (r) message is received from the Machine, the Buffer returns a send (s) message to the Machine. In the Buffer model of Fig. 9.12, the input message (a? or r?) is regarded as an event. The Buffer model in the figure is a *deterministic automaton with an output* message set: $\boldsymbol{G}_{do} = (S, E, f, \Gamma, s_0, O)$, where (1) $S = \{\cdots 1, 0, -1 \cdots\}$; (2) $E = \{a, r\}$; (3) $f(k, a) = k + 1$ and $f(k, r) = k - 1$; (4) $\Gamma(k) = \{a, r\}$ for $k \in S$; (5) $s_0 = -1$; and (6) $O = \{s\}$.

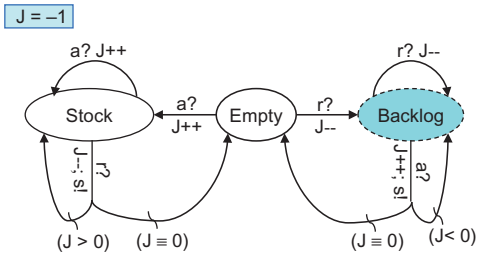
The Machine model in Fig. 9.12 has two output messages: job request message (r) and job departure message (d). The invariant boundary condition ($x \equiv t_s$) is regarded as an event of the e-transition edge. The Machine model is a timed automaton with guards with output messages: $G_{tgo} = (S, E, C, Tra, Inv, s_0, O)$, where (1) $S = \{Idle, Run\}$; (2) $E = \{s\}$; (3) $C = \{x\}$; (4) $Tra = \{(Idle; -; s; x = 0; Run), (Run; x \equiv t_s; \varepsilon; r!\ d!; Idle)\}$; (5) $Inv(Run) = x \leq t_s$; (6) $s_0 = Idle$; and (7) $O = \{r, d\}$.

9.4 STATE GRAPHS

This section presents an extended version of timed automata in which the concepts of state variables, system variables, and timers are incorporated. In this book, the resulting network of timed automata with state variables, timers, and system variables is called a *state graph*. To be more precise, a timed automaton is referred to as an *atomic state graph* and a network of timed automata as a *composite state graph*. An atomic state graph model is specified using a state transition table, and the interactions among the atomic models in a composite state graph are specified in an object interaction table.

9.4.1 State Variables and Macro States

The buffer model in Fig. 9.12 has an infinite number of states, which could be problematic in some situations such as when constructing its state transition



State Name	Input Event	Input Action	Transition Condition	Next State
Backlog	r?	J--	-	Backlog
	a?	J++; s!	J < 0	Backlog
			J ≡ 0	Empty
Empty	a?	J++	-	Stock
	r?	J--	-	Backlog
Stock	a?	J++	-	Stock
	r?	J--; s!	J > 0	Stock
			J ≡ 0	Empty
State variable: J = –1				

Fig. 9.13. State graph model of the buffer and its state transition table.

table. In general, an infinite state automaton can be developed into an FSA by introducing state variables and macro states.

Figure 9.13 presents a state graph for the buffer model in Fig. 9.12(a). The state graph model is obtained from the automaton model by treating J (number of jobs) as a state variable and by dividing the original state space into the three macro states: *Stock* = $\{J > 0\}$, *Empty* = $\{J \equiv 0\}$, and *Backlog* = $\{J < 0\}$. The state variable J is initially set to –1 and the start state is *Backlog*. As shown in Fig. 9.13, the graphical model is completely and concisely specified in the state transition table where (1) the input events (a? and r?) are listed in the Input Event column; (2) the state variable updates and message outputs are specified in the Input Action column; and (3) the branching conditions are specified in the Transition Condition column.

Exercise 9.1. Construct a state graph model from the infinite capacity buffer model in Fig. 9.12 using the two macro states: *Stock* = $\{J > 0\}$ and *Empty* = $\{J \leq 0\}$.

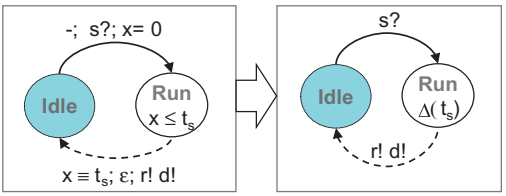
Exercise 9.2. Construct a state graph model of a finite capacity (n) buffer using three macro states: *Full* = $\{J \equiv n\}$, *Stock* = $\{n > J > 0\}$ and *Empty* = $\{J \leq 0\}$.

9.4.2 Timers and System Variables

Reproduced in Fig. 9.14(a) is the timed automaton with guards of the Machine in Fig. 9.12 with a clock variable (x), two timed transitions [(*Idle*; –; *s?*; $x = 0$; *Run*) and (*Run*; $x \equiv t_s$; ε; –; *Idle*)], and an invariant ($x \leq t_s$). We obtain the state graph model shown in Fig. 9.14(b) by replacing the invariant $x \leq t_s$ with a timer [$\Delta(t_s)$], eliminating the clock variable (x) and guards, and removing the ε event.

The ε-transition is now called an *internal transition*. The resulting atomic state graph model is concisely specified in the state transition table of Fig. 9.14(c).

Figure 9.15 shows an adventure game state graph model [Fishwick 1995]. At the Check state, a conditional branching is created based on the value of the standard uniform random variable (U), which is generated by a function call to RND. Variables such as U are called *system variables*. An internal transition



State		Input	Transition	Next
Name	Timer	Event	Action	State
Idle		s?		Run
Run	Δ(ts)		r! d!	Idle
State variables: -				

(c)

Fig. 9.14. (a) Timed automaton, (b) state graph, and (c) state transition table of the machine in Fig. 9.12.

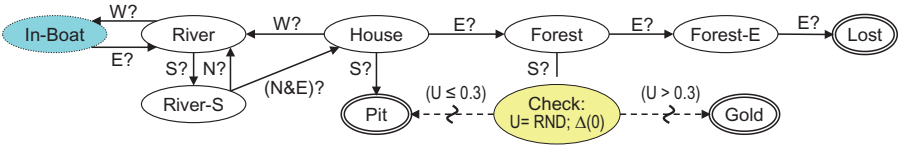


Fig. 9.15. State graph model of an adventure game.

is made into the Pit state if U is less than or equal to 0.3 or to the Gold state otherwise. A state transition triggered by an internal condition check is called an *internal transition* for which a dashed line arc is used. We adopt the convention that a function call for conditional branching is regarded as an entry action.

9.4.3 Conventions for Building State Graphs and State Transition Tables

An atomic state graph is an extended timed automaton in which (1) state variables and system variables are incorporated, (2) transition conditions are used instead of guards to restrict the transitions, (3) timers are used instead of the invariants and clocks, and (4) three types of actions (entry, input, and transition actions) are allowed. An atomic state graph without inputs and conditions becomes a *p*-time Petri net [Khansa et al. 1996] if the state nodes are treated as timed places of the Petri net.

Table 9.2 summarizes the conventions for constructing an atomic state graph. An external transition edge, which is denoted by the solid line arrow, may be specified by an input event followed by zero or more input actions and a transition condition followed by zero or more transition actions. An action may be an output message, a state variable update, or a function call for a system variable; a condition is a Boolean expression of the state variables or system variables. An internal transition edge (the dashed line arrow) may be specified by transition conditions and transition actions.

A number of symbols are introduced in order to increase readability: "**?**" denotes an input event, "**!**" denotes an output message, and "~" notes a condition. A state node must have a name and may be specified by an entry action

TABLE 9.2. Conventions for Constructing Atomic State Graphs

Components	Conventions
External Transition Edge	Input Event? Action0 Condition1 Action1
Internal Transition Edge	Condition2 Action2
State Nodes	Name Action3; $\Delta(t_0)$ · Initial State · Final State

TABLE 9.3. Template for a State Transition Table for an Atomic State Graph Model

State			Input		Transition		Next
Name	Action	Timer	Event	Action	Condition	Action	State
Initial State							
State							
. . .							
Initial values of state variables:							

and a timer [$\Delta(t_o)$]. A start state node is denoted using a dashed circle and a final state node using a double circle. In Table 9.2, Action0 is an input action; Action1 followed by Condition1 is a transition action; Action2 with Condition2 is a transition action and without a condition is regarded as an input action; and Action3 is an entry action. All conditions and actions are optional.

Table 9.3 presents a template for a state transition table for an atomic stage graph. There are up to eight columns in the table: State Name, State Action (i.e., entry action), State Timer, Input Event, Input Action, Transition Condition, Transition Action, and Next State. The initial state is listed first, and then the remaining (noninitial) states are listed in an arbitrary order. The initial values of the state variables are specified in the bottom row of the table.

A composite state graph is a network of atomic state graphs. Figure 9.16 presents an example of a composite state graph for a single server station. This composite state graph model is obtained by joining the Buffer state graph model in Fig. 9.13 and the Machine state graph model in Fig. 9.14. However, the Buffer model in Fig. 9.16 differs from that of Fig. 9.13. If the Buffer model of Fig. 9.13 (which can have an unlimited number of backlogs) is restricted to only one backlog, it can be reduced to the Buffer model given in Fig. 9.16. All actions in the Buffer model are input actions. The two models communicate with each other via send (s) and request (r) messages. Table 9.4 shows the state transition tables for the Buffer and Machine models of Fig. 9.16.

The message passing structure of the composite state graph is represented as an object interaction diagram. Figure 9.17 shows the object interaction diagram and object interaction table of the composite state graph in Fig. 9.16.

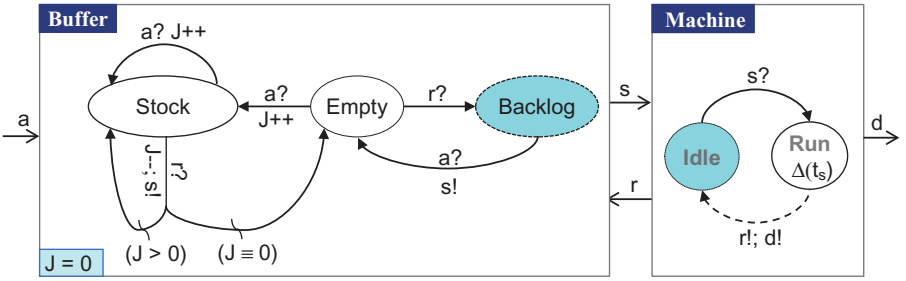


Fig. 9.16. Composite state graph model of a single server station.

TABLE 9.4. State Transition Tables for the Buffer and Machine Models in Figure 9.16

Buffer State			Input		Transition		
Name	Action	Timer	Event	Action	Condition	Action	Next State
Backlog	—	—	a?	s!	True	—	Empty
Empty	—	—	a?	J++	True	—	Stock
			r?	—	True	—	Backlog
Stock	—	—	a?	J++	True	—	Stock
			r?	J--; s!	J ≡ 0	—	Empty
					J > 0	—	Stock
Initial values of state variables: J = 0							

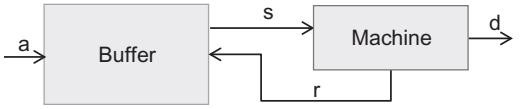
Machine State			Input		Transition		
Name	Action	Timer	Event	Action	Condition	Action	Next State
Idle	—	—	s?	—	True	—	Run
Run	—	$\Delta(t_s)$	Timeout	r!; d!	True	—	Idle
Initial values of state variables: —							

9.5 SYSTEM MODELING WITH STATE GRAPHS

This section presents examples of building state graph models for various discrete-event systems. The examples include a dining philosopher system, a table tennis game, a tandem line, a conveyor-driven serial line, and a traffic intersection system. More details on state graph modeling may be found on the website http://VMS-technology.com/Book/SGS.

9.5.1 State Graph Modeling of Dining Philosophers

Consider a dining philosopher system in which five philosophers sit at a round table where five chopsticks and five dishes have been placed [Fishwick 1995].



Atomic model	Input event	Output message
Buffer	a, r	s
Machine	s	r, d

Fig. 9.17. Object interaction diagram and object interaction table for the composite state graph in Fig. 9.16.

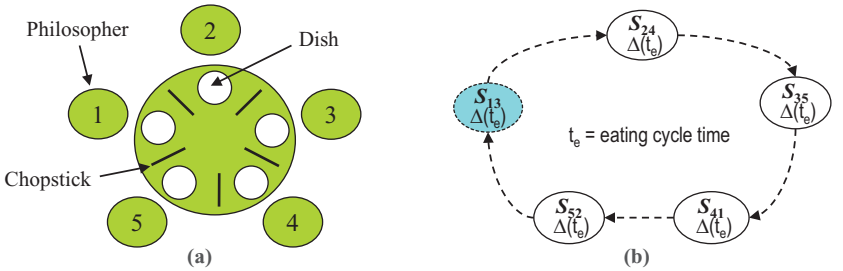


Fig. 9.18. Dining philosophers: (a) reference model and (b) state graph model.

For dining, each philosopher picks up two chopsticks, eats the food, and places the chopsticks back on the table. At most, two philosophers may be eating at the same time.

Let's assume the philosophers sit around the table and are numbered from 1 to 5 in a clockwise direction, as depicted in the reference model of Fig. 9.18(a). Let S_{jk} denote the situation where philosopher j and philosopher k dine simultaneously; then, the combinations for two philosophers eating at the same time are S_{13}, S_{24}, S_{35}, S_{41}, and S_{52}. Thus, the state set (S) is given by $S = \{S_{13}, S_{24}, S_{35}, S_{41}, S_{52}\}$. Figure 9.18(b) shows a state graph model of the dining philosopher system where t_e is the time needed for a philosopher to complete an eating cycle. It is assumed that the philosophers 1 and 3 are eating in the initial state.

9.5.2 State Graph Modeling of a Table Tennis Game

A good example for explaining the concept of state-based modeling is a simple table tennis game [Kim 1995]. In the simple table tennis game, each player serves twice before changing serves, and a game is over without a deuce if a player scores 11 points. We want to model one player (Player-A) of a single table tennis game with the proposed state graph.

Figure 9.19 presents the atomic state graph of a player (Player-A) in the single table tennis game. During a rally, the player is in either an Attack or Defense state. Upon moving into an Attack state, Player-A attacks its opponent with the ball. The time duration of Player-A in the Attack state is a_A. Then, Player-A switches to the Defense state if the ball has landed on the opponent's table (denoted by the Ball-A message and with a probability of

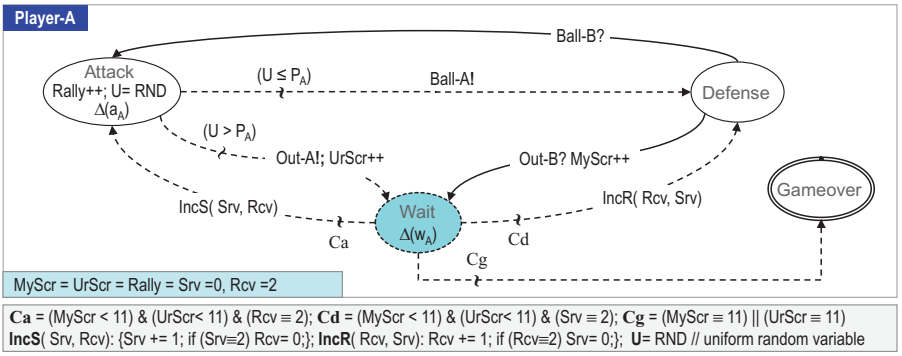


Fig. 9.19. Atomic state graph of Player-A in a single table tennis game.

TABLE 9.5. State Transition Table for the Atomic State Graph in Figure 9.19

State			Input		Transition		
Name	Action	Timer	Event	Action	Condition	Action	Next State
Wait	—	$\Delta(w_A)$	Timeout	—	Ca	IncS(Srv, Rcv)	Attack
				—	Cd	IncR(Rcv, Srv)	Defense
				—	Cg	—	Gameover
Attack	Rally++; U = RND	$\Delta(a_A)$	Timeout	—	$U \le P_A$	Ball-A!	Defense
				—	$U > P_A$	Out-A!; UrScr++	Wait
Defense	—	—	Ball-B?	—	True	—	Attack
			Out-B?	MyScr++	True	—	Wait

State variables: Clock = MyScr = UrScr = Rally = Srv = 0, Rcv = 2

P_A) or to the Wait state if the ball is out (denoted by the Out-A message and with a probability of $1 - P_A$). From the current state of Wait, Player-A switches to the Attack state to serve or to the Defense state to receive. The time delay in the Wait state is w_A. The scores are updated every time the ball is out, and the Gameover state occurs when the score of either player reaches 11. Table 9.5 shows the state transition table for the state graph model in Fig. 9.19.

The state set of the state graph model is S = {Attack, Defense, Wait, Gameover}, and the input event set and output message set are E = {Ball-B, Out-B} and O = {Ball-A, Out-A}. A number of state variables are required in order to manage the state graph model: MyScr (player's score), UrScr (opponent's score), Rally (number of rallies), Srv (serve count), and Rcv (receive count). In the atomic state graph, all state variables are set to zero, except Rcv which is set to 2, meaning that Player-A will serve first. The transition conditions associated with the Wait state are defined as follows:

Cg = (MyScr ≡ 11) || (UrScr ≡ 11) // Gameover condition (without deuce);
Cd = (MyScr < 11) & (UrScr < 11) & (Srv ≡ 2) // Receive (Defense) condition;
Ca = (MyScr < 11) & (UrScr < 11) & (Rcv ≡ 2) // Serve (Attack) condition.

The transition conditions associated with the Attack state are given by the probabilities of P_A and $1 - P_A$. The ++ symbol denotes an increment operation and the functions IncS ( ) and IncR ( ) are defined as follows:

$$IncS\ (\text{Srv, Rcv}): \{\text{Srv} = \text{Srv} + 1;\ If\ (\text{Srv} \equiv 2)\ then\ \text{Rcv} = 0;\}, \tag{9.6a}$$

$$IncR\ (\text{Rcv, Srv}): \{\text{Rcv} = \text{Rcv} + 1;\ If\ (\text{Rcv} \equiv 2)\ then\ \text{Srv} = 0;\}. \tag{9.6b}$$

Consider a table tennis game involving two players and a friend. The game rules are the same as before, but the friend watches the game and may ask the players to quit at any time while the game is in progress. Figure 9.20 shows a composite state graph involving the Friend, where t_q denotes the quit time. Observe that the state graph model for Player-A has an additional Quit state, as well as a new input event (Quit) and an output message (Over). Figure 9.21 shows the object interaction diagram involving the two players and the friend.

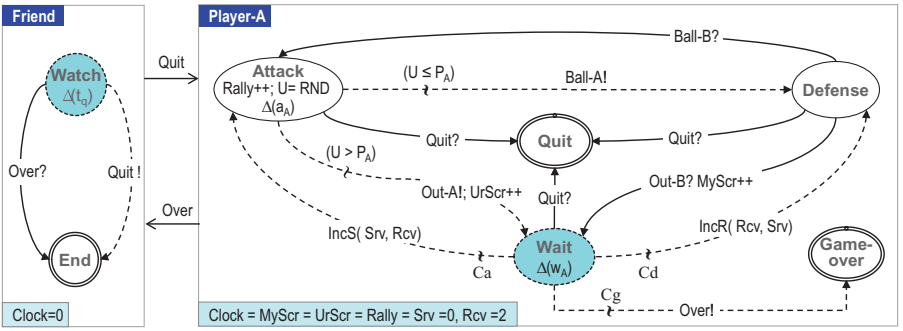


Fig. 9.20. Composite state graph model involving the Player-A model from Fig. 9.19 and the Friend.

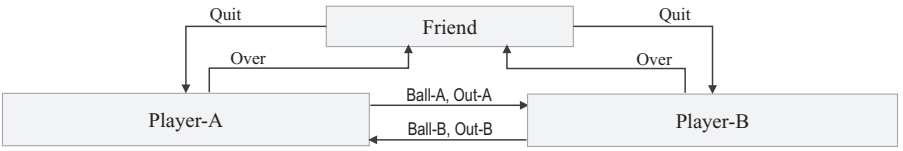


Fig. 9.21. Composite state graph model (object interaction diagram) of the table tennis game.

9.5.3 State Graph Modeling of a Tandem Line

Figure 9.22(a) shows the reference model of a single server system composed of a machine with service time (t_s), an unlimited capacity buffer, and a job generator with an inter-arrival time (t_a). Figure 9.22(b) shows a composite state graph model of the single server system. The generator model is a single state timed automaton that sends arrive message "a" to the buffer model every time a job is generated. The composite state graph model of the single server station in Fig. 9.22(b) is the same as that shown in Fig. 9.16.

By concatenating single server station models (of the form given in Fig. 9.16) to the single server system model in Fig. 9.22(b), a composite state graph model of a tandem line is constructed as depicted in Fig. 9.23. The messages passed between the atomic state graph models are as follows:

a = arrive; e = enter; s = start; r = request; u = unload; w = withdraw.

9.5.4 State Graph Modeling of a Conveyor-Driven Serial Line

The conveyor-driven serial line was modeled using an event graph in Chapter 4 and was modeled using an activity cycle diagram (ACD) in Chapter 6. This section presents a procedure for modeling the same serial line using a state graph. The tandem line in Fig. 9.23 becomes a conveyor-driven serial line if each buffer in the line, except Buffer-1, is replaced by an accumulating conveyor.

Figure 9.24(a) shows the reference model of a conveyor specified by its buffering capacity (b) and convey time (t_c). Let J denote the number of jobs accumulated (or stored) at the end of the conveyor segment; then, the available conveying capacity (A) is expressed as:

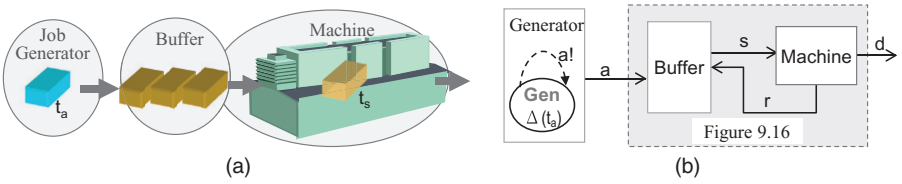


Fig. 9.22. (a) Reference model and (b) state graph model of a single server system.

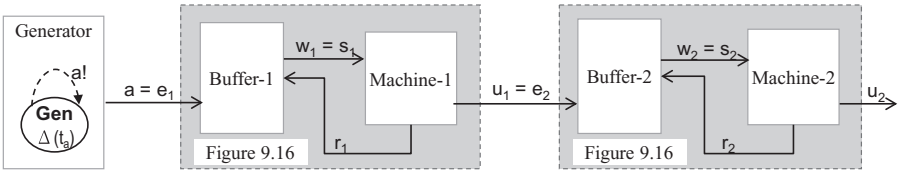


Fig. 9.23. Composite state graph model of a tandem line.

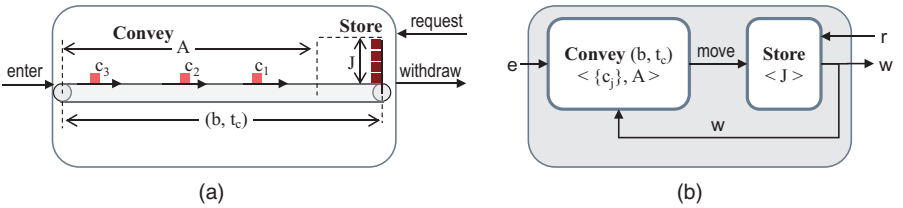


Fig. 9.24. (a) Reference model and (b) object interaction diagram of the conveyor.

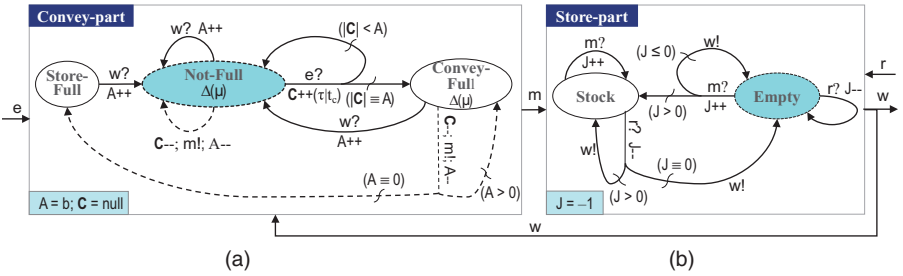


Fig. 9.25. Composite state graph model of the conveyor in Fig. 9.24.

$$A = b - J. \tag{9.7}$$

Let τ denote the current simulation clock value at the time job j enters the conveyor; then, the *end-of-convey (EOC) time* c_j of the job is expressed as:

$$c_j = \tau + t_c. \tag{9.8}$$

The input events in the conveyor system are *enter* (e) and *request* (r), while the output message is *withdraw* (w). As depicted in the reference model, each job is regarded as being conveyed to the end point of the conveyor where it is accumulated vertically. Thus, as shown in Fig. 9.24(b), the conveyor system has two parts: a Convey part and a Store part. The Convey part is specified using a set of EOC times $\mathbf{C} = \{c_j\}$ and the available conveying capacity (A); the Store part is specified by the number of accumulated jobs (J). The Convey part and Store part interact with each other via *job move* (m) and *job withdraw* (w) messages.

Figure 9.25 presents a composite state graph model of the conveyor. The state space of the Convey part can be partitioned into three macro states, as follows:

1. The Store-Full state (the store part is full): $|\boldsymbol{C}| \equiv A \equiv 0$
2. The Convey-Full state: $|\boldsymbol{C}| \equiv A > 0$
3. The Not-Full state: $|\boldsymbol{C}| < A$

The state space of the Store part, which is a finite capacity buffer, can be partitioned into two macro states: (1) the Empty state: $J \leq 0$; and (2) the Stock state: $J > 0$.

The Convey part has a *timer value* (μ) denoting the time duration from the current simulation clock time (τ) to the next EOC time (c_1). The timer value (μ) is given by:

$$\mu = c_1 - \tau. \tag{9.9}$$

If **C** is null, we set $\mu = \infty$. The en-queue and de-queue operations for manipulating $\boldsymbol{C} = \{c_j\}$ are defined as follows:

$$\begin{aligned} &\mathbf{C}{+}{+}\,(\tau \mid \mathrm{t_c}): \text{add a new element } c_{n+1} = \tau + t_c \text{ into } \mathbf{C}, \text{ where } \mathrm{n} = |\mathbf{C}| \\ &\mathbf{C}{-}{-}: \text{remove the first element } c_1 \text{ from } \boldsymbol{C}. \end{aligned} \tag{9.10}$$

A state transition table for the Convey part of the composite state graph model of the conveyor in Fig. 9.25 is given in Table 9.6.

Exercise 9.3. Construct a state transition table for the store part of the composite state graph model of the conveyor presented in Fig. 9.25(b).

Figure 9.26 shows the object interaction diagram of a conveyor-driven serial line of Buffer-1 → Machine 1 → Conveyor 2 → Machine 2 → · · · . The messages passed between the objects of the conveyor are *move* and *withdraw*, and the messages passed between the machines and conveyors are *request*, *withdraw/start*, *finish*, *grant*, and *unload/enter*. The interactions among the objects in Fig. 9.26 are as follows.

TABLE 9.6. State Transition Table for the Convey Part of the Composite State Graph Model of the Conveyor in Figure 9.25(a)

State			Input		Transition		Next
Name	Action	Timer	Event	Action	Condition	Action	State
Not-Full	—	$\Delta(\mu)$	w?	A++	True	—	Not-Full
			e?	**C**++($\not\exists$t$_c$)	|**C**| < A	—	Not-Full
					|**C**| ≡ A	—	Convey-Full
			Timeout	**C**—; m**!**; A—	True	—	Not-Full
Convey-Full	—	$\Delta(\mu)$	w?	A++	True	—	Not-Full
			Timeout	**C**—; m**!**; A—	A ≡ 0	—	Store-Full
					A > 0	—	Convey-Full
Store-Full	—	—	w?	A++	True	—	Not-Full

State variables: A = b, **C** = Null

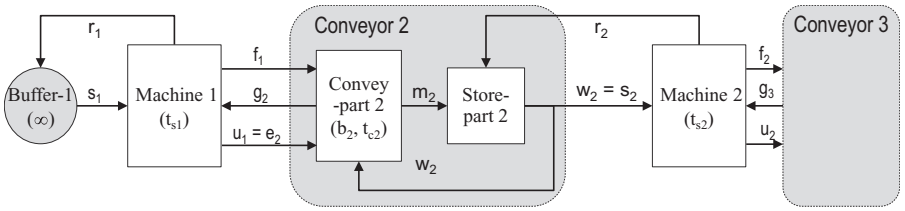


Fig. 9.26. Object interaction diagram of a conveyor-driven serial line.

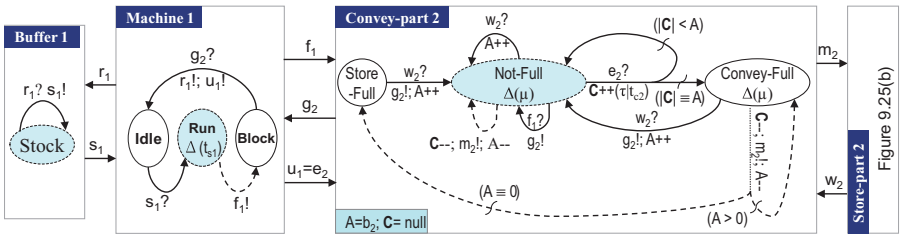


Fig. 9.27. Composite state graph model of the conveyor-driven serial line.

1. Machine 1 sends a job *request message* (r_1) to the infinite capacity buffer Buffer-1, which in turn sends back a machine *start message* (s_1).
2. Machine 1 sends a job *finish message* (f_1) to Convey-part 2, which in turn sends a *grant message* (g_2) back if Conveyor 2 is not full. Upon receiving g_2, Machine 1 sends a job *unload message* (u_1) to Convey-part 2 in which u_1 is regarded as a job *enter message* (e_2).
3. Convey-part 2 sends a job *move message* (m_2) to Store-part 2 when a job reaches the end of the conveyor.
4. Store-part 2 sends a job *withdraw message* (w_2) to Machine 2 (and to Convey-part 2) upon receiving a job request message (r_2) from Machine 2 in which w_2 is regarded as a machine start message (s_2).

Figure 9.27 shows a composite state graph model for the front region of the conveyor-driven serial line (i.e. Buffer 1, Machine 1, and Convey-part 2). The atomic state graph model for the Store-part is the same as that in Fig. 9.25(b). The Buffer 1 atomic model is a simple automaton generating an output message (s_1) every time it receives an input event message (r_1). The Machine 1 model, which is initially in the Run state, sends out an output message (f_1) and goes into the Block state after a time delay of t_{s1}. Upon receiving g_2, the Machine 1 model sends r_1 (to Buffer-1) and u_1 (to Convey-part 2) messages, and then moves into the Idle state. The structure of the Convey-part 2 model in Fig. 9.27 is the same as that of the Convey-part in Fig. 9.25(a), except the additional function for handling the finish (f_1) and grant (g_2) messages. A state transition table for the atomic state graph model Convey-part 2 is presented in Table 9.7.

TABLE 9.7. State Transition Table for the Convey-Part 2 Model in Figure 9.27

State			Input		Transition		Next State
Name	Action	Timer	Event	Action	Condition	Action	
Not-Full	—	$\Delta(\mu)$	w_2?	A++	True	—	Not-Full
			f_1?	g_2!	True	—	Not-Full
			e_2?	**C**++(τ|t_{c2})	|**C**| < A	—	Not-Full
					|**C**| ≡ A	—	Convey-Full
			Timeout	**C**––; m_2!; A––	True	—	Not-Full
Convey-Full	—	$\Delta(\mu)$	w_2?	g_2!; A++	True	—	Not-Full
			Timeout	**C**––; m_2!; A––	A ≡ 0	—	Store-Full
					A > 0	—	Convey-Full
Store-Full	—	—	w_2?	g_2!; A++	True	—	Not-Full

State variables: A = b_2, **C** = Null

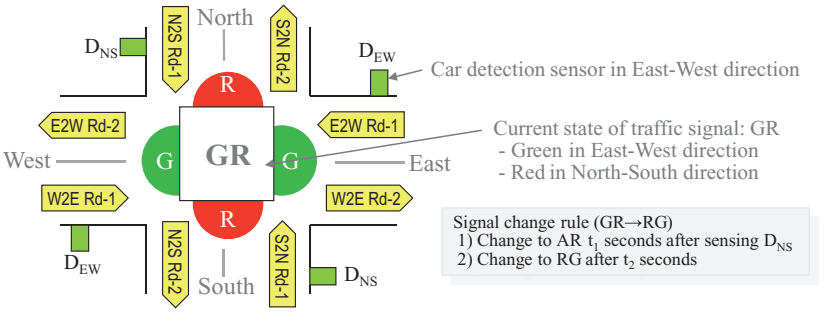
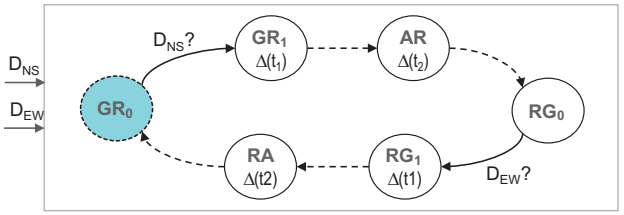


Fig. 9.28. Reference model of a traffic intersection system.

9.5.5 State Graph Modeling of Traffic Intersection Systems

Figure 9.28 shows the reference model of a traffic control system at a simple junction [Fishwick 1995]. The current state is designated as green-red (GR) because the traffic lights in the east-west (EW) direction are green and those in the north-south (NS) direction are red. There is a car detection sensor at each road near the junction as indicated in the figure. A detection signal coming from the east or west road is designated as D_{EW} and that from the north or south road as D_{NS}. The traffic light changes its state from GR to RG as follows: the state is changed from GR to AR t_1 seconds after sensing a D_{NS} signal, and then it is automatically changed to RG after t_2 seconds.

Figure 9.29 shows the state graph model and its state transition table of the traffic signal system depicted in Fig. 9.28 assuming that the system is initially in the GR state. When a D_{NS} signal is sensed, the traffic lights in the EW



State		Input	Next
Name	Timer	Event	State
GR0	-	D_{NS}?	GR1
GR1	$\Delta(t_1)$	-	AR
AR	$\Delta(t_2)$	-	RG0
RG0	-	D_{EW}?	RG1
RG1	$\Delta(t_1)$	-	RA
RA	$\Delta(t_2)$	-	GR0

Fig. 9.29. State graph model and state transition table of the traffic signal system.

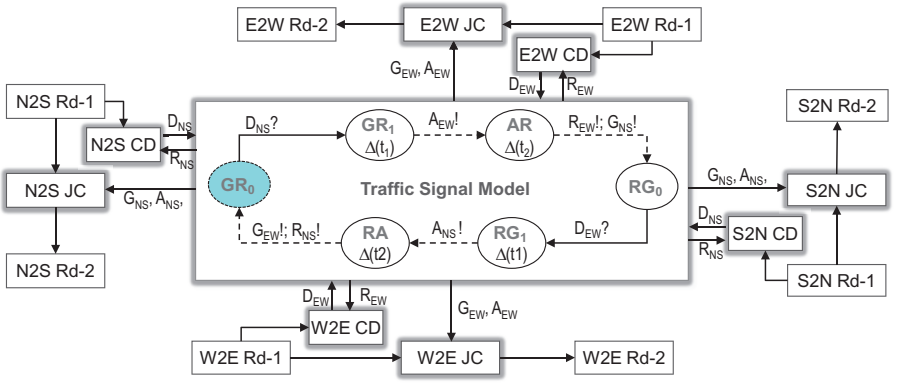


Fig. 9.30. Composite state graph model of the traffic intersection system in Fig. 9.28.

direction are changed to amber after t_1 (while the lights in the NS direction stay red), which is designated as AR. Then, after t_2 seconds, the traffic lights in the EW direction change to red and those in the NS direction change to green, which is designated as RG. The GR state is divided into GR_0 (GR before a D_{NS} is detected) and GR_1 (GR after a signal detection). Because the system is symmetric, the RG state is divided into RG_0 and RG_1 in the same way. Thus, the state set is given by $S = \{GR_0, GR_1, AR, RG_0, RG_1, RA\}$. Then, the input event set is given by $E = \{D_{NS}, D_{EW}\}$.

Figure 9.30 presents the overall structure of a composite state graph model for the traffic intersection system in Fig. 9.28. Located at the center of the composite model is the traffic signal model of Fig. 9.29, which interacts with the junction (JC) models and car detection (CD) models. There are four JC models and four CD models: one for each of the four traffic directions of E2W (east-to-west), W2E (west-to-east), N2S (north-to-south), and S2N (south-to-north). The traffic signal model receives a car detection input (D) from the CD models and sends out traffic light outputs G (green), A (Amber), and R (Red).

Let's assume that in Fig. 9.30 the traffic signal model is at the RG_0 state (red lights in east-west direction and green in north-south direction) and has just received an input message D_{EW} (i.e., a car is detected in the E2W or W2E

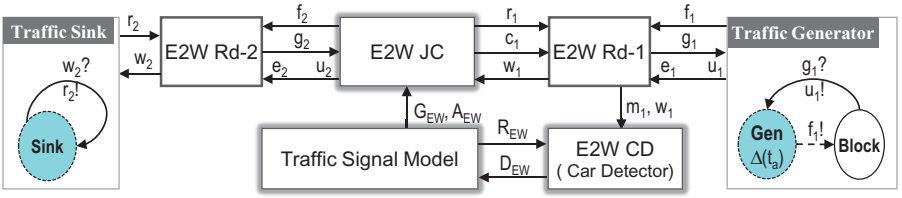


Fig. 9.31. Object interaction diagram for the east-west direction traffic subsystem.

direction). Then, the traffic signal model performs the following sequence of transitions: (1) the state is changed to RG_1 immediately and stays there for t_1 seconds; (2) the state is changed to RA after sending an output message (A_{NS}) to the S2N JC and N2S JC and remains there for t_2 seconds; (3) finally, the state is changed to GR_0 after sending output messages R_{NS} (to the S2N CD and N2S CD) and G_{EW} (to E2W JC and W2E JC).

Figure 9.31 shows the object interaction diagram for the E2W direction traffic subsystem together with the state graph models of the Traffic Generator and Traffic Sink. The atomic state graph models in the object interaction diagram include the E2W Rd-1 (a road segment leading to the junction), E2W JC (the junction in the E2W direction), E2W CD (car detection sensor in the E2W direction), E2W Rd-2 (a road segment leaving the junction), and the traffic signal model. There are a number of messages that are passed among the atomic models, as follows:

c = cancel; e = enter; f = finish; g = grant;
m = move; r = request; u = unload; w = withdraw;
G_{EW}, A_{EW}, R_{EW} = green, amber, and red lights in east-west direction;
D_{EW} = car detection signal in east-west direction.

The Traffic Generator, which generates a car every t_a seconds in the Gen state, sends f_1 to E2W Rd-1 when a car is generated and waits in the Block state for a grant message (g_1). Upon receiving a g_1, it returns to the Gen state after sending an unload message (u_1) back to E2W Rd-1 [where u_1 is regarded as an enter message e_1]. The Traffic Sink, which accepts unlimited number of cars from E2W Rd-2, sends a request message (r_2) to E2W Rd-2 every time it receives a car withdraw message (w_2).

Figure 9.32 shows the detailed state graph models for E2W Rd-1, which is modeled as an accumulating conveyor (refer to Fig. 9.25). On top of the baseline conveyor model Fig. 9.27, the canceling of a job request is introduced as follows: In the Empty state, a cancel message (c_1) from E2W JC would decrease the backlog (J++). This cancellation prevents the illegal withdrawal of cars from E2W Rd-1 (i.e., prevents cars from moving into E2W JC when the traffic light is not green). Assume that E2W Rd-1 remains in the Empty state and E2W JC sends a request message (r_1) to E2W Rd-1. If E2W JC receives the

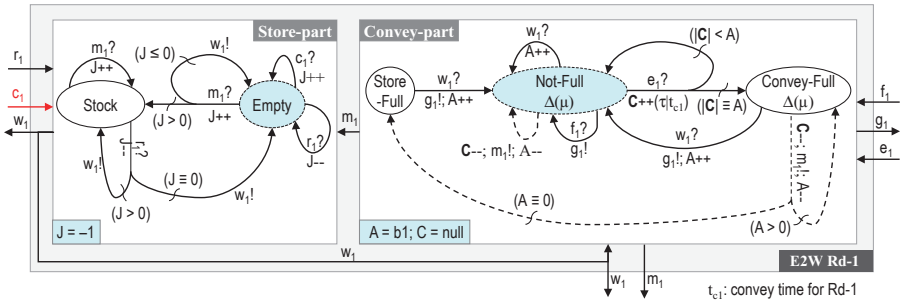


Fig. 9.32. State graph models for the road segment E2W Rd-1 (refer to Fig. 9.25).

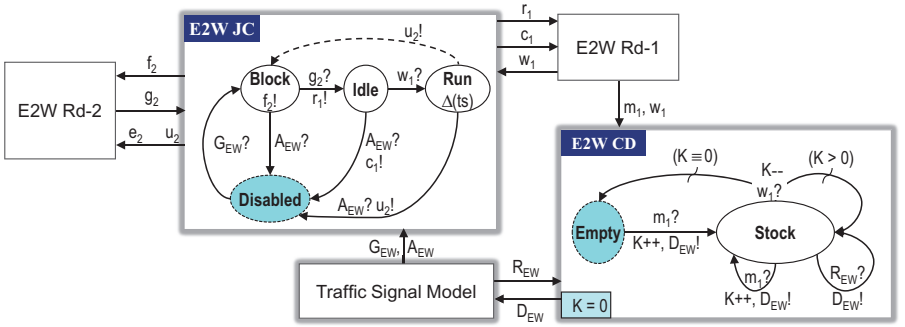


Fig. 9.33. State graph models for the junction E2W JC and the car detector E2W CD.

amber signal (A_{EW}) prior to a withdraw message (w_1), the withdraw message that will arrive later cannot be accepted, which results in an illegal withdrawal (the withdrawn car is lost from the traffic intersection system).

The atomic model of E2W Rd-2 in Fig. 9.31 has the same structure as that of E2W Rd-1. Recall from the previous section that b_1 and t_{c1} denote the capacity (maximum number of cars) and convey time of the road segment Rd-1, respectively.

Figure 9.33 shows the remaining two models (E2W JC and E2W CD) in Fig. 9.31. The junction model E2W JC in Fig. 9.33 is essentially a Machine model (see Fig. 9.27) augmented with a Disabled state. The E2W JC model goes into the Disabled state every time an A_{EW} is received (i.e., the traffic light becomes amber) while sending a cancel message (c_1) to E2W Rd-1 if it was in the Idle state or an unload message (u_2) to E2W Rd-2 if it was in the Run state, and it moves into the Block state (i.e., enabled) if a G_{EW} is received. Unlike the Machine 1 model in Fig. 9.27 (where the machine accepts a job for processing and then tries to unload the finished job), the E2W JC model accepts a job for processing only when the finished job is guaranteed to be unloaded. The car detect model E2W CD in Fig. 9.33 is essentially a buffer model where the state variable K denotes the number of cars stored at the end of the road segment E2W Rd-1.

9.6 SIMULATION OF COMPOSITE STATE GRAPH MODELS

This section presents a structured method of constructing a state graph simulator for executing composite state graph models. Recall that a composite state graph is a network of atomic state graph models and that an atomic state graph model is a timed automaton augmented with state variables and system variables.

9.6.1 Framework of a State Graph Simulator

Figure 9.34 shows the overall framework for constructing a state graph simulator of a composite state graph model. As depicted in the figure, a state graph simulator consists of a synchronization manager (*sync manager* for short) and a number of atomic simulators, one for each atomic model in the composite state graph model. Each atomic simulator is an FSM, and the sync manager, which is responsible for synchronizing the local simulation clocks of the atomic simulators, is represented as an atomic state graph with an instantaneous timer [$\Delta(0)$]. The synchronization method used here is based on the concept of the sync manager presented in Lee et al. [2010b] and its earlier version is presented in Lee et al. [2010a].

