SZENT ISTVÁN
EGYETEM
Gödöllő

Simulation of Logistical Processes

Logisztika
Oktatásért és
Kutatásért
Alapítvány

Szent István University, Gödöllő
2013.

János Benkő
university professor

Simulation of Logistical Processes

Szent István University, Gödöllő
2013.

CONTENTS

CHAPTER 1

Introduction to Modeling and Simulation

1.1. Problem solving

The problems facing industry, commerce, government, and society in general continue to grow in size and complexity. The need for procedures and techniques for resolving such problems is apparent. This book advocates the use of modeling and, in particular, simulation modeling for the resolution of problems. Simulation models can be employed at four levels:

as explanatory devices to define a system or problem,
as analysis vehicles to determine critical elements, components and issues,
as design assessors to synthesize and evaluate proposed solutions,
as predictors to forecast and aid in planning future developments.

In order to resolve problems using simulation models, it is necessary to understand the systems and to define problems relating to those systems. In our judgment, models' should be developed to resolve specific problems. The form of the model, although dependent on the problem solver's background, requires an organized structure for viewing systems. A simulation language provides such a vehicle. It also translates a model description into a form acceptable by a computing system. The computer is used to exercise the model to provide outputs that can be analyzed in order that decisions relating to problem resolution can be made.

The goal of this book is to provide useful information for problem solving. The book is both an introduction to simulation methodology and an introduction to the simulation language for alternative modeling, Arena. Arena supports the modeling of systems from diverse points of view. In this book, we model systems using these points of view and thus the book contains information on different methods of structuring models of systems.

Arena has been designed to support engineers, managers, and researchers. To do this it provides, in addition to modeling views, extensive input and output capabilities. Since many of today's problems are statistical in nature, the input and output' capabilities require a background in probability and statistics. Thus, parts of this book are devoted to presenting probabilistic and statistical concepts related to problem solving using simulation models.

In this chapter, we present general discussions and definitions of simulation-related topics and our suggested procedure for conducting projects that resolve problems by employing simulation models.

1.2. Systems

A system is a collection of items from a circumscribed sector of reality that is the object of study or interest. Therefore, a system is a relative thing. In one situation, a particular collection of objects may be only a small part of a larger system-a subsystem; in another situation that same collection of objects may be the primary focus of interest and would be considered as the system. The scope of every system, and of every model of a system, is determined solely by its reason for being identified and isolated. The scope of every simulation model is determined by the particular problems the model is designed to solve.

To consider the scope of a system, one must contemplate its boundaries and contents. The boundary of a system may be physical; however, it is better to think of a boundary in terms of cause and effect. Given a tentative system definition, some external factors may affect the system. If they completely govern its behavior, there is no merit in experimenting with the defined system. If they partially influence the system, there are several possibilities:

The system definition may be enlarged to include them.

They may be ignored.

They may be treated as inputs to the system.

If treated as inputs, it is assumed that the factors are functionally specified by prescribed values, tables, or equations. For example, when defining the model of a company's manufacturing system, if the sales of the company's product are considered as inputs to the manufacturing system, the model will not contain a cause and effect sales relation; it only includes a statistical description of historical or predicted sales, which is used as an input. In such a model of the manufacturing system the sales organization is outside the boundaries of the "defined" system. In systems terminology, objects that are outside the boundaries of the system, but can influence it, constitute the environment of the system. Thus, systems are collections of mutually interacting objects that are affected by outside forces. *Figure 1.1* shows such a system.

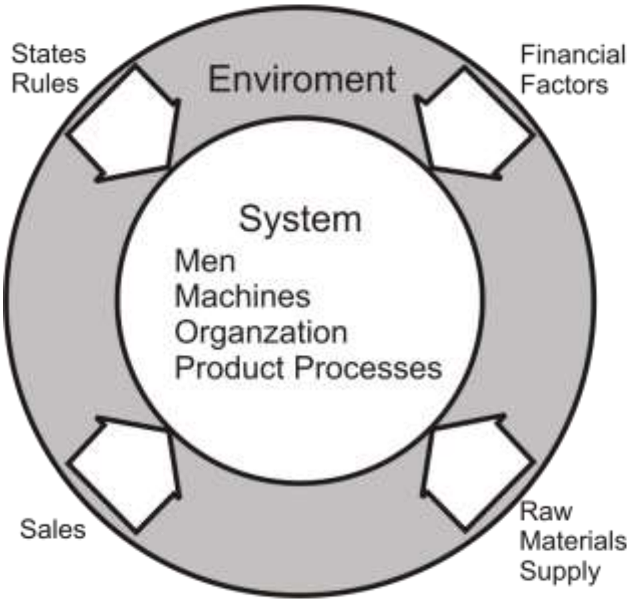


Figure 1.1 Manufacturing system model showing external influences.

Models are descriptions of systems. In the physical sciences, models are usually developed based on theoretical laws and principles. The models may be scaled physical objects (iconic models), mathematical equations and relations (abstract models), or graphical representations (visual models). The usefulness of models has been demonstrated in describing, designing, and analyzing systems. Many students are educated in their discipline by learning how to build and use models. Model building is a complex process and in most fields is an art. The modeling of a system is made easier if: 1) physical laws are available that pertain to the system; 2) a pictorial or graphical representation can be made of the system; and 3) the variability of system inputs, elements, and outputs is manageable [18].

The modeling of complex, large-scale systems is often more difficult than the modeling of physical systems [21] for the following reasons: 1) few fundamental laws are available; 2) many procedural elements are involved which are difficult to describe and represent; 3) policy inputs are required which are hard to quantify; 4) random components are significant elements; and 5) human decision making is an integral part of such systems. Through the use of a simulation approach, we will illustrate methods for alleviating these difficulties.

1.4. Model building

Since a model is a description of a system, it is also an abstraction of a system. To develop an abstraction, a model builder must decide on the elements of the system to include in the model. To make such decisions, a purpose for model building should be established. Reference to this purpose should be made when deciding if an element of a system is significant and, hence, should be modeled. The success of a modeler depends on how well he or she can 'define significant elements and the relationships between elements.

A pictorial view of our proposed model building approach is shown in *Figure 1.2*. A system as discussed in Section 1.2 is considered as a set of interdependent objects united to perform a specified function. The concept of a system is not well-defined. A particular definition of a system's objects, and their function is subjective and depends on the individual who is defining the system. Because of this, the first step of our approach is the development of a purpose for modeling that is based ' on a stated problem or project goal. Based on this purpose, the boundaries of the system and a level of modeling detail are established. This abstraction results in a model that smooths out many of the rough ill-defined edges of the actual system. We also include in the model the desired performance measures and design alternatives to be evaluated. These can be considered as part of the model or as inputs to the model. Assessments of design alternatives in terms of the specified performance measures are considered as model outputs. Typically, the assessment process requires redefinitions and redesigns. In fact, the entire model building approach is performed iteratively. When recommendations can be made based on the assessment of alternatives, an implementation phase is initiated. Implementation should be carried out in a well.-defined environment with an explicit set of recommendations. Major decisions should have been made before implementation is attempted.

Simulation models are ideally suited for carrying out the problem-solving approach illustrated in *Figure 1.2*. Simulation provides the flexibility to build either aggregate or detailed models. It also supports the concepts of iterative model building by allowing models to be embellished through simple and direct additions. These aspects of simulation models are described in the next section.

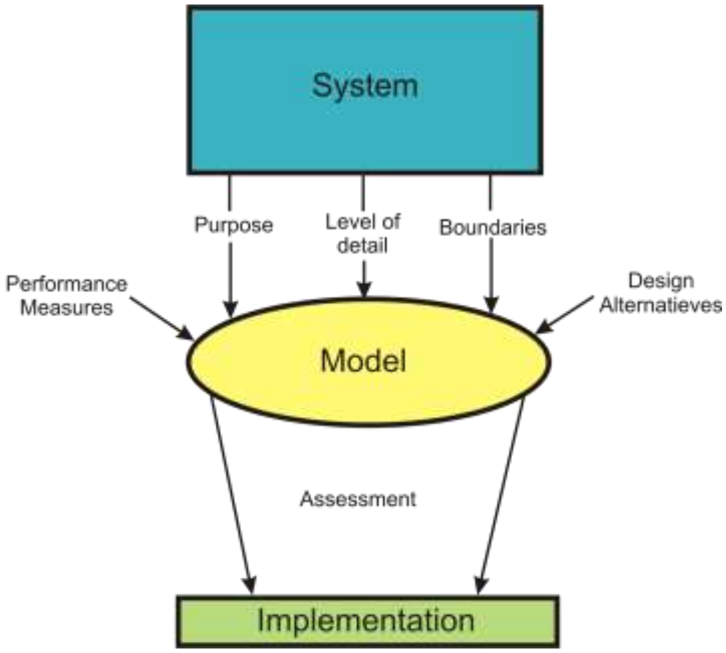


Figure 1.2 A model building approach for problem solving.

1.5 Definition of simulation

In its broadest sense, computer simulation is the process of designing a mathematical-logical model of a real system and experimenting with this model on a computer [17, 20]. Thus simulation encompasses a model building process as well as the design and implementation of an appropriate experiment involving that model. These experiments, or simulations, permit inferences to be drawn about systems

Without building them, if they are only proposed systems;

Without disturbing them, if they are operating systems that are costly or unsafe to experiment with;

Without destroying them, if the object of an experiment is to determine their limits of stress.

In this way, simulation models can be used for design, procedural analysis, and performance assessment [17].

Simulation modeling assumes that we can describe a system in terms acceptable to a computing system. In this regard, a key concept is that of a system state description. If a system can be characterized by a set of variables, with each combination of variable values representing a unique state or condition of the system, then manipulation of the variable values simulates movement of the system from state to state. A simulation experiment involves observing the dynamic behavior of a model by moving from state to state in accordance with well-defined operating rules designed into the model.

Changes in the state of a system can occur continuously over time or at discrete instants in time. The discrete instants can be established deterministically or stochastically depending on the nature of model inputs. Although the procedures for describing the dynamic behavior of discrete and continuous change models differ, the basic concept of simulating a system by portraying the changes in the state of the system over time remains the same. In the next section, we will illustrate the type of dynamic behavior obtained from a simulation experiment with a discrete change model.

1.6 Simulation of a bank teller

As an example of the concept of simulation, we will examine the processing of customers by a teller at a bank. Customers arrive to the bank, wait for service by the teller if the teller is busy, are served, and then depart the system. Customers arriving to the system when the teller is busy wait in a single queue in front of the teller. For simplicity, we assume that the time of arrival of a customer and the service time by the teller for each customer are known. These values are given in *Table 1*.1. Our objective is to manually simulate the above system to determine the percent of time the teller is idle and the average time a customer spends at the bank.

Since a simulation is the dynamic portrayal of the changes in the state of a system over time, the states of the system must be defined. For this example, they can be defined by the status of the teller (busy or idle) and by the number of customers at the bank. The state of the system is changed by: 1) a customer arriving to the bank; and 2) the completion of service by the teller and subsequent departure of the customer. To illustrate a simulation, we will determine the state of the system over time by processing the events corresponding to the arrival and departure of customers in a time-ordered sequence.

The manual simulation of this example corresponding to the values in *Table 1.1* is summarized in *Table 1.2* by customer number. It is assumed that initially there are no customers in the system, the teller is idle, and the first customer is to arrive at time 3.2.

In *Table 1.2*, columns (1) and (2) are taken from *Table 1.1*. The start of service time given in column (3) depends on whether the preceding customer has departed the bank. It is taken as the larger value of the arrival time of the customer and the departure time of the preceding customer: column (4), the departure time, is the sum of the column (3) value and the service time for the customer given in *Table 1.1*. Values for time in queue and time in bank for each customer are computed as shown in *Table 1.2*. Average values per customer for these variables are 2.61 minutes and 5.81 minutes, respectively.

Table 1.1

Customer arrival and service times

Customer Number	Time of Arrival (Minutes)	Service Time (Minutes)
1	3.2	3.8
2	10.9	3.5
3	13.2	4.2
4	14.8	3.1
5	17.7	2.4
6	19.8	4.3
7	21.5	2.7
8	26.3	2.1
9	32.1	2.5
10	36.6	3.4

Table 1.2 presents a good summary of information concerning the customer but does not provide information about the teller and the queue size for the teller. To portray such information, it is convenient to examine the events associated with the situation.

Table 1.2

Manual simulation of bank teller

Customer Number (1)	Arrival Time (2)	Start Service Time (3)	Departure Time (4)	Time in Queue (5)=(3)–(2)	Time in Bank (6)=(4)–(2)
1	3.2	3.2	7.0	0.0	3.8
2	10.9	10.9	14.4	0.0	3.5
3	13.2	14.4	18.6	1.2	5.4
4	14.8	18.6	21.7	3.8	6.9
5	17.7	21:7	24.1 "	4.0	6.4
6	19.8	24.1	28.4	4.3	8.6
7	21.5	28.4	31.1	6.9	9.6
8	26.3	31.1	33.2	4.8	6.9
9	32.1	33.2	35.7	1.1	3.6
10	36.6	36.6	40.0	0.0	3.4

The logic associated with processing the arrival and departure events depends on the state of the system at the time of the event. In the case of the arrival event, the disposition of the arriving customer is based on the status of the teller. If the teller is idle, the status of the teller is changed to busy and the departure event is scheduled for the customer by adding his service time to the current time. However, if the teller is busy at the time of an arrival, the customer cannot begin service at the current time arid, therefore, he enters the queue (the queue length is increased by 1). For the departure event, the logic associated with processing the event is based on queue length. If a customer is waiting in the queue, the teller status remains busy, the queue length is reduced by 1, and the departure event for the first waiting customer is scheduled. However, if the queue is empty, the status of the teller is set to idle.

An **event-oriented** description of the bank teller status and the number of customers at the bank is given in *Table 1.3*. In *Table 1.3*, the events are listed in chronological order. A graphic portrayal of the status variables over time is shown in *Figure 1.3*. These results indicate that the average number of customers at the bank in the first 40 minutes is 1.4525 and that the teller is idle 20 percent of the time.

Table 1.3

Event-oriented description of bank teller simulation

Event Time	Customer Number	Event Type	Number in Queue	Number in Bank	Teller Status	Teller Idle Time
0.0	-	Start	0	0	Idle	-
3.2	1	Arrival	0	1	Busy	3.2
7.0	1	Departure	0	0	Idle	
10.9	2	Arrival	0	1	Busy	3.9
13.2	3	Arrival	1	2	Busy	
14.4	2	Departure	0	1	Busy	
14.8	4	Arrival	1	2	Busy	
17.7	5	Arrival	2	3	Busy	
18.6	3	Departure	1	2	Busy	
19.8	6	Arrival	2	3	Busy	
21.5	7	Arrival	3	4	Busy	
21.7	4	Departure	2	3	Busy	
24.1	5	Departure	1	2	Busy	
26.3	8	Arrival	2	3	Busy	
28.4	6	Departure	1	2	Busy	
31.1	7	Departure	0	1	Busy	
32.1	9	Arrival	1	2	Busy	
33.2	8	Departure	0	1	Busy	
35.7	9	Departure	0	0	Idle	
36.6	10	Arrival	0	1	Busy	0.9
40.0	10	Departure	0	0	Idle	

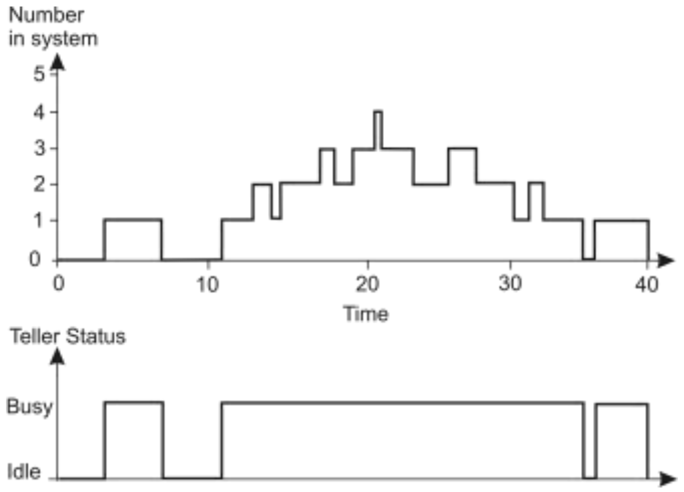


Figure 1.3. Graphic portrayal of bank teller simulation

In order to place the arrival and departure events in their proper chronological order, it is necessary to maintain a record or calendar of future events to be processed. This is done by maintaining the times of the next arrival event and next departure event. The next event to be processed is then selected by comparing these event times. For situations with many events, an order0ed list of events would be maintained which is referred to as an event file or event calendar.

There are several important concepts illustrated by this example. We observe that at any instant in simulated time, the model is in a particular state. As events occur, the state of the model may change as prescribed by the logical-mathematical relationships associated with the events. Thus, the events define the dynamic structure of the model. Given the starting state, the logic for processing each event, and a method for specifying sample values, our problem is largely one of bookkeeping. An essential element in our bookkeeping scheme is an event calendar which provides a mechanism for recording and sequencing future events. Another point to observe is that we can view the state changes from two perspectives: the process that the customer encounters as he seeks service (the customer's view); or the events that cause the state of the teller to change (the teller's or bank's view). These so called world views are described in detail in Chapter 3.

1.7 The simulation process

As we alluded to earlier, the process for the successful development of a simulation model consists of beginning with a simple model which is embellished in an evolutionary fashion to meet problem-solving requirements. Within this process the following stages of development can be identified.

1. Problem Formulation
The definition of the problem to be studied including a statement of the problem-solving objective.

2. Model Building
The abstraction of the system into mathematical-logical relationships in accordance with the problem formulation.

3. Data Acquisition
The identification, specification, and collection of data.

4. Model Translation
The preparation of the model for computer processing.

5. Verification
The process of establishing that the computer program executes as intended.

6. Validation
The process of establishing that, a desired accuracy or correspondence exists between the simulation model and the real system.

7. Strategic and Tactical Planning
The process of establishing the experimental conditions for using the model.

8. Experimentation
The execution of the simulation model to obtain output values.

9. Analysis of Results
The process of analyzing the simulation outputs to draw inferences and make recommendations for problem resolution.

10. Implementation and Documentation
The process of implementing decisions resulting from the simulation and documenting the model and its use.

The stages listed are a slightly modified version of those presented by Shannon [22].

Although some of these steps were discussed in conjunction with model building, we prefer to restate them here due to the importance of the concepts [4, 5, 10, 13].

The first task in a simulation project is the construction of a clear definition of the problem and an explicit statement of the objectives of the analysis. Because of the evolutionary nature of simulation, problem definition is a continuing process which typically occurs throughout the duration of the study. As additional insights into the problem are gained and additional questions become of interest, the problem definition is revised accordingly.

Once an initial problem statement is formulated; the task of formulating a model of the system begins. The model of a system consists of both a static and dynamic description. The static description defines the elements of the system and the characteristics of the elements. The dynamic description defines the way in which the elements of the system interact to cause changes to the state of the system over time.

The actual process of formulating the model is one which is largely an art. The modeler must understand the structure and operating rules of the system and be able to extract the essence of the system without including unnecessary detail. The model should be easily understood, yet sufficiently complex to realistically reflect the important characteristics of the real system. The crucial decisions concern what simplifying assumptions are valid, what elements should be included in the model, and what interactions occur between the elements. The amount of detail included in the model should be based on the purpose for which the model is being built. Only those elements that could cause significant differences in decision-making need be considered.

Both the problem formulation and modeling phases require close interaction among project personnel. "First cut" models should be built, analyzed, and discussed. In many cases, this will require heroic assumptions and a willingness on the part of the modeler to expose his incomplete knowledge of the system under study. However, an evolutionary modeling process allows inaccuracies to be discovered more quickly and corrected more efficiently than would otherwise be possible. Furthermore, the close interaction in the problem definition and model formulation phases creates confidence in the model on the part of the model user and therefore helps to ensure a successful implementation of simulation results.

The model formulation phase will generate data input requirements for the model. Some of the data required may be readily available while other data requirements may involve considerable time and cost for collection. Typically, such data input values are initially hypothesized or based on a preliminary analysis. In some cases, the exact values for one or more of the input parameters may have little effect on the simulation results. The sensitivity of the simulation results to changes in the input data to the model can be evaluated by making a series of simulation runs while varying the input parameter values. In this way, the simulation model can be used to determine how best to allocate money and time in refining the input data to the model.

Once a model has been developed and, initial estimates have been established for the input data, the next task is to translate the model into a computer acceptable form: Although a simulation model can be programmed using a general purpose language, there are distinct advantages to using a simulation language. In addition to the savings in programming time, a simulation language also assists in model formulation by providing a set of concepts for articulating the system description. In this text, we describe in detail the use of the Arena simulation language which provides a graphical vehicle that combines the model formulation and translation task into a single activity. Arena also includes a programming capability to allow models to be embellished in an evolutionary fashion to any level of detail required to reflect the complexities of the system being studied.

The verification and validation stages are concerned with evaluating the performance of the simulation model. The verification task consists of determining that the translated model executes on the computer as the modeler intended. This is typically done by manual checking of calculations. Fishman and Kiviat [11] describe statistical methods which can aid in the verification process: The validation task consists of determining that the simulation model is a reasonable representation of the system [10]. Validation is normally performed in levels. We recommend that a validation be performed on data inputs, model elements, subsystems, and interface points. Validation of simulation models, although difficult, is a significantly easier task than validating other types of models, for example, validating a linear programming formulation. In simulation models, there is a correspondence between the model elements and system elements. Hence, testing for reasonableness involves a comparison of model and system structure and comparisons of the number of times elemental decisions or subsystem tasks are performed.

Specific types of validation involve evaluating reasonableness using all constant values in the simulation model or assessing the sensitivity of outputs to parametric variation of data inputs. In making validation studies, the comparison yardstick should encompass both past system outputs and experiential knowledge of system performance behavior. A point to remember is that past system outputs are but one sample record of what could have happened.

Strategic and tactical planning refer to the task of establishing the experimental conditions for the simulation runs [4]. The strategic planning task consists of developing an efficient experimental design either to explain the relationship between the simulation response and the controllable variables or to find the combination of values for the controllable variables which either minimize or maximize the simulation response. In contrast, tactical planning is concerned with how each simulation within the experimental design is to be made to glean the most information from the data. Two specific issues in tactical planning are the starting conditions for simulation runs and methods for reducing the variance of the mean response.

The next stages in the simulation development process are` experimentation and analysis of results. These phases of simulation development involve the exercising of the simulation model and the interpretation of the outputs. When simulation results are used to draw inferences or to test hypotheses, statistical methods should be employed.

The final stages in the simulation development process are the implementation of results and the documentation of the simulation model and its use. No simulation project should be considered complete until its results are used in the decision-making process. The success of the implementation task is largely dependent upon the degree to which the modeler has successfully performed the other activities in the simulation development process. If the model builder and model user have worked closely together and both understand the model and its outputs, then it is likely that the results of the project will be implemented with vigor. On the other hand, if the model formulation and underlying assumptions are not effectively communicated, then it is more difficult to have recommendations implemented, regardless of the elegance and validity of the simulation model.

The stages of simulation development outlined above are rarely performed in a structured sequence beginning with problem definition and ending with documentation. A simulation project may involve false starts, erroneous assumptions which must later be abandoned, reformulation of the problem objectives, and repeated evaluation and redesign of the model. If properly done, however, this iterative process should result in a simulation model which properly assesses alternatives and enhances the decision-making process.

1.8 Chapter summary

Simulation is a technique that has been employed extensively to solve problems. Simulation models are abstractions of systems. They should be built quickly, explained to all project personnel, and changed when necessary. The implementation of recommendations to improve system performance is an integral part of the simulation methodology.

CHAPTER 2

Probability and statistics

2.1 Introduction

Systems to be simulated are generally composed of one or more elements that have uncertainty associated with them. Such systems evolve through time in a manner that is not completely predictable and are referred to as stochastic systems. The modeling of stochastic systems requires that the variability of the elements in the system be characterized using probability concepts. The outputs from a simulation model are also probabilistic, and therefore statistical interpretations about them are usually required. Although the reader will likely have some familiarity with probability and statistics, this chapter is included to provide a review of important probability and statistics concepts related to simulation modeling and analysis. We presume the reader has had previous exposure to probability and statistics at the level of one of the introductory books listed at the end of the chapter [6, 18, 23].

2.2 Experiment, sample space, and outcomes

An experiment is a well defined procedure or process whose outcome is observable but is not known with certainty in advance. The set of all possible outcomes is called the sample space. If the sample space is finite or countable infinite, it is said to be discrete; otherwise it is continuous.

Outcomes can be combined to form new outcomes[1] by the set theory operations of union ($\cup$) and intersection ($\cap$). If the outcome C is defined as the union of a set of outcomes A and a set of outcomes B, denoted $C = A \cup B$, then C consists of the set of all outcomes within A or B. If the outcome D consists of the intersection of A and B, denoted $D=A \cap B$, then D consists of the set of outcomes that are in both A and B.

As an example of the concepts discussed above, consider the operation of a single teller bank system. Customers arrive to the bank, possibly wait, and are processed by the teller. Both the time between arrivals of customers to the bank and the service time by the teller will be assumed to exhibit variability. Let us define our first experiment as observing the time between customer arrivals to the bank. The sample space for the experiment consists of all possible observations for the time between arrivals. Since the time between customer arrivals can be any nonnegative real value, the sample space is continuous. An outcome is defined as any subset of the sample space, and therefore one possible outcome could be defined as the occurrence of an interarrival time between 8 and 9 minutes.

As a second example, consider the experiment of observing the number of customers processed during the first hour of operation. The number of customers processed during the first hour can be any of the values 0, 1, 2, 3, ..., that is, the set of nonnegative integers. In this case, the sample space is discrete. One possible outcome could be defined as the processing of five customers during the one hour period.

[1] Typically, the term "event" is used to describe combinations of outcomes. In simulation terminology an event is defined differently so we avoid its use here.

2.3 Probability

The probability of an outcome is a measure of the degree of likelihood that the outcome will occur. More formally, a probability measure is a function *P*() which maps outcomes into real numbers and satisfies the following axioms of probability:

1. $0 \le P(E) \le 1$ for any outcome *E*

2. $P(S) = 1,$ where S is the sample space or "certain outcome"

3. If $E_1, E_2, E_3, \ldots$ are mutually exclusive outcomes, then

$$P(E_1 \cup E_2 \cup E_3 \cup \ldots) = P(E_1) + P(E_2) + P(E_3) + \ldots$$

From these three axioms and the rules of set theory, the basic laws of probability can be derived. However, these axioms are not sufficient to compute the probability of an outcome. Numerical values for probabilities are usually difficult to obtain; nevertheless it is useful to postulate their existence.

In some simple cases, the exact probability of an outcome can be calculated using combinatorial analysis. Examples of this include determining the probability of h heads in n tosses of a fair coin and computing the probability of three aces in a five card hand. However, in most cases the exact probability of an outcome cannot be calculated. In such cases, an approximate value for the probability of an outcome can sometimes be obtained using the frequency interpretation of probability. If we repeat an experiment *n* times and outcome *E* occurs *k*/*n* is the proportion of times that *E* occurs. The probability of *E* can be interpreted as

$$P(E) = \lim_{n \to \infty} \frac{k}{n},$$

assuming the limit exists. By selecting a sufficiently large value of *n*, the proportion of occurrences *k*/*n*, approximates the probability of *E*. The approximate values for probabilities obtained in this way can be shown to satisfy the axioms of probability stated earlier. The practical limitation of this approach is that it is some times not possible or economical to perform the required experimentation.

2.4 Random variables and probability distributions

A function which assigns a real number to each outcome in the sample space is called a random variable. Discrete random variables are those that take on a finite or a countable infinite set of values. Continuous random variables can take on a continuum of values. In our example of the bank teller system, the interarrival time is a continuous random variable and the number of customers processed during the first hour is a discrete random variable.

A probability distribution is any rule which assigns a probability to each possible value of a random variable. The rule for assigning probabilities takes on two distinct forms depending upon whether the random variable is discrete or continuous.

For discrete random variables, the probability associated with each value of the random variable is commonly specified using a probability mass function, *p*(*x*), defined as[2]

$$p(x_i) = P(X = x_i)$$

[2] When presenting probability and statistics concepts, we will attempt to use a capital letter to indicate a random variable and a lower case letter to indicate an observed potential numerical value. Sometimes this is not feasible; however, the context of the discussion should clarify the situation.

For each possible value x_i the function assigns a specific probability that the random variable X assumes the value x_i. The axioms of probability impose the following restrictions on $p(x_i)$.

$$0 \le p(x_i) \le 1 \text{ for all } i$$

$$\sum_{all\, i} p(x_i) = 1$$

An aiternate representation for the probability distribution is the cumulative distribution function, $F(x)$, defined as follows:

$$F(x) = P(X = x)$$

In this case, the function $F(x)$ specifies the probability that the random variable X assumes a value less than or equal to x. From the axioms of probability, $F(x)$ must have the following properties:

$$0 \le F(x) \le 1 \text{ for all } i$$

$$F(-\infty) = 0,$$

$$F(\infty) = 1.$$

The distribution function is related to the probability mass function by

$$F(x) = \sum_{x_i \le x} p(x_i) = 1$$

As an example of a discrete probability distribution, consider an experiment consisting of three tosses of a fair coin. Let the random variable X denote the number of heads obtained from the three losses. The random variable X can assume the discrete value of 0, 1, 2, or 3. There are eight possible outcomes of which 1 has 0 heads, 3 have 1 head, 3 have 2 heads and 1 has 3 heads. The probability mass function for the random variable X is depicted in *Figure 2.1* and the cumulative distribution function is depicted in *Figure 2.2*.

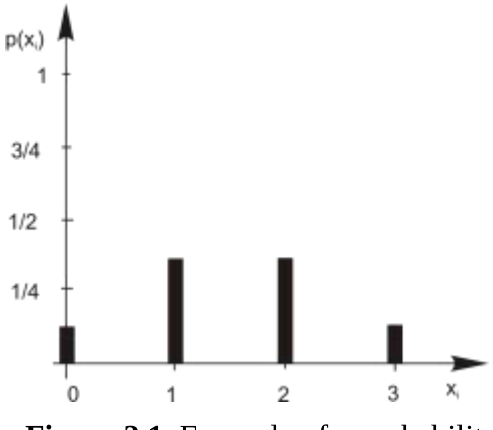


Figure 2.1. Example of a probability mass function

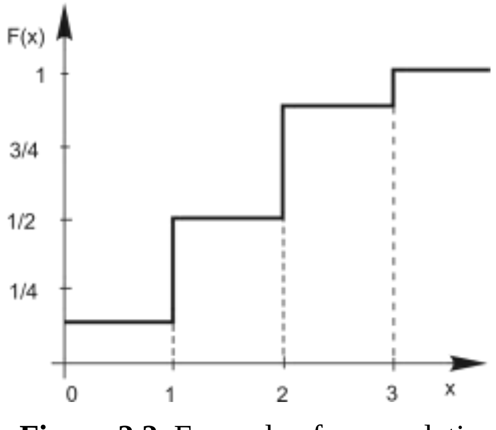


Figure 2.2. Example of a cumulative distribution function

For continuous random variables, a different form for the probability distribution is required. Since the random variable can assume any of an uncountably infinite number of values, the probability of a specific value is zero. This does not say that the value is impossible, but that the value is extremely unlikely given the infinite number of alternative values. However, the probability that the variable assumes a value in the interval between two distinct points a and b will generally not be zero. Therefore, the probability mass function as defined for the discrete case is replaced in the continuous case by the probability density function, $f(x)$, defined according to the following relationship

$$P(a \le X \le b) = \int_a^b f(x)dx$$

Thus the probability density function, when integrated between *a* and *b*, gives the probability that the random variable will assume a value in the interval between *a* and *b*. To be consistent with the axioms of probability, the probability density function must satisfy the following conditions

$$f(x) \ge 0$$

$$\int_{-\infty}^{\infty} f(x)dx = 1$$

The cumulative distribution function, *F*(*x*), defined for the continuous case is

$$F(x) = \int_{-\infty}^{x} f(y)dy = P(X \le x)$$

The function *F*(*x*) defines the probability that the continuous random variable *X* assumes a value less than or equal to *x*.

As an example of a continuous probability distribution, consider a random variable *X* which can assume any value in the range between 0 and 1. Assuming that each of the uncountably infinite number of possible values are equally likely, the corresponding probability density function and cumulative distribution function are shown in *Figures 2.3* and *2.4*, respectively. The probability that the random variable *X* assumes a value in the interval between 0.50 and 0.75 is the area under the probability density function between 0.50 and 0.75. For the random variable depicted in *Figure 2.3*, this probability is equal to 0.25.

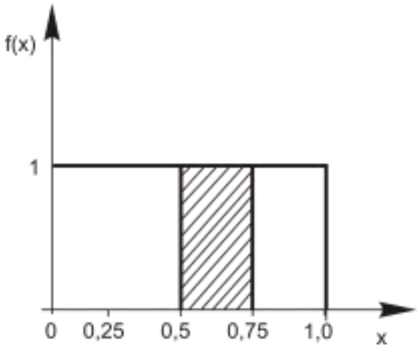


Figure 2.3. Example of a probability density function

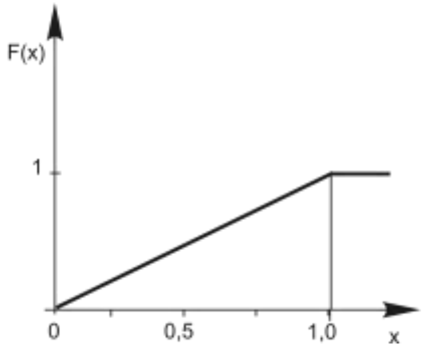


Figure 2.4. Example of a cumulative distribution function

A random variable can also be both continuous and discrete. Random variables of this type are referred to as having "mixed" distributions. A random variable having a mixed distribution can assume either discrete values with finite probabilities or a continuum of values as prescribed by a probability density function. *Figure 2.5* depicts such a distribution where the discrete values 1 and 2 each occur with a probability of 1/3 as denoted by the spikes on the graph. The values between 1 and 2 are governed by the density function *f*(*x*) = 1/3.

Such a distribution would result if samples were drawn from a continuous distribution with equally likely values in the range from 0 to 3, with values greater than 2 set to 2, and values less than 1 set to 1. The equation for the cumulative distribution function for this random variable is

$$F(x)=\begin{cases}0 & x<1\\ \dfrac{1}{3}+\dfrac{x-1}{3} & 1\le x\le 2\\ 1 & x\ge 2\end{cases}$$

From this equation and *Figure 2.5*, we see that $F(x)$ has discontinuities at x=1 and x=2. At points of discontinuity, $P(X=x)$ is equal to the jump which $F(x)$ makes at the point x. For example, $P(X=1)=1/3$. However, for l<x<2, $F(x)$ is continuous at x and $P(X=x)=0$.

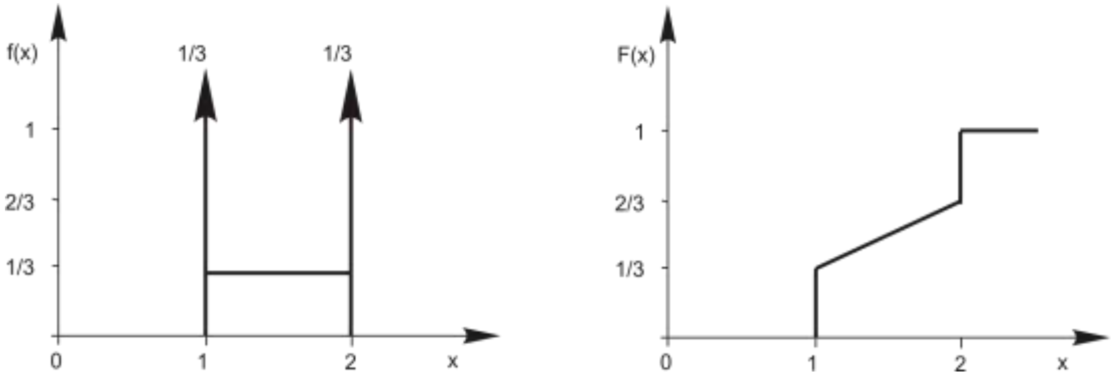


Figure 2.5. Example of a mixed distribution

Distribution functions capture the probabilistic characteristic of a variable. Some of the common distribution functions are: norm al or *Gaussian, uniform, triangular, exponential, lognormal, Erlang, beta, gamma* and *Poisson*. These distributions are presented in graphic form in Appendix. Also, a discussion is given of when it is appropriate to use a particular distribution.

Procedures for generating observations for computer simulation from these distributions require the use of a random number which, when generated on a digital computer, is referred to as a pseudo random number. Pseudo random numbers are uniformly distributed random samples between the values of 0 and 1 (see *Figure 2.3*) with successive samples having the property of perceived independence. A random observation generated from a distribution using a pseudorandom number is referred to as a random variate, a random deviate, or a random sample. What is meant by these terms is that a collection of the random observations approximates the underlying distribution from which the random samples were generated, that is, the observations when all taken together characterize the distribution from which the random samples were obtained.

2.5 Expectation and moments

It is sometimes desirable to characterize a random variable by one or more values which summarize information contained in its probability distribution function. The expectation or expected value of a random variable X, denoted $E[X]$, is such a value and is defined as follows:

$$E(X)=\sum_i x_i\,p(x_i)\,,\qquad \text{when } X \text{ is discrete,}$$

$$E(X)=\int_x x\,f(x)\,dx\,,\qquad \text{when } X \text{ is continuous.}$$

The expectation is a probability weighted average of all possible values of X and therefore a measure of centrality for the distribution. For this reason, it is called the mean value.

Expectations can be taken of functions of random variables. In particular, the expectation of X^n is defined as the n^th moment of a random variable and can be expressed as follows

$$E\{X^n\} = \sum_i x_i^n \, p(x_i)\,, \qquad \text{when } X \text{ is discrete,}$$

$$E\{X^n\} = \int_x x^n f(x)\, dx\,, \qquad \text{when } X \text{ is continuous.}$$

The expected value is a special case of the above when n=1, and, hence, is called the first moment.

A variant of the nth moment is the nth moment about the mean which is defined as

$$E\{[X - E(X)]^n\}\,,$$

In this case, the expected value of X is subtracted from X before computing the n^th moment.

A moment of particular importance in probability theory is the second moment about the mean, commonly referred to as the variance of X, and is denoted as σ^2 or Var(X). The variance of a random variable is a measure of the spread of the probability distribution. If a random variable has a small variance, then samples tend to occur near the expected value. The square root of the variance is referred to as the standard deviation of the random variable.

If X and Y are random variables, then the covariance of X and Y, denoted Cov(X,Y), is defined as

$$\text{Cov}(X,Y) = E\{[X - E(X)][Y - E(Y)]\}.$$

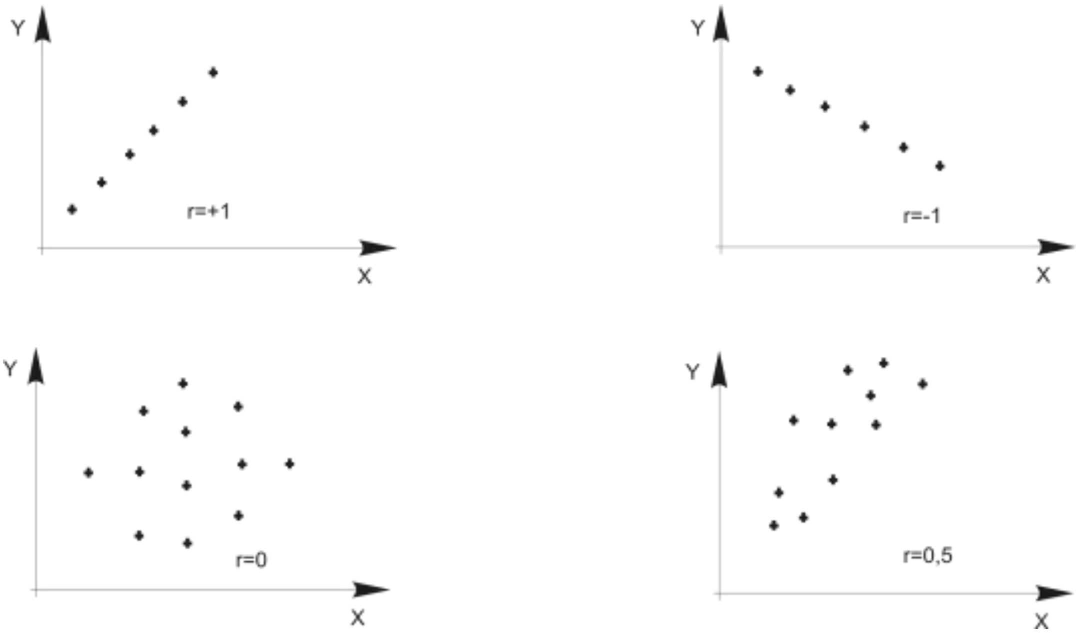


Figure 2.6, Scatter diagrams of Y versus X for various values of r.

The covariance is important because it measures the linear association, if any, between X and Y. If the outcome of X has no influence on the outcome of Y, then X and $Y$are said to be independent and the Cov(X,Y) will be zero. More formally, X and Y are independent if and only if:

$p(y|x) = p(y)$ in the discrete case where $p(y|x)$ is the probability that Y=y given X=x,

$f(y|x) = f(y)$ in the continuous case where $f(y|x)$ is the conditional density function of *Y* for *X*=*x*.

These statements specify that the probability distribution of *Y* given knowledge of *X* is the same as the probability distribution of *Y* without knowledge of *X*.

A measure of dependence which is related to the covariance is the correlation coefficient, *r*, defined as

$$r = \frac{\mathrm{Cov}(X,Y)}{\sqrt{\mathrm{Var}(X)\cdot\mathrm{Var}(Y)}}$$

The correlation coefficient has a range from −1 to 1 with a value of zero indicating no correlation between *X* and *Y*. A positive sign indicates that *Y* tends to be high when *X* is high, and a negative sign indicates that *Y* tends to be low when *X* is high. The magnitude of *r* indicates the degree of linearity of *Y* plotted against *X*. If a plot of *Y* versus *X* is a straight line, then $r=\pm1$. If *X* and *Y* are independent, then a plot of *Y* versus *X* will produce random points and *r* will equal zero. Typical scatter diagrams of *Y* versus *X* for different values of *r* are depicted in *Figure 2.6*.

2.5.1 Theorem on Total Probability

The probability of the outcome *B* is equal to the sum of the conditional probabilities associated with *B* given the occurrence of mutually exclusive and exhaustive outcomes A_i weighted by the probability of A_i that is,

$$P(B) = \sum_i P(B|A_i)P(A_i)$$

An analogous result for the expectation of a random variable, *Y*, is

$$E(Y) = \sum_i E(Y|X = x_i)\,P(X = x_i)$$

2.5.2 Joint Probabilities

The probability of the joint outcome associated with a set of random variables can be expressed as a product of conditional probabilities

$$P(Y_1, Y_2, ..., Y_n) = P(Y_1)P(Y_2|Y_1)P(Y_3|Y_1, Y_2)...P(Y_n|Y_1...Y_{n-1}).$$

If we assume the random variables have the Markovian property that $P(Y_j|Y_1, Y_2, ..., Y_{j-1}) = P(Y_j|Y_{j-1})$, we have

$$P(Y_1, Y_2, ..., Y_n) = P(Y_1)P(Y_2|Y_1)P(Y_3|Y_2)...P(Y_n|Y_{n-1}).$$

If independence is assumed then

$$P(Y_1, Y_2, ..., Y_n) = P(Y_1)P(Y_2)P(Y_3)...P(Y_n).$$

2.6 Functions of random variables

A function of a random variable is itself a random variable. In this section, we summarize several important properties of functions of random variables.

If *X* and *Y* are random variables and k is any arbitrary constant, then the following properties for expectation can be derived:

$$E(X+Y)=E(X)+E(Y)$$
$$E(kX)=kE(X)$$
$$E(X+k)=E(X)+k$$

The similar properties for variances are less obvious than those· for expectation and are

$$\mathrm{Var}(X+Y)=\mathrm{Var}(X)+\mathrm{Var}(y)+2\mathrm{Cov}(X,Y)$$
$$\mathrm{Var}(kX)=k^2\,\mathrm{Var}(X)$$
$$\mathrm{Var}(X+k)=\mathrm{Var}(X)$$
$$\mathrm{Var}(kX+nY)=k^2\,\mathrm{Var}(X)+n^2\,\mathrm{Var}(Y)+2kn\mathrm{Cov}(X,Y)$$

Note that if *X* and *Y* are independent random variables, then the Cov(*X*,*Y*) is zero, and

$$\mathrm{Var}(X+Y)=\mathrm{Var}(x)+\mathrm{Var}(Y) \qquad \text{for } X \text{ and } Y \text{ independent.}$$

A random variable of considerable importance in statistics is the sample mean, $\overline{X}_N$ of *N* samples from a probability distribution. The sample mean is defined as the sum of the samples divided by the number of samples and can be notationally expressed as

$$\overline{X}_N=\frac{1}{N}\sum_{i=1}^{N}X_i$$

Assuming that the X_i are independent and identically distributed (iid) random variables and using the properties of expectation and variance, we can derive the following results:

$$E(\overline{X}_N)=E(X),$$
$$\mathrm{Var}(\overline{X}_N)=\frac{\mathrm{Var}(X)}{N}.$$

The variance of the sample mean of *N* independent samples is a factor 1/*N* smaller than the variance of the random variable from which the samples are drawn. Hence, by selecting a sufficiently large *N*, the variance of the sample mean can be reduced to an arbitrarily small value.

Note that the relationship given above for the variance of $\overline{X}_N$ applies only if the samples are independent. If the samples are not independent, the calculation of Var($\overline{X}_N$) requires the consideration of the covariance between samples. For example, in the bank teller problem, the waiting times for successive customers will be correlated because there is a greater likelihood that the (*i*+1)*st* customer will be delayed if the *ith* customer waits than if the *ith* customer begins service immediately. Hence the variance of the average waiting time cannot be estimated by simply dividing the variance of the waiting time by the number of samples. Such a sequence of correlated samples is referred to as an auto-correlated series. In Section 2.13, we will address the problem of estimating the variance of a sample mean for an auto-correlated series.

2.6.1 Random Sum of Independent Random Variables

If X_1, X_2,…,X_K are independent and identically distributed random variables and *K* is a discrete random variable independent of X_i then for the sum

$$Y=\sum_{i=1}^{K}X_i$$

we have

$$E(Y) = E(X)E(K),$$

$$\mathrm{Var}(Y) = E(K)\,\mathrm{Var}(X) + \mathrm{Var}(K)E^2(X).$$

2.6.2 Change of Variables Formula

Given:

$$Y_j = g_j(W_1, W_2, ..., W_n),\ j = 1,2,...,n,$$

then the joint density function of the Y_j, $f_y(.)$, is

$$f_y(y_1, y_2, ..., y_n) = f_w(w_1, w_2, ..., w_n)\frac{1}{|J|},$$

where $f_w(.)$ is the joint density function for $W_1, W_2, ..., W_n$ and J is the *Jacobian* defined as the determinant of the matrix

$$\begin{pmatrix} \frac{\partial g_1}{\partial w_1} & \frac{\partial g_1}{\partial w_2} & \cdots\frac{\partial g_1}{\partial w_n} \\ \cdots & \cdots & \cdots\ \cdots \\ \frac{\partial g_n}{\partial w_1} & \frac{\partial g_n}{\partial w_2} & \cdots\frac{\partial g_n}{\partial w_n} \end{pmatrix}$$

and $|J|$ is the absolute value of J. This formula prescribes a procedure for making a transformation of random variables.

2.6.3 Rao-Blackwell Theorem (12)

Let X and Y denote random variables such that Y has mean μ and variance $\sigma_y^2 > 0$. Let $E(Y|x) = \phi(X)$. Then $E\{\phi(X)\} = \mu$ and $\sigma^2_{\phi(X)} \le \sigma^2_Y$.

This theorem states that if we are interested in the statistical properties of the random variable Y and can define a related random variable $\phi(X)$ which is the expected value of Y conditioned on X then we can estimate μ from the expected value of $\phi(X)$ and the variance of this estimate will be at least as small as the variance of a direct estimator. This theorem establishes the worth of using prior information in estimating sample means.[3]

2.7 Generating functions

A commonly referred to function of a random variable is the generating function. Several types of generating functions have been defined and we will discuss only the probability generating function and the moment generating function. The probability generating function for a discrete random variable is defined as

$$A(s) = \sum_i p(x_i)s^i$$

If $A(s)$ is known in a closed form then the probabilities, $p(x_i)$, can be obtained by taking the *ith* derivative with respect to s and setting s equal to zero. (If $A(s)$ is in a polynomial form, $p(x_i)$ can be seen by inspection.) The expectation of X can be obtained from $A(s)$ by taking the first

[3] This observation concerning the Rao-Blackwell Theorem as a basis for use of prior information as a variance reduction technique was pointed out to the author by James R. Wilson.

derivative with respect to *s* and setting *s*=1. Higher order moments can be obtained in a similar fashion but require combinations of derivatives. A function related to the probability generating function is the Z-transform.

The moment generating function, MGF, of a random variable X, is defined as

$$M(s) = E\left[e^{sX}\right].$$

The nth moment about the origin is obtained by taking the nth derivative with respect to *s*, that is,

$$\frac{d^n M(s)}{ds^n} = E(X^n e^{sX})$$

By setting *s*=0, we have $E[X^n]$. A function related to the moment generating function is the characteristic function.

In addition to obtaining moments of a random variable from generating functions, they are useful for obtaining moments of sums of independent random variables. For example, if W=X+Y and X and Y are independent then

$$E(e^{sW}) = E(e^{s(X+Y)}) = E(e^{sX})E(e^{sY})$$

Thus, the MGF of *W* is the product of the MGFs of *X* and *Y*. The moments for *W* can then be obtained from its MGF. Tables of probability generating functions and moment generating functions for the commonly used random variables are available [3, 9].

2.8 Law of large numbers and central limit theorem

There are two important theorems which characterize the behavior of X_N as the number of samples increases to infinity. The first theorem is the strong law of large numbers and states the intuitive result that as the sample size, *N*, increases, that with probability one, $\overline{X}_N$ approaches *E*(*X*). An associate result referred to as the weak law of large numbers is:

$$\lim_{N\to\infty} P\left\{\left|\overline{X}_N - E(X)\right| < \varepsilon\right\} = 1,$$

for any positive ε.

This simply says that for any positive value of ε, however small, the probability that the difference between $\overline{X}_N$ and *E*(*X*) exceeds ε approaches zero as *N* approaches infinity.

The second important theorem which characterizes the behavior of $\overline{X}_N$ is the Central Limit Theorem. This theorem states that under certain mild conditions, the distribution of the sum of *N* independent samples of *X* approaches the normal distribution as *N* approaches infinity, regardless of the distribution of *X*. Hence, sample means are approximately normally distributed for sufficiently large *N*. It is difficult to say what sample size is sufficient for assuring normality. However, relatively small sample sizes, like 10 to 15, are often sufficient. Feller [13] presents two theorems that deal with the quality of the normal approximation used in conjunction with the Central Limit Theorem[4]. These theorems are discussed in Bratley, Fox, and Schrage [7]. Many variations of the central limit theorem exist. In particular, one variation involves the conditions under which the central limit theorem is applicable for sequences of dependent random variables. These conditions are described in the next section.

[4] Feller's terms recurrent and persistent, although out of fashion, provide understandable descriptions of the concepts involved.

2.9 Asymptotic normality of recurrent events

If a recurrent event is persistent and the time between events has a finite mean μ and variance σ^2, then T_r and N_t are asymptotically normally distributed where T_r is the time until the rth event occurrence with $E(T_r) = r\mu$ and $\mathrm{Var}(T_r) = r\sigma^2$; and N_t is the number of event occurrences in t time units with $E(N_t) = t/\mu$ and $\mathrm{Var}(N_t) = t\ \sigma^2/\mu^3$. For example, if in a simulation the time between arrivals has a mean $\mu = 10$ and variance ($\sigma^2 = 4$, then the number of the arrivals in $t = 1000$ time units is approximately normal with a mean of 100 and a variance of 4.

The above statement is a central limit theorem for a sequence of dependent variables and can be used to check the reasonableness of input generators for a simulation model.

In addition to the above central limit theorem for recurrent events, there are central limit theorems that establish normality conditions for the sample mean or stationary stochastic processes. Moran [13] presents a theorem for moving average (MA) type processes and Dianada [12] for processes in which independence occurs after r lags or time periods.

2.10. Data collection and analysis

An essential function in simulation modeling is the collection and analysis of data. This function is required in both defining inputs for the model and in obtaining performance measures from experimentation with the model. In this section, we will review some of the important statistical concepts applicable to data collection and analysis.

2.10.1 Data Acquisition

Data acquisition is the process of obtaining data on a phenomenon of interest. There are a variety of methods by which the data can be acquired. In some cases, the data are available in existing documents, and the problem is that of locating and accessing the data. In other cases, data acquisition may involve the use of questionnaires, field surveys, and physical experimentation.

In aggregate models such as those of urban or economic systems, the required data can frequently be obtained from existing documentation. Common sources of data for these models include census reports, the Statistical Abstract of the Hungary publications, and other publications of governmental and international organizations. Sometimes such data are available in both report form and on computer tape.

In models of business systems, a valuable source of data is the accounting and engineering records of the company. These records are rarely sufficient to form the complete basis for estimating product demand, production cost, and other relevant data. However, they represent a starting point. Questionnaires and field surveys are also potential methods for obtaining data for industrial models.

Physical experimentation is commonly the most expensive and time consuming method for obtaining data. This process includes measurement, recording, and editing of the data. Considerable care must be taken in planning the experiment to assure that the experimental conditions are representative and that the data are recorded correctly. For a discussion of experimental design considerations in data collection, the reader is referred to the text by Bartee [6].

In some cases, there may be no existing data and the available budget or nature of the system may preclude experimentation. An example of such a case would be the use of simulation modeling to compare several proposed assembly line layouts.

A possible approach to data acquisition in such cases is the use of synthetic or predetermined data [5, 14]. In this method, estimates of activity durations are synthesized by using tables of standard data. Thus, this method permits activity times to be estimated before the process is actually in operation.

2.10.2 Descriptive Statistics

In both collecting data for defining inputs to the model and collecting data on system performance from the model, we encounter the problem of how to convert the raw data to a usable form. Hence, we are interested in treatments designed to summarize OT describe important features of a set of data. These treatments normally summarize the data at the expense of a loss of certain information contained within the data.

Grouping Data. One method for transforming data into a more manageable form is to group the data into classes or cells. The data is then summarized by tabulating the number of data points which fall within each class. This kind of table is called a frequency distribution table and normally gives a good overall picture of the data. An example of a frequency distribution table for data collected on customer waiting times is depicted below.

Waiting Time (Seconds)	Number of Customers
0-20	21
20-40	35
40-60	42
60-80	35
80-100	19
100-120	10
>120	10

The numbers in the right-hand column denote the number of customers falling into each class and are called the class frequencies. The numbers in the left-hand column define the range of values in each class and are referred to as the class limits. The difference between the upper class limit and lower class limit in each case is called the class width. Classes with an unbounded upper or lower class limit are referred to as open. If a class has bounded limits, it is denoted as closed. Frequently the first and/or last class in a frequency distribution will be open.

There are several variations of the class frequency tables which are useful for displaying grouped data. One variation is the cumulative frequency which is obtained by successively adding the frequencies in the frequency table. The cumulative frequency table for the customer waiting time data is depicted below.

Waiting Time Less Than (sec)	Cumulative Number of Customers
20	21
40	56
60	98
80	133
100	152
120	162
∞	172

The values in the right hand column represent the cumulative or total number of customers whose waiting time was less than the upper class limit specified in the left hand column. Another variation is obtained by converting the class frequency table (or cumulative table) into a corresponding frequency distribution by dividing each class frequency (cumulative frequency) by the total number of data points. Frequency distributions are particularly useful when comparing two or more distributions.

The frequency and cumulative distribution are sometimes presented graphically in order to enhance the interpretability of the data. The most common among graphical presentations is the histogram which displays the class frequencies as rectangles whose lengths are proportional to the class frequency. *Figure 2.7* depicts a histogram for the customer waiting time data.

The primary consideration in the construction of frequency distributions is the specification of the number of classes and the upper and lower class limits for each class. These choices depend upon the nature and ultimate use of the data; however, the following guidelines are offered.

1. Whenever possible, the class widths should be of equal length. Exceptions to this are the first and last classes which are frequently open.

2. Class intervals should not overlap and all data points should fall within a class. In other words, each data point should be assignable to one and only one class.

3. Normally at least five but no more than twenty classes are used.

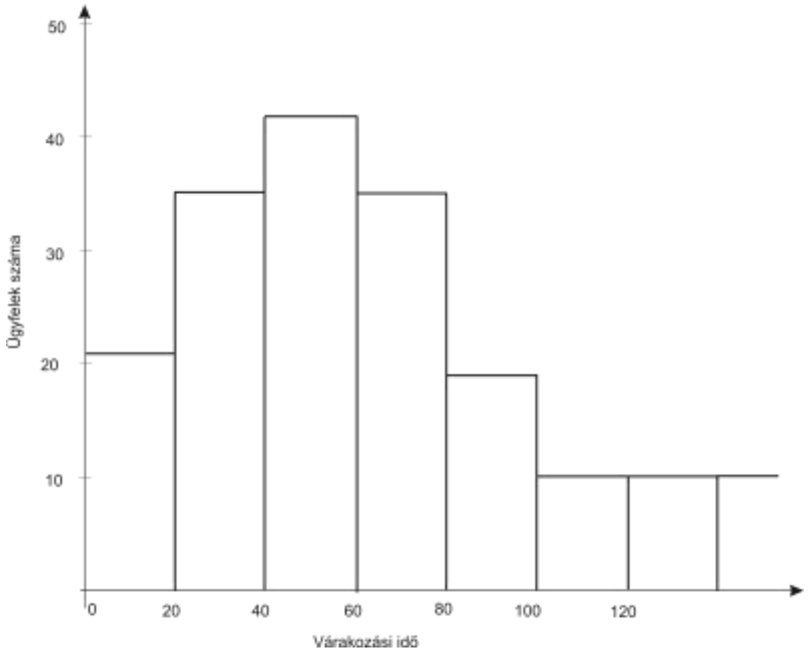


Figure 2.7: Histogram for customer waiting time data.