The message types passed between the sync manager and the individual atomic simulators are the time advance request (TAR), time advance grant (TAG), message send request (MSR), and message delivery packet (MDP).

```
TAR = (Time, IMS, ID); // IMS = input message set; ID = atomic simulator
    ID
TAG = (Time, ID); // Time = current simulation time
MSR = (Msg, ID); // Msg = input/output message
MDP = (Msg, Time, ID);
```

Figure 9.35 shows the overall structure of the state graph simulator for the table tennis game model given in Fig. 9.21. There are three atomic simulators (Player-A, Player-B, and Friend) that are connected to the sync manager. The time advance request table (TART), the message send request queue (MSRQ),

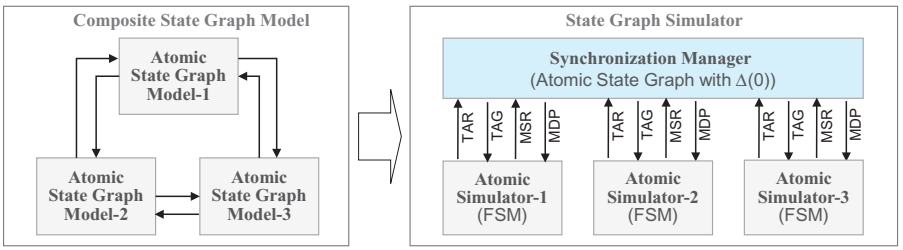


Fig. 9.34. Building a state graph simulator from a composite state graph model.

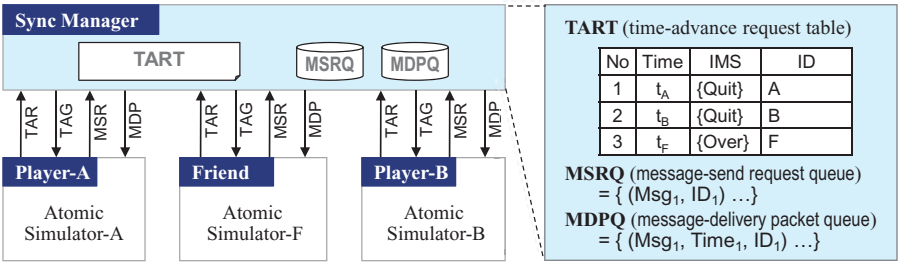


Fig. 9.35. State graph simulator for composite state graph model in Fig. 9.21.

and the message delivery packet queue (MDPQ) are stored in the sync manager.

9.6.2 Synchronization Manager

In the state graph simulator, all interactions among the atomic simulators are made through the sync manager. At the beginning, each atomic simulator sends a TAR message to the sync manager. Then, the sync manager builds a TART and sends back a TAG message to the atomic simulator that has the smallest TAR time value. An instance of a TART is depicted in Fig. 9.35.

Upon receiving the TAG message, the atomic simulator advances its simulation clock and moves into the next state. Immediately after this state transition, the atomic simulator may send an MSR message to the sync manager, which will store the received MSR message in a queue called the message send request queue (MSRQ). Because the Msg in the MSR could be an "input" to more than one atomic simulator, it is temporally stored in another queue named MDPQ.

Figure 9.36(a) shows the state graph model of the sync manager. At the beginning of the execution, each atomic simulator sends a TAR message to the sync manager, which then constructs the TART while remaining in the start state Start. When the counter (n) reaches N_S (number of atomic simulators), the sync manager moves to the FindTAG state where the function Find (TAG, Found) is invoked to find a valid TAG record from the TART. A pseudocode form of the function Find (TAG, Found) is given in Fig. 9.36(b). If a TAG record is found, the sync manager moves to the Receive state after setting the simulation clock to the TAG time (Now = TAG.Time), sending out the TAG message (TAG!), and setting the TAR counter to 1 (m = 1).

In the Receive state, the sync manager accepts MSR messages from the atomic simulator that has just received a TAG. The atomic simulator will send MSR messages first if any exist and send a TAR message. All MSR messages received are stored in the MSRQ using the en-queue function MSR→MSRQ, and each of the received TAR messages is stored in the TART using the table update function TAR→TART. When all input TAR messages are processed (m ≡ 0), the sync manager moves to the BuildMDPQ state.

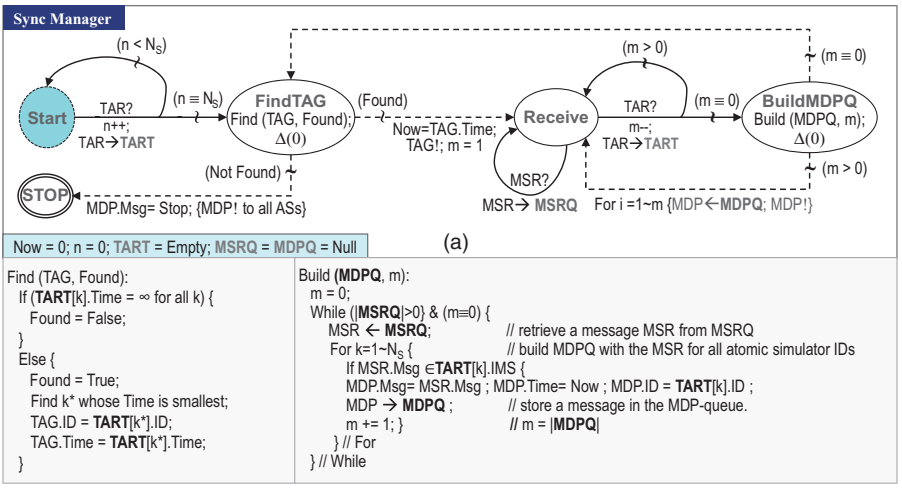


Fig. 9.36. State graph model of the sync manager presented in Fig. 9.35.

In the BuildMDPQ state, the function Build (MDPQ, m) is invoked to build a MDPQ with an MSR record retrieved from the MSRQ for all matched atomic simulator IDs. A pseudocode form of the function Build (MDPQ, m) is given in Fig. 9.36(c). If m is positive, which means MDPQ is not empty, all MDP records in MDPQ are retrieved individually and sent to the respective atomic simulators; then, the state is changed to Receive. If a MDPQ is not built (m ≡ 0), the sync manager moves again to the FindTAG state.

In the FindTAG state, the function Find (TAG, Found) is invoked to find a valid TAG record from the TART. If a TAG record is found, the sync manager moves again to the Receive state; if a message is not found, the sync manager moves to its final state Stop after sending the MDP message Stop to all atomic simulators (ASs).

Table 9.8 is a state transition table for the sync manager model presented in Fig. 9.36. It is a general purpose sync manager that can be used for any composite state graph model. The time synchronization procedure may be summarized as follows:

1. In the Start state, put the TAR messages received from the atomic simulators into the TART. If finished, move to the FindTAG state.
2. In the FindTAG state, invoke the function Find (TAG, Found) to choose an atomic simulator with the smallest next event time. If found, move to the Receive state after updating the simulation time (Now = TAG.Time) and sending a TAG message to the chosen simulator. If not found (because every atomic simulator is in its final state), send a Stop message to every atomic simulator and move to the STOP state.

TABLE 9.8. State Transition Table for the Sync Manager Model of Figure 9.36

State			Input		Transition		
Name	Action	Timer	Event	Action	Condition	Action	Next State
Start	—	—	TAR?	n++;	($n < N_S$)	—	Start
				TAR→ TART	($n \equiv N_S$)	—	FindTAG
FindTAG	Find (TAG, Found)	Δ(0)	—	—	(Found)	Now = TAG. Time; TAG!; m = 1	Receive
					(Not Found)	MDP.Msg = Stop; MDP! to all ASs	STOP
Receive	—	—	MSR?	MSR→ MSRQ	True	—	Receive
			TAR?	m—;	(m > 0)	—	Receive
				TAR→ TART	(m ≡ 0)	—	BuildMDPQ
BuildMDPQ	Build (MDPQ, m)	Δ(0)	—	—	(m > 0)	For i = 1~m { MDP←MDPQ; MDP!}	Receive
					(m ≡ 0)	—	FindTAG

State variables: Now = 0; n = 0; TART = Empty; MSRQ = Null; MDPQ = Null

3. In the Receive state, receive MSR/TAR messages from the atomic simulator that has just received a TAG. (The atomic simulator will send MSRs first, if any, and then send a TAR.) Store the MSRs in the MSRQ, store the TARs in the TART, and move to the BuildMDPQ state.
4. In the BuildMDPQ state, invoke the function Build (MDPQ, m) to build an MDPQ for an MSR record retrieved from the MSRQ. If the MDPQ is not empty (m > 0), all MDPs are retrieved individually and sent to their respective atomic simulators. Then, the state is changed to Receive. If m = 0, the state is changed to FindTAG.

As discussed in Section 9.2.4 (Fig. 9.7), writing a simulation module for an FSM model from its state transition table is straightforward. Figure 9.37 shows the sync manager simulation module written in pseudocode form. It consists of a main program Sync-Manager () in which the state variables are initialized and four state transition routines {State-Start (N_s), State-FindTAG (), State-Receive (m), and State-BuildMDPQ ()} are created, which is one for each state. It is assumed that the communication between the main program and state transition routines are made using the two I/O functions:

```
Sync-Manager (N_S) { // main program of Sync manager (N_S = number of atomic simulators)
   Now = 0; n = 0; TART = Empty; MSRQ = Null; MDPQ = Null;          // initialize simulation clock & state variables
   State-Start (N_S);                                               // start with "start state"
}
State-Start (N_S) { // state-transition routine for start-state "START"
   STATE = START;                                                   // state update
   Repeat { Get-Input( INPUT);                                      // input
            If (INPUT.Type ≡ "TAR") { n++; TART[n] = INPUT.Record; }  // transition action
   } Until (n ≡ N_S);
   State-FindTAG( );                                                // move to next state
}
State-FindTAG () {
   STATE = FINDTAG;
   Find (TAG, Found);  // find an atomic simulator that has the smallest local simulation time
   If Found { Now = TAG.Time;  Send-Output( "TAG", TAG); m = 1;  State-Receive(m); }
   Else { For each ID∈ ASs { MDP = (Stop, Now, ID); Send-Output( "MDP", MDP); } State-STOP(); }
}
State-Receive (m) {
   STATE = RECEIVE;
   Repeat { Get-Input( INPUT); } Until ((INPUT.Type ≡ "MSR") || (INPUT.Type ≡ "TAR")); // read a MSR or TAR message
   If (INPUT.Type ≡ "MSR") { Enqueue(INPUT.Record, MSRQ); State-Receive(m); }
   If (INPUT.Type ≡ "TAR") { m--; Store(INPUT.Record, TART);  If (m > 0) State-Receive(m) Else State-BuildMDPQ(); }
}
State-BuildMDPQ() {
   STATE = BUILDMDPQ
   Build (MDPQ, m);  // build the message delivery packet queue
   If (m > 0) { do { MDP = Dequeue( MDPQ); Send-Output ( "MDP", MDP); } while (|MDPQ|>0); State-Receive (m); }
   Else { State-FindTAG( ); }
}
```

Fig. 9.37. Sync manager simulation module.

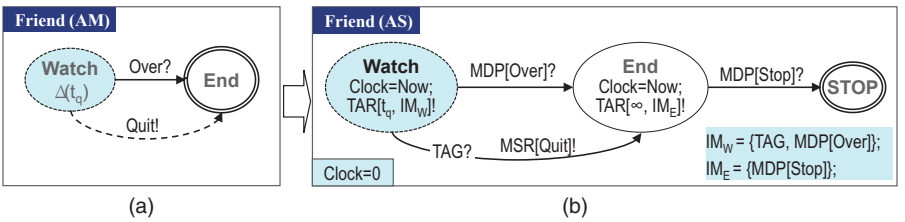


Fig. 9.38. Conversion of (a) an atomic state graph model to (b) an atomic simulator.

Get-Input (INPUT), where INPUT = (Type, Record); // receive data; and
Send-Output (OUTPUT), where OUTPUT = ("Data", Data) and Data = TAR, TAG, MSR, or MDP; // send data.

9.6.3 Atomic Simulators

Figure 9.38(a) and (b) show the Friend atomic model and its atomic simulator, respectively, from the table tennis game state graph in Fig. 9.20. The atomic model consists of an initial state (Watch) and a final state (End), and the state transition occurs either (1) when the input event Over occurs or (2) when the timer $[\Delta(t_q)]$ goes off at time t_q. The former transition is called an *external transition* and the latter an *internal transition* (with a transition action Quit!).

TABLE 9.9. State Transition Table of Atomic Simulator Friend in Figure 9.38(b)

State			Input		Transition		Next
Name	Action	Timer	Event	Action	Condition	Action	State
Watch	Clock = Now; TAR[t_q, {TAG, MDP[Over]}]!	—	TAG?	MSR [Quit]!	True	—	End
			MDP [Over]?	—	True	—	End
End	Clock = Now; TAR[∞, {MDP[Stop]}]!	—	MDP [Stop]?	—	True	—	STOP

State variables: Clock = 0;

In this chapter, a state node with a timer is called a *timed state* and one without a timer is called a *timeless state*. In order to simulate the behavior of the Friend model in Fig. 9.38(a) using the sync manager of Fig. 9.36, the atomic simulator in Fig. 9.38(b) is constructed with the following transformations: (1) the timer [$\Delta(t_q)$] in the timed state Watch is replaced with two entry actions Clock = Now; TAR[t_q, {TAG, MDP[Over]}]!; (2) the internal transition edge with the output message Quit is replaced with an external transition edge with an input message TAG and an output message MSR[Quit]; (3) the input Over at the external transition edge is replaced with an input MDP[Over]; (4) the final state End is replaced with a nonfinal state with two entry actions Clock = Now; TAR[∞, {MDP[Stop]}]!; (5) a final state STOP is added; and (6) an external transition edge with an input MDP[Stop] is added from End to STOP.

Table 9.9 shows the state transition table of the atomic simulator Friend shown in Fig. 9.38(b). The Friend module consists of a main program Friend () and three state transition routines: State-Watch (), State-End (), and State-STOP (). A pseudocode program listing of the Friend atomic simulator module is given in Fig. 9.39. Communications between the main program and the state transition routines are made using two I/O functions: Get-Input (INPUT), where INPUT = (Type, Record) and Send-Output (OUTPUT), where OUTPUT = ("Data", Data) and Data = TAR, TAG, MSR, or MDP.

In general, the rules for converting an atomic state graph model with final states to an atomic simulator are as follows:

1. A new final state STOP is added.
2. Every final state except the STOP state in the atomic model is converted to a nonfinal state, and an external transition edge with an input MDP[Stop]? is defined from each converted nonfinal state to the STOP state.
3. An entry action Clock = Now is added to all states except the STOP state.

```
Friend (t_q) {  // Atomic simulator module for the Friend in the Ping-Pong game (t_q = time to quit)
   Clock = 0; Now = 0; // Clock is the local clock in Friend; Now is the global simulation clock
   State-Watch (Now);                                                    // start with "start state"
}
State-Watch (Now) {  // state-transition routine for start-state "Watch"
   STATE = WATCH;                                                        // state update
   Clock = Now; TAR = (Clock+ t_q, {TAG, MDP[Over]}, Friend); Send-Output ("TAR", TAR);   // state entry actions
   Repeat { Get-Input (INPUT); } Until ((INPUT.Type == "TAG") || (INPUT.Type == "MDP"));  // input event
   If (INPUT.Type == "TAG") { TAG = INPUT.Record;                        // transition action and move to the END state
      MSR= (Quit, Friend); Send-Output ("MSR", MSR); State-End (TAG.Time); }
   If (INPUT.Type == "MDP") { MDP = INPUT.Record;                        // transition action and move to the END state
      If (MDP.Msg == Over) { State-End (MDP.Time); } Else { Stop (Error); } }
}
State-End (Now)  { // state-transition routine for the "End" state
   STATE = END;                                                          // state update
   Clock= Now; TAR = (∞, {MDP[Stop]}, Friend); Send-Output ("TAR", TAR);  // state entry actions
   Repeat { Get-Input (INPUT); } Until (INPUT.Type == "MDP");            // input event
   If (INPUT.Type == "MDP") { MDP = INPUT.Record;
      If (MDP.Msg == Stop) { State-STOP (MDP.Time); } Else { Stop (Error); } }
}
State-STOP (Now) {
   STATE = STOP;  Stop ();
}
```

Fig. 9.39. Atomic simulator module for the Friend in the table tennis game.

4. The timer $[\Delta(t_o)]$ in a timed state is replaced with an entry action TAR$[t_o$, IMS]! where IMS denotes an input message set of the state.
5. Each internal transition edge is converted to an external transition edge with an input TAG?.
6. An entry action TAR $[\infty$, IMS]! is added to each timeless state.
7. Each output Out! is replaced with MSR[Out]! and each input In? is replaced with MDP[In]?.

Reproduced in Fig. 9.40(a) is the Player-A atomic state graph model in Fig. 9.20, and Fig. 9.40(b) shows its atomic simulator obtained by applying the conversion rules. We can verify that all conversion rules are reflected in the atomic simulator: (1) a new final state STOP is added; (2) the final states Quit and Gameover in the atomic model have become nonfinal states, and there is an external transition edge with an input MDP [Stop] from Quit to STOP and another one from Gameover to STOP; (3) all nonfinal states have an entry action Clock = Now; (4) the timer $\Delta(a_A)$ in the Attack state is replaced with an entry action TAR$[a_A$, {TAG, MDP[Quit]}]! and $\Delta(w_A)$ in Wait is replaced with TAR$[w_A$, {TAG, MDP[Quit]}]!; (5) each of the five internal transition edges (Attack→Defense, Attack→Wait, Wait→Attack, Wait→Defense, and Wait→Gameover) are converted to an external transition edge with an input TAG?; and so on.

Table 9.10 shows the state transition table of the atomic simulator Player-A given in Fig. 9.38(b). The initial values of the state variables (Clock = 0, MyScr = 0, UrScr = 0, Rally = 0, Srv = 0, Rcv = 2), system parameters ($w_A = 4.0$, $a_A = 0.8$, $P_A = 0.9$), transition conditions (Ca, Cd, Cg), and algebraic functions (IncS, IncR) are described at the bottom entry of the table. The system parameter w_A denotes the waiting time delay, a_A is the attack time delay, and P_A is

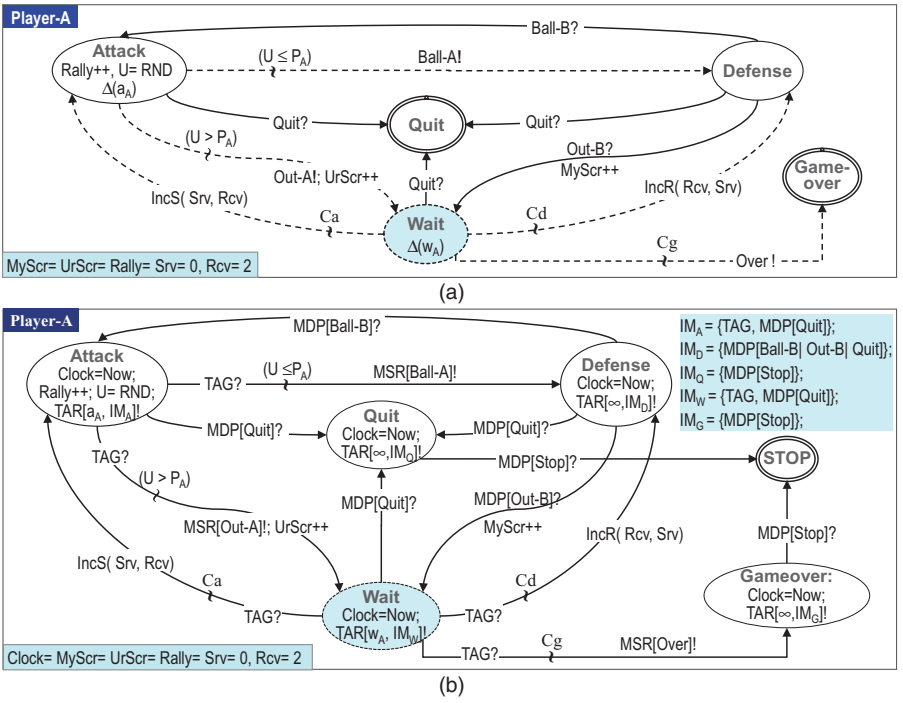


Fig. 9.40. The (a) atomic model and (b) its atomic simulator for Player-A.

the probability of an attack success. RND denotes a uniform random variate generating function.

As in the case of the Friend simulator module (see Table 9.9 and Fig. 9.39), an atomic simulator module for Player-A is obtained easily from the state transition table given in Table 9.10. As before, it is assumed that communication between the main program and the state transition routines are made through the I/O functions Get-Input (INPUT) and Send-Output (OUTPUT). The atomic simulator module for Player-A consists of a main program Player-A () and its state transition routines. A pseudocode program listing of the atomic simulator module for Player-A is given in Fig. 9.41(a) and (b).

9.6.4 Table Tennis Game Simulator

Reproduced in Fig. 9.42(a) and (b), respectively, are the composite state graph model (Fig. 9.21) and the state graph simulator (in Fig. 9.34) of the table tennis game.

Thus far, we have written the Sync Manager Simulation Module listed in Fig. 9.37, the Friend Atomic Simulator listed in Fig. 9.39, and the Player-A Atomic Simulator listed in Fig. 9.41 in pseudocode form. The atomic simulator

TABLE 9.10. State Transition Table of Atomic Simulator Player-A from Fig. 9.40(b)

State			Input		Transition		
Name	Action	Timer	Event	Action	Condition	Action	Next State
Wait	Clock = Now; TAR[w_A, {TAG, MDP[Quit]}]!	—	TAG?	—	Ca	IncS(Srv, Rcv)	Attack
					Cd	IncR(Rcv, Srv)	Defense
					Cg	MSR[Over]!	Gameover
			MDP[Quit]?	—	True	—	Quit
Attack	Clock = Now; Rally++; U = RND TAR[a_A, {TAG, MDP[Quit]}]!	—	TAG?	—	U ≤ P_A	MSR[Ball-A]!	Defense
					U > P_A	MSR[Out-A]!; UrScr++	Wait
			MDP[Quit]?	—	True	—	Quit
Defense	Clock = Now; TAR[∞, {MDP[Ball-B], MDP[Out-B], MDP[Quit]}]!	—	MDP[Ball-B]?	—	True	—	Attack
			MDP[Out-B]?	—	True	MyScr++	Wait
			MDP[Quit]?	—	True	—	Quit
Gameover	Clock = Now; TAR[∞, {MDP[Stop]}]!	—	MDP[Stop]?	—	True	—	STOP
Quit	Clock = Now; TAR[∞, {MDP[Stop]}]!	—	MDP[Stop]?	—	True	—	STOP

State variables: Clock = 0, MyScr = 0, UrScr = 0, Rally = 0, Srv = 0, Rcv = 2; System parameters: w_A = 4.0, a_A = 0.8, P_A = 0.9
Ca = (MyScr < 11) & (UrScr < 11) & (Rcv ≡ 2); Cd = (MyScr < 11) & (UrScr < 11) & (Srv ≡ 2); Cg = (MyScr ≡ 11) || (UrScr ≡ 11)
IncS(Srv, Rcv): {Srv+ = 1; if (Srv≡2) Rcv = 0;} IncR(Rcv, Srv): Rcv+ = 1 ; if (Rcv≡2) Srv = 0;}; U = RND //uniform random variable

```
Player-A (w_A, a_A, P_A) { // Atomic simulator module for Player-A
   // a_A= attack time-delay of Player-A; w_A = wait time-delay; P_A = prob. of an attack-success of Player-A
   Clock= 0; Now = 0; MyScr = UrScr = Rally = Srv =0; Rcv= 2;      // initialize state variables
   State-Wait (Now);                                               // start with "start state"
}
State-Wait (Now) {  // state-transition routine for the start-state "Wait"
   STATE = WAIT;                                                                          // state update
   Clock = Now; TAR = (Clock + w_A, {TAG, MDP[Quit]}, Player-A); Send-Output("TAR", TAR);  // state entry actions
   Repeat { Get-Input (INPUT); } Until ((INPUT.Type ≡ "TAG") || (INPUT.Type ≡ "MDP"));    // input event
   If (INPUT.Type ≡ "TAG") { TAG = INPUT.Record;
      If (Ca) { IncS (Srv, Rcv); State-Attack(TAG.Time); }
      If (Cd) { IncR (Rcv, Srv); State-Defense(TAG.Time); }
      If (Cg) { MSR= (Over, Player-A); Send-Output("MSR", MSR); State-Gameover(TAG.Time); } }
   If (INPUT.Type ≡ "MDP") { MDP = INPUT.Record;
      If (MDP.Msg ≡ Quit) { State-Quit (MDP.Time); } Else "error" }
}

State-Attack(Now) { // state-transition routine for the "Attack" state
  STATE = ATTACK;  Clock= Now; Rally++; U = RND; TAR = (Clock+ a_A, {TAG, MDP[Quit]}, Player-A); Send-Output("TAR", TAR);
   Repeat { Get-Input (INPUT); } Until ((INPUT.Type ≡ "TAG") || (INPUT.Type ≡ "MDP"));
   If (INPUT.Type ≡ "TAG") { TAG = INPUT.Record;
      If (U ≤ P_A) { MSR= (Ball-A, Player-A); Send-Output("MSR", MSR); State-Defense(TAG.Time); }
      If (U > P_A) { MSR= (Out-A, Player-A); Send-Output("MSR", MSR); UrScr++; State-Wait(TAG.Time); } }
   If (INPUT.Type ≡ "MDP") { MDP = INPUT.Record; If (MDP.Msg ≡ Quit) { State-Quit(MDP.Time); } }
}
State-Defense(Now) {
  STATE = DEFENSE;  Clock= Now; TAR = (∞, {MDP[Ball-B], MDP[Out-B], MDP[Quit]}, Player-A); Send-Output("TAR", TAR);
  Repeat { Get-Input (INPUT); } Until (INPUT.Type ≡ "MDP");
  MDP = INPUT.Record;
  If (MDP.Msg ≡ Ball-B) { State-Attack(MDP.Time); }
  If (MDP.Msg ≡ Out-B) { MyScr++; State-Wait (MDP.Time); }
  If (MDP.Msg ≡ Quit) { State-Quit (MDP.Time); }
}
State-Gameover(Now) {
  STATE = GAMEOVER;  Clock= Now; TAR = (∞, {MDP[Stop]}, Player-A); Send-Output("TAR", TAR);
  Repeat { Get-Input (INPUT); } Until (INPUT.Type ≡ "MDP");
  MDP = INPUT.Record;
  If (MDP.Msg ≡ Stop) { State-STOP (MDP.Time); }
}
State-Quit (Now) {
  STATE = QUIT;  Clock= Now; TAR = (∞, {MDP[Stop]}, Player-A); Send-Output("TAR", TAR);
  Repeat { Get-Input (INPUT); } Until (INPUT.Type ≡ "MDP");
  MDP = INPUT.Record;
  If (MDP.Msg ≡ Stop) { State-STOP (MDP.Time); }
}
State-STOP (Now) {
  STATE = STOP;  Stop ();
}
```

Fig. 9.41. Atomic simulator module for Player-A (Fig. 9.40(b)).

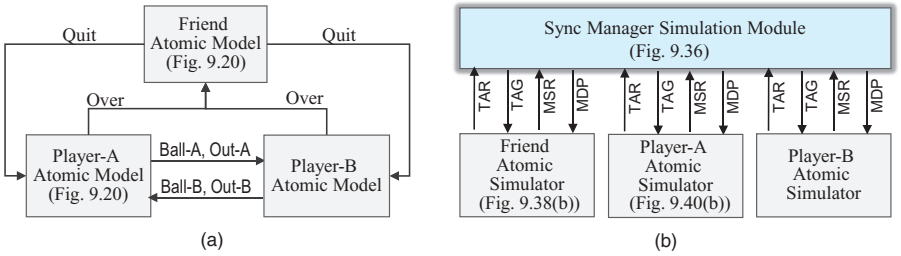


Fig. 9.42. (a) Composite state graph model and (b) state graph simulator of the table tennis game.

module for Player-B may be written in the same way. Then, the sync manager simulation module [Sync-Manager (N_S)] is joined with the atomic simulator modules [Player-A (w_A, a_A, P_A), Player-B (w_B, a_B, P_B), and Friend (t_q)] to build a table tennis game simulator, as follows. (A complete list of C# codes for the table tennis game simulator may be found in the official website of this book: http://VMS-technology.com/Book/StateGraphSimulator)

```
Program Table Tennis-Game ()
{
N_S = 3; // number of atomic simulators
a_A = a_B = 0.8; // attack-time delays of Player-A and
  Player-B are 0.8 seconds.
w_A = w_B = 4; // wait-time delays of both players are 4
  seconds.
P_A = P_B = 0.9; // probability of an attack success for both
  players are 0.9.
t_q = 600; // quit time (the friend can wait for 600 seconds)
Sync-Manager (N_S); // Fig. 9.37
Player-A (w_A, a_A, P_A); // Fig. 9.41
Player-B (w_B, a_B, P_B);
Friend (t_q); // Fig. 9.39
};
```

9.6.5 State Graph Simulator for Reactive Systems

A reactive system that continuously reacts to inputs from the environment by generating corresponding outputs does not have explicit final states. The single server system introduced in Fig. 9.22(b) is an example of a reactive system. Reproduced in Fig. 9.43 is the composite state graph model of the single server system consisting of three atomic state graph models. The interactions among the three atomic models are made via three types of messages: arrive (a) messages from GEN to Buffer, request (r) messages from Machine to Buffer, and send (s) messages from Buffer to Machine.

The single server system of Fig. 9.43 will continue to run until its operation is terminated externally. Figure 9.44 shows a composite state graph model of the single server system augmented with a terminator model whose role is to send an end message (e) to the remaining atomic models in the system. For this purpose, the terminator atomic model is provided with a timer $\Delta(t_e)$ that will go off at the end time t_e, and the existing atomic models are provided with a final state END.

Now, the conversion rules introduced in Section 9.6.3 are applied to the composite state graph model of Fig. 9.44. Figure 9.45 shows the resulting state graph simulator for the single server system. The atomic simulator Terminator

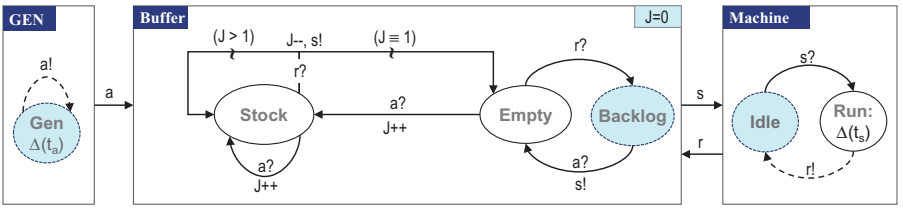


Fig. 9.43. Composite state graph model of a single server system.

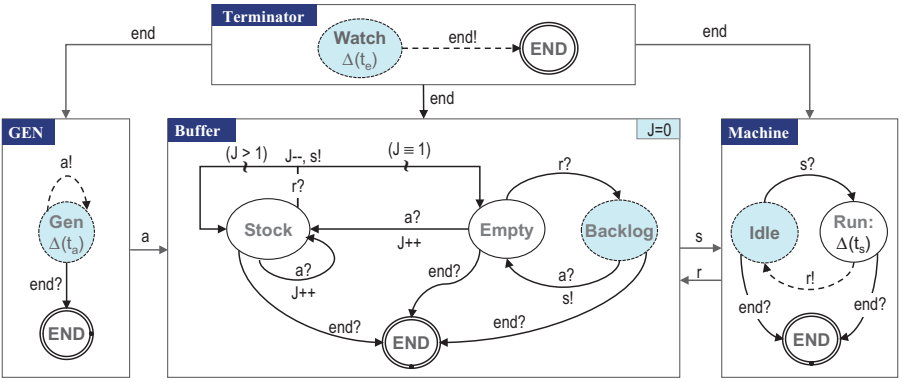


Fig. 9.44. Composite state graph model of the single server system with a terminator.

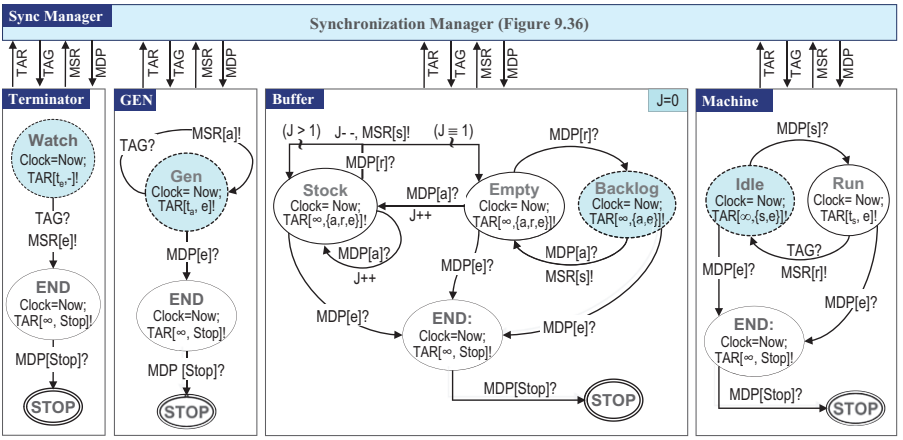


Fig. 9.45. State graph simulator for the single server model in Fig. 9.44.

sends a TAR message to the Sync Manager at the start of the simulation. When the simulation clock reaches the end time (t_e), the following sequence of actions are taken in order to end the simulation: (1) the Sync Manager will send a TAG message to the Terminator; (2) upon receiving the TAG message, the Terminator moves to the END state after sending back an MSR[e] message to the Sync Manager; (3) upon receiving the MSR[e] message, the Sync

Manager sends the MDP[e] message to all atomic simulators except the Terminator; (4) upon receiving the MDP[e] message, all the atomic simulators move to the END state; (5) the atomic simulators send the TAR[∞, Stop] message to the Sync Manager; and (6) the Sync Manager sends the MDP[Stop] message to all atomic simulators to terminate the simulation.

9.6.6 SGS®

A state graph model executor toolkit called SGS was developed by the authors. A free copy of SGS together with a number of modeling examples may be found in the official website of this book (http://VMS-technology.com/Book/SGS).

APPENDIX 9A: DEVS

The concept of DEVS (Discrete EVent system Specification) was proposed by Zeigler and has become the *de facto* choice for state-based M&S tools since his first book on the subject was published [Zeigler 1976]. There has been a considerable amount of progress in the DEVS theory leading to different types of DEVSs [Zeigler et al. 2000]. In this section, the *classic DEVS* will be briefly introduced to demonstrate how the DEVS theory may be used in state-based modeling of discrete-event systems.

9A.1 Definitions of DEVS

As with the definitions of FSM given in Section 9.2.1, DEVS is defined as an algebraic structure. An *atomic DEVS* model (M) is a septuple structure [Zeigler et al. 2000]:

$$M = (S, X, Y, \delta_{int}, \delta_{ext}, \lambda, ta), \tag{9A.1}$$

where

(1) S is a set of states (not necessarily finite)
(2) X is a set of input values
(3) Y is a set of output values
(4) δ_{int}: $S \rightarrow S$ is an internal transition function
(5) δ_{ext}: $Q \times X \rightarrow S$ is an external transition function, where $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ is the total state set, and e is the time elapsed since last transition
(6) λ: $S \times X \rightarrow Y$ is the output function (transition action)
(7) ta: $S \rightarrow R+$ is the time-advance function

If the above definition, which is often referred to as a *classic DEVS*, is compared with that of the engineering definition of the FSM in Section 9.2.1,

it is seen that DEVS is also a timed automaton in which time is included in terms of δ_{int}, Q, and ta.

A DEVS model can also be specified using a state transition table. Reproduced in Table 9A.1 is the template of the state transition table for the state graph model in Table 9.3. Table 9A.1 can be used as a template of a state transition table for a DEVS model if the columns State-Action, Transition-Condition, and Transition-Action are deleted and the state variable row at the bottom is removed.

Reproduced in Fig. 9A.1 is the composite state graph (or coupled DEVS) model of a single server system from Fig. 9.16. Observe in Fig. 9A.1 that the Buffer model (M2) is modeled as an infinite state machine because DEVS does not allow state variables. The Machine model (M3) is specified as an atomic DEVS model, as follows:

$$M3 = (S, X, Y, \delta_{int}, \delta_{ext}, \lambda, ta),$$

where

(1) S = {Idle, Run}
(2) X = {s}
(3) Y = {r, d}
(4) δ_{int}(Run) = Idle
(5) δ_{ext}(Idle, s) = Run
(6) λ(Run, –) = {r, d}; and (7) ta(Run) = t_s; and ta(Idle) = ∞

TABLE 9A.1. Template of State Transition Table for a DEVS Model

State			Input		Transition		
Name	Action	Timer	Event	Action	Condition	Action	Next State
Initial State							
State							
. . .							
Initial values of state variables:							

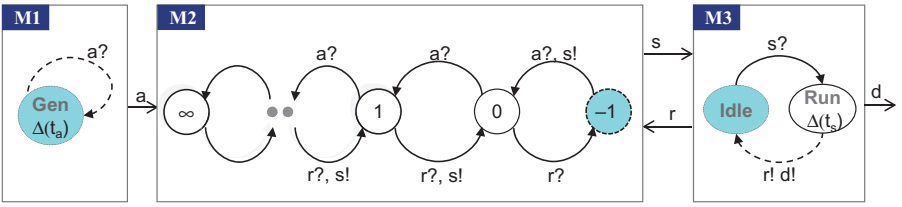


Fig. 9A.1. Composite state graph model of a single server system.

A DEVS model for a system consisting of a number of objects is referred to as *coupled DEVS model*. A classic DEVS coupled model (CM) is a septuple structure [Zeigler et al. 2000]:

$$\text{CM} = (\text{X}, \text{Y}, \{\text{M}\}, \text{EIC}, \text{EOC}, \text{IC}, \text{Select}), \tag{9A.2}$$

where

(1) X is an input events set
(2) Y is an output events set
(3) {M} is a set of all component models
(4) EIC is an external input coupling relation
(5) EOC is an external output coupling relation
(6) IC is an internal coupling relation
(7) Select is a tie-breaking selector

In general, a coupled DEVS model can be structured to have a hierarchical form as depicted in Fig. 9A.2, which is referred to as a *modular hierarchical DEVS*. Shown in Fig. 9A.2(a) is the coupled FSM model of Fig. 9A.1, and Fig. 9A.2(b) is a modular hierarchical DEVS model for the coupled FSM model.

9A.2 DEVS Simulators

The steps for building a hierarchical DEVS simulator are as follows: (1) a simulator is built for each atomic model; (2) a coordinator is built for each coupled model; (3) a root coordinator is constructed; and (4) the simulators and coordinators are connected to form a hierarchical structure called an *abstract simulation model*, as depicted in Fig. 9A.3(a). The abstract simulation model linked with the hierarchical DEVS model is referred to as a *hierarchical DEVS simulator*. Depicted in Fig. 9A.3(b) are the messages passed between a parent coordinator and its children in the abstract simulation model.

In essence, the abstract simulation model consisting of coordinators and simulators functions as the sync manager in the state graph simulator (see Fig.

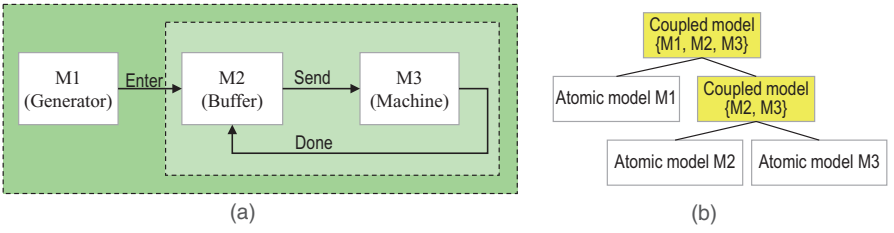


Fig. 9A.2. (a) Coupled FSM model and (b) hierarchical DEVS model.

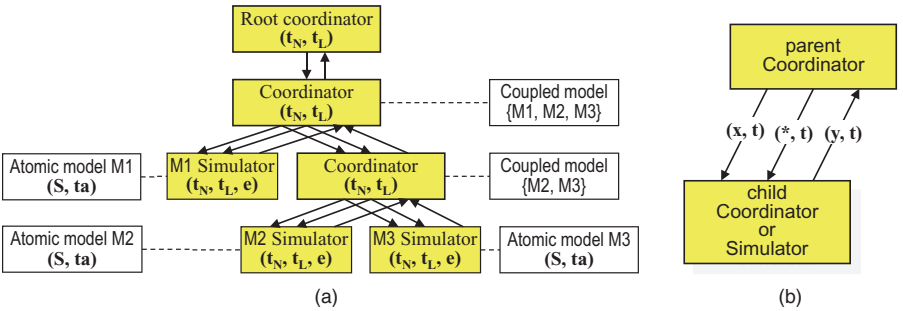


Fig. 9A.3. (a) Hierarchical DEVS simulator and (b) message passing protocol.

9.36) and manages the simulation times. The user defines each atomic model as specified by model (Eq. 9A,1) and each coupled model as specified by model (Eq. 9A.2). For simulation, an atomic model contains information about its state (S) and time advance function (ta); a simulator maintains information about last simulation time (t_L), next simulation time (t_N), and elapsed time (e) of its atomic model; and the coordinator has information about the t_L and t_N of its coupled model. A parent coordinator sends the input event message (x, t) and internal transition message (*, t) to its child coordinator and simulator, and receives the output message (y, t) from its children. For further details, you may refer to seminal text of Zeigler et al. [2000]. Recently, a parallel simulation technique based on the concept of flat coordinator for executing coupled DEVS models has been proposed [Glinsky and Wainer 2006]. Observe that the synchronization manager introduced in Section 9.6 is a kind of flat coordinator.

CHAPTER 10

Advanced Topics in Activity-Based Modeling and Simulation

To everything, there is a season and a time to every purpose under the heavens.

—Ecclesiastes 3:1

10.1 INTRODUCTION

This chapter, combined with Chapter 6, aims to provide a comprehensive treatment of activity-based modeling and simulation (M&S). The activity cycle diagram (ACD) topics covered in Chapter 6 are: (1) the execution rules and specifications; (2) basic modeling templates; (3) representative modeling examples; (4) parameterized ACD and its application to the modeling of tandem lines and job shops; and (5) ACD model execution using the formal ACD simulator ACE®.

The topics to be covered in this chapter are: (1) methods of developing dedicated ACD simulators; (2) the canceling arc and its applications; (3) work-cell cycle time analyses using ACD; (4) ACD modeling of automated manufacturing systems; and (5) formal model conversion. After studying this chapter, you should be able to do the following:

1. Build a dedicated ACD simulator for any parameterized ACD model
2. Develop a general ACD simulation engine
3. Construct ACD models involving canceling arcs for modeling time-constrained processing and resource failure
4. Perform cycle-time analysis for robot work-cells, hoist plating lines, etc.
5. ACD modeling and simulation of flexible manufacturing systems
6. Convert ACD models to event graph models

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.

7. Convert ACD models to state graph or Discrete-EVent system Specification (DEVS) models

The remainder of this chapter is organized as follows: How to develop your own ACD simulators is explained in Section 10.2, and methods of modeling with canceling arcs are given in Section 10.3. Activity-based modeling methods applied to robotic work-cells and flexible manufacturing systems (FMSs) are presented in Sections 10.4 and 10.5, respectively. The last section is devoted to the issue of formal model conversion. A brief description of Petri nets is given in the Appendix.

10.2 DEVELOPING YOUR OWN ACTIVITY CYCLE DIAGRAM SIMULATORS

As mentioned in Chapter 6, there are three ways of executing an ACD model: use of the formal ACD simulator ACE®; use of a process-oriented simulation language such as Arena®; or developing a dedicated ACD simulator. This section describes how to develop a dedicated ACD simulator. If you have a working knowledge of a programming language such as Java or C#, you should be able to develop your own ACD simulator.

10.2.1 Tocher's Three-Phase Process

It is described in Hollocks [2008, p. 5] that the core idea of the Tocher's three-phase process came to him at Christmas 1957, evidently while in his bath! The notion started from the concept of a system consisting of individual components progressing as time unfolds through states that change only at discrete events. The three-phase process is:

Phase A: Advance the clock to the time of the next *bound-to-occur (BTO) event*.

Phase B: Execute the BTO event.

Phase C: Initiate "conditioned" activities that the conditions in the model now permit.

As depicted in Fig. 10.1, the three-phase process is a cyclic process. Tocher argued that the B and C phases represent the event phase and activity phase, respectively. In the following, the three-phase process will be applied to the execution of a single server (system) ACD model.

The ACD model and activity transition table of a single server system given in Chapter 6 are reproduced in Fig. 10.2. In developing an ACD simulator based on the three-phase process, we make use of the data structure *future event list* (FEL) and the two event-handling functions Schedule-event () and Retrieve-event () that were introduced in Chapter 4 (Section 4.7.1). Figure

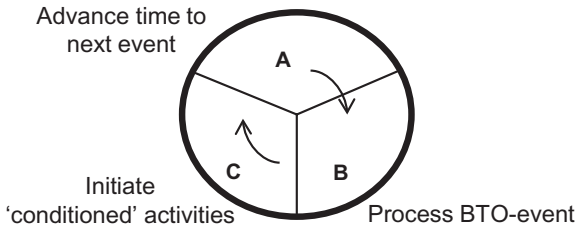
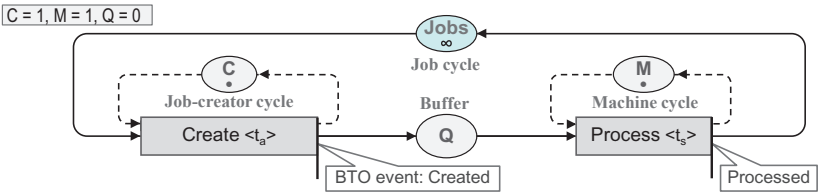


Fig. 10.1. The three-phase process [Hollocks 2008].



No	Activity	At-begin		BTO-event		At-end			
		Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
1	Create	(C>0)	C--;	t_a	Created	1	True	C++;	Create
						2	True	Q++;	Process
2	Process	(M>0) & (Q>0)	M--; Q--;	t_s	Processed	1	True	M++;	Process
Initialize		*Initial Marking* = {C=1, M=1, Q=0}; *Enabled Activities* = {Create}							

Fig. 10.2. ACD model and activity transition table of a single server system.

10.3 shows a three-phase execution program (pseudocode) for the single server ACD model. In this pseudocode program, the three-phase process is implemented with the sequence of *Phase-C*→ *Phase-A*→ *Phase-B*, and the inter-arrival time t_a and service time t_s are set to 8 and 10, respectively.

The Initialization phase of the pseudocode program in Fig. 10.3 performs the following: resets the simulation clock (Clock = 0); initializes the queues (C = 1, M = 1, Q = 0); and sets the end-of-simulation time (EOS = 200). Then, the simulation execution is undertaken in three phases: (1) The scanning phase, where the BTO event of each enabled activity is scheduled in the FEL; (2) the timing phase, where the next event is retrieved from the FEL, and the simulation clock is advanced; and (3) the executing phase, where the retrieved event is executed.

It may be instructional to go through the program statements one by one in Fig. 10.3 to verify that the simulation is carried out correctly. The Scanning phase starts with Clock = 0, C = 1, M = 1, and Q = 0. Thus, the At-begin condition of the Create activity (C > 0) is true, which in turn leads to the execution of {C—; Schedule-event (Created, 8)}. Now, we have the next event Created(8) stored in the future event list (FEL) and C = 0. The Timing phase retrieves the

```
// Initialization
   Clock = 0;                  // Set simulation clock to zero
   C = 1; M = 1; Q = 0;        // Initialize queues
   EOS = 200;                  // Set the end-of-simulation (EOS) time to 200
// Simulation Execution (while the simulation clock is less than the EOS time)
   Do {
   // 1. Scanning the activities (Phase C) → schedule BTO-events into FEL
         If (C > 0) { C--; Schedule-event (Created, Clock + 8)};
         If (M > 0) & (Q > 0) {M--; Q--; Schedule-event (Processed, Clock + 10)};
   // 2. Timing (Phase A) ← retrieve the next BTO-event from FEL
         Retrieve-event (EVENT, TIME);  // Retrieve 1st event from
         Clock = TIME;                  // Advance simulation clock
   // 3. Executing the retrieved event routine (Phase B)
         Case EVENT of {
               Created: If (True) C++; If (True) Q++;
               Processed:  If (True) M++;
         }
    } while (Clock < EOS);
```

Fig. 10.3. Three-phase execution program of the single server ACD model.

Created(8) event from the FEL and advances the simulation clock to 8. The Executing phase executes the Created event to execute {C++, Q++}. Then, the next cycle of the three-phase process begins with Clock = 8, C = 1, M = 1, and Q = 1, and so on. The results of executing the three-phase execution program for a few cycles are summarized in Table 10.1.

10.2.2 Activity Scanning Algorithm

The logic behind the three-phase execution program shown in Fig. 10.3 can be expressed as a general algorithm. The Scanning phase may be expressed as:

```
For each Activity in the activity transition table
(see Fig. 10.2) do {
If (At-begin Condition for the Activity is True)
then {
 (a) Execute At-begin Action; (b) Schedule the BTO-
event into FEL; }
}
```

The Timing phase may be described as:

```
Retrieve the next Event from FEL and advance Time;
```

The Executing phase may be described as:

```
If (At-end Condition for the retrieved Event is True)
  then {
Execute At-end Action;
}
```

TABLE 10.1. Results of Executing the Three-Phase Execution Program

		Current				Newly scheduled event	Updated				
Phase	Clock	"True" condition	Enabled activity	Selected event	Action		Clock	C	M	Q	Events in FEL
0	0	—	—	—	—	—	0	1	1	0	
1	0	(C > 0)	Create	—	C—	Created (8)	0	0	1	0	Created (8)
2	0	—	—	Created (8)	Clock = 8	—	8	0	1	0	—
3	8	—	—	Created	C++; Q++	—	8	1	1	1	
1	8	(C > 0)	Create	—	C—	Created (16)	8	0	1	1	Created (16)
		(M > 0)&(Q > 0)	Process	—	M—; Q—	Processed (18)	8	0	0	0	Created (16), Processed (18)
2	8	—	—	Created (16)	Clock = 16	—	16	0	0	0	Processed (18)
3	16	—	—	Created	C++; Q++	—	16	1	0	1	Processed (18)
1	16	(C > 0)	Create	—	C—	Created (24)	16	0	0	1	Processed (18), Created (24)
2	16	—	—	Processed (18)	Clock = 18	—	18	0	0	1	Created (24)
3	18	—	—	Processed	M++	—	18	0	1	1	Created (24)

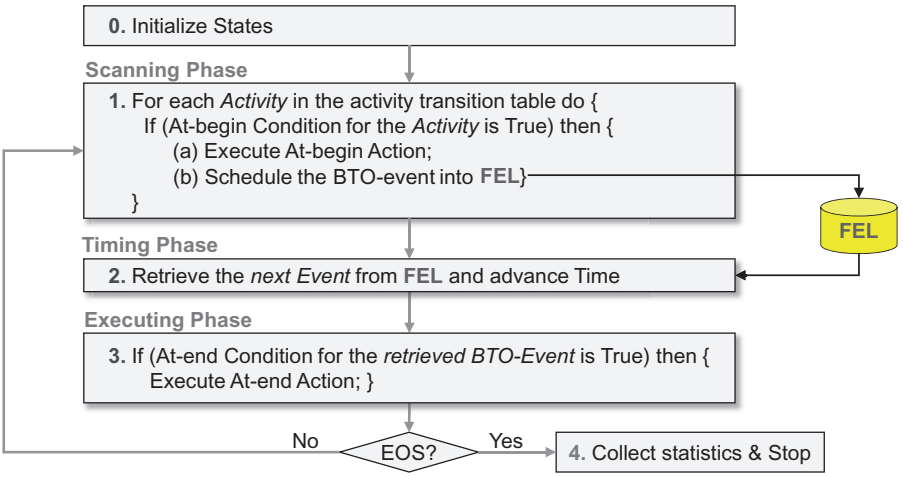


Fig. 10.4. Schematic description of the three-phase execution program.

Figure 10.4 shows a schematic description of the Tocher's three-phase process.

The schematic description given in Fig. 10.4 is a valid algorithm for executing an ACD model. However, it is not an efficient algorithm when the number of activities in the ACD model becomes large because "each and every activity in the model is scanned during the Scanning phase while the number of enabled activities is very small." When a current activity is executed at the Executing phase, the enabled activities at the Scanning phase in the next cycle are among the influenced activities of the current activity. Thus, we introduce a FIFO (first-in-first-out) queue called CAL (candidate activity list) for storing the influenced activities, and modify the execution algorithm in Fig. 10.4. A modified version of the three-phase execution algorithm called the *activity scanning algorithm* is given in Fig. 10.5.

Another, perhaps more critical, benefit of introducing CAL is that it allows to handle tie-breaking among the concurrent activities. For example, in the simple service station model described in Chapter 6 (see Table 6.15 in Section 6.6.4), the Trigger activity has to be executed before the Process activity in order to obtain a valid result. This is ensured by listing Trigger before Process in the Influenced Activity entry in Table 6.15.

10.2.3 ACD Simulator

If we apply the activity scanning algorithm, the three-phase execution program given in Fig. 10.3 would become the ACD simulator shown in Fig. 10.6. It is a template that can be used for any ACD model if the three shaded regions in Fig. 10.6 are modified according to the activity transition table: (1) the

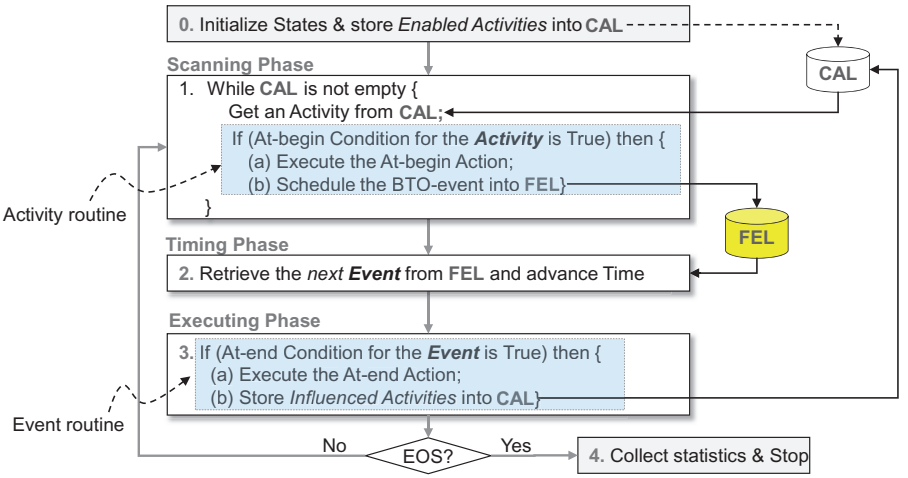


Fig. 10.5. Activity scanning algorithm.

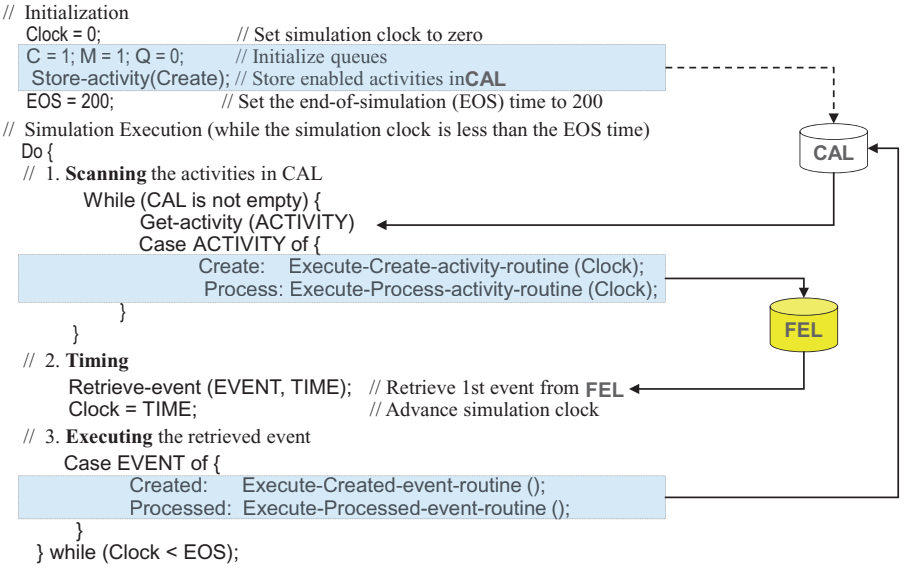


Fig. 10.6. ACD simulator for the single server system.

initialization region reflecting the Initialize row of the activity transition table; (2) the activity routine region; and (3) the event routine region.

At the Initialization phase, the initially enabled activity Create is stored in CAL by invoking Store-activity (Create), and then it is retrieved at the Scanning phase via Get-activity(). The retrieved activity Create is executed at the activity routine region by invoking Execute-Create-activity-routine () to

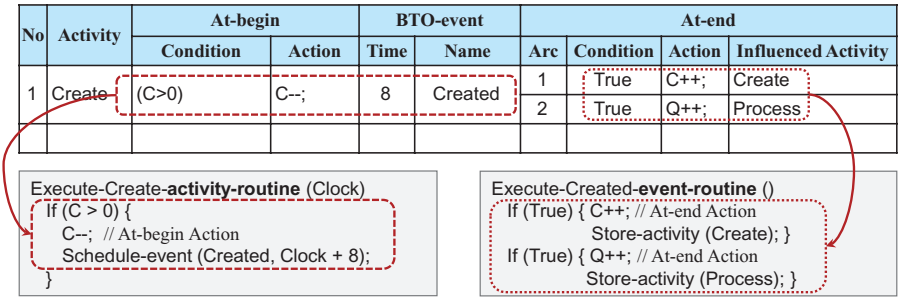


Fig. 10.7. Building activity routine and event routine from the activity transition table.

schedule the BTO-event Created into the FEL. At the Timing phase, the BTO-event is retrieved from FEL and the simulation clock is advanced. Lastly, at the Executing phase, the retrieved event Created is executed by invoking Execute-Created-event-routine () to store the newly enabled activities Create and Process into CAL. The cycle is repeated until the end-of-simulation.

Figure 10.7 shows how to build the Create activity routine and Created event routine from the information provided in the activity transition table. In general, an activity routine is built from the data (At-begin Condition, At-begin Action, Time, and Event Name) of the activity transition table as follows:

```
If (At-begin Condition) {
    At-begin Action;
    Schedule-event (Event Name, Clock + Time); }
```

Similarly, an event routine is built from the data (At-end Condition, At-end Action, and Influenced Activity) of the activity transition table as follows:

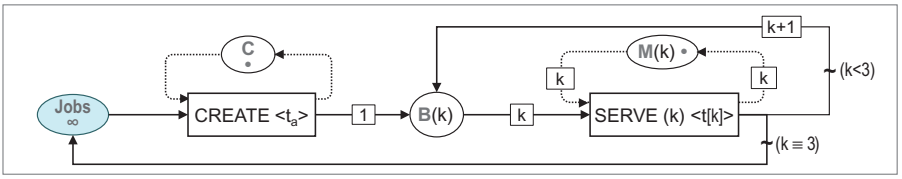
```
If (At-end Condition) {
    At-end Action;
    Store-activity (Influenced Activity); }
```

The above statement is repeated for each At-end arc.

Exercise 10.1. Write an ACD simulator program for the car repair shop model specified in Table 6.8 of Chapter 6.

10.2.4 P-ACD Simulator

An ACD simulator for a parameterized ACD (P-ACD) model, which we call a *P-ACD simulator*, can be built exactly the same procedure as that used for building an ordinary ACD simulator. In this section, how to build a P-ACD simulator will be explained by employing a simple P-ACD model.



No	Activity	At-begin		BTO-event		At-end				
		Condition	Action	Time	Name	Arc	Condition	Parameter	Action	Infl. Act.
1	CREATE	(C>0)	C--;	t_a	CREATED	1	True	-	C++;	CREATE
						2	True	1	B[k]++;	SERVE(k)
2	SERVE(k)	(B[k]>0) & (M[k]>0)	B[k]--; M[k]--;	t[k]	SERVED	1	True	k	M[k]++;	SERVE(k)
						2	(k < 3)	k+1	B[k]++;	SERVE(k)
						3	(k ≡ 3)	-	-	-
Initialize		*Initial Marking* = {C=1, B[k]=0 for k=1~3, M[k]=1 for k=1~3}; *Enabled Activities* = {CREATE}								

Fig. 10.8. P-ACD model and activity transition table of a three-stage tandem line.

Figure 10.8 shows a P-ACD model and its activity transition table for an N-stage unlimited-buffer tandem line (with N = 3) discussed in Chapter 6 (see Section 6.5.1). Recall that, as a convention, a parameter variable is enclosed by a pair of parentheses and a parameter value is put in a small rectangle on an arc. In order to build a P-ACD simulator, we have to upgrade our list-handling functions to handle parameter k:

(1) Store-activity (ACTIVITY, k) for storing an activity in the FIFO queue CAL;
(2) Get-activity (ACTIVITY, k) for retrieving an activity from the CAL;
(3) Schedule-event (EVENT, TIME, k) for scheduling an event in the priority queue the FEL;
(4) Retrieve-event (EVENT, TIME, k) for retrieving an event from the FEL.

Observe in Fig. 10.8 that the parameterized queue nodes B(k) and M(k) in the P-ACD are represented as arrays B[k] and M[k] in the activity transition table.

Figure 10.9 shows the main program of our P-ACD simulator for the three-stage tandem line specified in Fig. 10.8. The structure of the P-ACD simulator is exactly the same as that of the ACD simulator given in Fig. 10.6. At the Initialization phase, the queues (C, B[k], M[k]) are initialized and the initially enabled activity CREATE is stored into the CAL.

Figure 10.10 shows how to build the SERVE activity routine and the SERVED event routine used in the P-ACD simulator. As in the case of the ordinary ACD simulator, a parameterized activity routine is built from the data (At-begin Condition, At-begin Action, Time, and Event Name) of the activity transition table as follows:

```
Main Program (for executing the P-ACD model in Fig. 10.8)
// Initialization
  Clock = 0;  EOS = 1,000 // set simulation clock to zero and EOS time to 1,000
  C=1, B[k] = 0 for k = 1~3, M[k] = 1 for k = 1~3; // set initial marking
  Store-activity (CREATE, -); // store the enabled activity into CAL
// Simulation
  Do {
    While (CAL is not empty) {
      Get-activity (ACTIVITY, k); // get an activity from CAL
      Case ACTIVITY of {
        CREATE: Execute-CREATE-activity-routine (Clock);
        SERVE: Execute-SERVE-activity-routine (Clock, k);
      }
    } //end of while
    Retrieve-event (EVENT, TIME, k); // get the first event from FEL
    Clock = TIME; // advance simulation clock
    Case EVENT of {
      CREATED: Execute-CREATED-event-routine ();
      SERVED: Execute-SERVED-event-routine (k);
    } // end of Case
  } while (Clock < EOS);
// Statistics
```

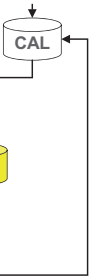


Fig. 10.9. P-ACD simulator for the three-stage tandem line.

No	Activity	At-begin		BTO-event		At-end				
		Condition	Action	Time	Name	Arc	Condition	Parameter	Action	Influenced Activity
2	SERVE(k)	(B[k]>0) & (M[k]>0)	B[k]--; M[k]--;	t[k]	SERVED	1	true	k	M[k]++	SERVE(k)
						2	k < 3	k+1	B[k]++	SERVE(k)
						3	k ≡ 3	-	-	-

```
Execute-SERVE-activity-routine (Clock, k)
  If (B[k]>0 & M[k]>0) {
    B[k]--; M[k]--;  // At-begin Action
    Schedule-event (SERVED, Clock + t[k], k);
  }
```

```
Execute-SERVED-event-routine (k)
  If (True) { M[k]++; // At-end Action
              Store-activity (SERVE, k); }
  If (k < 3) { B[k+1]++; // At-end Action
               Store-activity (SERVE, k+1); }
```

Fig. 10.10. Building activity and event routines in the P-ACD simulator.

```
If (At-begin Condition) {
    At-begin Action;
    Schedule-event (Event Name, Clock + Time, Parameter); }
```

Similarly, a parameterized event routine is built from the data (At-end Condition, At-end Action, Parameter, and Influenced Activity) of the activity transition table as follows:

```
If (At-end Condition) {
    At-end Action;
    Store-activity (Influenced Activity, Parameter); }
```

The above statement is repeated for each At-end arc.
Activity and event routines for the CREATE activity are also listed below:

Execute-CREATE-activity-routine (Clock)

If (C > 0) { // Check the at-begin condition
C−−; // Execute the At-begin action
Schedule-event (CREATED, Clock + t_a, −); } // Schedule the BTO-event

Execute-CREATED-event-routine ()

If (True) { C++; Store-activity (CREATE, −); } // At-end-action & Influenced-activity of Arc-1

If (True) { B[1]++; Store-activity (SERVE, 1); } // At-end-action & Influenced-activity of Arc-2

A complete list of C# codes for the three-stage tandem line ACD simulator may be found in the official website of this book (http://VMS-technology.com/Book/ACDSimulator).

Exercise 10.2. Write a P-ACD simulator program (in pseudocode) for the conveyor-driven serial line model specified in Table 6.11 of Chapter 6.

10.2.5 Collecting Statistics

As discussed in Chapter 4 (see Section 4.7.5), the average queue length (AQL) statistics can be collected as follows. Let $\{T_j\}$ denote the queue length change times, then the j^{th} queue length change interval becomes $\Delta_j = T_{j+1} - T_j$. Let Q_j be the queue size during Δ_j, then we have $SumQ = \Sigma(Q_j \times \Delta_j)$ and $AQL = SumQ / \Sigma(\Delta_j) \equiv SumQ / CLK$.

Figure 10.11 shows additional statements that are added in the main program to collect statistics on the AQL. We want to compute the AQL of each buffer B[k]. The previous queue length change time of B[k] is denoted by Before[k]. At the very beginning of the main program, SumQ[k] and Before[k] are reset. Then, at the start of the statistics phase, SumQ[k] are updated and AQL[k] are computed. As can be seen in Fig. 10.12, the intermediate values of SumQ[k] are collected at the CREATED event routine, SERVE activity routine, and SERVED event routine.

```
Main Program (for collecting Average Queue Length statistics)
// Initialization
  SumQ[k] = 0 for k = 1~3; Before[k] = 0 for k = 1~3; // initialize statistics variables
  Clock = 0;  EOS = 1,000 // set simulation clock to zero and EOS time to 1,000
  ….
// Simulation
  ….

// Statistics
  SumQ[k] += B[k] * (Clock – Before[k] ) for k = 1~3;
  AQL[k] = SumQ[k] / Clock for k = 1~3;
  …. // Print out etc.
```

Fig. 10.11. Statements added at the main program for computing AQL.

```
Execute-CREATED-event-routine ()
   If (True) { C++; Store-activity (CREATE, -)};
   If (True) {SumQ[1] += B[1] * (Clock – Before[1]); Before[1] = Clock;
              B[1]++; Store-activity (SERVE, 1)};
```

```
Execute-SERVE-activity-routine (Clock, k)
  If (B[k]>0 && M[k]>0) {SumQ[k] += B[k] * (Clock – Before[k]); Before[k] = Clock;
                         B[k]--; M[k]--;  Schedule-event (SERVE, Clock + t[k], k) };
```

```
Execute-SERVED-event-routine (k)
   If (True) { M[k]++; Store-activity (SERVE, k)};
   If (k < 3) {SumQ[k+1] += B[k+1] * (Clock – Before[k+1]);  Before[k+1] = Clock;
              B[k+1]++; Store-activity (SERVE, k+1) };
```

Fig. 10.12. Statements for collecting AQL statistics at individual routines.

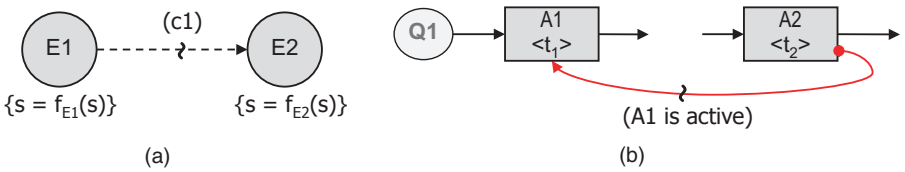


Fig. 10.13. (a) Canceling edge in event graph and (b) canceling arc in ACD.

Collecting sample statistics of individual entities is more involved. In order to collect sojourn time statistics, each entity is provided with a record containing its attribute values such as the arrival time, departure time, and entity type. It is an important issue in designing a simulator but will not be elaborated further in this book.

10.3 MODELING WITH CANCELING ARC

In Chapter 4, it was shown that the use of a canceling edge in an event graph proved to be quite convenient when modeling reneging, resource failure, and time-constrained processing. In this section, we will show that the role of a canceling arc in an ACD model is the same as that of a canceling edge in an event graph model.

Figure 10.13(a) shows an event graph with a canceling edge denoted by the dashed arrow. As described in Chapter 4 (Section 4.3.1), the event graph indicates that "whenever the originating event E1 occurs, the state changes to $f_{E1}(s)$. Then, if the edge condition c1 is true, the scheduled event E2 is canceled immediately." Shown in Fig. 10.13(b) is an ACD with a canceling arc denoted by a circle-tailed arrow. This ACD indicates that "whenever the originating activity A2 is completed, the target activity A1 is canceled if it is active."

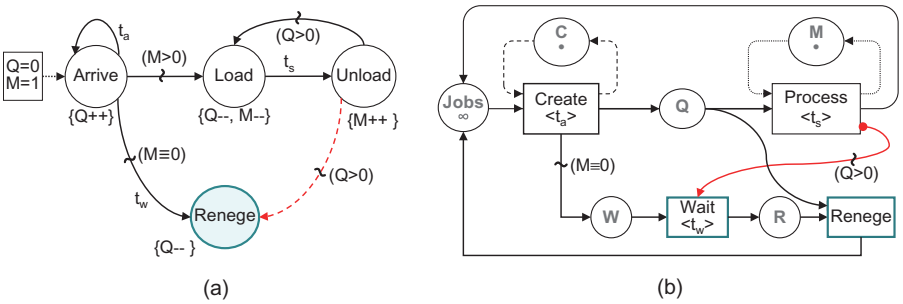


Fig. 10.14. (a) Event graph and (b) ACD of a single server system with reneging.

TABLE 10.2. Activity Transition Table of the Reneging ACD Model in Figure 10.14(b)

		At-begin		BTO-event		At-end			
No	Activity	Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
1	Create	(C > 0)	C—;	t_a	Created	1	true	Q++;	Process
						2	true	C++;	Create
						3	(M≡0)	W++;	Wait
2	Process	(Q > 0) & (M > 0)	Q—; M—;	t_s	Processed	1	true	M++;	Process
						2	(Q > 0)	*cancel*;	Wait
						3	true	—	—
3	Wait	(W > 0)	W—;	t_w	Waited	1	true	R++;	Renege
4	Renege	(R > 0) & (Q > 0)	R—; Q—;	0	Reneged	1	true	—	—
Initialize		*Initial Marking* = {C = 1, M = 1, Q = 0, W = 0, R = 0}; *Enabled Activities* = {Create}							

10.3.1 ACD Model of Single Server System with Reneging

As discussed in Chapter 4 (Section 4.4.1), customers waiting for a service in a line may choose to leave the line if they have waited too long, which is called *reneging* in queuing theory. Figure 10.14 shows an event graph model and an ACD model of a single server system with reneging, where the customer would not wait in line more than t_w minutes.

In the ACD model, Q denotes the number of customers waiting in the queue. Customers are created with an inter-arrival time of t_a and are processed by the server with a service time of t_s. An arriving customer is put into the queue Q, and at the same time, a clone of the customer is put into the queue W if no servers are idle (i.e., M ≡ 0). The clone arrived at the queue W is immediately put into the Wait activity where it is processed for t_w minutes. Observe that Q is equal to the number of clones residing in the Wait activity. At the end of the Process activity, the oldest clone residing in the Wait activity is deleted if there exists any (Q > 0). The ACD model of Fig. 10.14(b) is specified in the activity transition table of Table 10.2.

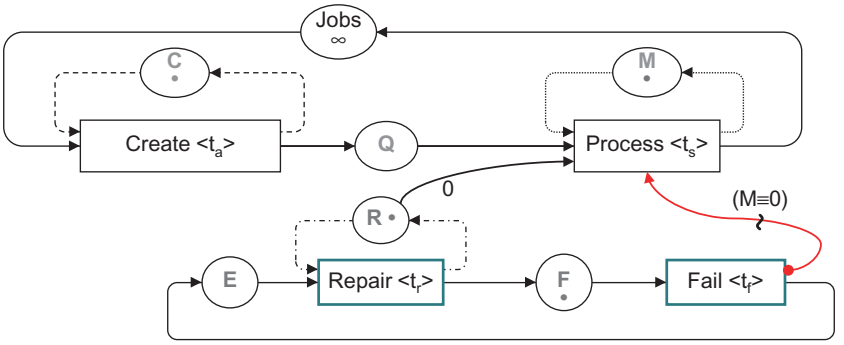


Fig. 10.15. ACD of a single server system with resource failure.

TABLE 10.3. Activity Transition Table of the Resource Failure ACD Model

		At-begin		BTO-event		At-end			
No	Activity	Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
1	Create	(C > 0)	C—;	t_a	Created	1	true	C++;	Create
						2	true	Q++;	Process
2	Process	(Q > 0) & (M > 0) & (R > 0)	Q—; M—;	t_s	Processed	1	true	M++;	Process
						2	true	—	—
3	Repair	(R > 0) & (E > 0)	R—; E—;	t_r	Repaired	1	true	R++;	Repair, Process
						2	true	F++;	Fail
4	Fail	(F > 0)	F—;	t_f	Failed	1	true	E++;	Repair
						2	(M≡0)	M++; cancel;	Process
Initialize		*Initial Marking* = {C = 1, M = 1, Q = 0, R = 1, F = 1, E = 0}; *Enabled Activities* = {Create, Fail}							

10.3.2 ACD Model of Resource Failure

Figure 10.15 shows an ACD model of the single server system with resource failure whose event graph model was presented in Chapter 4 (see Fig. 4.13 in Section 4.4.1). It is a single server system (t_a = inter-arrival time; t_s = service time) augmented with resource failures (t_f = interfailure time) and repairs (t_r = repair time). The completion of a Fail activity will cancel a Process activity (if it is active) in addition to triggering a Repair activity to start. Observe that the arc multiplicity from queue R to activity Process is zero. This model assumes that a resource may fail even when it is idle and that the interrupted job due to failure is discarded without reprocessing. The activity transition table of the failure-repair ACD model is given in Table 10.3. Observe in the table that the resource is set to idle (M++) after canceling an active Process activity.

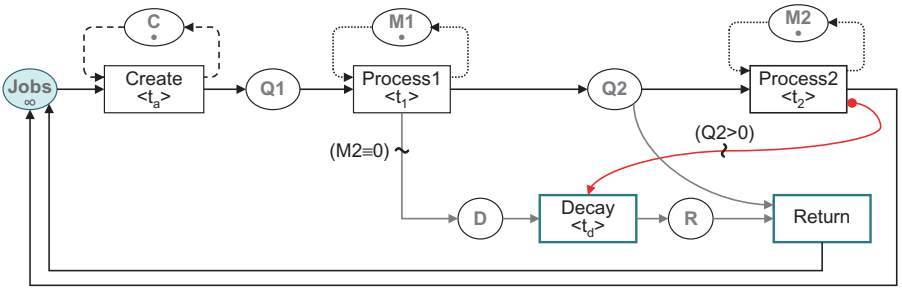


Fig. 10.16. ACD model of two-stage line with time-constrained processing.

10.3.3 ACD Model of Time-Constrained Processing

Figure 10.16 shows an ACD model of a time-constrained processing system whose event graph model was presented in Chapter 4 (see Fig. 4.18 in Section 4.4.2). It is a two-stage tandem line where a job that had been processed on the first machine (M1) must be processed on the second machine (M2) within t_d minutes. Otherwise, the job is discarded.

Referring to Fig. 10.16, a job that has completed its processing at M1 is put into the queue Q2, and at the same time, a clone of the job is put into the queue D if the second machine is busy (i.e., M2 ≡ 0). The clone in the queue D is immediately put into the Decay activity where it is processed for t_d minutes. At the end of the Process2 activity, the oldest clone residing in the Decay activity is deleted if Q2 > 0 (Q2 is equal to the number of clones residing in the Decay activity). The ACD model of Fig. 10.16 is specified in the activity transition table of Table 10.4.

10.3.4 Execution of Canceling Arc

Figure 10.17 shows how to build an event routine for the Fail activity appearing in Table 10.3. The At-end Action Cancel in the activity transition table is implemented with the event-handling function Cancel-event () that was introduced in Chapter 4 (Section 4.7.1). Other activity routines and event routines are implemented the same way as described in Section 10.2.

Exercise 10.3. Write an activity routine and an event routine for the Process2 activity appearing in Table 10.4.

10.4 CYCLE TIME ANALYSIS OF WORK CELLS VIA AN ACTIVITY CYCLE DIAGRAM

Work cells play a key role in lean manufacturing that is widely accepted in industries. The physical configuration of a lean manufacturing system is a linked work-cell system [Black and Hunter 2003]. Popular types of work cells

TABLE 10.4. Activity Transition Table of the ACD Model in Figure 10.16

		At-begin		BTO-event		At-end			
No	Activity	Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
1	Create	(C > 0)	C−−;	t_a	Created	1	true	C++;	Create
						2	true	Q1++;	Process1
2	Process1	(Q1 > 0) & (M1 > 0)	Q1−−; M1−−;	t_1	Processed1	1	true	M1++;	Process1
						2	true	Q2++;	Process2
						3	(M2≡0)	D++;	Decay
3	Process2	(Q2 > 0) & (M2 > 0)	Q2−−; M2−−;	t_2	Processed2	1	true	M2++;	Process2
						2	(Q2>0)	Cancel;	Decay
						3	true	—	—
4	Decay	(D > 0)	D−−;	t_d	Decayed	1	true	R++;	Return
5	Return	(R > 0) & (Q2 > 0)	R−−; Q2−−;	0	Returned	1	true	—	—
Initialize		*Initial Marking* = {C = 1, M1 = 1, M2 = 1, Q1 = Q2 = D = R = 0}; *Enabled Activities* = {Create}							

No	Activity	At-begin		BTO-event		At-end			
		Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
4	Fail	(F>0)	F--;	t_f	Failed	1	true	E++;	Repair
						2	(M≡0)	M++; Cancel;	Process
Initialize		*Initial Marking* = {C=1, M=1, Q=0, R=1, F=1, E=0}; *Enabled Activities* = {Create, Fail}							

```
Execute-Failed-event-routine ()
  If (true) {E++; Store-activity (Repair)};
  If (M≡0) {M++; Cancel-event (Processed)};
```

Fig. 10.17. Event routine for the Fail activity in the resource failure model.

include robot work cells in the mechanical industry, hoist plating lines used in the PCB (printed circuit board) industry, and cluster tools in semiconductor industry. A key issue in work-cell design is to estimate or reduce its cycle time. This section shows how ACD-based modeling can be used in the performance analysis of work cells.

10.4.1 Cycle Time Analysis of Single-Armed Robot Work Cell

Figure 10.18 shows a robot work cell [Asfahl 1992] in which a single-armed robot performs loading and unloading operations for three machines. Each machine unloads its finished job before loading a new job. The in-port and out-port are designated as S0 and S4, and the machines as S1, S2, and S3. A job is introduced at the in-port S0, goes through the machines, and is dropped at the out-port S4.

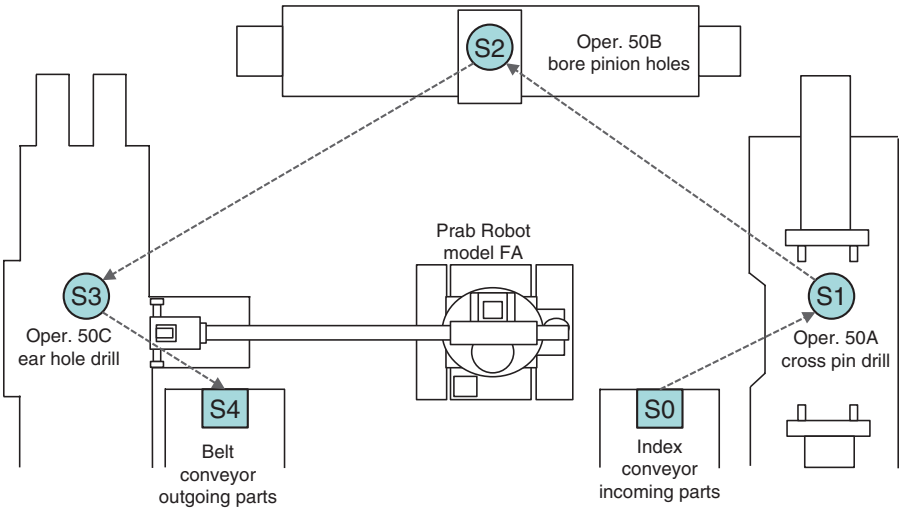


Fig. 10.18. Robot work cell [Asfahl 1992].

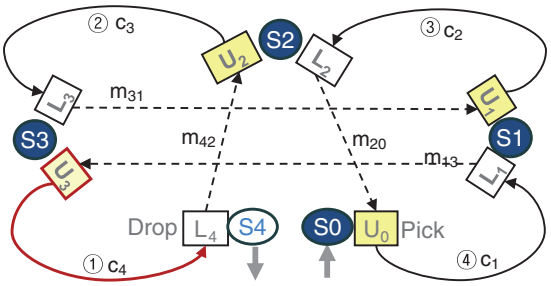


Fig. 10.19. Operation sequence diagram of the robot in a three-stage work cell.

When all machines are full (i.e., loaded with jobs), the robot has to start its activity cycle by unloading (U_3) the finished job from the last machine S3. Then, it carries (c_4) the finished job to the out-port S4 to drop (L_4) the finished job there, moves (m_{42}) to machine S2 to unload (U_2) the job, carries (c_3) the unloaded job to machine S3 to load (L_3) the job, and so on. The activity cycle of the robot is depicted in the operation sequence diagram of Fig. 10.19.

If you follow the *operation sequence* of the robot starting from S0, you will easily construct its ACD. Namely, the r*obot activity cycle* is: U_0(Pick) $\rightarrow c_1 \rightarrow L_1 \rightarrow m_{13} \rightarrow U_3 \rightarrow c_4 \rightarrow L_4$(Drop) $\rightarrow m_{42} \rightarrow U_2 \rightarrow c_3 \rightarrow L_3 \rightarrow m_{31} \rightarrow U_1 \rightarrow c_2 \rightarrow L_2 \rightarrow m_{20}$. On the other hand, each machine is involved in loading (L) a new job, processing (P) the job, and unloading (U) the finished job. For example, S1 ***machine activity cycle*** is: $L_1 \rightarrow P_1 \rightarrow U_1$. By adding the machine activity cycles to the robot activity cycle, we obtain an ACD of the robot work cell as shown in Fig. 10.20. The ***job activity cycle*** starts from Pick, and then it

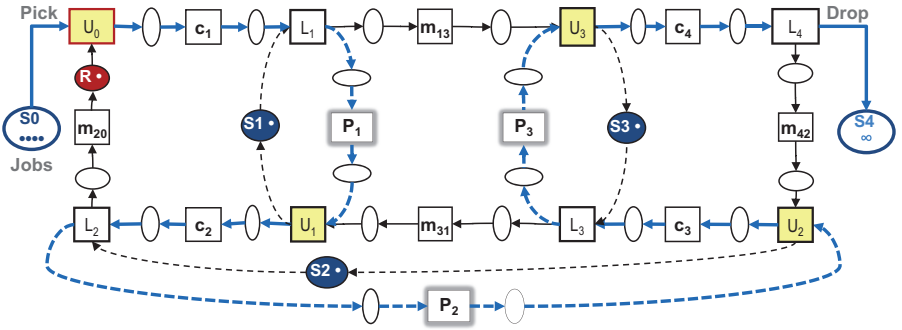


Fig. 10.20. ACD of the robot work cell.

goes to machines S1, S2, S3 and ends at Drop (i.e., $U_0 \rightarrow c_1 \rightarrow L_1 \rightarrow P_1 \rightarrow U_1 \rightarrow c_2 \rightarrow \cdots \rightarrow U_3 \rightarrow c_4 \rightarrow L_4$ as indicated with the sequence of thick arrows in the figure).

There are four active resources in the work cell: robot (R) and three machines (S1, S2, and S3). Let L_k, U_k and P_k denote the loading, unloading and processing times at Sk (for k = 1, 2, 3), respectively, and m_{ij} and c_j denote the move time (without carrying a job) and carry time of the robot from Si to Sj, respectively. Then, the work-cell cycle time (CT) is expressed as:

$$\mathrm{CT(Robot\text{-}cell)} = \max\{\mathrm{CT(R)}, \mathrm{CT(S1)}, \mathrm{CT(S2)}, \mathrm{CT(S3)}\}, \tag{10.1}$$

where

$$\begin{aligned}
\mathrm{CT(R)} &= \{U_0 + U_1 + U_2 + U_3\} + \{L_1 + L_2 + L_3 + L_4\} \\
&\quad + \{c_1 + c_2 + c_3 + c_4\} + \{m_{20} + m_{31} + m_{42} + m_{13}\}; \\
\mathrm{CT(S1)} &= \{U_0 + U_1\} + \{L_1 + L_2\} + \{c_1 + c_2\} + \{m_{20}\} + P_1; \\
\mathrm{CT(S2)} &= \{U_1 + U_2\} + \{L_2 + L_3\} + \{c_2 + c_3\} + \{m_{31}\} + P_2; \\
\mathrm{CT(S3)} &= \{U_2 + U_3\} + \{L_3 + L_4\} + \{c_3 + c_4\} + \{m_{42}\} + P_3,
\end{aligned}$$

where CT(R) is the robot cycle time and CT(Sk) denotes the cycle time of machine Sk.

10.4.2 Cycle Time Analysis of Single Hoist Plating Line

Shown in Fig. 10.21 is a hoist plating line [Chen et al. 1998] in which an automated hoist performs all the loading and unloading operations for a sequence of tanks. Tank_0 (S_0) is designated as the loading station and Tank_{N+1} (S_{N+1}) as the unloading station.

Let's assume that N = 3. Then, when all the operation tanks (S_1–S_3) are loaded with jobs, the hoist has to start its activity cycle by unloading (U_3) the finished job from S_3. Then, it carries (c_4) the finished job to the unloading station S4 to drop (L_4) the finished job there, makes a move m_{42} to tank S2 to

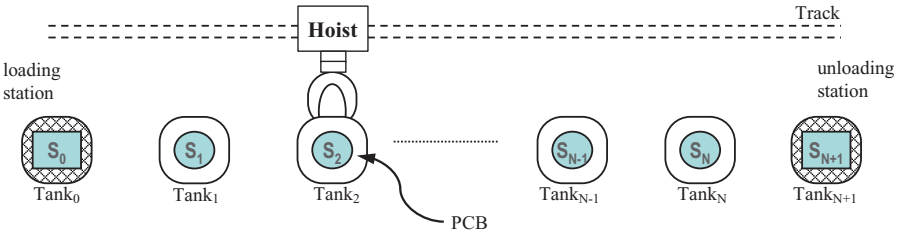


Fig. 10.21. Hoist plating line [Chen et al. 1998].

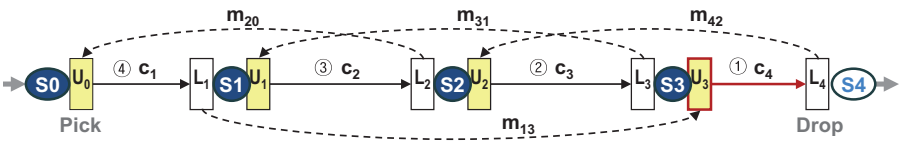


Fig. 10.22. Operation sequence diagram of the hoist in a three-stage line.

unload (U_2) the job, carries (c_3) the job to tank S3 to load (L_3) the job, and so on. The operation sequence diagram is as shown in Fig. 10.22.

In fact, the behavior of the hoist line is exactly the same as that of the robot work cell, and the hoist operation sequence diagram in Fig. 10.22 is identical to the robot operation sequence diagram of Fig. 10.19. Thus, its ACD is the one given in Fig. 10.20. However, a hoist plating line has a time-constraint issue due to the nature of chemical treatments: the time a job spends in a tank is upper and lower bounded, which imposes time constraints on the hoist moves [Chen et al. 1998].