Parameter Estimation. If a set of data points consists of all possible observations of a random variable, we refer to it as a population; if it contains only part of these observations, we refer to it as a sample. Another method for summarizing a set of data is to view the data as a sample which is then used to estimate the parameters of the parent or underlying population. The population parameters of most frequent interest are the mean which provides a measure of centrality, and the variance which provides a measure of dispersion.

To illustrate, consider again the data on customer waiting times. This data can be viewed as a sample from the population which consists of all possible customer waiting times. We can use this sample data to estimate the mean customer waiting time and the variance of the customer waiting time for the population of all possible customers.

Different symbols are used to distinguish between population parameters and estimates of these parameters based upon a sample. The greek letters μ and σ^2 are often used to denote the population mean and variance, respectively. The corresponding estimates of these parameters based upon the sample record x_1, $x_2 ... , x_N$ are the average, denoted as $\bar{x}_N$, and the variance estimate, denoted as s_X^2. In order to further distinguish between descriptions of populations and descriptions of samples, the first are referred to as parameters and the second are referred to as statistics.

Before proceeding with this discussion of descriptive statistics, a clarifying point regarding the notation used to describe random samples and experimental values of a random variable or stochastic sequence is necessary. Before an experimental value is observed, it is a random variable denoted by X_i. After a value is observed, it is denoted by x_i. By the sample mean, $\overline{X}_N$, we refer to a random variable that is the sum of N random samples before they are observed divided by N. The average, $\bar{x}_N$, however, is the sum of N observed values x_i divided by N. In an analogous fashion, S_X^2 is the random variable describing an estimate of the sample variance before experimental values are observed and s_X^2 is the estimate of the variance of observed values. This notation conforms to our policy of using capital letters for random variables where possible and lower case letters for numerical quantities.

In constructing estimates of the population parameters from sample data, there are two distinct cases to consider. In the first case, we consider a sample record where we are concerned only with the value of each observation and not the times at which the observations were recorded. The data on customer waiting times is an example of such a record. Statistics derived from a time independent sample record are referred to as statistics based upon observations.

The second case to be considered is for variables which have values defined over time. For example, the number of busy tellers in a bank is a random variable that has a value which is defined over time. In this case, we require knowledge of both the values assumed by the random variable and the time periods for which each value persisted. Statistics derived from time dependent records are referred to as statistics on time-persistent variables.

The formulas for calculating $\bar{x}_N$ and s_X^2 for both statistics based upon observations and statistics on time-persistent variables are summarized in *Table 2.1.* For the time-persistent case the sample mean is designated by $\bar{x}_T$ where T is the total time interval observed. Sometimes the formulas for s~ are given in a slightly different form, however the form shown is the most convenient for computational purposes. Note that for statistics based upon observations, the

$$\sum_{i=1}^{N} x_i \,,\ \sum_{i=1}^{N} x_i^2 \,,$$

and the number of samples N, are sufficient to compute both $\bar{x}_N$ and s_X^2. Similarly, for statistics on time-persistent variables,

$$\int_0^T x\,dt \,,\ \int_0^T x^2\,dt$$

and T are required.

Another statistic which is commonly employed in summarizing a set of data is the coefficient of variation, $s_X / \bar{x}_N$. The coefficient of variation expresses the sample standard deviation relative to the sample mean. The use of the coefficient of variation is advantageous when comparing the variation between two or more sets of data.

Distribution Estimation. A related but more difficult problem is the use of the sample record to identify the distribution of the population. This problem frequently arises in modeling because of the need to characterize random elements of a system by particular distributions.

Although an understanding of the properties of the theoretical distributions described in Chapter 18 will aid the modeler in hypothesizing an appropriate distribution, it is frequently desired to test the hypothesis by applying one or more goodness-of-fit tests to the sample record. The chi-square and Kolmogorov-Smirnov are probably the best known tests, and descriptions and examples of these can be found in most statistics textbooks. A discussion of identifying a distribution function that fits a sample record and programs to perform the distribution fitting tasks is given in Chapter 4.

Table 2.1

Formulas for calculating the average and variance of a sample record

Statistic	Formula	
	Statistics based upon observation	Statistics for time persistent Variables
Sample Mean	$\bar{x}_N = \frac{\sum_{i=1}^{N} x_i}{N}$	$\bar{x}_T = \frac{\int_0^T x(t)\,dt}{T}$
Sample Variance	$s_X^2 = \frac{\sum_{i=1}^{N} x_i^2 - N\,\bar{x}_N^2}{N-1}$	$s_X^2 = \frac{\int_0^T x^2(t)\,dt}{T}$

2.11 Statistical inference

In simulation studies, inferences or predictions concerning the behavior of the system under study are to be made based on experimental results obtained from the simulation. Because a simulation model contains random elements, the outputs from the simulation are observed samples of random variables. As a consequence, any assertions which are made concerning the operation of the system based on simulation results should consider the inherent variability of the simulation outputs. This variability is summarized or taken into account by the use of confidence intervals or through hypothesis testing.

2.11.1 Confidence Intervals

In Section 2.10.2, we discussed methods for estimating the mean and variance parameters of a population based on a sample record. The estimates were calculated as a single number from the sample record and are referred to as point estimates. In general, an estimate will differ from the true but unknown parameter as the result of chance variations. The use of a point estimate has the disadvantage that it does not provide the decision maker with a measure of the accuracy of the estimate. A probability statement which specifies the likelihood that the parameter being estimated falls within prescribed bounds provides such a measure and is referred to as confidence interval or an interval estimate.

The parameter of primary interest in simulation analysis is the population mean. In the classical development of the confidence interval for the mean, it is assumed that the samples are independent and identically distributed (iid). Hence, by the Central Limit Theorem, the sample mean, $\bar{X}_N$ is approximately normally distributed for sufficiently large *N*. As stated previously, the assumption of independence is not a necessary condition for the application of the Central Limit Theorem.

If we assume that $\bar{X}_N$ is normally distributed, then the statistic

$$Z = \frac{\overline{X}_N - \mu}{\sigma_{\overline{X}}}$$

$$P(-Z_{\alpha/2} < Z < Z_{\alpha/2}) = 1 - \alpha$$

(2.1) $$\overline{X}_N - Z_{\alpha/2}\sigma_{\overline{X}} < \mu < \overline{X}_N + Z_{\alpha/2}\sigma_{\overline{X}}$$

that is, a proportion, 1–*a*, of confidence intervals based on *N* samples of *x* should contain (cover) the mean *m*. This proportion is called the coverage for the confidence interval.

The above formula assumes knowledge of the standard deviation of the mean, $\sigma_{\overline{X}}$, which is usually unknown. If we use the sample standard deviation of the mean, $S_{\overline{X}}$, to estimate $\sigma_{\overline{X}}$ we can develop a similar relationship by noting that the statistic

$$t = \frac{\overline{X}_N - \mu}{S_{\overline{X}}}$$

is a random variable having a student *t*-distribution with *N*–1 degrees of freedom. Hence, a 1–*α* a confidence interval for μ using the estimate $S_{\overline{X}}$ is given by

(2.2) $$\overline{X}_N - t_{\alpha/2,N-1}S_{\overline{X}} < \mu < \overline{X}_N + t_{\alpha/2,N-1}S_{\overline{X}}$$

where $t_{\alpha/2,N-1}$is a critical value of the *t*-statistic with (*N*–1) degrees of freedom.

If the samples X_i are iid, the confidence intervals given by (2.1) and (2.2) are modified by the substitutions

$$\sigma_{\overline{X}} = \frac{\sigma_X}{\sqrt{N}} \text{ and } S_{\overline{X}} = \frac{S_X}{\sqrt{N}}$$

respectively. This substitution provides an expression for the confidence interval based on samples. However, this simple relationship between the variance of the samples and the variance of the mean of the samples is valid only if the samples are independent.

Methods for defining $S_{\overline{X}}$ for use in Expression (2.2) in the case of auto-correlated samples are described in Chapter 19. The most direct approach is to organize the experiment to obtain independent observations which can be accomplished through replicating the simulation or organizing the data into batches.

2.11.2 Tolerance Intervals

A tolerance interval provides a range for an observation or for an average of a set of observations. Remember that the observation and its average are random variables. Because we are setting an interval on a random variable, it is necessary to specify the fraction of samples of the random variable that is desired to be within the tolerance interval and then to further specify the confidence with which we desire the fraction of samples to be contained within the interval. Hence, we specify that, with (1–*δ*) probability, we desire a range such that (1–*ε*) fraction of observations of $\overline{X}_N$ each based on a sample size of *N*, will fall in the tolerance interval. Wilson (19) developed the following formula for such a range:

$$\overline{X}_N \pm Z_{\varepsilon/2} Q \frac{S_X}{\sqrt{N}}$$

where $Z_{\varepsilon/2}$ is a critical value of the normal distribution corresponding to a (1–ε) confidence, and Q is given by

$$Q = \frac{2N+1}{2N}\sqrt{\frac{N-1}{\chi^2_{\delta,N-1}}}$$

where $\chi^2_{\delta,N-1}$ is a critical value from the chi-square distribution with (1–δ) confidence and (N–1) degrees of freedom.

In an experiment consisting of N runs, Wilson also derived the following tolerance interval formula that the probability is at least (1–δ)(1–ε) that a single additional observation X_N+1 will fall in the interval

$$\overline{X}_N \pm Z_{\varepsilon/2} Q S_X$$

2.12 Hypothesis testing

In some applications of simulation, the objective is to decide if a statement concerning a parameter is true or false. For example, we might want to decide whether a change in a dispatching rule for a job shop reduces the average late time for the jobs processed. Due to the experimental nature of simulation, we must account for the chance variation in the estimates of the parameters being compared. This is done using hypothesis testing.

The general procedure of hypothesis testing calls for defining a null hypothesis (denoted H_0) and an alternate hypothesis (denoted H_1). The null hypothesis is usually set up with the objective of determining whether or not it can be rejected. For example, if we wish to establish that job loading rule A reduces average late time relative to job loading rule B, we would define the null and alternate hypotheses as

H_0: average waiting time for rule A equals average waiting time for rule B

H_1: average waiting time for rule A is less than the average waiting time for rule B

We would then use the experimental results from simulations with rules A and B to attempt to reject H_0 in favor of H_1.

Testing the null hypothesis against the alternate hypothesis involves selecting a decision rule based on the sample data which leads to the acceptance or rejection of the null hypothesis. Acceptance of the null hypothesis does not infer that the null hypothesis is true, but that there is insufficient evidence based on the sample data to reject the hypothesis.

There are two types of errors that can be made in applying the decision criterion. The *Type I* error is to reject the null hypothesis when the hypothesis is true. The *Type II* error is to accept the null hypothesis when it is false. A decision rule can be judged by the probabilities associated with *Type I* and *Type II* errors. These probabilities are typically denoted as α and β probabilities, respectively. The probability α of a *Type I* error is referred to as the **level of significance** of the test.

The decision criterion is established by constructing a **test statistic** which has a known distribution. The test statistic is calculated from the sample data and is compared using a rejection rule. If the test statistic falls within the critical region, then the null hypothesis is rejected.

The test statistic and rejection rule for hypothesis tests concerning means are summarized in *Table 2.2*. Tests 1 and 2 are for a mean being equal to a given value μ_0. Tests 3 and 4 concern the comparison of two means. The equations for the test statistics are expressed in terms of $\sigma_{\overline{X}}$ and S_X since assumptions of independence cannot be prescribed.

2.13 Statistical problems related to simulation

Decision analysis based on the results of a simulation model normally requires an estimate of the average simulation response and an estimate of its variance. Both of these estimators are affected by experimental conditions. The experimental conditions which the modeler must establish include the initial or starting states for the simulation, the time at which statistics collection is to begin, and the run length and number of replications. In this section, we will introduce some of the considerations and problems associated with establishing these conditions. In Chapter 7, we discuss these problems in detail.

2.13.1 Initial Conditions

Implicit in every simulation model is an initial condition or starting state for the simulation. The simplest and probably most commonly used initial state is "*empty* and *idle*" in which the simulation begins with no entities in the system and all servers idle. The appropriateness of this starting condition depends on the nature of the system being modeled and whether we are interested in the **transient** or **steady-state** behavior[5] of the system.

When the purpose of our analysis is to study the steady-state behavior of a system, we can frequently improve our estimate of the mean by beginning the simulation in a state other than empty and idle. The starting condition can be established by estimating an initial state which is representative of the long term behavior of the system, perhaps by observing the plotted output from a pilot simulation run. For a transient analysis, the starting condition should reflect the initial status of the system.

2.13.2 Data Truncation

A method which is frequently used to reduce any bias in estimating the steady-state mean resulting from the initial conditions is to delay the collection of statistics until after a "*warm up*" period. This is normally done by specifying a truncation point before which data values are not included in the statistical estimates. The intent is to reduce the initial condition bias in the estimates by eliminating values recorded during the transient period of the simulation. However, by discarding a portion of the data, we are not using observations and, hence, may be increasing the estimated variance of the mean. Thus, by truncation we improve the quality of the estimate of the mean at the possible expense of increased variability in the simulation outputs.

The most common method for determining the truncation point is to examine a plot of the response from a pilot simulation run. The truncation point is selected as the time at which the response "appears" to have reached steady state. There are also methods which attempt to formalize this procedure in the form of a rule which can be incorporated into the simulation program to automatically determine the truncation point during the execution of the simulation. These rules are discussed in Chapter 7. A theorem is presented in the next section that addresses the question of returns to a state or the crossing of an expected value.

[5] Steady-state behavior does not denote a lack of variability in the simulation response, but specifies that the probability mechanism describing this variability is unchanging and is no longer affected by the starting condition.

2.13.3 The First Arc Sine Law

Consider a binomial random variable Y_n with $P(Y_n=1)=1/2$ and $P(Y_n=-1)=-1/2$ for $n=1,2, ...,N$. Consider the sequence of partial sums

$$Z_n = \sum_{i=1}^{n} Y_i$$

for all times up to time N. For a fixed a ($0<\alpha<1$), we focus on the experimental outcome in which $Z_n>0$ for at most N_α time units, that is, we look for experiments in which the sequence $\{Z_n:1\leq n:N\}$ up to time N spends at most α percent of the time above the axis. As $N \to \infty$, the probability of observing such a result tends to

$$\frac{2}{\pi}\sin^{-1}(\sqrt{\alpha}$$

For example, the probability that the fraction of time is less than α =0.976 is 0.90. Thus, with probability 0.20, the fraction of time spent on one side of zero or the other is 0.976.

Another result is that, in a time period of length $2N$, the probability that the number of partial sums $\{Z_\mathrm{n}\}$ equal to zero being at most a $\alpha\sqrt{2N}$ tends to .

$$\sqrt{\frac{2}{\pi}}\int_0^{\alpha} \mathrm{e}^{-q^2/2}\,dq, \ \text{ ahol } \ N \to \infty$$

For example, if we drew 10,000 samples of Y_n then there is a probability of 0.50 that there will be fewer than 68 times when $\sum Y_n$ =0. A related result also given by Feller specifies that the number of changes of sign (crossings) in the sequence of partial sums in N time units increases as the $\sqrt{N}$, that is, in $100N$ time units we should only expect 10 times as many crossings of 0 as in N time units. These theorems illustrate the conceptual difficulties and non-intuitive behavior associated with even simple stochastic processes. These results indicate potential difficulties associated with the use of returns to a state or the crossing of a state in statistical analysis.

2.13.4 Run Length and Number of Replications

An important experimental design decision which the analyst must make is the tradeoff between run length and number of replications of the simulation. The use of a few long runs as opposed to many short runs generally produces a better estimate of the steady state mean because the initial bias is introduced fewer times and less data is truncated. However, the reduced number of samples corresponding to fewer replications may increase our estimate of the variance of the mean. The use of many short runs, on the other hand, may introduce a bias due to the starting conditions. The larger the initial bias, the more important it is to use longer runs to reduce the effects of the starting conditions.

There are several alternate methods for specifying the duration of a simulation. Perhaps the most common method is to specify a time at which the simulation is to end. A disadvantage of this method is that the number of samples collected is a random variable and may differ in each replication. A method which allows us to control the sample size is to specify the number of entities which are to be entered into the model. In this case the simulation executes until the prescribed number of entities which are entered into the model are completely processed through the system. Thus, the simulation stops in the empty and idle state. A similar but different method is to specify the number of entities which are processed through the system. Note that in this case the system is not necessarily empty and idle at the time it is stopped. When using this approach, it is necessary to ensure that the entities remaining to be processed are representative. An example where this stopping method may be inappropriate is when a shortest processing time dispatching rule is employed and, hence, jobs with long processing times may be the ones still remaining in the queues.

Another approach for controlling the duration of a simulation is the use of automatic stopping rules. These methods automatically monitor the simulation results at selected intervals during the execution of the simulation. The simulation is stopped when the estimate of the variance of the mean is within a prescribed tolerance. The use of automatic stopping rules is discussed in more detail in Chapter 7.

If we are estimating the variance of an output variable *X* by replication and if we assume that *X* is normally distributed (which if *X* is a mean value is a good assumption) then the number of independent replications of the simulation required to attain a specified confidence interval for $\overline{X}$ is given by

$$N = \left(\frac{t_{\alpha/2,N-1} S_X}{g} \right)^2$$

where

$t_{\alpha/2,N\text{-}1}$is a value from the table of critical values of the *t*-statistic with *N*–1 degrees of freedom

g is the half-width of the desired confidence interval

Unfortunately, the use of this formula for *N* requires knowledge of the *t*-statistic with *N*–1 degrees of freedom and S_X. Typically, we must assume a value for *N*, make the *N* replications of the simulation, obtain values of *t* and s_X based on these runs, and then use the above formula with these values inserted to test the sufficiency of our initial assumption or to determine the number of additional replications which are required.

2.14. Chapter summary

This chapter has provided the probability and statistics background required for simulation analysis. Detailed developments have not been presented as the intent was to cover a wide range of simulation related topics. The material introduces sufficient simulation subject matter to permit the understanding of simulation modeling concepts and the experimental nature of simulation analysis. It also provides a basis for understanding the detailed aspects associated with the statistical analysis of simulation results.

CHAPTER 3

Getting Started

While you may not realize it quite yet, you now have the power to transform your business. Whenever you and others in your organization are wondering "what if…?," you can look into the future to find the answer.

With Arena, you can:

Model your processes to define, document, and communicate.

Simulate the future performance of your system to understand complex relationships and identify opportunities for improvement.

Visualize your operations with dynamic animation graphics.

Analyze how your system will perform in its "as-is" configuration and under a myriad of possible "to-be" alternatives so that you can confidently choose the best way to run your business.

3.1. Our task: Analyze a home mortgage application process

In this chapter, we will examine a simple mortgage application process to illustrate how you can model, simulate, visualize, and analyze with Arena. To begin, we'll look at the process of receiving and reviewing a home mortgage application. We will build the flowchart shown below, introducing you to the process of modeling and simulating with Arena.

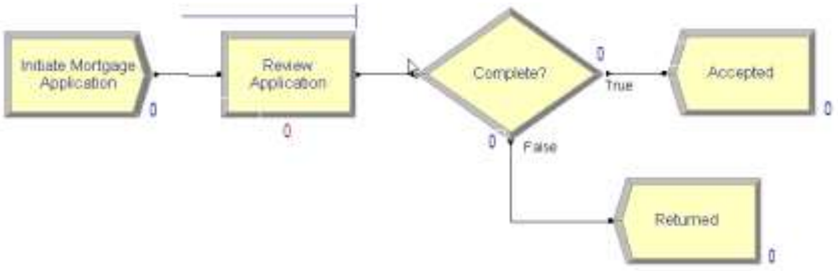


Figure 3.1: The Flowchart

3.2. The Arena modeling environment

If Arena is not already running, start it from the Windows Start menu and navigate to *Programs* > *Rockwell Software* > *Arena*. The Arena modeling environment will open with a new model window, as shown *Figure 3.2*.

To model your process in Arena, you'll work in three main regions of the application window. The *Project Bar* (usually is docked at the left of the Arena application window, but can tear off or dock at another position like any toolbar.) hosts panels with the primary types of objects that you will work with:

Basic Process, **Advanced Process**, and **Advanced Transfer panels**: Contain the modeling shapes, called *modules*, that you'll use to define your process.

Reports panel: Contains the reports that are available for displaying results of simulation runs.

Navigate panel: Allows you to display different views of your model, including navigating through hierarchical submodels.

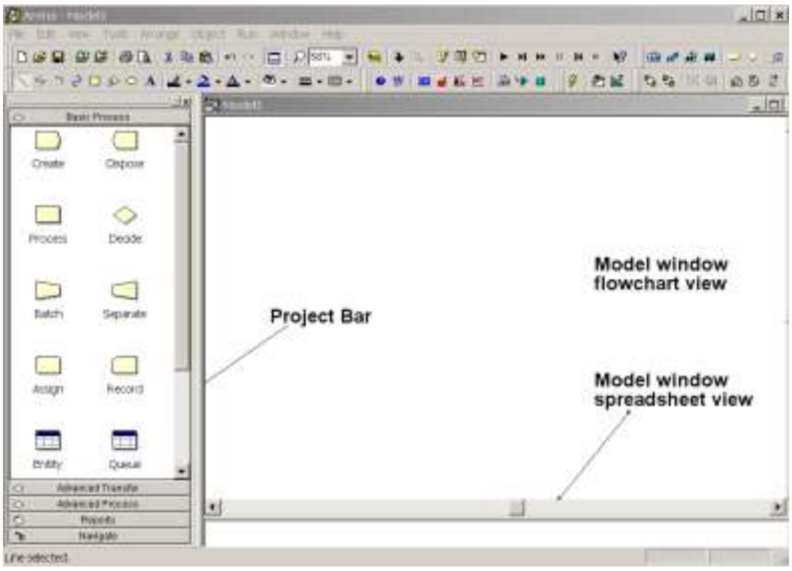


Figure 3.2: The Arena Modeling Environment

In the model window, there are two main regions. The *flowchart view* will contain all of your model graphics, including the process flowchart, animation, and other drawing elements. The lower, *spreadsheet view* displays model data, such as times, costs, and other parameters.

As we model the mortgage application process, we'll work in all three of these regions of Arena.

3.3. Map your process in a flowchart

Let's start by examining what we're going to do: **Build a flowchart**. The word itself *flowchart* suggests two of the main concepts behind modeling and simulation. We'll be building a *chart* also referred to as a *process map* or a *model* that describes a *flow*.

This raises a key question in process modeling: What exactly is it that will *flow* through the chart?

We're modeling the process of reviewing mortgage applications. These mortgage applications are the items, referred to as *entities*, that will move through the process steps in our model. They are the data, whether on paper or in electronic form, that are associated with our client's request for a mortgage. As we build the flowchart, it's helpful to think of the process from the perspective of the *entity* (the mortgage application), asking questions like:

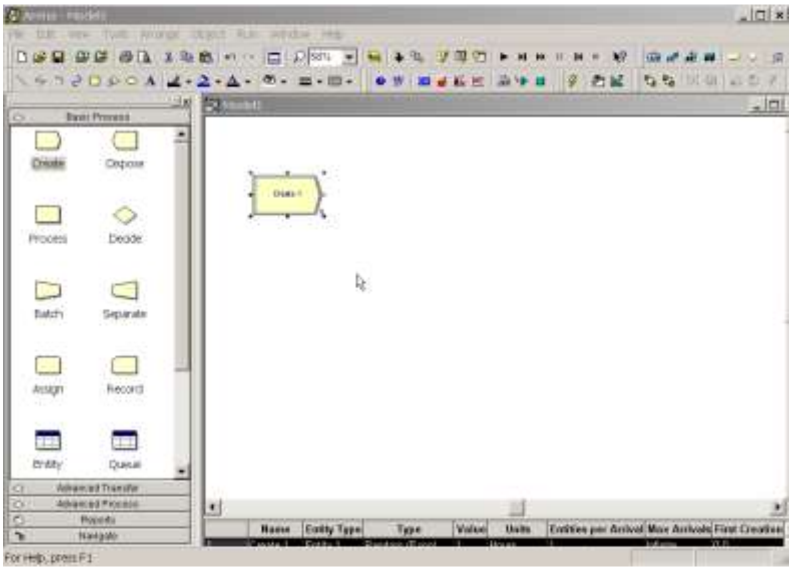
> Where do the mortgage applications enter the process?
>
> What happens to them at each step?
>
> What resources are needed to complete work?

First, we'll draw the flowchart representing the mortgage application process. Refer to the Mortgage Application Process Flowchart (shown previously) so you'll know what we'll be creating.

3.3.1 Create the mortgage application entities

We'll start the flowchart using a **Create** module**,** from the Basic Process panel. Every process flow starts with a **Create** module. When you simulate the flowchart, individual entities will be created according to timing information you supply in the **Create** module properties. After it's created, each entity moves from the **Create** module to the next shape in the process flow. This is the starting point for the flow of entities through the model.

1. Drag the **Create** module from the Basic Process panel into the model window.



	Name	Entity Type	Type	Value	Units	Entities per Arrival	Max Arrivals	First Creation
1	Create 1	Entity 1	Random (Expo)	1	Hours	1	Infinite	0.0

For Help, press F1

Figure 3.3: Using a **Create** module from the Basic Process Panel

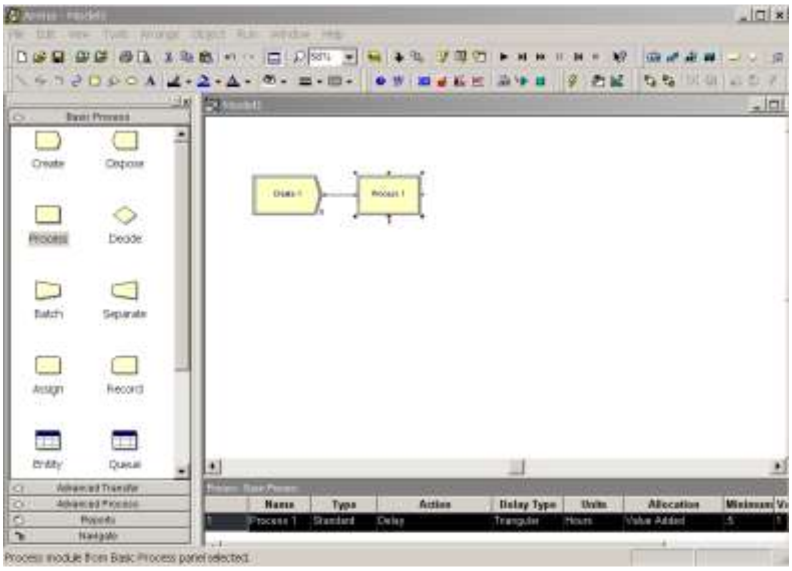
A default name, Create 1, is given to the module when it's placed. We'll return later to provide a more meaningful description as well as some data to support the simulation.

3.3.2 Process the applications

Next in our flowchart is a **Process** module, from the Basic Process panel, representing the Review Application step.

1. So that Arena will automatically connect the **Process** to the **Create** module, be sure that the **Create** module is selected.

2. Drag a **Process** module from the Basic Process panel into the model window, placing it to the right of the **Create**. Arena will automatically connect the two modules.



Process - Basic Process

	Name	Type	Action	Delay Type	Units	Allocation	Minimum	V
1	Process 1	Standard	Delay	Triangular	Hours	Value Added	.5	1

Process module from Basic Process panel selected.

Figure 3.4: Connecting a **Process** module from the Basic Process Panel

If your **Create** and **Process** modules weren't connected automatically when you placed the Process, check the *Object>Auto-Connect* menu to verify that it's checked. If it's not, select it to turn on this option.

As with the **Create**, the **Process** module has a default name that we'll replace later.

Note: If no connection appears between **Create** and **Process**, click the *Object >Connect* menu item or the **Connect** toolbar button to draw a connection. Your cursor will change to a cross hair. Start the connection by clicking the exit point (▶) of the **Create** module, then click the entry point (■) of the **Process** module to complete the connection.

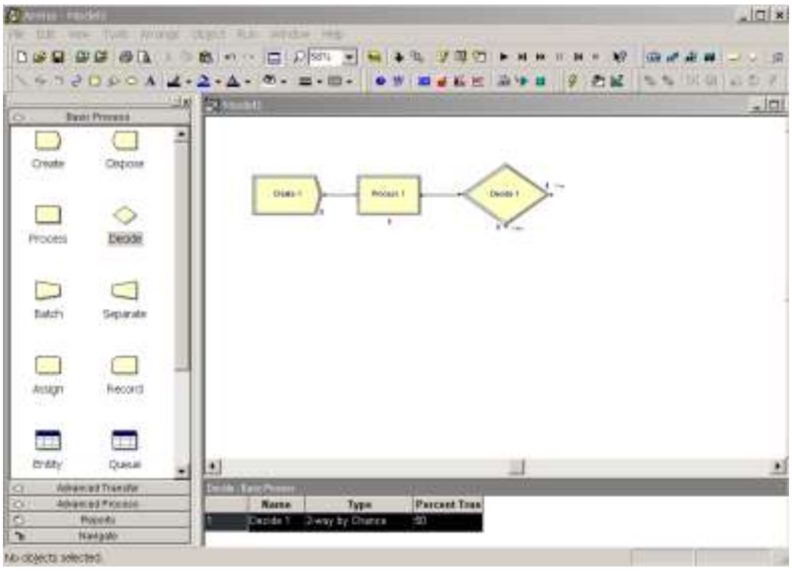
How do I use Snap and Grid?

If your flowchart shapes aren't lining up properly, you can use Arena's snap and grid features to straighten them out. First, check the *Snap* option on the *View* menu so that newly placed shapes will be positioned at regular snap points. To realign the shapes you've already placed, select the main module shapes (the yellow boxes) by holding the *Ctrl* key and clicking on each shape. Then, click the *Arrange>Snap to Grid* menu option to adjust their positions to align with grid points.

You can display the grid by checking the *Grid* option on the *View* menu. Both snap and grid are turned off by clicking on the menu option again, turning off the check box.

3.3.3 Decide whether applications are complete

After the Process, we have a **Decide** module, from the Basic Process panel, which determines whether the mortgage application is complete.



	Name	Type	Percent True
1	Decide 1	2-way by Chance	50

Figure 3.5: Connecting a **Decide** module from the Basic Process Panel

1. If you're using the Auto-Connect feature (i.e., it's checked on the *Object>Auto-Connect* menu), be sure that the **Process** module is selected so that the **Decide** will be connected to it.

2. Drag a **Decide** module to the right of the **Process** module.

If the mortgage application has a complete set of information, it will leave the **Decide** module from the right side of the diamond shape, representing the True condition. Incomplete applications (False result to the Decide test) will leave via the bottom connection.

3.3.4 Dispose the applications to terminate the process

Next we'll place the **Dispose** module, from the Basic Process panel, representing accepted applications, connecting to the *True* (right) output from the **Decide** shape. Then, we'll complete the flowchart with another **Dispose** for returned applications.

1. Select the Decide shape so that our first Dispose will be connected automatically.

2. Drag a **Dispose** module to the right of the **Decide** module. Arena will connect it to the primary (*True*) exit point of the **Decide** module. (We won't include a graphic display since you're now familiar with the drag-and-drop sequence.)

3. To add the second **Dispose** module, once again select the **Decide** module, so that Arena will automatically connect its False exit point to the new **Dispose** module, and drag another **Dispose** module below and to the right of the **Decide** module.

4. Drag and drop another **Dispose** module, placing it below and to the right of the **Decide** shape, completing the process flowchart.

What is a module?

In Arena, *modules* are the flowchart and data objects that define the process to be simulated. All information required to simulate a process is stored in modules.

For now, we're working with flowchart modules those that are placed in the model window to describe the process. In the Basic Process panel, these are the first eight shapes:

Create: The start of process flow. Entities enter the simulation here.

Dispose: The end of process flow. Entities are removed from the simulation here.

Process: An activity, usually performed by one or more resources and requiring some time to complete.

Decide: A branch in process flow. Only one branch is taken.

Batch: Collect a number of entities before they can continue processing.

Separate: Duplicate entities for concurrent or parallel processing, or separating a previously established batch of entities.

Assign: Change the value of some parameter (during the simulation), such as the entity's type or a model variable.

Record: Collect a statistic, such as an entity count or cycle time.

Simulation settings are defined in the *Run>Setup>Replication Parameters* dialog. There is also a set of *data modules* for defining the characteristics of various process elements, such as resources and queues.

Entity flow always begins with a **Create** module and terminates with a **Dispose** module. You may have as many of each of these modules as you need to generate entities into the model and to remove them when their processing is complete.

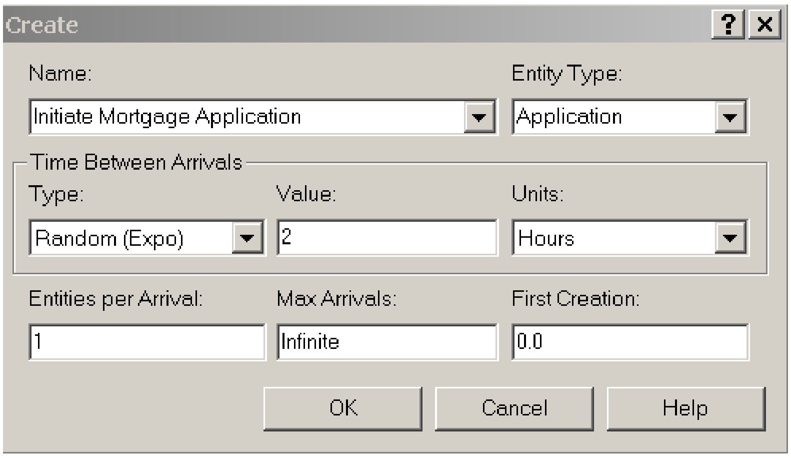
3.4. Define model data

Now that we've drawn the basic flowchart for our mortgage application process, let's define the data associated with the modules, including the name of the module and information that will be used when we simulate the process.

3.4.1 Initiate mortgage application (Create module)

First, let's visit the **Create** module, which will be named *Initiate Mortgage Application*. Its data will include the type of entity to be created in our case, a mortgage *Application*. We also need to define how often mortgage applications are initiated. We'll use an average of *2 hours* between applicants as a starting point for our model, and we'll make this a random activity to represent the natural variation in the timing of mortgage applications being submitted.

1. Double-click the **Create** module to open its property dialog (*Display 3.1*).
2. In the Name field, type *Initiate Mortgage Application*.
3. For the Entity Type, name our entities by typing *Application*.
4. Type of the Time Between Arrivals section is *Random*(*Expo*)
5. Write *2* in the Value field of the Time Between Arrivals section.
6. Click *OK* to close the dialog.


Create

Name: Initiate Mortgage Application

Entity Type: Application

Time Between Arrivals

Type: Random (Expo)

Value: 2

Units: Hours

Entities per Arrival: 1

Max Arrivals: Infinite

First Creation: 0.0

Display 3.1: The **Create** module Dialog Box

For now, we'll leave the default values for the other **Create** module properties. Feel free to explore their purposes through online help or the *Entity Arrivals* models in the SMARTs library.

What are entities?

Entities are the items customers, documents, parts that are being served, produced, or otherwise acted on by your process. In business processes, they often are documents or electronic records (checks, contracts, applications, purchase orders). In service systems, entities usually are people (the customers being served in a restaurant, hospital, airport, etc.). Manufacturing models typically have some kind of part running through the process, whether it's raw material, a subcomponent, or finished product. Other models might have different types of entities, such as data packets in network analysis or letters and boxes in package-handling facilities.

You may have different types of entities in the same model. For example, customers moving through a check-in counter at an airport might be separated into regular, first-class, and priority entity types. In some cases, entity types might be of an altogether different form rather than classifications of some basic type. For instance, in a pharmacy, prescriptions would be modeled as entities, running through the process of being filled. At the same time, customers might be competing for the pharmacist's attention with medical inquiries; they would also be modeled as entities.

3.4.2 Review application (Process module)

Remember that as we create the flowchart, we're looking at the process from the perspective of the entity. The **Create** module is a starting point for an entity's flow through the system being modeled. Next, in our case, the application will be reviewed for completeness by a *Mortgage Review Clerk*. Because this will take some amount of time, holding the entity at this point in the flowchart for a *delay* and requiring a *resource* to perform the activity, we use a **Process** module. We'll call this process *Review Application*.

For the time delay, we also want to capture the natural variability that exists in most processes. Very often, for work done by people or equipment, a triangular distribution provides a good approximation. You specify the *minimum* time in which the work could be done, the *most likely value* for the time delay, and the *maximum* duration of the process.

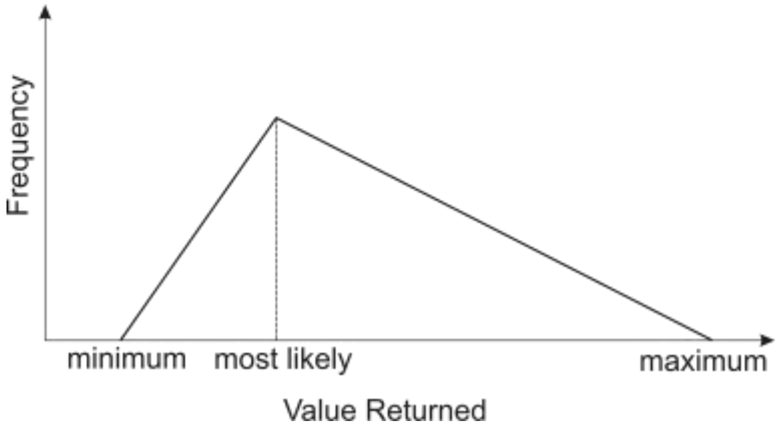
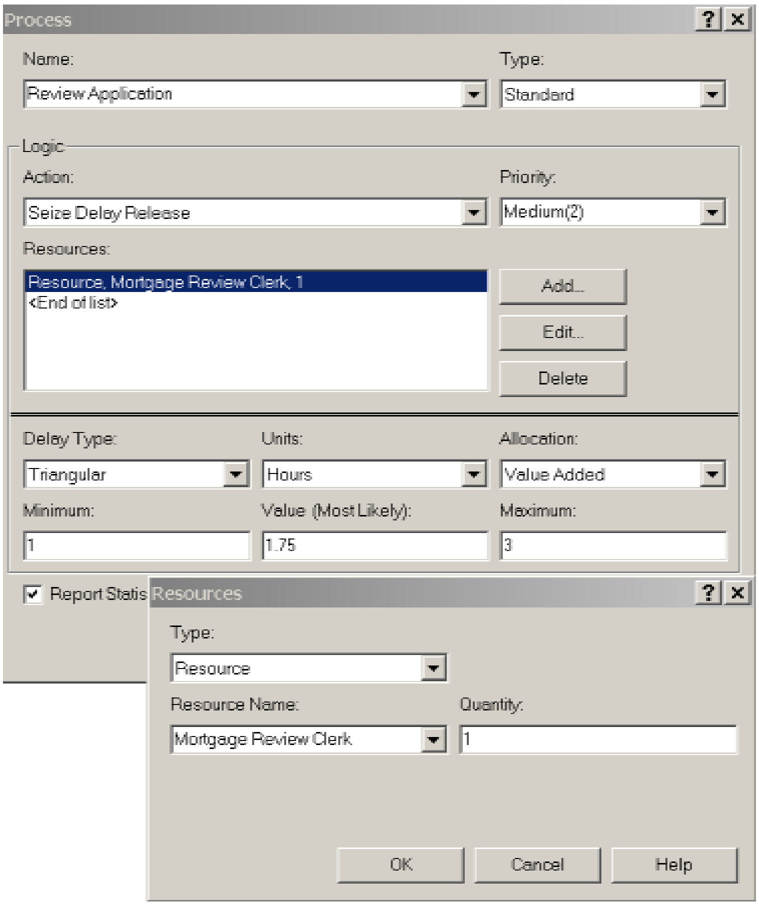


Figure 3.6: The Triangular Distribution

During the simulation run, each time an entity enters the process, Arena will calculate a sample from the distribution information you've provided in our case, a triangular distribution. Over the course of a long simulation run where thousands of individual samples are taken, the times will follow the profile illustrated *Figure 3.6*. Appendix A describes the distributions available in Arena.



Name: Review Application

Type: Standard

Action: Seize Delay Release

Priority: Medium(2)

Resources:
Resource, Mortgage Review Clerk, 1
<End of list>

Delay Type: Triangular

Units: Hours

Allocation: Value Added

Minimum: 1

Value (Most Likely): 1.75

Maximum: 3

☑ Report Statis

Type: Resource

Resource Name: Mortgage Review Clerk

Quantity: 1

Display 3.2: The **Process** module Dialog Box

For our Review Application process, we'll use a minimum time of 1 *hour*, most likely value of 1.75 *hours*, and a maximum of 3 *hours*. We will assign a *resource,* the *Mortgage Review Clerk*, to perform this process.

1. Double-click the **Process** module to open its property dialog (*Display 3.2*).

2. In the Name field, type *Review Application*.

3. To define a resource to perform this process, pull down the Action list and select *Seize Delay Release*.

Arriving entities will wait their turn for the resource to be available. When its turn comes, the entity will *seize* the resource, *delay* for the process time, and then *release* the resource to do other work.

4. A list of resources will appear in the center of the dialog. To add a resource for this process, click *Add.*

If more than one resource is required for a process to be performed, add as many as are necessary in the process dialog's Resources list. An entity won't commence its process delay until all listed resources are available.

5. In the Resource dialog, type *Mortgage Review Clerk* in the Resource Name field.

6. Click *OK* to close the **Resource** dialog.

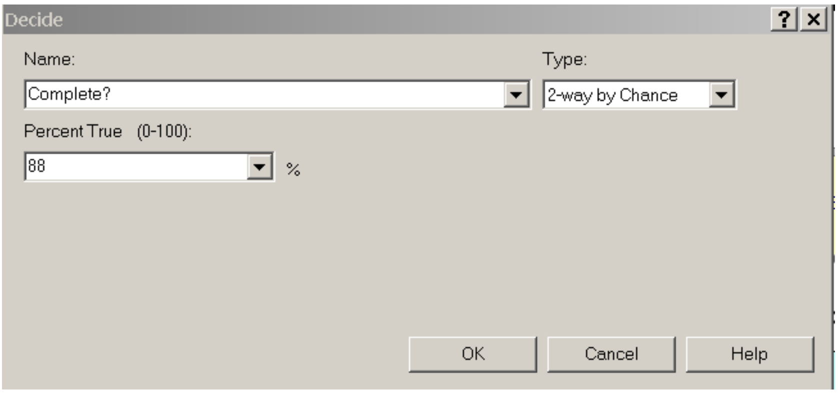
7. Define the process delay parameters in the Minimum, Most Likely Value, and Maximum fields as 1, 1.75, and 3. (Note that the default delay type is *Triangular* and the default time units are in *hours*.)

8. Click *OK* to close the dialog.

For now, we'll leave the default values for the other **Process** module properties. Feel free to explore their purposes through online help or the "Modeling Concepts" and "Resources" models in the SMARTS library.

3.4.3 Complete? (Decide module)

After the mortgage application has been reviewed, we determine whether to accept or return the application. In Arena, whenever an entity selects among branches in the process logic, taking just one of the alternatives, a **Decide** module is used.



Decide

Name: Complete?

Type: 2-way by Chance

Percent True (0-100): 88 %

Display 3.3: The **Decide** module Dialog Box

For the mortgage application process, we'll use a simple probability to determine the outcome of the decision, with 88% of applications accepted as complete.

1. Double-click the **Decide** module to open its property dialog (*Display 3.3*).

2. In the Name field, type *Complete?*.

When you use a 2-way **Decide** module, the entity that enters the module leaves via one of the two exit points. If you want to make copies of an entity to model parallel processes, use a **Separate** module.

3. For the Percent True field, type 88 to define the percent of entities that will be treated with a “True” decision (i.e., will depart through the exit point at the right of the **Decide** module).

4. Click *OK* to close the dialog.

3.4.4 Accepted, Returned (Dispose module)

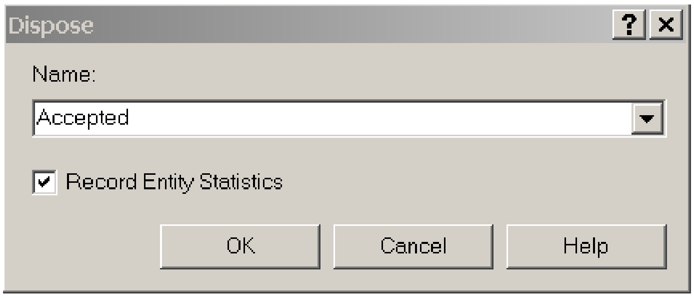
In our simple process for reviewing mortgage applications, all the work that we’re interested in is done. Now, we’ll remove the mortgage applications from the model, terminating the process with a **Dispose** module. Because there are two possible outcomes of the mortgage application process applications can be accepted or returned we’re using two **Dispose** modules that will count the number of applications under each outcome.

1. Double-click the first **Dispose** module (connected to the *True* condition branch of the **Decide** module) to open its property dialog (*Display 3.4*), and in the Name field, type *Accepted*.

Click *OK* to close the dialog.

2. Double-click the other **Dispose** module to open its property dialog. In the Name field, type *Returned*.

3. Click *OK* to close the dialog.



Name:

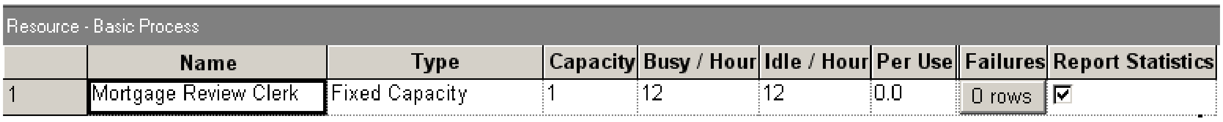
Accepted

☑ Record Entity Statistics

Display 3.4: The *Accepted* **Dispose** module Dialog Box

3.4.5 Mortgage review clerk (Resource module)

Along with our flowchart, we also can define parameters associated with other elements of our model, such as resources, entities, queues, etc. For the mortgage process, we’ll simply define the cost rate for the Mortgage Review Clerk so that our simulation results will report the cost associated with performing this process. The clerk’s costs are fixed at €12 per hour.



Resource - Basic Process

	Name	Type	Capacity	Busy / Hour	Idle / Hour	Per Use	Failures	Report Statistics
1	Mortgage Review Clerk	Fixed Capacity	1	12	12	0.0	0 rows	☑

Display 3.5: The **Resource** data module Dialog Box

To provide these parameters to the Arena model, you’ll enter them in the Resources spreadsheet (*Display 3.5*).

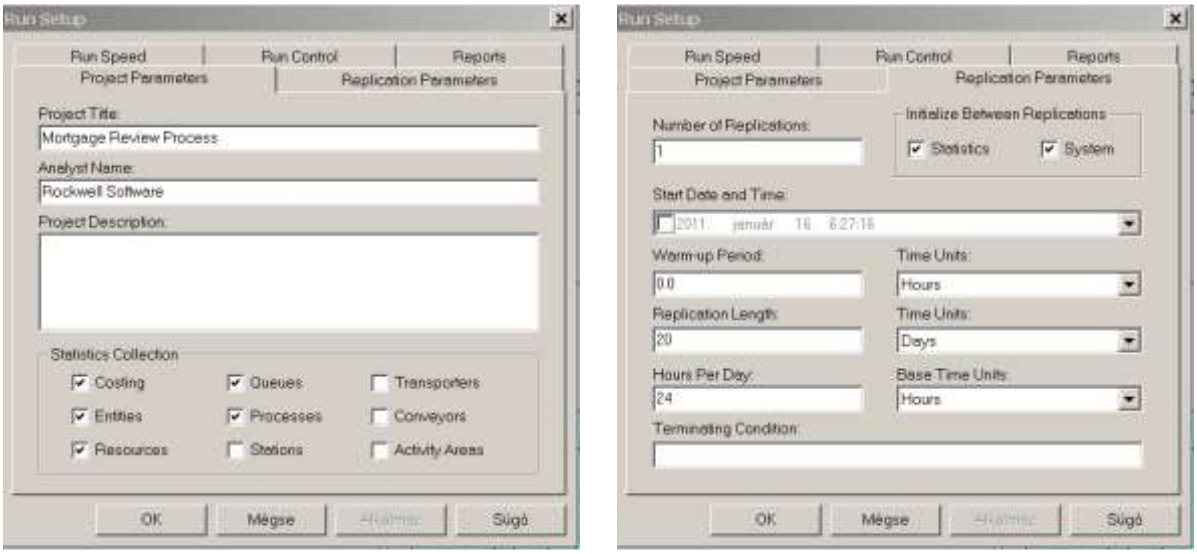
1. In the Basic Process panel, click the Resource icon to display the Resources spreadsheet.

2. Because we defined the *Mortgage Review Clerk* as the resource in the Review Application process, Arena has automatically added a resource with this name in the Resources spreadsheet. Click in the Busy/Hour cell and define the cost rate when the clerk is busy by typing 12. Click in the Idle/Hour cell and assign the idle cost rate by typing 12.

You can edit the fields for any module using Arena's spreadsheet, including flowchart modules. Just click on the icon in the Basic Process panel to display its spreadsheet.

3.4.6 Prepare for the simulation

To make the model ready for simulation, we'll specify general project information and the duration of the simulation run. Since we're just testing our first-cut model, we'll perform a short, 20-day run.



Project Title: Mortgage Review Process

Analyst Name: Rockwell Software

Project Description:

Statistics Collection

☑ Costing ☑ Queues ☐ Transporters

☑ Entities ☑ Processes ☐ Conveyors

☑ Resources ☐ Stations ☐ Activity Areas

Number of Replications: 1

Initialize Between Replications: ☑ Statistics ☑ System

Start Date and Time: ☐ 2011. január 16. 6:27:16

Warm-up Period: 0.0 Time Units: Hours

Replication Length: 20 Time Units: Days

Hours Per Day: 24 Base Time Units: Hours

Terminating Condition:

Display 3.6: The Run Setup *Project Parameters* (left) and *Replication Parameters* (right) Dialog Boxes

1. Open the Project Parameters dialog by using the *Run>Setup* menu item and clicking the *Project Parameters* tab (*Display 3.6*). In the Project Title field, type *Mortgage Review Analysis*; we'll leave the Statistics Collection check boxes as the defaults, with Entities, Queues, Resources, and Processes checked and also check the costing box.

2. Next, click the *Replication Parameters* tab within the same Run Setup dialog. In the Replication Length field, type 20; and in the Time Units field directly to the right of Replication Length, select *days* from the pull-down list. Click OK to close the dialog.

3.4.7 Save the simulation model

Now that you've done some work on your model, it seems like a good time to save it. Click **Save** on the Standard toolbar or select the *File>Save* menu item. Arena will prompt you for a destination folder and file name. Browse to the target folder in which you want to save the model (e.g., C:\My Documents\Arena) and type a name (e.g., Model 03-01) in the file name field.

Arena's model files store all of the model definition, including the flowchart, other graphics you've drawn, and the module data you entered. When you perform a simulation run, the results are stored in a database using the same name as the model file.

If Arena displays an error message, you can use the *Find* button in the error window to locate the source of the problem. You can change between the error and model windows by selecting them from the Window menu.

3.5. Simulate the process

With these few, short steps, we are ready to predict the future! The mortgage application model contains all of the information needed to run the simulation.

Start the simulation run by clicking the *Go* button or clicking the *Run*>*Go* menu item. Arena first will check to determine whether you've defined a valid model, then will launch the simulation.

As the simulation progresses, you'll see small entity pictures resembling pages moving among the flowchart shapes. Also, a variety of variables change values as entities are created and processed, as illustrated by *Figure 3.7*.

If the animation is moving too fast, you can slow it down by adjusting the animation scale factor. For this, you have two choices:

Open the Run Setup dialog via the *Run*>*Speed*>*Animation Speed Factor* menu item and enter a smaller value (e.g., *0.005*) for the scale factor*;* or

Use the less-than (<) key during the run to decrease the scale factor by 20%. Be sure that the model window is active not the Navigate panel or > and < won't take effect. Pressing < repeatedly is an easy way to fine tune the animation speed. The greater-than (>) key speeds up animation by 20%.

To pause the simulation, click the *Pause* button or press the *Esc* key.

If the run finishes before you have a chance to explore these controls, answer No when you're asked if you want to view the results. Then click the *Start Over* button on the *Run* toolbar to begin the run again.

The animation scale factor is the amount of simulated time between successive screen updates. Smaller values provide smoother, slower animation.

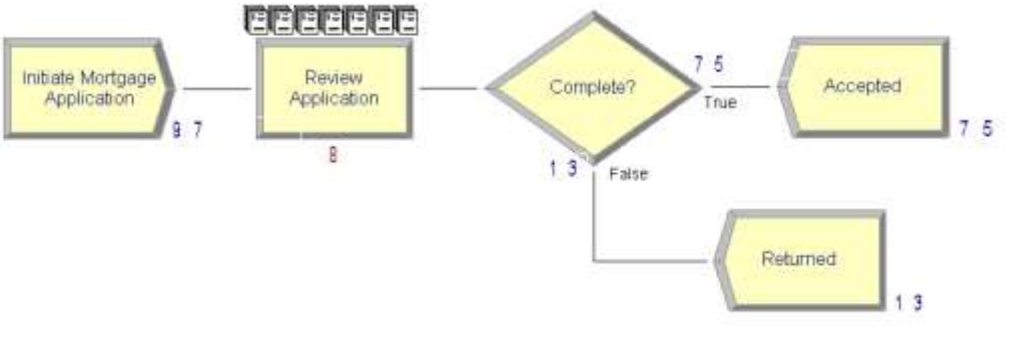


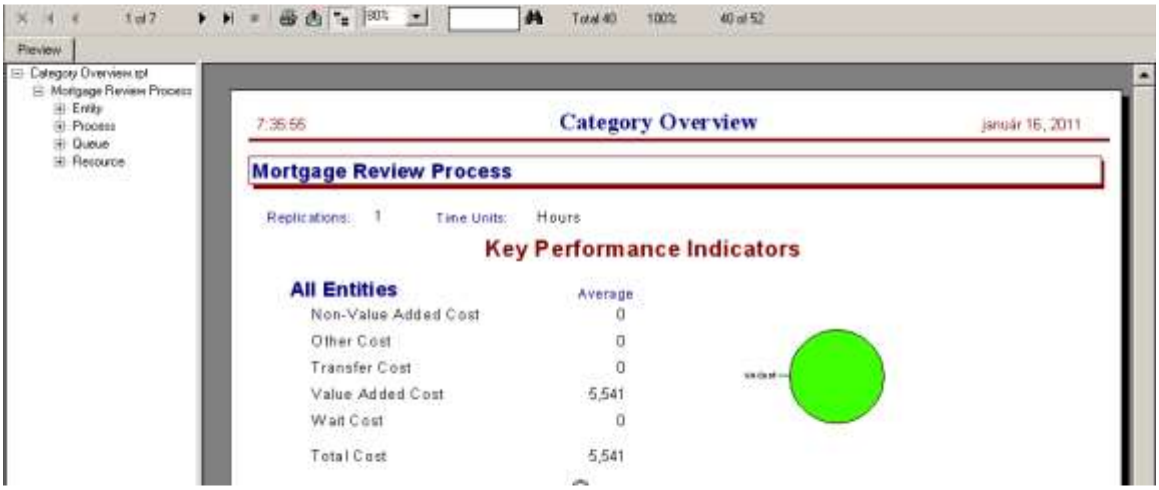
Figure 3.7: The Simulation Progresses

With the automatic flowchart animation, you can see how many entities have been created, are currently in the Review Application process (Figure 3.7), have left each branch of our Decide module, and have left the model at each of our terminating Dispose modules. These variables can be helpful in verifying the model. For example, if the probability in the Decide shape was entered wrong (e.g., if you typed 12 the rejection probability instead of 88), the variables would show that many more applications were leaving the Returned branch.

You also can step through the simulation one event at a time. Pause (⏸) the simulation, then click the Step button or press the F10 key. Each time you step the simulation, an entity is moved through the flowchart. Usually, you'll see animation of the entity's movement, though sometimes no visual change will take place (e.g., when the next event is creating a new entity). When this occurs, just step again to move forward to the next event.

3.5.1 View simulation reports

After you've watched some of the animated flowchart, you can quickly run to the end of the simulation to view reports. *Pause* the simulation, then click the *Fast Forward* button to run the simulation without updating the animation.



7:35:56 **Category Overview** január 16, 2011

Mortgage Review Process

Replications: 1 Time Units: Hours

Key Performance Indicators

All Entities	Average
Non-Value Added Cost	0
Other Cost	0
Transfer Cost	0
Value Added Cost	5,541
Wait Cost	0
Total Cost	5,541

Figure 3.8: The Category Overview Report

At the end of the run, Arena will ask whether you'd like to view reports. Click *Yes*, and the default report (the Category Overview Report) will be displayed in a report window, as shown *Figure 3.8*.

Each of Arena's reports is displayed in its own window within the Arena application. You can use the standard window options (maximize, minimize, etc.) by clicking on the window control buttons or by pulling down the window menu.

Table 3.1

Question	Report Section	Answer
On average, how long did mortgage applications spend in the modeled process?	Total Time (Entity), Average column	16.51 hrs
What was the average, cost of reviewing a mortgage application?	Total Cost (Entity), Average column	€22.99
What was the longest time an application spent in review?	Total Time (Process), Maximum column	33.45 hrs
What was the maximum number of applications waiting for review?	Number Waiting (Queue), Maximum column	21 applications
What proportion of time was the Mortgage Review Clerk busy?	Utilization (Resource), Average column	97%

On the left side of each report window is a tree listing the types of information available in the report. The project name (in our case, Mortgage Review) is listed at the top of the tree, followed by an entry for each category of data. This report summarizes the results across all replications (although, in this model, we have only one replication). Other reports provide detail for each replication.

By clicking on the entries inside the category sections, you can view various types of results from the simulation run. The *Table 3.1* illustrates some of the questions you could answer from the Category Overview Report on our simple mortgage application process simulation.

After you've browsed the **Category Overview Report**, you can close it by clicking on the window icon to the left of the *File* menu and clicking *Close*. You can look at other reports by clicking on their icons in the Project Bar. Each report will be displayed in its own window. To return to the model window, close all of the report windows or select the model file from the Window menu.

After you have viewed the reports and returned to the model window, end the Arena run session by clicking the *End* button.

3.6. Enhance the visualization of the process

Now that we've completed the basic steps for analyzing the mortgage application process, we can return to our model and embellish the graphical animation to gain further insight into the process dynamics. Animation also can be of great benefit in enticing others in the organization to be interested in process improvement.

We'll add two animation components to the mortgage model. First, we'll show our Mortgage Review Clerk working at a desk, either busy or idle. To gain a better sense of how many applications are waiting in the Review Application process over time, we'll also add a dynamic plot of the work-in-process (WIP) simulation variable. Our Arena model will appear as shown *Figure 3.9* after we add these two objects.

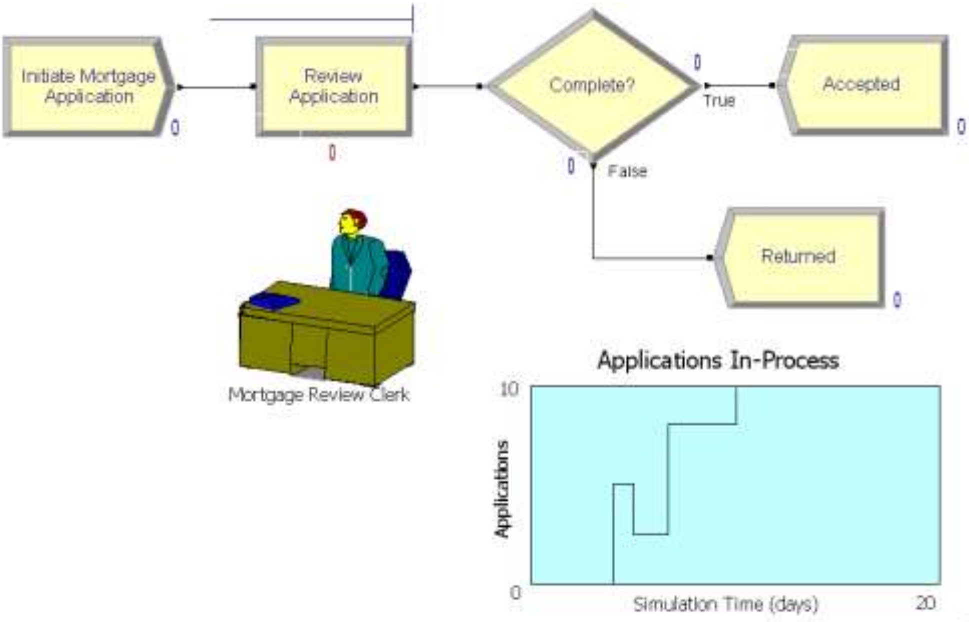


Figure 3.9: Embellishing the Graphical Animation

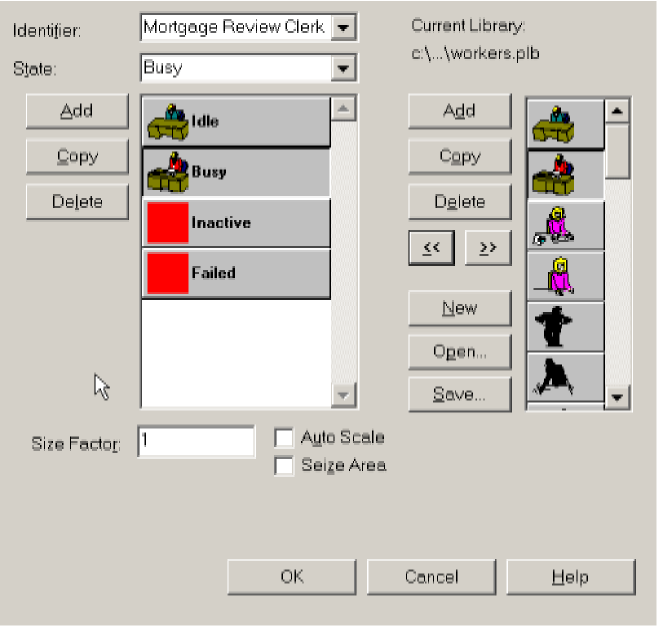
You can toggle between the split view (flowchart and spreadsheet) and a full-screen view of either area by clicking the **Split Screen** toolbar button or selecting the **View > Split Screen** menu item. When in full-screen view, clicking the icons on the Basic Process panel displays the appropriate view (flowchart for flowchart modules and spreadsheet for data-only modules).

3.5.1 Animate the mortgage review clerk resource

During the simulation run, our Mortgage Review Clerk resource can be in one of two states. If no mortgage application entity is in-process, then the resource is *idle*. We'll use a picture of a person sitting at a desk to depict idleness. When an entity seizes the resource, the Mortgage Review Clerk's state is changed to *busy*, in which case our picture will show the person reviewing a document.

1. Click the *Resource* button on the **Animate** toolbar.

2. The Resource Placement dialog appears (*Display 3.7*). Select the *Mortgage Review Clerk* from the pull-down list in the Identifier field so that this object animates the Mortgage Review Clerk.



Identifier: Mortgage Review Clerk

State: Busy

Current Library: c:\...\workers.plb

Size Factor: 1

□ Auto Scale

□ Seize Area

Display 3.7: The Resource Placement Dialog

3. Open the Workers picture library by clicking the *Open* button, then browsing to the *Workers.plb* file in the Arena application folder (e.g., C:\Program Files\Arena) and double-clicking on it.

4. To change the idle picture:

   Click the *Idle* button in the table on the left.

   Select from the picture library table on the right the picture of the worker sitting down.

   Click the *Transfer* button between the tables to use the worker picture for the Idle resource state.

5. To change the busy picture:

   Click the *Busy* button in the table on the left.

   Select from the picture library table on the right the picture of the worker reading a document.

   Click the *Transfer* button between the tables to use the selected picture when the Mortgage Review Clerk is busy.

6. Click *OK* to close the dialog. (All other fields can be left with their default values.)

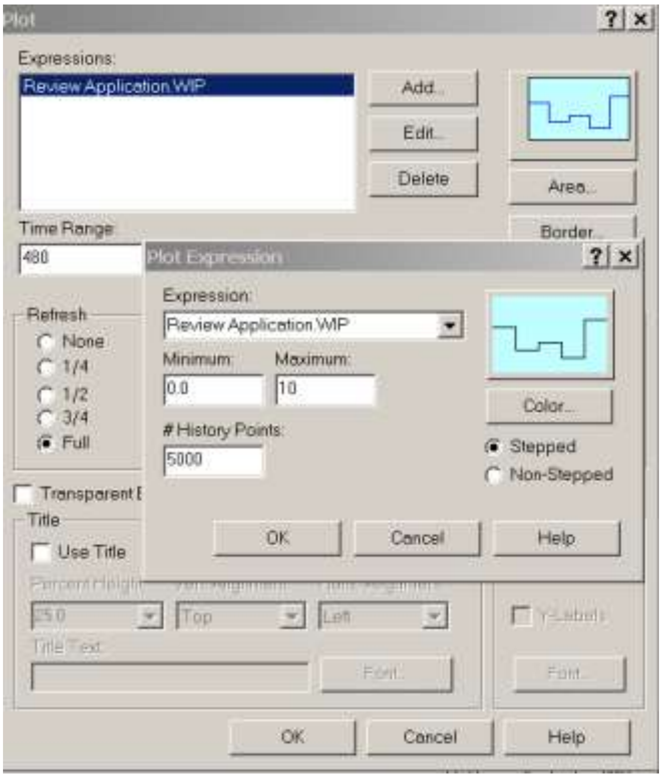
7. The cursor will appear as a cross hair. Move it to the model window and click to place the Mortgage Review Clerk resource animation picture.

8. If you'd like to have the clerk appear a bit larger, select the picture and use the resize handles to enlarge it.

3.5.2 Plot the number of applications in-process

Our second animation enhancement is a plot of how many mortgage applications are under review as the simulation progresses. This will give us a sense of the dynamics of the workload, which can vary quite a bit when the random nature of processes is incorporated into a simulated model.

1. Click the *Plot* button on the **Animate** toolbar.



Plot

Expressions: Review Application.WIP

Time Range: 480

Refresh: ○ None ○ 1/4 ○ 1/2 ○ 3/4 ● Full

□ Transparent B

Title: □ Use Title

Plot Expression

Expression: Review Application.WIP

Minimum: 0.0

Maximum: 10

History Points: 5000

● Stepped ○ Non-Stepped

Display 3.8: The Plot Dialog Box

2. The Plot dialog appears (*Display 3.8*). We'll plot a single expression, the work-in-process (WIP) at the Review Application process. To add the expression, click *Add*.

3. In the Plot Expression dialog that appears, right-click in the Expression field to open the Expression Builder.

4. We want to plot the number of entities in our Review Application process over time. Select **Review Application** from the pull-down list in the Process Name field, then choose **WIP** from the Information pull-down list. Click *OK* to close the Expression Editor.

5. From our reports in the earlier simulation run, we noted that the maximum number of applications in the process was 9. Let's set our plot Maximum value to *10*.

6. In the History Points field type *5000*, which will plot the most recent 5000 values of the variable during the simulation run. Click **OK** to close the Plot Expression dialog. Note that Arena places the formula (Review Application.WIP) in the Plot Expression field.

7. To complete the plot definition, change the Time Range to *480*. Our plot's horizontal axis will represent 480 hours (20 days) of simulated time, matching our run length. Click *OK* to close the Plot dialog.

8. The cursor changes to a cross hair. Draw the plot in the model window by clicking to locate each of the two opposite corners (e.g., the top-left and bottom-right corners), placing the plot below the flowchart and to the right of the resource.

With the edits complete, you may want to save them by clicking *Save* or pressing *Ctrl+S*.

You can plot many expressions on the same set of axes by adding multiple expressions in the Plot dialog. Each can be color-coded so that you can readily compare data such as workloads in processes, waiting customers, etc.

3.5.3 Rerun the simulation

Now that we've made our animation more interesting and valuable, let's run the simulation again. Because we didn't change any of the process parameters (i.e., data in the modules) the simulation will provide the same results.

Click *Run* (or press the *F5* key) to start the simulation. As the simulation progresses, you'll notice the Mortgage Review Clerk's picture change from idle (sitting at the desk) to busy (reading a document) and back again, as mortgage application entities move through the Review Application process.

The plot shows some significant peaks in the number of applications that are under review, caused by the combination of the variation in the time between arrivals of applications (defined in the **Create** module) and the time to process applications (**Process** module).

3.6. Next steps

You've succeeded in modeling, simulating, visualizing, and analyzing a simple mortgage application process. To further explore Arena's capabilities, try solving a few of these extensions to the process.

1. Add a screening process before the application is reviewed.

Applications can be screened in as little as 15 minutes. Most often, it takes about 25 minutes for the screening, though sometimes it can require as much as 45 minutes. Assign a Receptionist (rate of €6.75/hour) to perform the screening. What proportion of the Receptionist's time will be used in this task?

2. Return some applications to the mortgage applicants after the screening process.

On completion of the screening, 8% of the applications are returned. Also, because many of the deficient applications are caught in the new screening, the percentage of applications that are accepted in the formal review is raised from 88% to 94%, and the Mortgage Review process time is reduced by 10%. By how much did the cost of reviewing an application change? How about the total time to review applications?

To view completed Arena models for the main tutorial and these two extensions, browse to the Examples folder and open Model 03-01.doe, Model 03-02.doe, and Model 03-03.doe.

CHAPTER 4

Modeling Basic Operations and Inputs

In Chapters 2 and 3, we introduced you to a simple processing system (Model 3-1), conducted a hand simulation (Chapter 2), and examined an Arena model (Chapter 3). In this chapter, we'll work with a more realistic system and develop complete models of several versions of that system, with each version adding complexity and new modeling concepts. We'll also talk about how you can realistically specify input probability distributions that represent the actual system under study.

Section 4.1 describes this more complicated system-a sealed electronic assembly and test system. We then discuss how to develop a modeling approach, introduce several new Arena concepts, build the model, and show you how to run it and view the results. By this time, you should start to become dangerous in your modeling skills. In Section 4.2, we'll enhance the model by enriching the scheduling, failure, and states of resources, and give you alternate methods for studying the results. Section 4.3 shows you how to dress up the animation a little bit. In Section 4.4, we'll generalize how entities move around, introducing the notions of Stations, Routes for non-zero travel times, and animation of transfers. Finally, in Section 4.5, we take up the issue of how you specify quantitative inputs, including probability distributions from which observations on random variables are "generated" to drive your simulation. When you finish this chapter, you should be able to build some reasonably elaborate models of your own, as well as specify appropriate and realistic distributions as input to your models.

Model 4-1: An Electronic Assembly and Test System

This system represents the final operations of the production of two different sealed electronic units, shown in *Figure 4.1*. The arriving parts are cast metal cases that have already been machined to accept the electronic parts.

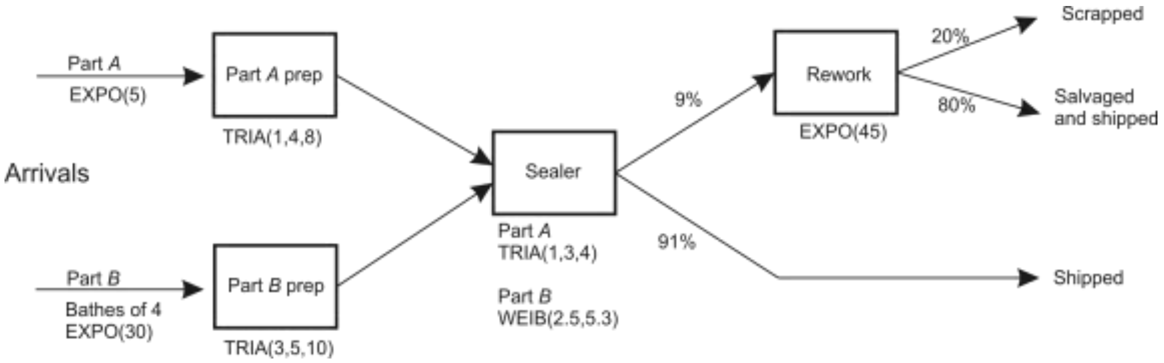


Figure 4.1. Electronic Assembly and Test System

The first units, called Part A, are produced in an adjacent department, outside the bounds of this model, with inter-arrival times to our model being exponentially distributed with a mean of 5 (all times are in minutes). Upon arrival, they're transferred (instantly) to the Part A Prep area, where the mating faces of the cases are machined to assure a good seal, and the part is then deburred and cleaned; the process time for the combined operation at the Part A Prep area follows a TRIA(1, 4, 8) distribution. The part is then transferred (instantly, again) to the sealer.

The second units, called *Part B*, are produced in a different building, also outside this model's bounds, where they are held until a batch of four units is available; the batch is then sent to the final production area we are modeling. The time between the arrivals of successive batches of *Part B* to our model is exponential with a mean of 30 minutes.

Upon arrival at the *Part B Prep area*, the batch is separated into the four individual units, which are processed individually from here on, and the individual parts proceed (instantly) to the *Part B Prep area*. The processing at the *Part B Prep area* has the same three steps as at the Part A Prep area, except that the process time for the combined operation follows a TRIA(3, 5, 10) distribution. The part is then sent (instantly) to the sealer.

At the sealer operation, the electronic components are inserted, the case is assembled and sealed, and the sealed unit is tested. The total process time for these operations depends on the part type: TRIA(1, 3, 4) or Part A and WEIB(2.5, 5.3) for *Part B* (2.5 is the scale parameter β and 5.3 is the shape parameter α; see Appendix D). Ninety-one percent of the parts pass the inspection are transferred immediately to the shipping department; whether a part passes is independent of whether any other parts pass. The remaining parts are transferred instantly to the rework area where they are disassembled, repaired, cleaned, assembled, and re-tested. Eighty percent of the parts processed at the rework area are salvaged and transferred instantly to the shipping department as reworked parts, and the rest are transferred instantly to the scrap area. The time to rework a part follows an exponential distribution with mean of 45 minutes and is independent of part type and the ultimate disposition (salvaged or scrapped).

We want to collect statistics in each area on resource utilization, number in queue, time in queue, and the cycle time (or total time in system) separated out by shipped parts, salvaged parts, or scrapped parts. We will initially run the simulation for four consecutive 8-hour shifts, or 1,920 minutes.

4.1.1 Developing a Modeling Approach

Building a simulation model is only one component of a complete simulation project. We will discuss the entire simulation project later. Presume for now that the first two activities are to state the study objective and define the system to be studied. In this case, our objective is to teach you how to develop a simulation model using Arena. The system definition was given above. In the real world, you would have to develop that definition, and you may also have to collect and analyze the data to be used to specify the input parameters and distributions (see Section 4.5). We recommend that the next activity be the development of a modeling approach. For a real problem, this may require the definition of a data structure, the segmentation of the system into submodels, or the development of control logic. For this problem, it requires only that we decide which Arena modules will provide the capabilities we need in order to capture the operation of the system at an appropriate level of detail. In addition, we must decide how we're going to model the different processing times at the sealer operation. To simplify this task, let's separate the model into the following components: part arrival, prep areas, sealer operation, rework, part departure, and part animation. Also, we'll assume that all entities in the system represent individual parts that are being processed.

Because we have two distinct streams of arriving entities to our model, each with its own timing pattern, we will use two separate **Create** modules (one for each part type) to generate the arriving parts.

We also have different processing times by part type at the sealer operation, so we'll use two **Assign** modules to define an attribute called *Sealer Time* that will be assigned the appropriate sealer processing time after the parts are generated by the **Create** modules. When the parts are processed at the sealer operation, we'll use the time contained in the *Sealer Time* attribute for the processing time there, rather than generating it on the spot as we did in Model 3-1.

Each of the two prep areas and the sealer operation will be modeled with its own **Process** module, very much like the **Process** module used in Model 3-1. An inspection is performed after the sealer operation has been completed, which results in parts going to different places based on a "coin flip" (with just the right bias in the coin). We'll use a **Decide** module with the pass or fail result being based on the coin flip. The rework area will be modeled with Process and **Decide** modules, as it also has a pass or fail option. The part departures will be modeled with three separate **Record** and **Dispose** modules (shipped, salvaged, and scrapped) so we can keep corresponding cycle-time statistics sorted out by shipped vs. salvaged vs. scrapped. All of these modules can be found on the Basic Process panel.

4.1.2 Building the Model

To build the model, you need to open a new model window and place the required modules on the screen: two **Create**, two **Assign**, four **Process**, two **Decide**, three **Record**, and three **Dispose** modules.

Your model window should now look something like *Figure 4.2*, assuming you've made the Connections or used the Auto-Connect feature (Object menu) while placing the modules in the appropriate sequence (the numbers inside your module shapes might be different if you placed your modules in a different order, but that doesn't matter since they're all "blanks" at this point). You might want to use the File > Save function now to save your model under a name of your choosing.

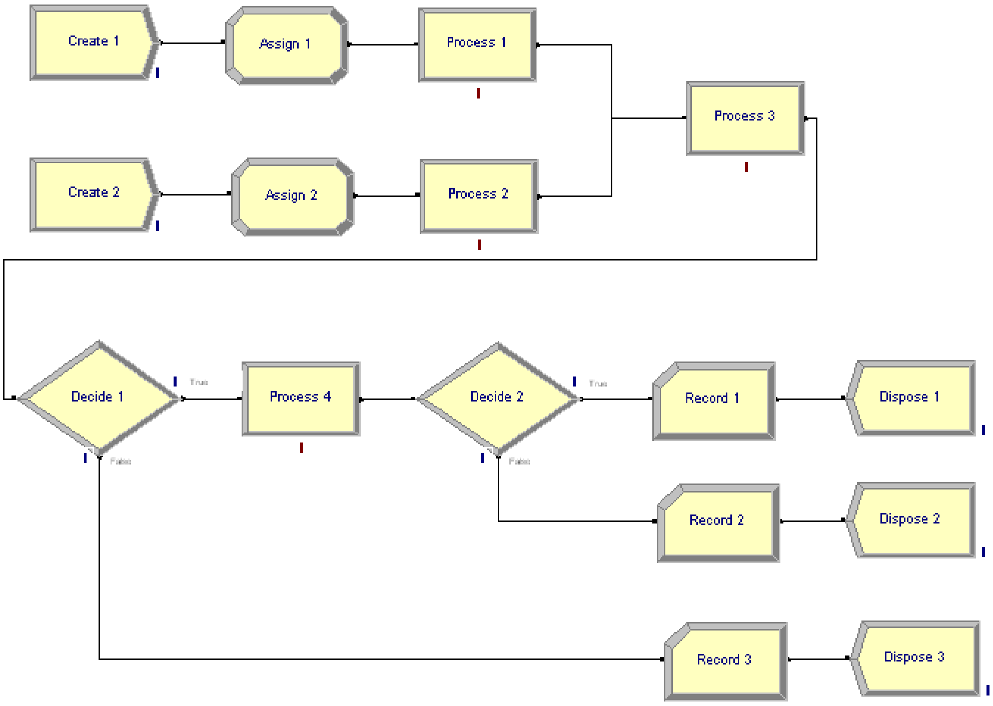
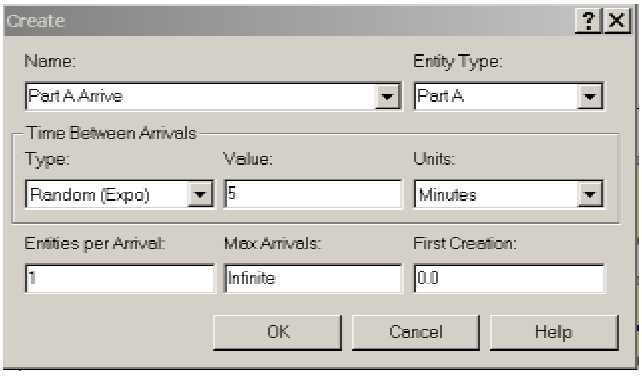


Figure 4.2. Model Window of Placed Modules

Now let's open each module and enter the information required to complete the model. Start with the **Create 1** module that will create the arriving Part A entities. *Display 4.1* (the "Display" device was described in Section 3.4.4) provides the information required to complete this module. Note that this is very similar to the **Create** module used in Model 3-1. We've given the module a different Name and specified the Entity Type as Part A. The Time Between Arrivals is Random (i.e., an exponential distribution) with a Value (i.e., mean) of 5, and the units are set to Minutes. The remaining entries are the default options. We can now accept the module by clicking OK.



Name: Part A Arrive
Entity Type: Part A
Time Between Arrivals
Type: Random (Expo)
Value: 5
Units: Minutes
Entities per Arrival: 1
Max Arrivals: Infinite
First Creation: 0.0

Name Entity Type	Part A Arrive Part A
Type Value Units	Random (Expo) 5 Minutes
Entities per Arrival	1

Display 4.1: The Completed Part A Create Dialog Box

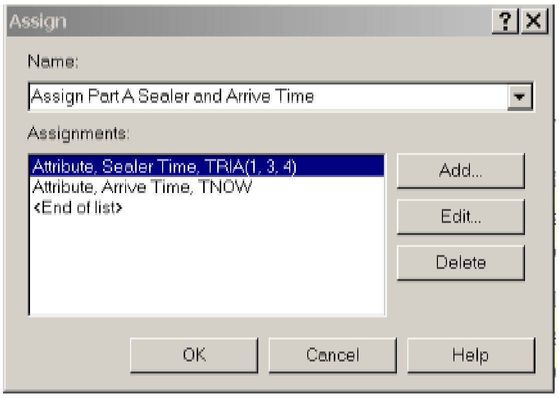
Name Entity Type	Part B Arrive Part B
Type Value Units	Random (Expo) 30 Minutes
Entities per Arrival	4

Display 4.2. The Completed Part B Create Dialog Box Entries

The **Create** module for the *Part B* arrivals is very similar to that for *Part A*, as shown in *Display 4.2* (we'll skip the graphics since they're almost the same as what you just saw), except we have filled in one additional field (Entities per Arrival) to reflect the batch size of 4. Recall that the *Part B* entities arrive in batches of four. Thus, this entry will cause each arrival to consist of four separate entities rather than one.

Having created the arriving parts, we must next define an attribute *Sealer Time* and assign it the sealer processing time, which is different for each part type. We'll assign these values in the **Assign 1** and **Assign 2** modules that we previously placed. The *Part A* assignment is shown in *Display 4.3*. We've defined the new attribute and assigned it a value from a TRIA(1,3,4) distribution. We've also defined an attribute, Arrive Time, which is used to record the arrival time of the entity. The Arena variable TNOW provides the current simulation time, which in this case is the time the part arrived or was created (a good way to discover TNOW is to right-click in the New Value field of the **Assign** module's Assignments dialog box, select Build Expression, guess Date and Time Functions, and it's first in the list, described as Current Simulation Time).

The assignment to the *Sealer Time* and Arrive Time attributes for *Part B* is shown in *Display 4.4*. Although four entities are created in the previous module for each arrival, they'll each be assigned a different (independent) value from the sealer-time distribution in the following **Assign** module.



Name:
Assign Part A Sealer and Arrive Time

Assignments:
Attribute, Sealer Time, TRIA(1, 3, 4)
Attribute, Arrive Time, TNOW
<End of list>

Name	Assign Part A Sealer and Arrive Time
Type Attribute Name New Value	Attribute Sealer Time TRIA (1,3,4)
Type Attribute Name New Value	Attribute Arrive Time TNOW

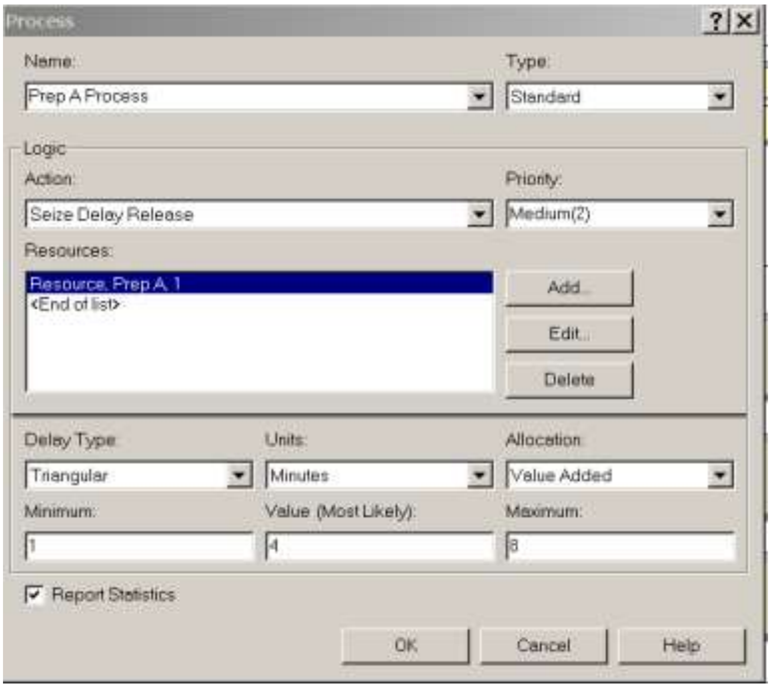
Display 4.3. Assigning the Part A Sealer Time and Arrival Time

Name	Assign Part B Sealer and Arrive Time
Type Attribute Name New Value	Attribute Sealer Time WEIB (2.5, 5.3)
Type Attribute Name New Value	Attribute Arrive Time TNOW

Display 4.4. Assigning the Part B Sealer Time and Arrival Time

Having completed the two part-arrival modules and the assignment of the sealer times, we can now move to the two prep areas that are to be modeled using the two **Process** modules previously placed. The completed dialog box for the *Prep A Process* area is given in *Display 4.5*.

The **Process** module has four possible Actions: Delay, Seize Delay, Seize Delay Release, and Delay Release. The Delay action will cause an entity to undergo a specified time Delay. This Action does not require a Resource. This implies that waiting will occur and that multiple entities could undergo the Delay simultaneously. Since our prep area requires the use of a machine or Resource, we need an Action that will allow for waiting, queuing until the prep resource is available, and delaying for the processing time. The Seize Delay Action provides the waiting and delaying, but it does not release the Resource at the end of processing to be available for the next entity. If you use this Action, it is assumed that the Resource would be Released downstream in another module. The Seize Delay Release option provides the set of Actions required to model our prep area accurately. The last Action, Delay Release, assumes that the entity previously Seized a Resource and will undergo a Delay here, followed by the Release of the Resource.



Name: Prep A Process

Type: Standard

Logic

Action: Seize Delay Release

Priority: Medium(2)

Resources:
Resource, Prep A, 1
<End of list>

Delay Type: Triangular

Units: Minutes

Allocation: Value Added

Minimum: 1

Value (Most Likely): 4

Maximum: 8

☑ Report Statistics

Name	Prep A Process
Action	Seize Delay Release
Resource	
Type	Resource
Resource Name	Prep A
Quantity	1
Delay Type	Triangular
Units	Minutes
Minimum	1
Value (Most Likely)	4
Maximum	8

Display 4.5. Prep A Process Dialog Box

You might notice that when you select one of the last three options, a list box appears in the empty space below the Action selection box. Click Add to enter the Resource information.

In entering data, we strongly urge you to make use of drop-down lists whenever possible. The reason for this caution is that once you type a name, you must always match what you typed the first time. Arena names are not case-sensitive, but the spelling and any embedded blanks must be identical. Picking the name from the list assures that it is the same. If you type in a slightly different name, Arena will give you an error message the first time you check or attempt to run the model (or, worse yet, your model might run but it will be wrong).

Also note that when you place a module, Arena automatically provides default names and values. These default names are the object name (module, resource, etc.) with an appended number. The appended number is incremented for each additional name, if a unique name is required; for example, Process 1, Process 2, and so on. There are two reasons for this. The first is a matter of convenience-you can accept the default resource name, or you can change it. The second reason is that all names for any objects in Arena must be unique, even if the object type is different. Otherwise, Arena could not determine which object to associate with a name that had been used more than once.

To help you, Arena does a lot of automatic naming, most of which you won't even notice. For example, if you click on the **Queue** data module, you'll see that Arena also assigned the name *Prep A Process.Queue* to the queue at this prep area. In most cases, you can assign your own names rather than accepting the default names.

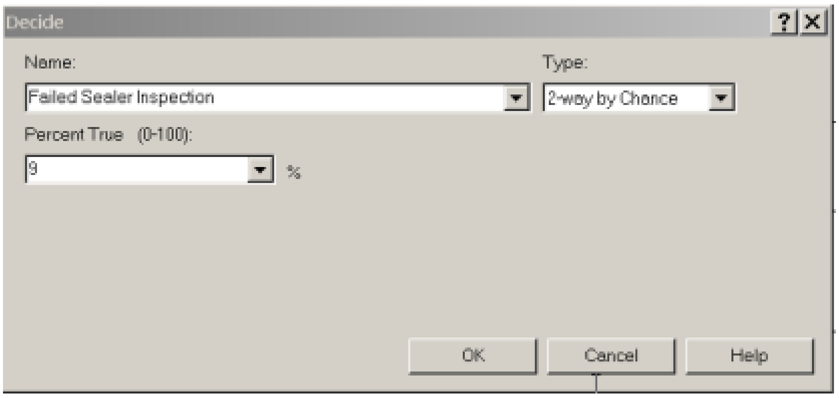
You might also notice that when you select either of the two actions that include a Seize and then accept the module, Arena will automatically place an animated queue (a horizontal line with a small vertical line at the right) near the associated **Process** module. This will allow you to visualize entities waiting in the queue during the simulation run. If you click on this queue, the queue name will be displayed.

The second **Process** module is filled out in an almost-identical fashion, with the exception of the name (*Prep B Process*), the resource name (*Prep B*), and the parameters for the process time (3,5,10). We have not included a display for this module.

The next step is to enter data for the sealer operation, which is the third **Process** module we placed. The entries for the dialog box are shown in *Display 4.6*. Note that in the upstream **Assign** modules we defined the attribute *Sealer Time* when the arriving parts were created. When an entity gains control of, or seizes, the resource, it will undergo a process delay equal to the value contained in its *Sealer Time* attribute.

Name Action	Sealer Process Seize Delay Release
Resource Resource Name Quantity	 Sealer 1
Delay Type Units Expression	Expression Minutes Sealer Time

Display 4.6. The Sealer Dialog Box


Decide

Name: Failed Sealer Inspection

Type: 2-way by Chance

Percent True (0-100): 9 %

Name Percent True	Failed Sealer Inspection 9

Display 4.7. The Sealer Inspection Dialog Box

The inspection following the sealer operation is modeled using the first **Decide** module. We'll accept the default Type, 2-way by Chance, as we have only a pass or fail option. The dialog box requires that we enter a Percent True, and it provides two ways for entities to leave the module-*True* or *False*. In this case, we will enter the Percent True as 9%. This will result in 9% of the entities (which we'll treat as the failed items) following the True branch, and 91 %

(the passed items) following the False branch.[*] Parts that pass are sent to Shipping, and parts that fail are sent to Rework. The data for this **Decide** module are shown in *Display 4.7*. By the way, if you've been building this model as we've moved through the material, now would be a good time to click the Save button-you never know when somebody might bump the power switch!

The remaining **Process** module will be used to model the rework activity. The data for this **Process** module are in *Display 4.8*. This module is very similar to the *Prep A* and *B* **Process** modules with the exceptions of the Name, Resource Name, and Expression.

Name	Rework Process
Action	Seize Delay Release
Resource	
Resource Name	Rework
Quantity	1
Delay Type	Expression
Units	Minutes
Expression	EXPO(45)

Display 4.8. The Rework Process Dialog Box

Name	Failed Rework Inspection
Percent True	20

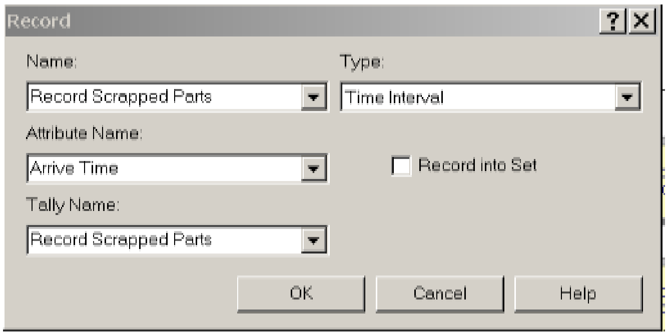
Display 4.9. The Rework Inspection Dialog Box

The final **Decide** module is used to separate the salvaged and scrapped parts following rework. The data for this **Decide** module are in *Display 4.9*. We've chosen the True branch to represent the scrapped parts (20%) and the False branch to represent the salvaged parts.

Having defined all of the operations, we now need to fill in the **Record** and **Dispose** modules. Remember that as part of the simulation output, we wanted to collect statistics on resource utilization, number in queue, and time in queue at each of the operations. These three statistics are automatically collected whenever you use a **Process** module with an Action option that requires a Resource (assuming that the Report Statistics box for the module is checked and that the Processes box is checked in *Run>Setup>Project Parameters*). We also wanted statistics on the cycle time separated by shipped parts, salvaged parts, and scrapped parts. The **Record** module provides the ability to collect these cycle times in the form of Tallies. The completed dialog box for the scrapped parts tally is shown in *Display 4.10*. We picked the Type Time Interval from the dropdown list. The Tally Name defaults to the module name. This will cause Arena to record as a Tally statistic the time between the attribute carrying the arrival time (Arrive Time) of the part to the system and the time that it arrived at this **Record** module, which will be the entity's time in system.

The remaining two **Record** modules are named Record Salvaged Parts and Record Shipped Parts. We have not bothered to include displays on these modules as they are completely analogous to the Record Scrapped Parts module in *Display 4.10*.

[*] We could have just as well reversed the interpretation of True as fail and False as pass, in which case the Percent True would be 91 (and we a probably change the module Name to Passed Sealer Inspection).



Record

Name: Record Scrapped Parts

Type: Time Interval

Attribute Name: Arrive Time

□ Record into Set

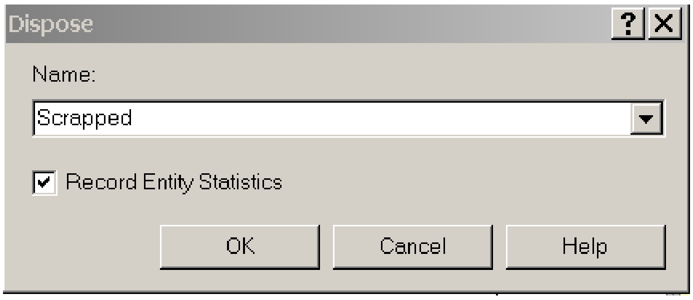
Tally Name: Record Scrapped Parts

Name	Record Scrapped Parts
Type	Time Interval
Attribute Name	Arrive Time
Tally Name	Record Scrapped Parts

Display 4.10. The Scrapped Parts Tally Dialog Box

The final three modules Dispose of the entities as they leave the system. For this model, we could have directed all entities to a single **Dispose** module. However, one of the features of a **Dispose** module is the inclusion of an animation variable, which appears near the lower right-hand corner of the module. This animation variable will display the current count for the total number of entities that have passed through this module during the run and allow the viewer to discern the proportion of entities that have taken each of ' the three possible paths through the system.

The data for the Scrapped **Dispose** module are shown in *Display 4.11*. We have defaulted on the check in the box entitled Record Entity Statistics. However, if we had wanted to keep entity flow statistics only on the parts that were shipped, including the salvaged parts, then we could have cleared this box (leaving checks in the remaining two **Dispose** modules). Doing so would have caused only the selected parts to be included in the automatic entity-flow statistics. Of course, you need to make sure the Entities box in *Run>Setup>Project Parameters* is checked in order to get any of these results.



Dispose

Name: Scrapped

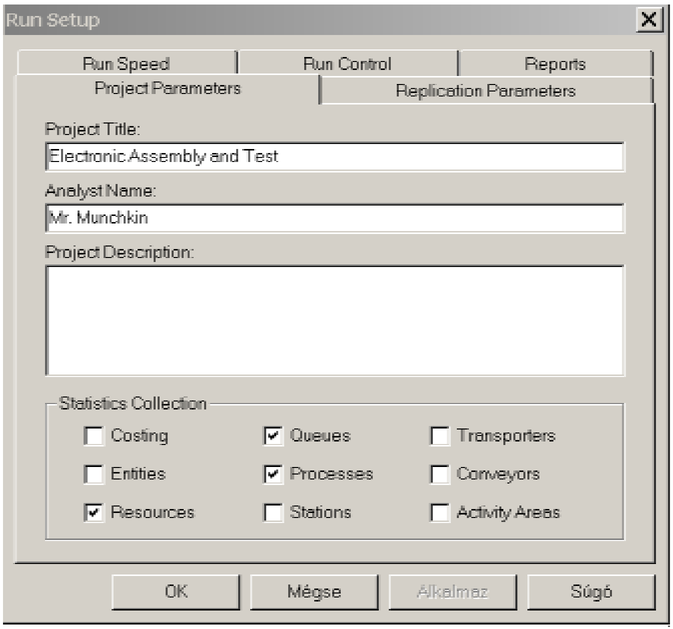
☑ Record Entity Statistics

Display 4.11. The Scrapped Dispose Dialog Box

The two other **Dispose** modules, Salvaged and Shipped, are filled out in a similar way.

You're nearly ready to run the model. Although it has taken you some time to get this far, once you get accustomed to working with Arena, you'll find that you could have accomplished these steps in only a few minutes.

The model could actually run at this point, but once started, it would continue to run forever because Arena doesn't know when to stop the simulation. You establish the run parameters for the model by selecting *Run* > *Setup*. The Run Setup dialog box has five tabbed pages that can be used to control the simulation run. The data for the first tab, Project Parameters, are shown in *Display 4.12*. We've entered the project title and analyst name so they will appear on the output reports, and also a brief synopsis under Project Description for documentation purposes. In the statistics collection area, we have cleared the Entities selection as we don't need those data for our analysis (so we won't get statistics on time in system sorted out by entity type). You might try running the simulation with this box checked in order to see the difference in the output reports.



Project Title: Electronic Assembly and Test

Analyst Name: Mr. Munchkin

Project Description:

Statistics Collection

□ Costing ☑ Queues □ Transporters

□ Entities ☑ Processes □ Conveyors

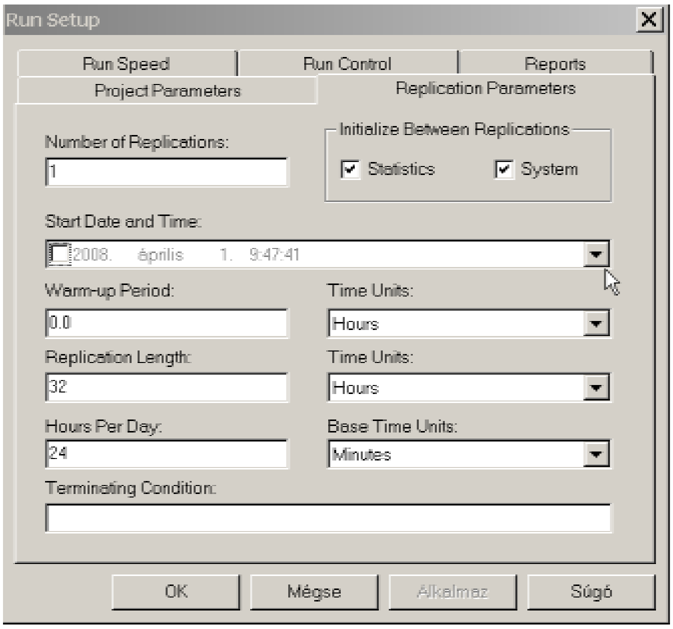
☑ Resources □ Stations □ Activity Areas

Project Title Analyst Name	Electronic Assembly and Test Mr. Munchkin
Project Description	The first version of the electronic assembly and test model, as described in Section 4.1.
Statistics Collection Entities	clear

Display 4.12. The Run Setup Project Parameters

You also need to specify the run length, which is done under the Replication Parameters tab. We've set the Replication Length to 3 2 hours (four consecutive 8-hour shifts), the Base Time Units to Minutes, and defaulted the remaining fields. The completed dialog box is shown in *Display 4.13*. We've also accepted the defaults for the remaining three tabs in *Run>Setup*: Run Speed, Run Control, and Reports. You might want to look at these tabs to get an idea of the options available.

Before we run our newly created model, let's give it one final tweak. Since we have two different part types, it might be nice if we could distinguish between them in the animation. Click on the Entity data module found in the Basic Process panel and note that the initial picture for both parts is *Picture.Report*. When we run our model, all of our parts will use this same icon for displaying entities on the animation.



Run Setup ☒

Run Speed | Run Control | Reports | Project Parameters | Replication Parameters

Number of Replications: 1

Initialize Between Replications: ☑ Statistics ☑ System

Start Date and Time: □ 2008. április 1. 9:47:41

Warm-up Period: 0.0

Time Units: Hours

Replication Length: 32

Time Units: Hours

Hours Per Day: 24

Base Time Units: Minutes

Terminating Condition:

OK | Mégse | Alkalmaz | Súgó

Replication Length Base Time Units	32 Minutes

Display 4.13. The Run Setup Replication Parameters

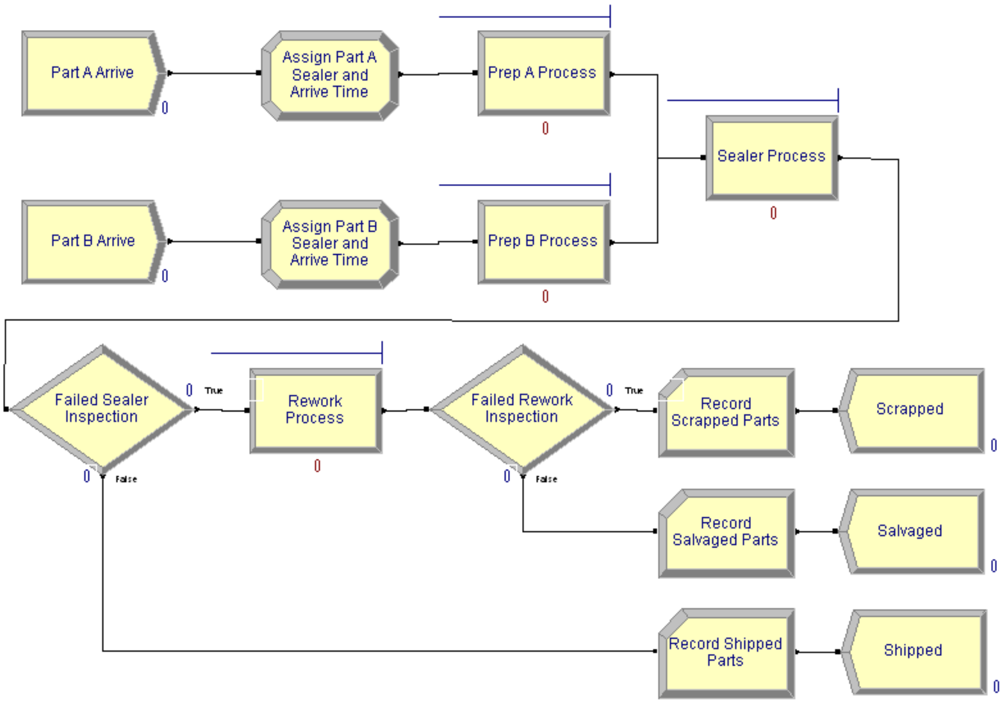
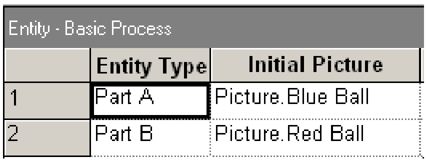


Figure 4.3. The Final Model 4-1

Now click on the Initial Picture cell for Part A, and use the list to select a different picture. We've chosen the blue ball for *Part A* and the red ball for *Part B*, as shown in *Display 4.14*. This will allow us to distinguish easily between the two parts in the animation. If you're interested in seeing what these icons look like, you can select *Edit* > *Entity Pictures* from the menu bar at the top of your screen to open the Entity Picture Placement window, which will allow you to see the icons currently available down the left column. We'll show you later how to use this feature in more detail.

Your final model should look something like *Figure 4.3*.



Entity - Basic Process

	Entity Type	Initial Picture
1	Part A	Picture.Blue Ball
2	Part B	Picture.Red Ball

Initial Picture (Part A) Initial Picture (Part B)	Picture.Blue Ball Picture.Red Ball

Display 4.14. The **Entity** data module

4.1.3 Running the Model

Before running your model, you might want to check it for errors. You can do this by clicking the *Check* button (☑) on the Run Interaction toolbar, the *Run* > *Check Model* command, or the F4 key on the keyboard. With a little luck, the response will be a small window with the message "No errors or warnings in model." If you have no luck at all, an error window will open with a message describing the error. If this occurs, you might want to select the Find option, if the button is enabled. This feature attempts to point you to where Arena thinks the error might be. We suggest you intentionally insert an error into your model and try these features. As you build more complex models, you might just find yourself using these features quite often.

If your model check results in no errors, you're now ready to run the simulation. There are four ways to run a simulation, but we'll only talk about three of them here. The first way is to run the simulation with the animation. Use the *Go* button (▶) on the Standard toolbar, the *Run* > *Go* command, or the FS key. If you've not yet checked your model or if you've made a change since the last check, Arena will first check the model, then initialize the model with your data, and finally run it. You'll notice during the run that Arena hides some of the graphics so that your focus is on the animation. Don't worry, though. They'll return when you end the run (or you can check to see that they're still there by using the *View* > *Layers* command).

If you leave the status bar active (at the bottom of the screen), you can tell what Arena is doing. Toward the right of this bar are three pieces of information: the replication number, the current simulation time, and the simulation status.

After the simulation starts to run, you may want to speed up or slow down the animation. You can do this while the model is running by pressing the "<" key to slow it down or the ">" key to speed it up. If you press one of these keys, the current Animation Speed Factor is displayed at the far left of the status bar. You can also increase or decrease the Animation Speed Factor from the Run Speed tab of the Run Setup dialog box. This option can also be used to enter an exact speed factor.

During the simulation run, you can also pause the simulation using the *Pause* button (⏸) on the Run toolbar, *Run*>*Pause*, or the *Esc* key. This temporarily suspends the simulation, and the message "User interrupted" will appear on the status bar.

While you're in Pause mode, you might want to double-click on one of the entities that is visible in the animation. An Entity Summary dialog box lists the values of each of the entity's attributes. This can be a very useful feature when trying to debug a model. You can also use the Step button () on the Run toolbar to move entities through the system one step at a time. You can continue the simulation run "normally" at any time with the Go button.

This method of running a simulation provides the greatest amount of information, but it can take a long time to finish. In this case, the time required to complete the run depends on the Animation Speed Factor. You can skip ahead in time by Pausing and then selecting the *Fast-Forward button* (⏩) on the Run toolbar, or *Run*>*Fast-Forward*. This will cause the simulation to run at a much faster speed by not updating the animation graphics. At any time during the run, you can Pause and return to the animation mode, or you can Zoom In (+), Zoom Out (-), or move about in the simulation window (arrow keys or scroll bars).

Using Fast-Forward will run the simulation in much less time, but if you're only interested in the numerical simulation results, you might want to disable the animation (computation and graphics update) altogether. You do this with the *Run*>*Run Control*>*Batch Run* (*No Animation*) option.

The *Run*>*Run Control* option also allows you to configure a number of other runtime options. For now, select the *Batch Run* (*No Animation*) option. Note that if you revisit this option, there is a check to the left. Accept this option and click the *Run* button. Note how much faster the simulation runs. The only disadvantage is that you must terminate the run and reset the animation settings in order to get the animation back. If you have large models or long runs and you're interested only in the numerical results this is the option to choose since it is even faster than Fast-Forward.

While you're building a model, you should probably have most of the toolbars visible and accessible. However, when you're running a model, many of the toolbars simply consume space because they are not active during runtime. Arena recognizes this and will save your toolbar settings for each mode. To take advantage of this, pause during the run and remove the toolbars that you don't want to have active while the simulation is running. When you end the run, these toolbars will reappear.

4.1.4 Viewing the Results

If you selected the *Run* > *Go* menu option (or the *Go* button, ▶), you might have noticed that, in addition to the blue and red balls (our *Part A* and B entities) moving through the system, there are several animated counters being incremented during the simulation run. There is a single counter for each **Create**, **Process**, and **Dispose** module and two counters for each **Decide** module. The counters for the **Create**, **Dispose**, and **Decide** modules are incremented each time an entity exits the module. In the case of the **Process** modules, the counter is the number of entities that are currently at that module, including any entities in the queue waiting for the resource plus any entities currently in process. If you selected the *Run*>*Fast-Forward* menu option (or the *Fast-Forward* button ⏩), these counters (and the entities in the queues) will be updated at the end of the run and whenever you pause or change views of the model. The final numbers resulting from our simulation are shown in *Figure 4.4*.

If you run the model to completion, Arena will ask if you want to see the results. If you select *Yes*, you should get a window showing the Category Overview Report (the default report). When the first page of the report appears, you might find it strange that the message "No Summary Statistics Are Available" is displayed. The system summary statistics are entity and costing statistics, which we elected not to collect when we *cleared* the Entities selection on the Project Parameters tab of the Run Setup dialog box (see *Display 4.12*). At some point, you might want to change these selections and view the difference in the reports.

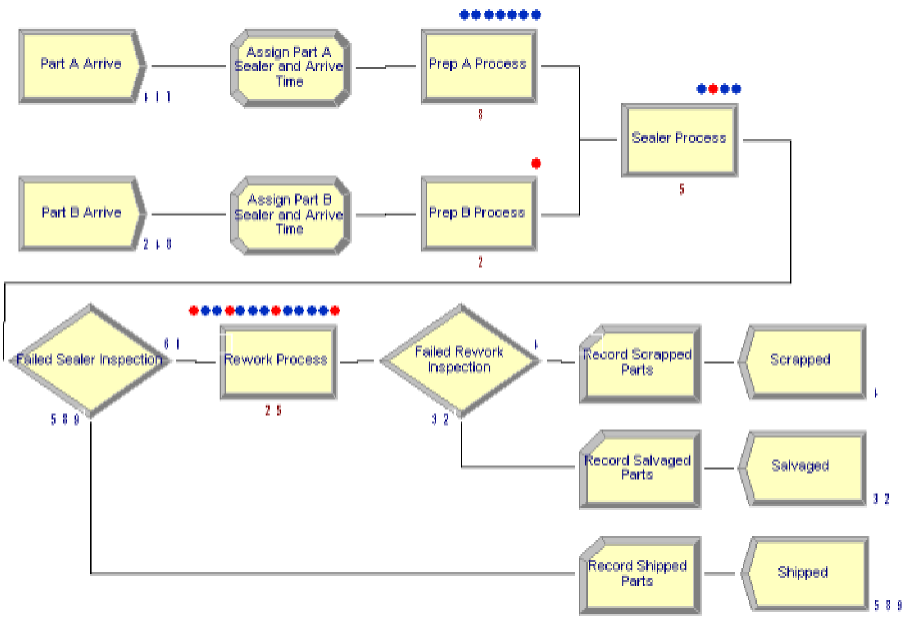


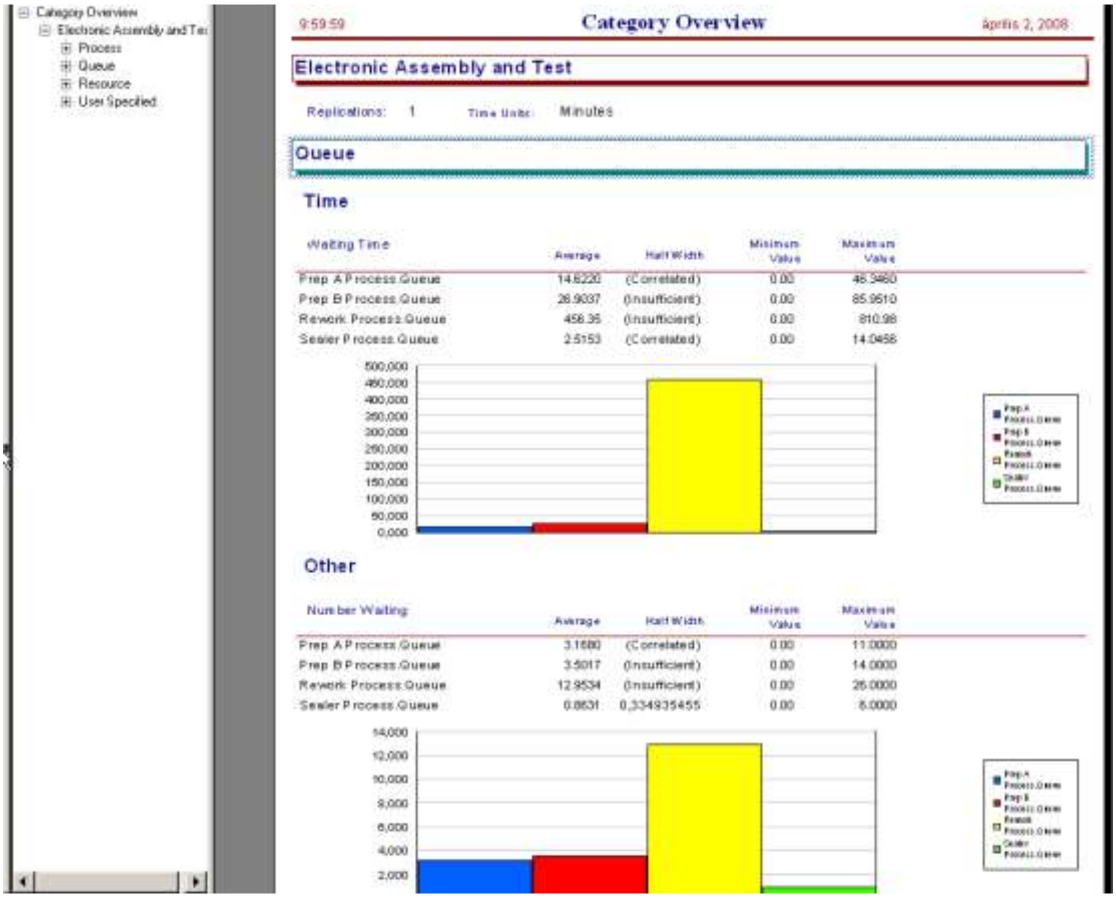
Figure 4.4. The Animated Results for Model 4-1

Recall that you can navigate through the report using the tree listing under the Preview tab at the left side of the report window or by using the arrow buttons () in the upper-left corner of the report window. This report will provide statistics by the categories selected in the Run Setup dialog box (Project Parameters tab, Statistics Collection area). For our model, you will find sections on Process, Queue, Resource, and User Specified. The User Specified section is there because we included **Record** modules in our model to collect statistics on the cycle times sorted by departure type.

As in Chapter 3, you will find three types of statistics in our report: *tally*, *time-persistent*, and *counter*. A fourth statistic (outputs) is available when multiple replications are made (we'll talk about them in Chapter 6, when we do multiple replications). The tally statistics here include several process times, queue times, and the interval times collected by our **Record** modules. The time-persistent statistics include number waiting in queue, resource usage, and resource utilization. Counters include accumulated time, number in, number out, and total number seized.

The tally and time-persistent statistics provide the average, 95% confidence-interval half width, and the minimum and maximum observed values. With the exception of the half width column, these entries are as described in Chapters 2 and 3, and should be self explanatory.

At the end of each replication, Arena attempts to calculate a 95% confidence-interval half width for the steady-state (long-run) expected value of each observed statistic, using a method called *batch means* (see Section 7.2). Arena first checks to see if sufficient data have been collected to test the critical statistical assumption (uncorrelated batches) required for the batch-means method. If not, the annotation "Insufficient" appears in the report and no confidence-interval half width is produced, as can be seen for several of the results. If there are enough data to test for uncorrelated batches, but the test is failed, the annotation "Correlated" appears and once again there's no half width, which is the case for several of the results. If there was enough data to perform the test for uncorrelated batches and the test were passed, the half width (the "plus-or-minus" amount) of a 95% confidence interval on the long-run (steady-state) expected value of the statistic would be given (this happens for a few of the results in this run). In this way, Arena refuses to report unreliable half-width values even though it could do so from a purely computational viewpoint. The details and importance of these tests are further discussed in Section 7.2.3.