One type of time-constraints is *immediate removal constraints*, requiring that a processed job should be removed from the tank immediately. Namely, the hoist that had loaded (L_k) a job at Sk has to come back to be unloaded (U_k) no later than the actual processing time p_k. Referring back to the ACD in Fig. 10.20, the immediate removal constraints for k = 1 ~ 3 are expressed as:

$$\begin{aligned} m_{13} + U_3 + c_4 + L_4 + m_{42} + U_2 + c_3 + L_3 + m_{31} &\le p_1 \\ m_{20} + U_0 + c_1 + L_1 + m_{13} + U_3 + c_4 + L_4 + m_{42} &\le p_2 \\ m_{31} + U_1 + c_2 + L_2 + m_{20} + U_0 + c_1 + L_1 + m_{13} &\le p_3. \end{aligned} \tag{10.2}$$

In order to reduce the expressional complexity, let's assume that:

1. All the carry times are no less than a fixed value (γ);
2. Each processing time p_k is bounded by its min value (α_k) and max value (β_k);
3. All the load/drop times are equal to a fixed value δ;
4. All the unload/pick times are equal to π;
5. All the move times are equal to μ.

Then, the hoist scheduling problem can be formulated as a minimization problem as follows [Chen et al. 1998]:

$$\text{Minimize}(\max\{CT(\text{Hoist}), CT(S1), CT(S2), CT(S3)\}) \tag{10.3}$$

$$CT(\text{Hoist}) = \{4\pi + 4\delta + 4\mu\} + \{c_1 + c_2 + c_3 + c_4\}$$

$$CT(S1) = \{2\pi + 2\delta + \mu\} + \{c_1 + c_2\} + p_1$$

$$CT(S2) = \{2\pi + 2\delta + \mu\} + \{c_2 + c_3\} + p_2$$

$$CT(S3) = \{2\pi + 2\delta + \mu\} + \{c_3 + c_4\} + p_3$$

Decision variables: p_1, p_2, p_3, c_1, c_2, c_3, c_4
Subject to:

(1) Time window constraints: $\alpha_k \le p_k \le \beta_k$ for k = 1 to 3
(2) Carry time constraints: $\gamma \le c_k$ for k = 1 to 4
(3) Immediate removal constraints:

$$\{2\pi + 2\delta + 3\mu\} + \{c_{3+}c_4\} \le p_1$$

$$\{2\pi + 2\delta + 3\mu\} + \{c_1 + c_4\} \le p_2$$

$$\{2\pi + 2\delta + 3\mu\} + \{c_1 + c_2\} \le p_3$$

If the hoist scheduling problem (Eq. 10.3) does not produce a feasible solution, another scheduling may be generated with a reduced load factor and the resulting hoist scheduling problem is formulated and solved. The load factor of the operation sequence in Fig. 10.22 is 3/3 (or 100%) because three tanks out of three tanks are occupied by the job initially. Figure 10.23 shows an operation sequence diagram with a reduced load factor (=2/3) where the second tank S2 is not occupied initially.

If you follow the *operation sequence* diagram in Fig. 10.23 starting from S1, you will easily construct the ***hoist activity cycle***: $U_1 \rightarrow c_2 \rightarrow L_2 \rightarrow m_{23} \rightarrow U_3 \rightarrow c_4 \rightarrow L_4(\text{Drop}) \rightarrow m_{40} \rightarrow U_0(\text{Pick}) \rightarrow c_1 \rightarrow L_1 \rightarrow m_{12} \rightarrow U_2 \rightarrow c_3 \rightarrow L_3 \rightarrow m_{31}$. As before, each tank (Sk) is involved in loading (L_k), processing (P_k), and unloading (U_k). From these individual activity cycles, an ACD like the one given in Fig. 10.20 may be obtained. Hoist scheduling problems have been widely studied and a comprehensive survey is available in the literature [Manier and Bloch 2003].

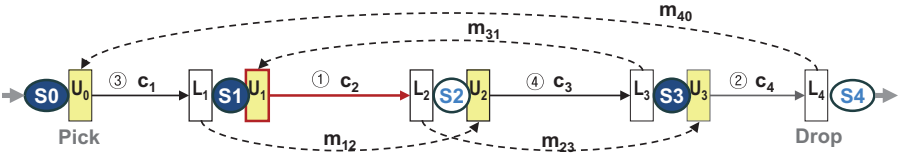


Fig. 10.23. Operation sequence diagram of the hoist with a reduced load factor.

Exercise 10.4. Build an ACD for the operation sequence diagram given in Fig. 10.23.

10.4.3 Cycle Time Analysis of Dual-Armed Robot Cluster Tool

Figure 10.24 shows a cluster tool in which a dual-armed robot performs swapping operations [Kim et al. 2003]. The cluster tool consists of several processing chambers, an aligner and cooler, two load-locks, and a dual-armed robot. A serial wafer flow is depicted in Fig. 10.24(a) and a serial-parallel wafer flow in Fig. 10.24(b). Shown in Fig. 10.24(c) is the physical image of the cluster tool.

In the 3-step 3-chamber serial flow of Fig. 10.24(a), the robot picks up a job at the load-lock A, and it carries the job to chamber C1 to swap the new job with a finished one. Then, it goes to C2 to make another swapping, and so on. It completes its activity cycle by dropping the completed job at the load-lock B. The robot's move (m) from B to A may be neglected as its time duration is negligible.

Figure 10.25(a) shows an operation sequence diagram for the serial flow case where each job is processed in C1, C2, and C3 in series. Figure 10.25(b)

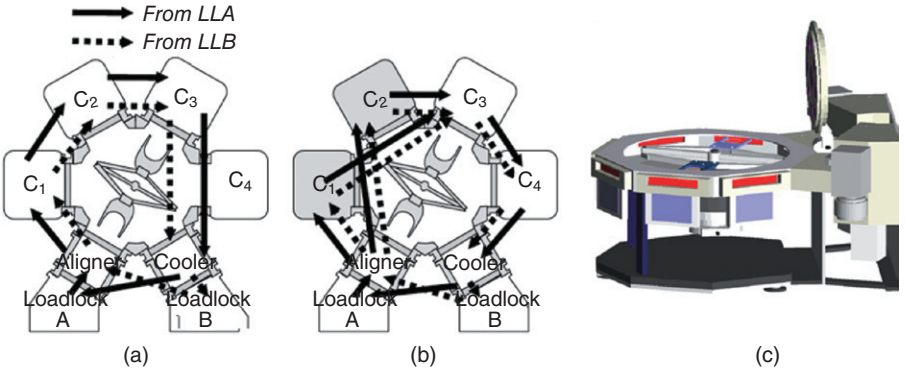


Fig. 10.24. Cluster tool with dual-armed robot [Kim et al. 2003].

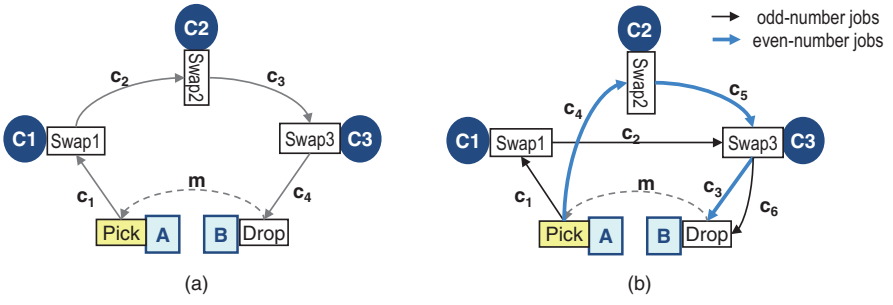


Fig. 10.25. Operation sequence diagram: (a) serial flow, (b) serial/parallel flow.

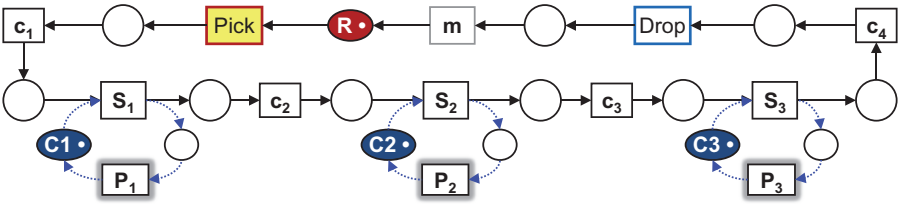


Fig. 10.26. ACD for the serial flow operation of Fig. 10.25(a).

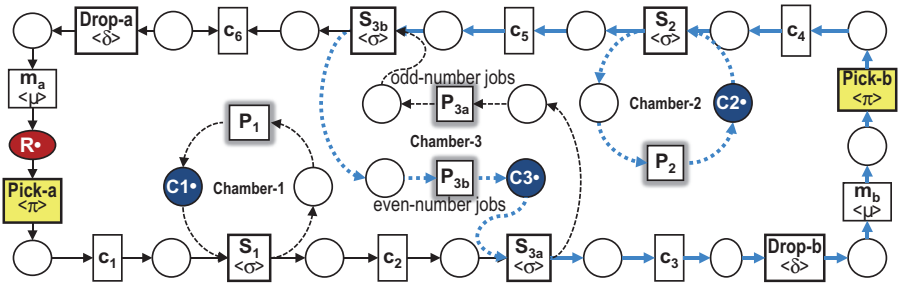


Fig. 10.27. ACD for the serial/parallel flow operation of Fig. 10.25(b).

shows an operation sequence diagram for a 2-step 3-chamber serial-parallel flow case where the first processing step is handled by two chambers (C1 and C2) and the second step is covered by one chamber (C3). The serial-parallel flow is used when the processing time of the first step operation is much larger than that of the second step. The flow of odd-number jobs (Pick-c1-c2-c6-Drop) is denoted by a thin arrow and that of even-number jobs by a thick arrow.

Figures 10.26 and 10.27, respectively, show ACDs for the 3-chamber serial flow of Fig. 10.25(a) and the 3-chamber serial-parallel flow of Fig. 10.25(b). The activity names and durations are: $c_1 - c_6$ = carry; S_k = swap at chamber k; P_k = process at chamber k; σ = swapping time; δ = dropping time; π = picking-up time; μ = moving time. In both ACDs, there are four resource activity cycles. The cycle time of the cluster tool is determined by the maximum of the four activity cycles.

There are four resources in the ACD of Fig. 10.27: Robot (R), chamber-1 (C1), chamber-2 (C2), and chamber-3 (C3). The activity cycle of the robot is the outer loop of the ACD and the activity cycles of C1 and C2 are simple loops as indicated in the ACD of Fig. 10.27. Thus, their cycle times are easily identified as:

$$\mathrm{CT(Robot)} = \{2\pi + 4\sigma + 2\delta + 2\mu\} + \{c_1 + c_2 + c_3 + c_4 + c_5 + c_6\};\ //\ \text{per two job} \tag{10.4}$$

$$\mathrm{CT(C1)} = \sigma + p_1;\ //\ \text{per one job}$$

$$\mathrm{CT(C2)} = \sigma + p_2;\ //\ \text{per one job}$$

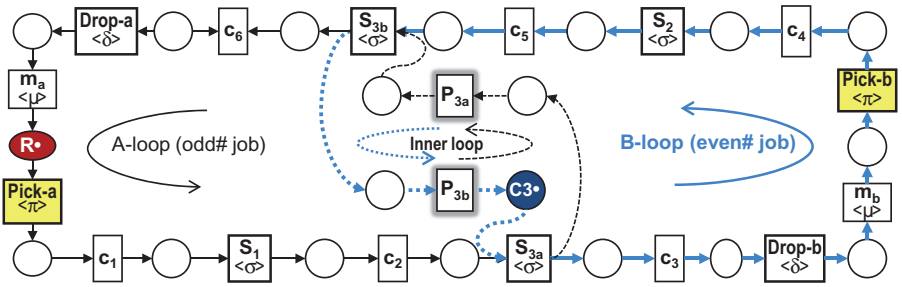


Fig. 10.28. Loops involved in the cycle time of chamber-3 (C3).

However, as depicted in Fig. 10.28, the cycle time of chamber-3 (C3) are constrained by three loops: The inner loop ($S_{3a} \rightarrow P_{3a} \rightarrow S_{3b} \rightarrow P_{3b}$), A-loop ($S_{3a} \rightarrow P_{3a} \rightarrow S_{3b} \rightarrow c_6 \rightarrow \cdots \rightarrow c_2$), and B-loop ($S_{3a} \rightarrow c_3 \rightarrow$ Drop-b$\rightarrow \cdots \rightarrow c_5 \rightarrow S_{3b} \rightarrow P_{3b}$). Thus, the cycle time of C3 can be expressed as:

$$CT(C3) = \max\{CT(\text{A-loop}), CT(\text{B-loop}), CT(\text{Inner-loop})\}; \text{// per two jobs} \tag{10.5}$$

$$CT(\text{A-loop}) = \{\pi + 3\sigma + \delta + \mu\} + \{c_1 + c_2 + c_6\} + p_{3a}$$
$$CT(\text{B-loop}) = \{\pi + 3\sigma + \delta + \mu\} + \{c_3 + c_4 + c_5\} + p_{3b}$$
$$CT(\text{Inner-loop}) = 2\sigma + p_{3a} + p_{3b}$$

As in the case of the hoist plating line, wafers in the cluster tool are subject to the same processing-time constraints. Thus, utilizing the results given in Eqs. 10.4 and 10.5, the cluster tool scheduling problem may be formulated as an LP problem as follows [Kim et al. 2003]:

$$\text{Minimize}(\max\{CT(\text{Robot}), CT(\text{C1}), CT(\text{C2}), CT(\text{A}), CT(\text{B}), CT(\text{Inner-loop})\}) \tag{10.6}$$

$$CT(\text{Robot}) = \{2\pi + 4\sigma + 2\delta + 2\mu\} + \{c_1 + c_2 + c_3 + c_4 + c_5 + c_6\};$$
$$CT(\text{C1}) = 2(\sigma + p_1);$$
$$CT(\text{C2}) = 2(\sigma + p_2);$$
$$CT(\text{A}) = \{\pi + 3\sigma + \delta + \mu\} + \{c_1 + c_2 + c_6\} + p_{3a};$$
$$CT(\text{B}) = \{\pi + 3\sigma + \delta + \mu\} + \{c_3 + c_4 + c_5\} + p_{3b};$$
$$CT(\text{Inner-loop}) = 2\sigma + p_{3a} + p_{3b};$$

Decision variables: p1, p2, p3a, p3b, c1, c2, c3, c4, c5, c6

Subject to:

1. Time window constraints: $\alpha_k \le p_k \le \beta_k$ for k = 1, 2, 3a, 3b
2. Carry time constraints: $\gamma \le c_k$ for k = 1 ~ 6
3. Immediate removal constraints:

$$\{2\pi + 3\sigma + 2\delta\} + \{c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + 2\mu\} \le p_1$$

$$\{2\pi + 3\sigma + 2\delta\} + \{c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + 2\mu\} \le p_2$$

$$\{\pi + \sigma + \delta + \mu\} + \{c_3 + c_4 + c_5\} \le p_{3a}$$

$$\{\pi + \sigma + \delta + \mu\} + \{c_1 + c_2 + c_6\} \le p_{3b}$$

Exercise 10.5. Formulate a cluster tool scheduling problem for the ACD in Fig. 10.26.

10.5 ACTIVITY CYCLE DIAGRAM MODELING OF A FLEXIBLE MANUFACTURING SYSTEM

Figure 10.29 shows a linear-type FMS consisting of four machining centers (MCT), a washing machine (WM), a coordinate measuring machine (CMM), three load/unload stations (LU), a stacker crane or automated guided vehicle (AGV), and a central buffer (CB). This FMS is a Mazatrol FMS, which is one of the most popular FMSs globally [Choi et al. 1996]. Each processing machine (MCT, WM, or CMM) is equipped with an input buffer (IB) and an output buffer (OB). In this section, we present a step-by-step procedure for building a P-ACD model of the FMS line in Fig. 10.29: ACD modeling of job flows, P-ACD modeling of job routing, P-ACD modeling of AGV dispatching, and P-ACD modeling of refixture operations.

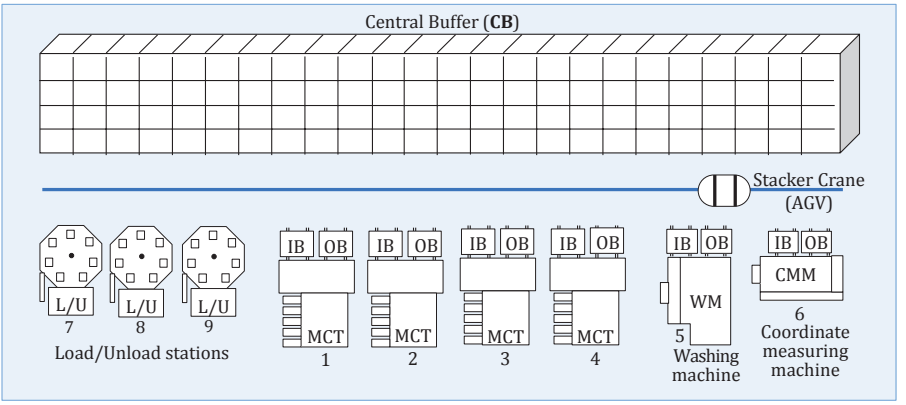


Fig. 10.29. Layout of the Mazatrol FMS.

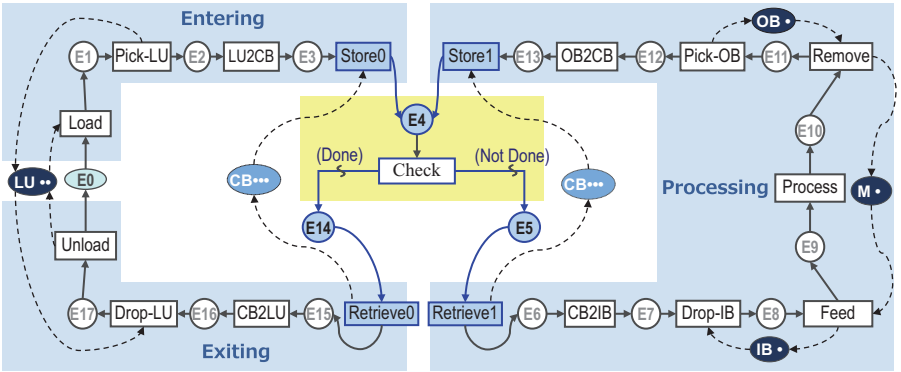


Fig. 10.30. ACD modeling of job flows in the FMS of Fig. 10.29.

10.5.1 ACD Modeling of Job Flows in FMS

As depicted in the classical ACD model in Fig. 10.30, the flow of the jobs in the FMS is divided into entering, processing, and exiting phases. In the Entering phase, a job that has been loaded (Load) at an LU station is picked up by the AGV (Pick-LU), moved to the central buffer CB (LU2CB), and stored in the CB (Store0). The Processing phase begins with retrieving a job from the CB (Retrieve1), and then the job is moved to the IB of a machine (CB2IB), dropped on the IB (Drop-IB), fed into the machine (Feed), and processed by the machine (Process). The processed job is removed from the machine (Remove), picked up at the OB (Pick-OB), moved to the CB (OB2CB), and stored back in the CB (Store1).

A done job stored in the CB is sent out of the FMS line during the Exiting phase, which consists of retrieving the job from the CB (Retrieve0), bringing it to a LU station (CB2LU), dropping it in a LU station (Drop-LU), and unloading it (Unload). Also depicted in Fig. 10.30 are the activity cycles of the load/unload station (LU), central buffer (CB), and each machine's input buffer (IB), output buffer (OB), and machine table (M). In Fig. 10.30, the number of active LU stations is assumed to be 2.

10.5.2 P-ACD Modeling of Job Routing in FMS

In an FMS, each job type has its own routing sequence, and attached to a job are its job-type (j) and current processing-step number (p). When a new job is stored in the central buffer, p is set to 1. Then, p is incremented by one every time the job completes a processing step.

Figure 10.31 shows a P-ACD model of job routes in the FMS. By comparing the P-ACD model in Fig. 10.31 with the classical ACD model in Fig. 10.30, we can see that (1) the activity nodes in the entering and exiting phases (which are now combined into the Handling phase) are parameterized with job type

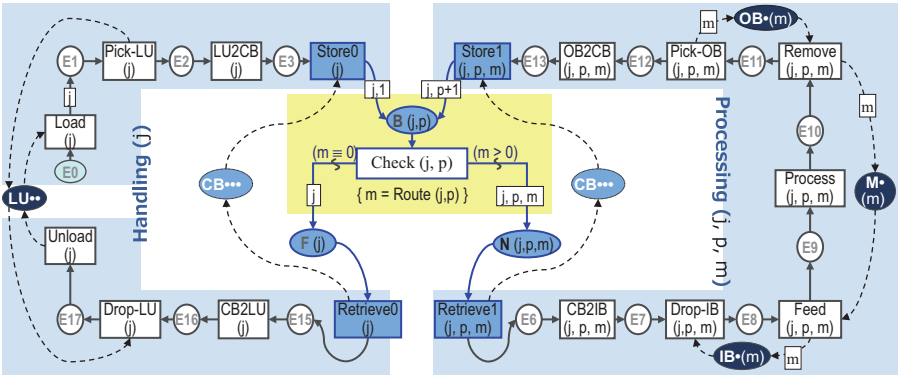


Fig. 10.31. P-ACD modeling of job routing in the FMS.

j, (2) the Check activity node is parameterized by j and p, and (3) the activity nodes in the Processing phase are parameterized by j, p, and m.

For each job, its type j is assigned at the Load activity, and it is passed along up to the Store0 activity where the processing-step p is set to 1 and its parameters {j, p = 1} are passed to the buffer queue B (j, p). Then, the machine number (m) for the p-th processing operation of a type j job is obtained at the Check activity by invoking the route function Route(j, p). The route function will return 0 (m = 0) if the job is done, otherwise it will return m > 0. When m > 0, the job with parameters {j, p, m} is stored in the new job queue N (j, p, m), and it is passed all the way to the activity node Store1 where the parameters {j, p+1} are passed to Check. If the job is done (i.e., m ≡ 0), it is stored in the finished job queue F (j) and passed along up to the activity node Unload.

Descriptions of major activity nodes of the P-ACD model in Fig. 10.31 are given in the following:

- Load (j): load a job and assign its job type j
- Pick-LU (j): pick up a job at an LU station
- LU2CB (j): move the job from an LU station to the CB
- Store0 (j): store the job at the CB
- Check (j, p): get m (machine number) and check
- Retrieve1 (j, p, m): retrieve a new job from the CB
- CB2IB (j, p, m): move the job from the CB to the IB (input buffer)
- Drop-IB (j, p, m): drop the job at the IB of machine m
- Store1 (j, p, m): store the processed job at the CB
- Retrieve0 (j): retrieve a finished job from the CB

Descriptions of queue nodes of the P-ACD model in Fig. 10.31 are also given in the following:

- **E0**: jobs to be released
- **E1** (j): loaded jobs at the LU station
- **B** (j,p): buffered jobs at the CB
- **F** (j): finished jobs stored at the CB
- **N** (j, p ,m): new (unfinished) jobs stored at the CB
- **CB**: available slots in the CB
- **IB** (m): input buffer of machine m
- **OB** (m): output buffer of machine m
- **M** (m): machine table of machine m
- **LU**: available slots of load/unload stations

10.5.3 P-ACD Modeling of AGV Dispatching Rules in FMS

In the FMS, all job movements are handled by AGVs. As shown in Fig. 10.32, there exist four types of requests for AGV in an FMS: Request **R1** to bring in a new job to a machine; request **R2** to take out a processed job from a machine; request **R3** to bring in a newly loaded job from the load/unload station to the central buffer; request **F** (tokens in the finished job queue) to take out a finished job from the central buffer. In general, request **R1** has the highest priority because preventing machines from starving is most critical, request **R2** has the next priority because preventing machines from blocking is also important, and request **F** has the lowest priority.

Descriptions of the activity nodes that are newly introduced for AGV dispatching in Fig. 10.32 are given in the following:

- Move2CB1 (j, p, m): AGV moves to **CB** to pick up a new job of type j at processing step p to bring it to **IB**(m).
- Move2OB (m): AGV moves to **OB**(m) to pick up a job that was processed at m.

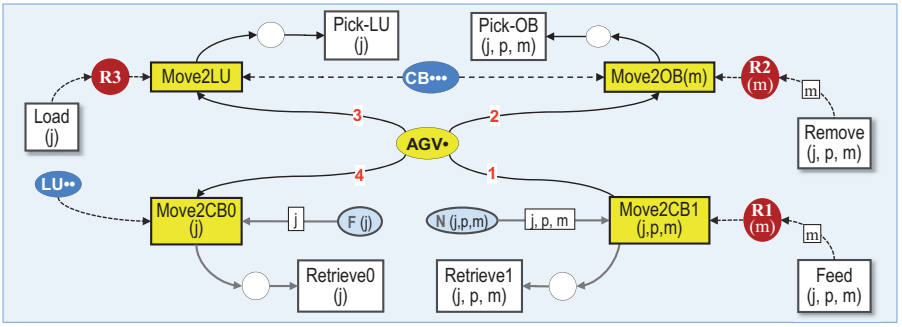


Fig. 10.32. P-ACD modeling of AGV dispatching in the FMS.

- Move2LU: AGV moves to **LU** station to pick up a new job.
- Move2CB0(j): AGV moves to **CB** to pick up a finished job of type j to bring it to **LU** station for unloading.

Descriptions of the queue nodes that are newly introduced for AGV dispatching in Fig. 10.32 are given in the following:

- **R1** (m): request for AGV to bring in a new job to **IB**(m)
- **R2** (m): request for AGV to take out a job from **OB**(m)
- **R3**: request for AGV to bring in a newly loaded job from **LU** station to **CB**
- **F** (j): finished job count acts as a request for AGV to take out a finished job from **CB**

Upon receiving **R1**(m), which is a request for AGV to bring in a new job (i.e., a job that is not finished) to machine m, the AGV performs the following actions: (1) moves to the central buffer after reserving a new job that is bound to machine m (Move2CB1) and (2) retrieves the reserved job (Retrieve1). Other types of requests are handled similarly.

By merging the AGV dispatching model in Fig. 10.32 into the job routing model in Fig. 10.31, a P-ACD model of the entire FMS operation is obtained as shown in Fig. 10.33. Note in Fig. 10.33 that there are four AGV activity cycles, one for each type of AGV requests. For example, the AGV activity cycle for request **R1** is Move2CB1 → Retrieve1 → CB2IB → Drop-IB →. It should be also noted that the job activity cycle has been changed slightly: in Fig. 10.33, activity node Move2CB1 is inserted in the job activity cycle right after the queue node **N**, and Move2CB0 is inserted right after **F**.

Conceptually, the components of the P-ACD model in Fig. 10.33 can be aggregated into two handling and processing boxes as depicted in Fig. 10.34.

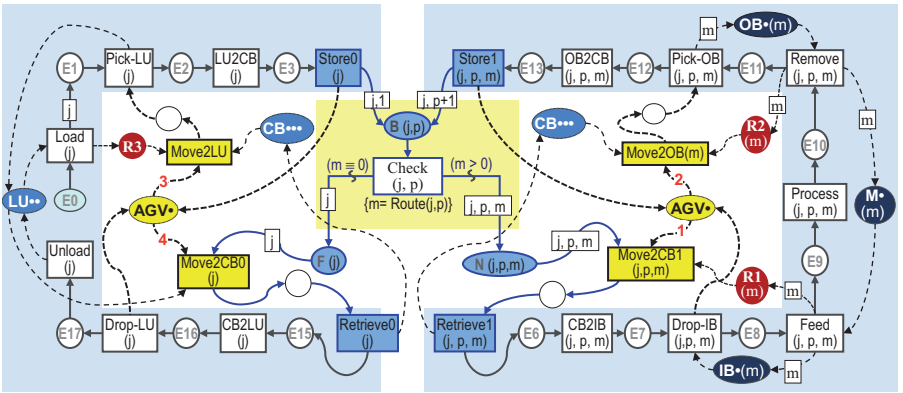


Fig. 10.33. P-ACD modeling of the entire FMS operation.

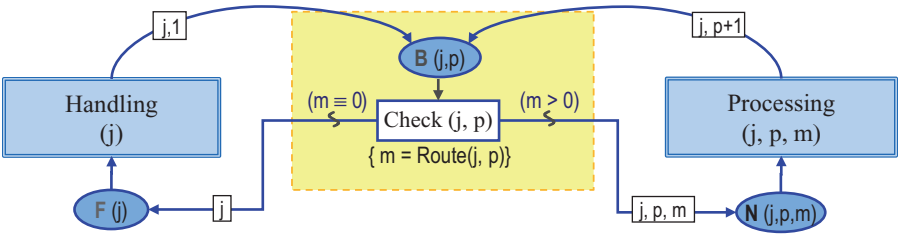


Fig. 10.34. Aggregate P-ACD model of the Mazatrol FMS without refixture.

10.5.4 P-ACD Modeling of Refixture Operation and Heterogeneous FMS

In the FMS model in Fig. 10.34, there are three types of processing operations (machining, wash, and measuring). Machine numbers for machining are m = 1–4, for wash it is m = 5, and for measuring it is m = 6. A job that does not require a refixture operation goes through the routing sequence Load → Machining → Wash → Measuring → Unload. The P-ACD model in Fig. 10.33 assumes that the activity cycle of the job does not contain a refixture operation. In this case, the FMS is a homogeneous FMS in which all the processing operations follow the same activity sequence given by:

$$\begin{aligned}\text{Retrieve} \rightarrow \text{CB2IB} \rightarrow \text{Drop-IB} \rightarrow \text{Feed} \rightarrow \textbf{Process} \\ \rightarrow \text{Remove} \rightarrow \text{Pick-OB} \rightarrow \text{OB2CB} \rightarrow \text{Store.}\end{aligned} \tag{10.7}$$

In practice, jobs that require a refixture operation may go through the routing sequence Load→Machining1→Wash→Refixture→Machining2→Measuring→Unload. The refixture operation is performed at a LU station (m = 7–9), and its activity sequence is given by:

$$\text{Retrieve} \rightarrow \text{CB2LU} \rightarrow \text{Drop-LU} \rightarrow \textbf{Refix} \rightarrow \text{Pick-LU} \rightarrow \text{LU2CB} \rightarrow \text{Store.} \tag{10.8}$$

Figure 10.35 shows an aggregate P-ACD model of the Mazatrol FMS operations involving the refixture operation where individual components are aggregated into three boxes: handling box, processing box, and refixture box. Observe in Fig. 10.35 that the new job queue **N** in Fig. 10.34 has been divided into **NP** (the new job queue for processing) and **NR** (the new job queue for refixture). The machine-number set for processing machines is denoted by **P** and that for refixture machines by **R**, where $\mathbf{P} = \{1, 2 \cdots 6\}$ and $\mathbf{R} = \{7, 8, 9\}$.

The P-ACD model in Fig. 10.35 is in effect a heterogeneous FMS model supporting two different classes of processing operations: one class of processing operations follows the activity sequence given by Eq. 10.7 and the other class follows Eq. 10.8. In this book, we use the term (processing) *operation class* for a set of (processing) operation types having the same activity sequence.

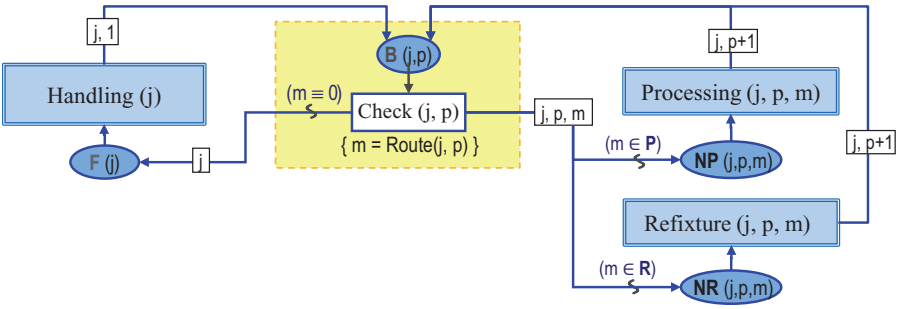


Fig. 10.35. Aggregate P-ACD model of the Mazatrol FMS with refixture operation.

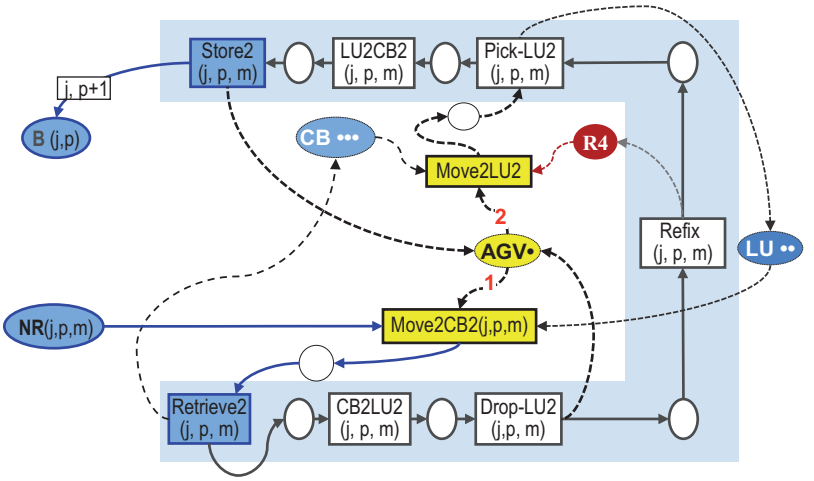


Fig. 10.36. P-ACD model of the refixture part in Fig. 10.35.

Here, the refixture operation is regarded as a class of processing operation. The aggregate P-ACD model in Fig. 10.35 provides a framework for building a generalized P-ACD model of heterogeneous FMSs.

Figure 10.36 shows the P-ACD model of the Refixture part in the aggregate P-ACD model in Fig. 10.35. Descriptions of the major activity nodes and queue nodes that are newly introduced in Fig. 10.36 are given in the following:

- Move2CB2 (j ,p, m): AGV moves to **CB** to pick up a new job of type j at processing step p to bring it to **LU** station m
- Retrieve2 (j, p, m): retrieve a new job from **CB** for refixture
- CB2LU2 (j, p, m): move the job from **CB** to **LU** station m
- Move2LU2: AGV moves to **LU** to pick up a refixed job
- **R4**: request for AGV to bring in a refixed job from **LU**

A complete list of C# codes for the FMS simulator may be found in the official website of this book (http://VMS-technology.com/Book/ACDSimulator).

10.6 FORMAL MODEL CONVERSION

This section presents methods of formal model conversion from ACD to EG (event graph) and SG (state graph) together with examples of converting ACD models into EG models and SG models.

10.6.1 Conversion of ACD Models to Event Graph (EG) Models

Methods of mapping timed Petri net (TPN) models into event graph models were investigated by Schruben and Yucesan [1994]. Recall that a classical ACD is a TPN. In principle, any ACD model may be converted into an EG model. The basic rules for a formal model conversion are:

1. An arc in the ACD model becomes an event node in the event graph (EG)
2. A token variable of ACD becomes a state variable of EG
3. A queue node *K* of ACD becomes a conditional edge of EG with (K > 0), preceded by an event node with K++ and succeeded by a event node with K—
4. An activity node with duration t_d becomes a time-delay edge of EG with a time delay t_d

Figure 10.37 shows the basic conversion relations: (1) Arcs β, γ, δ are converted to event nodes; (2) token variable J becomes a state variable; (3) queue node J, preceded by arc β and followed by arc γ, is converted to the conditional edge (J > 0), preceded by event β with {J++} and followed by event γ with {J—}; (4) activity node Process <t_p> is converted to the time-delay edge with t_p.

Figure 10.38(a) shows an ACD-to-EG conversion template for the Job Creator model where the basic ACD-to-EG conversion rules are applied as follows:

1. ACD arcs (α and β) became event nodes (α and β) in the EG
2. Token variables C, S, and J became state variables

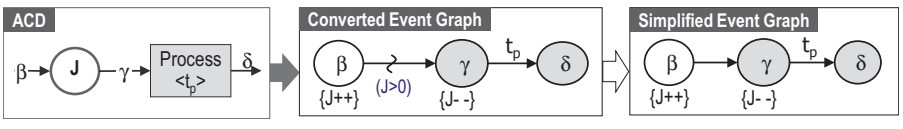


Fig. 10.37. Basic relations for converting an ACD model into an event graph model.

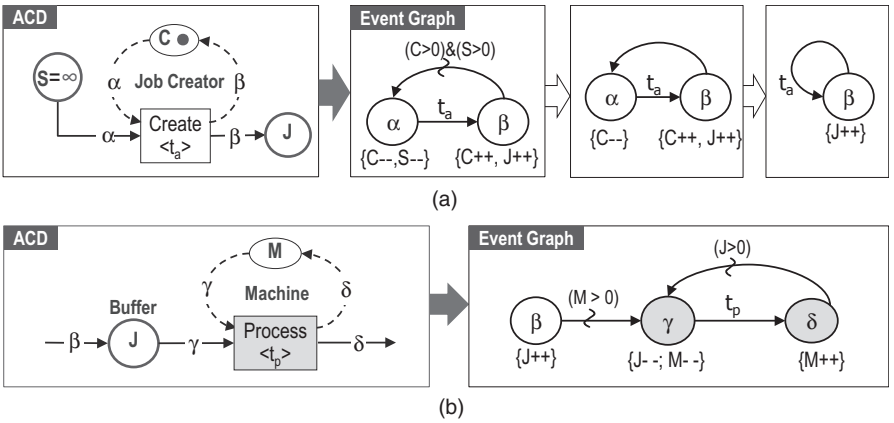


Fig. 10.38. ACD to event graph conversion templates for (a) the Job Creator and (b) Buffer-Machine models.

3. Queue node C became the edge $\beta \rightarrow \alpha$, having a condition (C > 0) & (S > 0), with state update at node-α {C--, S--} and state update at node-β {C++, J++}
4. The Create activity with duration t_a became the edge $\alpha \rightarrow \beta$ having a time delay t_a

Figure 10.38(b) represents an ACD-to-EG conversion template for the Machine model, which was also obtained by applying the basic ACD-to-EG conversion rules.

10.6.2 Conversion of ACD Models to State Graph (SG) Models

As mentioned earlier, an arc in an ACD model denotes an event; an entity queue (i.e., a queue node in an entity activity cycle) represents a passive resource; and a resource activity cycle represents an active resource. Thus, the basic ACD-to-SG conversion rules are:

1. An *arc* in the ACD model becomes an event message in the state graph
2. A token variable maps into a set of state nodes
3. An entity queue node is a passive resource and it becomes an object
4. A resource queue node becomes an external transition state node
5. An activity node becomes an internal transition state node
6. A resource cycle becomes an object (one object for each active resource)

Figure 10.39 shows an ACD-to-SG conversion template for the Job Creator model. The job creator model consists of the Source object and Creator object

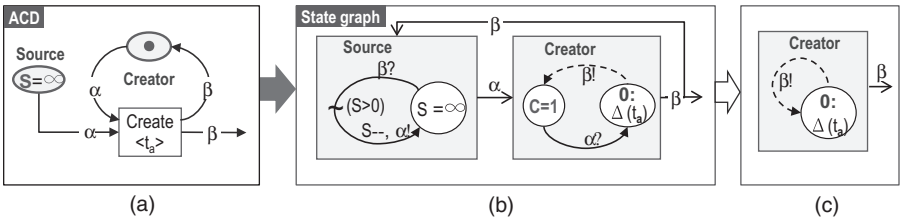


Fig. 10.39. ACD to state graph conversion template for the Job Creator model.

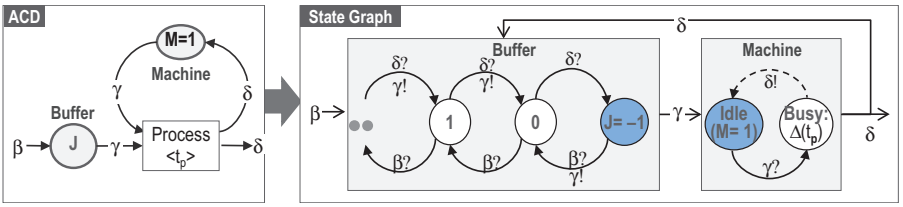


Fig. 10.40. ACD to state graph conversion template for the Buffer-Machine model.

that are interacting with each other via the event messages α and β as depicted in Fig. 10.39(b). The Source object is a single state machine, which, upon receiving an input message β, decreases the value of S by one and sends out an output message α. However, since the value of S is infinite, an input message β always generates an output message α. Thus, the redundant Source object in Fig. 10.39(b) can be removed and the Job Creator model reduces to the single state atomic state graph model of Fig. 10.39(c).

Figure 10.40 shows an ACD-to-SG conversion template for the Buffer-Machine model: the Buffer queue becomes an infinite state atomic state graph model "Buffer"; the Machine activity cycle becomes a two-state atomic state graph model "Machine"; the events β, γ, δ are used as messages in the state graph model. The initial states of the two atomic models in the state graph are Backlog (J = –1) and Idle (M = 1), respectively.

10.6.3 Examples of Formal Model Conversion

10.6.3.1 Examples of ACD-to-EG conversion Figure 10.41 shows an ACD-to-EG (event graph) conversion example for a two-server model. The two-server model conversion of Fig. 10.41 is obtained by concatenating the two conversion templates in Fig. 10.38. It becomes a single server model conversion case if M = 1, and a general multi-server model if M = n > 1.

Reproduced in Fig. 10.42(a) is the ACD model of the car repair shop given in Fig. 6.24 with an event name given to each arc of the ACD. If we apply the basic ACD-to-EG conversion rules given in Section 10.6.1 to the ACD model, a converted event graph is obtained as shown in Fig. 10.42(b).

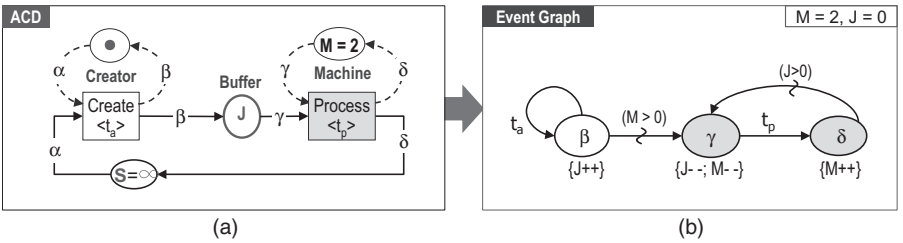


Fig. 10.41. Converting a multi-server ACD model into an event graph model.

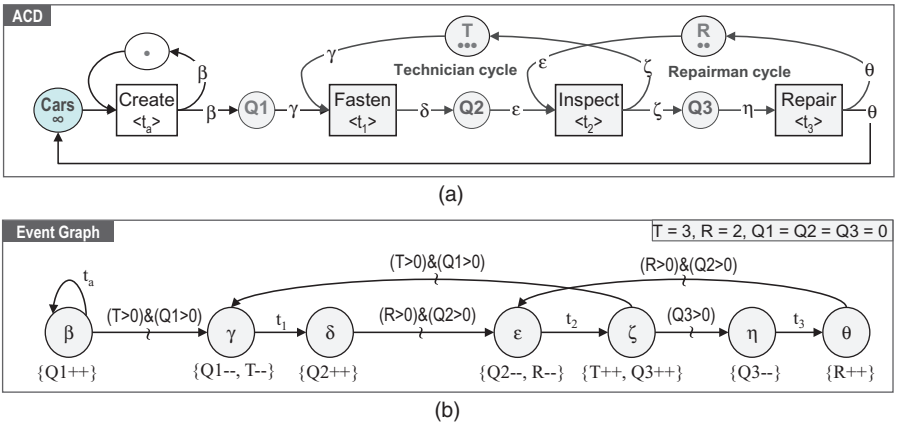


Fig. 10.42. Converting the car repair shop ACD model into an event graph model.

10.6.3.2 Examples of ACD-to-SG conversion As mentioned earlier, converting an ACD model into a state graph (SG) model is not that simple. The basic ACD-to-SG conversion rules say that (1) an individual resource in the ACD becomes one object (i.e., an atomic model) in the state graph model, and (2) an entity queue node becomes one object. Thus, the ACD model in Fig. 10.42(a) would result in a state graph model having seven objects (i.e., atomic models). In order to simplify the discussion somewhat, it is assumed that there are one technician and one repairman in the car repair shop.

Figure 10.43(a) shows the ACD model of the reduced car repair shop where it is highlighted that (1) Q1 is an entity queue node but Q2 and Q3 are resource queue nodes and (2) the in/out arcs of the shared activity (Inspect) are individually identified (e.g., ε_R and ε_T). If we apply the basic ACD-to-SG conversion rules given in Section 10.6.2 (as well as an additional rule) to the ACD model Fig. 10.43(a), a converted state graph may be obtained as shown in Fig. 10.43(b).