9:59:59 Category Overview Aprile 2, 2008

Electronic Assembly and Test

Replications: 1 Time Units: Minutes

Queue

Time

Waiting Time	Average	Half Width	Minimum Value	Maximum Value
Prep A Process.Queue	14.8220	(Correlated)	0.00	46.3460
Prep B Process.Queue	26.9037	(Insufficient)	0.00	85.9510
Rework Process.Queue	456.35	(Insufficient)	0.00	810.98
Sealer Process.Queue	2.5153	(Correlated)	0.00	14.0456

Other

Number Waiting	Average	Half Width	Minimum Value	Maximum Value
Prep A Process.Queue	3.1680	(Correlated)	0.00	11.0000
Prep B Process.Queue	3.5017	(Insufficient)	0.00	14.0000
Rework Process.Queue	12.9534	(Insufficient)	0.00	26.0000
Sealer Process.Queue	0.8631	0,334935455	0.00	6.0000

Figure 4.5. The Queue Summary Report: Model 4-1

Trying to draw conclusions from this single short run could be misleading because we haven't yet addressed issues like run length, number of replications, or even whether long run steady-state results are appropriate (but we will, in Section 7.2). However, if you look at the results for the waiting time in queue and number waiting in queue (*Figure 4.5*), you see that the rework station is plagued by waiting times and queue lengths that are much longer than for the other stations (you can corroborate this imbalance in congestion by looking at the Process and Resources sections of the Category Overview report as well). This implies either that the rework area doesn't have enough capacity to handle its work, or there is a great deal of variability at this station. We'll address this issue in a minute in Section 4.2. (By the way, note in *Figure 4.5* that when there's more than one entry in a category, in this case more than one queue, the Category Overview report gives you a graphic for the averages, which is actually color-coded even though you probably can't see it in this antediluvian monochromatic book.)

4.2 Model 4-2: The Enhanced Electronic Assembly and Test System

Having constructed and run our model, the next activity would be to verify that the "code" (Arena file) is free of bugs and also to validate that the conceptual model really represents the system being studied. For this example, that's fairly easy. We can examine the logic constructs we selected from the modules we used and compare them to the problem definition. With much larger and more complex systems, this can become a challenging task. An animation is often useful during the verification and validation phases because it allows you to view the entire system being modeled as it operates. If you ran the model we developed and viewed its animation, you should have noted that it appeared to operate quite similarly to the way we described the system. If verification can be very difficult, complete validation (the next activity) can sometimes be almost impossible. That's because validation implies that the simulation is behaving just like the real-world system, which may not even exist, so you can't tell. And even if the system does exist, you have to have output performance data from it, as well as convince yourself and other nonbelievers that your model can really capture and predict the events of the real system. We'll discuss both of these activities in much more detail later.

For now, let's assume that as part of this effort you showed the model and its accompanying results to the production manager. Her first observation was that you didn't have a complete definition of how the system works. Whoever developed the problem definition looked only at the operation of the first shift. But this system actually operates two shifts a day, and on the second shift, there are two operators assigned to the rework operation. This would explain our earlier observation when we thought the rework operation might not have enough capacity. The production manager also noted that she has a failure problem at the sealer operation. Periodically, the sealer machine breaks down. Engineering looked at the problem some time ago and collected data to determine the effect on the sealer operation. They felt that these failures did not merit any significant effort to correct the problem because they didn't feel that the sealer operation was a bottleneck. However, they did log their observations, which are still available. Let's assume that the mean uptime (from the end of one failure to the onset of the next failure) was found to be 120 minutes and that the distribution of the uptime is exponential (which, by the way, is often used as a realistic model for uptimes if failures occur randomly at a uniform rate over time). The time to repair also follows an exponential distribution with a mean of 4 minutes.

In addition, the production manager indicated that she was considering purchasing special racks to store the waiting parts in the rework area. These racks can hold ten assemblies each, and she would like to know how many racks to buy. Our next step is to modify the model to include these three new aspects, which will allow us to use some additional Arena features.

In order to incorporate these changes into our model, we'll need to introduce several new concepts. Changing from a one- to a two-shift operation is fairly easy. In Model 4-1, we set our run length to four 8-hour shifts and made no attempt to keep track of the day/ shift during the run. We just assumed that the system conditions at the end of a shift were the same at the start of the next shift and ignored the intervening time. Now we need to model explicitly the change in shifts, because we have only one operator in the first shift and two in the second shift for the rework process.

We'll add this to our model by introducing a Resource Schedule for the rework resource, which will automatically change the number of rework resources throughout the run by adjusting the resource capacity. While we're making this change, we'll also increase the run length so that we simulate more than just two days of a two-shift operation. We'll model the sealer failures using a Resource Failure, Which allows us to change the available capacity of the resource (much like the Resource Schedule), but has additional features specifically designed for representing equipment failures. Finally, we'll use the Frequencies statistic to obtain the type of information we need to determine the number of racks that should be purchased.

4.2.1 Expanding Resource Representation: Schedules and States

So far we've modeled each of our resources (prep area, sealer, and rework) as a single resource with a fixed capacity of 1. You might recall that we defaulted all of this information in the **Resource** module. To model the additional rework operator, we could simply change the capacity of the rework resource to 2, but this would mean that we would always have two operators available. What we need to do is to schedule one rework operator for the first shift (assume each shift is 8 hours) and two rework operators for the second shift. Arena has a built-in construct to model this, called a Schedule, that allows you to vary the capacity of a resource over time according to a fixed pattern. A resource Schedule is defined by a sequence of time-dependent resource capacity changes.

We also need to capture in our model the periodic random breakdowns (or failures) of the sealer machine. This could be modeled using a Schedule, which would define an available resource capacity of 1 for the uptimes and a capacity of 0 for the time to repair. However, there is a built-in construct designed specifically to model failures. First, let's introduce the concept of Resource States.

Arena automatically has four Resource States: *Idle, Busy*, *Inactive*, and *Failed*. For statistical reporting, Arena keeps track of the time the resource was in each of the four states. The resource is said to be Idle if no entity has seized it. As soon as an entity seizes the resource, the state is changed to *Busy*. The state will be changed to *Inactive* if Arena has made the resource unavailable for allocation; this could be accomplished with a Schedule's changing the capacity to 0. The state will be changed to Failed if Arena has placed the resource in the Failed state, which also implies that it's unavailable for allocation.

When a failure occurs, Arena causes the entire resource to become unavailable. If the capacity is 2, for example, both units of the resource will be placed in the Failed state during the repair time.

4.2.2 Resource Schedules

Before we add our resource schedule for the rework operation, let's first define our new 16-hour day. We do this in the Replication Parameters tab of the *Run* > *Setup* option, by changing the Hours Per Day from 24 to 16 (ignore the warning about Calendars that you might get here). While we're in this dialog box, let's also change our Time Units for the Replication Length to Days and the Replication Length itself to 10 days.

You can start the definition of a resource schedule in either the **Resource** or **Schedule** data module. We'll start with the **Resource** data module. When you click on this module from the Basic Process panel, the information on the current resources in the model will be displayed in the spreadsheet view of the model window along the bottom of your screen. Now click in the Type column for the Rework resource row and select Based on Schedule from the list. When you select this option, Arena will add two new columns to the spreadsheet view-Schedule Name and Schedule Rule. Note that the Capacity cell for the Rework resource has been dimmed because the capacity will instead be based on a schedule. In addition, the cells for two new columns are also dimmed for the other three resources because they still have a fixed capacity. Next you should enter the schedule name (e.g., Rework Schedule) in the Schedule Name cell for the Rework resource.

Finally, you need to select the Schedule Rule, which can affect the specific timing of when the capacity defined by the schedule will actually change. There are three options for the Schedule Rule: Wait (the default), Ignore, and Preempt. If a capacity decrease of x units is scheduled to occur and at least x units of the resource are idle, all three options immediately cause x units of the resource unit(s) to become *Inactive*. But if fewer than x units of the resource are idle, each Schedule Rule responds differently: (The three options are illustrated in *Figure 4.6*.)

> The *Ignore* option immediately decreases the resource capacity, ignoring the fact that the resource is currently allocated to an entity, but "work" on the in-service entities continue unabated. When units of the resource are released by the entity, these units are placed in

the *Inactive* state. However, if the resource capacity is increased again (that is, the scheduled time at the lower capacity expires) before the entities release the units of the resource, it's as if the schedule change never occurred. The net effect is that the time for which the resource capacity is scheduled to be reduced may be shortened with this option.

The *Wait* option, as its name implies, will wait until the in-process entities release their units of the resource before starting the actual capacity decrease. Thus the reduced capacity time will always be of the specified duration, but the time between these reductions may increase.

The *Preempt* option attempts to preempt the last unit of the resource seized by taking it away from the controlling entity. If the preempt is successful and a single unit of capacity is enough, then the capacity reduction starts immediately. The preempted entity is held internally by Arena until the resource becomes available, at which time the entity will be reallocated the resource and continue with its remaining processing time. This provides an accurate way to model schedules and failures because, in many cases, the processing of a part is suspended at the end of a shift or when the resource fails. If the preempt is unsuccessful or if more than one unit is needed, then the Ignore rule will be used for any remaining capacity.

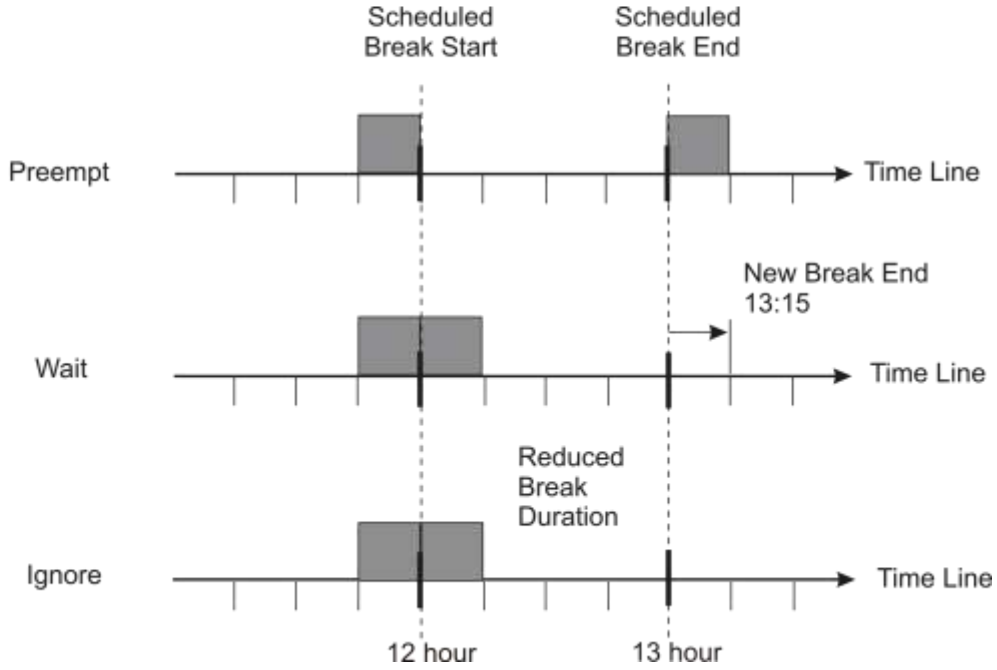
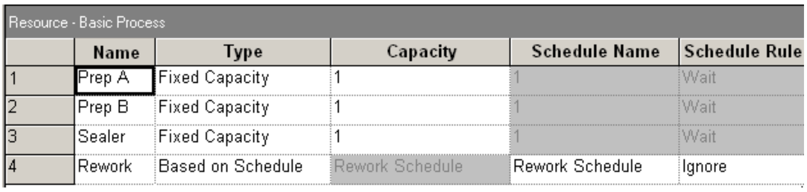


Figure 4.6. The Ignore, Preempt, and Wait Options

So when should you use each of the rules? Generally, we recommend that you examine closely the actual process and select the option that best describes what actually ' occurs at the time of a downward schedule change or resource failure. If the resource under consideration is the bottleneck for the system, your choice could significantly affect the results obtained. But sometimes it isn't clear what to do, and, while there are no strict guidelines, a few rules of thumb may be of help. First, if the duration of the scheduled decrease in capacity is very large compared to the processing time, the Ignore option may be an adequate representation. If the time between capacity decreases is large compared to the duration of the decrease, the Wait option could be considered. For this model, we've selected the Ignore option because, in most cases, an operator will finish his task before leaving and that additional work time is seldom considered.

The final spreadsheet view of the first four columns is shown in *Display 4.15* (there are actually additional columns to the right, which we don't show here). There is also a dialog form for entering these data, which can be opened by right-clicking on the rework cell in the Name column and selecting the Edit via Dialog option.

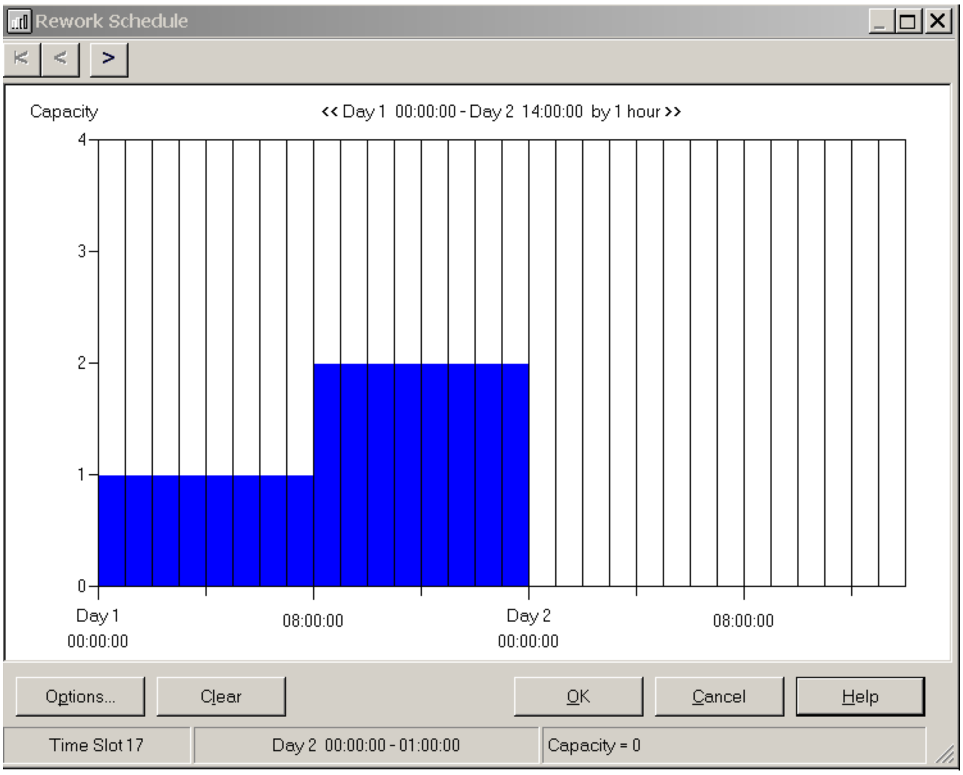


Resource - Basic Process

	Name	Type	Capacity	Schedule Name	Schedule Rule
1	Prep A	Fixed Capacity	1	1	Wait
2	Prep B	Fixed Capacity	1	1	Wait
3	Sealer	Fixed Capacity	1	1	Wait
4	Rework	Based on Schedule	Rework Schedule	Rework Schedule	Ignore

Rework Resource	
Type	Based on Schedule
Schedule Name	Rework Schedule
Schedule Rule	Ignore

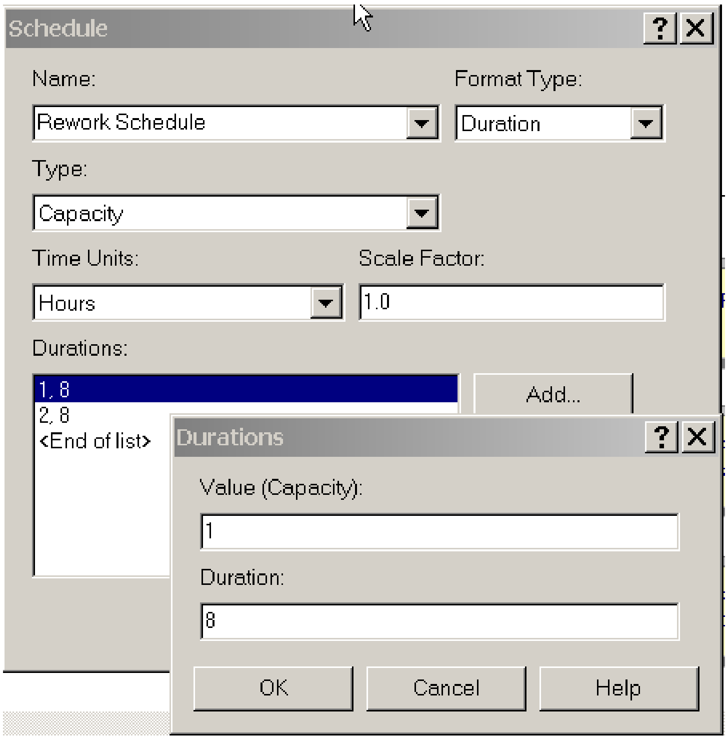
Display 4.15. The **Resource** data module: Selecting a Resource Schedule



Time Slot 17 | Day 2 00:00:00 - 01:00:00 | Capacity = 0

Figure 4.7. The Graphical Schedule Editor: The Rework Schedule

Now that you've named the schedule and indicated the schedule rule, you must define the actual schedule the resource should follow. One way to do this is by clicking on the **Schedule** data module and entering the schedule information in the spreadsheet view. A row has already been added that contains our newly defined Rework Schedule. Clicking in the Durations column will open the Graphical Schedule Editor, a graphical interface for entering the schedule data. The horizontal axis is the calendar or simulation time. (Note that a day is defined as 16 hours based on our day definition in the Run Setup dialog box.) The vertical axis is the capacity of the resource. You enter data by clicking on the *x-y* location that represents day one, hour one, and a capacity value of one. This will cause a solid blue bar to appear that represents the desired capacity during this hour; in this case, one. You can complete the data entry by repeatedly clicking or by clicking, holding, and then dragging the bar over the first eight hours. Complete the data schedule by entering a capacity of 2 for hours nine through 16. It's not necessary to add the data for Day 2, as the data entered for Day 1 will be repeated automatically for the rest of the simulation run. The complete schedule is shown in *Figure 4.7*. You might note that we used the Options button to reduce our maximum vertical axis (capacity) value from ten to four; other things can be changed in the Options dialog box, like how long a time slot lasts, the number of time slots, and whether the schedule repeats from the beginning or remains forevermore at a fixed-capacity level.


Schedule

Name: Rework Schedule

Format Type: Duration

Type: Capacity

Time Units: Hours

Scale Factor: 1.0

Durations:
1, 8
2, 8
<End of list>

Durations

Value (Capacity): 1

Duration: 8

Name	Rework Schedule
Value Duration	1 8
Value Duration	2 8

Display 4.16. The **Schedule** data module Dialog Box

You can also enter these data manually by right-clicking in the Durations column in the **Schedule** module spreadsheet view and selecting the Edit via Dialog option. If you select this option, first enter the schedule name, then click the Add button to open the Durations window. Here you define the (Capacity, Duration) pairs that will make up the schedule. In this case, our two pairs are 1, 8 and 2, 8 (*Display 4.16*). This implies that the capacity will be set to 1 for the first 480 minutes, then 2 for the next 480 minutes. This schedule will then repeat for the duration of the simulation run. You may have as many (Capacity, Duration) pairs as are required to model your system accurately. For example, you might want to include operator breaks and the lunch period in your schedule. There is one caution, or feature,3 of which you should be aware. If, for any pair, no entry is made for the Duration, it will default to infinity. This will cause the resource to have that capacity for the entire remaining duration of the simulation run. As long as there are positive entries for all durations, the schedule will repeat for the entire simulation run.

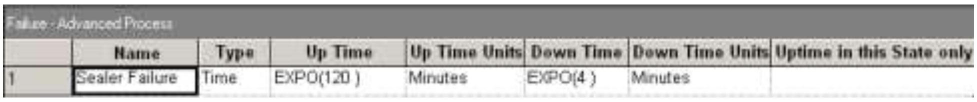
If you use the Graphical Schedule Editor to create the schedule and then open the dialog box, you will find that the data have been entered automatically. Note that you cannot use the

Graphical Schedule Editor if you have any time durations that are not integer, or if any entries require an Expression (for example, a time duration that's a draw from a random variable).

4.2.3 Resource Failures

Schedules are intended to model the planned variation in the availability of resources due to shift changes, breaks, vacations, meetings, etc. Failures are primarily intended to model random events that cause the resource to become unavailable. You can start your failure definition in either the **Resource** or the **Failure** data module. The **Failure** data module can be found in the Advanced Process panel (which you might need to attach at this point via the 📂 button or *File* > *Template Panel* > *Attach* if it's not already accessible in the Project Bar). Since we started with the **Resource** module in developing our schedule, let's start with the **Failure** data module for our Sealer failure.

If you need to make the Advanced Process panel visible in the Project Bar, click on its name, and then click on the **Failure** data module. The spreadsheet view for this module will show no current entries. Double-click where indicated to add a new row. Next select the default name in the Name column and replace it with a meaningful failure name, like *Sealer Failure*. Then select whether the failure is Count-based or Time-based using the list in the Type cell. A Count-based failure causes the resource to fail after the specified number of entities have used the resource. This count may be a fixed number or may be generated from any expression. Count-based activities are fairly common in industrial models. For example, tool replacement, cleaning, and machine adjustment are typically based on the number of parts that have been processed rather than on elapsed time. Although these may not normally be viewed as "failures," they do occur on a periodic basis and prevent the resource from producing parts. On the other hand, we frequently model failures as Time-based because that's the way we've collected the failure data. In our model, the problem calls for a Time-based failure. So click on the Type cell and select the Time option. When you do this, the column further to the right in the spreadsheet will change to reflect the different data requirements between the two options. Our Up Time and Down Time entries are exponential distributions with means of 120 minutes and 4 minutes, respectively. We also need to change the Up Time Units and Down Time Units from Hours to Minutes.



Failure - Advanced Process

	Name	Type	Up Time	Up Time Units	Down Time	Down Time Units	Uptime in this State only
1	Sealer Failure	Time	EXPO(120)	Minutes	EXPO(4)	Minutes	

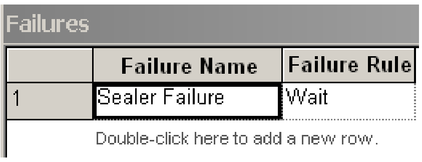
Name	Sealer Failure
Type	Time
Up Time	EXPO (120)
Up Time Units	Minutes
Down Time	EXPO (4)
Down Time Units	Minutes

Display 4.17. The Sealer Failure Spreadsheet View

The last field, Uptime in this State Only, allows us to define the state of the resource that should be considered as "counting" for the uptimes. If this field is defaulted, then all states are considered. Use of this feature is very dependent on how your data were collected and the calendar timing of your model. Most failure data are simply logged data; for example, only the time of the failure is logged. If this is the case, then holidays, lunch breaks, and idle time are included in the time between failures, and you should default this field. Only if your time between failures can be linked directly to a specific state should this option be chosen. Many times equipment vendors will supply failure data based on actual operating hours; in this case, you would want to select this option and specify the *Busy* state. Note that if you select this option you must also define the *Busy* state using the **StateSet** data module found in the Advanced Process panel.

The final spreadsheet view for our Sealer Failure is shown in *Display 4.17*.

Having completed the definition of our Sealer Failure, we now need to attach it to the Sealer resource. Open the **Resource** data module (back in the Basic Process panel) and click in the Failures column for the Sealer resource row. This will open another window with the Failures spreadsheet view. Double-click to add a new row and, in the Failure Name cell, select Sealer Failure from the list. We must also select the Failure Rule-Ignore, Wait, or Preempt. These options are the same as for schedules, and you respond in an identical manner. Returning to our rules of thumb for choosing the Failure Rule option, because our expected uptime (120 minutes) is large compared to our failure duration (4 minutes), we'll use the Wait option. The final spreadsheet view is shown in *Display 4.18*. If you have multiple resources with the same failure profile, they can all reference the same Failure Name. Although they will all use the same failure profile, they will each get their own independent random samples during the simulation run.



Failures

	Failure Name	Failure Rule
1	Sealer Failure	Wait

Sealer Resource Failure	
Failure Name	Sealer Failure
Failure Rule	Wait

Display 4.18. The Sealer **Resource** data module: Failures Spreadsheet View

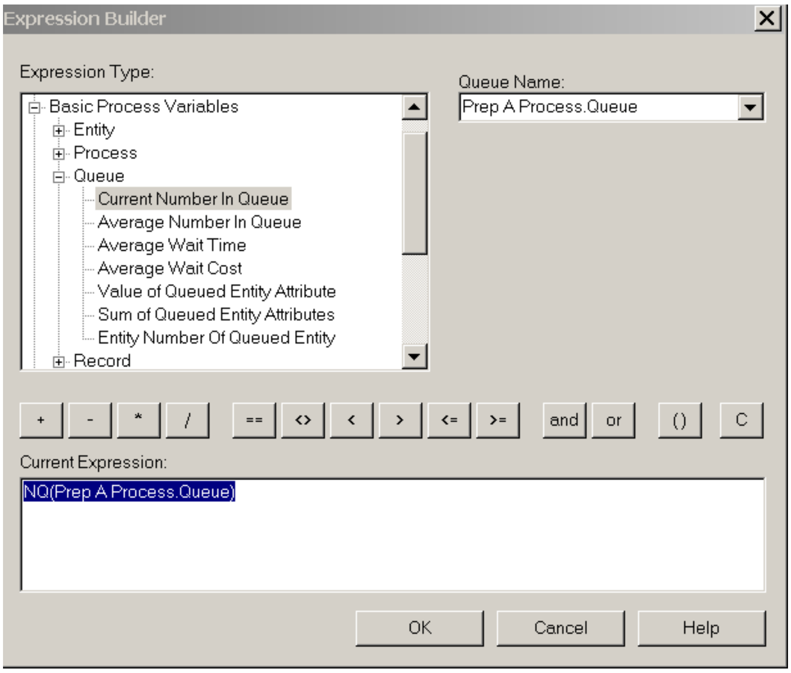
4.2.4 Frequencies

Frequencies are used to record the time-persistent occurrence frequency of an Arena variable, expression, or resource state. We can use the *Frequencies* statistic type to obtain the information we need to determine the number of racks required at the Rework area. We're interested in the status of the rework queue-specifically, how many racks of 10 should we buy to ensure that we have sufficient storage almost all of the time. In this case, we're interested in the amount of time the number in queue was 0 (no racks needed), greater than 0 but no more than 10 (one rack needed), greater than 10 but no more than 20 (two racks needed), etc.

Frequency statistics are entered using the **Statistic** data module, which can be found in the Advanced Process panel. Clicking on this data module will open the spreadsheet view, which is initially empty; double-click to add a new row. We'll first enter the name as *Rework Queue Stats*. Next, select *Frequency* in the Type list and default on the Value entry for the Frequency Type. You might note that when we selected the Type, the Report Label cell was automatically given the same name as the Name cell, Rework Queue Stats, which we'll accept to label this output in the reports.

We now need to develop an Expression that represents the number in the rework queue. To request this information, we need to know the Arena variable name for the number in queue, NQ. We can get the name of the queue, *Rework Process.Queue,* from the **Queue** data module found in the Basic Process panel. Thus, the Expression we want to enter is *NQ*(*Rework Process.Queue*). At this point, you should be asking, "Did you expect me to know all that?" To an experienced (and thus old) SIMAN user, this is obvious. However, it is clearly not obvious to the new user. Fortunately, Arena provides an easy way to develop these types of expressions without the need to know all the secret words (e.g., NQ). Place your pointer in the blank Expression cell, right-click, and select Build Expression. This will open the Arena Expression Builder window shown in *Display 4.19*. Under the Expression Type category Basic Process Variables, you'll find the sub-category Queue. Click on the + sign to expand the options and then select Current Number In Queue. When you do this, two things will happen: the Queue Name field will appear at the right and the Current Expression at the bottom will be filled in using the queue name shown. In our case, the Current Expression was *NQ*(*Prep A Process.Queue*), which is not yet what we want (it's the wrong queue). Now use the drop-down list arrow for Queue Name to view and select the *Rework Process.Queue*. Now when you click OK, that expression will automatically be entered into the field from which the Expression Builder was opened.

You can right-click on any field in which an expression can be entered to open the Expression Builder. For example, we could have used the Expression Builder to find the expression for the current simulation time (TNOW). You can also build complex expressions by using the function buttons in the Expression Builder, as well as typing things (the old-fashioned way, assuming that you know what to type) into the Current Expression field at the bottom.



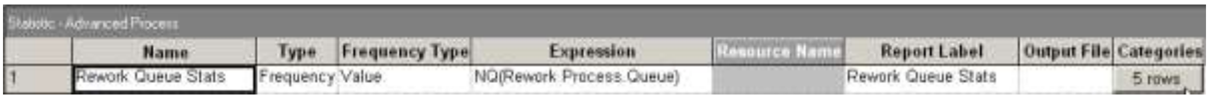
Queue Name: Prep A Process.Queue

Current Expression: NQ(Prep A Process.Queue)

Display 4.19. The Expression Builder Dialog Box

The spreadsheet view to this point is shown in *Display 4.20*.

The last step in setting up the rework queue statistics is to build the categories that define how we want the values displayed, done in the Categories column at the far right (click the "0 Rows" button to open a spreadsheet to which you add a row for each category). *Display 4.21* shows the entries for the first three categories. The first entry is for a queue size of a Constant 0 (in which case, we'd need 0 racks); the next entry is for one rack, two racks, etc. For now, we'll only request this information for up to four racks. If the queue ever exceeds 40 parts, Arena will create an out-of range category on the output report. In the case of a Range, note that Value is not included in the range, but High Value is. Thus, for instance, Value = 10 and High Value = 20 defines a range of numbers (10, 20]; that is, (strictly) greater than 10 and less than or equal to 20.

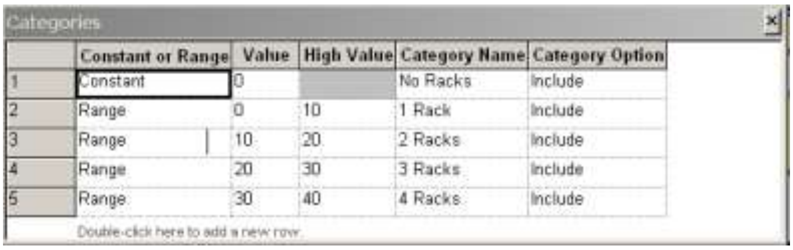


Statistic - Advanced Process

	Name	Type	Frequency Type	Expression	Resource Name	Report Label	Output File	Categories
1	Rework Queue Stats	Frequency	Value	NQ(Rework Process.Queue)		Rework Queue Stats		5 rows

Name Type Frequency Type Expression	Rework Queue Stats Frequency Value NQ (Rework Process.Queue)

Display 4.20. The **Statistic** data module



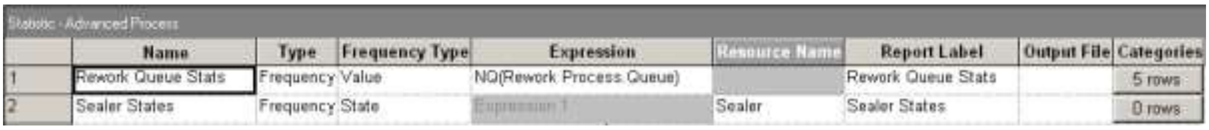
Categories

	Constant or Range	Value	High Value	Category Name	Category Option
1	Constant	0		No Racks	Include
2	Range	0	10	1 Rack	Include
3	Range	10	20	2 Racks	Include
4	Range	20	30	3 Racks	Include
5	Range	30	40	4 Racks	Include

Double-click here to add a new row.

Constant or Range Value Category Name	Constant 0 0 Racks
Constant or Range Value High Value Category Name	Range 0 10 1 Rack
Constant or Range Value High Value Category Name	Range 10 20 2 Racks

Display 4.21. Categories for Rework Queue Stats Frequency Statistic



Statistic - Advanced Process

	Name	Type	Frequency Type	Expression	Resource Name	Report Label	Output File	Categories
1	Rework Queue Stats	Frequency	Value	NQ(Rework Process.Queue)		Rework Queue Stats		5 rows
2	Sealer States	Frequency	State	Expression 1	Sealer	Sealer States		0 rows

Name Type Frequency Type Resource Name	Sealer States Frequency State Sealer

Display 4.22. The **Statistic** data module for the Sealer States

Before leaving the **Statistic** data module, we might also want to request additional information on the Sealer resource. If we run our current model, information on the utilization of the Sealer resource will be included in our reports as before. However, it will not specifically report the amount of time the resource is in a failed state. We can request this statistic by adding a new row in our **Statistic** data module as shown in *Display 4.22*. For this statistic, we enter the Name and Type, Sealer States and Frequency, and select State for the Frequency Type. Finally, we select the Sealer resource from the list in the Resource Name cell. This will give us statistics based on all the states of the Sealer resource-*Busy*, *Idle*, and *Failed*.

Before you run this model, we recommend that you check the *Run* > *Run Control* > *Batch Run* (*No Animation*) option, which will greatly reduce the amount of time required to run the model. Although slower, an alternative is to select *Run* > *Fast-Forward* (). This would also be a good time to save your work. Note that you can still pause the run at any time to determine how far you've progressed.

4.2.5 Results of Model 4-2

Table 4.1 gives some selected results from the Reports for this model (rightmost column), as well as for Model 4-1 for comparison. We rounded everything to two decimals except for the two kinds of utilization results, which we give to four decimals (in order to make a particular point about them).

Table 4.1

Selected Results from Model 4-1 and Model 4-2

Result	Model4-1	Model4-2
Average Waiting Time in Queue		
Prep A	14.62	19.20
Prep B	29.90	51.42
Sealer	2.52	7.83
Rework	456.35	116.25
Average Number Waiting in Queue		
Prep A	3.17	3.89
Prep B	3.50	6.89
Sealer	0.86	2.63
Rework	12.95	3.63
Average Time in System		
Shipped Parts	28.76	47.36
Salvaged Parts	503.85	203.83
Scrapped Parts	737.19	211.96
Instantaneous Utilization of Resource		
Prep A	0.9038	0.8869
Prep B	0.7575	0.8011
Sealer	0.8595	0.8425
Rework	0.9495	0.8641
Scheduled Utilization of Resource		
Prep A	0.9038	0.8869
Prep B	0.7575	0.8011
Sealer	0.8595	0.8425
Rework	0.9495	0.8567

The results from this model differ from those produced by Model 4-1 for several reasons. We're now running the simulation for ten 16-hour days (so 160 hours) rather than the 32 hours we ran Model 4-1. And, of course, we have different modeling assumptions about the Sealer and Rework Resources. Finally, all of these facts combine to cause the underlying random-number stream to be used differently (more about this in Chapter 12).

Going from Model 4-1 to Model 4-2 didn't involve any changes in the Prep A or Prep B parts of the model, so the differences we see there are due just to the differences in run length or random bounce. The difference for the Prep B queue results are pretty noticeable, so either this area becomes more congested as time goes by, or else the results are subject to a lot of uncertainty-we don't know which (all the more reason to do statistical analysis of the output, which we're not doing here).

For the Sealer, the queue statistics (both average waiting time and average length) display considerably more congestion for Model 4-2. This makes sense, since we added the Failures to the Sealer in this model, taking it out of action now and then, during which time the queue builds up. The utilization statistics for the Sealer are not much different across the two models, though, since when it's in the *Failed* state the Sealer is not available, so these periods don't count "against" the Sealer's utilizations.

Unlike the Sealer, the Rework operation seems to be going much more smoothly in Model 4-2. Of course, the reason for this is that we added a second unit of the Rework resource during the second eight-hour shift of each 16-hour day. This increases the capacity of the Rework operation by 50% over time, so that it now has a time-average capacity of 1.5 rather than 1. And accordingly, the utilization statistics of the Rework operation seem to have decreased substantially (more on these different kinds of utilizations below).

Looking at the average time in system of the three kinds of exiting parts, it seems clear that the changes at the Sealer and Rework operations are having their effect. All parts have to endure the now-slower Sealer operation, accounting for the increase in the average time in system of shipped parts. Salvaged and scrapped parts, however, enjoy a much faster trip through the Rework operation (maybe making them feel better after failing inspection), with the net effect being that their average time in system seems to decrease quite a lot.

Now we need to discuss a rather fine point about utilizations. For each Resource, Arena reports two utilization statistics, called Instantaneous Utilization and Scheduled Utilization.

> *Instantaneous Utilization* is calculated by computing the utilization at a particular instant in time (that is, [number of Resource units busy]/[number of Resource units scheduled] at that point in time), and then calculating a time-weighted average of this over the whole run to produce the value shown in the reports. If there are no units of the Resource scheduled at a particular instant in time, the above ratio is simply defined to be zero, and it is counted as a zero in the time-weighted average reported. This can be a useful statistic for tracking utilization over time (for example, a utilization plot). However, it should be used with caution as it can provide confusing results under certain conditions (more on this shortly). If you like math (and even if you don't), we can express all this as follows. Let $B(t)$ be the number of units of the Resource that are busy at time t, and let $M(t)$ be the number of units of that Resource that are scheduled (busy or not) at time t. Let $U(t) = B(t)/M(t)$ whenever $M(t) > 0$, and simply define $U(t)$ to be 0 if $M(t) = 0$. If the run is from time 0 to time T, then the reported Instantaneous Utilization is
>
> $$\int_0^T U(t)dt/T = \frac{1}{T}\int_0^T \frac{B(t)}{M(t)}dt,$$
>
> which is just the time average of the $U(t)$ function as defined above. It's important to note that in reporting this, Arena accounts for time periods when there's no capacity scheduled as time periods with a utilization of zero.
>
> *Scheduled Utilization* is the time-average number of units of the Resource that are busy (taken over the whole run), divided by the time-average number of units of the Resource that are scheduled (over the whole run). In the above notation, the reported Scheduled Utilization is

$$\frac{\int_0^T B(t)dt/T}{\int_0^T M(t)dt/T} = \frac{\int_0^T B(t)dt}{\int_0^T M(t)dt}$$

There's no problem with dividing by zero here as long as the Resource was ever scheduled at all to have units available (the only situation that makes sense if the Resource is even present in the model).

Under different conditions, each of these reported utilizations can provide useful information. If the capacity of a Resource stays constant over the entire simulation run (the Resource is never in an *Inactive* or *Failed* state), the two reported utilizations will be the same (you're asked to prove this in Exercise 4-19, and you can verify from *Table 4.1* that this is so in Model 4-2 for the Resources that have fixed capacity). If a Schedule is attached to the Resource and the scheduled capacities are either zero or a single positive constant (for example, 1), the reported Instantaneous Utilization tells you how busy the Resource was over the entire run (counting zero-capacity periods as zero-utilization periods); a plot of $U(t)$ as defined above would tell you how closely your Schedule tracks what's being asked of the Resource, so it could be useful to investigate staffing plans. The reported Scheduled Utilization tells you how busy the Resource was over the time that it was available at all (in other words, capacity was non-zero).

For example, consider the case where a Resource is scheduled to be available 2/3 of the time with a capacity of 1, and unavailable the other 1/3 of the time (capacity of 0). Suppose further that during the time the Resource was available, it was busy half the time. For this example, the reported Instantaneous Utilization would be 0.3333 and the reported Scheduled Utilization would be 0.5000. This tells you that over the entire run the Resource was utilized 1/3 of the time, but during the time that it was scheduled to be available, it was utilized 1/2 the time. In this case, the reported Instantaneous Utilization is less than or equal to the reported Scheduled Utilization.

We need to modify the above with an even finer point. If you select a Schedule Rule other then Wait, it is possible to get a reported Scheduled Utilization greater than 1. For example, if you select the Ignore option and the Resource is allocated when it is scheduled to become unavailable (capacity of 0), the Resource capacity is decreased to zero, but the Resource stays allocated until it is released. Say you have a ten-hour run time and the Resource is scheduled to be available for only the first five hours. Let's assume that the Resource was allocated at time 0 to an activity with duration of 6 hours. At time 5, the capacity of the Resource would be reduced to 0, but the Resource would remain allocated until time 6. The reported Instantaneous Utilization would be 0.6 (it was utilized 6 of the 10 hours), while the reported Scheduled Utilization would be 1.2 (it was utilized 6 hours, but only scheduled to be available 5 hours).

A further (and, we promise, final) complication to the above arises when a Schedule is attached to the Resource and the scheduled capacities vary among different positive values over time (e.g., 1, 7, 4, etc.) rather than varying only between 0 and a single positive constant. If you have this situation, we strongly recommend that you not use Instantaneous Utilization even though it will still be reported. Depending on the capacities, the time durations, and the usage, this utilization can be less than, equal to, or greater than the Scheduled Utilization (you're asked to demonstrate this in Exercise 4-20). In this case, we suggest that you instead take advantage of the Frequencies statistic, which will give you detailed and precise information on how the Resource was utilized; see the Help topic "Resource Statistics: Instantaneous Utilization Vs. Scheduled Utilization" for more on this. So, for example, in Model 4-2 for the Rework Resource, we'd suggest that you use the Scheduled Utilization of 0.8567 in *Table 4.1* rather than the Instantaneous Utilization of 0.8641.

The new frequency statistics are not part of the normal Category Overview report. You must click on the Frequencies report in the Reports panel of the Project Bar. The results are given in *Figure 4.8*.

Rework Queue Stats	Number Obs	Average Time	Standard Percent	Restricted Percent
0 Racks	41	69.4722	29.67	29.67
1 Rack	52	119.96	64.98	64.98
2 Racks	12	42.8210	5.35	5.35

Sealer States	Number Obs	Average Time	Standard Percent	Restricted Percent
BUSY	697	11.6044	84.25	84.25
FAILED	68	4.1861	2,97	2.97
IDLE	640	1.9173	12.78	12,78

Figure 4.8. The Arena Frequencies Report: Model 4-2

The first section shows the statistics we requested to determine the number of racks required at the rework process. In this particular run, there were never more than 20 in the rework queue (that is, there are no data listed for three or four racks), and there were more than ten only 5.35% of the time. This might imply that you could get by with only one rack, or at most, two. The Sealer States statistics give the percentage of times the Sealer spent in the *Busy*, *Failed*, and *Idle* states.

One last note is worth mentioning about frequency statistics. For our results, the last two columns, Standard and Restricted Percent, have the same values. It is possible to exclude selective data resulting in differences between these columns. For example, if you exclude the *Failed* state for the sealer Frequency, the Standard Percent would remain the same, but the Restricted Percent column would have values only for *Busy* and *Idle*, which would sum to 100%.

4.3 Model 4-3: Enhancing the Animation

So far in this chapter we've simply accepted the default animation provided with the modules we used. Although this base animation is often sufficient for determining whether your model is working correctly, you might want the animation to look more like the real system before allowing decision makers to view the model. Making the animation more realistic is really very easy and seldom requires a lot of time. To a large extent, the amount of time you decide to invest depends on the level of detail you decide to add and the nature of the audience for your efforts. A general observation is that, for presentation purposes, the higher you go in an organization, the more time you should probably spend on the animation. You will also find that making the animation beautiful can be a lot of fun and can even become an obsession. So with that thought in mind, let's explore some of what can be done.

We'll modify Model 4-2 into what we'll call Model 4-3, and we'll start by looking at the existing animation. The current animation has three components: entities, queues, and variables. The entities we selected using the **Entity** data module can be seen when they travel from one module to another or when they're waiting in a queue. For each **Process** module we placed, Arena automatically added an animation queue, which displays waiting entities during the run. Variables were also placed automatically by Arena to represent the number of entities resident in or that have exited a module.

As suggested by how they came to exist in the model-added when you placed the module-the animation constructs are "attached" to the module in two respects. First, their names, or Identifiers, come from values in the module dialog box; you can't change them directly in the animation construct's dialog box. Second, if you move the module, its animation objects move with it; however, if you want the animation to stay where it is, just hold the Shift key when you move the module.

Sometimes it's helpful to "pull apart" the animation to a completely different area in the model window, away from the logic. If you do so, you might consider setting up some Named Views (see Section 3.4.13) to facilitate going back and forth. If you want to disconnect an animation construct completely from the module it originally accompanied, Cut it to the Clipboard and Paste it back into the model. It will retain all of its characteristics, but no longer will have any association with the module. An alternative method is to delete the animation construct that came with the module and add it back from scratch using the constructs from the **Animate toolbar**.

You might even want to leave some of the automatically placed animation constructs with the modules and just make copies of them for the separate animation. If you decide to do this, there are some basic rules to follow. Any animation construct that provides information (e.g., variables and plots) can be duplicated. Animation constructs that show an activity of an entity (e.g., queues and resources) should not be duplicated in an animation. The reason is quite simple. If you have two animated queues with the same name, Arena would not know which animated queue should show a waiting entity Although Arena will allow you to duplicate an animated queue, it will generally show all waiting entities in the last animated queue that you placed.

We'll use the "pull apart" method to create our enhanced animation. Let's start by using the zoom-out feature, *View > Zoom Out* (or the 🔍 key), to reduce the size of our model. Now click on a queue and use *Edit > Cut* (or *Ctrl+X*) to cut or remove it from the current model and then *Edit > Paste* (or *Ctrl+V*) to place the queue in the general area where you want to build your animation (click to place the floating rectangle). Repeat this action for the remaining queues, placing them in the same general pattern as in the original model. You can now zoom in on the new area and start building the enhanced animation. We'll start by changing our queues. Then we'll create new pictures for our entities and add resource pictures. Finally, we'll add some plots and variables.

4.3.1 Changing Animation Queues

If you watched the animation closely, you might have noticed that there were never more than about 14 entities visible in any of the queues, even though our variables indicated otherwise. This is because Arena restricts the number of animated entities displayed in any queue to the number that will fit in the drawn space for the animation queue. The simulation may have 30 entities in the queue, but if only 14 will fit on the animation, only the first 14 will show. Then, as an entity is removed from the queue, the next undisplayed entity in the queue will be shown. While the output statistics reported at the end will be correct, this can be rather deceptive to the novice and may lead you to assume that the system is working fine when, in fact, the queues are quite large. There are three obvious ways (at least to us) to avoid this problem: one is to watch the animation variable

for the number in queue, the second is to increase the size of the animation queue, and the third is to decrease the size of the entity picture (if it remains visually accurate).

Let's first increase the size of the queue. *Figure 4.9* shows the steps we'll go through as we modify our queue. We first select the queue (View 1) by single-clicking on the Rework queue, *Rework Process.Queue*. Notice that two handles appear, one at each end. You can now place your pointer over the handle at the left end, and it will change to cross hairs. Drag the handle to stretch the queue to any length and direction you want (View 2). If you now run the simulation, you should occasionally see a lot more parts waiting at Rework.

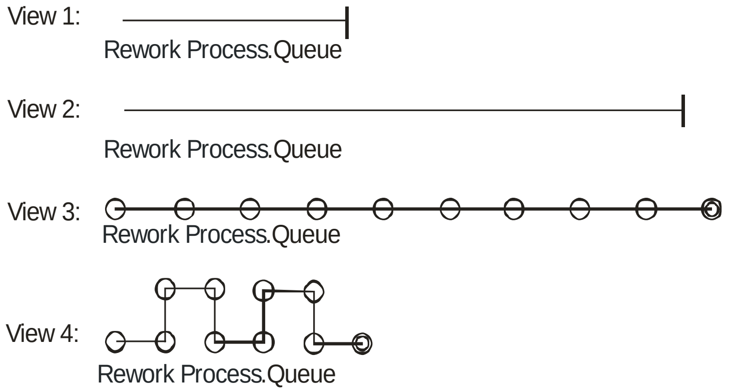
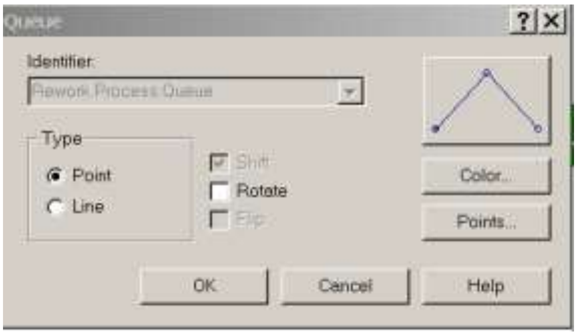


Figure 4.9: Alternate Ways to Display a Queue



Type Point	 *select*

Display 4.23. The Queue Dialog Box

We can also change the form of the queue to represent the physical location point of each entity in it. Double-click on the selected queue and the Queue dialog box will appear as in *Display 4.23*.

Select the Point Type of the queue and click the Points button. Then add points by successively clicking Add. We could change the rotation of the entity at each point, but for now we'll accept the default of these values. When you accept these changes, the resulting queue should look something like the one shown in View 3 of *Figure 4.9*. Note that the front of the queue is denoted by a point surrounded by two circles. You can then drag any of these points into any formation you like (View 4). If you want all these points to line up neatly, you may want to use the Snap option discussed in Chapter 3. Arena will now place entities on the points during an animation run and move them forward, much like a real-life waiting line operates.

For our animation, we've simply stretched the queues (as shown in View 2 of *Figure 4.9*) for the Prep A, Prep B, and Sealer areas. We did get a little fancy with the Rework queue. We changed the form of the queue to points and added 38 points. This allowed us to align them in four rows of ten to represent four available racks as shown in *Figure 4.10* (this was much easier to do with the Snap option on).

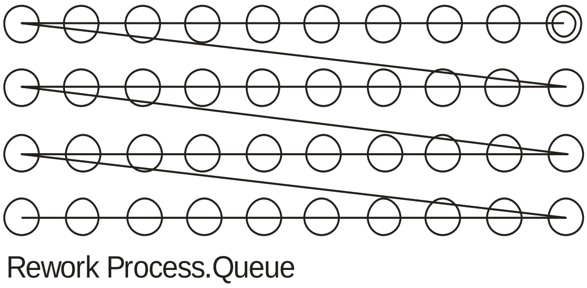
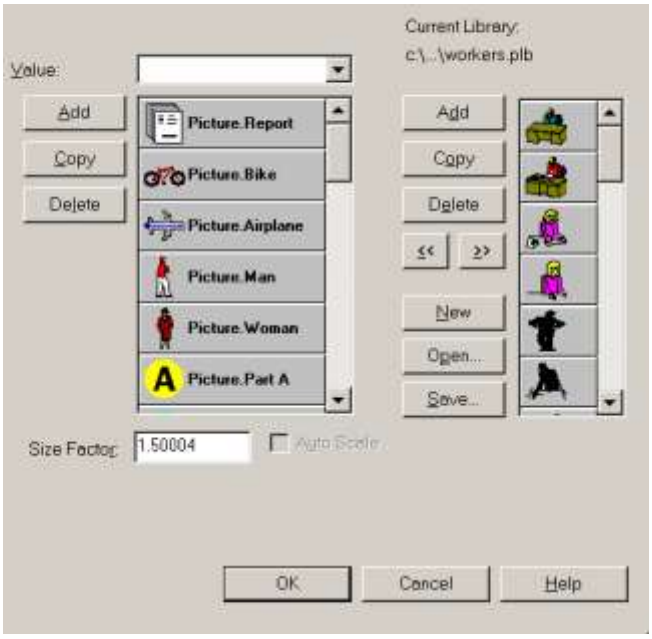


Figure 4.10. The Rework Queue with 40 Points

4.3.2 Changing Entity Pictures

Now let's focus our attention on our animation entities. In our current animation, we arbitrarily selected blue and red balls for the two kinds of entities. Let's say we want our entities to be similar to the balls but have the letter "A" or "B" displayed inside the ball. You create new pictures in the Entity Picture Placement window, as shown in *Figure 4.11*, which is opened using *Edit > Entity Pictures*. The left side of this window contains entity pictures currently available in your model, displayed as a list of buttons with pictures and associated names. The right side of this window is used for accessing picture libraries, which are simply collections of pictures stored in a file. Arena provides several of these libraries with a starting selection of icons; you might want to open and examine them before you animate your next model (their file names end with .plb).


Current Library:
c:\...\workers.plb

Value:

Picture.Report
Picture.Bike
Picture.Airplane
Picture.Man
Picture.Woman
Picture.Part A

Size Factor: 1.50004

☐ Auto Scale

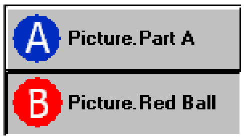
Figure 4.11. The Entity Picture Placement Window

There are several ways to add a new picture to your animation. You can use the Add button (on the left) to draw a new picture for the current list, or you can use the Copy button (on the left) to copy a picture already on the current list. If you use the Add function, your new entry will not have a picture or a name associated with it until you draw it and give it a name in the Value field above the list. If you use the Copy function, the new picture and name will be the same as the picture selected when you copied.

To add a picture from a library to your current entity picture list, highlight the picture you want to replace on the left, highlight the new selection from a library on the right, and click on the left arrow button («) to copy the picture to your picture list. You can also build and maintain your own picture libraries by choosing the *New* button, creating your own pictures, and saving the file for future use. Or you can use clip art by using the standard copy and paste commands.

For this example, as for most of our examples, we'll keep our pictures fairly simple, but you can make your entity and resource pictures as fancy as you want. Since the blue and red balls were about the right size, let's use them as our starting point. Click on the *Picture.Blue Ball* icon in the current list on the left and then click Copy. Now select one of these two identical pictures and change the name (in the Value dialog box) to *Picture.Part* A (now wasn't that obvious?). Note that as you type in the new name it will also change on the selected icon. To change the picture, double-click on the picture icon. This opens the Picture Editor window that will allow you to modify the picture drawing. Before you change this picture, notice the small gray circle in the center of the square; this is the entity reference point, which determines the entity's relation with the other animation objects. Basically, this point will follow the paths when the entity is moving, will reside on the seize point when the entity has control of a resource, and so on.

We'll change this picture by inserting the letter "A" in the center of the ball and changing to a lighter fill color so the letter will be visible. When you close this window, the new drawing will be displayed beside the Picture.Part A name. Now repeat the same procedure to make a new picture for *Part B*. Your final pictures should look something like *Figure 4.12*.



Picture.Part A

Picture.Red Ball

Figure 4.12. The Final Entity Pictures

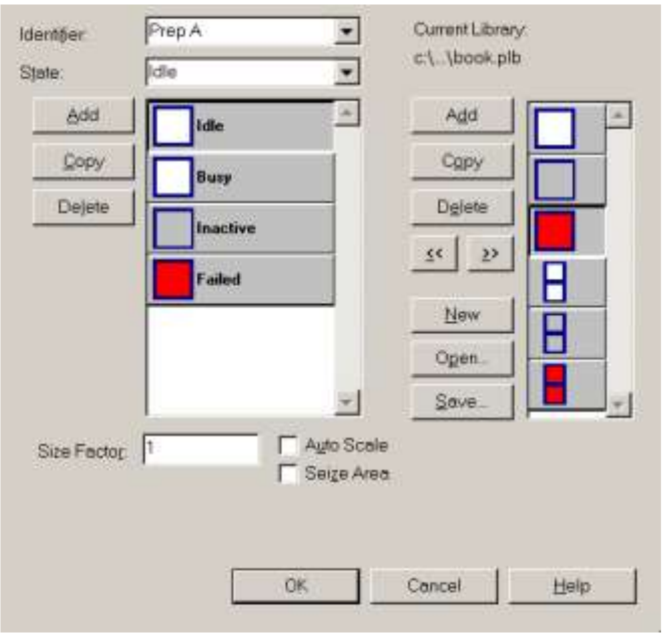
Before we close this window, you might notice that the "A" and "B" do not appear in the picture name because there is not enough room. However, if you click on one of the pictures, the full name will be displayed in the Value field at the top. Also note that there is a Size Factor field in the lower left-hand portion of the window (see *Figure 4.11*). You can increase or decrease your entity picture size by changing this value. For our animation, we increased the Size Factor from 1 to 1.3.

The final step is to assign these new pictures to our parts so they will show up in the animation. You do this by clicking on the **Entity** data module and entering the new names in the Initial Picture cell for our two parts. You might note that your new names will not appear on the drop-down list, so you will need to type them in. However, once you have entered the new names and accepted the data, they will be reflected on the list.

4.3.3 Adding Resource Pictures

Now that we've completed our animated queues and entities, let's add resource pictures to our animation. You add a resource picture by clicking the Resource button () found in the Animate toolbar. This will open the Resource Picture Placement window, which looks very similar to the Entity Picture Placement window. There's very little difference between an entity picture and a resource picture other than the way we refer to them. Entities acquire pictures by assigning a picture name somewhere in the model. Resources acquire pictures depending on their state. In Section 4.2.1, we discussed the four automatic resource states (Idle, *Busy*, *Failed*, and *Inactive*). When you open a Resource Picture Placement window, you might notice that there is a default picture for each of the four default states. You can, however, change the drawings used to depict the resource in its various states, just like we changed our entity pictures.

First we need to identify which resource picture we are creating. You do this by using the drop-down list in the Identifier box to select one of our resources (e.g., Prep A). Now let's replace these pictures as we did for the entity pictures. Double-click on the Idle picture to open the Picture Editor window. Use the background color for the fill, make the line thickness 3 points (from the Draw toolbar), and change the line color. Note that the box must be highlighted in order to make these changes. The small circle with the cross is the reference point for the resource, indicating how other objects (like entity pictures) align to its picture; drag this into the middle of your box. Accept this icon (by closing the Picture Editor window) and return to the Resource Picture Placement window. Now let's develop our own picture library. Choosing the New button from the Resource Picture Placement window opens a new, empty library file. Now select your newly created icon, click Add under the current library area, and then click the right arrow button. Click the Save button to name and save your new library file (e.g., Book. plb).



Identifier: Prep A

State: Idle

Current Library: c:\...\book.plb

Size Factor: 1

□ Auto Scale

□ Seize Area

Figure 4.13. The Resource Pictures

We'll now use this picture to create the rest of our resource pictures. Highlight the Busy picture on the left and the new library picture on the right and use the left arrow button to make your busy picture look just like your idle picture. When the animation is running, the entity picture will sit in the center of this box, so we'll know it's busy. However, you do need to check the Seize Area toggle at the bottom of the window for this to happen. Now copy the same library picture to the inactive and failed states. Open each of these pictures and fill the box with a color that will denote whether the resource is in the failed or inactive state (e.g., red for failed and gray for inactive). Now copy these two new pictures to your library and save it. Your final resource pictures should look something like those shown in *Figure 4.13*. We also increased our Size Factor to 1.3 so the resource size will be consistent with our entity size.

When you accept the resource pictures and return to the main model window, your pointer will be a cross hair. Position this pointer in the approximate area where you want to place the resource and click. This places the new resource on your animation. (By the way, have you remembered to save your model recently?) Your new resource icon may be larger than you want, so adjust it appropriately by dragging one of its corner handles. The resource picture also contains an object that appears as a double circle with a dashed line connected to the lower left portion of the resource picture-the *Seize Area*. This Seize Area is where your entity will sit when it has control of the resource; drag it to the center of your resource picture if necessary. Now run your animation to see if your entity and resource pictures are the sizes you want. If not, you can adjust the position of the seize area by pausing the run, displaying seize areas (using the *View* > *Layers* menu option), and dragging it to the desired location. After you've fine-tuned its position, you'll probably want to turn off the display of the seize area layer before ending the run.

Once you're satisfied with your animation of the Prep A resource, you'll want to add animations for the rest of the resources. You could repeat the above process for each resource, or you can copy and paste the Prep A resource for the Prep B and Sealer pictures. Once you've done this, you'll need to double-click on the newly pasted resource to open the Resource Picture Placement window and select the proper name from the drop-down list in the Identifier field.

The picture of the rework resource will have to be modified because it has a capacity of 2 during the second shift. Let's start by doing a copy/paste as before, but when we open the Resource Picture Placement window, edit the Idle picture and add another square (Edit > Duplicate or Edit > Copy followed by Edit > Paste) beside or under the first picture. This will allow room for two entities to reside during the second shift. Copy this new picture to your library and use it to create the remaining rework pictures. Rename the resource (Rework) and close the window.

The original resource animation had only one seize area, so double-click on the seize area, click the Points button, and add a second seize area. Seize areas are much like point queues and can have any number of points, although the number of points used depends on the resource capacity. Like a queue, seize areas can also be shown as a line. Close this window and position the two seize-area points inside the two boxes representing the resource.

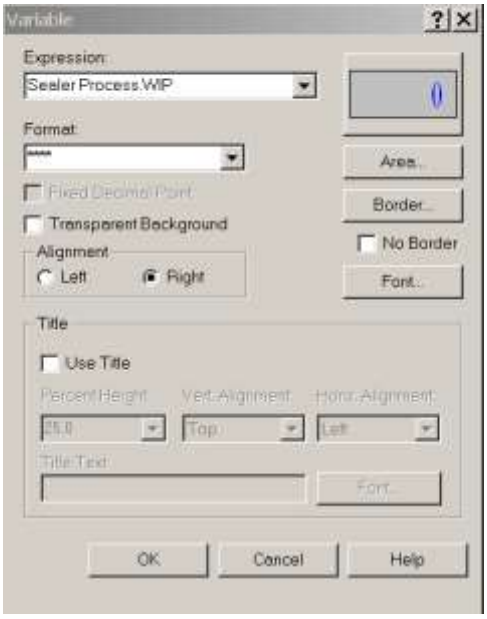
You should now have an animation that's starting to look more like your perceived system. You may want to reposition the resources, queues, stations, and so forth, until you're happy with the way the animation looks. If you've been building this model yourself and based it on Model 4-2, you'll need to clear (uncheck) *Run* > *Run Control* > *Batch Run* (*No Animation*) in order to enjoy the fruits of your artistic labors to create Model 4-3. You also could add text to label things, place lines or boxes to indicate queues or walls, or even add a few potted plants.

4.3.4 Adding Variables and Plots

The last thing we'll do is add some additional variables and a plot to our animation. Variables of interest are the number of parts at each process (in service plus in queue) and the number of parts completed (via each of the three exit possibilities). We could just copy and paste the variables that came with the flowchart modules.

First copy and paste the four variables that came with our process modules; put them right under the resource picture that we just created. Then resize them by highlighting the variable and dragging one of the handles to make the variable image bigger or smaller. You can also reformat, change font, change color, etc., by double-clicking on the variable to open the Variable window shown in *Figure 4.14*.

We then repeated this process for the three variables that came with our **Dispose** modules. Finally, we used the Text tool from the Animate toolbar to label these variables. Now let's add a plot for the number in the rework queue. Clicking the Plot button (📈) from the Animate toolbar opens the Plot window. Use the Add button to enter the expression *NQ*(*Rework Process.Queue*) . Recall that we used this same expression when we created our Frequencies data. We also made a number of other entries as shown in *Display 4.24*.



Expression: Sealer Process.WIP

Format: *****

□ Fixed Decimal Point

□ Transparent Background

□ No Border

Alignment: ○ Left ● Right

□ Use Title

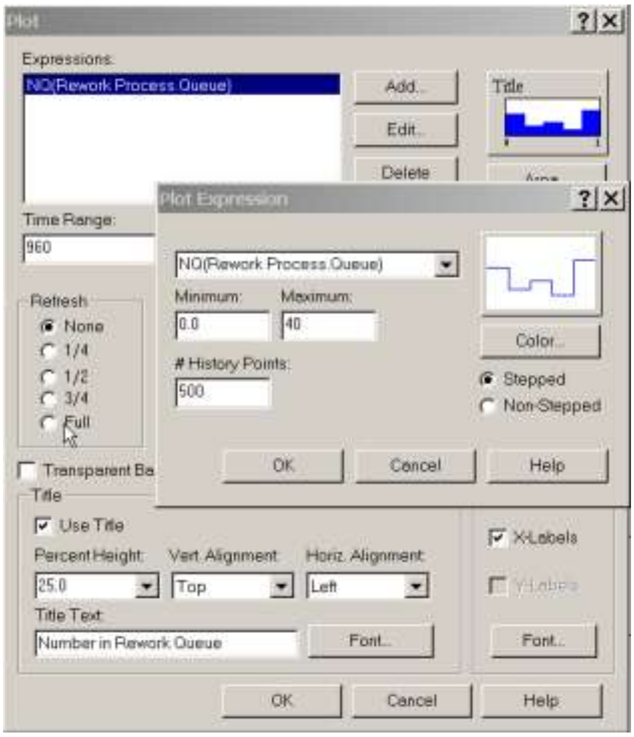
Percent Height: 25.0

Vert. Alignment: Top

Horiz. Alignment: Left

Title Text:

Figure 4.14. The Variable Window



Expressions: NQ(Rework Process.Queue)

Time Range: 960

Refresh: ● None ○ 1/4 ○ 1/2 ○ 3/4 ○ Full

□ Transparent Ba

Plot Expression: NQ(Rework Process.Queue)

Minimum: 0.0

Maximum: 40

History Points: 500

● Stepped ○ Non-Stepped

Title

☑ Use Title

Percent Height: 25.0

Vert. Alignment: Top

Horiz. Alignment: Left

Title Text: Number in Rework Queue

☑ X-Labels

□ Y-Labels

Plot Expression Expression Maximum	 NQ(Rework Process.Queue) 40
Plot Time Range Refresh – None X-Labels Use Title Title Text	 9600 Select check Check Number in Rework Queue

Display 4.24. The Plot Window

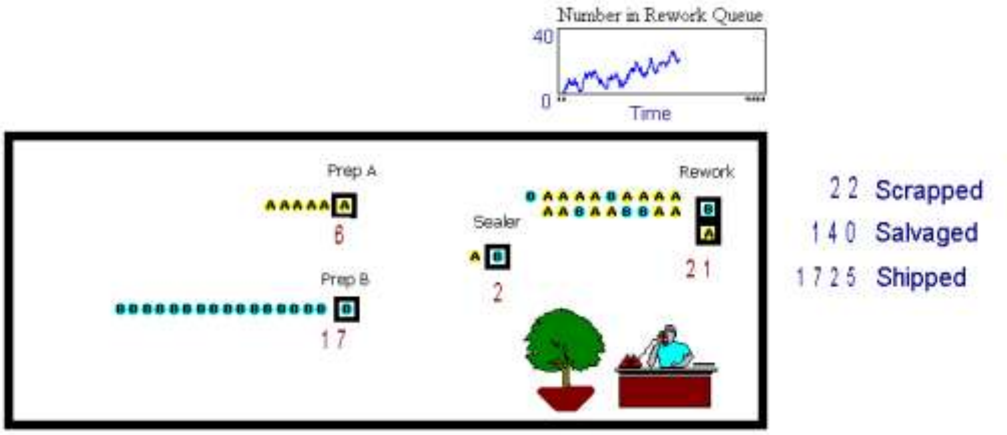


Figure 4.15. The Animation at Time 5555.0829 Minutes: Model 4-3

After you've accepted these data, you may want to increase the size of the plot in the model window and move it somewhere else on the animation. Finally, we used the Text tool from the Animate toolbar to add the *y*-axis limits for our plot.

Your animation should now be complete and look something like the snapshot shown in *Figure 4.15*, which was taken at simulation time 5555.0829 minutes. The final numerical results are, of course, the same as those from Model 4-2 since we changed only the animation aspects to create the present Model 4-3.

4.4 Model 4-4: The Electronic Assembly and Test System with Part Transfers

So far in this chapter, we've developed successive models of the electronic assembly and test system with the assumption that all part transfers between operations occurred instantaneously. Let's generalize that assumption and now model the system with all part transfers taking two minutes, regardless of where they're coming from or going to. This includes the transfer of arriving parts to the prep areas and transfer of the departing parts from either the Sealer or Rework station to be scrapped, salvaged, or shipped. We'll modify Model 4-3 to create Model 4-4.

4.4.1 Some New Arena Concepts: Stations and Transfers

In order to model the two-minute transfer times and to show the part movement, we need to understand two new Arena concepts: Stations and Station Transfers. Arena approaches the modeling of physical systems by identifying locations called Stations. Stations may be thought of as a place at which some process occurs. In our example, stations will represent the locations for the part arrivals, the four manufacturing cells, and part departures. Each station is assigned a unique name. In the model, stations also appear as entry points to sections of model logic, working in conjunction with our other new topic, Station Transfers.

Station Transfers allow us to send an entity from one station to another without a direct connection. Arena provides several different types of station transfers that allow for positive transfer times, constrained movement using material-handling devices, and flexible routings that depend on the entity type. The station transfer we'll use here is called a Route, which allows the movement of entities from one station to another. Routes assume that time may be required for the movement between stations, but that no additional delay is incurred because of other constraints, such as blocked passageways or unavailable material-handling equipment. The route time can be expressed as a constant, a sample from a distribution, or, for that matter, any valid expression.

We often think of stations as representing a physical location in a system; however, there's no strict requirement that this be so, and in fact, stations can be used effectively to serve many other modeling objectives. Stepping back for a moment from their intended use in representing a system, let's examine what happens in Arena when an entity is transferred (e.g., routed) to a station. First, we'll look at the model logic-moving entities from module to module during the run. Underneath the hood, as we discovered (in painstaking detail) in Chapter 2, a simulation run is driven by the entities creating them, moving them through logic, placing them on the event calendar when a time delay is to be incurred, and eventually destroying them. From this perspective, a station transfer (route) is simply another means of incurring a time delay.

When an entity leaves a module that specifies a Route as the transfer mechanism, Arena places the entity on the event calendar with an event time dictated by the route duration (analogous to a **Delay** module). Later, when it's the entity's turn to be removed from the event calendar, Arena returns the entity to the flow of model logic by finding the module that defines its destination station, typically a **Station** module. For example, let's assume a **Route** module is used to send an entity to a station named Sealer. When the entity comes off the event calendar, Arena finds the module that defines the Sealer station, and the entity is directed or sent to that module. This is in slight contrast to the direct module connections we've seen so far, where the transfer of an entity from module to module occurred without placing the entity on the event calendar and was represented graphically in the model by a Connection line between two modules. While the direct connections provide a flowchart-like look to a model, making it obvious how entities will move between modules, station transfers provide a great deal of power and flexibility in dispatching entities through a model, as we'll see when we cover Sequences in Section 7.1.

Stations and station transfers also provide the driving force behind an important part of the model's animation-displaying the movement of entities among stations as the model run progresses. The stations themselves are represented in the flowchart view of the model using station marker symbols. These stations establish locations on the model's drawing where station transfers can be initiated or terminated. The movement of entities between the stations is defined by route path objects, which visually connect the stations to each other for the animation and establish the path of movement for entities that are routed between the stations.

You'll soon see that the station markers define either a destination station (for the ending station of a route path) or an origin station that allows transfer out of the module via a station transfer (for the beginning of a route path). You add animation stations via the Station object from the Animate Transfer toolbar. You add route paths by using the Route object from the Animate Transfer toolbar and drawing a polyline that establishes the graphical path that entities follow during their routes. When the simulation is running, you'll see entity pictures moving smoothly along these route paths. This begs the question: How does this relate to the underlying logic where we just learned that an

entity resides on the event calendar during its route time delay? The answer is that Arena's animation "engine" coordinates with the underlying logic "engine"; in this case, the event calendar. When an entity encounters a route in the model logic and is placed on the event calendar, the animation shows the entity's picture moving between the stations on the route path. The two engines coordinate so that the timing of the entity finishing its animation movement on the route and being removed from the event calendar to continue through model logic coincide, resulting in an animation display that is representative of the model logic at any point in time.

4.4.2 Adding the Route Logic

Since the addition of stations and transfers affects not only the model, but also the animation, let's start with our last model (Model 4-3). Open this model and use *File > Save As* to save it as *Model 04-0 4.doe*. Let's start with the part arrivals. Delete the connections between the two **Assign** modules and the *Part A Prep* and *Part B Prep* **Process** modules. Now move the two **Create**/**Assign** module pairs to the left to allow room for the additional modules that we'll add for our route logic. To give you a preview of where we're headed here, our final modification to this section of our model, with the new modules, is shown in *Figure 4.16*.

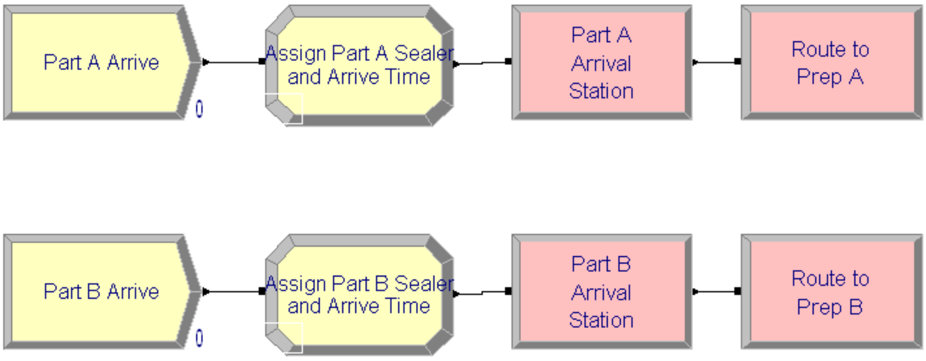
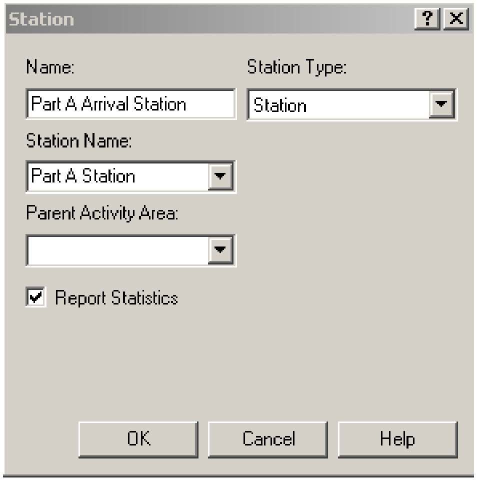


Figure 4.16. The Part Arrival Logic Modules

The existing **Create** and **Assign** modules for the *Part A* and *Part B* arrivals remain the same as in the original model. In order to add our stations and transfers, we first need to define the station where the entity currently resides and then route the entity to its destination station. Let's start with Part A. We place a **Station** module (from the Advanced Transfer panel, which you may need to attach to your model) to define the location of Part A, as seen in *Display 4.25*. We've entered a Name (*Part A Arrival Station*), defaulted the Station Type to Station, and defined our first station (*Part A Station*). This module defines the new station (or location) and assigns that location to the arriving part. Note that the Name box in the **Station** module simply provides the text to appear in the model window and does not define the name of the station itself. The Station Name box defines the actual station name to be used in the model logic *Part A Station*.



Name: Part A Arrival Station

Station Type: Station

Station Name: Part A Station

Parent Activity Area:

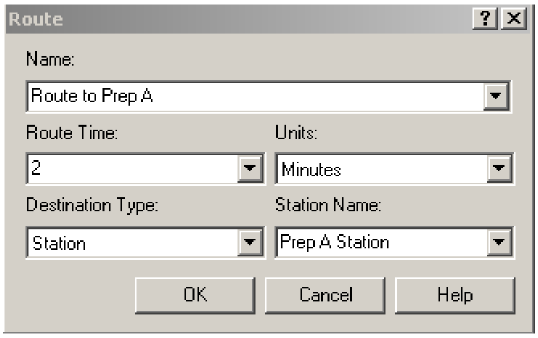
☑ Report Statistics

Name Station Name	Part A Arrival Station Part A Station

Display 4 25. The **Station** module

We'll next add the **Route** module (from the Advanced Transfer panel), which will send the arriving part to the Prep A area with a transfer time of two minutes. We'll provide a module name, enter a Route Time of 2, in units of Minutes, default the Destination Type (Station), and enter the destination Station Name as Prep A S tat ion, as shown in *Display 4.26*. This will result in any Part A part arrival being sent to the yet-to-be-established *Prep A Station* (though we just defined its name here).

We've added the same two modules to the *Part B* arrival stream. The data entries are essentially the same, except that we use *Part B* in place of all Part A occurrences (modelers who are both lazy and smart would duplicate these two modules, make the few required edits, and connect as needed). You might note that there are no direct connects exiting from the two **Route** modules. As discussed earlier, Arena will take care of getting the entities to their correct stations.


Route

Name: Route to Prep A

Route Time: 2

Units: Minutes

Destination Type: Station

Station Name: Prep A Station

Name	Route to Prep A
Route Time	2
Units	Minutes
Station name	Prep A Station

Display 4.26. The **Route** module

Now let's move on to the prep areas and make the necessary modifications, the result of which is depicted in *Figure 4.17* (almost...we'll make one slight modification here before we're done). The two modules preceding the prep areas' **Process** modules are **Station** modules that define the new station locations, *Prep A Station* and *Prep B Station*. The data entries are basically the same as for our previous **Station** modules (*Display 4.25*), except for the module Name and the Station Name. The parts that are transferred from the part arrival section using the **Route** modules will arrive at one of these stations. The two **Process** modules remain the same. We then added a single **Route** module, with incoming connectors from both Prep areas. Although the two prep areas are at different locations, the parts are being transferred to the same station, Sealer Station. Arena will keep track of where the parts originated, the Prep A or Prep B station. Again, the only change from the previously placed **Route** modules are the module Name, Route to Sealer, and the destination Station Name, Sealer Station.

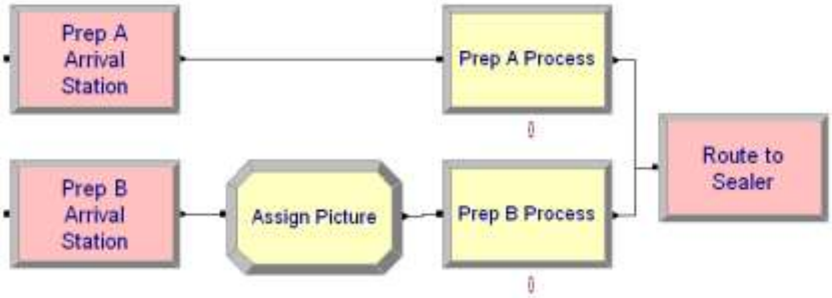


Figure 4.17. The Prep Area Logic Modules

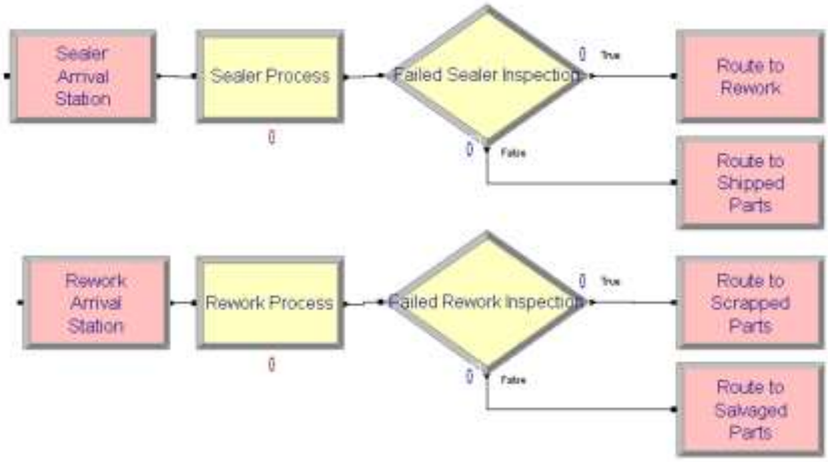


Figure 4.18. The Sealer and Rework Area Logic Modules

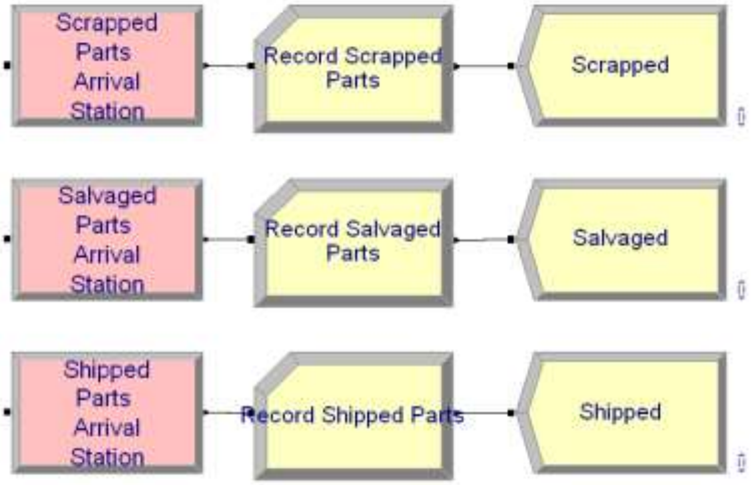

Scrapped Parts Arrival Station
Record Scrapped Parts
Scrapped
Salvaged Parts Arrival Station
Record Salvaged Parts
Salvaged
Shipped Parts Arrival Station
Record Shipped Parts
Shipped
0

Figure 4.19. The Scrapped, Salvaged, and Shipped Logic Modules

By now you can see what is left to complete our model changes. We simply need to break out our different locations and add **Station** and **Route** modules where required. The model logic for the Sealer and Rework areas is shown in *Figure 4.18*. We've used the same notation as before with our new stations named *Sealer Station*, *Rework Station*, *Scrapped Station*, *Salvaged Station*, and *Shipped Station*.

The model logic for our Scrapped, Salvaged, and Shipped areas is shown in *Figure 4.19*.

At this point, your model should be ready to run (correctly), although you probably will not notice much difference in the animation, except maybe that the queues for *Part B Prep* and Rework seem to get a lot longer (so we made more room in these queue animations and re-scaled the y-axis on the Rework Queue plot). If you look at your results, you might observe that the part cycle times are longer than in Models 4-3 and 4-2, which is caused by the added transfer times; however, if you were to make multiple replications of each of these models (see Section 2.6.2), you'd notice quite a lot of variability in these output performance measures across replications, so concluding much from single replications of this model vs. Model 4-3 (or, equivalently, Model 4-2) is risky business.

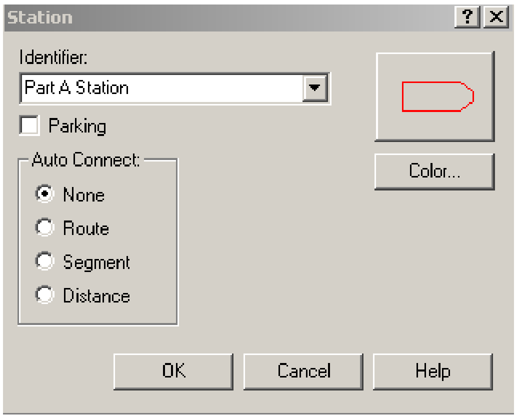
Maybe you're wondering about a simpler approach to modeling non-zero transfer times-instead of all these **Station** and **Route** modules all over the place, why not just stick in a **Pro-**

cess module interrupting each arc of the flowchart where the transfer time occurs with Action = Delay and a Delay Type that's Constant at 2 Minutes? Actually, this would work and would produce valid numerical output results. However, it would not allow us to animate the entities moving around, which we'd like to do (for one thing, it's fun to watch...briefly). So we'll now add the route transfers to our animation.

4.4.3 Altering the Animation

In the process of modifying the animation, we moved a lot of the existing objects. If you want your animation to look like ours, you might want to first take a look ahead at *Figure 4.20*. (Of course, you could always just open Model 04-04.doe.)

We'll now show you how to add stations and routes to your animation. Let's start by adding the first few animation station markers. Click the *Station* button ( ) found in the Animate Transfer toolbar to open the Station dialog box shown in *Display 4.27*. If you don't have the Animate Transfer toolbar open, you can use *view* > *Toolbars* (or right click in any toolbar area) to choose to display it. Now use the drop-down list to select from the already-defined stations our first station, *Part A Station*. When you accept the dialog box, your pointer will have changed to cross hairs. Position this new pointer to the left of the *Prep A queue* and click to add the station marker.



Station

Identifier: Part A Station

□ Parking

Auto Connect:

● None

○ Route

○ Segment

○ Distance

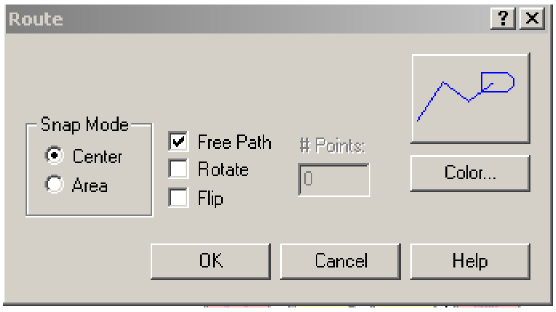
Identifier	Part A Station

Display 4.27. The Station Dialog Box

Repeat the same set of steps to place the *Prep A Station* marker just above the Prep A queue. Now that we have our first two stations placed, let's add the route path. Click the Route button (R) in the Animate Transfer toolbar. This will open the Route dialog box shown in *Display 4.28*. For our animation, we simply accepted the default values, although you can change the characteristics of the path by selectively clicking different buttons. To explore what the other Route options are, click the What's This? help button, then click on the item of interest to display a brief description. After you accept the Route dialog box, the pointer changes to cross hairs. Place the cross hairs inside a station marker (*Part A Station*) at the start of a path and click; this will start the route path. Move the pointer and build the remainder of the path by clicking where you want "corners," much like drawing a polyline. The route path will automatically end when you click inside the end station marker (*Prep A Station*).

If this is your first animated route, you might want to go ahead and run your model so you can see the arriving parts move from the *Part A Station* to the *Prep A Station*. If you're not happy with the placement of the routes, they can be easily changed. For example, if you click on the station marker above the *Prep A.Process queue* (or, if you have the options checked under View > Data Tips, merely hover your pointer over the station marker), the name of the station

(*Prep A Station*) will appear below the marker. You can drag the station marker anywhere you want (well, almost anywhere, as long as you stay in the model window). Note that if you move the station marker, the routes stay attached. Once you've satisfied your curiosity, place the stations and route path for the *Part B* arrival and transfer to the Prep B station.



Route

Snap Mode
● Center
○ Area

☑ Free Path
□ Rotate
□ Flip

Points: 0

Display 4.28. The Route Dialog Box

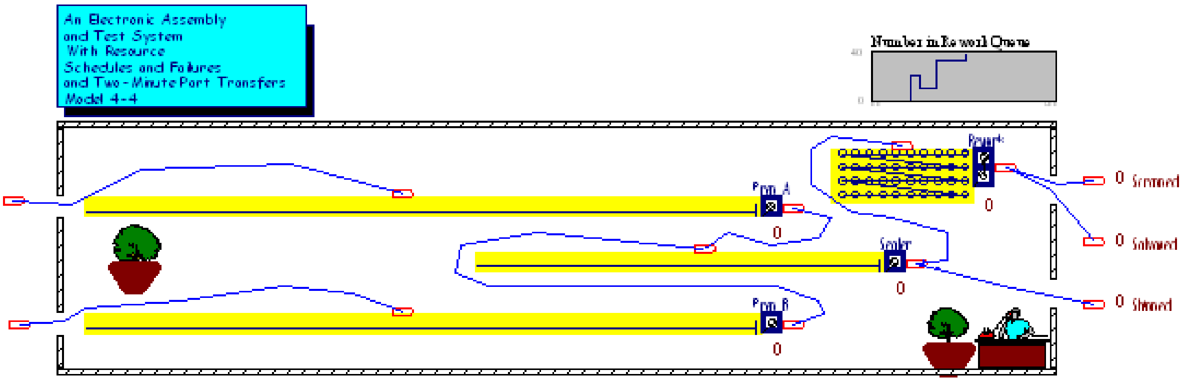


Figure 4.20. The Final Animation for Model 4-4

If you add the remaining stations and routes, your animation will be complete. However, from a visual point of view you may not have an animation that accurately captures the flow of parts. Recall that we placed our prep stations just above the prep queues, which are to the left of the prep areas. Let's assume that in the physical facility the parts exit from the right side of the area. If you had used the existing prep stations, the parts would have started their routes from the left side of the areas. The solution to this problem is simple just add a second station marker at the prep areas (placed on the right side) with the same Identifier for the logical station. When Arena needs to route an entity, it will look for the route path, not just the station. So as you build the remaining routes, you'll need to have two station markers (with the same name) at each of the prep areas, the sealer area, and the rework area. Your final animation should look something like *Figure 4.20*.

Finally, there's one subtle point that you may not have noticed. Run your animation and watch the *Part B* entities as they travel from arrival to the *Part B* Prep area. Remember that we're creating four entities at a time, but we see only one moving. Now note what happens as they arrive at the queue-suddenly there are four entities. As it turns out, there are really four entities being transferred, it's just that they're all on top of one another during the transfer-they're all moving at precisely the same speed because the transfer times are a constant for all entities. You can check this out by opening the Route to Prep B Route module and changing Route Time from a constant 2 to EXPO(2) , causing the entities' individual travel times to differ. Now when you run the animation, you should see all four entities start out on top of one another and then separate as they move toward the queue. You can exaggerate this by clicking on the route path and making it much longer. Although this will show four entities moving, it may not represent reality since, if you believe the modeling assumptions, this travel time was supposed to be a constant 2 minutes (but your boss will probably never know).

Here's a blatant sleight of hand to keep the model assumptions yet make the animation look right. If we show the B parts arriving, with each part animated as a "big batch" picture, we'll get the desired visual results. There will still be four moving across to the Prep B Station, but only the top one will show. We need only convert the picture back to a single entity when they enter the queue at the prep area. In order to implement this concept, we'll need to create a new "big batch" picture and make two changes to the model.

First use *Edit > Entity Pictures* to open the Entity Picture Placement window. Next make a copy of the *Picture.Part B* picture and rename it *Picture.Batch B* in the Value field above it. Edit this new picture's drawing by double-clicking on its button in the list. In the Picture Editor, use Copy and Paste to get four *Part B* circles as shown in *Figure 4.21*; move the entity reference point to be near the center of your four-pack. Then close the Picture Edit window and click *OK* to return to the model window.

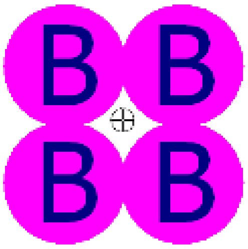


Figure 4.21. The Batch B Entity Picture

Now that we have a batch picture, we need to tell Arena when it should be used. Open the *Assign Part B Sealer and Arrive Time* **Assign** module and Add another assignment (in this case, it doesn't matter where in the list of assignments it's added since there's no dependency in these assignments of one on another). This new assignment will be of Type *Entity Picture*, with the Entity Picture being *Picture.Batch B* (which you'll need to type in here for the first time). This will cause Arena to show the batch picture when it routes the *Part B* arrivals to the Prep B station. Now we need to convert the picture back to a single entity when it arrives at the Prep B area. You'll need to insert a new **Assign** module (*Assign Picture*) between the *Prep B station*, *Prep B Arrival Station*, and the *Prep B Process* **Process** module. This assignment will have Entity Picture Type, with the Entity Picture being *Picture.Part B* (which will be on the drop-down list for Entity Picture). Thus, when the entities enter the queue or service at Prep B Process, they will be displayed as individual parts.

Now if you run the simulation (with the animation turned on), you'll see an arriving batch being routed to the Prep B area and individual parts when the batch enters the queue. Defining different entity pictures allows you to see the different types of parts move through the system. Note the batch of four B parts when they enter the system. Although there are still four of these pictures, each on top of the other, the new Batch B picture gives the illusion of a single batch of four parts entering.

Now that you understand the concept of routes, we'll use them in Chapter 7 in a more complex system.

4.5 Input Analysis: Specifying Model Parameters and Distributions

As you've no doubt noticed, there are a lot of details that you have to specify completely in order to define a working simulation model. Probably you think first of the logical aspects of the model, like what the entities and resources are, how entities enter and maybe leave the model, the resources they need, the paths they follow, and so on. These kinds of activities might be called structural modeling since they lay out the fundamental logic of what you want your model to look like and do.

You've also noticed that there are other things in specifying a model that are more numerical or mathematical in nature (and maybe therefore more mundane). For example, in specifying Model 4-2, we declared that interarrival times for Part A were exponentially distributed with a mean of 5 minutes, total processing times for *Part B Prep* followed a triangular (3, 5, 10) distribution, the "up" times for the sealer were draws from an exponential (120) distribution, and we set up a schedule for the number of rework operators at different times. You also have to make these kinds of specifications, which might be called quantitative modeling, and are potentially just as important to the results as are the structural-modeling assumptions.

So where did we get all these numbers and distributions for Model 4-2 (as well as for practically all the other models in this book)? OK, we admit that we just made them up, after playing around for a while, to get the kinds of results we wanted in order to illustrate various points. We get to do this since we're just writing a book rather than doing any real work, but unfortunately, you won't have this luxury. Rather, you need to observe the real system (if it exists) or use specifications for it (if it doesn't exist), collect data on what corresponds to your input quantitative modeling, and analyze these data to come up with reasonable "models," or representations of how they'll be specified or generated in the simulation. For Model 4-2, this would entail collecting data on actual interarnval times for *Part A*, processing times for *Part B Prep*, up times for the sealer, and actual staffing at the rework station (as well as all the other quantitative inputs required to specify the model). You'd then need to take a look at these data and do some kind of analysis on them to specify the corresponding inputs to your model in an accurate, realistic, and valid way.

In this section, we'll describe this process and show you how to use the Arena Input Analyzer (which is a separate application that accompanies and works with Arena) to help you fit probability distributions to data observed on quantities subject to variation.

4.5.1 Deterministic vs. Random Inputs

One fundamental issue in quantitative modeling is whether you're going to model an input quantity as a deterministic (non-random) quantity, or whether you're going to model it as a *random variable* following some probability distribution. Sometimes it's clear that something should be deterministic, like the number of rework operators, though you might want to vary the values from run to run to see what effect they have on performance.

But sometimes it's not so clear, and we can only offer the (obvious) advice that you should do what appears most realistic and valid so far as possible. In Section 4.5.2, we'll talk a little about using your model for what's called *sensitivity analysis* to measure how important a particular input is to your output, which might indicate how much you need to worry about whether it should be modeled as deterministic or random.

You might be tempted to assume away your input's randomness, since this seems simpler and has the advantage that the model's outputs will be non-random. This can be pretty dangerous, though, from the model-validity viewpoint because it's often the randomness itself that leads to important system behavior that you'd certainly want to capture in your simulation model. For instance, in a simple single-server queue, we might assume that all interarrival times are exactly 1 minute and that all processing times are exactly 59 seconds; both of these figures might agree with average values from observed data on arrivals and service. If the model starts empty with the server idle and the first arrival is at time 0, then there will never be a queue since each customer will finish service and leave 1 second before the next customer arrives. However, if the reality is that the interarrival and service times are exponential random variables with respective means of 1 minute and 59 seconds (rather than constant at these values), you get a very different story in terms of queue length (go ahead and build a little Arena model for this, and run it for a long time); in fact, in the long run, it turns out that the average number of customers in the queue is 58.0167, a far cry from 0. Intuitively, what's going on here is that, with the (correct) random model, it sometimes happens that some obnoxious customer has a long service demand, or that several customers arrive at almost the same time; it's precisely these kinds of random occurrences that cause the queue to build up, which never happen in the (incorrect) deterministic model.

4.5.2 Collecting Data

One of the very early steps in planning your simulation project should be to identify what data you'll need to support the model. Finding the data and preparing them for use in your model can be time-consuming, expensive, and often frustrating; and the availability and quality of data can influence the modeling approach you take and the level of detail you capture in the model.

There are many types of data that you might need to collect. Most models require a good bit of information involving time delays: interarrival times, processing times, travel times, operator work schedules, etc. In many cases, you'll also need to estimate probabilities, such as the percentage yield from an operation, the proportions of each type of customer, or the probability that a caller has a touch-tone phone. If you're modeling a system where the physical movement of entities among stations will be represented in the model, the operating parameters and physical layout of the material-handling system will also be needed.

You can go to many sources for data, ranging from electronic databases to interviews of people working in the system to be studied. It seems that when it comes to finding data for a simulation study, it's either "feast or famine," with each presenting its own unique challenges.

If the system you're modeling exists (or is similar to an actual system somewhere else), you may think that your job's easier, since there should be a lot of data available. However, what you're likely to find is that the data you get are not the data you need. For example, it's common to collect processing-time data on machines (that is, a part's time span from arriving at the machine to completing the process), which at first glance might look like a good source for processing times in the simulation model. But, if the observed processing-time data included the queue time or included machine failure times, they might not fit into the model's logic, which explicitly models the queuing logic and the machine failures separately from the machining time.

On the other hand, if you're about to model a brand new system or a significant modification to an existing one, you may find yourself at the other end of the spectrum, with little or no data. In this case, your model is at the mercy of rough approximations from designers, equipment vendors, etc. We'll have a few suggestions (as opposed to solutions) for you in Section 4.5.5.

In either case, as you decide what and how much data to collect, it's important to keep the following helpful hints in mind:

Sensitivity analysis: One often-ignored aspect of performing simulation studies is developing an understanding of what's important and what's not. Sensitivity analysis can be used even very early in a project to assess the impact of changes in data on the model results. If you can't easily obtain good data about some aspect of your system, run the model with a range of values to see if the system's performance changes significantly If it doesn't, you may not need to invest in collecting data and still can have good confidence in your conclusions. If it does, then you'll either need to find a way to obtain reliable data or your results and recommendations will be coarser.

Match model detail with quality of data: A benefit of developing an early understanding of the quality of your input data is that it can help you to decide how much detail to incorporate in the model logic. Typically, it doesn't make any sense to model carefully the detailed logic of a part of your system for which you have unreliable values of the associated data, unless you think that, at a later time, you'll be able to obtain better data.

Cost: Because it can be expensive to collect and prepare data for use in a model, you may decide to use looser estimates for some data. In making this assessment, sensitivity analysis can be helpful so that you have an idea of the value of the data in affecting your recommendations.

"Garbage in, garbage out": Remember that the results and recommendations you present from your simulation study are only as reliable as the model and its inputs. If you can't locate accurate data concerning critical elements of the model, you can't place a great deal of confidence in the accuracy of your conclusions. This doesn't mean that there's no value in performing a simulation study if you can't obtain "good" data. You still can develop tremendous insight into the operation of a complex system, the interactions among its elements, and some level of prediction regarding how it will perform. But take care to articulate the reliability of your predictions based on the quality of the input data.

A final hint that we might offer is that data collection (and some of their analysis) is often identified as the most difficult, costly, time-consuming, and tedious part of a simulation study. This is due in part to various problems you might encounter in collecting and analyzing data, and in part due to the undeniable fact that it's just not as much fun as building the logical model and playing around with it. So, be of good cheer in this activity, and keep reminding yourself that it's an important (if not pleasant or exciting) part of why you're using simulation.

4.5.3 Using Data

If you have historical data (e.g., a record of breakdowns and repair times for a machine), or if you know how part of a system will work (e.g., planned operator schedules), you still face decisions concerning how to incorporate the data into your model. The fundamental choice is whether to use the data directly or whether to fit a probability distribution to the existing data. Which approach you decide to use can be chosen based on both theoretical issues and practical considerations.

From a theoretical standpoint, your collected data represent what's happened in the past, which may or may not be an unbiased prediction of what will happen in the future. If the conditions surrounding the generation of these historical data no longer apply (or if they changed during the time span in which the data were recorded), then the historical data may be biased or may simply be missing some important aspects of the process. For example, if the historical data are from a database of product orders placed over the last 12 months, but four months ago a new product option was introduced, then the order data stored in the preceding eight months are no longer directly useful since they don't contain the new option. The tradeoffs are that if you use the historical data directly, no values other than those recorded can be experienced; but if you sample from a fitted probability distribution, it's possible to get values that aren't possible (e.g., from the tails of unbounded distributions) or to lose important characteristics (e.g., bimodal data, sequential patterns).

Practical considerations can come into play too. You may not have enough historical data to drive a simulation run that's long enough to support your analysis. You may very well need to consider the effect of file access on the speed of your simulation runs as well. Reading a lot of data from a file is typically slower than sampling from a probability distribution, so driving the model with the historical data during a run can slow you down.

Regardless of your choice, Arena supplies built-in tools to take care of the mechanics of using the historical data in your model. If you decide to fit a probability distribution to the data, the Input Analyzer facilitates this process, providing an expression that you can use directly in your model; we'll go into this in Section 4.5.4. If you want to drive the model directly from the historical data, you can bring the values in once to become part of the model's data structure, or you can read the data dynamically during the simulation run, which will be discussed in Section 10.1.

4.5.4 Fitting Input Distributions with the Input Analyzer

If you decide to incorporate your existing data values by fitting a probability distribution to them, you can either select the distribution yourself, and use the Input Analyzer to provide numerical estimates of the appropriate parameters, or you can fit a number of distributions to the data and select the most appropriate one. In either case, the Input Analyzer provides you with estimates of the parameter values (based on the data you supply) and a ready-made expression that you can just copy and paste into your model.

When the Input Analyzer fits a distribution to your data, it estimates the distribution's parameters (including any shift or offset that's required to formulate a valid expression) and calculates a number of measures of how good the distribution fits your data. You can use this information to select which distribution you want to use in your model, which we discuss below.

Probability distributions can be thought of as falling into two main types: theoretical and empirical. Theoretical distributions, such as the exponential and gamma, generate samples based on a mathematical formulation. Empirical distributions simply divide the actual data into groupings and calculate the proportion of values in each group, possibly interpolating between points for more accuracy.

Each type of distribution is further broken into continuous and discrete types. The continuous theoretical distributions that Arena supports for use in your model are the exponential, triangular, and Weibull distributions mentioned previously, as well as the beta, Erlang, gamma, lognormal, uniform, and normal distributions. These distributions are referred to as continuous distributions because they can return any real-valued quantity (within a range for the bounded types). They're usually used to represent time durations in a simulation model. The Poisson distribution is a discrete distribution; it can return only integer-valued quantities. It's often used to describe the number of events that occur in an interval of time or the distribution of randomly varying batch sizes.

You also can use one of two empirical distributions: the discrete and continuous probability distributions. Each is defined using a series of probability/value pairs representing a histogram of the data values that can be returned. The discrete empirical distribution returns only the data values themselves, using the probabilities to choose from among the individual values. It's often used for probabilistically assigning entity types. The continuous empirical distribution uses the probabilities and values to return a real-valued quantity. It can be used in place of a theoretical distribution in cases where the data have unusual characteristics or where none of the theoretical distributions provide a good fit.

The Input Analyzer can fit any of the above distributions to your data. However, you must decide whether to use a theoretical or empirical distribution, and unfortunately, there aren't any standard rules for making this choice. Generally, if a histogram of your data (displayed automatically by the Input Analyzer) appears to be fairly uniform or has a single "hump," and if it doesn't have any large gaps where there aren't any values, then you're likely to obtain a good fit from one of the theoretical distributions. However, if there are a number of value groupings that have many observations (multimodal) or there are a number of data points that have a value that's significantly different from the main set of observations, an empirical distribution may provide a better representation of the data; an alternative to an empirical distribution is to divide the data somehow, which we'll describe at the end of this section.

The Input Analyzer is a standard tool that accompanies Arena and is designed specifically to fit distributions to observed data, provide estimates of their parameters, and measure how well they fit the data. There are four steps required to use the Input Analyzer to fit a probability distribution to your data for use as an input to your model:

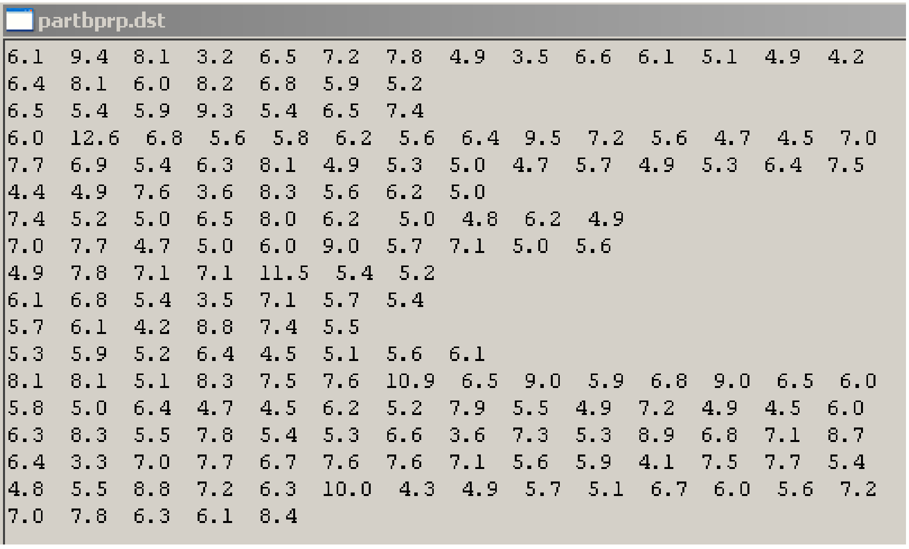
Create a text file containing the data values.

Fit one or more distributions to the data.

Select which distribution you'd like to use.

Copy the expression generated by the Input Analyzer into the appropriate field in your Arena model.

To prepare the data file, simply create an ordinary ASCII text file containing the data in free format. Any text editor, word processor, or spreadsheet program can be used. The individual data values must be separated by one or more "white space characters" (blank spaces, tabs, or line feeds). There are no other formatting requirements; in particular, you can have as many data values on a line as you want, and the number of values per line can vary from line to line. When using a word processor or spreadsheet program, be sure to save the file in a "text only" format. This eliminates any character or paragraph formatting that otherwise would be included. For example, *Figure 4.22* shows the contents of an ASCII file called *partbprp.dst* (the default file extension for data files for the Input Analyzer is *.dst*) containing observations on 187 *Part B Prep times*; note that the values are separated by blanks or line feeds, there are different numbers of observations per line, and there is no particular order or layout to the data.



```
6.1  9.4  8.1  3.2  6.5  7.2  7.8  4.9  3.5  6.6  6.1  5.1  4.9  4.2
6.4  8.1  6.0  8.2  6.8  5.9  5.2
6.5  5.4  5.9  9.3  5.4  6.5  7.4
6.0  12.6  6.8  5.6  5.8  6.2  5.6  6.4  9.5  7.2  5.6  4.7  4.5  7.0
7.7  6.9  5.4  6.3  8.1  4.9  5.3  5.0  4.7  5.7  4.9  5.3  6.4  7.5
4.4  4.9  7.6  3.6  8.3  5.6  6.2  5.0
7.4  5.2  5.0  6.5  8.0  6.2   5.0  4.8  6.2  4.9
7.0  7.7  4.7  5.0  6.0  9.0  5.7  7.1  5.0  5.6
4.9  7.8  7.1  7.1  11.5  5.4  5.2
6.1  6.8  5.4  3.5  7.1  5.7  5.4
5.7  6.1  4.2  8.8  7.4  5.5
5.3  5.9  5.2  6.4  4.5  5.1  5.6  6.1
8.1  8.1  5.1  8.3  7.5  7.6  10.9  6.5  9.0  5.9  6.8  9.0  6.5  6.0
5.8  5.0  6.4  4.7  4.5  6.2  5.2  7.9  5.5  4.9  7.2  4.9  4.5  6.0
6.3  8.3  5.5  7.8  5.4  5.3  6.6  3.6  7.3  5.3  8.9  6.8  7.1  8.7
6.4  3.3  7.0  7.7  6.7  7.6  7.6  7.1  5.6  5.9  4.1  7.5  7.7  5.4
4.8  5.5  8.8  7.2  6.3  10.0  4.3  4.9  5.7  5.1  6.7  6.0  5.6  7.2
7.0  7.8  6.3  6.1  8.4
```

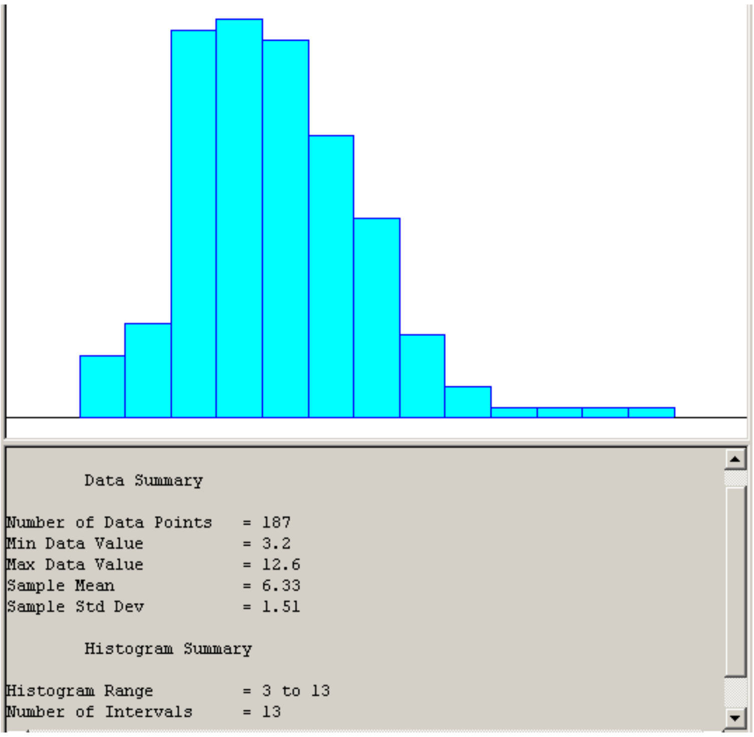
Figure 4 22. Listing of the ASCII File partbprp.dst

To fit a distribution to these data, run the Input Analyzer (e.g., select *Tools* > *Input Analyzer* from Arena). In the Input Analyzer, load the data file into a data fit window by creating a new window (*File* > *New* or the *Ctrl N* button) for your fitting session (not for a data file), and then attaching your data file using either *File > Data File > Use Existing* or the F5 button. The Input Analyzer displays a histogram of the data in the top part of the window and a summary of the data characteristics in the bottom part, as shown in *Figure 4.23*.

You can adjust the relative size of the windows by dragging the splitter bar in the center of the window. Or, to see more of the data summary, you can scroll down through the text. Other options, such as changing the characteristics of the histogram, are described in online Help.

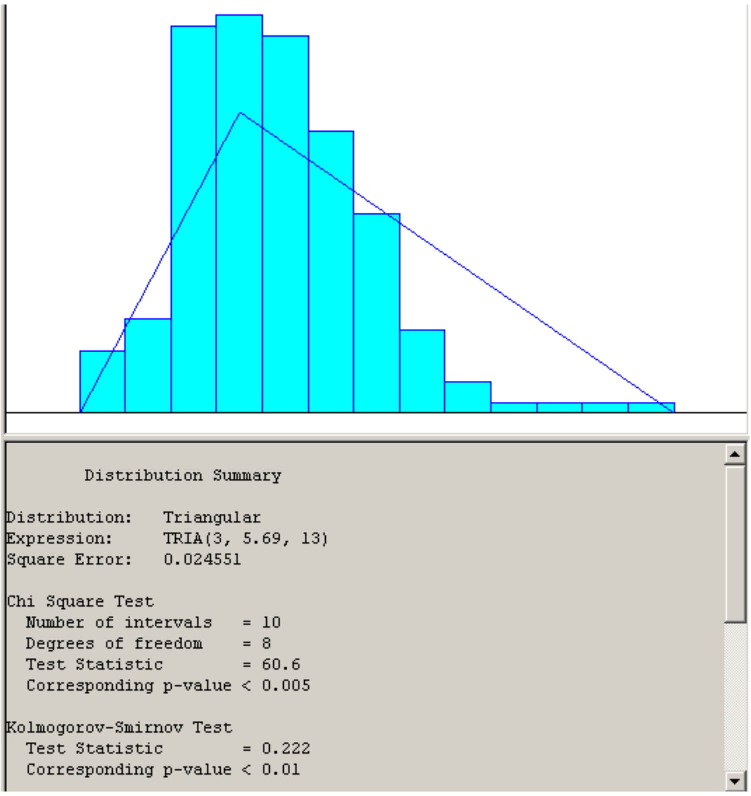
The Input Analyzer's Fit menu provides options for fitting individual probability distributions to the data (that is, estimating the required parameters for ~ a given distribution). After you fit a distribution, its density function is drawn on top of the histogram display and a summary of the characteristics of the fit is displayed in the text section of the window. (This information also is written to a plain ASCII text file named DISTRIBUTION.OUT, where DISTRIBUTION indicates the distribution you chose to fit, such as TRIANGLE for triangular.) The exact expression required to represent the data in Arena is also given in the text window. You can transfer this to Arena by selecting *Edit > Copy Expression* in the Input Analyzer, opening the appropriate dialog box in Arena, and pasting the expression (*Ctrl+V*) into the desired field. *Figure 4.24* shows this for fitting a triangular distribution to the data in *partbprp.dst*. Though we'll go into the goodness-of fit issue below, it's apparent from the plot in *Figure 4.24* that the triangular distribution doesn't fit the data particularly well.

If you plan to use a theoretical distribution in your model, you may want to start by selecting *Fit > Fit All*. This automatically fits all of the applicable distributions to the data, calculates test statistics for each (discussed below), and displays the distribution that has the minimum square error value (a measure of the quality of the distribution's match to the data). *Figure 4.25* shows the results of the Fit All option for our *Part B Prep* times and indicates that a gamma distribution with (β= 0.775 and α = 4.29, shifted to the right by 3, provides the "best" fit in the sense of minimum square error.



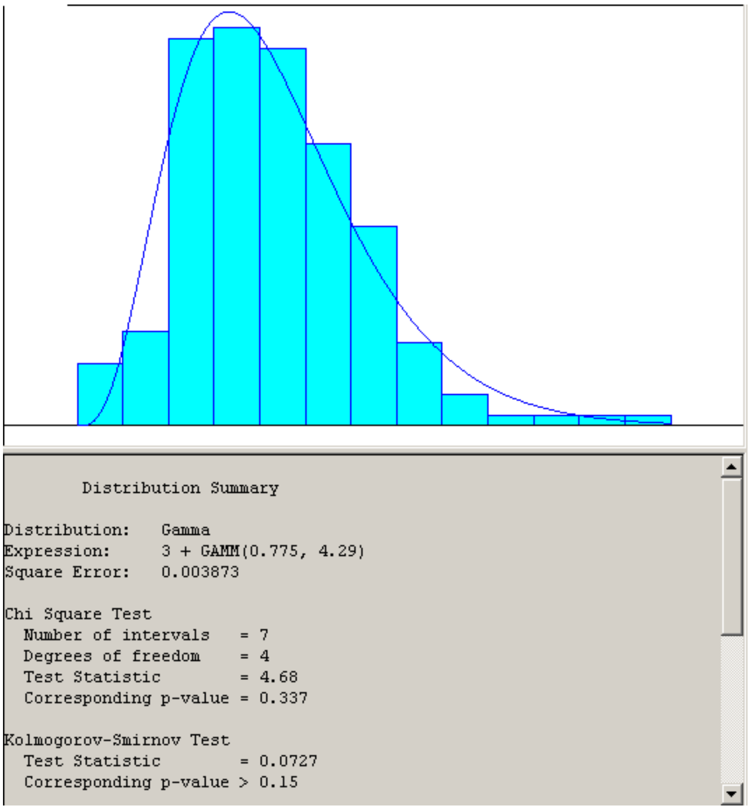
```
        Data Summary

Number of Data Points   = 187
Min Data Value          = 3.2
Max Data Value          = 12.6
Sample Mean             = 6.33
Sample Std Dev          = 1.51

        Histogram Summary

Histogram Range         = 3 to 13
Number of Intervals     = 13
```

Figure 4.23. Histogram and Summary of partbprp.dst



```
       Distribution Summary

Distribution:   Triangular
Expression:     TRIA(3, 5.69, 13)
Square Error:   0.024551

Chi Square Test
  Number of intervals   = 10
  Degrees of freedom    = 8
  Test Statistic        = 60.6
  Corresponding p-value < 0.005

Kolmogorov-Smirnov Test
  Test Statistic        = 0.222
  Corresponding p-value < 0.01
```

Figure 4.24. Fitting a Triangular Distribution to partbprp.dst



```
       Distribution Summary

Distribution:   Gamma
Expression:     3 + GAMM(0.775, 4.29)
Square Error:   0.003873

Chi Square Test
  Number of intervals   = 7
  Degrees of freedom    = 4
  Test Statistic        = 4.68
  Corresponding p-value = 0.337

Kolmogorov-Smirnov Test
  Test Statistic        = 0.0727
  Corresponding p-value > 0.15
```

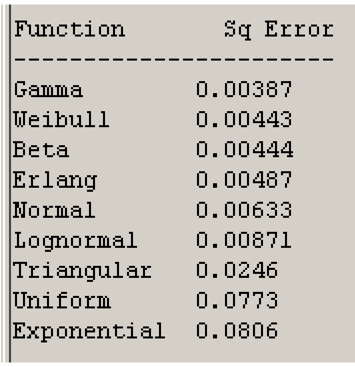
Figure 4.25. Fit All Option to partbprp.dst

Comparing the plot with that in *Figure 4.24* indicates that this fitted gamma distribution certainly appears to be a better representation of the data than did the fitted triangular distribution. Other considerations for selecting which theoretical distribution to use in your model are discussed below.