Reproduced in Fig. 10.44(a) is the ACD model of a two-server system given in Fig. 10.41(a). Figure 10.44(b) shows an alternative representation of

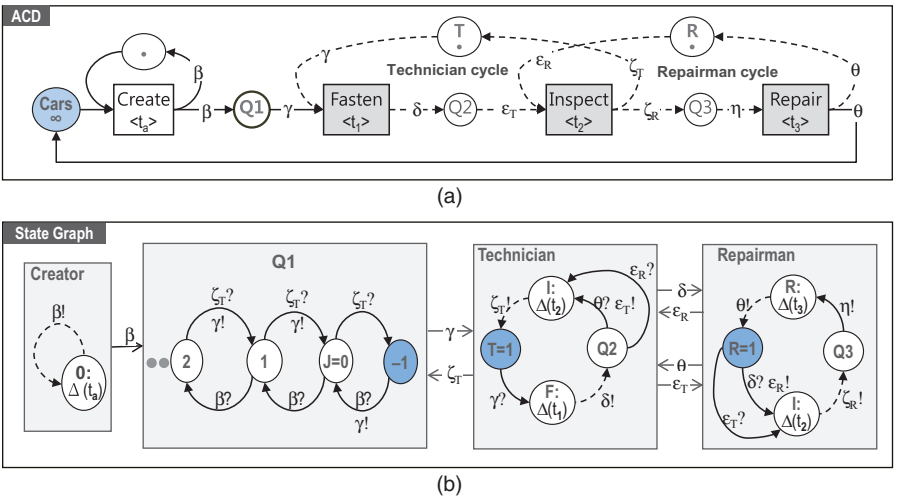


Fig. 10.43. (a) ACD model and (b) converted state graph model of the car repair shop.

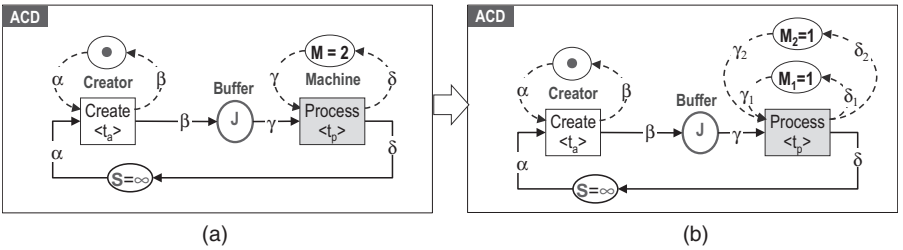


Fig. 10.44. Two ACD models for the same two-server system given in Fig. 10.41(a).

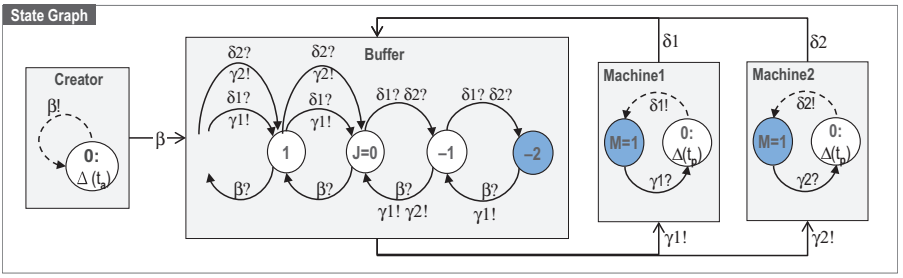


Fig. 10.45. State graph model of a two-server system converted from the ACD model given in Fig. 10.44(b).

multiple (identical) resources in which a separate queue is defined for each resource. Figure 10.45 shows a multi-server system state graph model corresponding to the two-server (M = 2) ACD model given in Fig. 10.44(b). When M = 1, a single server system state graph model is obtained by concatenating the Creator model in Fig. 10.39 and the Buffer-Machine model in Fig. 10.40.

If the number of machines becomes large, say M = 20, the state graph model would become quite complicated. In general, compared to the ACD-to-EG conversion, the ACD-to-SG conversion is harder to define. A conversion approach to developing a complex DEVS model is proposed in the literature [Choi et al. 2003].

APPENDIX 10A: PETRI NETS

Petri nets, which have been developed from the Carl Adam Petri's doctoral dissertation *Communication with Automata* in 1962, were designed to model systems with interacting concurrent components [Peterson 1981]. Basics of the Petri net, together with its relationship to the ACD, will be described.

10A.1 Definitions of Petri Nets

A Petri net is defined either as a graph or as a structure. A Petri-net graph is a graphical representation of Petri net where a circle node represents a place, and a bar node represents a transition. Directed arcs connect the places and transitions. Figure 10A.1 is a Petri-net graph with four circle nodes and three bar nodes. Multiple arcs are allowed from one node to another. Thus, a Petri-net graph is a bipartite directed multi-graph. Petri-net graph G is formally defined as:

Petri-net graph $G = (V, A)$

1. $V = \langle v1, v2 \cdots vn \rangle$ is a set of vertices
2. $A = \{a_1, a_2 \cdots a_m\}$ is a bag of directed arcs with $a_i = (v_j, v_k)$
3. $V = P \cup T$, where P is the set of places and T is the set of transitions
4. For each arc $a_i = (v_j, v_k)$, we need $(v_j \in P \ \& \ v_k \in T)$ or $(v_j \in T \ \& \ v_k \in P)$

A Petri-net structure is a set theoretic representation of Petri net composed of a set of places (P), a set of transitions (T), an input function (I), and an output function (O). The input function $I(t_j)$ is a mapping from a transition t_j to the input places of the transition, and the output function $O(t_j)$ maps a

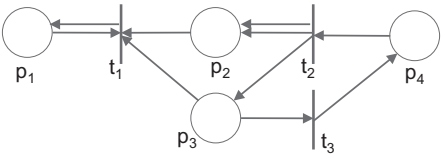


Fig. 10A.1. Petri-net graph.

transition t_j to the output places of the transition. Petri-net structure S is formally defined as:

$$\text{Petri-net Structure } S = \langle P, T, I, O \rangle$$

1. $P = \{p_1, p_2 \cdots p_n\}$: a finite set of places
2. $T = \{t_1, t_2 \cdots t_m\}$: a finite set of transitions
3. $I: T \rightarrow P^{\infty}$: input function (from transitions to bags of places)
4. $O: T \rightarrow P^{\infty}$: output function (from transitions to bags of places)

For the Petri net of Fig. 10A.1, the components of the Petri-net structure are given by:

1. $P = \langle p1, p2, p3, p4 \rangle$
2. $T = \langle t1, t2, t3 \rangle$
3. $I(t_1) = \{p_1, p_2, p_3\}$, $I(t_2) = \{p_4\}$, $I(t_3) = \{p_3\}$
4. $O(t_1) = \{p_1\}$, $O(t_2) = \{p_2, p_2, p_3\}$, $O(t_3) = \{p_4\}$

10A.2 Petri-Net State and Execution

The state of a Petri net is defined by its marking μ, which is an assignment of tokens to the places of a Petri net. Tokens are assigned to the places and can be thought to reside in the places. A Petri net with marking is called a marked Petri net which is formally defined as:

A marked Petri-net M = (P, T, I, O, μ) is defined by a Petri-net structure S and a marking μ.

Shown in Fig. 10A.2 is a marked Petri-net graph with a marking $\mu = (1, 0, 1, 0)$. The state space of a Petri-net with n places is the set of all markings N^n. The state change of Petri net is defined by a change function (δ), called the *next-state function*, which is defined as [where #(e, B) denotes number of occurrences of "e" in the bag B]:

The *next-state function* ($\delta: N^n \times T \rightarrow N^n$) for a marked Petri-net M and transition t_j is defined iff $\mu(p_i) \geq \#(p_i, I(t_j))$ for all $p_i \in P$. If $\delta(\mu, t_j)$ is defined $\delta(\mu, t_j) = \mu'$, where $\mu'(p_i) = \mu(p_i) + \#(p_i, O(t_j)) - \#(p_i, I(t_j))$

In the marked Petri net of Fig. 10A.2, the transition t_3 is enabled with the current marking $\mu = (1, 0, 1, 0)$. After firing t_3, the new marking $\mu' = (1, 0, 0, 1)$. The execution of a Petri net is controlled by tokens: It executes by firing transitions; a transition fires by removing tokens from its input places and creating new tokens that are distributed to its output places. A transition may fire if it is enabled. A transition is enabled if each of its input places has at

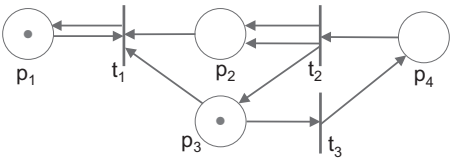


Fig. 10A.2. Marked Petri net.

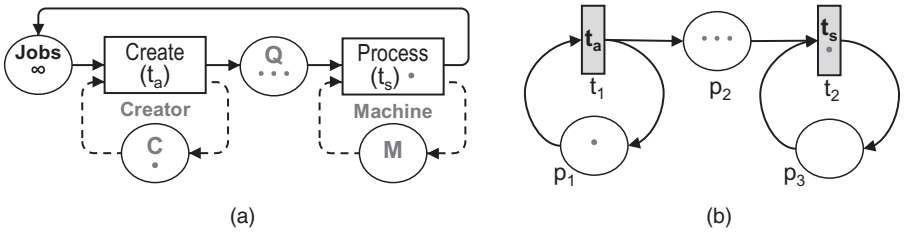


Fig. 10A.3. (a) ACD and (b) its Petri net.

least as many tokens as arcs from the place to the transition, which may be formally stated as:

A transition $t_j \in T$ in a marked Petri-net $M = \langle P, T, I, O, \mu \rangle$ is enabled if for all $p_i \in P$, $\mu(p_i) \geq \#(p_i, I(t_j))$.

10A.3 Extended Petri Nets and the ACD

The modeling power of the "standard" Petri net is not enough for modeling "real-life" systems. Thus, extended Petri nets have been proposed such that

1. The firing of transitions takes time (timed Petri net)
2. Tokens have different attributes (colored Petri net)
3. Firing is based on zero-testing (Inhibitor arc)
4. A token may branch based on condition (Test arc), etc.

The original ACD having no arc attributes is by definition a *timed Petri net* without multiple arcs. Figure 10A.3 shows the single server system ACD (Fig. 10.2) together with its Petri-net graph.

An extended ACD with arc conditions (see Section 6.2.2 of Chapter 6) is a timed Petri net with zero testing. Zero testing decreases the decision power of Petri nets such as deadlock detection, but it increases the modeling power of Petri nets. In fact, a Petri net with zero testing produces a modeling scheme capable of modeling a Turing machine [Peterson 1981]. Thus, a Petri net with zero testing can model any discrete-event system that can be represented in a digital computer; so can the extended ACD with arc conditions.

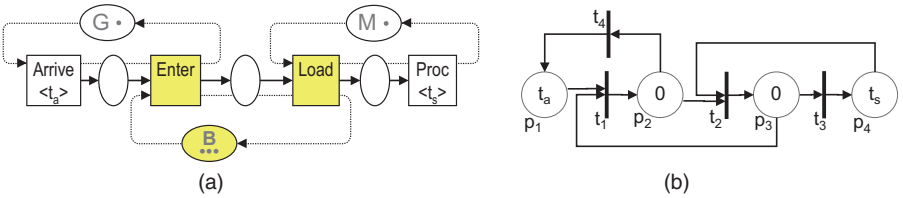


Fig. 10A.4. (a) ACD and (b) p-time Petri net.

10A.4 Restricted Petri Nets

There are special types of Petri nets that have some restrictions in terms of number of tokens in a place or number of arcs associated with a place. A Petri net in which a place is not allowed to have multiple tokens and has times associated with it is called a *p-time Petri net* [Khansa et al. 1996]. Thus, an ACD without queues having multiple tokens is easily converted to a p-time Petri net by converting each activity of ACD to a place of Petri net and each queue to a transition as depicted in Fig. 10A.4. A Petri net in which each place has exactly one input and output arc is called an *event graph*, and an event graph with timed transitions is referred to as a *timed event graph*. The Petri net shown in Fig. 10A.3(b) is a timed event graph. These restricted Petri nets are mostly used in scheduling of cyclic systems such as cluster tools [Kim et al. 2003] and cyclic flow shop [Ren et al. 2005].

10A.5 Modeling with Petri Nets

Modeling with Petri nets by itself is a huge area, and it is one of the most intensively investigated subjects. An excellent treatment of the subject is given in Peterson [1981], and there are a number of books such as Dicesare et al. [1993] dealing with Petri-net modeling. There have been annual conferences on the applications and theory of Petri nets since 1980. However, as far as the art of M&S of industrial systems is concerned, modeling with Petri nets in most cases does not seem to be "natural" compared to modeling with one of the "world-view" formalisms (i.e., event graph, ACD, and finite state machine) even though there are certain cases where modeling with Petri nets is more natural [Peterson 1981].

An alternative approach to modeling with Petri nets may be to construct a formal model in other forms such as event graph, ACD, or finite state machine and convert it into a Petri-net model. A Petri-net model may be constructed from an event graph model by using the event-condition method of Petri-net modeling [Peterson 1981]. Also, a method of building a Petri-net model from a finite state machine is given in Peterson [1981]. An ACD model, perhaps except a parameterized ACD, is trivially converted into a timed Petri-net model.

CHAPTER 11

Advanced Event Graph Modeling for Integrated Fab Simulation

In the practice of tolerance, one's enemy is the best teacher.

—Dalai Lama

11.1 INTRODUCTION

Production simulation, often referred to as *simulation-based scheduling*, is widely accepted in the high-tech industry, which covers the semiconductor and flat panel display (FPD) industries. Detailed simulation of automated material handling systems (AMHS) is also widely adopted in the high-tech industry [Wang and Lin 2004]. Among the commercial simulation packages, AutoSched AP® (ASAP) seems to be the most popular tool for production simulation [Gan et al. 2007], while AutoMod® is reported to be most popular in material handling simulation [Kim et al. 2009].

On the other hand, there has been a growing need for an integrated simulation where production simulation is carried out together with detailed material handling simulation. To meet this need, a software module called MCM® (Model Communication Module) has been developed, which provides socket-based communication between ASAP and AutoMod. With MCM®, IBM had developed the AMHS-embedded integrated simulation system depicted in Fig. 11.1(a) in which ASAP's fabrication process models for production simulation can communicate with AutoMod's material handling models for AMHS simulation [Norman et al. 1999]. INTEL also developed a similar system as depicted in Fig. 11.1(b), which was reported to have been applied successfully [Pillai et al. 2004].

This chapter presents a detailed procedure for developing an AMHS-embedded integrated Fab (fabrication line) simulation software system similar

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.

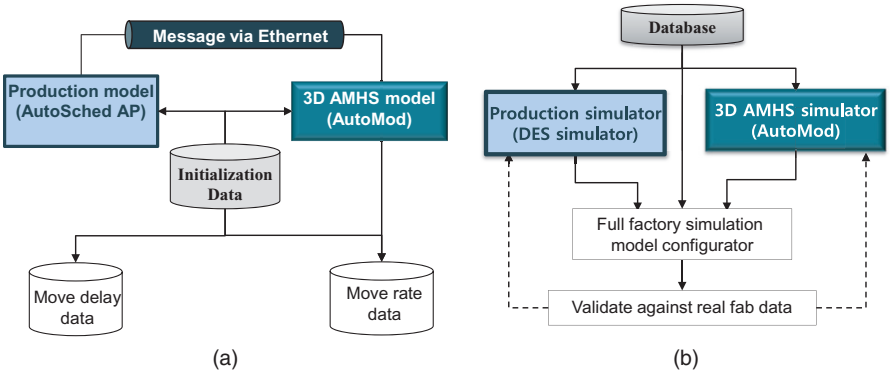


Fig. 11.1. AMHS-embedded integrated simulation by (a) IBM and (b) INTEL.

to the INTEL's system depicted in Fig. 11.1(b). The authors' team had a chance to develop IFS®, an AMHS-embedded integrated Fab simulation software system, with a couple of FPD manufacturers in Korea.

The rest of the chapter is organized as follows. A brief description of FPD fabrication systems is provided in Section 11.2. An object-oriented event graph modeling framework for a production simulation of FPD Fab is given in Section 11.3, followed by Section 11.4 on the framework for integrated simulation of production and material handling in a FPD Fab. A comprehensive procedure for developing an AMHS-embedded integrated simulator is described in Section 11.5 where a prototype system named IFS® is introduced.

11.2 FLAT PANEL DISPLAY FABRICATION SYSTEM

In this section, a brief description of the FPD fabrication system is provided. First, an overview of FPD Fab is given followed by a schematic description of processing equipment in a typical FPD Fab. Configurations of material handling system hardware and software are also briefly described.

11.2.1 Overview of FPD Fab

In a typical thin-film transistor liquid crystal display (TFT-LCD) fabrication line (called *Fab* for short), a large number of product types are produced concurrently, 24 hours a day, 365 days a year. The fabrication process of TFT-LCD is similar to that of semiconductor wafer: It basically is a series of layer patterning operations. In a modern TFT-LCD Fab, glasses go through approximately four to five patterning cycles to form TFTs on the surface of each glass. Each patterning cycle usually consists of deposition, cleaning, photolithography, etching, stripping, and inspection. Fabrication of a typical TFT

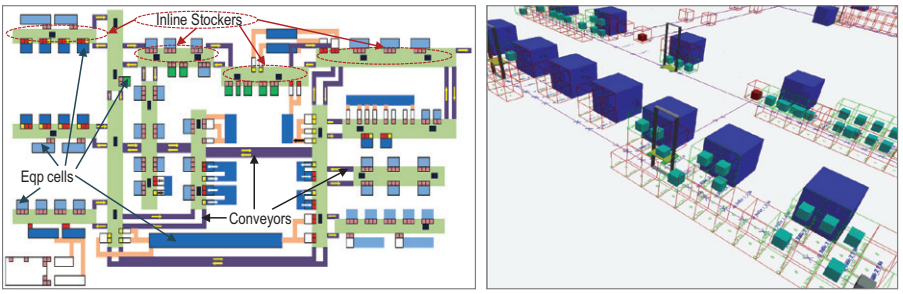


Fig. 11.2. (a) Layout and (b) animation screen of TFT-LCD Fab.

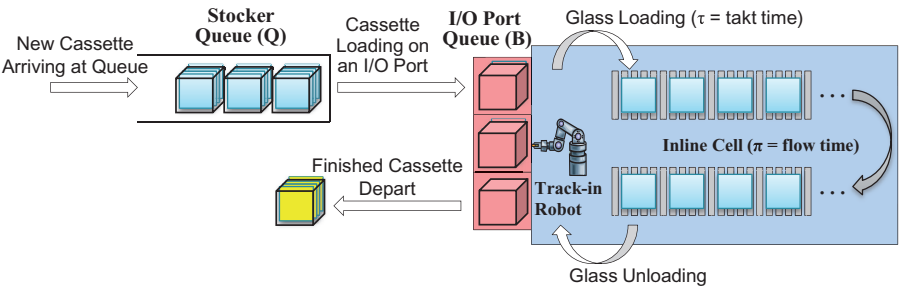


Fig. 11.3. Schematic view of a uni-inline cell.

requires approximately 30 to 40 steps and its turn-around time (i.e., sojourn time) is about 48 hours.

Figure 11.2(a) shows a schematic view of a modern TFT-LCD Fab layout where processing equipment cells are attached to inline stockers (there are 14 of them in the figure) that are connected via conveyors. Figure 11.2(b) is an animation picture of a TFT-LCD Fab.

11.2.2 FPD Processing Equipment

As described in Chapter 4 (see Section 4.5.5), the jobs in a TFT-LCD Fab are glasses that go through the processing equipment in batches with each batch (or lot) stored in a cassette. According to their material handling characteristics, the processing equipment in a FPD Fab can be classified into four types: uni-inline cell, bi-inline cell, oven type equipment, and chamber type equipment.

Figure 11.3 shows a schematic view of a uni-inline type processing equipment cell called a *uni-inline cell*. A cassette with new glasses that is stored in the stocker queue (**Q**) is loaded on a slot (or port) in the I/O port queue (**B**), which is called *Cassette Loading*. The glasses are then loaded into the inline

cell using the track-in robot, with one glass being loaded at every takt time (τ), which is called *Glass Loading*. It takes a flow time (π) for a glass to reach the end of the cell where it is unloaded into the unloading cassette located at the I/O port queue. The unloading cassette departs when it is filled with finished glasses.

Figure 11.4 shows a schematic view of a bi-inline cell. The bi-inline cell can be divided into the cassette-loading section and cassette-unloading section. The behavior of a bi-inline cell is the same as that of a uni-inline cell, except the in-port is located in one inline stocker and the out-port is located in another inline stocker. As a result, a mechanism for handling empty cassettes has to be provided. (If no empty cassettes are available at the out-port, the finished glasses cannot be unloaded.)

Figure 11.5 shows a schematic view of oven type equipment. The cassette loading and cassette departure in the oven type equipment are the same as those in the uni-inline cell. However, in the oven type case, all the glasses in a cassette are loaded together to process the batch (of size b) for a given processing time (π), and then the oven is cleared. The last type is the chamber type equipment consisting of parallel processing chambers, input buffer, output buffer, I/O port, and a track-in robot. At the I/O port, glasses in a cassette are moved into the input buffer from which the glasses are loaded into and unloaded from the processing chambers. The finished glasses are unloaded from the chambers into the output buffer and then moved into the cassettes in the I/O port.

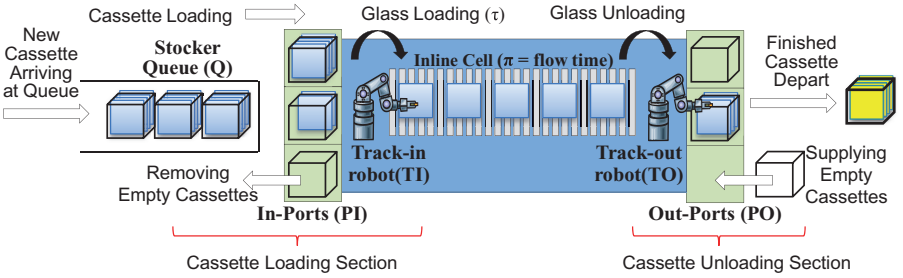


Fig. 11.4. Schematic view of a bi-inline cell.

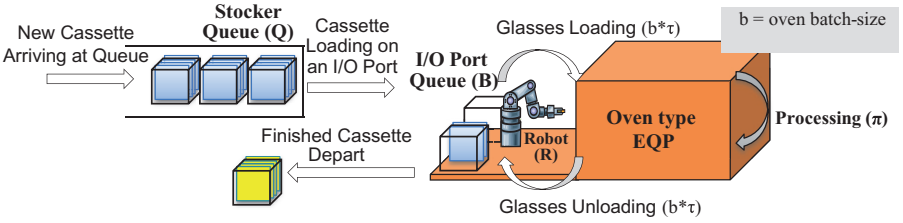


Fig. 11.5. Schematic view of oven type equipment.

11.2.3 Material Handling System

Figure 11.6 shows a schematic view of the material handling hardware in an FPD Fab. The AMHS (automated material handling system) in an FPD Fab is a network of inline stockers connected via conveyors (as well as via bi-inline cells). An inline stocker has a number of shelves that are used as stocker buffers (for temporarily storing cassettes), stocker in-ports (for receiving a cassette from an incoming conveyor), stocker out-ports (for sending a cassette to an outgoing conveyor), equipment (EQP) I/O ports, etc. Material handling in an inline stocker is performed by one or two stacker cranes.

A schematic view of a material handling control software system in a modern Fab is given in Fig. 11.7. Decisions regarding where-next (destination equipment) selection and what-next (input jobs) selection are made by the RTD (real-time dispatcher), while route planning and handling equipment scheduling are performed by the MCS (material control system) in the Fab. These decisions are made based on the current states of the Fab, and they have

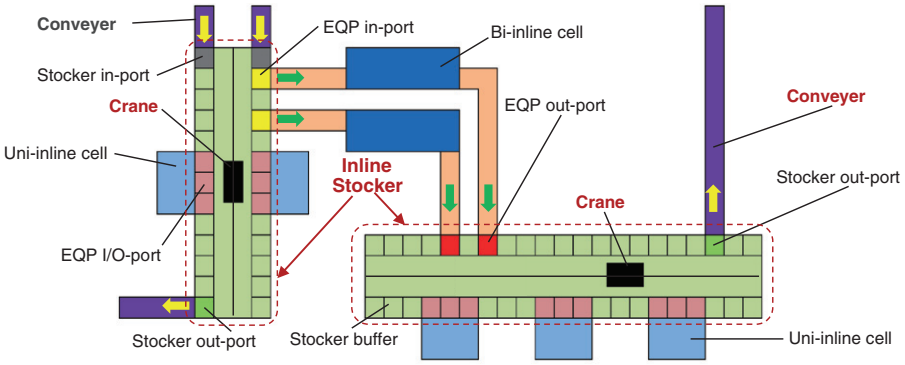


Fig. 11.6. Schematic view of material handling hardware in an FPD Fab.

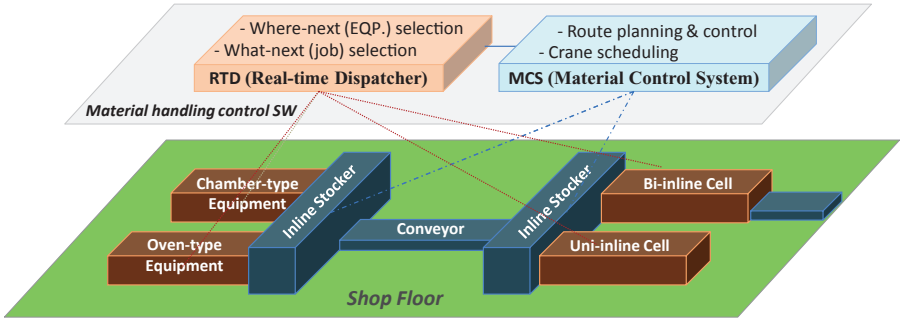


Fig. 11.7. Schematic view of material handling software in an FPD Fab.

to be made in real time. The software system that provides the real time Fab data is called an MES (manufacturing execution system).

11.3 PRODUCTION SIMULATION OF A FLAT PANEL DISPLAY FAB

A flat panel display (FPD) Fab is a job shop consisting of different types of processing equipment. Such a job shop is often referred to as a *heterogeneous job shop* or a *mixed job shop*. As described in the previous section, the four types of processing equipment are widely found in a FPD Fab: uni-inline cell, bi-inline cell, chamber type equipment, and oven type equipment. In this section, we present an object-oriented event graph modeling approach to developing a production simulator for a FPD Fab consisting of uni-inline cells and oven type equipment. (The integrated Fab simulator IFS® covers all the four equipment types, but the remaining types are excluded to simplify the explanation.)

11.3.1 Modeling of Uni-Inline Job Shop

A job shop consisting of uni-inline cells is called a *uni-inline job shop*. Figure 11.8 shows a parameterized event graph model of a job shop consisting of uni-inline cells whose reference model is depicted in Fig. 11.3. It is a simplified version of the uni-inline job shop model introduced in Chapter 5 (see Fig. 5.21 in Section 5.5.3) obtained by removing the Arrive and Exit event nodes and the job-type variable JT[u] from the original model in Fig. 5.21. As described in Section 5.5.3 of Chapter 5, the state variables in the model are as follows:

- Q[u]: Stocker queue of cassettes {(j, p)} in the uni-inline cell (u)
- B[u]: I/O-Port queue of cassettes {(j, p)} in the uni-inline cell (u)
- E[u]: number of empty ports (shelves) in the I/O Port of a uni-inline cell (u)
- R[u]: status of the track-in robot of a uni-inline cell (u)

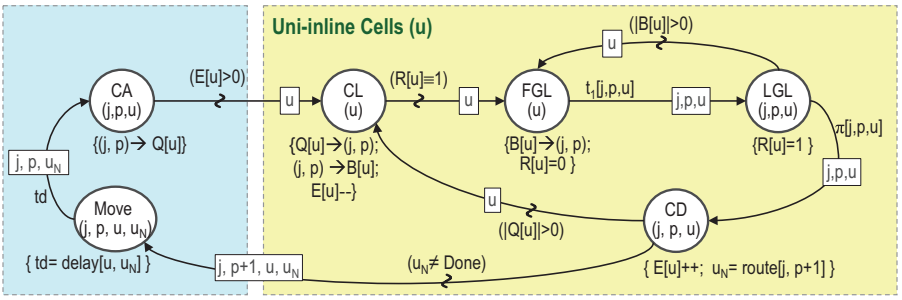


Fig. 11.8. Event graph model of a uni-inline job shop (from Fig. 5.21).

TABLE 11.1. Record Variables Declared for Use in Event Graph Models of Job Shops

Variable		Type/Value	Description
cst	j	int	Job type of the glasses in the cassette
	p	int	Processing step of the glasses in the cassette
	d	string	ID of the equipment for the next processing step of the cassette
	n	int	Number of glasses in the cassette
P[u]	x	0 ~ port capacity	Number of empty ports at the I/O Port of uni-inline cell u or oven v
	rx	0 ~ port capacity	Number of reserved empty ports at the I/O Port
	f	0 ~ port capacity	Number of full-cassette ports at the I/O Port
	e	0 ~ port capacity	Number of empty-cassette ports at the I/O Port
	dr	0 ~ port capacity	Number of depart-reserved-cassette ports at the I/O Port

In Fig. 11.8, the array variables t_1[j, p, u] and π[j, p, u] denote the cycle time for a cassette of glasses and the flow time of the cell, respectively. The en-queue operation is expressed as (j, p) → Q[u] and de-queue operation as Q[u] → (j, p). The arrays delay[] and route[] are used for obtaining the move time-delay (td) and the next route (u_n).

In order to provide a more compact model description, the cassette object and the port object in a uni-inline cell (u) are declared as record variables cst and P[u] as summarized in Table 11.1. The admissible states of a port in the I/O Port are: occupied by a *full cassette* (f); occupied by an *empty cassette* (e); not occupied but reserved (rx); not occupied nor reserved (x); depart-reserved (dr). An empty cassette may contain processed glasses (but no unprocessed ones). The following port-state update functions are used in updating the state of a port (e.g., P[u](f →e): If (P[u].f>0){ P[u].f—; P[u].e++})

- P[u](rx →f): change the state of a port from reserved to full-cassette
- P[u](f →e): change the state of a port from full-cassette to empty-cassette
- P[u](e →x): change the state of a port from empty-cassette to no-cassettes
- P[u](x →rx): change the state of a port from no-cassette to reserved

Figure 11.9 shows an encapsulated event graph model of the uni-inline job shop obtained from the event graph of Fig. 11.8. In order to encapsulate a group of events into an event object, a mirror event is created for each boundary event at the receiving side. In Fig. 11.8, CL and Move are receiving side boundary events. Thus, their mirror events CL* and Move* are introduced in the encapsulated event graph.

Figure 11.9 employs en-queue functions (e.g., cst→ Q[u]), de-queue functions (e.g., Q[u]→ cst), and port-state update functions (e.g., P[u](f→e)). Also used in the event graph are two job-routing functions:

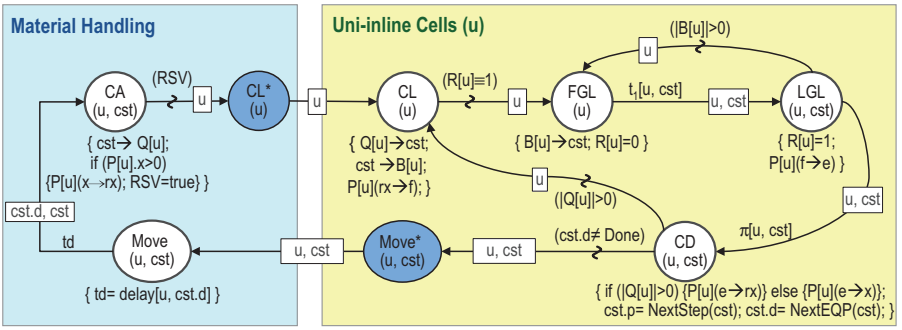


Fig. 11.9. Encapsulated event graph model of the uni-inline job shop in Fig. 11.8.

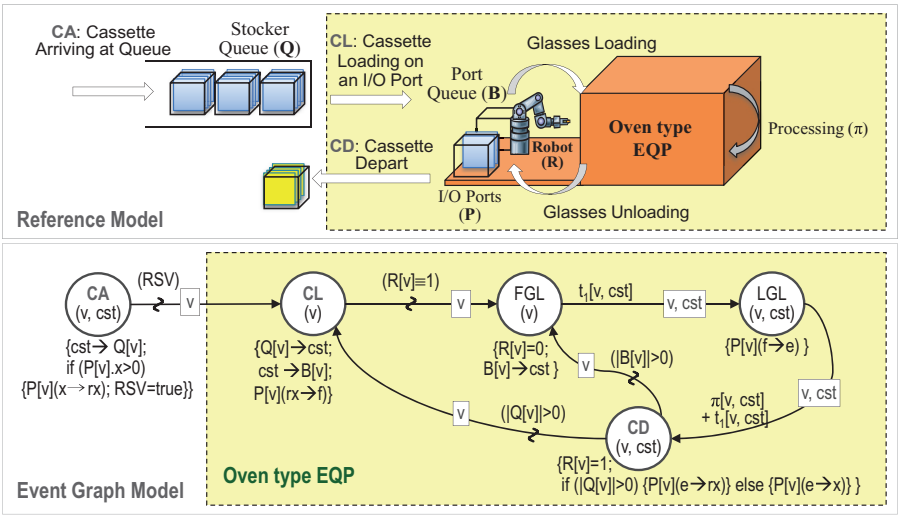


Fig. 11.10. Reference model and event graph model of oven type equipment.

- NextStep (cst) returns the next processing-step ID of a job with job-type cst.j and current processing-step cst.p.
- NextEQP (cst) returns the next equipment ID that will process a cassette having job-type cst.j and processing step cst.p.

11.3.2 Modeling of Oven Type Job Shop

A job shop consisting of oven type equipment is called an *oven type job shop*. Figure 11.10 shows a reference model (reproduced from Fig. 11.5) and event graph model of oven type equipment. As mentioned in Section 11.2.2, the characteristics of oven type equipment are the same as those of the uni-inline cell, except all the glasses in a cassette are loaded together to process all the

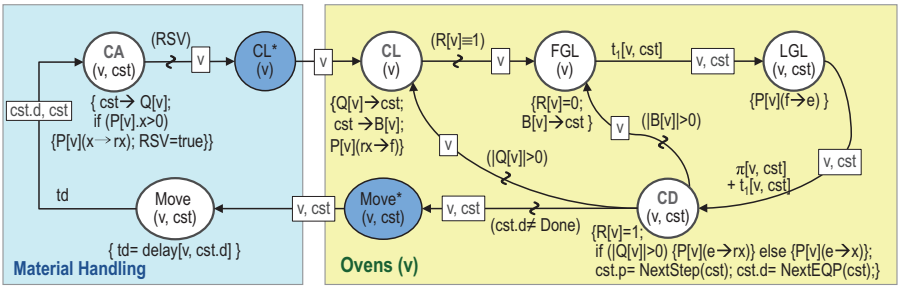


Fig. 11.11. Encapsulated event graph model of a job shop with oven type equipment.

glasses in the cassette at the same time. Thus, in the event graph of oven type equipment, the first glass loading (FGL) event is scheduled by the CD event if the port queue is not empty (as well as by the CL event if the Robot is idle).

Figure 11.11 shows an encapsulated event graph model of a job shop with oven type equipment presented in Fig. 11.10. As with the uni-inline case, two mirror evented CL*(v) and Move*() are introduced in order to encapsulate the event objects. Brief descriptions for the key events in Fig. 11.11 are provided below:

- CA (v, cst): (1) En-queue an arriving cassette cst into Q[v], (2) reserve an empty port if there is one, and (3) schedule a CL event if possible.
- CL (v): (1) De-queue a cassette from Q[v], (2) en-queue cst into B[v], (3) make the reserved port a full port, and (4) schedule an FGL event if Robot is free.
- FGL (v): (1) Set the Robot to busy, (2) de-queue a cassette cst from B[v], and (3) schedule an LGL event to occur after t_1[v,cst].

11.3.3 Modeling of Heterogeneous Job Shop

Figure 11.12 shows an encapsulated event graph model of a heterogeneous job shop consisting of three *event object (EO) models*: Material Handling EO model, Uni-inline EO model and Oven EO model. In general, an encapsulated event graph is a network of EO models. The heterogeneous job shop model in Fig. 11.12 is constructed by joining the two models in Fig. 11.9 and Fig. 11.11 together. All the events used in the heterogeneous job shop model are listed in Table 11.2.

11.3.4 Object-Oriented Event Graph Simulator for Production Simulation

Figure 11.13 shows an encapsulated event graph model of our heterogeneous job shop and its *object-oriented event graph (OOEG) simulator*. The OOEG simulator consists of a simulation coordinator and three *event object (EO)*

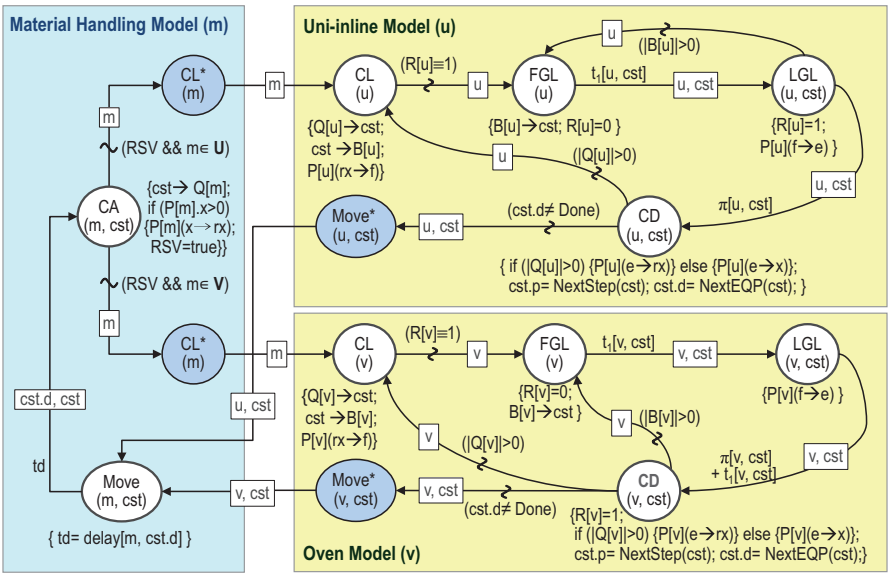


Fig. 11.12. Encapsulated event graph model of a heterogeneous job shop.

TABLE 11.2. Events Used in the Event Graph Model of the Job Shop in Figure 11.12

Type	Name	Full Name	Description
Processing equipment	CL	Cassette Loading	A cassette is loaded (deposited) at a port of a processing equipment
	FGL	First Glass Loading	Track-in robot starts to load the first glass of a cassette
	LGL	Last Glass Loaded	Track-in robot finishes loading a cassette
	CD	Cassette Departure	A cassette departs after the last glass is unloaded
Handling system	Move	—	A cassette starts to move to the destination
	CA	Cassette Arrival	A cassette arrives at the material handling queue

simulators, one EO simulator for each EO model in the encapsulated event graph model. An EO simulator does not schedule the local events by itself. Instead, it sends the enabled local event e to the Coordinator via the public function ScheduleLocalEvent (e). Then, the Coordinator (1) stores the local events in the LEL (local event list), (2) selects a next local event from the LEL, and (3) sends it back to the respective EO simulator via the ExecuteLocalEvent (e) function.

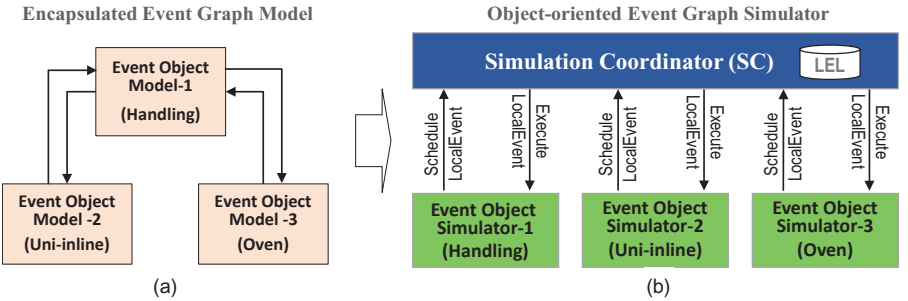


Fig. 11.13. (a) An encapsulated event graph model and (b) its object-oriented event graph production simulator.

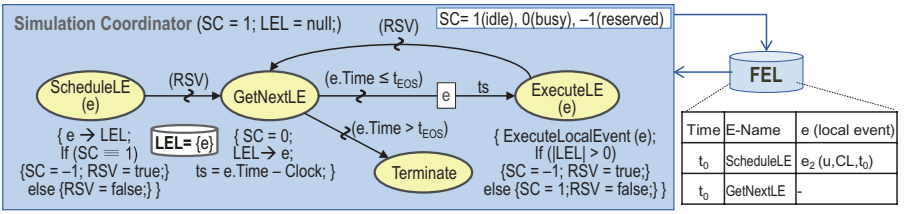


Fig. 11.14. Event graph model of the simulation coordinator.

11.3.4.1 Simulation Coordinator Figure 11.14 shows an event graph model of the simulation coordinator (SC or Coordinator). It is a kind of single server system model consisting of a buffer LEL and four event nodes: (1) ScheduleLE, which is generated by the EO simulators, acts as the Arrival event in a single server system; (2) GetNextLE gets a local event e from the priority queue LEL (LEL→ e) and loads it on the Coordinator (SC = 0) to schedule an Unload event ExecuteLE if the simulation time does not exceed the EOS (end-of-simulation) time; (3) ExecuteLE unloads the job (local event e) from the Coordinator to send it to its EO simulator via the public function ExecuteLocalEvent (e) and schedules a GetNextLE event if necessary. The FEL (future event list) stores event records consisting of event-time, event-name, and local event information.

Listed in Fig. 11.15 are the main program and event routines (in a pseudo-code form) of our simulation coordinator. The main program in Fig. 11.15(a) is a standard implementation of the next-event methodology described in Chapter 4 (Section 4.7.4), except that it has the two public functions: ScheduleLocalEvent (e) and ExecuteLocalEvent (e). The ScheduleLocalEvent function is invoked from EO simulators to schedule a ScheduleLE event into the FEL (see Fig. 11.15). The ExecuteLocalEvent function is invoked from the event routine Execute-ExecuteLE-Routine to make the respective EO simulator execute its next event (Fig. 11.17 provides more details).

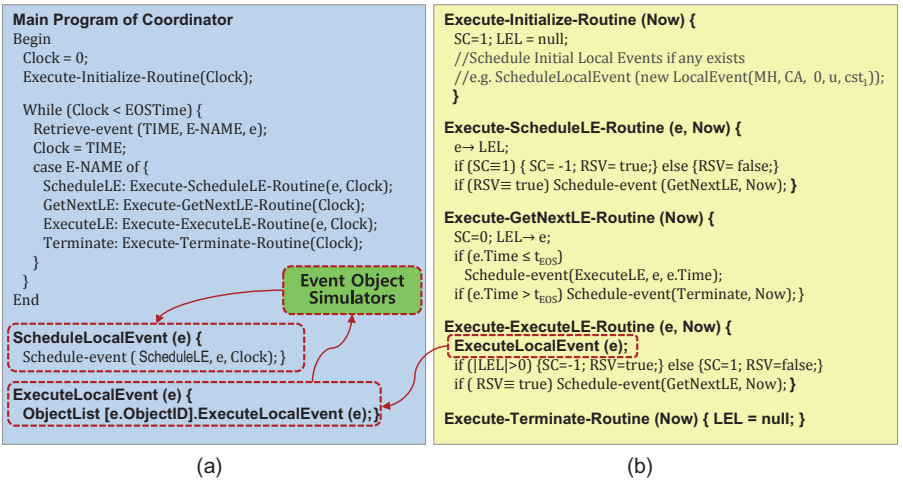


Fig. 11.15. Simulation coordinator: (a) main program and (b) event routines.