If you want to use a discrete or continuous empirical distribution, use the Empirical option from the Fit menu. You may first want to adjust the number of histogram cells, which determines how many probability/value pairs will be calculated for the empirical distribution. To do so, select Options > Parameters > Histogram and change the number of intervals.

In addition to "eyeballing" the fitted densities on top of the histograms, the Input Analyzer provides three numerical measures of the quality of fit of a distribution to the data to help you decide. The first, and simplest to understand, is the mean square error. This is the average of the square error terms for each histogram cell, which are the squares of the differences between the relative frequencies of the observations in a cell and the relative frequency for the fitted probability distribution function over that cell's data range. The larger this square error value, the further away the fitted distribution is from the actual data (and thus the poorer the fit). If you fit all applicable distributions to the data, the Fit All Summary table orders the distributions from smallest to largest square error (select *Window > Fit All Summary*), shown in *Figure 4.26* for *partbprp.dst*. While we see that gamma won the square-error contest, it was followed closely by Weibull, beta, and Erlang, any of which would probably be just as accurate as gamma to use as input to the model.

The other two measures of a distribution's fit to the data are the chi-square and Kolmogorov-Smirnov (K-S) goodness-of fit hypothesis tests. These are standard statistical hypothesis tests that can be used to assess whether a fitted theoretical distribution is a good fit to the data. The Input Analyzer reports information about the tests in the text window (see the bottoms of *Figures 4.24* and *4.25*). Of particular interest is the *Corresponding p-value*, which will always fall between 0 and 1[*]. To interpret this, larger p-values indicate better fits. Corresponding p-values less than about 0.05 indicate that the distribution's not a very good fit. Of course, as with any statistical hypothesis test, a high p-value doesn't constitute "proof" of a good fit just a lack of evidence against the fit.



```
Function       Sq Error
-----------------------
Gamma          0.00387
Weibull        0.00443
Beta           0.00444
Erlang         0.00487
Normal         0.00633
Lognormal      0.00871
Triangular     0.0246
Uniform        0.0773
Exponential    0.0806
```

Figure 4.26: Fit All Summary for partbprp.dst

When it comes down to fitting or choosing a distribution to use, there's no rigorous, universally agreed-upon approach that you can employ to pick the "best" distribution. Different statistical tests (such as the K-S and chi-square) might rank distributions differently, or changes in the preparation of the data (e.g., the number of histogram cells) might reorder how well the distributions fit.

Your first critical decision is whether to use a theoretical distribution or an empirical one. Examining the results of the K-S and chi-square tests can be helpful. If the p-values for one or more distributions are fairly high (e.g., 0.10 or greater), then you can use a theoretical distri-

[*] More precisely, the p-value is the probability of getting a data set that's more inconsistent with the fitted distribution than the data set you actually got, if the fitted distribution is truly "the truth: '

bution and have a fair degree of confidence that you're getting a good representation of the data (unless your sample is quite small, in which case the discriminatory power of goodness-of fit tests is quite weak). If the p-values are low, you may want to use an empirical distribution to do a better job capturing the characteristics of the data.

If you've decided to use a theoretical distribution, there often will be a number of them with very close goodness-of fit statistics. In this case, other issues may be worth considering for selecting among them:

> First of all, you may want to limit yourself to considering only bounded or unbounded distributions, based on your understanding of how the data will be used in your model. For instance, often the triangular (bounded) and normal (unbounded) distributions will be good fits to data such as process times. Over a long simulation run, they each might provide similar results, but during the run, the normal distribution might periodically return fairly large values that might not practically occur in the real system. A normal distribution with a mean that's close to zero may return an artificially large number of zero-valued samples if the distribution's being used for something that can't be negative, such as a time (Arena will change all negative input values to zero if they occur in a context where negative values don't make sense, like an interarrival or process time); this might be a compelling reason to avoid the normal distribution to represent quantities like process times that cannot be negative. On the other hand, bounding the triangular to obtain a faithful overall representation of the data might exclude a few outlying values that should be captured in the simulation model.
>
> Another consideration is of a more practical nature; namely, that it's easier to adjust parameters of some distributions than of others. If you're planning to make any changes to your model that include adjusting the parameters of the distribution (e.g., for sensitivity analysis or to analyze different scenarios), you might favor those with more easily understood parameters. For example, in representing interarrival times, Weibull and exponential distributions might provide fits of similar quality, but it's much easier to change an interarrival-time mean by adjusting an exponential mean than by changing the parameters of a Weibull distribution since the mean in the latter case is a complicated function of the parameters.
>
> If you're concerned about whether you're making the correct choice, run the model with each of your options to see if there's a significant difference in the results (you may have to wait until the model's nearly complete to be able to draw a good conclusion). You can further investigate the factors affecting distribution fits (such as the goodness-of fit tests) by consulting Arena's online Help or other sources, such as Pegden, Shannon, and Sadowski (1995) or Law and Kelton (2000). Otherwise, select a distribution based on the qualitative and practical issues discussed above.

Before leaving our discussion of the Input Analyzer, we'd like to revisit an issue we mentioned earlier, namely, what to do if your data appear to have multiple peaks or perhaps a few extreme values, sometimes called outliers. Either of these situations usually makes it impossible to get a decent fit from any standard theoretical distribution. As we mentioned before, one option is to use an empirical distribution, which is probably the best route unless your sample size is quite small. In the case of outliers, you should certainly go back to your data set and make sure that they're actually correct rather than just being some kind of error, perhaps just a typo; if data values appear to be in error and you can't backtrack to confirm or correct them, then you should probably just remove them.

In the case of either multiple peaks or (correct) outliers, you might want to consider dividing your data set into two (maybe more) subsets before reading them into the Input Analyzer. For example, suppose you have data on machine downtimes and notice from the histogram that there are two distinct peaks; i.e., the data are *bimodal*. Reviewing the original records, you discover that these downtimes resulted from two different situations-breakdowns and scheduled maintenance. Separating the data into two subsets reveals that downtimes resulting from breakdowns tend to be longer than downtimes due to scheduled maintenance, explaining the two peaks. You could then separate the data (before going into the Input Analyzer), fit separate distributions to these data sets, and then modify your model logic to account for both kinds of downtimes.

You can also do a different kind of separation of the data directly in the Input Analyzer, based purely on their range. After loading the data file, select *Options* > *Parameters* > *Histogram* to specify cutoffs for the Low Value and High Value, and these cutoffs will then define the bounds of a subset of your entire data set. For bimodal data, you'd focus on the left peak by leaving the Low Value alone and specifying the High Value as the point where the histogram bottoms out between the two peaks, and later focus on the right peak by using this cutpoint as the Low Value and using the original High Value for the entire data set; getting the right cutpoint could require some trial and error. Then fit whatever distribution seems promising to each subset (or use the Fit All option) to represent that range of the data. If you've already fit a distribution (or used Fit All) before making Low/High Value cut, the Input Analyzer will immediately re-do the fit(s) and give you the new results for the data subset included in your cut; if you've done Fit All, a different distribution form altogether might come up as "the best: ' You'd need to repeat this process for each subset of the data. As a practical matter, you probably should limit the number of subsets to two or three since this process can become cumbersome, and it's probably not obvious where the best cut-points are. To generate draws in your simulation model representing the original entire data set, the idea is to select one of your data subsets randomly, with the selection probabilities corresponding to the relative sizes of the subsets, then generate a value from the distribution you decided on for that subset. For instance, if you started out with a bimodal data set of size 200 and set your cut-point so that the smallest 120 points represented the left peak and the largest 80 points represented the right peak, you'd generate from the distribution fitted to the left part of the data with probability 0.6 and generate from the distribution fitted to the right part of the data with probability 0.4. This kind of operation isn't exactly provided automatically in a single Arena module, so you d have to put something together yourself. If the value involved is an activity time like a time delay an entity incurs, one possibility would be to use the **Decide** module with a Type of 2-way by Chance or N-way by Chance, depending on whether you have two versus more than two subsets to do a "coin flip" to decide from which subset to generate, then Connect to one of several **Assign** modules to Assign a value to an entity attribute as a draw from the appropriate distribution, and use this attribute downstream for whatever it's supposed to do. A different approach would be to set up an Arena Expression for the entire thing; Expressions are covered in Section 5.2.3.

4.5.5 No Data?

Whether you like it or not, there are times when you just can't get reliable data on what you need for input modeling. This can arise from several situations, like (obviously) the system doesn't exist, data collection is too expensive or disruptive, or maybe you don't have the cooperation or clearance you need. In this case, you'll have to rely on some fairly arbitrary assumptions or guesses, which we dignify as "ad hoc data." We don't pretend to have any great solutions for you here, but have a few suggestions that people have found useful. No matter what you do, though, you really should carry out some kind of sensitivity analysis of the output to these ad hoc inputs to have a realistic idea of how much faith to put in your results. You'll either need to pick some deterministic value that you'll use in your study (or run the model a number of times with different values), or you'll want to represent the system characteristic using a probability distribution.

If the values are for something other than a time delay, such as probabilities, operating parameters, or physical layout characteristics, you can either select a constant, deterministic value or, in some cases, use a probability distribution. If you use a deterministic value by entering a constant in the model (e.g., 15% chance of failing inspection), it's a good idea to perform some sensitivity analysis to assess what effect the parameter has on the model's results. If small changes to the value influence the performance of the system, you may want to analyze the system explicitly for a range of values (maybe small, medium, and large) rather than just for your best guess.

If the data represent a time delay, you'll almost surely want to use a probability distribution to capture both the activity's inherent variability as well as your uncertainty about the value itself. Which distribution you'll use will be based on both the nature of the activity and the type of data you have. When you've selected the distribution, then you'll need to supply the proper parameters based on your estimates and your assessment of the variability in the process.

Table 4.2

Possible No-Data Distributions

Distribution	Parameters	Characteristics	Example Use
Exponential	Mean	High variance Bounded on left Unbounded on right	Interarrival times Time to machine failure (constant failure rate)
Triangular	Min, Mode, Max	Symmetric or non-symmetric Bounded on both sides	Activity times
Uniform	Min, Max	All values equally likely Bounded on both sides	Little known about process

For selecting the distribution in the absence of empirical data, you might first look at the exponential, triangular, normal, and uniform distributions. The parameters for these distributions are fairly easy to understand, and they provide a good range of characteristics for a range of modeling applications, as indicated in *Table 4.2*.

If the times vary independently (i.e., one value doesn't influence the next one), your estimate of the mean isn't too large, and there's a large amount of variability in the times, the exponential distribution might be a good choice. It's most often used for interarrival times; examples would be customers coming to a restaurant or pick requests from a warehouse. .

If the times represent an activity where there's a "most likely" time with some variation around it, the triangular distribution is often used because it can capture processes with small or large degrees of variability and its parameters are fairly easy to understand. The triangular distribution is defined by minimum, most likely (modal), and maximum values, which is a natural way to estimate the time required for some activity. It has the advantage of allowing a non-symmetric distribution of values around the most likely, which is commonly encountered in real processes. It's also a bounded distribution-no value will be less than the minimum or greater than the maximum-which may or may not be a good representation of the real process.

You may be wondering why we're avoiding what might be the most familiar distribution of all, the normal distribution which is the classical "bell curve" defined by a mean and standard deviation. It returns values that are symmetrically distributed around the mean and is an unbounded distribution, meaning that you could get a very large or very small value once in a while. In cases where negative values can't be used in a model (e.g., the delay time in a process), negative samples from a normal distribution are set by Arena to a value of 0; if the mean of your distribution is close to 0 (e.g., no more than about three or four times the standard deviation from 0), the normal distribution may be inappropriate.

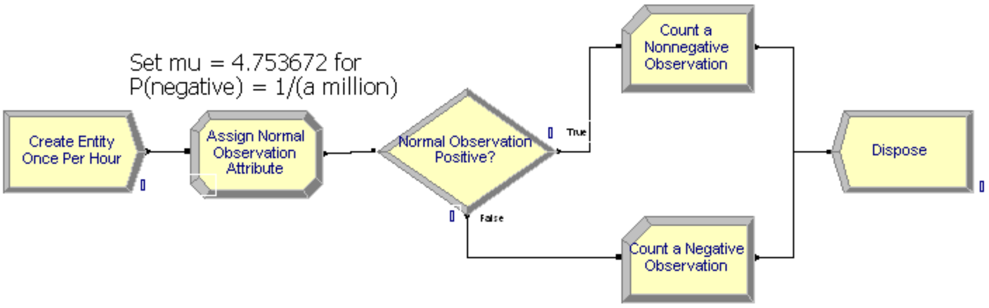

Set mu = 4.753672 for
P(negative) = 1/(a million)
Create Entity Once Per Hour
Assign Normal Observation Attribute
Normal Observation Positive?
True
False
Count a Nonnegative Observation
Count a Negative Observation
Dispose

Figure 4.27. Model 4-5 to Count Negative Normal Observations

Table 4.3

Getting Negative Values from a Normal Distribution with Standard Deviation σ= 1

Mean	Number of Draws	Number of Negative Draws	Exact Probability That an Observation Will Be Negative	"One in This Many" Will Be Negative (on Average
3.0	One million	1,409	0.001350	741
3.5	One million	246	0.000233	4,298
4.0	One million	37	0.000032	31,560
4.5	One million	3	0.000003	294,048
4.753672	Ten million	7	0.000001	1,000,000

On the other hand, if the mean is positive and quite a bit larger than the standard deviation, there will be only a small chance of getting a negative value, like maybe one in a million. This sounds (and is) small, but remember that in a long simulation run, or one that is replicated many times, one in a million can really happen, especially with modern, fast computers; and in that case, Arena would truncate the negative value to zero if its usage in the model can only be positive, perhaps causing an unwanted or extreme action to be taken and possibly even invalidating your results. *Figure 4.27* shows a fairly useless little Arena model (Model 4-5) in which entities arrive, spaced exactly an hour apart, an observation from a normal distribution with mean μ=3.0 and standard deviation σ=1 is assigned to the attribute Normal Observation, and the entity goes to one of two **Record** modules, depending on whether Normal Observation is non-negative or negative, which count the number of non-negative and negative observations obtained out of the total (we'll leave it to you to look through this model on your own). *Table 4.3* gives the results for several values of μ (holding σ at 1), and you can see that this is going to occur even if the mean is three or four (or more) standard deviations above zero. Using electronic normal tables, the exact probability of getting a negative value can be computed, and one over this number gives the approximate number of observations (on average) you'd need to get a negative value-not too many, considering the speed of generating these (it took about six seconds on a garden-variety 2GHz notebook computer to complete each of the million-draw runs, and under a minute to complete the ten-million-draw run). The last value of in the table was picked since it yields a probability of exactly one in a million of a getting a negative observation (the first million happened not to produce any negatives, but seven in ten million is close enough to one in a million). Now we respect and admire Gauss as a great mathematician, but we're just not big fans of using the normal as a simulation input distribution to represent logically non-negative things like process times, even though Arena allows it. For data sets that appear to be well fit (or even best fit) by a normal distribution, a different distribution that completely avoids negative values, like Weibull, gamma, lognormal, Erlang, beta, or perhaps empirical, will probably fit almost as well and not expose you to the risk of generating any naughty negative values at all.

Finally, if you really don't know much about the process but can guess what the minimum and maximum values will be, you might use the uniform distribution. It returns all values between a minimum and maximum with equal likelihood.

4.5.6 Non-stationary Arrival Processes

This somewhat specialized topic deserves mention on its own since it seems to come up often and can be very important in terms of representing system behavior validly. Many systems subject to external arrivals, like service systems for people, telephone call centers, and manufacturing systems with outside customer demands, experience arrival loads that can vary dramatically over the time frame of the simulation. Examples include a noon rush for burgers, heavy calls to a technical support line in the middle of the afternoon, and strong demand on a manufacturing system during certain seasons. A specific probabilistic model for this, the non-stationary Poisson process, is very useful and often provides an accurate way to reflect time-varying arrival patterns. You need to deal with two issues: how to estimate or specify the rate function, and then how to generate the arrival pattern in the simulation.

There are a lot of ways to estimate or specify a rate function from the data, some of which can be pretty complicated. We'll stick to one fairly simple method called the piecewise-constant rate function, which seems to work well in many applications. First, identify lengths of time within which the arrival rate appears to be fairly flat; for instance, a call center's arrivals might be fairly constant over half hour periods but could be quite different in different periods. Count up the numbers of arrivals in each period, and then compute a different rate for each period. For instance, suppose the call center experiences the following numbers of incoming calls for the four 30-minute periods between 8:00 am and 10:00: 20, 35, 45, and 50. Then the rates, in units of calls per minute, for the first four 30-minute periods would be 0.67, 1.17, 1.50, and 1.67.

Once you've estimated the rate function in this way, you need to make sure that Arena follows this pattern in generating the arrivals to your model. The **Create** module will do this if you select Schedule as the Type for Time Between Arrivals. You then must specify the rate function via the **Schedule** data module, similarly to what we did in Section 4.2.2 for a Resource Schedule. One caution here: Arena allows you to mix and match whatever time units you want, but you must be careful that the numbers and time units are defined properly. We'll do this in Model 5-2 in Chapter 5. In Chapter 12, we'll have more to say about estimating the rate function, as well as what underlies Arena's generation method for non-stationary arrivals.

4.5.7 Multivariate and Correlated Input Data

Most of the time we assume that all random variables driving a simulation are generated independently of each other from whatever distribution we decide on to represent them. Sometimes, though, this may not be the best assumption, for purely physical reasons. For instance, in Model 4-2 you could imagine that certain parts are "difficult"; maybe you'd detect this by noticing that a large prep time for a specific part tends to be followed by a large sealer time for that part; that is, these two times are positively correlated. Ignoring this correlation could lead to an invalid model and biased results.

There are a number of ways to model situations like this, to estimate the required parameters (including the strength of the correlations), and to generate the required observations on the random variables during the simulation. Some of these methods entail viewing the associated random variables as coordinates of a random vector having some joint multivariate distribution to be fitted and generated from. You might also be able to specify some kind of formula-based association between related input quantities. Frankly, though, this is a pretty difficult issue in terms of both estimating the behavior and generating it during the simulation. For more on these and related issues, see Law and Kelton (2000) or Devroye (1986).

4.6 Chapter summary

If you've read and understood the material in this chapter, you should be getting dangerous in the use of Arena. We encourage you to press other buttons in the modules we've used and even try modules that we've not used. If you get stuck, try the online Help feature, which may not answer your question, but will answer the questions you should be asking. You might also want to try other animation features or provide nicer pictures. At this point, the best advice we can give you is to use Arena. Chapters 5-12 will cover most of the modeling capabilities (and some of the statistical-analysis capabilities) of Arena in more depth.

CHAPTER 5

Modeling Packaging Lines

Manufacturing facilities in many of industrial sectors are structured as a series of production stages. Jobs in a variety of forms are transferred from one stage to another to be processed in a prescribed order; these jobs eventually leave the system as finished or semifinished products. Such systems are referred to as **production lines**.

The flexibility afforded by computer-controlled machinery enables production lines to handle a broad range of operations. Various operations permit the deployment of a sequence of intelligent workstations on the shop floor for processing or assembling various products. This chapter discusses simulation modeling of workstation sequences in the packaging line context.

5.1 Production lines

Consider the representation of a generic manufacturing facility depicted in a rather abstract form in *Figure 5.1*. The manufacturing facility is a production line composed of manufacturing stages consisting of workstations with intervening buffers to hold product flowing along the line. Each workstation contains one or more machines, one or more operators (possibly robots), and a work-in-process (WIP) buffer. Upon process completion at a workstation, departing jobs join the WIP buffer at the next workstation, provided that space is available; otherwise, such jobs are typically held at the current workstation until space becomes available in the next buffer.

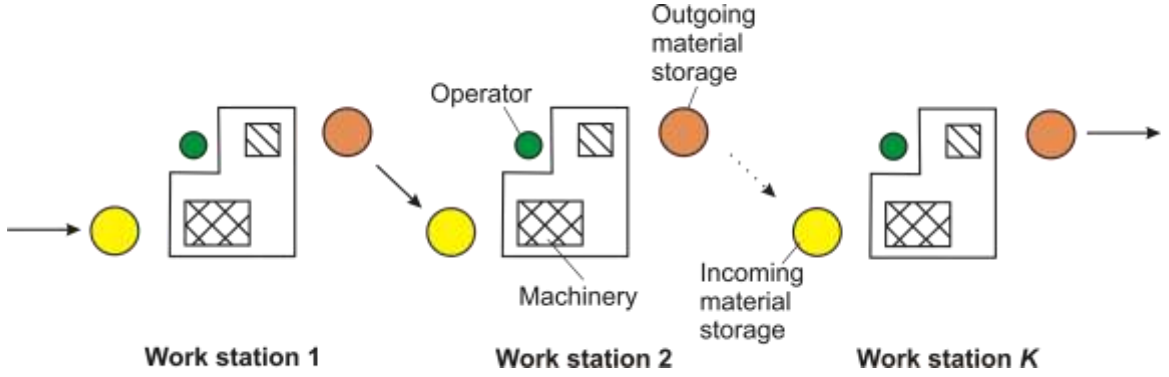


Figure 5.1. A generic production line

Many practical models may be formulated as variations on the generic production line of *Figure 5.1***,** or with additional wrinkles. For example, a model may call for one or more repetitions of a certain process or a set of processes, or some of the workstations may process jobs in batches. In other cases, the transfer of jobs from one workstation to another is of central importance, so that transportation via vehicles or conveyors is modeled in some detail (see Chapter 13 for a detailed treatment of this subject). Eventually, jobs depart from the system, and such departures can occur, in principle, from any workstation. In general, production lines employ a **push regime**, where little attention is given to the finished-product inventory. The manufacturing line simply produces (pushes) as much product as possible under the assumption that all finished products are to be used. Otherwise, when the accumulation of finished-products becomes excessive, the manufacturing line may stop producing, at which point the push regime is switched to a **pull regime** (the process only produces in response to specific demands; see the examples in Chapter 12). Push and pull types of manufacturing systems are studied in detail in Altiok (1997) and Buzacott and Shanthikumar (1993).

More generally, storage limitations in workstations give rise to a bottleneck phenomenon, involving both **blocking** and **starvation**. The sources of this phenomenon are space limitations and cost considerations, which impose explicit or implicit target levels for storage between stages in a production line. Space limitations (finite buffers) in a downstream workstation can, therefore, cause stoppages at upstream workstations– a phenomenon known as blocking. Blocking policies differ on the exact timing of stoppage. One policy calls for an immediate halt of processing of new jobs in the upstream workstation as soon as the downstream buffer becomes full. The upstream workstation is then forced to be idle until the downstream buffer has space for another job, at which point the upstream workstation resumes processing. This type of blocking is often called **communications blocking**, since it is common in communications systems (see Chapter 14). Another policy calls for processing the next job, but holding it (on completion) in the upstream machine until the downstream buffer can accommodate a new job. This type of blocking policy is called **production blocking**, since it often occurs in manufacturing context. Various types of blocking mechanisms are discussed in detail in Perros (1994).

It is important to realize that blocking tends to propagate backwards to successive workstations located upstream in the production line. Similarly, some workstations may experience idleness due to lack of job flow from upstream workstations. This phenomenon is called starvation for obvious reasons. It is further important to realize that starvation tends to propagate forward to successive workstations located downstream in the production line. Blocking and starvation are, in fact, the flip sides of a common phenomenon and tend to occur together—a **bottleneck workstation** can be identified, which separates a production line into two segments, such that upstream workstations experience frequent blocking and downstream workstations experience frequent starvation.

Another type of (forced) idleness in production lines is caused by failures. Usually, a failed workstation is taken in for repair as soon as possible and resumes operation once repair is completed; the alternating periods of operation and failure are referred to as **uptimes** and **downtimes**, respectively. Clearly, a failed workstation can become a bottleneck workstation, causing blocking in upstream workstations and starvation in downstream workstations. In the presence of failures, one needs to devise procedures for handling interrupted jobs (those being processed when a failure strikes). For instance, an interrupted job may need to be reprocessed from scratch or may merely need to resume processing. In some cases, however, the interrupted job is simply discarded.

5.2 Models of production lines

Productivity losses are potentially incurred, whenever machines are idle (blocked or starved) due to machine failures or bottlenecks stemming from excessive accumulation of inventories between workstations. Furthermore, production lines are rarely deterministic; their randomness is due to variable processing times, as well as random failures and subsequent repairs. Such randomness makes it difficult to control these systems or to predict their behavior. A mathematical model or a simulation model is then used to make such predictions.

Design problems in production lines are primarily resource allocation problems. These problems include **workload allocation** and **buffer capacity allocation** for a given set of workstations with associated processing times. Generally, design problems are quite difficult to solve in manufacturing systems. This is due in part to the combinatorial nature of such problems.

Performance analysis of production lines strives to evaluate their performance measures as function of a set of system parameters. The most commonly used performance measures follow:

Throughput.
Average inventory levels in buffers.
Downtime probabilities.
Blocking probabilities at bottleneck workstations.
Average system flow times (also called manufacturing lead times).

Using these measures to analyze manufacturing systems can reveal better designs by identifying areas where loss of productivity is most harmful. For a thorough coverage of design, planning, and scheduling problems in production and inventory systems, see Altiok (1997), Askin and Standridge (1993), Buzacott and Shanthikumar (1993), Elsayed and Boucher (1985), Gershwin (1994), Johnson and Montgomery (1974), and Papadopoulos et al. (1993), among others.

5.3 A packaging line

Consider a generic packaging line for some product, such as a pharmaceutical plant producing a packaged medicinal product, or a food processing plant producing pack-aged foods or beverages. The line consists of workstations that perform the processes of *filling*, *capping*, *labeling*, *sealing*, and *carton packing*. Individual product units will be referred to simply as units.

We make the following assumptions:

(1) The filling workstation always has material in front of it, so that it never starves.
(2) The buffer space between workstations can hold at most five units.
(3) A workstation gets blocked if there is no space in the immediate downstream buffer (manufacturing blocking).
(4) The *processing times* for filling, capping, labeling, sealing, and carton packing are 6.5, 5, 8, 5, and 6 seconds, respectively.

Note that these assumptions render our packaging line a push-regime production line. To keep matters simple, no randomness has been introduced into the system, that is, our packaging line is deterministic.

It is worthwhile to elaborate and analyze the behavior of the packaging line under study. The first workstation (filling) drives the system in that it feeds all downstream workstations with units. Clearly, one of the workstations in the line is the slowest (if there are several slowest workstations, we take the first among them). The throughput (output rate) of that workstation then coincides with the throughput of the entire packaging line. Furthermore, workstations upstream of the slowest one will experience excessive buildup of WIP inventory in their buffers. In contrast, workstations down-stream of the slowest one will always have lightly occupied or empty WIP inventory buffers. Thus, the slowest workstation acts as a bottleneck in our packaging line. Of course, this behavior holds for any deterministic push-regime production line.

5.3.1. An Arena model

Figure 5.2. depicts an Arena model for the packaging line, in which Arena modules represent each process and units are transferred from process to process. The model logic strives to maintain a sufficient number of units (Arena entities) in the filling station, so as to keep it busy for as long as possible. To achieve this goal, a total of 30 unit entities are created at time 0, and their arrival time attribute, *ArrTime*, is assigned the current value of *TNOW* in the *As-*

sign Arrival Times module. These are promptly fed into the filling process buffer (still at time 0). Prior to exiting the system from the *Tally* module, called *Interdeparture Time*, unit entities are not disposed of, but are sent back to the *Assign Arrival Times* module. Note carefully that this is merely a modeling device to avoid starvation in the filling station, in compliance with the first assumption above. Starting with the filling process, unit entities go through the modules associated with the filling, capping, labeling, sealing, and packing processes in that order, and finally proceed to statistics collection modules. The model logic will be discussed in some detail in the next section.

5.3.2 Model 5-1. Manufacturing process modules

Each packaging line process is modeled in *Figure 5.2.* by a **Process** module from the Basic Process template panel. Every process was declared to have a queue in front of it, as evidenced by the T-bar graphics in *Figure 5.2*. Recall that queue parameters can be examined in spreadsheet view in the **Queue** module from the Basic Process template panel. As an example, consider the **Process** module called *Filling Process*, whose dialog box is displayed in *Figure 5.3*. This **Process** module has a queue called *Filling Process*.*Queue* in front of a resource called *Filler*. The Action field in the *Filling Process* dialog box is *Seize Delay*, so that unit entities wait in the queue *Filling Process.Queue* whenever resource Filler is busy. As soon as resource Filler becomes available, it is seized by the first unit entity (if any) in the queue *Filling Process.Queue*, and a processing time delay of 6.5 seconds is then implemented in the associated *DELAY* block.

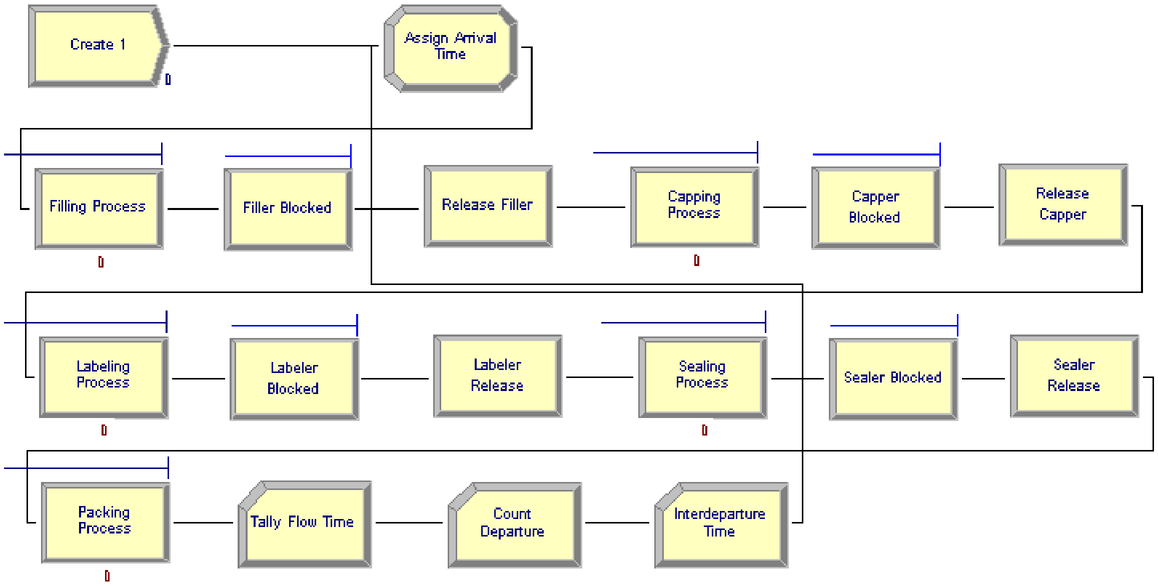


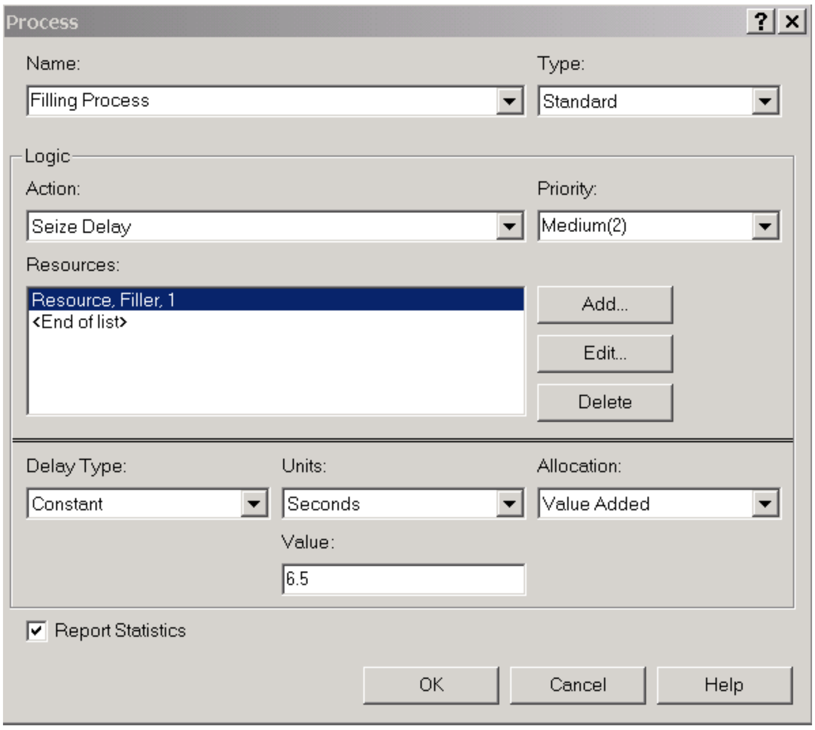
Figure 5.2. Arena model for the packaging line system.

On processing completion, the unit entity proceeds to the next **Hold** module, called *Filler Blocked*, which implements blocking as necessary (this is explained in detail in Section 5.3.3). Only when space is available in the capping process that follows the filling process, will the unit entity be permitted to proceed to the **Release** module, called *Release Filler*, where resource Filler is released, thereby completing the filling process. In fact, the sequence of Process, Hold, and Release modules is repeated for each of the first four processes (the fifth and last process, *Packing Process*, does not experience blocking).

5.3.3 Model blocking using the Hold module

Recall that the **Hold** module is used to hold entities in an associated queue until a condition is satisfied or until a future event takes place. In particular, such modules may be used to implement production blocking.

The Arena model of *Figure 5.2.* employs four **Hold** modules to this end (observe the corresponding queue T-bar graphics in *Figure 5.2.*) As an example, *Figure 5.4.* displays the dialog box for the **Hold** module, called *Filler Blocked*, associated with the *Filling Process* module.



Name: Filling Process

Type: Standard

Logic

Action: Seize Delay

Priority: Medium(2)

Resources:
Resource, Filler, 1
<End of list>

Delay Type: Constant

Units: Seconds

Allocation: Value Added

Value: 6.5

☑ Report Statistics

Figure 5.3. Dialog box of the **Process** module *Filling Process*.

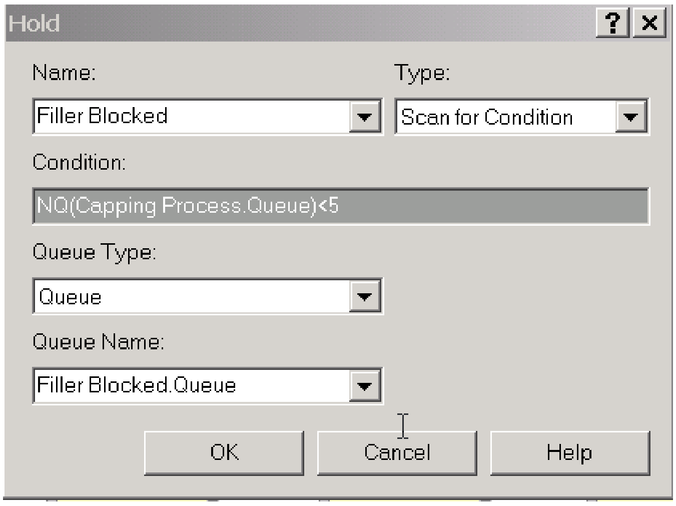
The *Condition* field in the **Hold** module dialog box was set to scan for the condition

$$NQ(Capping\ Process.Queue)<5.$$

In other words, it tests whether buffer space is available in the capping process—the manufacturing stage that immediately follows the filling process. If this condition is true, the corresponding unit entity leaves the filling process and joins the capping buffer. Otherwise, that unit entity becomes blocked and remains in the **Hold** module, or more specifically, in the associated queue (in our example, *Filler Blocked.Queue*) until the condition is satisfied. As soon as the unit entity leaves the **Hold** module, it enters the **Release** module, called *Release Filler*, and joins queue *Capping Process.Queue*. Clearly, the resource Filler must perforce be busy whenever a unit entity resides in the queue *Filler Blocked.Queue*. Furthermore, since the capacity of resource *Filler* was declared to be one unit, there can be at most one unit entity in the queue *Filler Blocked.Queue* at any point in time. It follows that

$$\Pr\{NQ(Filler\,Blocked.Queue) > 0\} = \frac{\text{Time } \mathit{Filler\ Blocked.Queue} \text{ is occupied}}{\text{Total simulation time}},$$

which also happens to be the average number of unit entities in *Filler Blocked.Queue*. This fact will later be used in obtaining blocking probabilities for any process *some_process* simply by defining (in the **Statistic** module) a *Time Persistent* statistic for the expression *NQ*(*some process.Queue*).



Hold

Name: Filler Blocked

Type: Scan for Condition

Condition: NQ(Capping Process.Queue)<5

Queue Type: Queue

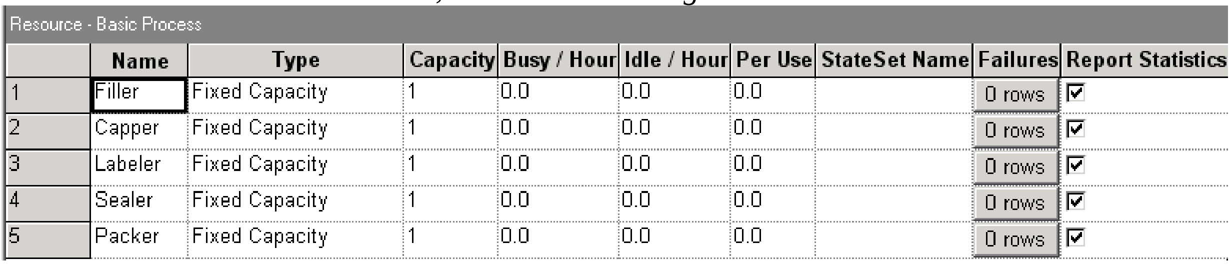
Queue Name: Filler Blocked.Queue

Figure 5.4. Dialog box of the **Hold** module associated with the filling process.

Note that a **Release** module follows each **Hold** module for each process. Consequently, the time spent by a unit entity in the **Hold** module's blocking queue is measured by Arena as part of the preceding **Process** module's resource busy time. This point will be revisited in Section 5.3.6 when we study the model's effective process utilizations.

5.3.4 Resources and queues

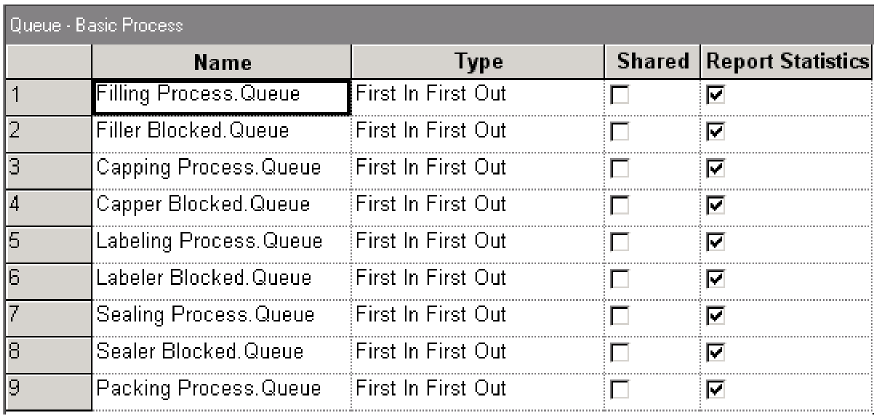
Recall that the **Resource** module in the Basic Process template panel provides a spreadsheet view of all resources in the model, as illustrated in *Figure 5.5*.



Resource - Basic Process

	Name	Type	Capacity	Busy / Hour	Idle / Hour	Per Use	StateSet Name	Failures	Report Statistics
1	Filler	Fixed Capacity	1	0.0	0.0	0.0		0 rows	☑
2	Capper	Fixed Capacity	1	0.0	0.0	0.0		0 rows	☑
3	Labeler	Fixed Capacity	1	0.0	0.0	0.0		0 rows	☑
4	Sealer	Fixed Capacity	1	0.0	0.0	0.0		0 rows	☑
5	Packer	Fixed Capacity	1	0.0	0.0	0.0		0 rows	☑

Figure 5.5. Dialog spreadsheet of the **Resource** module.



Queue - Basic Process

	Name	Type	Shared	Report Statistics
1	Filling Process.Queue	First In First Out	□	☑
2	Filler Blocked.Queue	First In First Out	□	☑
3	Capping Process.Queue	First In First Out	□	☑
4	Capper Blocked.Queue	First In First Out	□	☑
5	Labeling Process.Queue	First In First Out	□	☑
6	Labeler Blocked.Queue	First In First Out	□	☑
7	Sealing Process.Queue	First In First Out	□	☑
8	Sealer Blocked.Queue	First In First Out	□	☑
9	Packing Process.Queue	First In First Out	□	☑

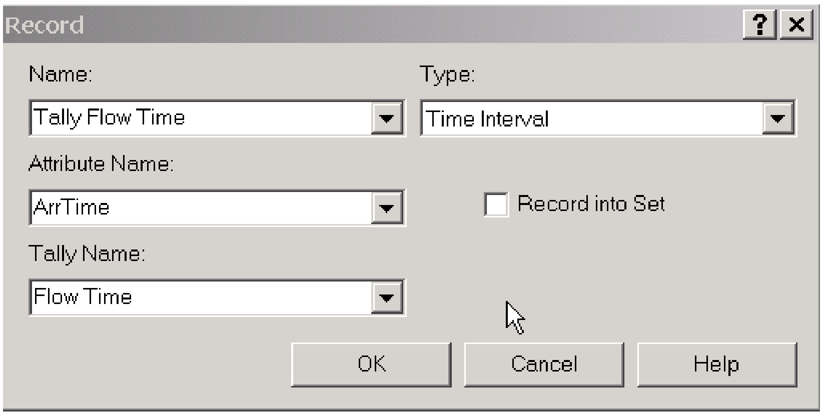
Figure 5.6. Dialog spreadsheet of the **Queue** module.

Similarly, the **Queue** module provides a spreadsheet view of all queues in the model, including blocking queues, as illustrated in *Figure 5.6*.

The column under the *Type* heading in *Figure 5.6.* is used to specify a service discipline for each queue via an option from a pull-down menu in each selected entry. Service discipline options are *FIFO*, *LIFO*, *Lowest Attribute Value*, and *Highest Attribute Value*. The last two options are implemented using a particular attribute of unit entities in the queue. A common example is priority-based service using a Priority attribute in entities. Another example is the *SPT* rule, which admits into service the job with the shortest processing time. An implementation of this rule would require the modeler to declare an attribute, say *Processing_Time*, and to select the options *Lowest Attribute Value* or *Highest Attribute Value* in the *Type* column. This selection would automatically create an additional column in *Figure 5.6*, labeled *Attribute Name*, in which the user would enter the appropriate attribute name (in our case, *Processing_Time*). Modeling a finite capacity for a queue can be implemented analogously to blocking (see Section 5.3.3). Entries in the column under the *Shared* heading are checked if a queue is shared by more than one **Seize** module or **SEIZE** block.

5.3.5 Statistics collection

Figure 5.7. displays the dialog box of the **Record** module, called *Tally Flow Time*, which tallies the time elapsed since the time stored in the unit entity attribute *ArrTime*. In particular, if we arrange for *ArrTime* to be a unit entity attribute that stores its arrival time in the system and set the Type field to the Time Interval option, and if unit entities proceed from the **Record** module *Tally Flow Time* to be disposed of, then this **Record** module will tally unit entity flow times through the system.



Name: Tally Flow Time

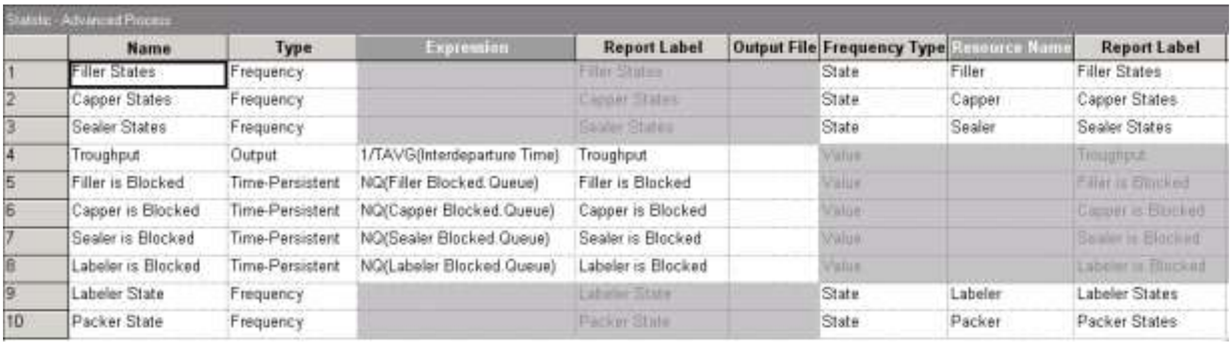
Type: Time Interval

Attribute Name: ArrTime

☐ Record into Set

Tally Name: Flow Time

Figure 5.7. Dialog box of the **Record** module *Tally Flow Time*.



Statistic - Advanced Process

	Name	Type	Expression	Report Label	Output File	Frequency Type	Resource Name	Report Label
1	Filler States	Frequency		Filler States		State	Filler	Filler States
2	Capper States	Frequency		Capper States		State	Capper	Capper States
3	Sealer States	Frequency		Sealer States		State	Sealer	Sealer States
4	Troughput	Output	1/TAVG(Interdeparture Time)	Troughput		Value		Troughput
5	Filler is Blocked	Time-Persistent	NQ(Filler Blocked.Queue)	Filler is Blocked		Value		Filler is Blocked
6	Capper is Blocked	Time-Persistent	NQ(Capper Blocked.Queue)	Capper is Blocked		Value		Capper is Blocked
7	Sealer is Blocked	Time-Persistent	NQ(Sealer Blocked.Queue)	Sealer is Blocked		Value		Sealer is Blocked
8	Labeler is Blocked	Time-Persistent	NQ(Labeler Blocked.Queue)	Labeler is Blocked		Value		Labeler is Blocked
9	Labeler State	Frequency		Labeler State		State	Labeler	Labeler States
10	Packer State	Frequency		Packer State		State	Packer	Packer States

Figure 5.8. Dialog spreadsheet of the **Statistic** module.

Figure 5.8. displays the dialog spreadsheet of the **Statistic** spreadsheet module for the packaging line model. Rows 1–4 and row 9 collect Frequency statistics (steady-state probability estimates) for the states of the filling, capping, sealing, packing, and labeling resources, respectively. The probability of any prescribed event can be similarly specified via an expression, provided that the Type field is set to the Time Persistent option. For example, the probability that the filling resource queue has two or more jobs in its buffer can be specified by the expression.

$$NQ\ (Filling\_R\_Q) >= 2.$$

Row 5 collects an *Output* type statistic specified by an expression that defines the reciprocal of the time average of the variable, called *Interdeparture Time,* which computes interdeparture times of unit entities from the system. Recalling that the expression is computed after the replication terminates, it follows that this *Output* statistic provides an estimate of the system throughput.

Finally, rows 6 through 10 collect blocking probabilities of the filling, capping, sealing, and labeling processes, respectively (recall that the packing process does not experience blocking). Note that each associated expression collects a time average of the size of the blocking queue in the corresponding **Hold** module. Since each blocking queue can hold at most one job, the computed time average is just the respective blocking probability.

5.3.6 Simulation output reports

The packaging line model was simulated for 100,000 seconds. *Figure 5.9.* displays the Queues report of the Reports panel for this model.

Note that each queue has Waiting Time and Number Waiting statistics, including the blocking queues of the **Hold** modules. The queue *Filling Process.Queue* had a maximum of 29 units (recall that 30 unit entities were placed there initially), and an average of 15.64 units. The queue *Capping Process.Queue* was nearly full most of the time, holding an average of 4.96 units with an average delay of 39.9 seconds. The labeling queue *Labeling Process.Queue* was also full most of the time holding an average of 4.99 units. In contrast, the sealing and packing queues were empty all the time. The *Queues* report also includes hold (blocking) queue statistics. These will be discussed, however, later in the context of blocking probabilities.

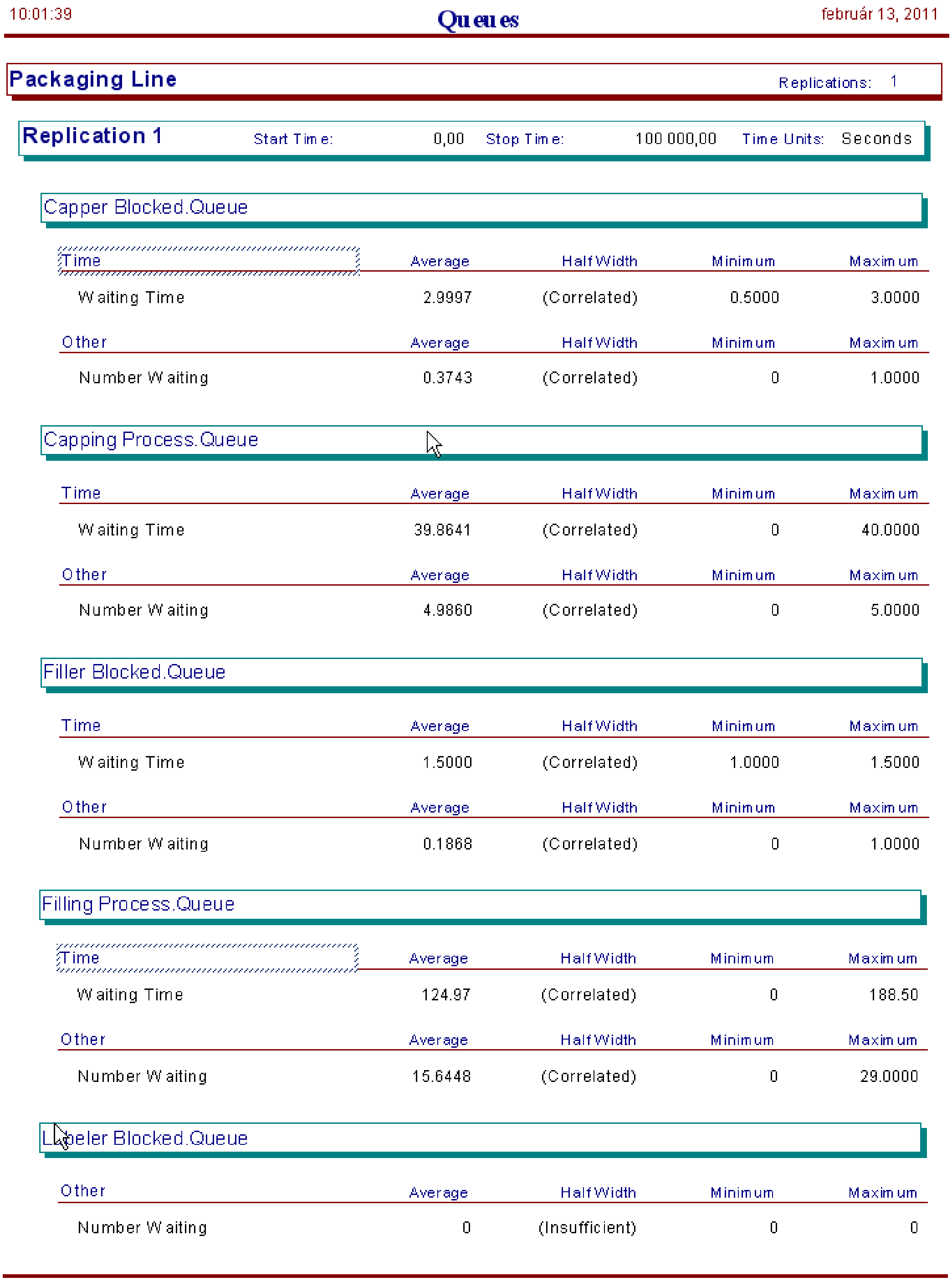
Figure 5.10. displays the Resources report of the Reports panel for the packaging line model. The Inst Util column in the report shows that the resources *Capper, Filler,* and *Labeler* were 100% occupied. This is expected of the *Filler* resource, since the system was designed that way. The *Capper* and *Filler* resources did not experience idleness, while the *Sealer* and *Packer* resources were busy with 75% and 62% utilization, respectively. The *Resources* report has additional detailed statistics similar to the *Queues* report, which are not shown here.

Figure 5.11. displays the *User Specified* report of the Reports panel for the packaging line model. Recall that these statistics are collected as a result of declarations in the **Statistic** and **Record** modules. The *Tally* section in *Figure 5.11.* provides interdeparture time and flow-time statistics. The mean interdeparture time for the run was estimated at 8 seconds (with all observations having the same value, 8). In spite of its apparent triviality, this statistic contributes to the verification of the simulation model—a topic that will be treated in more detail in Section 5.4. The mean flow time from the *Filling Process* module to the *Interdeparture Time* module was estimated at 239.78 seconds, based on 12,497 observations (unit entity departures). The minimal and maximal flow times observed were 30.5 seconds and 262.5 seconds, respectively. Note that the mean flow time is closer to the maximal rather than the minimal observation, showing that the majority of flow-time observations were skewed toward 262.5 seconds.

The Time Persistent section provides estimates for blocking probabilities. Evidently, the system experienced acute blocking in *Capping Process* (blocking occurred 37% of the time) due to long labeling times. Blocking at *Capping Process* has propagated upstream to *Filling Process*, which experiences 18.7% blocking.

The *Output* section estimates the system throughput as 0.125 units per second (equivalently, one unit every 8 seconds), in agreement with the mean interdeparture time in the *Tally* section.

The *Counter* section indicates that a total of 12,497 departures from the system were observed over the replication interval.



10:01:39 Queues február 13, 2011

Packaging Line Replications: 1

Replication 1 Start Time: 0,00 Stop Time: 100 000,00 Time Units: Seconds

Capper Blocked.Queue

Time	Average	Half Width	Minimum	Maximum
Waiting Time	2.9997	(Correlated)	0.5000	3.0000

Other	Average	Half Width	Minimum	Maximum
Number Waiting	0.3743	(Correlated)	0	1.0000

Capping Process.Queue

Time	Average	Half Width	Minimum	Maximum
Waiting Time	39.8641	(Correlated)	0	40.0000

Other	Average	Half Width	Minimum	Maximum
Number Waiting	4.9860	(Correlated)	0	5.0000

Filler Blocked.Queue

Time	Average	Half Width	Minimum	Maximum
Waiting Time	1.5000	(Correlated)	1.0000	1.5000

Other	Average	Half Width	Minimum	Maximum
Number Waiting	0.1868	(Correlated)	0	1.0000

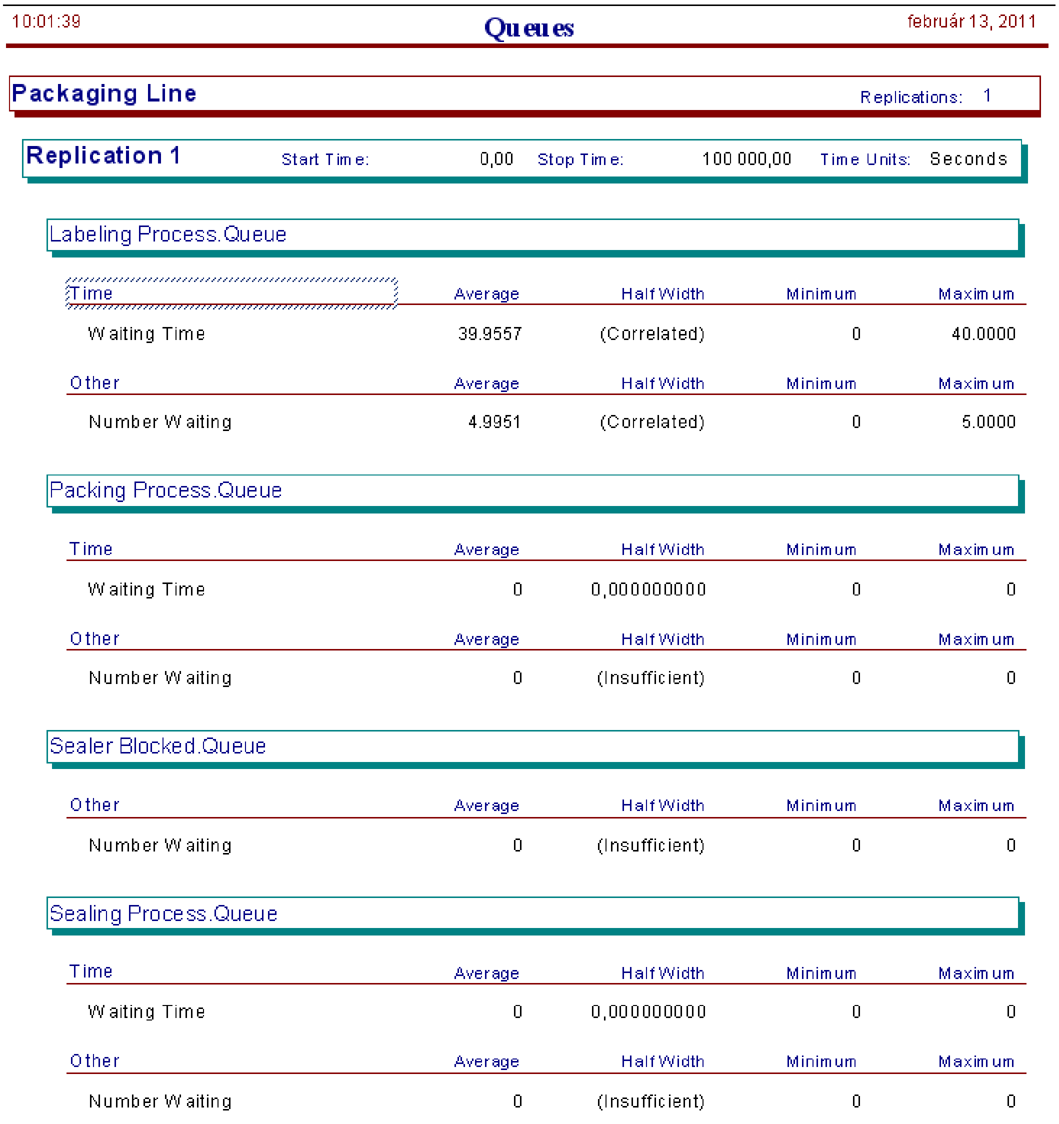
Filling Process.Queue

Time	Average	Half Width	Minimum	Maximum
Waiting Time	124.97	(Correlated)	0	188.50

Other	Average	Half Width	Minimum	Maximum
Number Waiting	15.6448	(Correlated)	0	29.0000

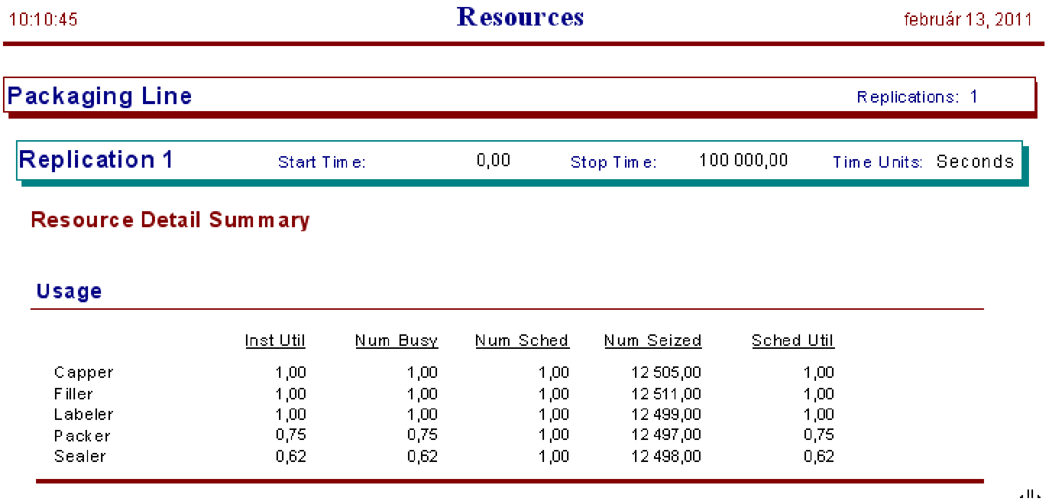
Labeler Blocked.Queue

Other	Average	Half Width	Minimum	Maximum
Number Waiting	0	(Insufficient)	0	0



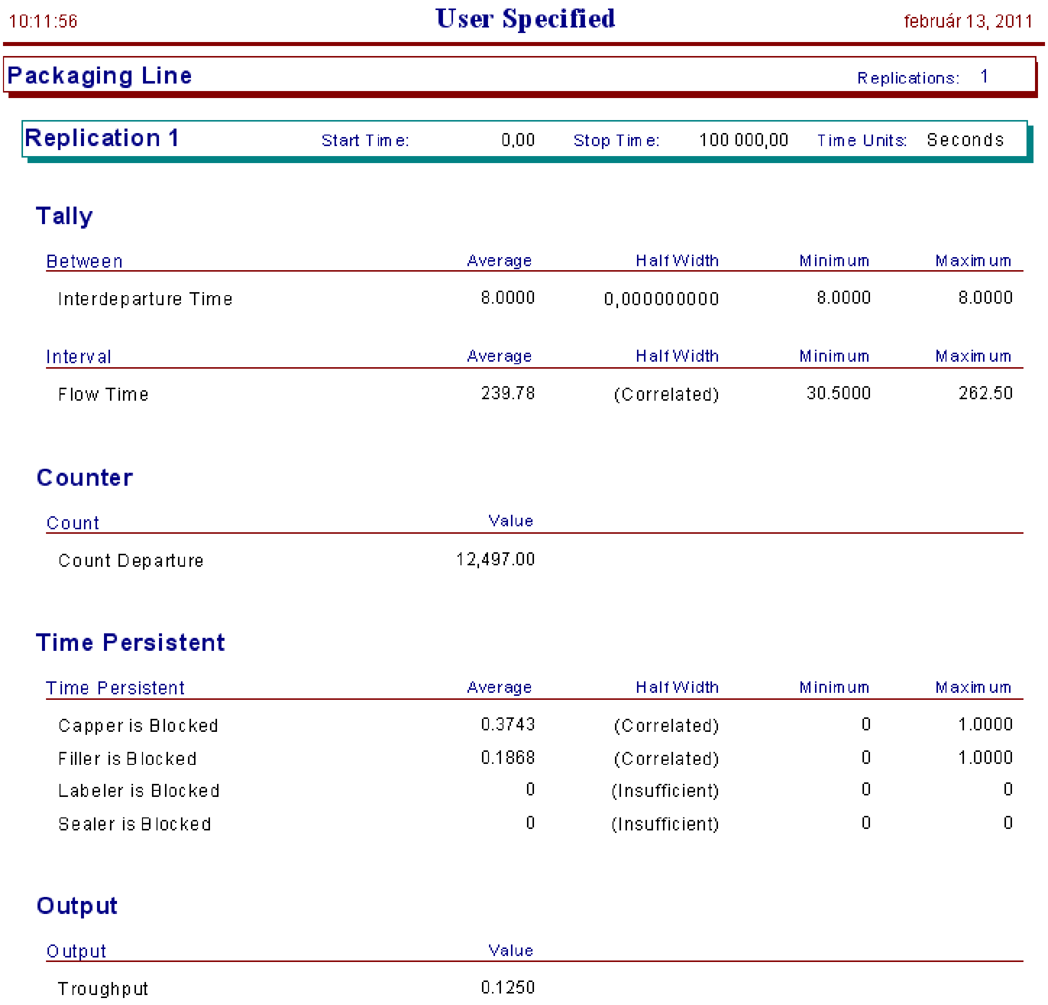
```
10:01:39                              Queues                          február 13, 2011

Packaging Line                                                        Replications:   1

Replication 1        Start Time:           0,00   Stop Time:       100 000,00   Time Units:  Seconds

Labeling Process.Queue

Time                                      Average       Half Width       Minimum       Maximum
  Waiting Time                            39.9557     (Correlated)             0       40.0000

Other                                     Average       Half Width       Minimum       Maximum
  Number Waiting                           4.9951     (Correlated)             0        5.0000

Packing Process.Queue

Time                                      Average       Half Width       Minimum       Maximum
  Waiting Time                                  0      0,000000000             0             0

Other                                     Average       Half Width       Minimum       Maximum
  Number Waiting                                0     (Insufficient)           0             0

Sealer Blocked.Queue

Other                                     Average       Half Width       Minimum       Maximum
  Number Waiting                                0     (Insufficient)           0             0

Sealing Process.Queue

Time                                      Average       Half Width       Minimum       Maximum
  Waiting Time                                  0      0,000000000             0             0

Other                                     Average       Half Width       Minimum       Maximum
  Number Waiting                                0     (Insufficient)           0             0
```

Figure 5.9. *Queues* report for the packaging line model.



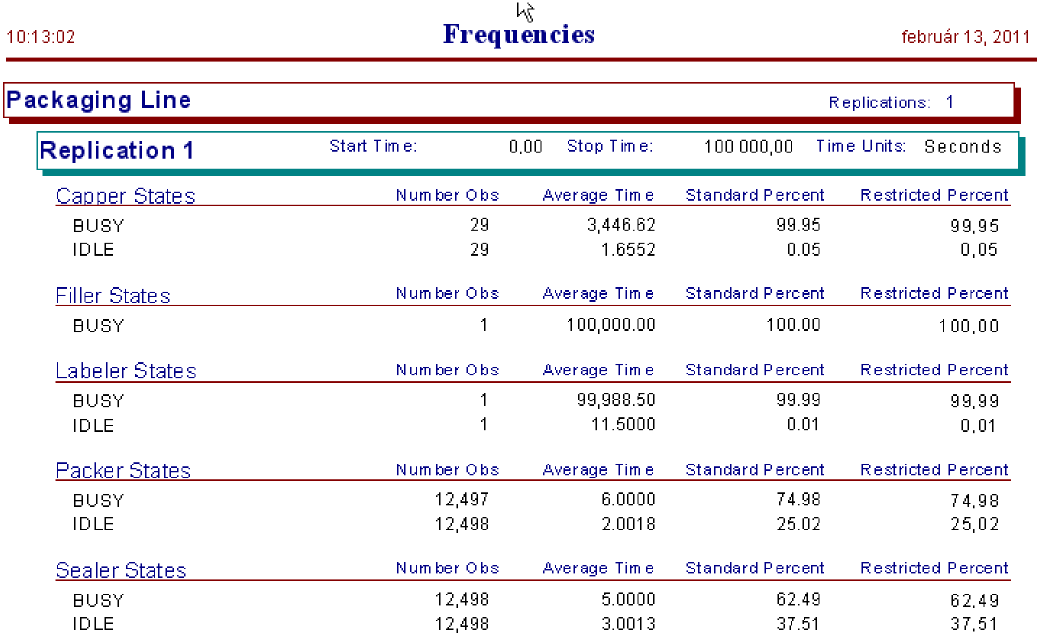
```
10:10:45                              Resources                       február 13, 2011

Packaging Line                                                        Replications:  1

Replication 1        Start Time:           0,00        Stop Time:     100 000,00     Time Units:  Seconds

Resource Detail Summary

Usage

                   Inst Util     Num Busy    Num Sched    Num Seized    Sched Util

Capper                  1,00         1,00         1,00     12 505,00          1,00
Filler                  1,00         1,00         1,00     12 511,00          1,00
Labeler                 1,00         1,00         1,00     12 499,00          1,00
Packer                  0,75         0,75         1,00     12 497,00          0,75
Sealer                  0,62         0,62         1,00     12 498,00          0,62
```

Figure 5.10. *Resources* report for the packaging line model.



```
10:11:56                         User Specified                        február 13, 2011

Packaging Line                                                    Replications:   1

  Replication 1          Start Time:        0,00    Stop Time:     100 000,00   Time Units:  Seconds

  Tally

   Between                                 Average        Half Width       Minimum      Maximum

    Interdeparture Time                     8.0000       0,000000000        8.0000       8.0000

   Interval                                Average        Half Width       Minimum      Maximum

    Flow Time                               239.78       (Correlated)       30.5000      262.50

  Counter

   Count                                     Value

    Count Departure                      12,497.00

  Time Persistent

   Time Persistent                         Average        Half Width       Minimum      Maximum

    Capper is Blocked                       0.3743       (Correlated)             0       1.0000
    Filler is Blocked                       0.1868       (Correlated)             0       1.0000
    Labeler is Blocked                           0      (Insufficient)            0            0
    Sealer is Blocked                            0      (Insufficient)            0            0

  Output

   Output                                    Value

    Troughput                               0.1250
```

Figure 5.11. *User Specified* report for the packaging line model.



```
10:13:02                          Frequencies                          február 13, 2011

Packaging Line                                                  Replications:   1

  Replication 1          Start Time:       0,00    Stop Time:     100 000,00   Time Units:  Seconds

   Capper States          Number Obs   Average Time   Standard Percent   Restricted Percent
    BUSY                          29       3,446.62              99.95                99,95
    IDLE                          29         1.6552               0.05                 0,05

   Filler States          Number Obs   Average Time   Standard Percent   Restricted Percent
    BUSY                           1     100,000.00             100.00               100,00

   Labeler States         Number Obs   Average Time   Standard Percent   Restricted Percent
    BUSY                           1      99,988.50              99.99                99,99
    IDLE                           1        11.5000               0.01                 0,01

   Packer States          Number Obs   Average Time   Standard Percent   Restricted Percent
    BUSY                      12,497         6.0000              74.98                74,98
    IDLE                      12,498         2.0018              25.02                25,02

   Sealer States          Number Obs   Average Time   Standard Percent   Restricted Percent
    BUSY                      12,498         5.0000              62.49                62,49
    IDLE                      12,498         3.0013              37.51                37,51
```

Figure 5.12. *Frequencies* report for the packaging line model.

Figure 5.12. displays the *Frequencies* statistics of the Reports panel for the first replication of the packaging line model. The displayed statistics pertain to the probabilities of the busy and idle states for each resource. An immediate observation is that the estimated busy probabilities for the resources Filler, Labeler, Packer, and Sealer include the corresponding blocking probabilities, since a blocked resource is not released, and therefore, is considered busy. Thus, in order to estimate the effective utilization of a resource (the fraction of time that it performs "actual work"), its estimated blocking probability is subtracted from its estimated probability in the busy state. *Table 5.1* displays the effective utilization of each process resource.

Table 5.1

Effective process utilization from Filling to Packing

Process	Filling	Capping	Labeling	Sealing	Packing
Utilization	0.813	0.625	0.999	0.625	0.750

5.4 Understanding system behavior and model verification

The generic packaging line model of Section 5.3 may be abstracted as a sequence of workstations (or servers) having fixed processing times per product unit, and with the proviso that the first workstation is never idle. Thus, the throughput of the system (production rate) coincides with that of the slowest workstation in the sequence.

In the example of Section 5.3, the labeling workstation (resource Labeler) is the slowest, with 8 seconds of unit processing time. Therefore, the utilization of the labeling workstation is expected to be 100%, since all upstream workstations have shorter (fixed) unit processing times. Since the filling and capping workstations are both faster than the labeling workstation, the labeling buffer is sure to fill up eventually, thereby blocking the capping workstation and later on the filling workstation, thereby giving rise to significant blocking probabilities in these workstations. Furthermore, every departure from the labeling workstation finds the downstream workstations (sealing and carton packing) in the *Idle* state, since they too have shorter processing times than the labeling workstation. In consequence, the labeling and sealing workstations are never blocked. Accordingly, a product unit departs from the labeling workstation every 8 seconds—which is also the interdeparture time in downstream machines—resulting in system throughput of 1/8 = 0.125 units per second. In fact, the throughput of each workstation in the system should also be 0.125. Indeed, using *Eq. 8.7*, which expresses the throughput $\bar{o}$ as the ratio of utilization to average unit processing time, the through-put of the filling workstation is $\bar{o}$ =0.813/6.5 =0.125, while the throughput of the capping workstation is $\bar{o}$ = 0.625/5 =0.125.

As one facet of simulation model verification, we next verify that the average flow time and the average number of product units obeys Little's formula (8.8), which is further discussed in Section 5.10. Define the subsystem *S* to be the sequence of processes from the *Filling Process* module to and including the *Packing Process* module (the raw material storage in front of *Filling Process* is included in *S*). The average flow time through S is measured from the arrival of a unit entity at queue *Filling Process.Queue* to the unit entity's arrival at module *Interdeparture Time*. The replication estimated this average flow time to be 239.79 seconds (see *Figure 5.11.*). The average number of units in *S*, $\overline{N}_S$, is just the sum of the corresponding average buffer contents (see *Figure 5.9.*) plus the average number of units in service (average time in busy state; see *Figure 5.12.*), that is,

(5.1) $\overline{N}_S = 15.6448 + 4.9860 + 4.9551 + 1.0 + 0.9995 + 0.9999 + 0.6249 + 0.7498 = 29.66$

which is close to the theoretical expected value 30 (note that this is a closed system with 30 circulating unit entities). Thus, in this case, Little's formula becomes the relation

(5.2) $$\overline{N}_S = \bar{o}\overline{F}_S$$

where $\overline{N}_S$ is the average number of units in *S*, $\overline{o}$ is the throughput of *S* as discussed above (the throughput of *S* equals the arrival rate at *S* in steady state), and $\overline{F}_S$ is the average flow time through *S*. Using the average flow time and the throughput in *Figure 5.11.* to calculate the right side of *Eq. 5.2* yields

(5.3) $$239.78 \text{ X } 0.125 = 29.9725.$$

Because the right sides of *Eqs. 5.1* and *5.3* are approximately the same, this fact tends to verify the replication's estimates. In fact, the discrepancy between *Eqs. 5.1* and *5.3* becomes progressively smaller, as the replication run length increases.

Next, we perform sensitivity analysis by studying the impact of buffer capacities on system behavior. Such analysis will aid us in gaining insight into blocking—a phenomenon stemming from mismatch in the service rates and finite buffer capacities. *Figure 5.13.* displays three performance measures as functions of simultaneously increasing buffer capacities in *Labeling Process* and *Capping Process*. From left to right, the plotted measures are the throughput of S, the WIP level at *Labeling Process*, and the average flow time through *S*, excluding the *Filling* buffer. (If flow times were to include time in the *Filling* buffer, then they would remain constant. Why?)

Interestingly, the throughput of *S* is largely unaffected, while the average WIP level at *Labeling Process* and the mean flow time through *S* increase as the buffer capacities are increased. To understand these observations, recall that the system under study has fixed processing times. Recall further that buffers are placed in production lines in order to absorb product flow fluctuations due to randomness in the system (e.g., random processing times, random downtimes, etc.). Such events introduce variability that tends to slow down product movement. But in systems with no variability (such as ours), the placement of buffers would have no impact on the throughput, as evidenced by *Figure 5.13*(*a*). However, placing larger buffers upstream of bottleneck workstations simply gives rise to larger WIP levels there, and consequently, to longer flow times. In contrast, buffers at workstations downstream of the bottleneck are always empty, regardless of capacity.

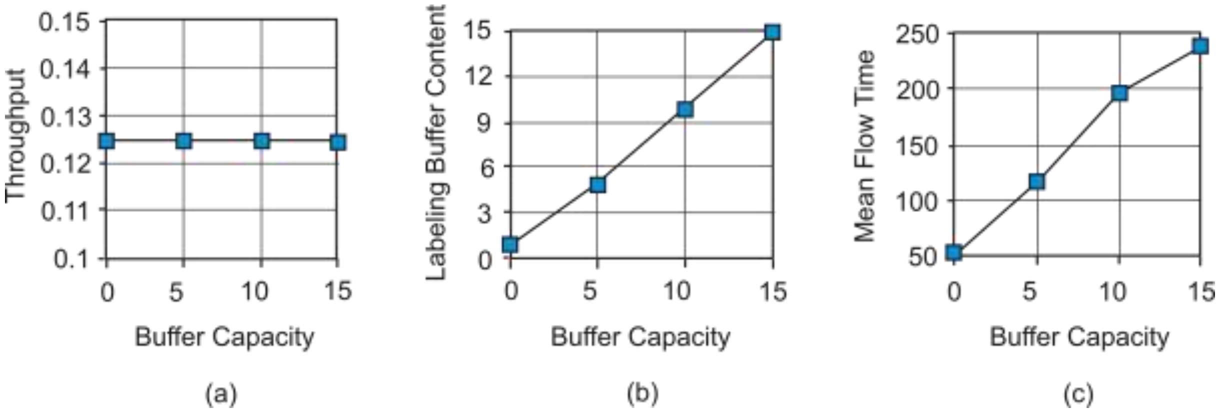


Figure 5.13. Impact of simultaneously increasing the capping and labeling buffer capacities on (*a*) trough put of *S*, (*b*) average number in labeling puffer, and (*c*) average flow time

To summarize, our sensitivity analysis has revealed some valuable and somewhat unexpected design principles for deterministic production lines, namely, that larger buffers actually can have an overall deleterious effect on the system. On the one hand, increasing buffer sizes will not increase the throughput. On the other hand, such increases will result in larger on-hand inventories, thereby tying up precious capital in inventory. Worse still, it would take jobs longer to traverse the system. These effects can be economically detrimental (think of the penalty of longer manufacturing lead times). Reasonably sized buffers are necessary, however, to absorb the effects of variability in product flow, in the presence of randomness. In Section 5.6, we will discuss the beneficial effect of placing buffers when machine failures and repairs are introduced.

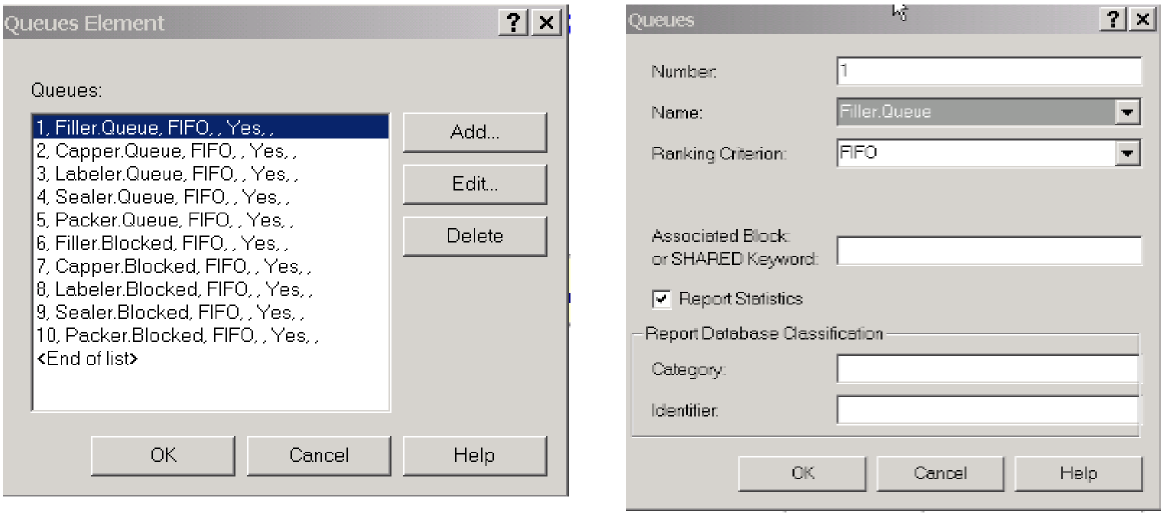
5.5 Model 5-2. Modeling production lines via indexed queues and resources

When models contain components with repetitive logic, then model construction can become tedious, and consequently, error-prone. A case in point is the packaging line model of Section 5.3, where component processes have analogous logic (although the parameters are different). More specifically, in each component process (filling, capping, labeling, sealing, and carton packing), units seize a resource, get processed, check if blocking occurs, and then release the corresponding resource. For models with such repetitive logic, Arena supports an efficient approach, called indexing, which largely eliminates the tedium of repetition. This approach exploits the fact that each object in Arena has an implicit integer ID number within its class; examples of object classes include such Arena constructs as queues, resources, and expressions. Thus, objects within a class can be viewed as object arrays, where individual objects are assigned an index for access and manipulation. An index is an integer that identifies the serial number of the object in the array, and as such it varies from 1 to the size of the array (the number of objects created). Note that the array size is dynamic, and the modeler can enforce any particular order of objects in the array to facilitate indexing logic. To this end, the modeler can use the Queues, Resources, and Expressions elements from the Elements template panel, accessible from the Template Attach button in the Standard toolbar.

To illustrate the use of indexing in modeling repetitive components, we will modify the packaging line example of Section 5.3. Exploiting the Arena indexing feature, we first number all resources from 1 to 5, all resource queues from 1 to 5, and all blocking queues from 6 to 10, and then declare these objects and their indexes using the old Arena elements of Resources and Queues.

Figure 5.14. depicts the dialog box of the element called Queues Element (left) and the associated dialog box for a particular queue (right). Initially, Queues Element on the left is empty. The modeler clicks the Add button to pop up the dialog box on the right to define the relevant queue attributes for the queue selected in the Name field (here Filler Queue). The most important fields in the Queues dialog box are the index of the queue in the Arena queue array (specified in the Number field), and the queueing discipline of the queue (specified in the Ranking Criterion field). In addition, the modeler can select one of two types of specifications in the Associated Block field: either specify a (SIMAN) block with which the present queue will be associated, or enter the keyword shared, thereby signaling that the queue will be used in multiple **Hold** and **Seize** modules.

The assignment of user-specified index numbers to resources is similar. *Figure 5.15.* depicts the dialog box of the element called Resources Element (top) and the associated dialog box for a particular resource (bottom). Note that the Resources dialog box is handy for specifying additional information for a resource (in this case resource Filler). In addition to the resource index and name, the modeler can use the Capacity or Schedule field to enter either a static resource capacity or a schedule name governing a dynamic time-dependent capacity. In *Figure 5.15,* we use the Resources dialog box merely to assign an index number to resource *Filler*.

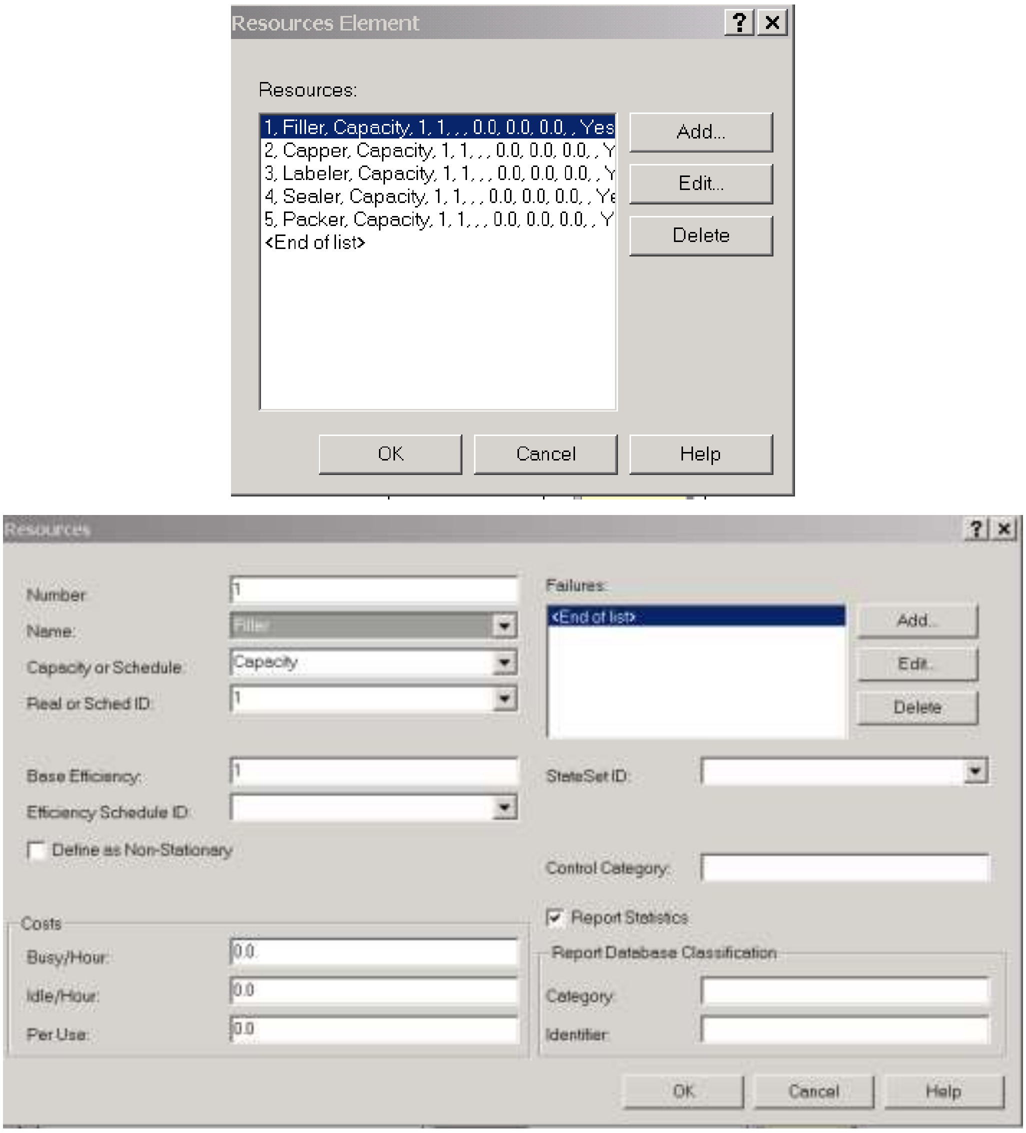


Queues:

1, Filler.Queue, FIFO, , Yes, ,
2, Capper.Queue, FIFO, , Yes, ,
3, Labeler.Queue, FIFO, , Yes, ,
4, Sealer.Queue, FIFO, , Yes, ,
5, Packer.Queue, FIFO, , Yes, ,
6, Filler.Blocked, FIFO, , Yes, ,
7, Capper.Blocked, FIFO, , Yes, ,
8, Labeler.Blocked, FIFO, , Yes, ,
9, Sealer.Blocked, FIFO, , Yes, ,
10, Packer.Blocked, FIFO, , Yes, ,
<End of list>

Number: 1
Name: Filler.Queue
Ranking Criterion: FIFO
☑ Report Statistics

Figure 5.14. Dialog boxes of **Queues** Element (left) and **Queues** (right).



Resources:

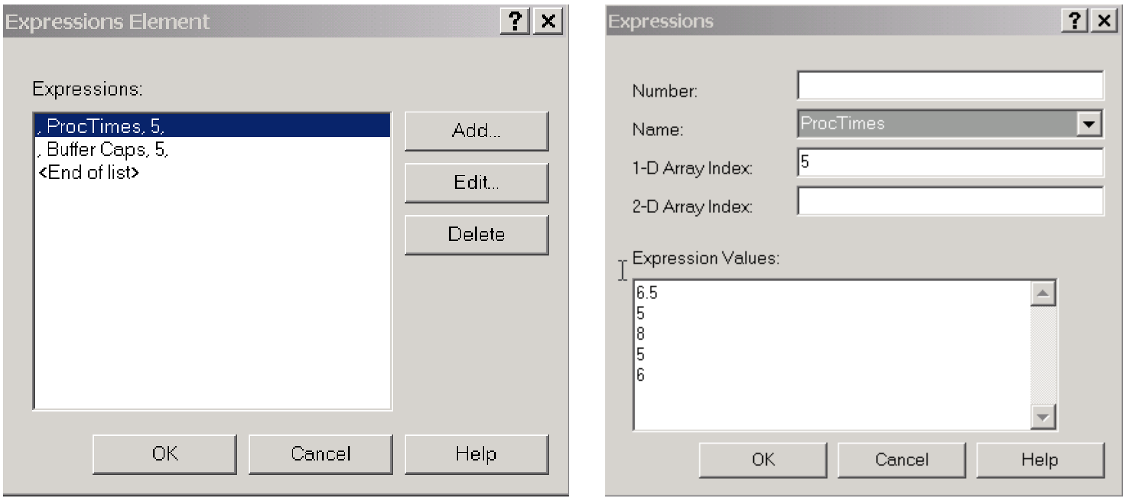
1, Filler, Capacity, 1, 1, , , 0.0, 0.0, 0.0, , Yes
2, Capper, Capacity, 1, 1, , , 0.0, 0.0, 0.0, , Y
3, Labeler, Capacity, 1, 1, , , 0.0, 0.0, 0.0, , Y
4, Sealer, Capacity, 1, 1, , , 0.0, 0.0, 0.0, , Ye
5, Packer, Capacity, 1, 1, , , 0.0, 0.0, 0.0, , Y
<End of list>

Number: 1
Name: Filler
Capacity or Schedule: Capacity
Real or Sched ID: 1
Base Efficiency: 1
☐ Define as Non-Stationary
Busy/Hour: 0.0
Idle/Hour: 0.0
Per Use: 0.0
Failures: <End of list>
☑ Report Statistics

Figure 5.15. Dialog boxes of Resources Element (top) and *Resources* (bottom).

Finally, indexing of Arena expressions in the packaging line is illustrated in *Figure 5.16*, which depicts the dialog box of the element called Expressions Element (left) and the associated dialog box for a particular expression (right). Expression arrays may be vectors (one-dimensional) or matrices (two-dimensional). To specify a vector, the user enters its dimension in the 1-D Array Index field. Similarly, to specify a matrix, the user enters its dimensions in the 1-D Array Index and 2-D Array Index fields. The actual elements of the expression array are entered in sequence column by column, one expression per line. Here, the entered expressions are constants that specify processing times and buffer capacities.

The equivalent version of the Arena model of Section 5.3, modified to incorporate indexing, is depicted in *Figure 5.17*, where all indexed queues, expressions, and resources are shown in this order under the corresponding headings on the right side of the figure. Starting at the upper left corner, the **Create** module, called Create Jobs, operates exactly as module *Create* 1, its counterpart in *Figure 5.2*. (the "jobs" alluded to are, of course, product units). This module generates 30 unit entities at time 0, which circulate repeatedly in the model. However, the logic of routing product units through the packaging line workstations is implemented in a manner analogous to a loop in a procedural programming language. In our case, the unit entity attribute *Proc_Index* keeps track of the loop's running index.



Expressions Element

Expressions:

, ProcTimes, 5,
, Buffer Caps, 5,
<End of list>

Expressions

Number:

Name: ProcTimes

1-D Array Index: 5

2-D Array Index:

Expression Values:

6.5
5
8
5
6

Figure 5.16. Dialog boxes of Expressions Element (left) and Expressions (right).

When a unit entity emerges from module *Create Jobs*, it proceeds to the **Assign** module, called *Assign Process Index* and *Time*, where its *Proc_Index* attribute is initialized to 1 (as well as saving the current simulation time in attribute *Arr_Time* for later collection of total flow-time observations (at the last workstation implementing the packing process). Note that the packing line workstations model the resources *Filler*, *Capper*, *Labeler*, *Sealer*, and *Packer*, and these are indexed by 1,2,3,4, and 5, respectively, as indicated by the list of resources under the Resources heading in *Figure 5.17*; the corresponding resource queues are indexed in the same way, as indicated by the list of queues under the Queues heading. Each unit entity will next be routed through the same sequence of modules, and its *Proc_Index* attribute will be incremented after each iteration so as to correspond to the next workstation to be visited. As will be seen, the appropriate service distribution for each workstation process will also be selected by this attribute.

The unit entity proceeds to the first module in the loop, namely, the **Seize** module called *Seize Process*. The dialog box for this module is displayed in *Figure 5.18*. Here, the Resources field indicates that the resource to be currently selected for seizing is specified by the current index number stored in attribute *Proc_Index* (initially the index is 1, which corresponds to resource Filler). The dialog box for the current resource in the loop is shown in *Figure 5.19*. Note that in this dialog box, the appropriate resource is selected by attribute (in the Type field), and the Attribute Name field specifies attribute *Proc_Index*.

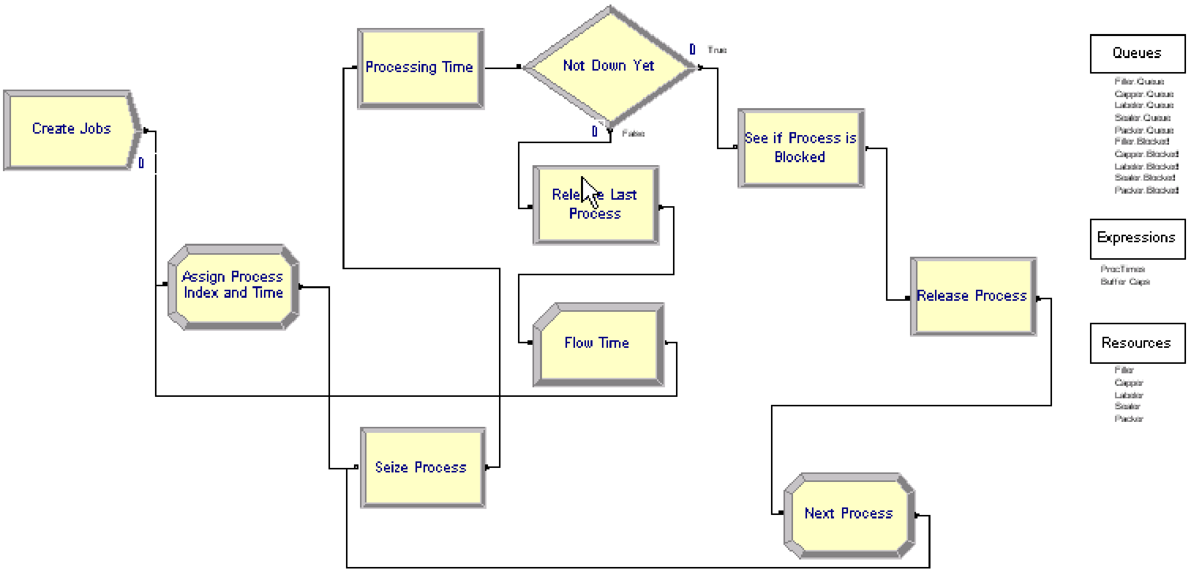
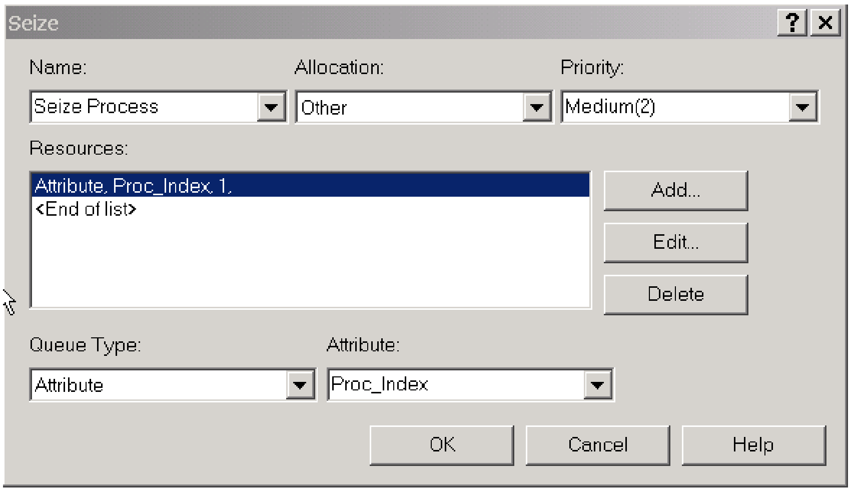


Figure 5.17. Arena model with indexing for the packaging line system.


Seize

Name: Seize Process

Allocation: Other

Priority: Medium(2)

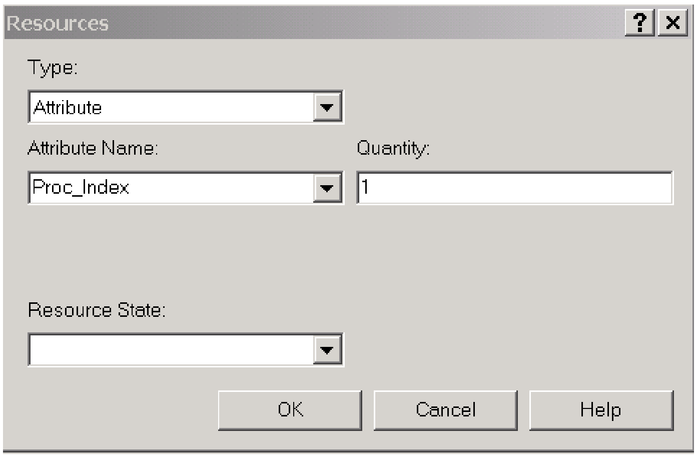
Resources:

Attribute, Proc_Index, 1,
<End of list>

Queue Type: Attribute

Attribute: Proc_Index

Figure 5.18. Dialog box of the first module (**Seize**) in the loop.


Resources

Type: Attribute

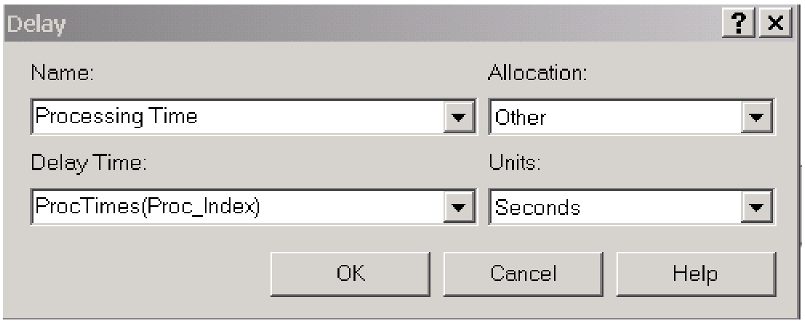
Attribute Name: Proc_Index

Quantity: 1

Resource State:

Figure 5.19. Dialog box of the current resource in the loop.

Having seized the selected resource, the unit entity proceeds to the second module in the loop, namely, the **Delay** module called *Processing Time*. The dialog box for this module is displayed in *Figure 5.20*. Here, the Delay Time field indicates that a delay time for this module is the value at the current index position (saved in attribute *Proc_Index*) of the *Proc Times* expression vector (recall that this expression vector was initialized previously, as illustrated in *Figure 5.16.*).



Name: Processing Time

Allocation: Other

Delay Time: ProcTimes(Proc_Index)

Units: Seconds

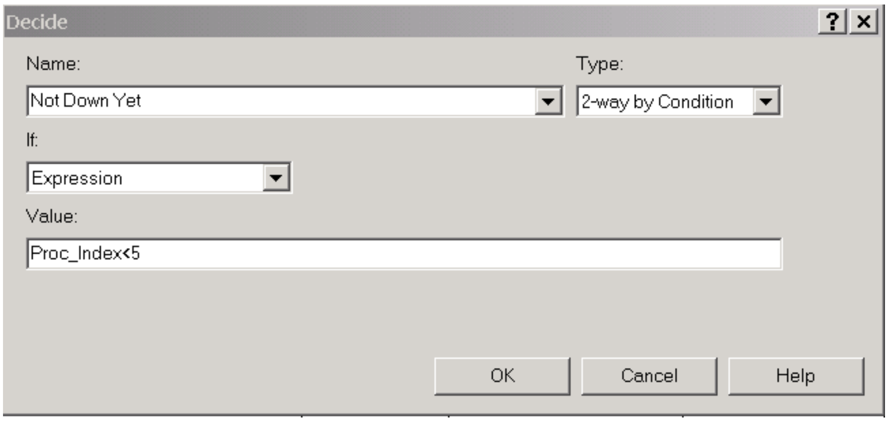
Figure 5.20. Dialog box of the second module (**Delay**) in the loop.

Once the unit entity completes its delay and emerges from the *Processing Time* module, its processing at the current workstation is also complete. It now needs to make the following decision:

If this workstation (process) is not the last in the sequence, then proceed to the next workstation in the loop. Since the last process (packing) has index number 5, this is equivalent to checking whether *Proc_Index*<5.

Otherwise (i.e., *Proc_Index*==5), proceed to collect the system flow time (sojourn time in all five workstations) for this product unit, and then restart another sequence of processes.

To this end, the unit entity enters the third module in the loop, namely, the **Decide** module called *Not Done Yet*. *Figure 5.21.* displays the dialog box for this module. Note that this **Decide** module implements a two-way decision (in the Type field), controlled by a (logical) expression (in the If field) specified as *Proc_Index*<5 (in the Value field). If the value of this expression is false (i.e., *Proc_Index*==5 is *true*), then the unit entity processing is essentially complete, with only a few remaining housekeeping chores left.



Name: Not Down Yet

Type: 2-way by Condition

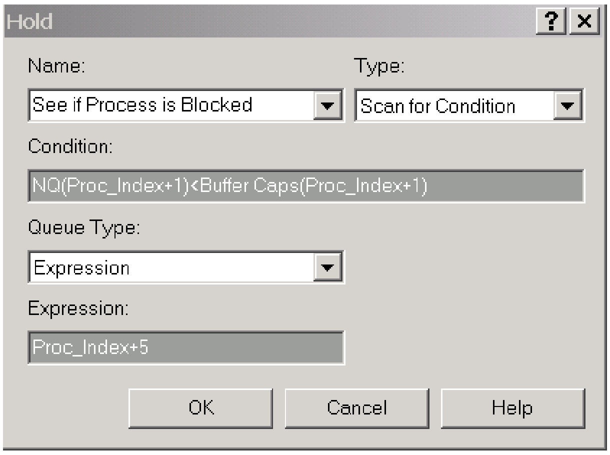
If: Expression

Value: Proc_Index<5

Figure 5.21. Dialog box of the third module (**Decide**) in the loop.

Specifically, the unit entity will proceed in this case to the **Release** module, called *Release Last Process*, to release the last resource, and will enter the **Record** module, called *Flow Time*, to collect the next system flow-time observation, before routing to the **Assign** module, called *Assign Process Index* and *Time*, for the next tour through all five workstations (new job processing). However, if *Proc_Index* < 5 is true, then the unit entity proceeds to the fourth module in the loop, namely, the **Hold** module called *See if Process is Blocked*; see the module's dialog box in *Figure 5.22*. In this module, the unit entity checks whether the next workstation (indexed by *Proc_Index*+1) is full (i.e., that transfer to the next process is blocked), by checking the logical expression in the Condition field:

$$NQ(Proc\_Index + 1) < Buffer\ Caps(Proc\_Index + 1).$$



Name: See if Process is Blocked
Type: Scan for Condition
Condition: NQ(Proc_Index+1)<Buffer Caps(Proc_Index+1)
Queue Type: Expression
Expression: Proc_Index+5

Figure 5.22. Dialog box of the fourth module (**Hold**) in the loop.

Recall that *Buffer Caps* is an expression vector holding buffer capacities, which were previously initialized by the modeler. Note carefully that blocking queue indexes are specified by expressions (Queue Type field) whose values are given by *Proc_Index* + 5 (Expression field). Thus, this **Hold** module is used for all processes, except for the last one; it holds a blocked unit entity at queue *Proc_index*+5 until the next workstation has room to accommodate it.

As soon as the next workstation is found to have room (i.e., the next process is found to be nonblocking), the unit entity will move to the fifth module in the loop, namely, the **Release** module, called *Release Process*, to release the current resource (the one indexed by *Proc_Index*). The unit entity then proceeds to the sixth and last module in the loop, namely, the **Assign** module called *Next Process*, where the unit entity's attribute *Proc_Index* is incremented by 1, thereby pointing at the next workstation (process) to visit. Finally, the unit entity completes the loop by returning to the first module in the loop, namely, the **Seize** module, called *Seize Process*, to start the next loop iteration.

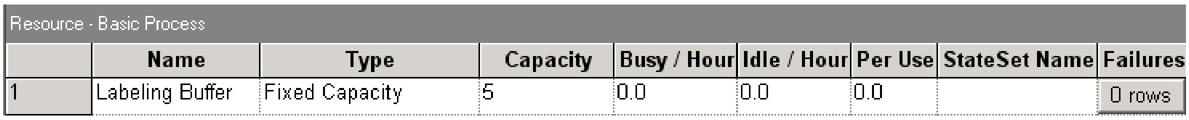
A comparison of run results of the equivalent packaging line models of *Figures 5.2* and *5.17* shows that they produce identical statistics, as expected.

5.6 Model 5-3. An alternative method of modeling blocking

There are several ways of modeling blocking in Arena. In the packaging line example of Section 5.3, we chose to model blocking via the **Hold** module of the Advanced Process template panel. An alternative way would be to treat each buffer as a resource. *Figure 5.23.* exemplifies this alternative modeling approach for a buffer in the packaging line model in *Figure 5.2*.

In *Figure 5.23*, the resource at the labeling workstation, which controls the number of units in the queue of *Labeling Process*, is named *Labeling Buffer*. The Capacity entry (number of servers) of this resource is set to 5 units in order to correspond to a buffer capacity of 5 in *Labeling Process*. An Arena implementation of this approach to modeling blocking is depicted in *Figure 5.24*.

In the previous model illustration our goal was to ensure that when the queue in *Labeling Process* is full, the *Capper* resource is not released for further capping. To this end, we arrange that each unit entity that has completed capping at *Capper* attempt to seize a unit capacity of *Labeling Buffer* before releasing the *Capper* resource. If no residual capacity of the resource *Labeling Buffer* is available, then the resource *Capper* gets blocked, thereby holding the unit entity until buffer space becomes available in the *Labeling Buffer* resource. As soon as this happens, the unit entity releases the *Capper* resource and moves on to *Labeling Process*, where it immediately joins *Labeling Process.Queue*, and waits for its turn to seize the resource *Labeler*. During this wait, the unit entity keeps its unit space of the resource *Labeling Buffer*, and only releases that space when it succeeds in seizing the resource *Labeler*. Since there are only five units of resource available for seizing at *Labeling Buffer*, there will be at most five unit entities in the "physical" buffer *Labeling Process.Queue*.



Resource - Basic Process

	Name	Type	Capacity	Busy / Hour	Idle / Hour	Per Use	StateSet Name	Failures
1	Labeling Buffer	Fixed Capacity	5	0.0	0.0	0.0		0 rows

Figure 5.23. Dialog spreadsheet of the **Resource** module for modeling blocking.

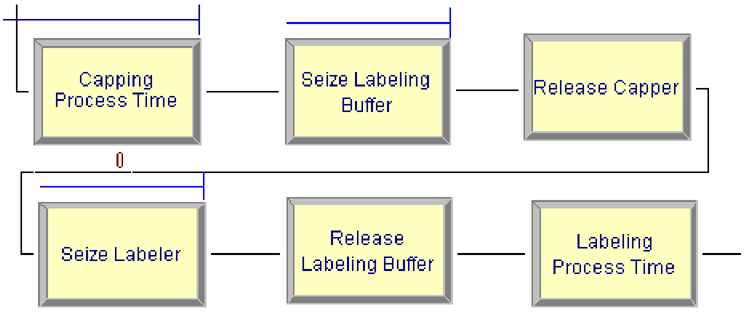


Figure 5.24. Alternate Arena modeling of blocking via **Resource** modules.

Note that this approach to modeling blocking in Arena is costlier than the one used in modeling the packaging line in Section 5.3. There, only one module (**Hold**) was used to model blocking per process, whereas here three modules (**Seize**, **Release**, and **Resource**) are used for the same purpose. This increased complexity can become problematic. For example, the student version of Arena allows only a limited number of modules to be used in any given model.

5.7 Model 5-4. Modeling machine failures

Process stoppages are unavoidable in manufacturing and service systems. In particular, machine failures of various kinds constitute an important source of idleness and variability in such systems. When modeling systems subject to failures, the modeler is usually interested in the long-run probabilities of down states, which translate operationally into the long-range fraction of time spent in those states. Certainly, efficient operation requires that downtimes be minimized, since these represent loss of production time. Simulation can help the modeler understand the impact of failures on system performance.

A stochastic process that models failures are characterized either by the statistical properties of the time intervals separating consecutive failures, or by the counting process that keeps track of the number of failures within a given period of time. These two characterizations are mathematically equivalent. However, the former characterization is commonly used to generate failures in simulation runs. In fact, failures can be modeled as a specialized arrival process of high priority, whose transactions simply preempt the server.

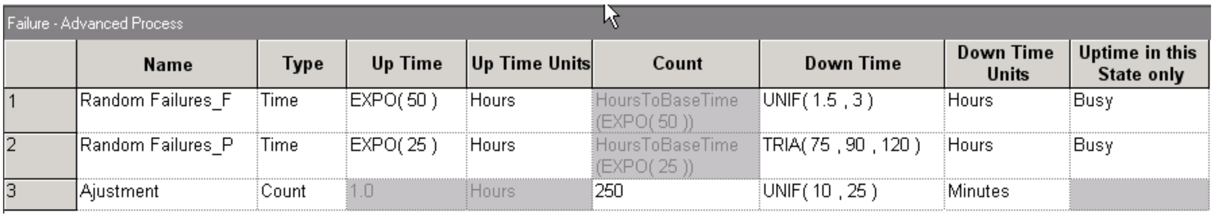
A typical failure scenario at a single workstation unfolds as follows. After a period of normal operation (uptime), a failure event occurs the workstation stops its processing, and then experiences a downtime while undergoing a repair. (Sometimes an additional delay for repair setup needs to be modeled, but this delay can usually be absorbed into a longer repair time.) Failures that occur while machines are actually processing jobs are called *operation dependent*. However, the new breed of highly computerized machines may fail at any time, regardless of machine status. Such failures are called *operation independent*, and may include machine malfunctions, startups, cleanups, adjustments, and other delays specific to a particular system. The two fundamental machine states, *Idle* and *Busy*, must then be augmented by additional failure and stoppage states, such as *Down*, *Cleaning*, *Adjustment*, and so on.

Arena admits any number of user-defined states in addition to the built-in ones (called auto-states). In particular, an Arena resource has four *auto-states*: *Idle*, *Busy*, *Failed*, and *Inactive*, and at any point in time a resource is in one of these states. A transition from one state to another is caused by an event occurrence. For instance, a failure-arrival event automatically causes the machine to undergo a transition to the *Failed* state. On the other hand, user-defined states and their transitions can be programmed entirely at the modeler's discretion. We are usually interested in the long-run probabilities of these states; such probabilities can be estimated via the *Frequency* option in the **Statistic** module.

The modeler can modulate the capacity of a resource over time by defining a time-varying capacity level in the **Schedule** module (see Section 5.8 for more details). In particular, to model forced downtimes (e.g., preventive maintenance), a resource can be inactivated by setting its capacity to zero, in which case the *Inactive* auto-state will have a non-zero probability.

Suppose that *Filling Process* in the packaging line model of Section 5.3 fails randomly and that it needs an adjustment after every 250 departures from the work-station. Assume that uptimes (times between a repair completion and the next failure, or time to failure) are exponentially distributed with a mean of 50 hours, while repair times are uniformly distributed between 1.5 hours to 3 hours. Also, the aforementioned adjustment time is uniformly distributed between 10 minutes to 25 minutes. Assume further that *Packing Process* can also experience random mechanical failures, and downtimes are triangularly distributed with a minimum of 75 minutes, a maximum of 2 hours, and a mode at 90 minutes. The corresponding uptimes are exponentially distributed with a mean of 25 hours. Finally, assume that random failures occur only while the machines are busy (operation-dependent failures). We shall refer to the modified packaging line model as the *failure-modified model*.

Figure 5.25. illustrates how failures in the failure-modified model are specified in a dialog spreadsheet for the **Failure** module from the Advanced Process template panel. The Name column in *Figure 5.25.* specifies failure names, while the Type column selects the type of failure arrivals: *Time* (for time-based arrivals) or *Count* (for count-based arrivals). For instance, random mechanical failures are normally time based, since their arrivals call for inter-failure time specification. In contrast, if a machine requires a cleanup action whenever it completes processing a prescribed number of units, then the cleanup stoppage is count based. Note that in *Figure 5.25*, the failure/stoppage named *Adjustment* is declared to be count-based.



Failure - Advanced Process

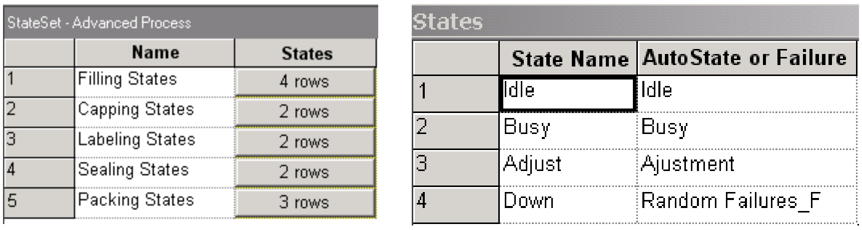
	Name	Type	Up Time	Up Time Units	Count	Down Time	Down Time Units	Uptime in this State only
1	Random Failures_F	Time	EXPO(50)	Hours	HoursToBaseTime (EXPO(50))	UNIF(1.5 , 3)	Hours	Busy
2	Random Failures_P	Time	EXPO(25)	Hours	HoursToBaseTime (EXPO(25))	TRIA(75 , 90 , 120)	Hours	Busy
3	Ajustment	Count	1.0	Hours	250	UNIF(10 , 25)	Minutes	

Figure 5.25. Dialog spreadsheet of the **Failure** module.

For time-based failures, the Up Time column specifies the time interval to the next failure (after a repair completion), while the Up Time Units column specifies the corresponding time unit. Similarly, for count-based failures, the Count column specifies the number of entity departures to the next failure (250 for the Adjustment failure). Note that unused column entries are shaded depending on failure type.

The Down Time column specifies the length of downtimes, while the Down Time Units column specifies the corresponding time unit. Finally, the Uptime in this State only column is used for time-based failures to specify the state in which the resource must be for the failure to occur. For example, the time-based failures in *Figure 5.25.* can only occur when the underlying resources are in the *Busy* state.

Arena provides a mechanism for defining resource states and for linking them to failures/stoppages in the form of the **StateSet** spreadsheet module from the Advanced Process template panel. *Figure 5.26.* illustrates the use of the **StateSet** module for the packaging line model of Section 5.3. The left dialog spreadsheet of *Figure 5.26.* specifies a state set name (under the *Name* column) and the number of associated states (under the *States* column). For example, the first row specifies a state set, called *Filling States*, for the Filler resource with 4 states (the button labeled 4 rows). Clicking that button pops up the dialog box to the right, which displays detailed state information. There, the column *State Name* displays user-defined or auto-state names, while the column *AutoState* or *Failure* indicates the association of each state name with the corresponding auto-state or user-defined failure name.



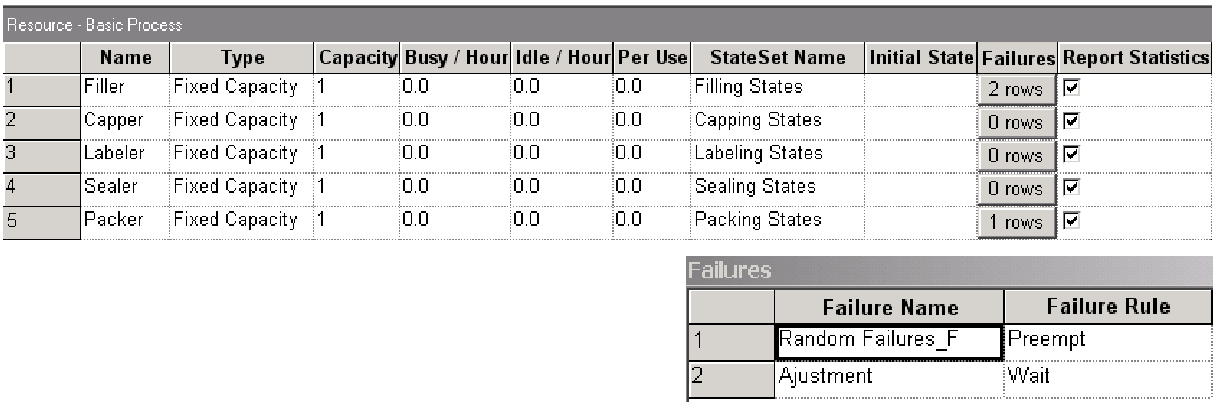
StateSet - Advanced Process

	Name	States
1	Filling States	4 rows
2	Capping States	2 rows
3	Labeling States	2 rows
4	Sealing States	2 rows
5	Packing States	3 rows

States

	State Name	AutoState or Failure
1	Idle	Idle
2	Busy	Busy
3	Adjust	Ajustment
4	Down	Random Failures_F

Figure 5.26. Dialog spreadsheet of the **StateSet** modul (left), and resource *Filler* states (right)



Resource - Basic Process

	Name	Type	Capacity	Busy / Hour	Idle / Hour	Per Use	StateSet Name	Initial State	Failures	Report Statistics
1	Filler	Fixed Capacity	1	0.0	0.0	0.0	Filling States		2 rows	☑
2	Capper	Fixed Capacity	1	0.0	0.0	0.0	Capping States		0 rows	☑
3	Labeler	Fixed Capacity	1	0.0	0.0	0.0	Labeling States		0 rows	☑
4	Sealer	Fixed Capacity	1	0.0	0.0	0.0	Sealing States		0 rows	☑
5	Packer	Fixed Capacity	1	0.0	0.0	0.0	Packing States		1 rows	☑

Failures

	Failure Name	Failure Rule
1	Random Failures_F	Preempt
2	Ajustment	Wait

Figure 5.27. Resource dialog spreadsheet (top) and **Failures** dialog spreadsheet (bottom) in the failure-modified packaging line model.