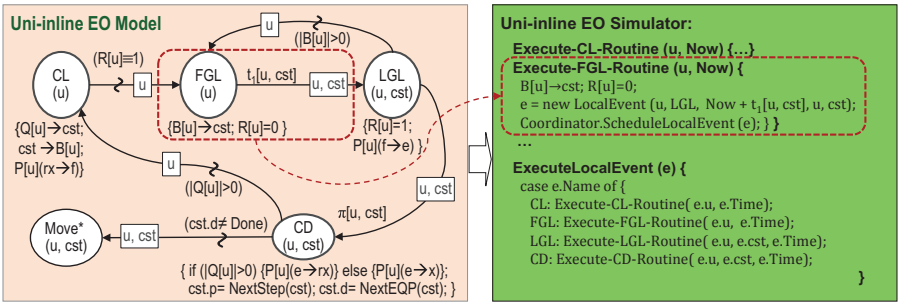


Fig. 11.16. Uni-inline EO simulator for the uni-inline EO model in Fig. 11.12.

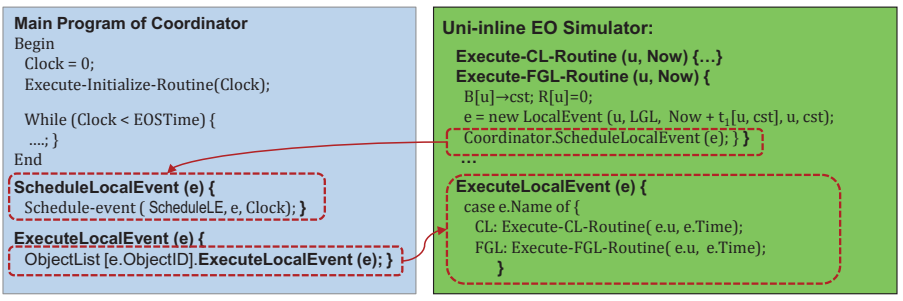


Fig. 11.17. Interactions between the Coordinator and EO simulators.

11.3.4.2 Event Object Simulator Figure 11.16 shows how the uni-inline EO simulator is constructed from the uni-inline EO model of Fig. 11.12. An event routine is defined for each event node in the uni-inline EO model. In the case of the FGL event, for example, (1) the state variables are updated as B[u]→ cst; R[u] = 0; and (2) the next event LGL is scheduled by invoking the Coordinator.ScheduleLocalEvent function. The main routine of the EO simulator is defined as the ExecuteLocalEvent function, which is invoked from the Coordinator. Other EO simulators are implemented the same way.

11.3.4.3 Interaction between Simulation Coordinator and EO Simulators Figure 11.17 shows the interactions between the Coordinator and EO simulators. The event routine Execute-FGL-Routine of the uni-inline EO simulator sends a local event e to the Coordinator by calling the function Coordinator.ScheduleLocalEvent, which will store the local event into the LEL of the Coordinator. On the other hand, the Coordinator will invoke the function ObjectList [e.ObjectID].ExecuteLocalEvent to send the next local event e to the EO simulator that has an ID equal to e.ObjectID.

11.4 INTEGRATED SIMULATION OF A FLAT PANEL DISPLAY FAB

This section presents an object-oriented approach to developing a Fab simulator for integrated simulation of production and material handling in FPD Fab.

11.4.1 Modeling of Job Shop for Integrated Simulation

Figure 11.18 shows a logical structure of a job shop equipped with an automated material handling system (AMHS). Now the job shop consists of (1) a production system having uni-inline cells and oven type equipment and (2) an AMHS composed of a number of inline stockers connected by conveyor segments. An example layout of an AMHS-equipped job shop was presented in Section 11.2 (see Fig. 11.2). The job shop denotes a FPD (flat panel display)

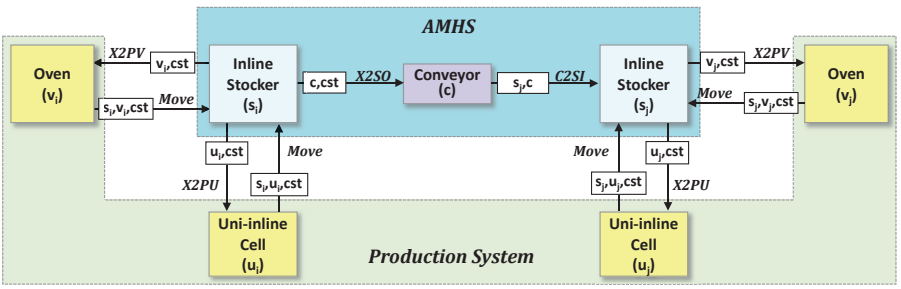


Fig. 11.18. Logical structure of AMHS-equipped job shop.

TABLE 11.3. Attributes of the Cassette Data Object "cst"

Variable		Type/Value	Description
cst	j	int	Job type of the glasses in the cassette
	p	int	Processing step of the glasses in the cassette
	n	int	Number of glasses in the cassette
	d	ID (string)	ID of the equipment for the next processing step of the cassette
	a	ID	ID of the equipment where the cassette is scheduled to enter after the current equipment
	b	ID	ID of the equipment where the cassette stayed before entering this equipment
	c	ID	ID of the equipment where the cassette stays currently
	dp	{B, PU, PV, SO}	type of drop point if it requests a crane for a movement
	pp	{B, PU, PV,SI}	type of pick-up point if it request a crane for a movement
	r	List of IDs	route information of the cassette in the form of array which contains equipment IDs

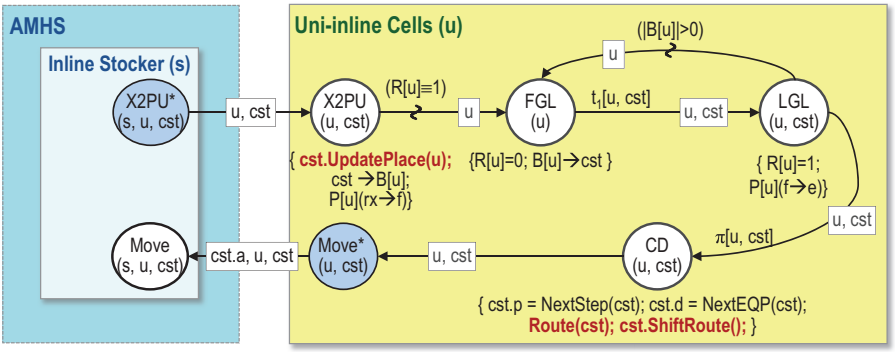


Fig. 11.19. Encapsulated event graph model of AMHS-equipped uni-inline job shop.

Fab and the entities in the system are cassettes containing glasses. In a typical TFT-LCD (thin-film transistor liquid crystal display) Fab, 15 or 24 glasses are stored in a cassette. In order to support the new functionalities of the cassette object, a number of new attributes are added to the data object cst (of Table 11.1) as listed in Table 11.3.

Figure 11.19 shows an encapsulated event graph model of an AMHS-equipped job shop consisting of uni-inline cells. It is an enhanced version of the encapsulated event graph model of the uni-inline job shop presented in Section 11.3.1 (see Fig. 11.9) with the following changes: (1) the name of the cassette-load event is changed to X2PU(u, cst) from CL(u); (2) a function cst.UpdatePlace(u) replaces the de-queue operation Q[u]→ cst at the state

update of the X2PU event node; (3) two functions Route(cst) and cst.ShiftRoute() are introduced at the state update of the CD event node; (4) the scheduling arc from CD to CL (i.e., X2PU) is removed; and (5) the arc condition $u_N \neq$ Done on the scheduling arc from CD to Move is deleted.

As depicted previously in Fig. 11.7 (Section 11.2.3), the AMHS in a Fab is controlled by MCS (material control systems) and equipment dispatching is handled by RTD (real-time dispatching) systems. In Fig. 11.19, NextStep() and NextEQP() are RTD functions; Route(), ShiftRoute(), and UpdatePlace() are MCS functions.

- Route (cst) finds the route of a cassette from the current location cst.c to the destination location cst.d. The route which is a sequence of equipment IDs is stored in the list cst.r. Note that cst.r [0] = cst.c by definition.
- cst.ShiftRoute () shifts the cst.r by one: cst.r[i] = cst.r[i+1] for i = 0, 1 · · ·
- cst.UpdatePlace (m) updates the current (c), before (b), and after (a) places of the cassette: {cst.b = cst.c; cst.c = m; cst.a = cst.r [1]}.

An encapsulated event graph model of an AMHS-equipped job shop consisting of oven type equipment is given in Fig. 11.20. As with the uni-inline job shop case, it is also an enhanced version of the encapsulated event graph model of the oven type job shop presented in Section 11.3.2 (see Fig. 11.11) with the following changes: (1) the name of the cassette-load event is changed to X2PV(v, cst) from CL(v); (2) a function cst.UpdatePlace(v) replaces the de-queue queue operation Q[v]→ cst at the state update of the X2PV event node; (3) two functions Route(cst) and cst.ShiftRoute() are introduced at the state update of the CD event node; (4) the scheduling arc from CD to CL (i.e., X2PV) is removed; and (5) the arc condition "$u_N \neq$ Done" on the scheduling arc from CD to Move is deleted.

Table 11.4 gives all the events introduced in Section 11.4 to describe the integrated simulation models (in addition to the events listed in Table 11.2 in

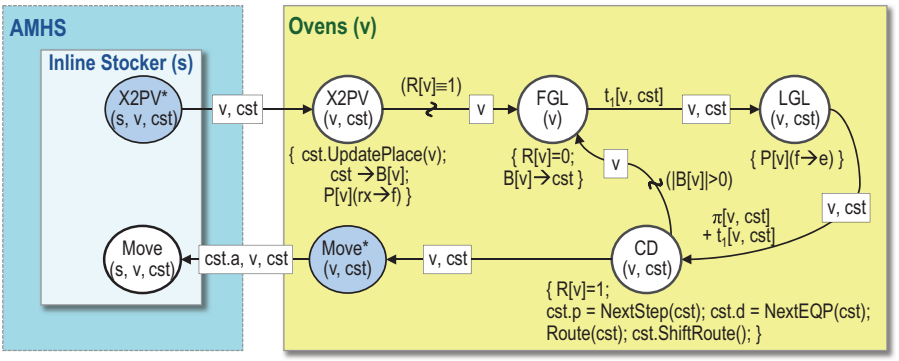


Fig. 11.20. Encapsulated event graph model of AMHS-equipped oven type job shop.

TABLE 11.4. Events Introduced in the Integrated Fab Simulation Model

Type	Name	Full Name	Description
Conveyor operation	SOC	Start of Convey	Cassette starts moving on the conveyor
	EOC	End of Convey	Cassette arrives at the end of the conveyor
	C2SI	Conveyor to Stocker In-port	Cassette moves into the stocker in-port and reserves a crane
Cassette deposition	X2SO	X to Stocker Out-port	A cassette is deposited (dropped) at a Stocker Out-port
	X2PU	X to Uni-inline Port	A cassette is deposited (dropped) at a Uni-inline Port
	X2PV	X to Oven Port	A cassette is deposited (dropped) at a Oven Port
	X2B	X to Buffer of stocker	A cassette is deposited (dropped) at a stocker Buffer
Cassette pick up by crane	SI2X	Stocker In-port to X	Crane starts picking up a cassette from a Stocker In-port.
	PU2X	Uni-inline Port to X	Crane starts picking up a cassette from a Uni-inline Port.
	PV2X	Oven Port to X	Crane starts picking up a cassette from a Oven Port.
	B2X	stocker Buffer to X	Crane starts picking up a cassette from a stocker Buffer.
Crane ready	SI2Xr	SI2X ready	Crane becomes ready for SI2X.
	PU2Xr	PU2X ready	Crane becomes ready for PU2X.
	PV2Xr	PV2X ready	Crane becomes ready for PV2X.
	B2Xr	B2X ready	Crane becomes ready for B2X.
Crane free	CU	Crane Unloaded	Crane finishes unloading (dropping) a cassette at a destination point.
	CI	Crane Idle	Crane becomes idle and ready for a next crane Request.

Section 11.3.3). There are 17 events grouped into five types: (1) conveyor operation events; (2) cassette drop events; (3) cassette pick-up events; (4) crane-ready events, and (5) crane-free events.

11.4.2 Modeling of Conveyor Operation

Figure 11.21 shows a reference model of conveyor operation. A cassette placed at the out-port of a stocker (SO) is moved by the conveyor to its end (CQ:

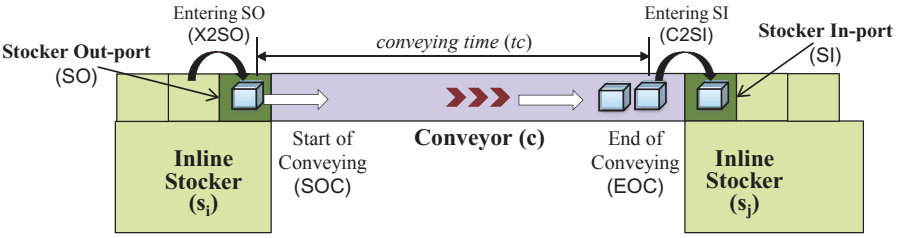


Fig. 11.21. Reference model of conveyor operation.

TABLE 11.5. State Variables Related to Conveyor Operation

Variable	Value	Description
SO[s, c]	–1: reserved, 0: occupied, 1: empty	Status of an output port of the inline stocker *s* connected to conveyor *c*
SI[s, c]	–1: reserved, 0: occupied, 1: empty	Status of an input port of the inline stocker *s* connected to conveyor *c*
CQ[c]	List of cassettes {*cst*}	list of conveyed cassettes waiting at the end of the conveyor

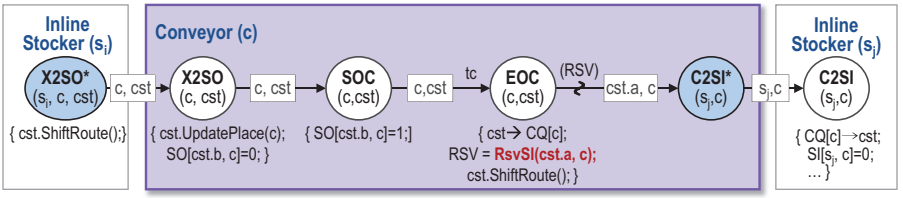


Fig. 11.22. Encapsulated event graph model of conveyor operation.

conveyor queue) where the cassettes are accumulated. The cassette at the very end of the conveyor is transferred to the in-port of another stocker (SI). There are four events related to the conveyor operation: X2SO (transfer from somewhere to SO), SOC (start of convey), EOC (end of convey), and C2SI (from conveyor to SI). The state variables related to conveyor operation are summarized in Table 11.5.

Figure 11.22 shows an encapsulated event graph model of conveyor operation. Since the operation of conveyor c starts from the out-port of stocker s_i, the stocker out-port SO[s_i,c] is treated as a part of the conveyor. When entering the conveyor model, the values of the parameter cst are {cst.c ≡ s_i; cst.a ≡ c; cst.r [0] ≡ c; cst.r [1] ≡ s_j; etc.}.

The first conveyor event is X2SO where (1) the places of the cassette are updated as {cst.b = cst.c ≡ s_i; cst.c = c; cst.a = cst.r [1] ≡ s_j} by the function cst. UpdatePlace(c); and (2) out-port of Stocker "s_i" is set to "busy (occupied)" (SO [s_i, c] = 0). The second event is "start of convey" SOC where (1) out-port

of Stocker "s_i" is set to "idle (empty)" (SO [s_i, c] = 1); and (2) the EOC event is scheduled to occur after the convey time (tc). The third event is "end of convey" EOC where (1) the record cst is stored in the conveyor queue (cst→ CQ[c]); (2) an in-port of Stocker s_j is reserved by the function RSV = RsvSI (cst.a, c); and (3) the cassette's route is updated as {cst.r [0] = cst.r [1] ≡ s_j; cst.r [1] = cst.r [2]; etc.} by the function cst.ShiftRoute (). The function RsvSI () is defined as follows:

- RSV = **RsvSI (s, c)** {
  If (SI[s, c] > 0) {SI[s, c] = −1; RSV = True}
  else {RSV = False}

11.4.3 Modeling of the Interface between Conveyor and Inline Stocker

Figure 11.23(a) is a reference model of conveyor-in interface between an incoming conveyor c_i and an inline stocker s at the stocker in-port SI[s, c_i]. The conveyed cassettes are accumulated in the conveyed queue CQ[c_i]. Events involved in the convey-in operation are EOC, C2SI (conveyor to stocker in-port transfer), SI2Xr (the crane is ready for a pick-up), and SI2X (the crane picks up a cassette from the stocker in-port). Figure 11.23(b) is a reference model of conveyor-out interface between an outgoing conveyor c_j and the inline stocker s at the stocker out-port SO[s, c_j]. The events involved in the convey-out operation are CU (crane unload), X2SO (cassette deposit at the stocker out-port), and SOC (start of convey).

Figure 11.24 shows an encapsulated event graph model of the convey-in interface operation depicted in Fig. 11.23(a). The event object model conveyor (c_i) is the same the one given in Fig. 11.22. At the C2SI event, the following state updates are made: (1) a cassette is retrieved from the conveyed queue {CQ[c_i]→ cst}; (2) the stocker in-port is set to busy {SI[s, c_i] = 0}; (3) make s the current place (or location) of the cassette and update other places {cst.UpdatePlace(s)};

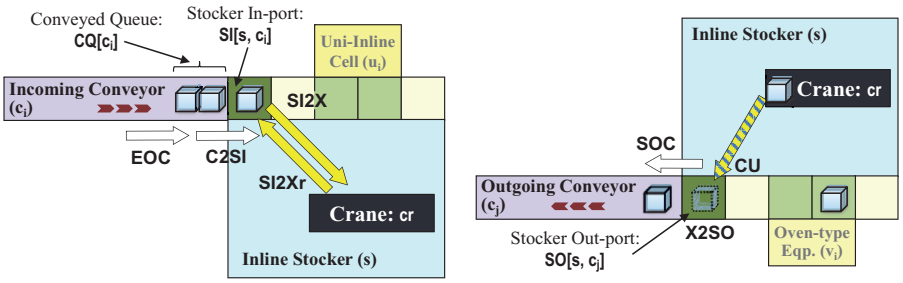


Fig. 11.23. Reference models of (a) conveyor-in interface and (b) convey-out interface.

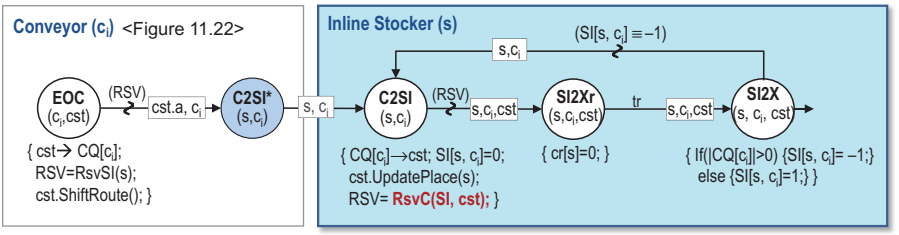


Fig. 11.24. Encapsulated event graph model of convey-in interface.

```
RsvC( ppType, cst): { //ppType: pickup point type = {B, PU, PV, SI}
  if (cr[s] ≡ 1) { // Crane is available
      if (|cst.r| ≡ 1) { // Destination = cst.r[0] ≡ s → drop-point = Stocker Buffer
        cst.a = cst.r[0]; cst.dp = B; // drop-point type = Buffer
      } else if (|cst.r| ≡ 2) { // Destination = cst.r[1] → drop-point = uni-inline or oven in the stocker
        cst.a = cst.r [1];
        if (cst.a ∈ U) { cst.dp = PU; } else if (cst.a ∈ V) { cst.dp = PV; } // drop-point type = uni-inline or oven
      } else {  //destination is not in this inline stocker (thus, select a stocker out-port to go out)
        s = cst.r [0]; c = cst.r [1]; //get the id of the conveyor (c) connected to this inline stocker (s)
        if( SO[s, c] ≡ 1) {
            SO[s, c] = −1 ; //reserve the stocker out-port connected to conveyor c
            cst.a = c; cst.dp = SO;
        } else {  //stocker out-port connected to conveyor c is blocked
          cst.pp = ppType; cst→ CRL[s]; return false;}
      return true; }
  else { // Crane is not available
          cst.pp = ppType;  //set cst's  pickup point to pp
          cst→ CRL[s];  // store the cassette in the CRL (crane request list)
          return false; }
}
```

Fig. 11.25. Pseudocode of the crane reserve function RsvC ().

(4) reserve the crane {RSV= RsvC(SI, cst)}; and (5) an SI2Xr event is scheduled to occur immediately if the crane was successfully reserved. At the SI2Xr event, (1) the crane becomes "ready" by setting its state to "busy" {cr[s] = 0} and (2) an SI2X event is scheduled to occur after tr (time taken to make a *retrieve move* to pick-up point SI). At the SI2X event, (1) the stocker in-port is reserved if there are cassettes in the conveyed queue {If ($|CQ[c_i]| > 0$) {$SI[s, c_i] = -1$}}; (2) the stocker in-port is set to "idle" otherwise {else {$SI[s, c_i] = 1$}}; and (3) a C2SI event is scheduled if the stocker in-port is in "reserved" state {{$SI[s, c_i] = -1$}}. A pseudocode of the function RsvC () is listed in Fig. 11.25.

Figure 11.26 shows an encapsulated event graph model of the convey-out interface operation depicted in Fig. 11.23(b). At the "crane-unload" event CU, a check is made if the cassette drop-point is the stocker out-port ($cst.dp \equiv SO$). If so, an X2SO event is scheduled to occur immediately. The event object model Conveyor (c_j) is the same the one given in Fig. 11.22.

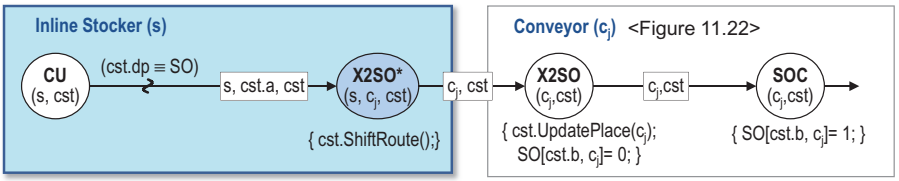


Fig. 11.26. Encapsulated event graph model of convey-out interface.

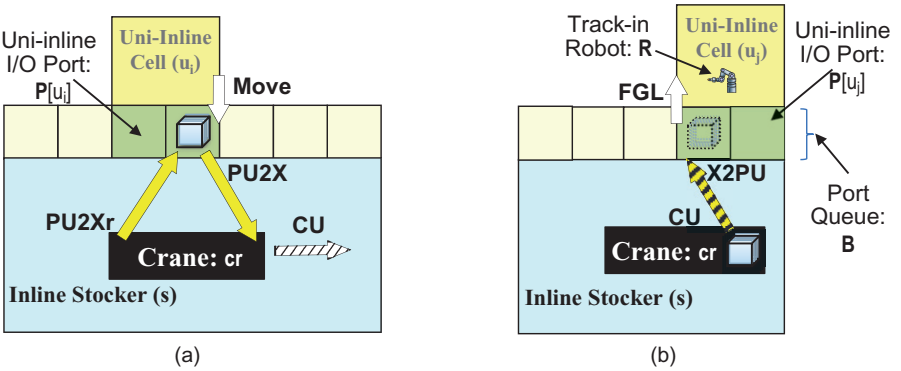


Fig. 11.27. Reference models of (a) pick-up interface and (b) drop interface.

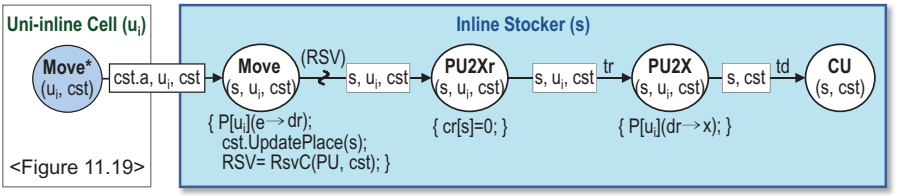


Fig. 11.28. Encapsulated event graph model of pick-up interface with uni-inline cell.

11.4.4 Modeling of the Interface between Uni-inline Cells and Inline Stocker

Figure 11.27(a) is a reference model of pick-up interface between a uni-inline cell u_i and an inline stocker s. A cassette at a uni-inline I/O port $P[u_i]$ is to be picked up by the crane and moved to its destination. Events involved in this pick up operation are Move (start moving to leave the uni-inline cell), PU2Xr (crane is ready for a pick-up), PU2X (crane picks up a cassette from a uni-inline I/O port), and CU (crane unload). Figure 11.27(b) is a reference model of drop interface between a uni-inline cell u_j and the inline stocker s. The events involved in this operation are CU (crane unload), X2PU (cassette deposit at the uni-inline I/O-port), and FGL (first glass loading).

Figure 11.28 shows an encapsulated event graph model of the pick-up interface in Fig. 11.27(a). The event object model Uni-inline Cell (u_i) is the

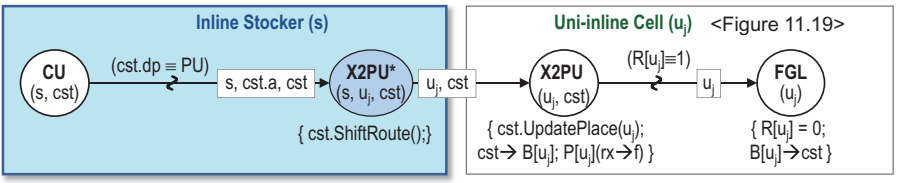


Fig. 11.29. Encapsulated event graph model of drop interface with uni-inline cell.

same the one given in Fig. 11.19. At the Move event, (1) the port state is changed from "empty-cassette" to "depart-reserved" {P[u_i](e→ dr)}; (2) the cassette places are updated {cst.UpdatePlace(s)}; (3) the crane is reserved {RSV= RsvC(PU, cst)}; and (4) PU2Xr is scheduled to occur immediately if the crane was reserved. At the PU2Xr event, (1) the crane is set to "busy" {cr[s] = 0}; and (2) PU2X is scheduled to occur after tr time units. At the PU2X event, (1) the "depart-reserved" port is changed to "no-cassette" {P[u_i](dr→ x)}; and (2) CU is scheduled to occur after a time delay of td (time to move to the drop point).

Figure 11.29 shows an encapsulated event graph model of the drop interface depicted in Fig. 11.27(b). At the "crane-unload" event CU, a check is made if the cassette drop point is a uni-inline port (cst.dp ≡ PU). If so, X2PU is scheduled to occur immediately.

11.4.5 Modeling of the Interface between an Oven and Inline Stocker

The interface mechanism between oven type equipment and an inline stocker is exactly the same as that between uni-inline cells and an inline stocker. Figure 11.30(a) and (b) shows encapsulated event graph models of pick-up interface and deposition interface between oven type equipment and an inline stocker, respectively.

11.4.6 Modeling of Inline Stocker Operation

Figure 11.31 shows a reference model of the crane operation in an inline stocker s (and the next inline stocker s_k). The slots (or shelves) where a cassette is located in the inline stocker are a stocker buffer (B), uni-inline I/O port (PU), oven I/O port (PV), stocker in-port (SI), and stocker out-port (SO). A cassette at a pick point (SI, PU, PV, B) is to be moved to a drop point (SO, PU, PV, B) by the crane. The cassette transport operation is executed in three phases: (1) making the crane ready for picking up a cassette; (2) the crane travels to a pick-up point and picks up a cassette; and (3) the crane transports the cassette to a drop point and drops it. If a cassette is to be transported from the stocker in-port (SI) to an oven I/O port (PV), for example, the stocker operation is executed by: (1) the idle crane gets ready for picking up a cassette at SI (SI2Xr); (2) the crane travels to SI and picks up the cassette (SI2X);

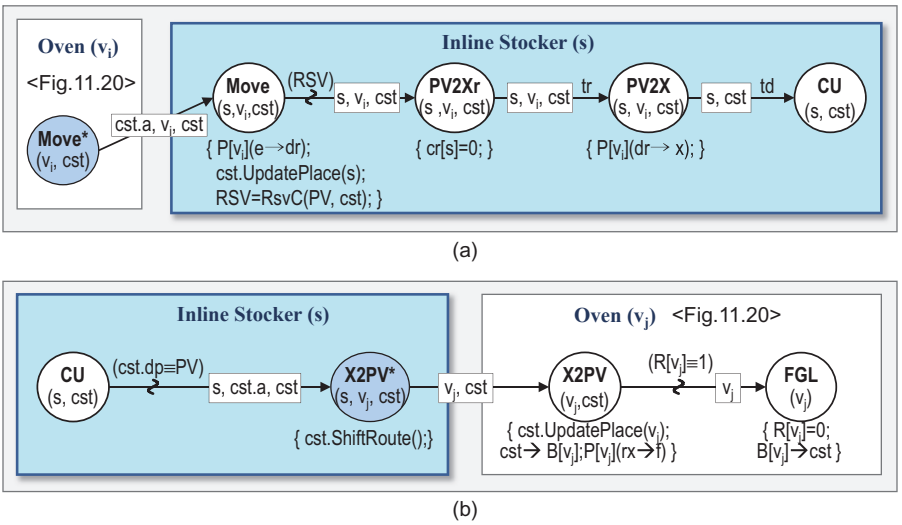


Fig. 11.30. Encapsulated event graph models of (a) pick-up interface and (b) deposition interface with an oven type Eqp.

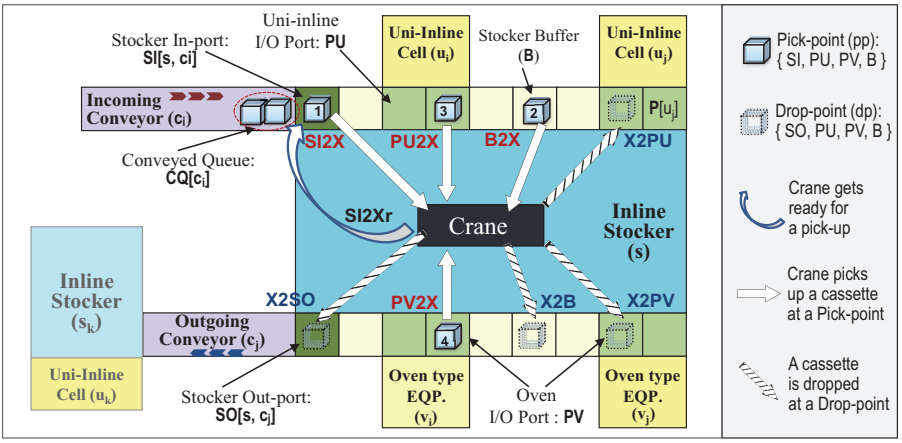


Fig. 11.31. Reference model of inline stocker crane operation.

and (3) the crane moves the cassette to PV and drops it (X2PV) to become idle again. The operation cycles of the crane are captured in the event graph model of crane operation given in Fig. 11.32.

In Fig. 11.32, an operation cycle of the crane starts from the crane idle event CI(s) where the crane state is set to "idle" {cr[s] = 1} and a cassette is selected for transportation {cst = SelectCraneRequest(s)}. The crane operation will be described using the state variables listed in Tables 11.6: (1) the state of the

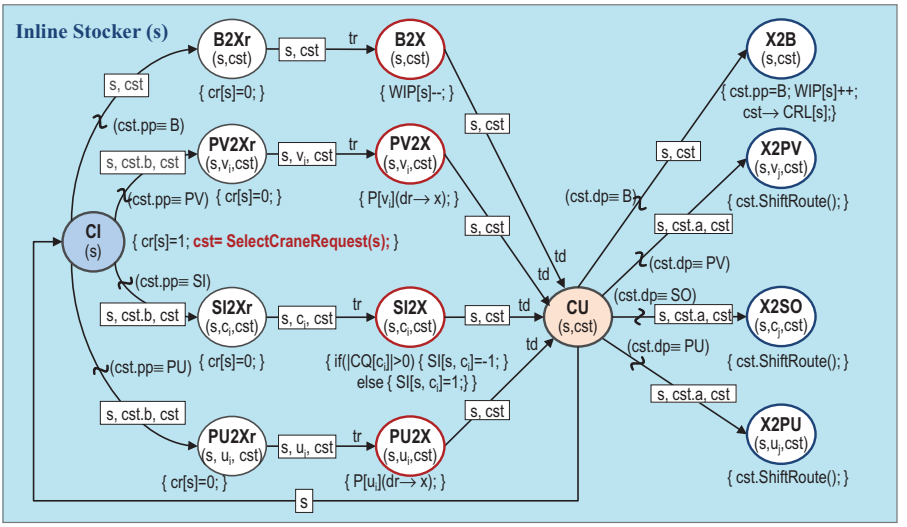


Fig. 11.32. Event graph model of crane operation in an inline stocker.

TABLE 11.6. State Variables Related to Crane Operation

Variable	Type/Value	Description
cr[*s*]	1, 0, –1	*cr*[*s*] = 1 if crane is idle; = 0 if crane is busy; = –1 if crane is reserved
WIP[*s*]	int	Number of cassettes waiting (*stored*) in the buffers of the inline stocker s
CRL[*s*]	{*cst*}	List of *cassettes requested* (*waiting*) for the service of crane at the inline stocker s

crane in the inline stocker is denoted by cr[s]; (2) number of cassettes stored in the stocker buffers is denoted by WIP[s]; and (3) the list of cassettes waiting for transport is denoted by CRL[s]. The function SelectCraneRequest () is defined as follows:

- cst = SelectCraneRequest (s) selects the "best" cassette from CRL[s].

Referring back to Fig. 11.31, four cassettes (cst1~cst4) are waiting for the crane. Let's assume that: (1) cst1 is heading for the uni-inline cell u_j located in the inline stocker s; (2) cst2 is also heading for u_j; (3) cst3 is heading for the oven type equipment v_j located in the inline stocker s; and (4) cst4 is heading for a uni-inline cell u_k located in the next inline stocker s_k. Thus, the values of the state variables are as summarized in Table 11.7. Assuming that cst1 has the highest priority, the event graph model in Fig. 11.32 is executed as follows:

TABLE 11.7. Values of State Variables in the Reference Model of Figure 11.31

Cassette/ Variables	After (.a)	Before (.b)	Current (.c)	Destination (.d)	Pick-point type (.pp)	Drop-point type (.dp)	Route (.r)	Remark
cst1	u_j	c_i	s	u_j	SI	PU	(s, u_j)	*Heading for* u_j
cst2	u_j	c_i	s	u_j	B	PU	(s, u_j)	*Waiting for* u_j
cst3	v_j	u_i	s	v_j	PU	PV	(s, v_j)	*Heading for* v_j
cst4	c_j	v_i	s	u_k	PV	SO	(s, c_j, s_k, u_k)	u_k *belongs to* s_k.
State variable	CRL[s] = {cst1, cst2, cst3, cst4}; CQ[ci] = 2; WIP[s] = 1							

1. At the CI event, the state variables are updated as {cr[s] = 1; cst = cst1} and SI2Xr is scheduled with parameter <s, ci, cst> since cst.pp ≡ SI.
2. At the SI2Xr event, the state variable is updated as {cr[s] = 0} and SI2X is scheduled to occur after tr (time to reach SI).
3. At the SI2X event, the state variable is updated as {SI[s, c_i] = −1} since |CQ| > 0 and CU is scheduled to occur after td (time for a delivery move to P[uj]).
4. At the CU event, X2PU is scheduled with parameter <s, uj, cst> since cst.dp ≡ PU and CI is scheduled with parameter s.

In the previous sections, the inline stocker interfaces with other equipment were modeled as encapsulated event graph models: (1) convey-in interface model in Fig. 11.24; (2) convey-out interface model in Fig. 11.26; (3) uni-inline pick-up interface model in Fig. 11.28; (4) uni-inline drop interface model in Fig. 11.29; (5) oven type equipment interface models in Fig. 11.30. By assembling these interface models into the crane operation model in Fig. 11.32, a completed event graph model of an inline stocker is obtained as shown in Fig. 11.33.

11.4.7 Integrated Fab Simulator

Figure 11.34(a) shows an encapsulated event graph model of our FPD Fab consisting of uni-inline cells, oven type equipment, inline stockers, and conveyors. As described previously (see Section 11.3.4), the encapsulated event graph model can be converted to an OOEG simulator as shown in Fig. 11.34(b) and then eventually implemented into an integrated Fab simulator.

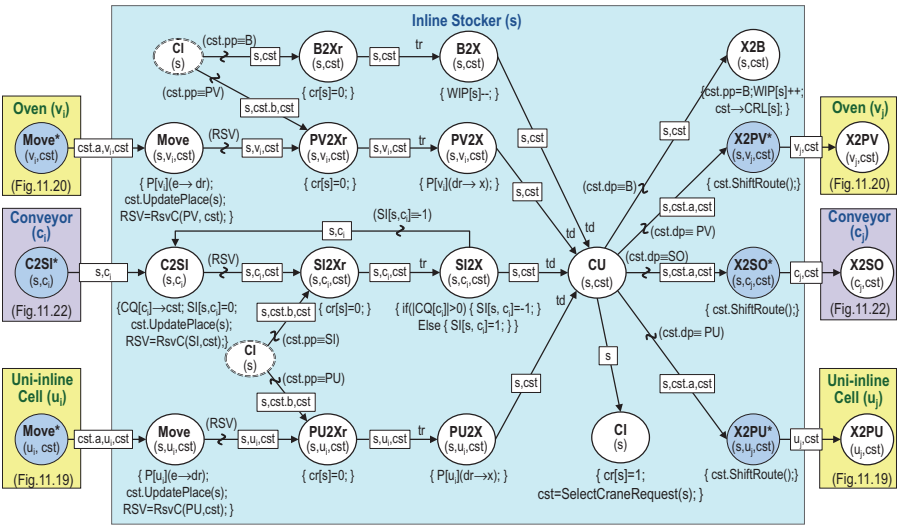


Fig. 11.33. Encapsulated event graph model of inline stocker operation.

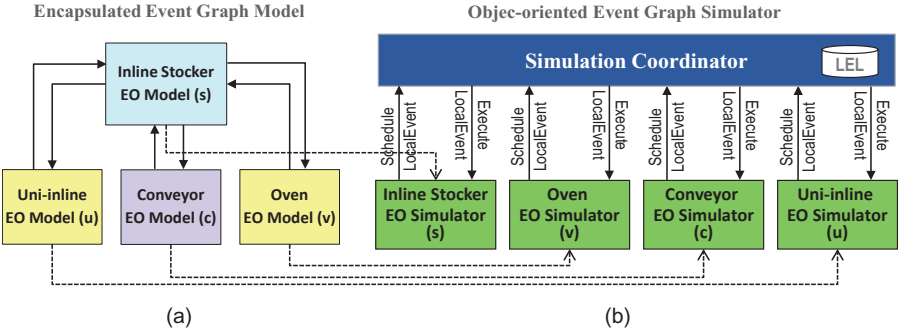


Fig. 11.34. (a) Encapsulated event graph model and (b) OOEG integrated Fab simulator.

11.5 AUTOMATED MATERIAL HANDLING SYSTEMS-EMBEDDED INTEGRATED SIMULATION OF FLAT PANEL DISPLAY FAB

As mentioned earlier, there is a growing need in the FPD industry for an automated material handling systems-embedded integrated simulation where production simulation is carried out together with detailed simulation of AMHS. This section presents a generic framework for an AMHS-embedded integrated Fab simulator where AutoMod models of AMHS are embedded into the integrated Fab simulator of Section 11.4.7. An earlier version of the AMHS-embedded integrated Fab simulator was presented elsewhere [Song et al. 2011].

11.5.1 Concept of AMHS-Embedded Fab Simulation

As depicted in Fig. 11.35, there are four MCM functions that can be used for socket-based communication between our Fab simulator and AutoMod® in which the virtual AMHS is stored. The four AutoMod MCM functions are:

- Send_Msg (msg) is used in Fab Simulator to send a record msg to AutoMod.
- msg= Read_Msg () is used in the Fab Simulator to read msg (sent by AutoMod).
- SendSocketString (msg) is used in AutoMod to send msg to the Fab Simulator.
- msg= ReadSocketString () is used in AutoMod to read msg.

Figure 11.36 shows the concept of AMHS-embedded integrated Fab simulation. In the simulation scheme discussed in the previous section, the SOC (start of convey) event of the Conveyor EO model (Fig. 11.22 in Section 11.4) schedules an EOC (end of convey) event to occur after tc time units for a given value of tc. In the AMHS-embedded simulation scheme consisting of (a) Fab Simulator Conveyor EO model and (b) AutoMod Conveyor model; however, the convey time (tc) has to be determined by AutoMod. Thus, at the SOC event, the Fab Simulator asks AutoMod to perform a virtual conveyor operation. Then, at the end of the virtual convey operation, AutoMod sends the "actual" convey time (tc) back to the Fab Simulator so that the latter schedules an EOC event to occur after tc time units.

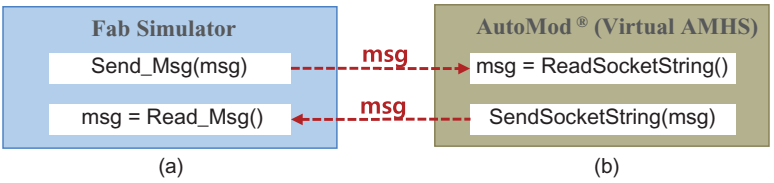


Fig. 11.35. Socket-based communications between a Fab Simulator and AutoMod.

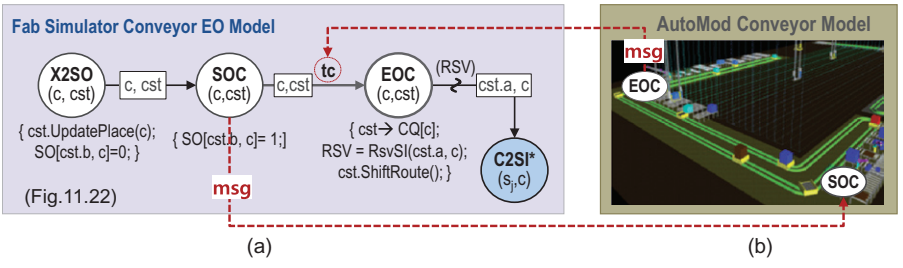


Fig. 11.36. Concept of AMHS-embedded Fab simulation.

11.5.2 Framework of AMHS-Embedded Fab Simulation System

Figure 11.37 shows a framework of AMHS-embedded integrated Fab simulation system realizing the AMHS-embedded simulation concept given in Fig. 11.36. In order for the Convey event SOC to schedule the EOC event with the "actual" convey time (tc) provided by AutoMod, the following steps of actions are taken:

1. At the SOC event of the Conveyor EO Model, the SOC event is stored in the *move-type event list (MEL)* and a convey message Cmsg is sent to AutoMod.
2. At the ExecuteLE event of Coordinator, a time-advance message Tmsg (containing $\mathbf{t_{ta}}$) is built with LEL[0] and sent to AutoMod, and wait until a message msg (containing **tc**) is received from AutoMod.
3. Upon receiving a message (with type = 'Convey'), the Coordinator invokes the function FireSchedulingArc (msg) so that the Conveyor EO Model is allowed to schedule an event.
4. The Conveyor Model asks Coordinator to schedule its destination event.

In Fig. 11.37, the MEL of Conveyor EO Model contains a list of move-type events that have been sent to AutoMod at step 1. Later, when the function FireSchedulingArc(msg) is called at step 3, a matching move-type event is retrieved from the MEL. The data record ***msg*** has the following fields: Type = {Time-advance, Retrieve, Deliver or Convey}, EventName, ObjectID (ID of the sending simulator object), CassetteID, EventTime, Source (current location of the cassette), and Destination (to-be location of the cassette).

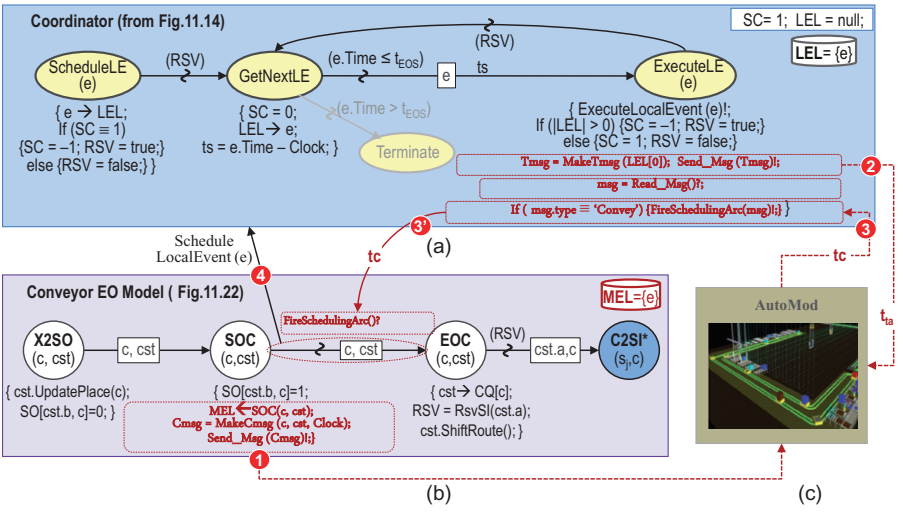


Fig. 11.37. Framework of an AMHS-embedded Fab simulation system.

"#" is used as a delimiter. Among the functions enclosed in dashed-line boxes in the figure, Send_Msg () and Read_Msg () are AutoMod MCM functions introduced in Section 11.5.1. The functions for building a time-advance message Tmsg and convey-move message Cmsg are defined as follows:

```
MakeTmsg (e) { // LEL[0] denotes the first local event
  stored in LEL
msg = "Time-advance#" + e.EventName +"#" + e.ObjectID
  + "# #" + e.EventTime + # #";
return msg;}
MakeCmsg (c, cst, Now) {
msg = "Convey# SOC#" + c + "#" + cst.ID + "#" + Now
  + # #";
return msg;};
```

The *time-advance message* Tmsg is constructed with LEL[0], the first event in the LEL, at step 2. Tmsg contains the time-advance time t_{ta}, which is the event time of LEL[0]. AutoMod is allowed to advance its clock up to the time-advance time t_{ta}. Figure 11.38 shows how the message is processed in AutoMod. In the figure, the retrieve-move message Rmsg and deliver-move message Dmsg are coming from the Inline Stocker model to be explained shortly. For details about AutoMod programming, the reader is referred to AutoMod manuals [Brooks Automation 2003a, 2003b].

An originating event related to the movement of a material handling device, like SOC, is called a *move-type event*. Among the four component models in the integrated Fab simulation system shown in Fig. 11.34, the Inline Stocker model also has move-type events whose delay times have to be determined by AutoMod. The move-type events are: Retrieve events (B2Xr, PV2Xr, SI2Xr,

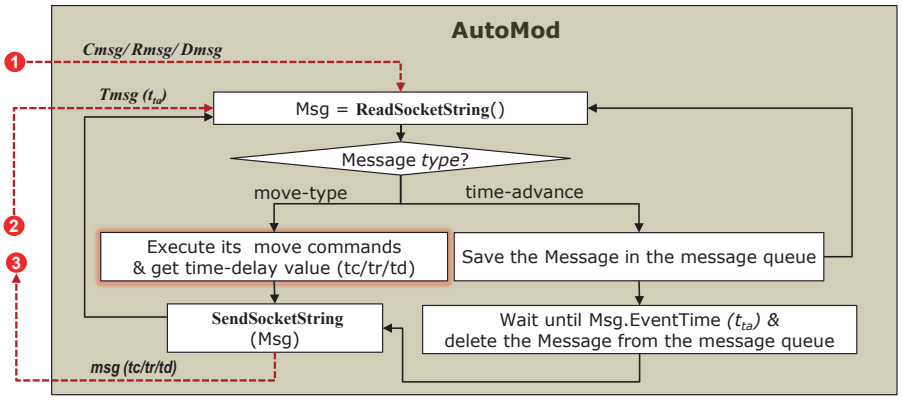


Fig. 11.38. Message processing in AutoMod.

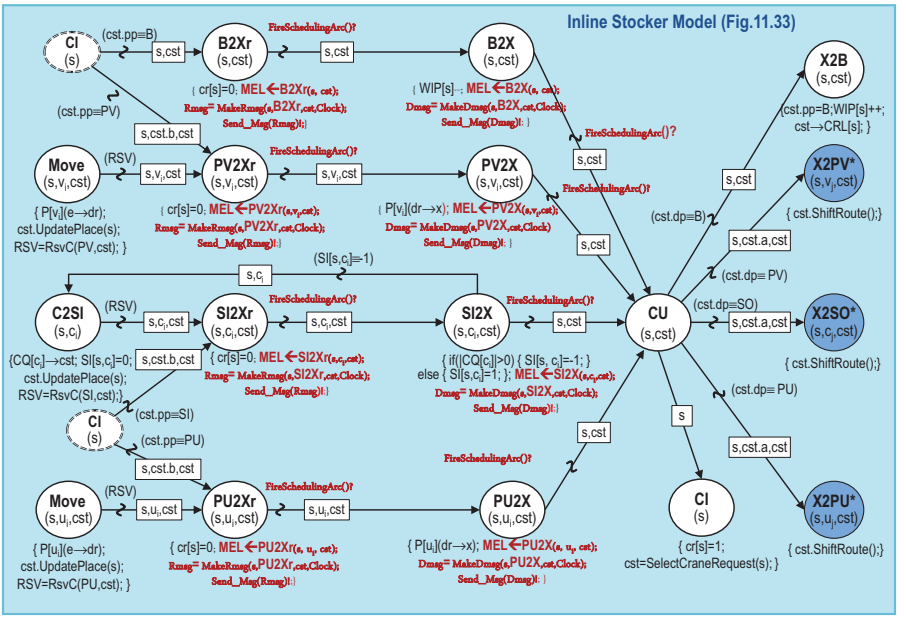


Fig. 11.39. Inline Stocker model modified for AMHS-embedded simulation.

and PU2Xr) and Deliver events (B2X, PV2X, SI2X, and PU2X). Figure 11.39 shows the Inline Stocker model (given in Fig. 11.33) modified for AMHS-embedded simulation.

At each move-type event of the Inline Stocker model, instead of scheduling the destination event to occur after a delay-time (tr or td), the move-type event ID is stored in the MEL and a move-type message (retrieve move or deliver-move message) is constructed and sent to AutoMod. Then, AutoMod returns the delay-time to the Coordinator that in turn pass the delay-time by invoking the FireSchedulingArc () function.

11.5.3 Simulator for AMHS-Embedded Integrated Fab Simulation

Figure 11.40 shows the structure of our AMHS-embedded integrated Fab simulator. Recall that the structure of the integrated Fab simulator was presented in Fig. 11.34 (the structure of the production simulator was given in Fig. 11.13). The processing-type simulators (Uni-inline Simulator and Oven Simulator) in Fig. 11.40 are the same as those in Fig. 11.34 but the handling-type simulators (Conveyor Simulator and Inline Stocker Simulator) in Fig. 11.40 are enhanced to handle communications with AutoMod. In the following, how to implement the AMHS-embedded integrated Fab simulator will be explained.

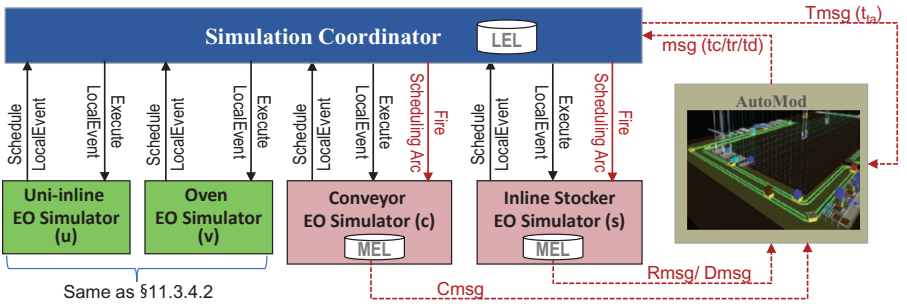


Fig. 11.40. Structure of the AMHS-embedded integrated Fab simulator.

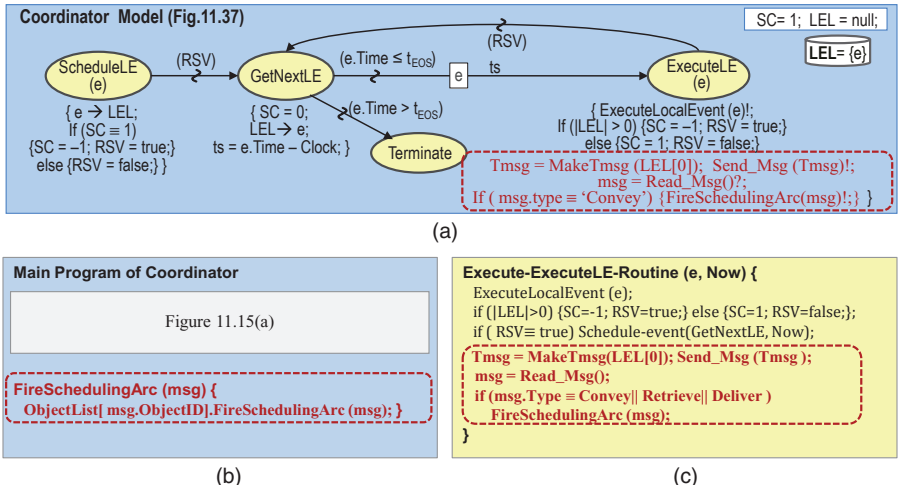


Fig. 11.41. (a) Coordinator model, (b) main program, and (c) ExecuteLE event-routine.

11.5.3.1 Simulation Coordinator of the* AMHS-*Embedded Simulator Figure 11.41 shows the event graph model, the main program, and event routine Execute-ExecuteLE-Routine () of the Simulation Coordinator of the AMHS-embedded integrated Fab simulator. The coordinator main program in Fig. 11.41(b) is obtained by appending the FireSchedulingArc function at the end of the coordinator main program of the production simulator in Fig. 11.15(a). The event routine Execute-ExecuteLE-Routine () in Fig. 11.41(c) is obtained similarly by appending the statements inside the dashed-line box to that of the production simulator in Fig. 11.15(b). The rest of the event routines are the same as those in Fig. 11.15(b).

11.5.3.2 Conveyor EO Simulator Module of the* AMHS-*Embedded Simulator Figure 11.42 shows a pseudocode of the Conveyor EO Simulator

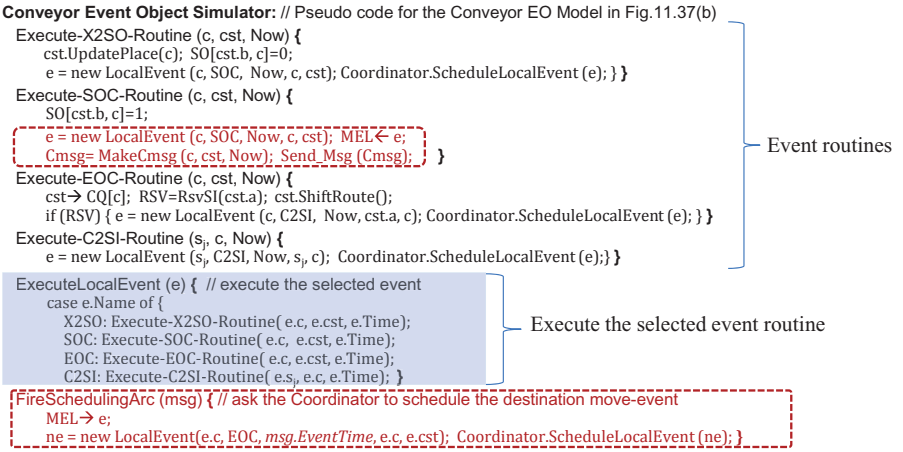


Fig. 11.42. Conveyor EO Simulator module of the AMHS-embedded simulator.

module for the Conveyor EO model in Fig. 11.37(b). The statements enclosed by the dashed-line box in the event routine Execute-SOC-Routine () are responsible for preparing a convey-type message and sending it to AutoMod. At the end of the module, the FireSchedulingArc function is added to handle the Coordinator's instruction to "fire the scheduling-arc from SOC to EOC" so that the EOC event is scheduled with the time-delay value provided by AutoMod.

11.5.3.3 Inline Stocker EO Simulator Module of the* AMHS-*Embedded Simulator Figure 11.43 shows a pseudocode of the Inline Stocker EO Simulator module for the Inline Stocker EO model in Fig. 11.39. The event routines and the function FireSchedulingArc are defined the same way as in the case the conveyor model in Fig. 11.42. In the FireSchedulingArc function, the matching move-type event is retrieved from MEL and the scheduling-arc is fired so that the destination move-event is scheduled with the time-delay value provided by AutoMod.

11.5.4 IFS®

The authors and their students have developed a prototype software system called IFS® for an AMHS-embedded integrated Fab simulation. Figure 11.44 shows the structure of the software system IFS® consisting of three modules: (1) an Integrated Fab Simulation module, which is the main part of IFS®; (2) a virtual AMHS module (AutoMod®) responsible for graphical simulation of the AMHS; and (3) a layout generation module.

The internal workings of the Integrated Fab Simulation (IFS) module are as described in Section 11.4 and the technical details of AutoMod® are

Inline Stocker Event Object Simulator: // Pseudo code for the Inline Stocker EO Model in Fig.11.39

```
Execute-B2Xr-Routine (s, cst, Now) {// event routines
    cr[s]=0;
    e = new LocalEvent (s, B2Xr, Now, s, cst);  MEL ← e;
    msg= MakeRMsg (s, B2Xr, cst, Now);  Send_Msg (msg); }
Execute-B2X-Routine (s, cst, Now) {….}
  ….
```

Event routines

```
ExecuteLocalEvent (e) {
    case e.Name of {
        B2Xr: Execute-B2Xr-Routine( e.s, e.cst, e.Time);
        B2X: Execute-B2X-Routine( e.s,  e.cst, e.Time);
        …. }
```

Execute the selected event routine

```
FireSchedulingArc (msg) { // fire the move-event whose delay-time was provided by AutoMod
    e = MEL.RetrieveEvent (msg.EventName); // retrieve the matching event from MEL (move-type event list)
    case msg.EventName of { // build the next-local-event record nle
        PU2Xr:  nle = new LocalEvent(e.s, PU2X, msg.Time, e.s, e.u_i, e.cst);
        …..
        B2X, PV2X, SI2X, PU2X:  nle = new LocalEvent(e.s, CU, msg.Time, e.s, e.cst);
    };
    Coordinator.ScheduleLocalEvent (nle); }
```

Fig. 11.43. Inline Stocker EO Simulator module of the AMHS-embedded simulator.

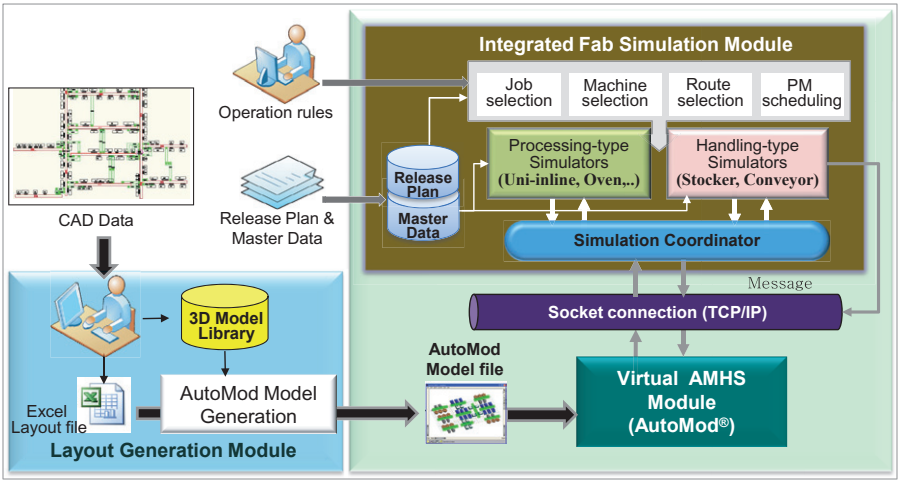


Fig. 11.44. Structure of the AMHS-embedded integrated Fab simulator.

available in the literature [Brooks Automation 2003a, 2003b]. The user has to provide the IFS module with (1) master data regarding the equipment and processes specifications in the Fab; (2) release plans for each day's production; and (3) various operation rules for RTD (real-time dispatching) and MCS (material control system).

The three-dimentional (3D) geometric and kinematic models of the equipment and facilities in the Fab layout are constructed in the layout generation module. In practice, the original 3D Fab layout data generated by using a CAD (computer-aided design) system, called *CAD Data*, are not suitable to be used

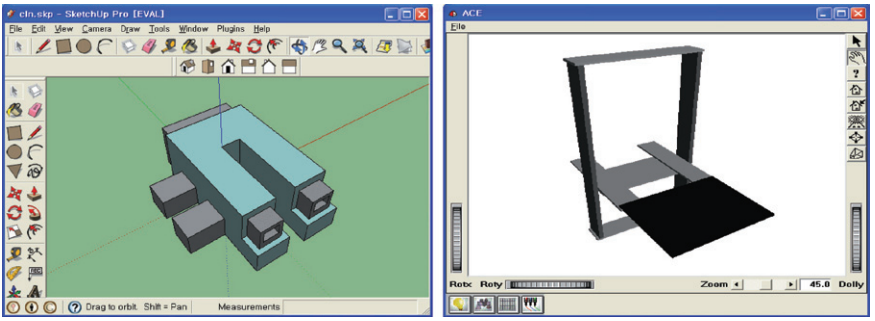


Fig. 11.45. Building a library of 3D models (uni-inline call and stacker crane).

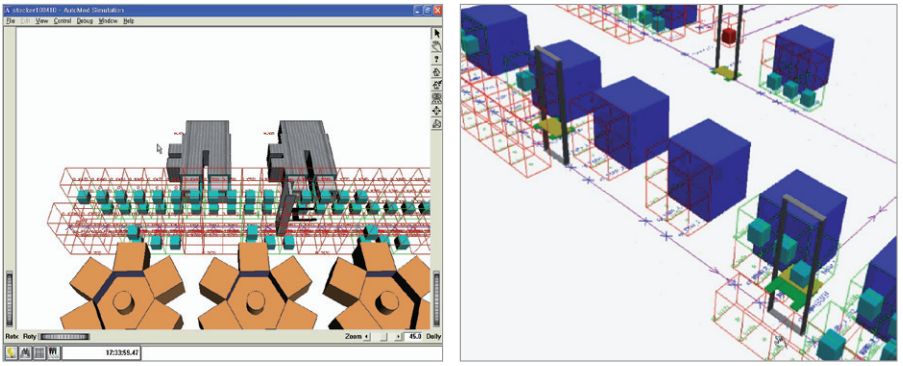


Fig. 11.46. Examples of 3D layout in IFS®.

as an AutoMod model file. In the layout generation module, the user manually defines a 2D layout model using Excel from the 3D CAD data.

At the same time, a library of 3D models of the equipment and devices is built employing CAD systems as depicted in Fig. 11.45. The left image in the figure is a 3D model of a uni-inline cell and the right image is a stacker crane in the inline stockers. The AutoMod model generation program will merge the 2D layout model (Excel file) and the 3D models (CAD data) to build an AutoMod model file. Examples of 3D layout of a hypothetical LCD Fab are shown in Fig. 11.46. A free copy of IFS® may be found in the official website of this book (http://VMS-technology.com/Book/IFS).

CHAPTER 12

Concepts and Applications of Parallel Simulation

Knowledge is a process of piling up facts; wisdom lies in their simplification.

—M. Fischer

12.1 INTRODUCTION

The terminologies used in this section are mostly from Fujimoto [2000], but the term *parallel simulation* is used somewhat differently. In Fujimoto, a simulation that executes on a set of computers confined to a single room is called a *parallel simulation*, whereas a *distributed simulation* executes on machines that are geographically distributed. In this book, a *parallel simulation* is defined as a simulation composed of a "collection of sequential simulations that exchange messages with each other" regardless of whether the computers are confined to a room or geographically distributed. In Fujimoto, a *sequential simulation* in the definition is referred to as a *logical process* (LP). A logical process has its own simulation clock. The key issue in parallel simulation is time synchronization to ensure that events are processed in a timestamp order when the logical processes are executed.

According to Fujimoto [2000], there exist two synchronization approaches: (1) one is *conservative synchronization*, which enforces all events be processed in timestamp order all the time; (2) the other is *optimistic synchronization* in which an out-of-order processing is allowed but the errors are recovered. Among the popular conservative synchronization methods are the centralized barrier method and the (distributed) null message method. The centralized barrier method, in which a controller LP is employed to implement the barrier for the simulator LPs, is as follows:

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.

1. The controller LP: determines when a barrier is reached and waits for messages from the simulator LPs. When a message is received from each simulator LP, it sends a "grant" message to the selected simulation LPs to release the barrier.
2. Simulator LPs: send a message to the controller LP and wait for a reply. When a "grant" message is received from the controller LP, safe events are executed.

In fact, the parallel simulation method with a centralized barrier has already been utilized quite extensively in this book. The state graph simulator introduced in Chapter 9 (Fig. 9.34 in Section 9.6.1) is a parallel simulation system where the Sync Manager plays the role of the controller LP. The object-oriented event graph simulator in Chapter 11 (Fig. 11.34 in Section 11.4.7) is also a parallel simulation system with the Simulation Coordinator playing the role of the controller LP.

A high-level architecture (HLA) is a general-purpose architecture for parallel simulation systems. Using HLA, computer simulations can interact with other computer simulations regardless of the computing platforms. The interaction between simulations is managed by a run-time infrastructure (RTI).

This chapter is organized as follows. A framework for direct workflow simulation based on parallel simulation is presented in Section 12.2. A brief description of HLA/RTI is provided in Section 12.3, and an implementation example of parallel simulation HLA/RTI is introduced in Section 12.4.

12.2 PARALLEL SIMULATION OF WORKFLOW MANAGEMENT SYSTEM

This section presents a parallel simulation method with which the enactment service processes of a workflow management system (WfMS) can be simulated directly, i.e., without converting to a simulation model such as Petri nets.

12.2.1 Enactment Service Mechanism of WfMS

The basics of enactment service are briefly described using Fig. 12.1. For each instance of workflow, a process instance is created from its process definition model (PDM). Figure 12.1 shows a Process Instance consisting of seven activities including Start and End activities. The completion of activity W1 will enable the two succeeding activities W2 and W3. At this point, the enactment server would provide the following sequence of enactment services:

1. Generates new workitems for the newly enabled activities W2 and W3.
2. Sends out the new workitems W2 and W3 to their respective work-list handlers to be processed by the participant.

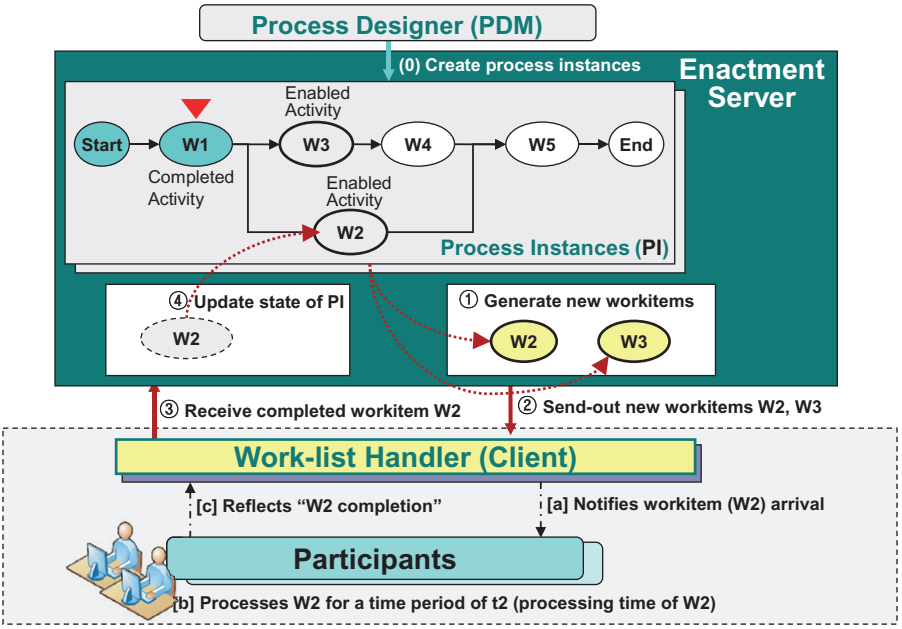


Fig. 12.1. Enactment service mechanism of workflow management system (WfMS).

3. Receives the completed workitem W2 from the work-list handler.
4. Updates the state of the PI such that W2 becomes a completed activity.

Upon receiving the new workitem W2, (a) the work-list handler notifies it to its participant, and the participant (b) works on the workitem W2 for a time period of t2 and (c) reflects the results at the work-list handler so that the completed workitem is returned back to the enactment server. The communications (i.e., sending new workitems and receiving completed workitems) between the enactment server and work-list handlers (or application handlers) are made based on the interface standards provided by a workflow management coalition (WfMC). For this purpose, the enactment server has a type of internal data called *workflow relevant data* that can be manipulated by work-list handlers and other applications via a set of standard API (application program interface) functions [WfMC 1995, 1998].

12.2.2 Framework of Parallel Simulation of WfMS

Shown in Fig. 12.2 are interactions among the software modules in the parallel simulation system. In this framework, data exchanges between the enactment server (Server) and participant simulators (Simulators) are made through the synchronization manager (Sync Manager). Thus, the role of the Sync Manager is to mediate the communications between the Server and Simulators while

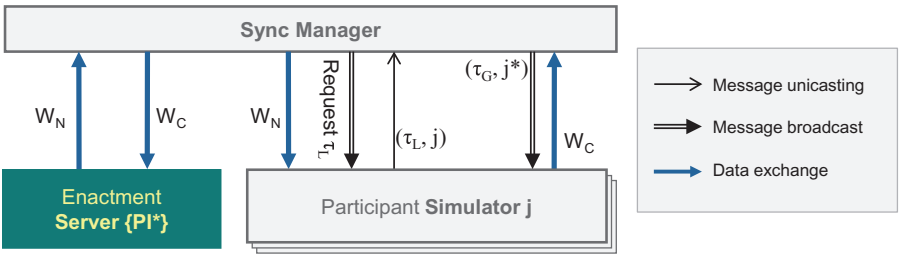


Fig. 12.2. Interactions among the workflow simulation modules.

managing time synchronization. Here, the Sync Manager has all connection information of the Simulators that handle all new and completed workitems, and thus it handles all the "standard" enactment services provided by the Server.

At the beginning of an enactment service cycle, the Server generates new workitems $\{W_N\}$ and starts sending them one by one to the Sync Manager. In the meantime, the Sync Manager "looks into" the Server to get the number (μ) of newly generated workitems by using the API function ListWorkitems () specified in the WfMC standard [WfMC 1998]. Then, the following sequence of actions is taken by the software modules involved:

1. The Sync Manager passes each new workitem (W_N) received from the Server to a pertinent Simulator while counting the number (m) of new workitems it has passed.
2. When m (number of passed workitems) becomes equal to μ (number of newly generated workitems), the Sync Manager broadcasts a message to every Simulator requesting to send its local next-event time (τ_L).
3. Each Simulator j reports τ_L (its local next-event time) to the Sync Manager.
4. The Sync Manager selects a Simulator j* that has the smallest next-event time (τ_G = Min $\{\tau_L$ for all j$\}$), and broadcasts the global next-event time (τ_G) and the selected Simulator ID (j*) to all Simulators.
5. The selected Simulator j* completes its workitem and advances its next-event time, and then returns the completed workitem (W_C) back to the Sync Manager.
6. The Sync Manager passes the completed workitem (W_C) to the Server, which in turn updates the pertinent process instance to initiate the next cycle of enactment service.

The above parallel simulation process employs a centralized barrier method of time synchronization with the Sync Manager playing the role of a controller LP. In the following sections, state graph models of the workflow simulation

TABLE 12.1. Parameters and Variables Used in the State Graph Models

Name	Description	Name	Description
a	arrival time of workitem	N	total number of participant simulators
c	completion time of workitem	q	number of workitems in queue
clock	local simulation time of participant simulator	t_d	time delay
G	group size (number of people in the group)	W_N	new workitem
j	ID number of a participant simulator	W_C	completed workitem
j^*	ID number of the selected participant simulator	π	processing time of workitem
LLT	list of local next-event times (τ_L)	τ_L	local next-event time of each participant simulator
m	number of new workitems received	τ_G	global next-event time
n	number of participant simulators replied	μ	total number of newly generated workitems

modules in Fig. 12.2 will be presented. Parameters and variables used in the state graph models are summarized in Table 12.1.

12.2.3 State Graph Modeling of an Enactment Server and Sync Manager

In this section, the behaviors of the enactment server and the Sync Manager are specified using the state graph modeling formalism presented in Chapter 9. A state graph model of the participant simulator will be presented in the next subsection. The materials presented in this section (Section 12.2) are mostly from the paper by Lee et al. [2010] where the term *DEVS model* was used in place of *state graph model*.

Shown in Fig. 12.3 is a simplified version of state graph model of the enactment server, which is a simple finite state machine having three states. It is initially in the Wait state and moves to the Processing state (Update PI & Generate $\{W_N\}$) if the Start input is received. Once it has generated all the new workitems (for the enabled activities), it sends out the new workitems $\{W_N\}$ to the Sync Manager and moves to the Ready state. Then, it waits in the Ready state until it receives a completed workitem W_C, and then moves back to the processing state. It should be noticed that the processing state may not generate a new workitem W_N, in which case $\{W_N\}$ is a null

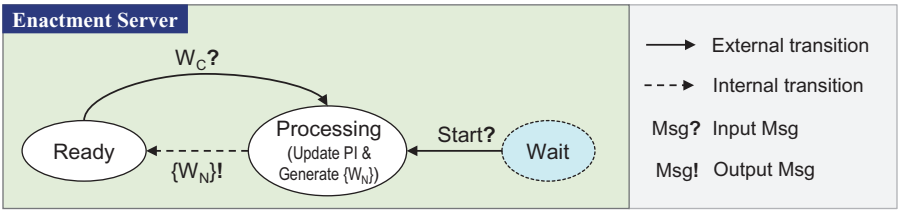


Fig. 12.3. State graph model of the enactment server.

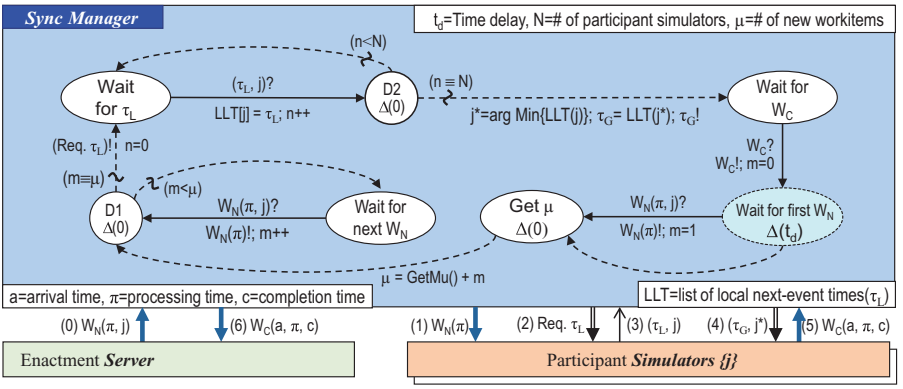


Fig. 12.4. State graph model of the Sync Manager.

set, meaning that the state is changed to Ready without sending out any workitem.

Figure 12.4 shows a state graph model of our Sync Manager together with its interactions with the enactment Server and participant Simulators. In the figure, the operation sequence is numbered from (0) to (6). At the beginning of an enactment service cycle, the Sync Manager stays in the Wait for first W_N state with m (the number of new workitems received) equal to zero. Then, (0) if a new workitem $W_N(\pi, j)$ is received from Server, (1) it moves to the Get μ state after sending $W_N(\pi)$ to Simulator j and setting m to one; otherwise, it moves to the Get μ state after a time delay of t_d. At the Get μ state, the Sync Manager obtains the value of μ = GetMu() (See Lee et al. 2010), and moves to the D1 state after setting $\mu = \mu + m$. Then, at the D1 state, (2) it goes to the Wait for next W_N state and comes back until m is equal to μ. At this point ($m \equiv \mu$), it moves to the Wait for τ_L state after sending the Request τ_L message to all Simulators and setting n (number of Simulators replied) to zero.

The Sync Manager waits in the Wait for τ_L state until (3) it receives the local next-event times (τ_L) from all Simulators (i.e., $n \equiv N$) while storing the value of each τ_L in the list of local next-event time LLT[j]. Then, (4) it selects a Simulator j* whose τ_L is the smallest, designating it as the global next-event time

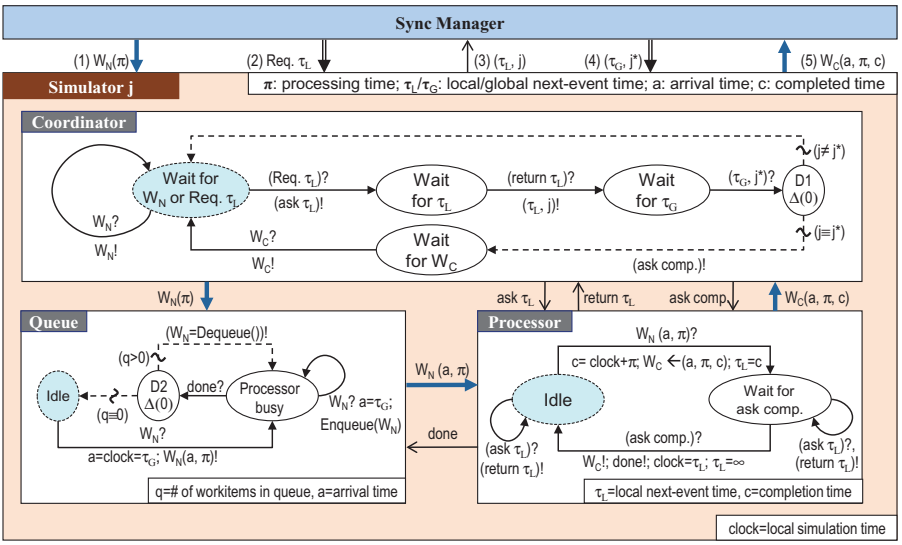


Fig. 12.5. State graph model of "single participant" simulator.

(τ_G), and moves to the Wait for W_C state after broadcasting τ_G and j* to all Simulators. Finally, (5) if a completed workitem $W_C(a, \pi, c)$ is received from the selected Simulator j*, (6) the Sync Manager sends the W_C to the Server and moves to the Wait for first W_N state to start a next cycle of enactment service.

12.2.4 State Graph Modeling of Participant Simulators

Each single participant in the workflow management system is modeled as a single server system processing new workitems $\{W_N(\pi)\}$ received from the Sync Manager. Shown in Fig. 12.5 is a composite state graph model of the single server system consisting of three atomic models: Coordinator, Queue and Processor. Each Simulator communicates with the Sync Manager as follows: (1) new workitems $W_N(\pi)$ received are stored in the Queue and a selected workitem is processed by the Processor; (2) upon receiving a Req. τ_L message, (3) its local next-event time τ_L is returned back to the Sync Manager; (4) if global next-event time τ_G is granted to this Simulator, (5) the completed workitem $W_c(a, \pi, c)$ is returned back to the Sync Manager.

12.2.5 Implementation of a Workflow Simulator

An existing workflow management system equipped with a Sync Manager and participant simulators can be used as a workflow simulator. Figure 12.6 shows the software structure of the workflow simulator: (1) a Workflow Engine

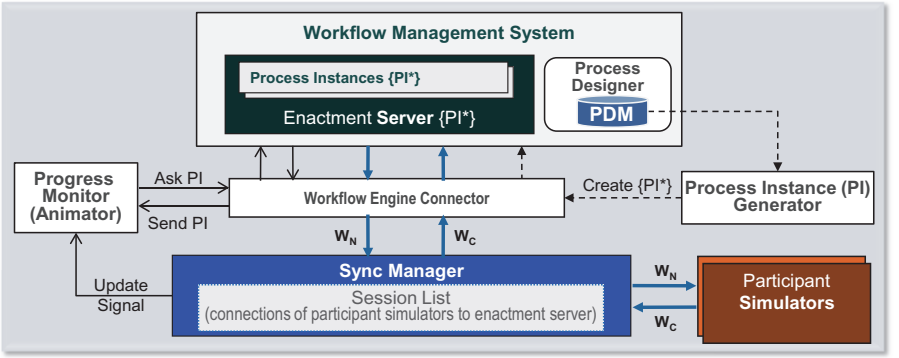


Fig. 12.6. Software structure of the workflow simulator.

Connector is used to connect the components of the simulation system; (2) a Process Instance Generator is added to the simulation system to be used in generating process instances {PI*} from the process definition models (PDMs) defined at the Process Designer of the workflow management system; (3) a Process Monitor is also added to visualize the simulation process.

The Enactment Server provides enactment services to Participant Simulators through the Sync Manager. The workflow simulator was implemented as a prototype workflow simulator using a commercial workflow management system and an academic workflow management system, both of which are in compliance with the WfMC standards [WfMC 1995]. The Sync Manager and single and group participant simulators have been developed under a Microsoft .NET Framework 3.5 environment using the C# programming language, and they are plugged into the workflow management systems via a workflow engine connector module [Lee et al. 2010].

12.3 OVERVIEW OF HIGH-LEVEL ARCHITECTURE/RUN-TIME INFRASTRUCTURE

As mentioned in the introduction, a high-level architecture (HLA) is a general-purpose architecture for parallel simulation systems, and the interactions between simulations are managed by a run-time infrastructure (RTI). HLA was mandated in September 1996 as the standard architecture for all modeling and simulation activities in the Department of Defense in the United States [Fujimoto 2000]. In HLA, a parallel simulation is referred to as a *federation*, and each individual sequential simulator as a *federate*. This section is based on the materials presented in the book by Kuhl et al. [2000] and in the lecture notes by Crosbie and Zenor [2006].

12.3.1 Basics of HLA/RTI

The objectives of HLA are to combine computer simulators into a larger simulation, or *federation*, to extend the simulation later by adding additional simulators, or *federates*, and to support component-based simulation development. In other words, it aims to enhance the reusability and interoperability of simulation models. Key components of HLA are HLA Rules, Interface Specification, and Object Model Template.

12.3.1.1 HLA Rules HLA Rules are to ensure proper interaction of federates in a federation and to describe the responsibilities of federates and federations. The ten HLA Rules are:

1. Federations shall have a federation object model (FOM), documented in accordance with the HLA object model template (OMT).
2. All representation of objects in the FOM shall be in the federate, not in the RTI.
3. During a federation execution, all exchange of FOM data among federates shall occur via the RTI.
4. During a federation execution, federates shall interact with the RTI in accordance with the HLA interface specification.
5. During a federation execution, an instance attribute shall be owned by at most one federate at any given time.
6. Federates shall have a simulation object model (SOM), documented in accordance with the HLA OMT.
7. Federates shall be able to update and/or reflect any instance attributes, and send and/or receive interactions, as specified in their SOMs.
8. Federates shall be able to transfer and/or accept ownership of attributes dynamically during a federation execution, as specified in their SOMs.
9. Federates shall be able to vary the conditions under which they provide updates of instance attributes, as specified in their SOMs.
10. Federates shall be able to manage local time in a way that will allow them to coordinate data exchange with other members of a federation.

Rules 1 to 5 are known as *federation rules* and rules 6 to 10 are *federate rules*.

12.3.1.2 Interface Specification The interface specification defines the functional interfaces between federates and the RTI. The RTI is software that provides HLA services to federates. As depicted in Fig. 12.7, the interface service mechanisms are implemented as ambassadors: Federate Ambassador for RTI-initiated services and RTI Ambassador for federate-initiated services.

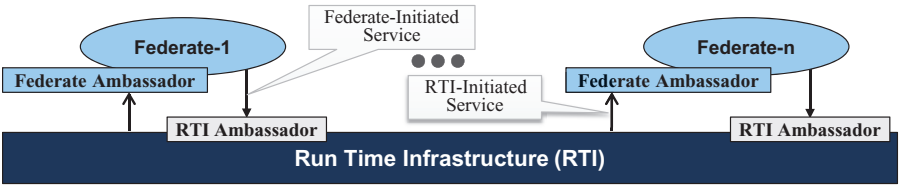


Fig. 12.7. Interface service mechanisms of the RTI.

The interface specification (the set of APIs) for HLA services is divided into six management areas or RTI module groups. The six management areas are:

1. *Federation management* for creating federation execution and permitting a federate to join or to resign from the execution
2. *Declaration management* to allow federates to declare their intent to *publish* or *subscribe* to data
3. *Object management* to send and receive interactions; register a new object instance and update its attributes; and discover new instances and reflect updated attributes
4. *Ownership management* to grant/transfer of ownership of an instance-attribute
5. *Data distribution management* to control the producer-consumer relationships among federates
6. *Time management* to allow federates to advance its logical time and control the delivery of timestamped events

12.3.1.3 Object Model Template (OMT) The OMT provides a standard for documenting HLA object model information. It defines the federation object model (FOM), the simulation or federate object model (SOM) and the management object model (MOM).

1. FOM introduces all shared data among federates: objects and interactions.
2. SOM describes salient characteristics (internal operations) of a federate.
3. MOM identifies objects and interactions used to manage a federation.

There are two types of shared data: *interaction* and *object*. An interaction is a collection of data (usually events) sent through the RTI to other federates. One federate sends an interaction; another receives it (and does not reside in the federation). Each interaction class has a set of named data called

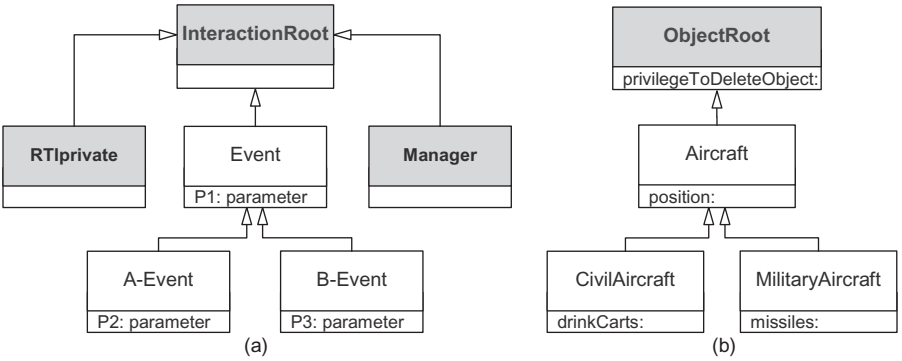


Fig. 12.8. Examples of (a) an interaction class and (b) an object class.

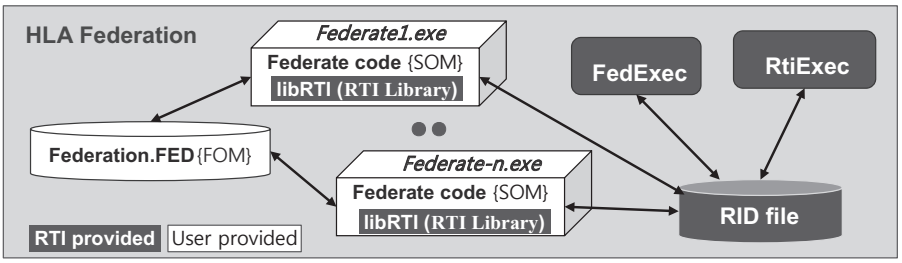


Fig. 12.9. Systems architecture (or software module structure) of an HLA federation.

parameters. An object is a simulation entity (usually a state variable) that is of interest to more than one federate and persists for some interval of simulation time. Each object class has a set of named data called *attributes*. Figures 12.8(a) and (b) show examples of an interaction class and of an object class, respectively.

In the class hierarchy diagrams of Fig. 12.8, the shaded boxes (InteractionRoot, RTIprivate, Manger, and ObjectRoot) are built-in classes provided by the RTI system. The fully qualified name of B-Event, for example, is InteractionRoot.Event.B-Event, and its available parameters are P1 and P3. The fully qualified name of the object CivilAircraft is ObjectRoot.Aircraft. CivilAircraft, and its available attributes are privilegeToDeleteObject, position, and drinkCarts.

12.3.2 HLA Federation Architecture

Figure 12.9 shows the system architecture of HLA federation. An HLA federation consists of a set of RTI-provided software modules—RID file, RtiExec, FedExec, libRTI—and user-defined modules—Federate codes and Federation.FED. The roles of the software modules are:

1. The RTI initialization data (RID) file contains information needed to run the RTI.
2. The RtiExec (RTI executive) manages the creation and destruction of FedExec.
3. The FedExec (federation executive) allows federates to join and resign from the federation, and facilitates data exchanges among them.
4. The libRTI (RTI library) is used by federates to invoke various HLA services.
5. The Federation.FED (federation execution data) contains information derived from the FOM in the form of FDD (FOM document data) file.
6. A Federate.exe consists of federate code and libRTI. Federate code contains various local simulation objects including SOM.

In a physical configuration of a federation, a copy of each Federation.FED and RID file is bundled with each Fedreate.exe file in a local computer running the federate, and RtiExec and FedExec files reside in the "console" computer.

12.3.3 Overview of Federation Execution

Figure 12.10 shows the steps in the process of starting a federation execution. The initial step is to prepare the federate code for each federate and the Federation.FED file. Then, the execution process is as follows:

1. When a federation is run, the RtiExec is started first.
2. Then a federate, acting as a manager, creates a federation execution by invoking the RTI method createFederationExecution on its RTIambassador.

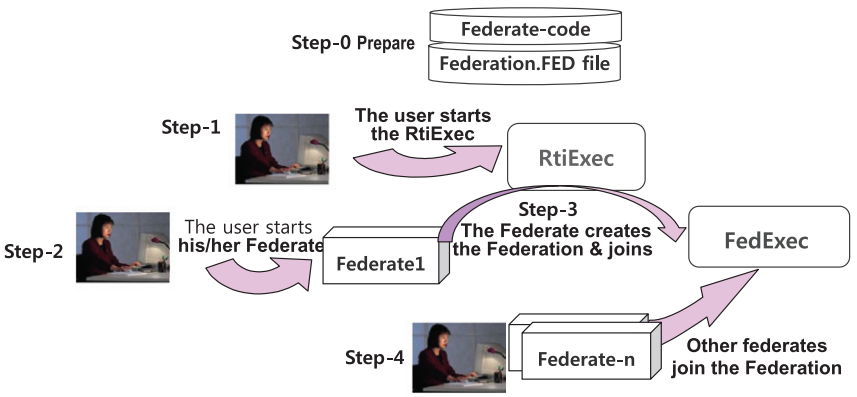


Fig. 12.10. Federation execution process.

3. The RTIambassador then reserves a name with RtiExec, and spawns a FedExec process, and that FedExec registers its communication address with RtiExec. The federation execution is underway.
4. Once a federation execution exists, other federates can join it. That RTIambassador consults RtiExec to get the address of FedExec, and invokes joinFederationExecution () on FedExec. Additional federates can join via the same process.

12.4 IMPLEMENTATION OF A PARALLEL SIMULATION WITH HIGH-LEVEL ARCHITECTURE/RUN-TIME INFRASTRUCTURE

This section aims to provide a beginner's guide to the implementation example given in a book, which we call the *HLA Book*, by Kuhl et al. [2000]. The HLA Book provides an excellent coverage of the subject, and it may be easier for a beginner to understand the contents of the book after reading this section. This section begins with an overall description of the "sushi boat" restaurant system presented in Chapter 4 of the HLA Book.

12.4.1 The Sushi Restaurant Federation

Figure 12.11(a) shows a "sushi boat" restaurant where the chefs work on an island surrounded by a circular canal on which boats are floating with the current (flow) of the water. The diners sit at a bar that surrounds the canal.

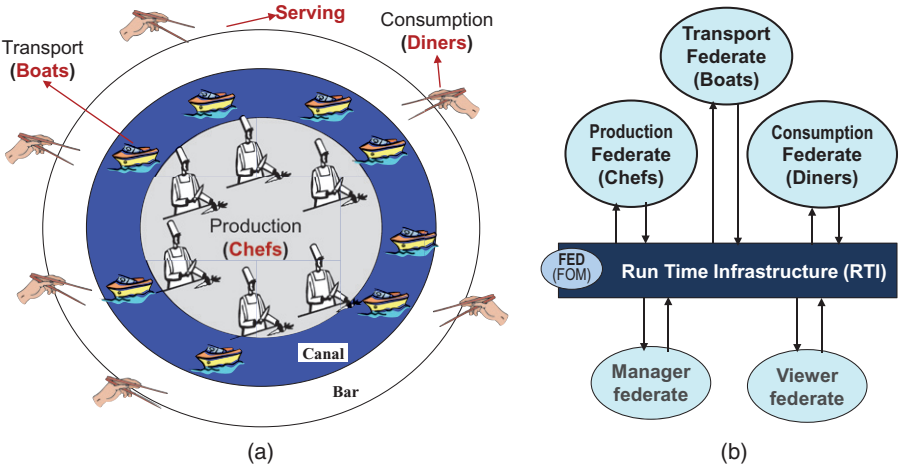


Fig. 12.11. (a) A sushi restaurant system and (b) its HLA federation.

As the chefs prepare servings, they place them on rectangular plates; when they have finished a batch of servings, which may fill several plates, the chefs place the plates on empty boats as they float by. Diners remove the plate of their choice as the boat comes by and enjoy it. The chefs are called a *production subsystem*, the boats a *transport subsystem*, and the diners a *consumption subsystem*.

Figure 12.11(b) shows the HLA federation of the sushi restaurant system consisting of five federates and the RTI including a FED file. The three federates corresponding to the three subsystems of the restaurant are called *subsystem federates*: Production federate, Transport federate, and Consumption federate. The Manager federate is used in managing the federation execution, and the Viewer federate acts as a passive recipient; its role is to display of simulation data from the rest of the federation. The FED (also called Federation.FED) file contains the federation object model (FOM). As mentioned in Section 12.3.2 (Fig. 12.9), an HLA federation consists of a number of files in addition to the federates and FED file. However, only the federates and FED file need to be prepared by the federation designer.

12.4.2 Preparation of an FED File

As was shown in Fig. 12.11(a), there are four types of objects in the restaurant system: Servings, Boats, Chefs, and Diners. Figure 12.12(a) shows an object class tree specifying the restaurant objects. The root of the object class tree is

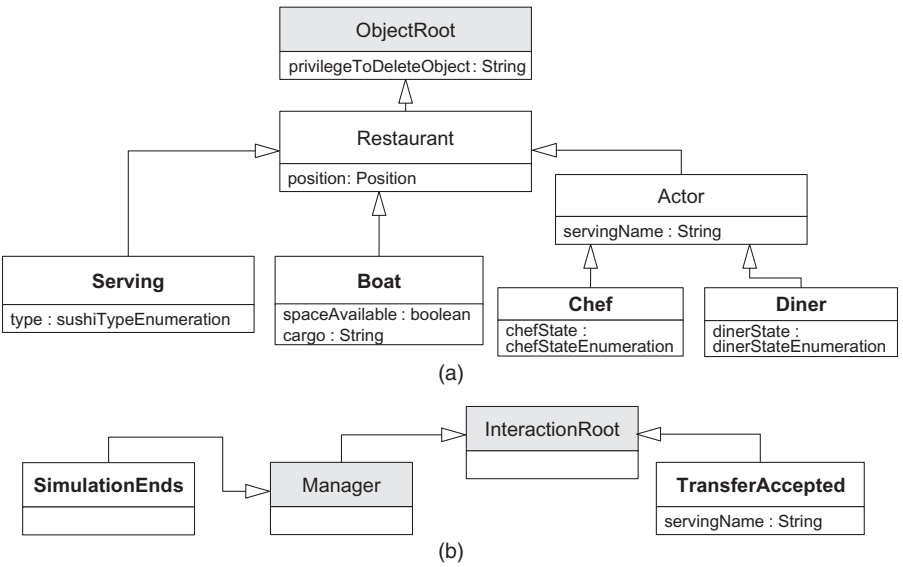


Fig. 12.12. (a) Object class tree and (b) interaction class tree.

called ObjectRoot. At the leaf level of the hierarchy are the four object classes: Serving, Boat, Chef, and Diner. The Actor class serves as a place to define the attribute servingName that is in common for Chef and Diner. The attributes chefState and dinerState are conceptually enumerations and will both be represented in Java as int.

Figure 12.12(b) shows a class tree for interactions. An interaction is a simulated occurrence (or event) that occurs at a point in time and does not persist. All interaction classes are subclasses of InteractionRoot. The TransferAccepted interaction has a parameter of servingName. It is a message sent from the Transport federate to the Production federate to signal a Boat's acceptance of a Serving. SimulationEnds is a message sent from the Consumption federate to other federates signaling the end of simulation. It is a subclass of Manager because it is a user-extension of the management object model (MOM).

Figure 12.13 shows a FED file containing the object class tree and interaction class tree given in Fig. 12.12. Observe in the FED file that each class attribute and each interaction class are appended by modifiers reliable timestamp or reliable receive. The choices of the communication network over which the messages (class attributes and interaction classes) are sent are either *reliable* or *best-effort*. A *reliable communication* guarantees that the data will be delivered or an exception will be indicated. The choices of message-delivery ordering are either *timestamp* or *receive*. In a timestamp order (TSO), the arrival of timestamped messages is sequenced in accordance with logical time;

```
(FED  ;; Defining object classes and interaction classes
   (Federation restaurant_1)  ;; we choose this tag
   (FEDversion v1.3)          ;; required; specifies RTI spec version
   (spaces                    ;; we define no routing spaces
   )
   ( objects
      (class ObjectRoot       ;; required
         (attribute privilegeToDeleteObject reliable timestamp)
         (class RTIprivate)
         (class Restaurant (attribute position reliable timestamp)
            (class Serving (attribute type reliable timestamp) )
            (class Boat (attribute spaceAvailable reliable timestamp) (attribute cargo reliable timestamp) )
            (class Actor (attribute servingName reliable timestamp)
               (class Chef (attribute chefState reliable timestamp) )
               (class Diner (attribute dinerState reliable timestamp) ) ) ) ;; end of Restaurant
         (class Manager …)
    ) ;; end ObjectRoot
 )  ;; end Objects
   ( interactions
      (class InteractionRoot reliable timestamp
          (class TransferAccepted reliable timestamp (parameter servingName) )
          (class RTIprivate reliable timestamp)
          (class Manager reliable receive
               (class SimulationEnds reliable receive) …)   ;; end InteractionRoot
   ) ;; end interactions
) ;; end FED
```

Fig. 12.13. FED file for the object and interaction class trees in Fig. 12.12.

in a receive order, the messages are delivered as they arrive without regard to logical time. In the FED file, both ObjectRoot and InteractionRoot have a required subclass called RTIprivate (not shown in Fig. 12.12) to be used by RTI implementers. It must be present but cannot be extended (subclasses) by a federation designer. ObjectRoot has another required subclass called Manager. It is the root of a further tree that defines the object portion of the Management Object Model (MOM). All the subclasses of Manager that you see in the sample FED files are required to be there.

12.4.3 Preparation of the Federate Code (of the Production Federate)

Table 12.2 lists the names of the nine Java files constituting the Production federate. The federation designer has to prepare a federate code for each of the five federates in Fig. 12.11(b). The first six files in Table 12.2 have the same fixed structure for all the federates. Thus, you as a beginner do not need to worry about them. The last two files (ChefTable and ProductionFrame) are concerned about the graphical user interface (GUI) for the federate. The main file of the Production federate code is Production.java that has to be prepared by you as a federation designer.

Table 12.3 the program (Java code) structure of Production federate. The Transport federate and the Consumption federate have the similar structure.

TABLE 12.2. Files in the Federate-Code of Production Federate

No	File Name	Description
1	Barrier.java	Define *Barrier* class that is used to coordinate the activities between threads.
2	CallbackQueue.java	Define *Callback Queue* class that is used to store the callbacks initiated by the RTI
3	InternalQueue.java	Define *Internal Queue* class that stores the internal events of Chef model.
4	ProductionInternalError.java	Define an exception raised within the Production federate
5	ProductionNames.java	Define the default "federate type" string of the Production federate
6	FedAmbImpl.java	Define *FedAmbImpl* class that represents the **Production** federate ambassador.
7	Production.java	Define Production *federate code* to execute the **Production** federate.
8	ChefTable.java	Define Chef Table View of the user interface of the **Production** federate.
9	ProductionFrame.java	Define the user interface (main window) of the Production federate.

TABLE 12.3. Program Structure of the Production Federate

Phase	Service group (Mngt Area)	Descriptions
1. Prepare	Declare Object and Variables	(1) ProductionFrame, (2) FedAmbImpl, etc.
	Join Federation (Fed Mngt-1)	(1) Obtain Ref., (2) Create Fed Execution, (3) Join Fed Execution
	Set Time Switches (Time Mngt-1)	(1) Enable Time Constrained, (2) Enable Time Regulation
	Publish/Subscribe (Decl. Mngt)	(1) Publish/Subscribe Objects, (2) Publish/Subscribe Interactions
2. Populate	Define Objects (Obj. Mngt-1)	(1) Register Object Instance, (2) Update Attribute Values
3. Run	Time Advance (Time Mngt-2)	(1) Next Event Request, (2) Time Advance Grant
	Produce data (Obj. Mngt-2)	(1) Register/Discover, (2) Update/ Reflect (3) Receive Interaction
	Ownership (Ownership Mngt)	Ownership Divestiture/Release.
4. Resign	Resign Federation (Fed Mngt-2)	Resign Federation Execution

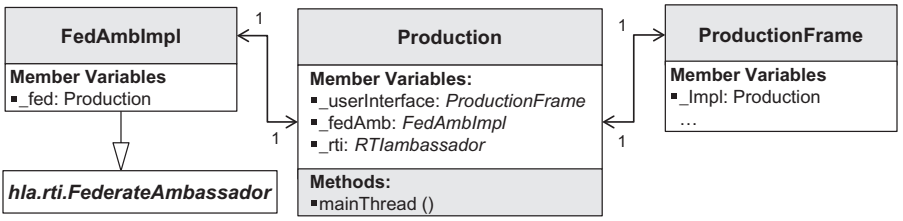


Fig. 12.14. Main objects in Production federate.

The program structure follows the phases of the federate lifecycle: *Prepare phase*, *Populate phase*, *Run phase*, and *Resign phase*. In this section, only the Prepare phase will be explained in some detail. Detailed description of the federate program structure is out of scope of this book, and interested readers are referred to the HLA Book [Kuhl et al. 2000].

12.4.3.1 Declare Objects and Variables Figure 12.14 shows two object classes in the Production federate: ProductionFrame and FedAmbImpl. The Production instance creates an instance of ProductionFrame that contains all the user interface code. It also creates an instance of FedAmbImpl that is an implementation of hla.rti.FederateAmbassador, which is passed to the RTI

when the federate joins. Callbacks from the RTI are invoked on FedAmbImpl, which in turn calls methods in the Production instance. The Production class contains the mainThread () method that contains the rest of the federate code. The mainThread () holds the references to other data structures and objects, and all the calls to RTIambassador occur in its code.

12.4.3.2 Join Federation Before a *federation execution* (FE) exists, it must be defined to the RTI. The federation execution must be created and associated with a FOM and the federation must join the FE.

Figure 12.15 shows a Production federate code for obtaining a reference to RTI ambassador. The RTI ambassador provides access to the RTI services. The hostname and portNumber indicates where the RTI executive runs. The hostname can be numerical IP address (e.g., 127.0.0.1) or string address (e.g., www.kaist.ac.kr). The portNumber is a positive integer (e.g. 8989—this is default value of Pitch Portal RTI used in this book)

Figure 12.16 shows a federate-code code for creating the FE. The FE can be created by letting each federate attempt to create the FE at the start. If the federate receives an exception reporting that the FE already exists, it is ignored. The following shows the partial Java code of creating the FE at the Production federate. Also, we assume that the federation execution data (FED) file is named "restaurant_1.fed" and located in the directory of the Production federate

Figure 12.17 shows a federate-code code for joining the FE. Each federate joins the FE by invoking rti.joinFederationExecution with three arguments: the name of the FE, a federate type (or federate name), and a federate ambassador (fedAmb). Then the RTI returns a handle, called *federate handle*, consisting of a small positive integer. We assume that the federate has constructed an instance of some implementation of the Java interface hla.rti.FederateAmbassador, a federate ambassador receiving messages originated by RTI, called fedAmb.

```
String hostname = "127.0.0.1";
int portNumber = 8989;
RTIambassador _rti = RTI.getRTIambassador (hostName, portNumber);
```

Fig. 12.15. Federate code for obtaining a reference to RTI ambassador.

```
String federationExecutionName = "Restaurant";
java.net.URL fedURL = new java.net.URL("restaurant_1.fed");
try {
 _rti.createFederationExecution (federationExecutionName, fedURL);
} catch (hla.rti.FederationExecutionAlreadyExists e) {
 System.out.println ("Federation" + federationExecutionName + "already exists");}
```

Fig. 12.16. Federate code for creating the federation execution (FE).

```
hla.rti.FederationAmbassador _fedAmb; //implementation of Federate ambassador
int federateHandle; //designator for Production federate
String federateType = "Production"; //for the production federate
try {
  federateHandle = _rti.joinFederationExecution (
          federationExecutionName, federateType, _fedAmb);
} catch (hla.rti.FederationExecutionAlreadyExists e) {
 System.out.println ("Exception on join" + e); }
```

Fig. 12.17. Federate code for joining the federation execution (FE).

TABLE 12.4. Attribute Publications and Subscriptions

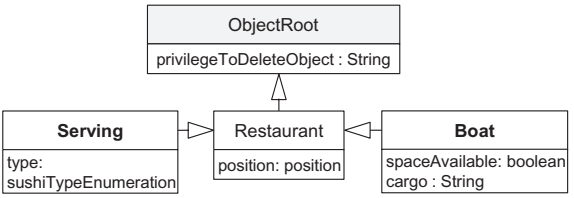
Object	Attributes	Production	Transport	Consumption	Viewer
Serving	position	publish	publish	publish	passive subscribe
	type	publish		subscribe	passive subscribe
Boat	position	subscribe	publish	subscribe	passive subscribe
	spaceAvailable	subscribe	publish	subscribe	passive subscribe
	cargo	subscribe	publish	subscribe	passive subscribe
Chef	position	publish			passive subscribe
	chefState	publish			passive subscribe
	servingName	publish			passive subscribe
Diner	position			publish	passive subscribe
	dinerState			publish	passive subscribe
	servingName			publish	passive subscribe

TABLE 12.5. Interaction Publications and Subscriptions

Interaction	Manager	Production	Transport	Consumption	Viewer
SimulationEnds	subscribe	subscribe	subscribe	publish	subscribe
TransferAccepted		subscribe	publish		

12.4.3.3 Publications and Subscriptions Table 12.4 shows attribute publications and subscriptions for the object class tree in Fig. 12.12(a). For example, attribute Serving.position is published by the subsystem federates and is subscribed by the Viewer federate, and Serving.type is published by the Production federate and is subscribed by the Consumption and Viewer federates. Each object class inherits the attribute privilegeToDeleteObject from the class ObjectRoot whose publication is set to "default publish." Thus, this "root attribute" does not need to be declared explicitly.

Table 12.5 shows interaction publications and subscriptions for the interaction class tree in Fig. 12.12(b). The simulation is to be ended if the number of servings exceeds certain value at the consumption subsystem. Thus,



(a)

Object	Attributes	Production
Boat	position	subscribe
	spaceAvailable	subscribe
	cargo	subscribe
Serving	position	publish
	type	publish

(b)

Fig. 12.18. (a) Object class tree and (b) pub/sub table for Serving and Boat objects.

```
//Get an object-class-handle for the Boat and Serving objects
int RestaurantClassHandle = _rti.getObjectClassHandle("ObjectRoot.Restaurant");
int BoatClassHandle = _rti.getObjectClassHandle("ObjectRoot.Restaurant.Boat");
int ServingClassHandle = _rti.getObjectClassHandle("ObjectRoot.Restaurant.Serving");
//Get an attribute-handle for each attribute of the Boat and Serving object classes
int positionAttributeHandle = _rti.getAttributeHandle("position", RestaurantClassHandle);
int spaceAvailableAttributeHandle = _rti.getAttributeHandle("spaceAvailable", BoatClassHandle);
int cargoAttributeHandle = _rti.getAttributeHandle("cargo", BoatClassHandle);
int typeAttributeHandle = _rti.getAttributeHandle("type", ServingClassHandle);
//Create an empty set by using the factory reference and store the attribute handles in the created sets
hla.rti.AttributeHandleSetFactory ahFactgory = RTI.attributeHandleSetFactory();
hla.rti.AttributeHandleSet boatAttributeHandles = ahFactory.create();
boatAttributeHandles.add(positionAttributeHandle);
boatAttributeHandles.add(spaceAvailableAttributeHandle);
boatAttributeHandles.add(cargoAttributeHandle);
hla.rti.AttributeHandleSet servingAttributeHandles = ahFactory.create();
servingAttributeHandles.add(positionAttributeHandle);
servingAttributeHandles.add(typeAttributeHandle);
//Declare publications
try {  //publish the object class attributes
       _rti.publishObjectClass (ServingClassHandle, servingAttributeHandles);
       //subscribe the object class attributes
       _rti.subscribeObjectClassAttributes (BoatClassHandle, boatAttributeHandles);
} catch (hla.rti.RTIexception e) { … }
```

Fig. 12.19. Sample implementation of pub/sub of Serving and Boat object attributes.

SimulationEnds is published by the Consumption federate and subscribed by other federates. As another example, TransferAccepted is published by the Transport federate and subscribed by the Production federate.

Figure 12.18(a) shows the object class tree reproduced from Fig. 12.12 for the Serving object and Boat object. Figure 12.18(b) is a portion of the pub/sub table (Table 12.4) for the Serving and Boat objects at the Production federate. Figure 12.19 is a sample Java code declaring the publication of the Serving and the subscription of the Boat attributes shown in Fig. 12.18. Declarations are made as follows: (1) Get an object class handle by invoking the

getObjectClassHandle () function; (2) get attribute handles by invoking the getAttributeHandle () method; (3) the attribute handles to be subscribed are stored in separate sets; and (4) the to-be-published or to-be-subscribed attributes are published or subscribed by invoking publishObjectClass () or subscribeObjectClassAttributes ().

12.4.4 Executing the Restaurant Federation

The overall procedure for preparing and executing your own federation was described earlier in Section 12.3.3 (See Fig. 12.10). In this section, we will show you how to download a sample implementation of the restaurant federation and run the program. More details may be found in the HLA Book [Kuhl et al. 2000]. Even if you do not sufficiently understand the internal workings of the sushi restaurant federation, you are advised to follow the steps explained below to get familiar with it.

12.4.4.1 Download and Installation Steps for downloading and installing the Sushi Restaurant Federation together with HLA/RTI software are as follows:

1. Download the restaurant federation sample code and HLA/RTI software from the following website: http://authors.phptr.com/kuhl.
2. Unzip the downloaded file into a directory named <kuhl>. The top-level directories in the directory <kuhl> are
   - <kuhl>\bin: class files for the implementation of restaurant federation
   - <kuhl>\config: configuration data needed to run the restaurant federation
   - <kuhl>\doc: all documents
   - <kuhl>\lib: Java archives needed to run RTI and the restaurant federation
   - <kuhl>\src: Java source for the implementation of restaurant federation
3. Install a Java run-time environment by running jre-1_2_2_005-win.exe located in the <kuhl> directory.

12.4.4.2 Executing the* RTI *Executive The RTI executive (RtiExec in Fig. 12.10) can be executed by running the batch file <kuhl>\rti.bat. If executed successfully, the GUI shown in Fig. 12.20 will be presented to you. Then, (1) check the Federates field to have the list of joined federates displayed, and (2) check the RTI activity field to monitor the activities processed in the RTI executive.

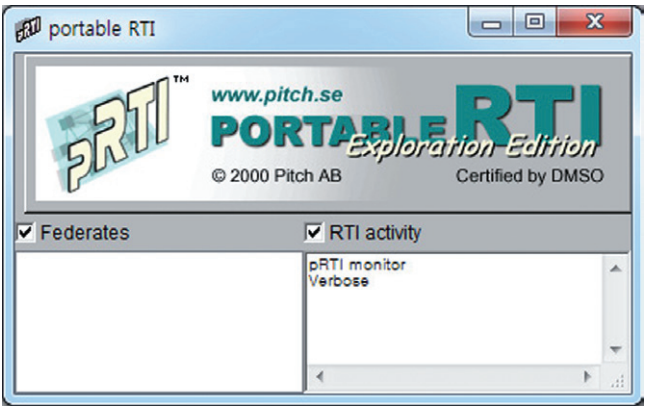


Fig. 12.20. RTI executive GUI.

12.4.4.3 Running the Restaurant Federation The Restaurant Federation is started by running the batch file <kuhl>\rest&view.bat. The batch file starts all federates one at a time in the following sequence: Manager, Production, Transport, Consumption, and Viewer. After running the batch file, the GUI of Fig. 12.21 will show up on your computer screen. Shown in the left side are user interfaces (UI) for Manager, Production, Transport, and Consumption federates, and the right side is the GUI for the Viewer federate.

If you have a problem with running the batch files, you may try the following: (1) Check if a Java run-time environment (JRE) is installed in the <kuhl> directory on your computer; (2) if you have the JRE installed already, check its version. If the JRE version is higher than 1.2.2 or was installed JRE from the http://www.java.com, you need to modify the batch files as shown in Fig. 12.22.

Figure 12.23 shows the UI of the Production federate. Transport and Consumption federates have the same structure. The main window is split into a chef table and a log area. The chef table contains a tabular display of the status of the chefs, and the log area displays messages written by the federate. The statuses of the chefs are specified as:

- Position: angular position of the chef along the canal
- State: chef's state {Making Sushi, Looking for Boat, Waiting to Transfer}
- Serving: the object instance handle of the Serving the chef has prepared
- Boat to Xfer to: the handle of the Boat instance the chef is trying to load

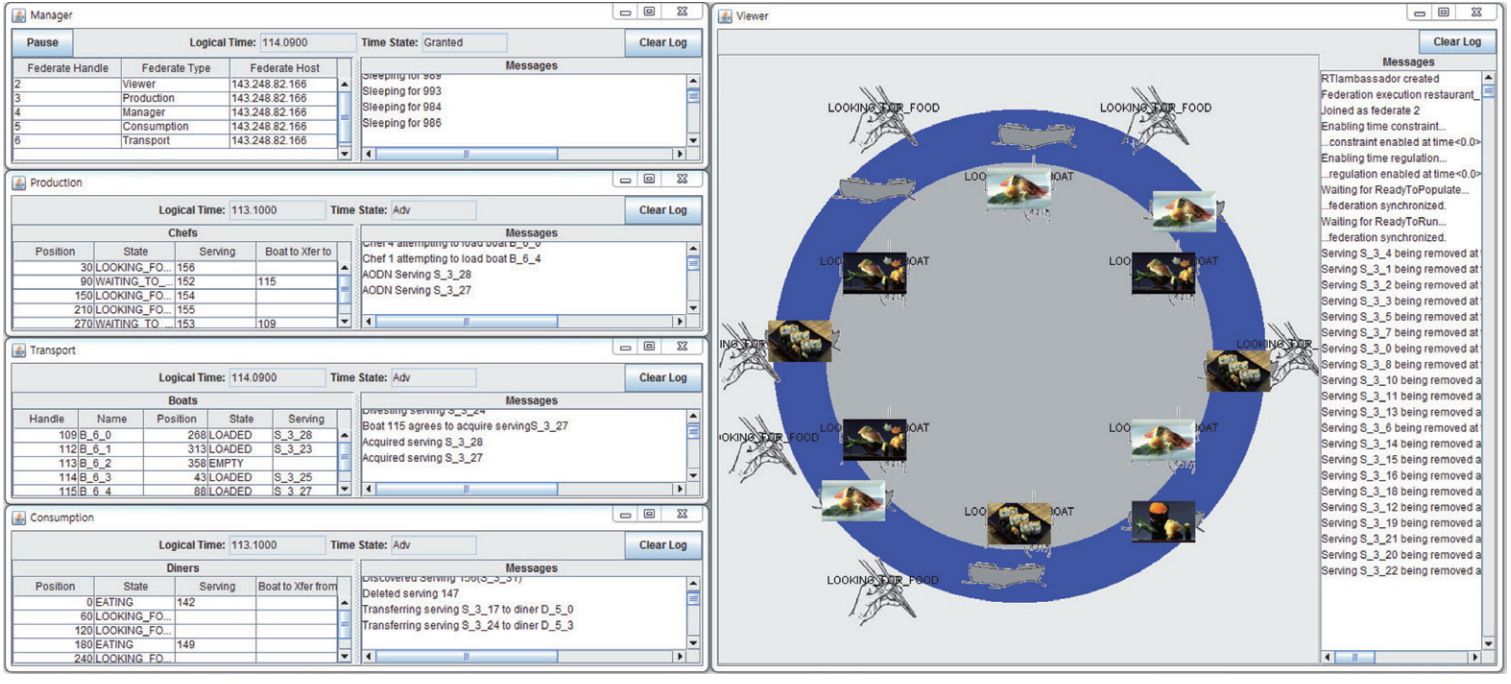


Fig. 12.21. Restaurant federation executions GUI.

```
[rti.bat]
java -cp bin;lib\prti.jar;lib\jgl3.1.0.jar se.pitch.prti.RTIexec
```

```
[rest&view.bat]
start "Manager" /B java -cp bin;lib\prti.jar;lib\jgl3.1.0.jar;lib\swingall.jar
org.mitre.hla.book.manager.Manager
start "roduction" /B java -cp bin;lib\prti.jar;lib\jgl3.1.0.jar;lib\swingall.jar
org.mitre.hla.book.restaurant.production.Production
start "Transport" /B java -cp bin;lib\prti.jar;lib\jgl3.1.0.jar;lib\swingall.jar
org.mitre.hla.book.restaurant.transport.Transport
start "Consumption" /B java -cp bin;lib\prti.jar;lib\jgl3.1.0.jar;lib\swingall.jar
org.mitre.hla.book.restaurant.consumption.Consumption
start "Viewer" /B java -cp bin;lib\prti.jar;lib\jgl3.1.0.jar;lib\swingall.jar
org.mitre.hla.book.restaurant.viewer.Viewer
```

Fig. 12.22. Modifying the batch files.

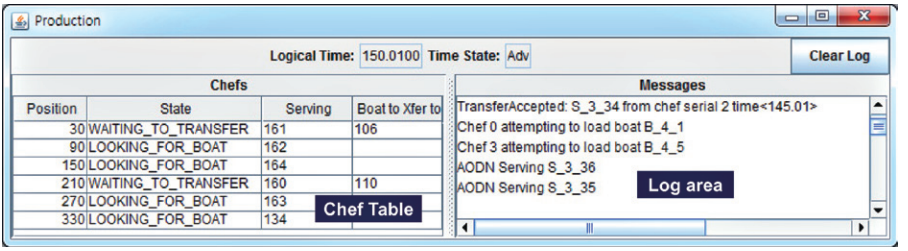


Fig. 12.23. User interface of Production federate.

REFERENCES

Activity cycle diagram. Available at http://minitorn.tlu.ee/~jaagup/uk/ds/chp3/CHAP3P~1.HTM, accessed 2012.

Altiok, T. and Melamed, B. 2007. *Simulation Modeling and Analysis with Arena*. Amsterdam: Elsevier.

Alur, R. Timed automata. 1999. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99), Lecture Notes in Computer Science*, Vol. 1633. New York: Springer-Verlag, pp. 8–22.

Alur, R. and Dill, D.L. 1994. A theory of timed automata. *Theoretical Computer Science* 126(2): 183–235.

Asfahl, J.T. 1992. *Robots and Manufacturing Automation*. New York: John Wiley & Sons.

Azadivar, F. 1999. Simulation optimization methodologies. In *Proceedings of the 1999 Winter Simulation Conference*, pp. 93–100.

Bengtsson, J. and Yi, W. 2004. Timed automata: Semantics, algorithms and tools. In Lectures on Concurrency and Petri Nets. *Lecture Notes in Computer Science, Vol. 3098*. Berlin: Springer. pp. 87–124.

Black, J.T. and Hunter, S.L. 2003. *Lean Manufacturing Systems and Cell Design*. Dearborn, MI: Society of Manufacturing Engineers.

Box, G.E.P. and Muller, M.E. 1958. A note on the generation of random normal deviates. *Annals of Mathematical Statistics* 29(2): 610–611.

Brooks Automation. 2003a. *AutoMod User's Guide*.

Brooks Automation. 2003b. *Model Communications User's Guide*.

Buluta, I. and Nori, F. 2009. Quantum simulators. *Science* 326(5949): 108–111.

Camurri, A. and Coglio, A. 1997. A Petri net-based architecture for plant simulation. In *Proceedings of the 6th IEEE Conference on Emerging Technologies and Factory Automation*, pp. 397–402.

Carrie, A. 1988. *Simulation of Manufacturing Systems*. New York: John Wiley & Sons.

Carson, J.S. 1993. Modeling and simulation worldviews. In *Proceedings of the 1993 Winter Simulation Conference*, pp. 18–23.

Cassandras, C.G. and Lafortune, S. 2010. *Introduction to Discrete Event Systems, 2nd Ed*. New York: Springer.

Chen, H., Chu, C., and Proth, J.M. 1998. Cyclic scheduling of a hoist with time window constraints. *IEEE Transactions on Robotics and Automation* 14(1): 144–152.

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.

Cheng, R.C.H. 1978. Generating beta variates with non-integral shape parameters, *Communications of the ACM* 21: 317–322.

Choi, B.K., Park, J.H. and Lee, T.E. 1996. Event graph modelling of automated sorting and buffering system. *International Journal of Computer Integrated Manufacturing* 9(5): 369–380.

Choi, B.K., Park, B.C., and Park, J.H. 2003. A formal model conversion approach to developing a DEVS-based factory simulator. *Simulation* 79(8): 440–461.

Cinlar, E. 1975. *Introduction to Stochastic Process*. Upper Saddle River, NJ: Prentice Hall.

Clementson, A.T. 1986. Simulating with activities using C.A.P.S./E.C.S.L. In *Proceedings of the 1986 Winter Simulation Conference*, pp. 113–122.

Crosbie, R. and Zenor, J. *High Level Architecture Modules* (Lecture Notes). California State University, Chico. Available at http://www.ecst.csuchico.edu/~hla/courses.html, accessed 2013.

Dellino, G., Kleijnen, P.C., and Meloni, C. 2008. Robust optimization in simulation: Taguchi and response surface methodology. Discussion Paper 2008-69. Tilburg University, Center for Economic Research.

Dicesare, F., Harhalakis, G., Proth, J.M., Silva, M., and Vernadat, F.B. 1993. *Practice of Petri-nets in Manufacturing*. London: Chapman & Hall.

Duncan W.R. 1996. *A Guide to the Project Management Body of Knowledge*. Upper Darby, PA: Project Management Institute.

Fishwick, P.A. 1995. *Simulation Model Design and Execution*. Upper Saddle River, NJ: Prentice Hall.

Fu, M.C., Glover, F.W., and April, J. 2005. Simulation optimization: A review, new development, and applications. In *Proceedings of the 2005 Winter Simulation Conference*, pp. 83–95.

Fujimoto, R.M. 2000. *Parallel and Distributed Simulation Systems*. New York: John Wiley & Sons.

Gan, B.P., Liow, L.F., Gupta, A.K., Lendermann, P., Turner, S.J., and Wang, X. 2007. Analysis of a borderless fab using interoperating AutoSched AP models. *International Journal of Production Research* 45(3): 675–697.

Glinsky, E. and Wainer, G. 2006. New parallel simulation techniques of DEVS and Cell-DEVS in CD++. In *Proceedings of the 39th Annual Simulation Symposium*, pp. 244–251.

Hannon, B. and Ruth, M. 2001. *Dynamic Modeling*, 2nd Ed. New York: Springer.

Harrell, C., Ghosh, B.K., and Bowden, R.O. 2012. *Simulation Using ProModel®*. New York: McGraw Hill.

Hlupic, V., and Paul, R.J. 1994. Simulation modelling of flexible manufacturing systems using activity cycle diagrams. *Journal of the Operational Research Society* 45(9): 1011–1023.

Hollocks, B.W. 2008. Intelligence, innovation and integrity: KD Tocher and the dawn of simulation. *Journal of Simulation* 2(3): 128–137.

Hopcroft, J.E., Motwani, R., Ullman, J.D. 2006. *Introduction to Automata Theory, Languages, and Computation*, 3rd Ed. Boston: Addison Wesley.

Kang, D. and Choi, B.K. 2011. The extended activity cycle diagram and its generality. *Simulation Modelling Practice and Theory* 19(2): 785–800.

Karnopp, D.C., Margolis, D.L., and Rosenberg, R.C. 2000. *System Dynamics: Modeling and Simulation of Mechatronic Systems*, 3rd Ed. New York: John Wiley & Sons.

Kelton, W.D., Sadowski, R.P., and Sturrock, D.T. 2007. *Simulation with ARENA*, 4th Ed. New York: McGraw Hill.

Khansa, W., Aygalinc, P., and Denat, J.P. 1996. Structural analysis of p-time Petri Nets. In *Proceedings of IEEE Computational Engineering in Systems Applications (CESA'96)*, pp. 127–136.

Kienbaum, G. and Paul, R.J. 1994. H-ACD: Hierarchical activity cycle diagrams for object-oriented simulation modelling. In *Proceedings of the 1994 Winter Simulation Conference*, pp. 600–610.

Kim, B.I., Jeong, S., Shin, J., Koo, J., Chae, J., and Lee, S. 2009. A Layout- and data-driven generic simulation model for semiconductor fab. *IEEE Transactions on Semiconductor Manufacturing* 22(2): 225–231.

Kim, J.H., Lee, T.E., Lee, H.Y., Park, D.B. 2003. Scheduling analysis of time-constrained dual-armed cluster tools. *IEEE Transactions on Semiconductor Manufacturing* 16(3): 521–534.

Kim, S. 2006. Gradient-based simulation optimization. In *Proceedings of the 2006 Winter Simulation Conference*, pp. 159–167.

Kim, T.G. 1995. DEVS framework in discrete event systems modeling simulation. In *Proceedings of the 1995 KSS Fall Conference*. The Korea Society for Simulation, pp. 3–28.

Kiviat, P.J., Villanueva, R., and Markowitz, H. 1969. *The SIMSCRIPT II Programming Languages*. Upper Saddle River, NJ: Prentice Hall.

Kleinberg, J.M. 2000. Navigation in a small world. *Nature* 406: 845.

Kuhl, F., Weatherly, R., and Damann, J. 2000. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Upper Saddle River, NJ: Prentice Hall.

Law, A.M. 2007. *Simulation Modeling & Analysis*, 4th Ed. New York: McGraw-Hill.

Lee, D., Choi, B.K., and Kong, J. 2010a. Timer embedded finite state machine modeling and its application. In *Proceedings of the 24th European Conference on Modeling and Simulation*, pp. 153–159.

Lee, D., Shin, H., and Choi, B.K. 2010b. Mediator approach to direct workflow simulation. *Simulation Modelling Practice and Theory* 18(5): 650–662.

Leemis, L. 2001. Input modeling techniques for discrete-event simulations. In *Proceedings of the 2001 Winter Simulation Conference*, pp. 600–610.

Manier, M.A. and Bloch, C. 2003. A classification for hoist scheduling problems. *The International Journal of Flexible Manufacturing Systems* 15(1): 37–55.

Martinez, J.C. 2001. EZStrobe: General-purpose simulation system based on activity cycle diagrams. In *Proceedings of the 2001 Winter Simulation Conference*, pp. 1556–1564.

Mealy, G.H., 1955. A method for synthesizing sequential circuits, *Bell Systems Technical Journal* 34(5): 1045–1079.

Mostafazadeh, A. 2004. Quantum mechanics of Klein-Gordon-type fields and quantum cosmology. *Annals of Physics* 309(1): pp.1–48.

Myers, R.H. and Montgomery, D.C. 1995. *Response Surface Methodology*. New York: John Wiley & Sons.

Nakayama, M.K. 2002. Simulation output analysis. In *Proceedings of the 2002 Winter Simulation Conference*, pp. 23–34.

Norman, M., Tinsley, D., Barksdale, J., Wiersholm, O., Campbell, P., and MacNair, E. 1999. Process and material handling models integration. In *Proceedings of the 1999 Winter Simulation Conference*, pp. 1262–1267.

Park, B.C., Park, E.S., Choi, B.K., Kim, B.H., and Lee, J.H. 2008. Simulation-based planning and scheduling system for TFT-LCD fab. In *Proceedings of the 2008 Winter Simulation Conference*, pp. 2271–2276.

Pegden, C.D. 1989. *Introduction to SIMAN*. State College, PA: System Modeling Corp.

Peterson, J.L. 1981. *Petri Net Theory and the Modeling of Systems*. Upper Saddle River, NJ: Prentice Hall.

Pidd, M. 2004. *Computer Simulation in Management Science*. 5th Ed. John Wiley & Sons.

Pillai, D.D., Cass, E.L., Dempsey, J.C., and Yellig, E.J. 2004. 300-mm full-factory simulations for 90- and 65-nm IC manufacturing. *IEEE Transactions on Semiconductor Manufacturing* 17(3): 292–298.

Pritsker, A.B. and Pegden, C.D. 1979. *Introduction to Simulation and SLAM*. West Lafayette, IN: Systems Pub. Corp.

Ren, S.C., Xu, D., Wang, F., and Tan, M. 2005. Timed event graph-based scheduling for cyclic permutation flow shop. *International Journal of Information Technology* 11(5): 10–17.

Richmond, B. 2003. *An Introduction to Systems Thinking*. High Performance Systems Inc.

Rockwell Automation. 2010. *Arena® User's Guide, Version 13.50*.

Rossetti, M.D. 2010. *Simulation Modeling and Arena*. New York: John Wiley & Sons.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. 1991. *Object-Oriented Modeling and Design*. Upper Saddle River, NJ: Prentice Hall.

Sargent, R.G. 2004. Validation and verification of simulation models. In *Proceedings of the 2004 Winter Simulation Conference*, pp. 17–28.

Savage, E.L. and Schruben, L.W. 1995. Eliminating event cancellation in discrete event simulation. In *Proceedings of the 1995 Winter Simulation Conference*, pp. 744–750.

Savage, E.L., Schruben, L.W., and Yücesan, E. 2005. On the generality of event-graph model. *INFORMS Journal on Computing* 17(1): 3–9.

Schmeiser, B.W. and Babu, A.J.G. 1980. Beta variate generation via exponential majorizing functions. *Operations Research* 28(4): 917–926.

Schruben, D.L. and Schruben, L.W. 2001. *Event Graph Modeling with SIGMA*, 4th Ed. Custom Simulations.

Schruben, D.L. and Schruben, L.W. 2006. *Event Graph Modeling Using SIGMA*, 5th Ed. Custom Simulations.

Schruben, D.L. and Yücesan, E. 1994. Transforming Petri nets into event graph models. In *Proceedings of the 1994 Winter Simulation Conference*, pp. 560–565.

Schruben, L.W. 1983. Simulation modeling with event graph models. *Communications of the Association of Computing Machinery* 26(11): 957–963.

Schruben, L.W. 1995. *Graphical Simulation Modeling and Analysis using SIGMA for Windows*. Danvers, MA: Boyd and Fraser Publishing Company.

Schruben, L.W. Lecture in the Pritsker Scholars Distinguished Lecture Series, School of Industrial Engineering, Purdue University, April 13, 2012.

Song, E., Choi, T., Choi, B.K., and Gu, S. 2011. A framework for integrated simulation of production and material handling systems of TFT-LCD fab. In *Proceedings of the IEEE 2011 Summer Computer Simulation Conference*, Hague, pp. 48–54.

Tocher, K.D. 1960. An integrated project for the design and appraisal of mechanized decision-making control systems. *Operational Research* 11(1/2): 50–65.

Wagner, F. 2005. Moore or Mealy model? Available at http://www.stateworks.com/active/download/TN10-Moore-Or-Mealy-Model.pdf.

Wagner, F. and Wolstenholme, P. 2003. Modeling and building reliable, re-useable software. In *Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'03)*, pp. 277–286.

Wagner, F., Schmuki, R., Wagner, T., and Wolstenholme, P. 2006. *Modeling Software with Finite State Machines: A Practical* Approach. Boca Raton, FL: Auerbach Publications.

Wang, F.K. and Lin, J.T. 2004. Performance evaluation of an automated material handling system for a wafer fab. *Robotics and Computer-Integrated Manufacturing* 20(2): 91–100.

WfMC (Workflow Management Coalition). 1995. *The Workflow Reference Model*, WfMC-TC-00-1003, Workflow Management Coalition, http://www.wfmc.org.

WfMC. 1998. *Workflow Management Application Programming Interface Spec*, WfMC-TC-1009, Workflow Management Coalition, http://www.wfmc.org.

Wu, B. 1992. *Manufacturing Systems Design and Analysis*. London: Chapman & Hall.

Zeigler, B.P. 1976. *Theory of Modelling and Simulation*. New York: John Wiley & Sons.

Zeigler, B.P., Praehofer H., and Kim, T.G. 2000. *Theory of Modeling and Simulation*, 2nd Ed. San Diego, CA: Academic Press.

INDEX

Modeling and Simulation of Discrete-Event Systems, First Edition. Byoung Kyu Choi and Donghun Kang.