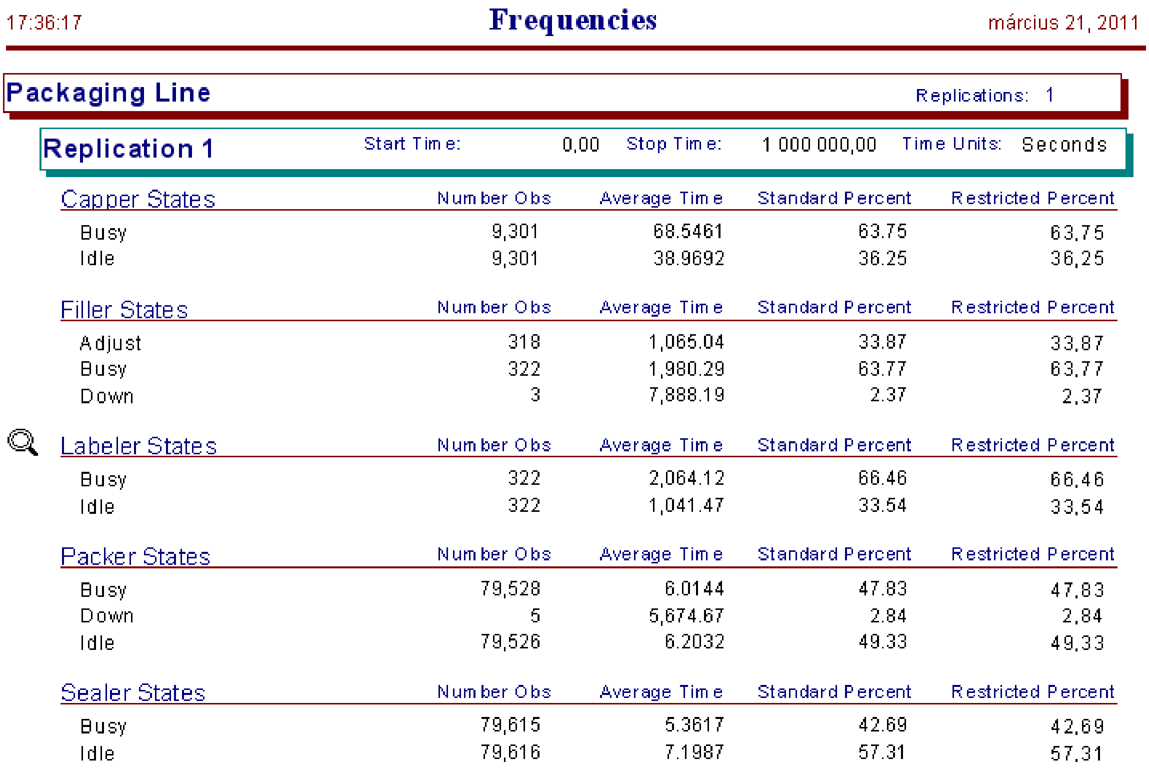
Finally, *Figure 5.27.* illustrates for the failure-modified model how the **Resource** module is used to associate resource states with failures and the action to be taken on failure occurrences. The top dialog spreadsheet of *Figure 5.27.* specifies in the first row that resource *Filler* (column *Name*) has 2 types of failures (the button labeled 2 rows in column *Failures*). Clicking that button pops up the dialog spreadsheet at the bottom of *Figure 5.27*, which shows that resource *Filler* has two failures, called *Random Failures_F* and *Adjustment* (column *Failure*

Name). The *Failure Rule* column specifies the requisite action to be taken on the unit entity (or entities) being processed when a time-based failure occurs (it does not apply to count-based failures). Actions are specified via options, the most common of which follow:

> The *Preempt* option starts a downtime by suspending the resource immediately on failure arrival, so that the remaining processing of the current unit entity will resume once the downtime is over.
>
> The *Wait* option allows the current unit entity to finish processing, after which the resource is suspended and downtime begins.
>
> The *Ignore* option starts the downtime after the current unit entity finishes processing. However, only that portion of the downtime following the current unit entity completion is recorded (in contrast, the *Wait* option records the full downtime).

In our example, failure *Random Failures_F* will apply the *Preempt* option, while failure *Adjustment*, being count-based, will formally apply the *Wait* option, even though any other option could have been selected (recall that options do not apply to count-based failures).

The state probabilities of each resource can be obtained by requesting the collection of *Frequency* statistics in the **Statistic** module with the *State* option selected in its Frequency Type column. *Figure 5.28.* displays the resulting *Frequencies* report for the failure-modified model.

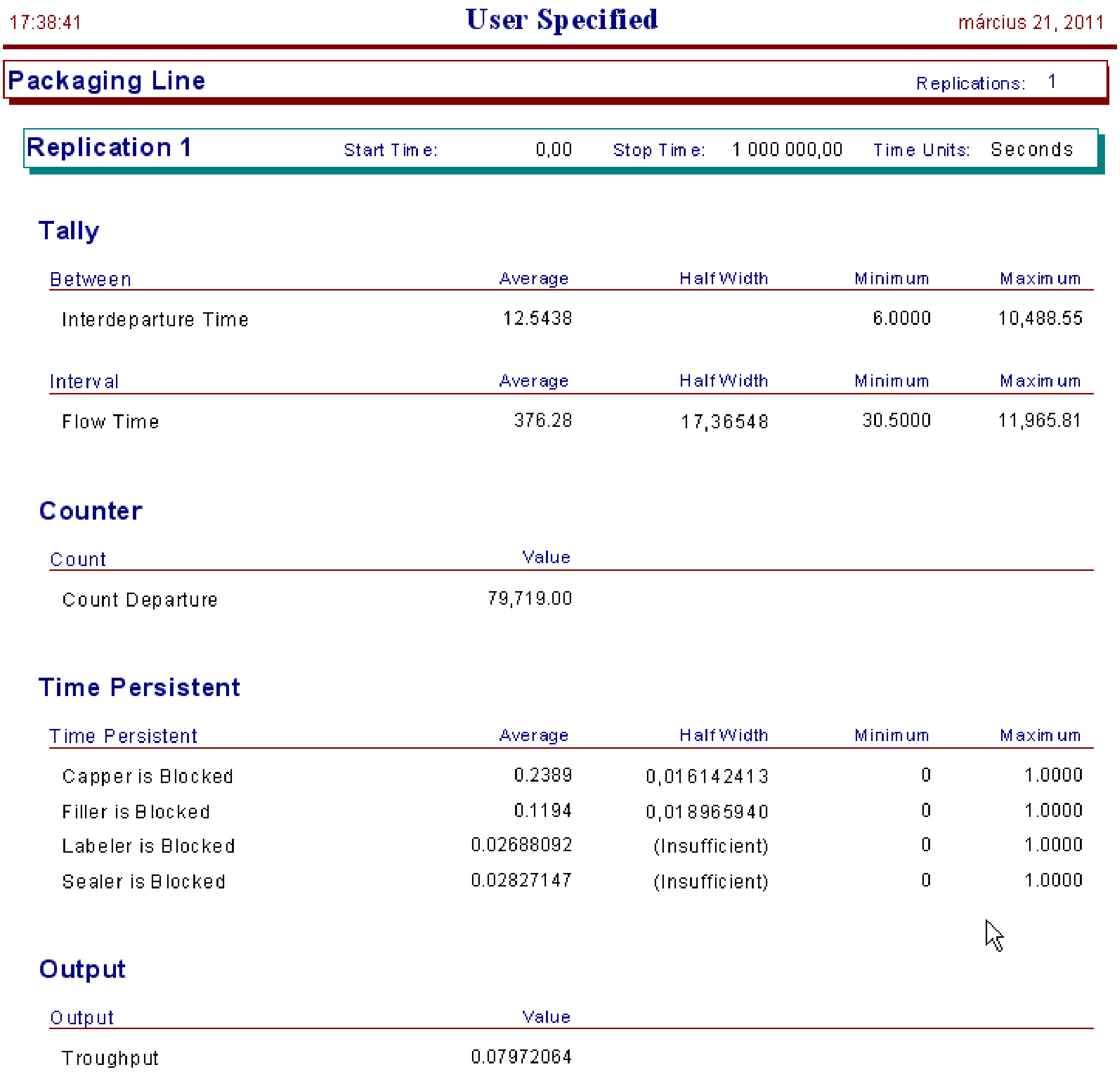


```
17:36:17                         Frequencies                          március 21, 2011

Packaging Line                                                       Replications:  1

  Replication 1        Start Time:         0,00   Stop Time:   1 000 000,00   Time Units:  Seconds

   Capper States                 Number Obs    Average Time    Standard Percent    Restricted Percent
     Busy                             9,301         68.5461               63.75                 63,75
     Idle                             9,301         38.9692               36.25                 36,25

   Filler States                 Number Obs    Average Time    Standard Percent    Restricted Percent
     Adjust                             318        1,065.04               33.87                 33,87
     Busy                               322        1,980.29               63.77                 63,77
     Down                                 3        7,888.19                2.37                  2,37

   Labeler States                Number Obs    Average Time    Standard Percent    Restricted Percent
     Busy                               322        2,064.12               66.46                 66,46
     Idle                               322        1,041.47               33.54                 33,54

   Packer States                 Number Obs    Average Time    Standard Percent    Restricted Percent
     Busy                            79,528          6.0144               47.83                 47,83
     Down                                 5        5,674.67                2.84                  2,84
     Idle                            79,526          6.2032               49.33                 49,33

   Sealer States                 Number Obs    Average Time    Standard Percent    Restricted Percent
     Busy                            79,615          5.3617               42.69                 42,69
     Idle                            79,616          7.1987               57.31                 57,31
```

Figure 5.28. Frequencies report for the failure-modified packaging line model.

It is instructive to compare the performance of the original packaging line model of Section 5.3 with its failure-modified version. Everything else being the same, we expect the failure-modified model to show poorer performance than the original model. Indeed, a comparison of *Figure 5.28.* to *Figure 5.12.* reveals that the resources downstream of the *Filler* resource in the failure-modified model experience markedly increased idleness. The explanation of this outcome is straightforward. Since the *Filler* resource in the failure-modified model is shut down by random failures, it cannot produce as much as in the original model. Consequently, downstream processes are more frequently starved, leading to overall increased idleness in the system. In a similar vein, *Figure 5.29.* displays the resultant *User Specified* report for the failure-modified model.

A comparison of *Figure 5.29.* with *Figure 5.11.* reveals additional evidence that the failure-modified model performs at a lower level than the original one. Because *Filling Process* has increased idleness in the failure-modified model, its mean interdeparture time increases to about 12.5 seconds (as opposed to 8 seconds in the original model), and concomitantly, its system throughput is only about 0.08 (as opposed to 0.125 units per second in the original model). Thus, this example amply demonstrates the deleterious effects of machine failures and stoppages on system performance measures. The economic consequences of the resultant system performance must necessarily follow suit.



17:38:41 User Specified március 21, 2011

Packaging Line Replications: 1

Replication 1 Start Time: 0,00 Stop Time: 1 000 000,00 Time Units: Seconds

Tally

Between	Average	Half Width	Minimum	Maximum
Interdeparture Time	12.5438		6.0000	10,488.55

Interval	Average	Half Width	Minimum	Maximum
Flow Time	376.28	17,36548	30.5000	11,965.81

Counter

Count	Value
Count Departure	79,719.00

Time Persistent

Time Persistent	Average	Half Width	Minimum	Maximum
Capper is Blocked	0.2389	0,016142413	0	1.0000
Filler is Blocked	0.1194	0,018965940	0	1.0000
Labeler is Blocked	0.02688092	(Insufficient)	0	1.0000
Sealer is Blocked	0.02827147	(Insufficient)	0	1.0000

Output

Output	Value
Troughput	0.07972064

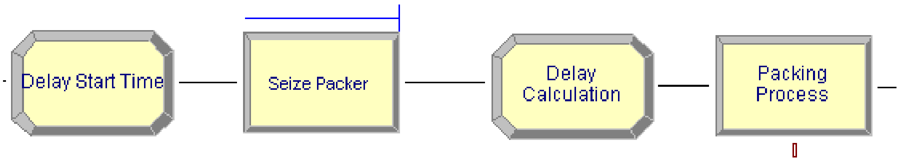
Figure 5.29. User Specified report for the failure-modified packaging line model.

The impact of machine failures on system performance and the reduction of this impact by placing buffers between machines have been extensively studied. For further discussions on the subject, see Buzacott and Shanthikumar (1993), Gershwin (1994), Altiok (1997), and Papadopoulos et al. (1993). Analysis of manufacturing networks with infinite buffers is found in Whitt (1983).

5.8 Model 5-5. Estimating distributions of sojourn times

Frequency statistics are handy Arena constructs that simplify the task of estimating the distribution of the number of units in a queue or buffer. However, estimating the distribution of a sojourn time (total time spent in a system or subsystem) requires a bit more work. For instance, suppose we are interested in the distribution of the delay time D_p that a product unit spends in the packing buffer of the packaging line model of Section 5.3. More specifically, we wish to estimate the following delay-time probabilities:

$$\mathrm{Pr}\{D_p<1\},\ \mathrm{Pr}\{1\le D_p<3\},\ \mathrm{Pr}\{3\le D_p<5\}\text{ and }\mathrm{Pr}\{D_p\ge 5\}.$$



Name	Delay Start Time
Assignments	
Type	Attribute
Attribute Name	Delay Start Time
New Value	TNOW

Name	Delay Calculation
Assignments	
Type	Attribute
Attribute Name	Delay in Pack_Q
New Value	TNOW − Delay Start Time
Type	Variable
Variable Name	N1
New Value	N1 + (Delay in Pack_Q <1)
Type	Variable
Variable Name	N3
New Value	N3 +(Delay in Pack_Q>=1 .and. Delay in Pack_Q<3)
Type	Variable
Variable Name	N5
New Value	N5+(Delay in Pack_Q>=3 .and.Delay in Pack_Q<5)
Type	Variable
Variable Name	NM
New Value	NM+(Delay in Pack_Q>=5)
Type	Variable
Variable Name	No of Obsevation
New Value	No of Obsevation + 1

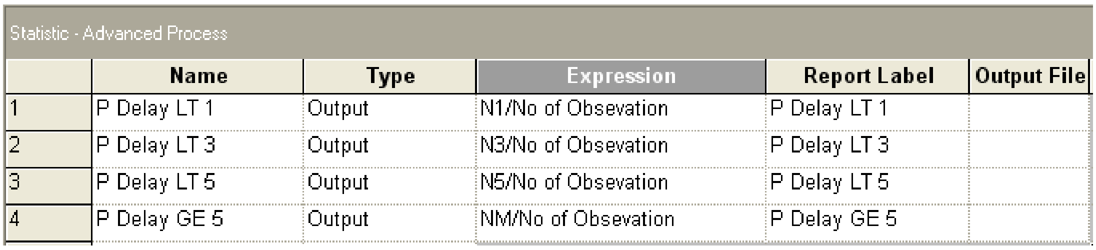
Figure 5.30. Arena fragment for estimating delay-time probabilities in the packing buffer.

Figure 5.30. displays an Arena fragment for estimating these delay-time probabilities. Here, an arriving unit entity traverses the fragment from left to right in *Figure 5.30*. On arrival in the packing workstation, the first **Assign** module time stamps the current time (*TNOW*) in the unit entity's attribute *Delay Start Time*. The unit entity then enters the second **Seize** module and eventually succeeds in seizing the packing resource *Packer* there, possibly after waiting in its queue. At this point it enters the third **Assign** module (Delay Calculation), where its delay time in the packing buffer is computed by the expression

$$\textit{Delay in Pack\_Q} = \textit{TNOW} - \textit{Delay Start Time}$$

and stored in the unit's *Delay in Pack_Q* attribute. The computed delay times are tallied by adding the outcomes of the parenthesized conditions, which evaluate either to 1 or 0 (if true or false, respectively). For example, the variable *N*1 counts the number of delay observations strictly less than 1, *N*3 counts those that fall in the interval [1, 3], *N*5 counts those that fall in the interval [3, 5], and *NM* counts those that are greater than or equal to 5. Note that every incoming unit entity increments one and only one of the variables *N*1, *N*3, *N*5, and *NM*. Finally, the variable *No of Observations* tracks the total number of delay observations.

When the replication terminates, the requisite delay-time probabilities are computed via *Output* type statistics, defined in the **Statistic** module. *Figure 5.31.* displays the dialog spreadsheet for estimating the four requisite delay-time probabilities using suitable ratio expressions.



Statistic - Advanced Process

	Name	Type	Expression	Report Label	Output File
1	P Delay LT 1	Output	N1/No of Obsevation	P Delay LT 1	
2	P Delay LT 3	Output	N3/No of Obsevation	P Delay LT 3	
3	P Delay LT 5	Output	N5/No of Obsevation	P Delay LT 5	
4	P Delay GE 5	Output	NM/No of Obsevation	P Delay GE 5	

Figure 5.31. Dialog spreadsheet of the **Statistic** module with *Output* statistics for estimating delay probabilities in the packaging buffer.

In a similar vein, one can also estimate other sojourn-time probabilities, such as the distribution of the manufacturing lead time (the time from the start of the first operation to the completion of the last one). The implementation of such sojourn-time estimation is left as an exercise to the reader.

5.9 Model 5-6. Batch processing

In typical manufacturing environments, it is quite common to encounter processes or machines that operate on a number of jobs (product units) as a group. The group size is usually a prescribed fixed number, but it may also be random. In many cases, the group of jobs becomes a permanent assembly and continues its manufacturing process as a single unit. In some cases, the converse occurs, and a group of jobs is split into the constituent individual members. In manufacturing parlance, job groups are said to be processed in *batches*. Examples of batch processes include painting, heat treatment, packaging, various assembly and montage operations in the auto and electronics industries, as well as a variety of operations such as sterilization, shaking, and centrifugation in chemical and pharmaceutical facilities and many others. The Arena constructs that support group processing functionality are the **Batch** and **Separate** modules. The **Batch** module supports both *permanent* and *temporary* batches. Conversely, the **Separate** module recovers the constituent individual members of temporary groups; permanent groups cannot be split into their constituent members.

To illustrate batch processing, we modify the failure-modified packaging line of Section 5.7 by incorporating batching and separating, as shown in *Figure 5.32*, and refer to this system as the *batch-modified* packaging line model. Let product unit interarrival times in the batch-modified model be uniformly distributed between 5 and 10 seconds. Assume that *Labeling Process* labels batches of five units at a time, following which the units proceed separately as individual units. Assume further that batches of 10 units are packed together in *Packing Process*. Each batch labeling time is 25 seconds, and each batch packing time is 30 seconds. To enable batching and separating, we need to increase the buffer capacity at *Packing Process* to accommodate batches. For simplicity, we just set all buffer capacities to infinity, thereby eliminating blocking. However, machine failures and stoppages are retained in the batch-modified model. We are interested in the mean flow time of unit entities from their arrival at queue *Filling Process.Queue* until their arrival at module *Interdeparture Time*.

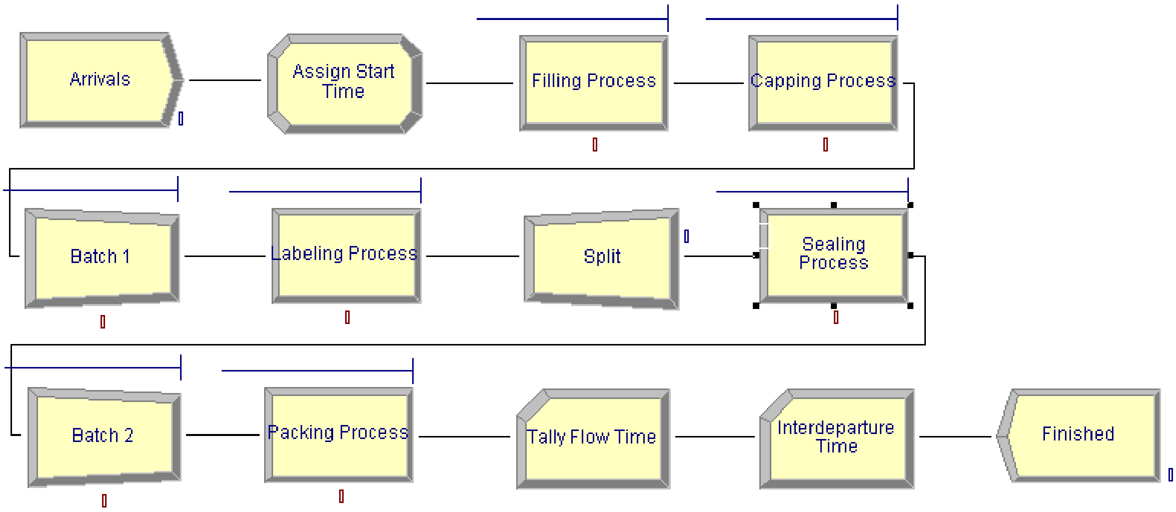
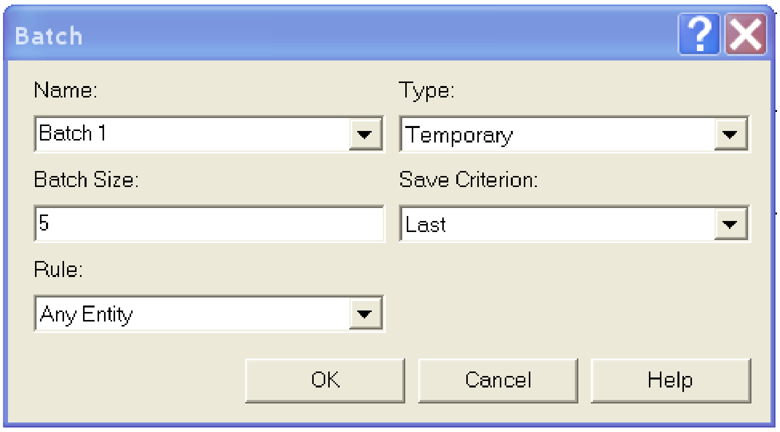


Figure 5.32. Arena model for the batch-modified packaging line system.



Batch

Name: Batch 1

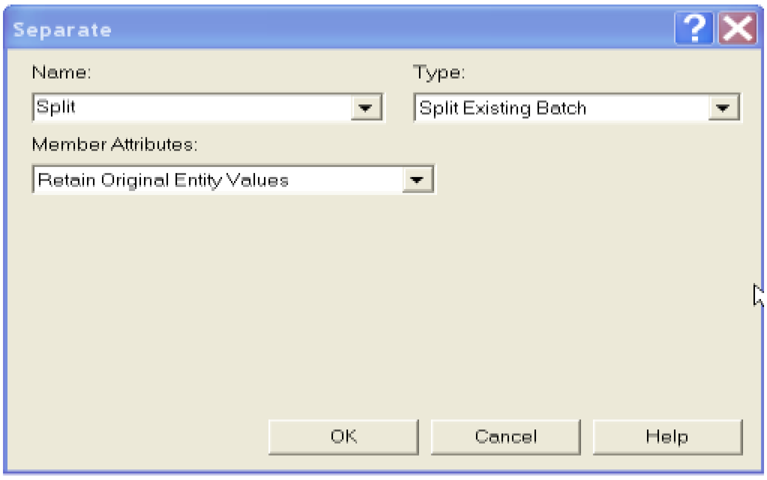
Type: Temporary

Batch Size: 5

Save Criterion: Last

Rule: Any Entity

Figure 5.33. Dialog box of the **Batch** module *Batch* 1.



Separate

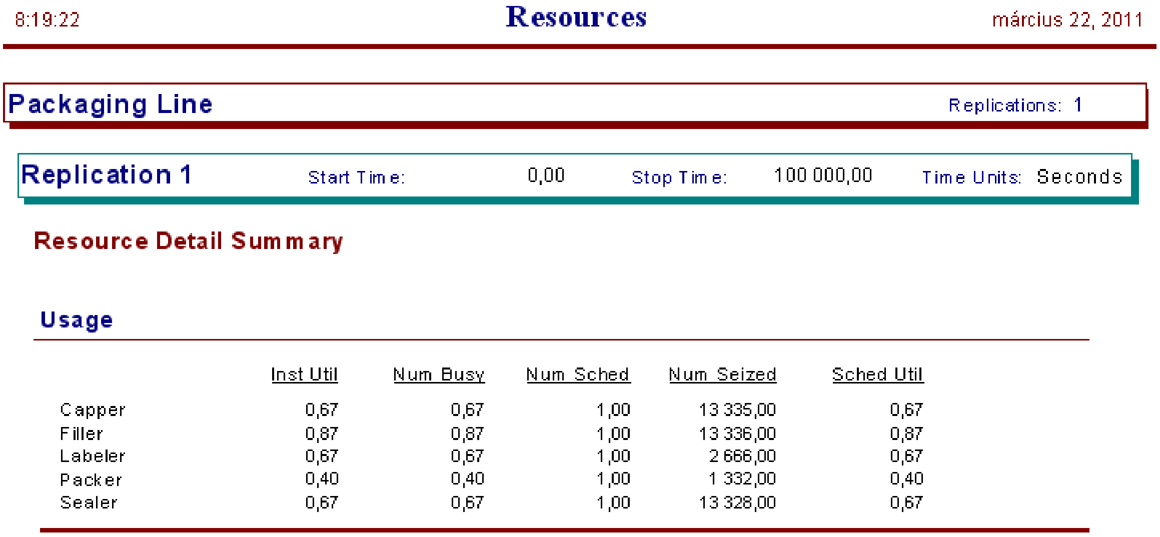
Name: Split

Type: Split Existing Batch

Member Attributes: Retain Original Entity Values

Figure 5.34. Dialog box of the **Separate** module *Split*.

Figure 5.33. displays the dialog box of the first **Batch** module following the capping operation. This **Batch** module, called *Batch* 1, collects unit entities into temporary batches (*Type* field) of size 5 (*Batch Size* field). Recall that labeling is then carried out for each batch as a whole. The *Rule* field specifies that batches will consist of any unit entity (option *Any Entity*), rather than restricting batching to unit entities with the same value of a prescribed attribute (*By Attribute* option). Next, the *Temporary* option in the *Type* field displayed in *Figure 5.33.* specifies that this batch is to be split later on (as opposed to the *Permanent* option). Finally, the *Save Criterion* field is used to select the set of attributes associated with each batch. For example, in the dialog box of *Figure 5.33*, the option *Last* was selected, indicating that batch attributes are inherited from the last unit entity of the batch (the one that completes the batch).



8:19:22 Resources március 22, 2011

Packaging Line Replications: 1

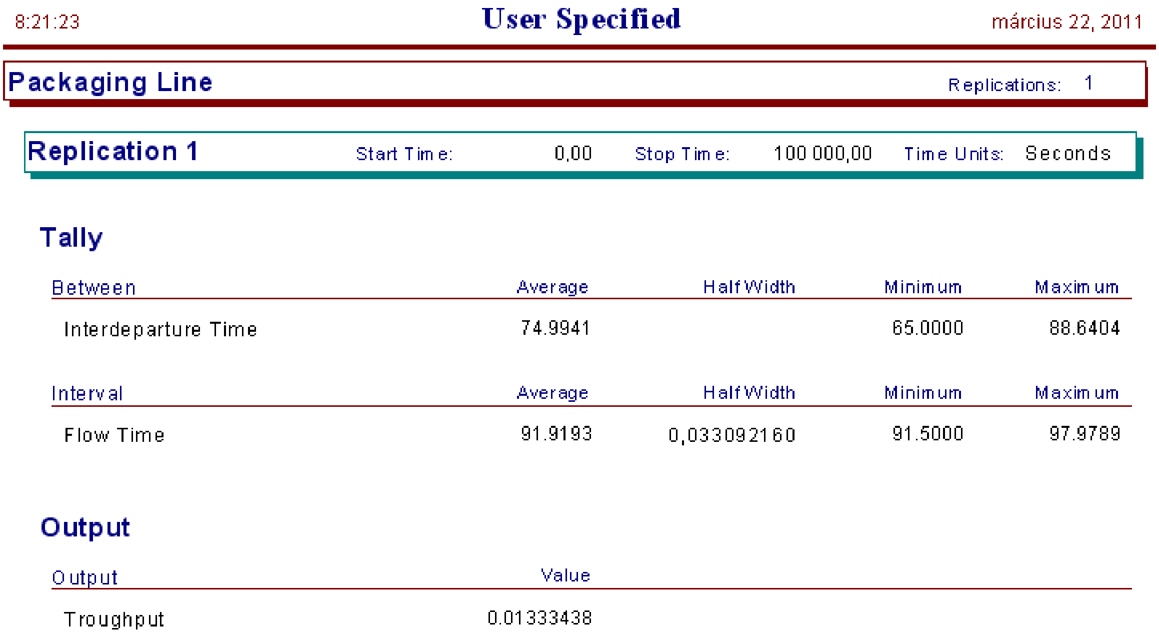
Replication 1 Start Time: 0,00 Stop Time: 100 000,00 Time Units: Seconds

Resource Detail Summary

Usage

	Inst Util	Num Busy	Num Sched	Num Seized	Sched Util
Capper	0,67	0,67	1,00	13 335,00	0,67
Filler	0,87	0,87	1,00	13 336,00	0,87
Labeler	0,67	0,67	1,00	2 666,00	0,67
Packer	0,40	0,40	1,00	1 332,00	0,40
Sealer	0,67	0,67	1,00	13 328,00	0,67

Figure 5.35. Resources report for the batch-modified model.



8:21:23 User Specified március 22, 2011

Packaging Line Replications: 1

Replication 1 Start Time: 0,00 Stop Time: 100 000,00 Time Units: Seconds

Tally

Between	Average	Half Width	Minimum	Maximum
Interdeparture Time	74.9941		65.0000	88.6404

Interval	Average	Half Width	Minimum	Maximum
Flow Time	91.9193	0,033092160	91.5000	97.9789

Output

Output	Value
Troughput	0.01333438

Figure 5.36. User Specified report for the batch-modified model.

Following the labeling operation, each batch enters the **Separate** module called *Split*, whose dialog box is displayed in *Figure 5.34*. The *Split Existing Batch* option in the *Type* field stipulates that batches are to be split back into their individual constituent members. The other *Type* field option, *Duplicate Original*, is used to create duplicates of the entity that enters this module. The option *Retain Original Entity Values* selected in the *Member Attributes* field ensures that the original attributes of all batch members will be recovered upon splitting.

After the sealing operation is completed, the units are packed at *Packing Process* into carton boxes in batches of 10. Package entities then enter additional modules, where they are counted and their system flow time is tallied before being disposed of at module *Finished*.

The simulation *Resources* report for one replication of the batch-modified model is displayed in *Figure 5.35*. Finally, the corresponding *User Specified* report is displayed in *Figure 5.36*.

5.10 Assembly operations

It is quite common in manufacturing environments to have assembly operations, where a number of parts from different buffers are assembled to produce a single unit of finished or semifinished product. Arriving matching part units are buffered in matching buffers in front of the assembly station as shown in *Figure 5.37*.

The number of units taken from the matching buffers to the assembly operation may vary from one case to case. In all cases, however, the assembly operation can start only after all the matching units are present in the matching buffers.

Arena provides a module called **Match**, which can be used for assembly of matching parts. More generally, the **Match** module is used to synchronize the movement of various unit entities in a model. For instance, the arriving matching entities may wait for each other in buffers until a batch is complete, at which time each entity resumes moving along its logical path in the model.

The **Match** module and its dialog box are shown in *Figure 5.38*. Note that the module icon contains as many matching buffers (T bars) as the value in the *Number to Match* field in the dialog box. Furthermore, there are that many connection entry points as well as exit points in the **Match** module.

The **Match** module works as follows. As soon as each buffer has at least one entity in it, precisely one entity is picked from each matching buffer, and these are sent to the associated connecting destinations. Thus, this module can be used to synchronize entity movements. In particular, the **Match** module may be used to model assembly operations by connecting exactly one exit point to the designated assembly station (the corresponding entity represents the assembled unit product), and all others to a **Dispose** module. The synchronization point is specified by the *Type* field: the *Any Entities* option synchronizes any entities in the matching buffers, while the *Based on Attribute* option selects the synchronized entities as those that have the same value in the specified attribute. For example, if entities have a color attribute, then only entities with the same color will be matched. Thus, a batch might be declared if each matching buffer contains at least one red entity. However, only one red entity is then selected from each matching buffer. If various numbers of units are assembled from matching buffers, then a **Batch** module can be used first, and then the batched units can be assembled using a **Match** module.

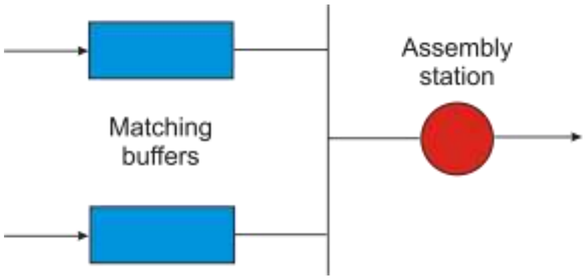
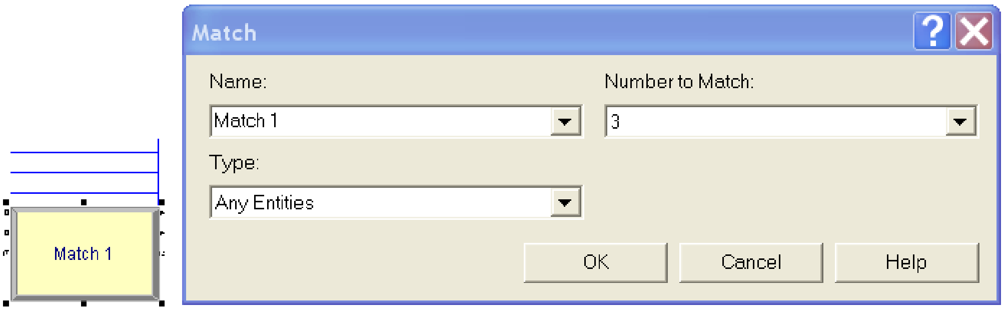


Figure 5.37. Schematic representation of an assembly operation.



Match

Name: Match 1

Number to Match: 3

Type: Any Entities

Figure 5.38. A **Match** module icon (left) and its dialog box (right).

5.11 Model verification for production lines

This section generalizes certain aspects of production lines that play an important role in understanding line behavior and verifying its logic.

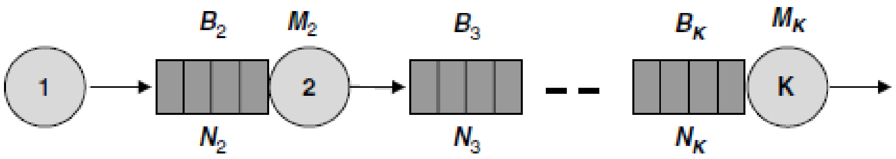


Figure 11.39. Schematic representation of a generic production line

Consider the production line depicted in *Figure 5.39*. Here, M_i denotes the i-th machine and B_i denotes the buffer between M_{i-1} and M_i with finite-capacity N_i (excluding the server positions in both machines). Let X_i be the random variable representing the processing time in M_i. We may assume that M_1 has a sufficient amount of raw material, so that it never starves. Another possibility is that M_1 has a buffer at which product units arrive in a particular manner (either randomly or according to a schedule).

Define the following probabilities:

$P_i(I)$ is the probability that M_i is idle.

$P_i(B)$ is the probability that M_i is blocked.

$P_i(D)$ is the probability that M_i is down.

$P_i(U)$ is the probability that M_i is up (producing).

Then, the utilization of M_i is given by

(5.4) $$P_i(U) = 1 - P_i(I) - P_i(B) - P_i(D)$$

indicating that M_i is in the up state when it is neither idle, nor blocked, nor down. While M_i is actually busy (with probability $P_i(U)$, it is producing at rate $1{=}E/[X_i]$ yielding a throughput $\bar{o}_i$ given by

(5.5) $$\bar{o}_i = \frac{P_i(U)}{E[X_i]}.$$

In the event that product units are not lost, the long-run flow rate (throughput) of units in a production line with K machines is the same at every machine, namely, (5.6) $$\bar{o}_1 = \bar{o}_2 = \ldots = \bar{o}_K$$

Thus, workstation throughputs are identical, each equal to the line throughput, $\bar{o}_l$. If product units are lost at a particular workstation *j* due to scrapping or rejection (with some prescribed probability), then the throughput should be adjusted accordingly. In addition, all throughputs of downstream workstations would be similarly adjusted. *Equations 5.4–5.6* can assist the modeler in verifying production line models.

Little's formula (*Eq. 8.8*) can also apply to the entire production line (see Section 5.4). Consider again the production line in *Figure 5.39.* with *K* workstations and *K*–1 buffers, and let $\bar{F}_S$ be the average total time a product unit spends in the line. Let $\bar{N}_j$ denote the average contents in buffer $\bar{B}_j$, and let $\bar{N}_S$ denote the average total number of units in the system. Little's formula is given by

(5.7) $$\bar{N}_S = \bar{o}_l \bar{F}_S$$

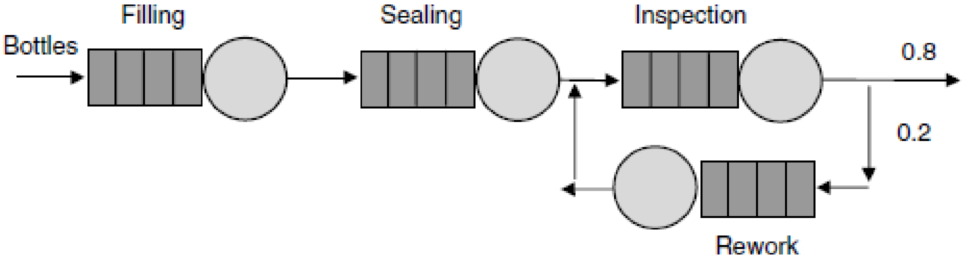
which was used in model verification in Section 5.4. On the other hand, $\bar{N}_S$ can be computed directly as

(5.8) $$\bar{N}_S = \sum_{j=2}^{K} \bar{N}_j + \sum_{j=1}^{K} P_j(U) + \sum_{j=1}^{K-1} P_j(B)$$

where the first term on the right side is the average total buffer contents of the line (excluding the first machine that has no buffer and further excluding server positions), the second term is the average total number of units in service positions while the servers are busy, and the last term is the average total number of units in server positions while the servers are blocked. For verification purposes, the modeler should check the agreement of computations for *Eqs. 5.7* and *5.8.*

EXERCISES

1. Three-stage production line. Consider the filling and inspection system consisting of three stages in series shown in the next illustration. Bottles arrive with iid exponentially distributed interarrival times with a mean of 2 minutes. They enter a filling operation with an infinite buffer in front of it to accommodate arriving bottles. The filler fills a batch of five bottles at a time. Filling times are iid uniformly distributed between 3 and 5 minutes. The entire batch of 5 bottles then attempts to join the sealer queue (of 10-bottle capacity) as individual bottles. However, if the sealer queue cannot accommodate the entire batch, the filler becomes blocked until space becomes available in the sealer queue. The filler is subject to failures with iid exponential times to failure with a mean of 50 minutes, and with iid triangularly distributed downtimes with parameters (2, 5, 10) minutes. Failures may occur at any point in time. The sealing operation is performed on a batch of three bottles at a time. It takes precisely 3 minutes to seal all three bottles. The inspection queue has a finite capacity of 10 bottles. Inspection takes precisely 1.45 minutes per bottle. Experience shows that 80% of bottles pass inspection and depart from the system, while the rest are routed to a rework station to modify the fill volume. The rework station has an infinite-capacity queue and a single server. Rework times are iid uniform between 1 and 4 minutes. After rework, bottles are routed back to the inspection queue for another inspection. However, rework bottles have a higher priority than non-rework bottles. Note that a bottle may go through rework more than once.



a) Develop an Arena model for the production line, and simulate it for 1 year, assuming 365 days of work per year and 8 hours of work per day.

b) Estimate the following statistics:

Average WIP contents in each buffer
Server utilization for each workstation
Average flow time through the system
Distribution (histogram) of flow times with five intervals
State probabilities (busy, idle, down, blocked) of every resource in the system
Probability that the filling, sealing, and rework workstations are simultaneously blocked by the inspection station

2. **Production system with inspection and rework.** Consider the following production system, which produces printed circuit boards in multiple stages with unlimited buffer capacities. Unit interarrival times are iid exponential with a mean of 2.2 hours. Arriving boards have two parts: part 1 and part 2. Upon arrival, parts separate and proceed to their respective processes separately in two branches. Part 1 units proceed along the first branch to a chemical disintegration process followed by a circuit layout 1 process, where disintegration times are iid uniform between 0.5 and 2 hours, and circuit layout times are iid uniform between 1 and 2 hours. Part 2 units proceed along the second branch to an electromechanical cleaning process followed by a circuit layout 2 process, and the corresponding processing times are iid exponentially distributed with means 1.4 and 1.5 hours, respectively. An assembly workstation then picks one unit from each branch and assembles them in an operation that lasts precisely 1.5 hours. Assembled units proceed to a testing station where batches of five units are tested simultaneously. Testing times are iid triangularly distributed with parameters 6, 8, and 10 hours. After testing, batches are split into individual units, which then depart from the system. After every 8-hour period, all processes shut down for a 1-hour maintenance operation. The following random stoppages are observed in each process.

Station Name	Failure Type	Time to Failure (hours)	Time to Repair (hours)
Disintegration	Random	Expo(1/20)	Unif(1, 2)
Cleaning	Random	Expo(1/30)	Beta(5, 1)
Layout 1	Random	Expo(1/25)	Unif(2, 5)
Layout 2	Random	Expo(1/25)	Beta(5, 1)

Recall that in Arena, the parameters of the beta distribution are in reverse order, and the parameter of the exponential distribution is the mean (not the rate).

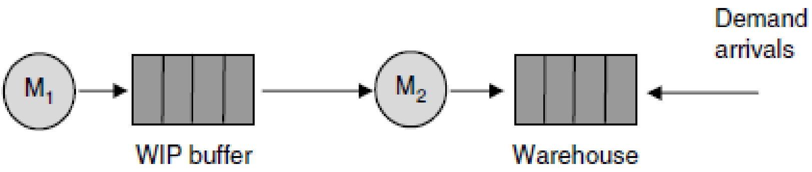
a) Develop an Arena model for the production system, and simulate it for 1 year of operation.

b) Estimate the following statistics:

Utilization of each process

Average delay at each process
Average system flow time of assembled units
Distribution of the number of jobs in the testing buffer
Probability that the system is in maintenance
State probabilities of each process (idle, busy, failed, and maintenance)

3. Two-stage manufacturing system with warehousing. Consider the following two-stage manufacturing system, where a product is produced at stage 1 and packaged at stage 2.



Customers arrive at the warehouse with product demands and their orders are filled (possibly backordered) from warehouse inventory. When the reorder point at the warehouse is reached, machine M_2 starts producing using material from the buffer, which is fed by machine M_1.

Stage 1 consists of machine M_1 and an output WIP buffer capacity of 40 units. Machine M_1 always has material to process (never starves) and takes 1 hour to process a unit. Stage 2 consists of machine M_2 and a finished product warehouse. Machine M_2 takes 0.75 hours to process a unit. Customers arrive at the finished product warehouse according to a Poisson process at the rate of 1 per 9 hours. The demand size distribution is Disc({(0.3, 3), (0.3, 6), (0.4, 12)}), and excess customer demand is backordered.

Machine M_1 is blocked whenever a finished unit cannot enter the WIP buffer due to lack of space, and remains blocked until room becomes available in the WIP buffer. Machine M_2 strives to maintain a level of 60 units in the warehouse: Production is in progress at M_2 as long as the inventory level at the warehouse is below 60, and gets blocked when that inventory level reaches 60. (Note: ware-housing is covered in the chapter 6 in more detail in the context of supply chains.)

a) Develop an Arena model for the manufacturing system, and simulate it for 10 years.

b) Estimate the following statistics:
Actual machine utilizations (excluding any idleness)
Blocking probabilities for each machine
Average inventory levels in the WIP buffer and the warehouse
Average backorder level in the finished product warehouse
Probability of backordering upon customer arrival

CHAPTER 6

Modeling Supply Chain Systems

A supply chain system (supply chain, for short) is a network that mediates the flow of entities involved in a product life cycle, from production to vending (Simchi-Levi et al. [2003]). Such a network consists of nodes and arcs. Nodes represent suppliers, manufacturers, distributors, and vendors (e.g., retail stores), as well as their inventory facilities for storing products and transportation facilities for shipping them among nodes. Arcs represent routes connecting the nodes along which goods are transported in a variety of modes (trucking, railways, airways, and so on). From a sufficiently high vantage point, supply chains have essentially a feed-forward network structure, with upstream and downstream components arranged in stages (stages), such that raw materials, parts, products, and so on flow downstream, payments flow upstream, and information flows in both directions. However, there are supply chains where entity flows may not be strictly directional because products may be returned for repair or refund. The key supply chain stages are as follows:

1. The supply stage feeds raw material or parts to the manufacturing operations.
2. The production stage converts raw material and parts to finished product.
3. The distribution stage consists of a distribution network (warehouses, distribution centers, and transportation facilities) that moves finished products to vendors.
4. The vendor stage sells products to end-customers.

In practice, the complexity of supply chains spans a broad range from simple supply chains where each stage is a single node, to complex ones where each stage is itself a complicated network consisting of a large number of nodes and arcs. For example, the production stage may be hierarchical such that upstream factories provide parts to a downstream factory that assembles the parts into more elaborate intermediate or final products. In a similar vein, a distribution network can consist of a large number of warehouses arranged hierarchically from distribution centers to wholesalers and retailers. In fact, supply chains may cross international boundaries and extend over multiple continents.

The importance of supply chains stems from the fact that they constitute a large chunk of economic activities (in the United States, supply chains contribute about 10% of gross domestic product). To improve their bottom line, gain competitive advantage, or just plain survive, companies are interested in effective and efficient supply chain operations that meet customer expectations. To this end, the discipline of supply chain management (SCM) sets itself the mission of producing and distributing products in the right quantities to the right locations at the right time, while keeping costs down and customer service levels up. In essence, SCM (also known by the older term logistics) aims to fulfill its mission by searching for good trade-offs between system costs and customer satisfaction. Thus, SCM is concerned with the efficient and cost-effective integration and coordination of the following supply chain elements:

1. Suppliers, factories, warehouses, and stores that span the nodes of a supply chain
2. The transportation network linking these nodes
3. The information technology infrastructure that enables data exchange among supply chain nodes, as well as information systems and tools that support supply chain planning, design, and day-to-day operations

SCM faces a number of challenging issues involving difficult decisions. Should the supply-chain decision making be centralized (single decision maker for the entire network that uses all available information) or distributed (multiple decision makers that use information local to their part of the network)? Are decision makers allowed to share information (transparency)? How can order-quantity variability be reduced? For example, it has been noted that lack of transparency in a supply chain gives rise to the so-called bullwhip effect, whereby the variability of orders increases as one moves up the supply chain. While these issues are company-oriented issues, supply chains also deal with customer-oriented issues involving customer satisfaction stemming from the frequency of stock-outs. For example, a customer s demand may not be fully satisfied from stock on hand. The shortage may be backordered or the sale may be lost. Either way, it is desirable to balance the cost of holding excessive inventory against the cost of not fully satisfying customer demand.

In studying such issues, modelers typically focus on the following key performance metrics, which eventually can be translated to monetary measures: (1) customer service levels (the rate of totally satisfied customer demands), (2) average inventory levels and backorder levels, (3) rate and quantity of lost sales, (4) inventory cost.

To achieve good performance, supply chains employ inventory control policies that regulate the issuing of orders to replenish stocks. Such control policies utilize the concept of level crossing as follows: An (inventory) level is said to be up-crossed if it is reached or overshot from below, and down-crossed, if it is reached or undershot from above. Inventories can be reviewed continuously or periodically, and orders are typically placed when inventory levels fall below a prescribed threshold (called reorder point). The well-known inventory policies used in industry: (t,Q), (t,S), (s,Q), (s,S), where t is the cycle time, Q is the order quantity, s is the reorder level and S is target level. In the first two cases the reorder point is determined by the cycle time. In the other cases replenishment of the inventory starts when the inventory level reaches the reorder level. Selected typical inventory control policies used in industry follow:

> **(s, S) inventory control policy**. The inventory has a target level S, and reorder point s. Replenishment of the inventory (i.e., product ordering) from a provider is suspended as soon as the inventory level up-crosses the target level, and remains suspended until it reaches the reorder point, whereupon replenishment is resumed. For example, when the provider is a production facility, suspension and resumption of replenishment mean the corresponding stopping and starting of production for the inventory.
>
> **Order up-to-*S* inventory control policy**. This is a special case of the (s, S) policy, where $s=S\geq1$. In other words, replenishment resumes as soon as the inventory down-crosses the target level. This policy is also known as the base-stock policy.
>
> **(s, Q) inventory control policy**. The inventory has an order quantity Q, and reorder point s. Whenever the inventory level down-crosses s, an order quantity of Q is placed with the provider.

Typically, when an order is placed with a provider, there is a lag in time (called lead time) after which the order is received. The lead-time demand is the magnitude of demand that materializes during lead time, and is typically random and therefore can result in stock-outs. Consequently, to mitigate the uncertainty in lead-time demand, companies carry safety stock, which is extra inventory stocked to maintain good customer service levels. Companies may also elect to place new orders before previous ones are received. The inventory position is the inventory level plus inventory on order minus backorders. Ordering decisions are typically made based on inventory positions rather than inventory levels.

Recall that the production stage of supply chains has already been treated in Chapter 11. In contrast, this chapter will focus on inventory management and material flow among stages.

6.1 A production/inventory system (Model 6-1)

This section presents a generic model of a production/inventory system consisting of a production facility that is subject to failure, which supplies a warehouse with one type of product. This generic model illustrates how an inventory control policy regulates the flow of product between production and inventory facilities.

6.1.1 Problem statement

Application of simulation will be demonstrated with a very simple production-inventory system consisting of a production facility which supplies a warehouse with one type of product. This generic model illustrates how an inventory control policy regulates the flow of product between production and inventory facilities.

Figure 1 depicts a schematic diagram of the system. The raw material storage feeds the production process, and finished product units are stored in the warehouse. Customers arrive at the warehouse with product requests (demands), and if a request cannot be fully satisfied by on-hand inventory, the unsatisfied portion represents lost sale.

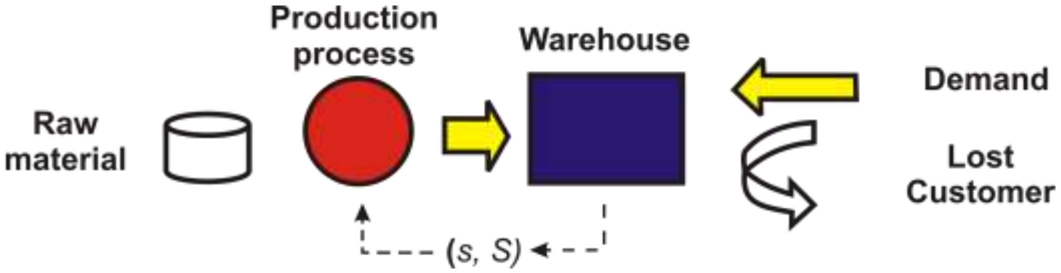


Figure 6.1 A generic production/inventory system.

The following assumptions are made:

There is always sufficient raw material in storage, so the process never starves.

Product processing is carried out in lots of five units, and finished lots are placed in the warehouse. Lot processing time is uniformly distributed between 10 and 20 minutes.

The production process experiences random failures that may occur at any point in time (see Chapter 4 for more details). Times between failures are iid exponentially distributed with a mean of 200 minutes, while repair times are iid normally distributed, with a mean of 90 minutes and a standard deviation of 45 minutes.

The warehouse operations implement the (*s, S*) inventory control policy with target level *S* = 500 units and reorder point *s* = 150 units.

The interarrival times between successive customers are iid uniformly distributed between 3 to 7 hours, and individual demand quantities are distributed uniformly between 50 and 100 units. For programming simplicity, demand quantities are allowed to be any real number in this range; however, for integer demand quantities one can use the Arena function *ANINT*(*UNIF*(50, 100)). On customer arrival, the inventory is immediately checked. If there is sufficient stock on hand, that demand is promptly satisfied. Otherwise, the unsatisfied portion of the demand is lost.

The initial inventory is 250, so the production process is initially idle.

Ordering cost has two components: *K* constant setup cost (regardless of the order quantity) is 10000 Ft/setup and *c* unit price 100 Ft/unit. The specific holding cost *h* is 2 Ft/unit/hour and the specific shortage cost *p* is 8 Ft/unit/hour.

We are interested in the following performance measures:

1. Production process utilization.
2. Downtime probability of the production facility.
3. Average inventory level at the warehouse.
4. Average demand level.
5. Percentage of customers whose demand is not completely satisfied.
6. Average lost demand quantity, given that it is not completely satisfied.
7. Average total operating (inventory) cost per hour.

The production/inventory problem described previously, though fairly elaborate, is still a gross simplification of real-life supply-chain systems. More realistic problems have additional wrinkles, including multiple types of products, multiple production stages, production setups, startups, cleanups, and so on.

6.1.2 Arena model

Having studied the problem statement, we now proceed to construct an Arena model of the system under study. *Figure 6.2* depicts our Arena model of the production/ inventory system.

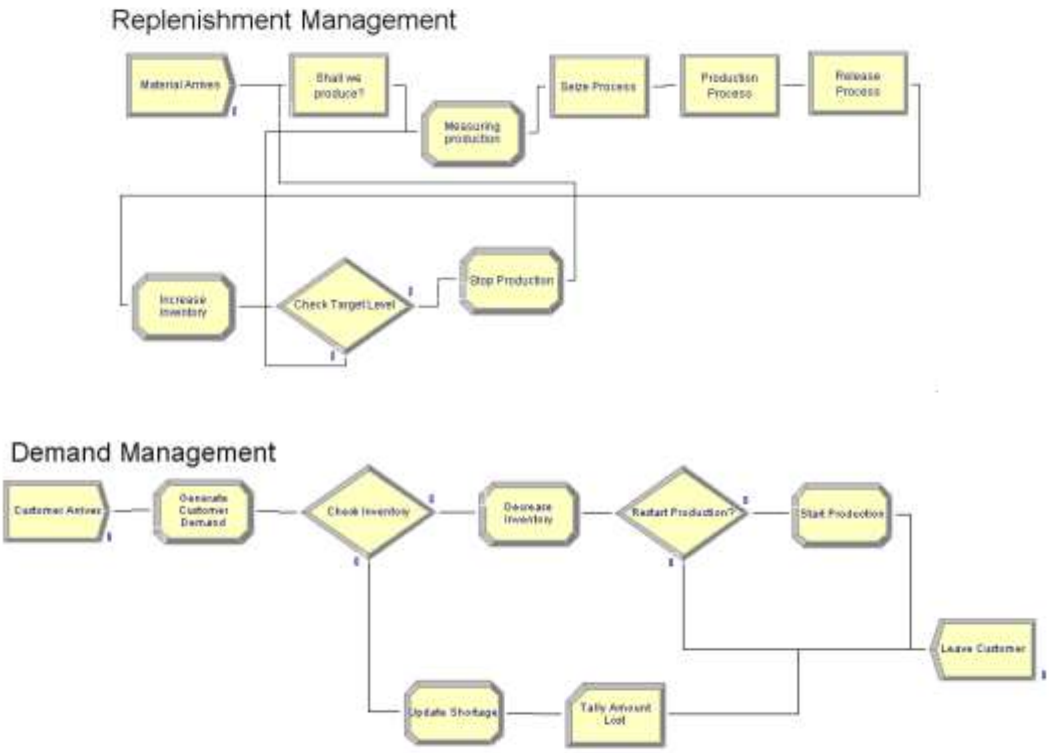


Figure 6.2 Arena model of the production/inventory system.

The model is composed of two segments:

Replenishment management segment. This segment keeps track of product unit entities. Entity definitions can be inspected and edited in the spreadsheet view of the **Entity** module from the **Basic Process** template panel. In this part of the model, the production process takes a unit of raw material from its queue, processes it as a batch of 5, and adds the finished lot to the warehouse inventory, represented by the variable *Inventory*. Thus, when a lot entity completes processing, *Inventory* is incremented by 5, and the simulation logic branches to one of two outcomes as follows: (1) If *Inventory* up-crosses the target level (variable *Target Stock*), then production stops until the reorder point (variable *Reorder Point*) is down-crossed again. (2) Processing of a new batch starts immediately when the reorder point is down-crossed. (Note that we always have sufficient raw material, so the production process never starves).

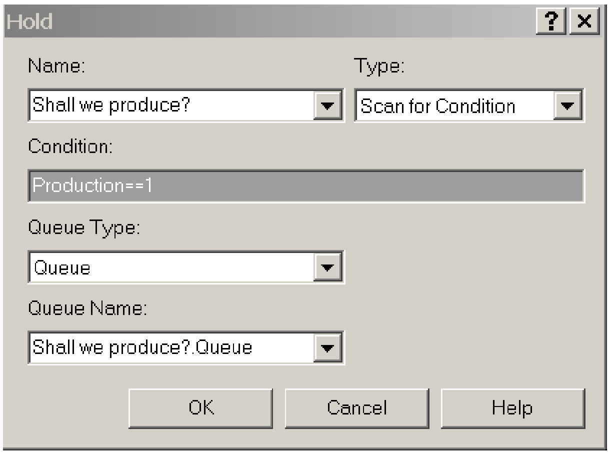
Demand management segment. This segment generates customers and their demands and adjusts variable Inventory upon customer arrival. It monitors the value of Inventory, and triggers resumption of suspended production when the reorder point is down-crossed. It also keeps track of lost demand (customers whose demand is not fully satisfied).

In addition, input and output data logic is embed in the two segments above. This logic consists of input/output modules (variables, resources, statistics, etc.) that set input variables, compute statistics, and generate summary reports. In the following sections, we proceed to examine the Arena model logic of *Figure 2* in some detail.

6.1.3 Replenishment management segment

We begin with the inventory management portion of the logic. The **Create** module, called *Raw Material*, generates product units for the production (batching) operation. A **Hold** module, called somewhat whimsically *Shall We Produce?*, serves to control the start and stop of the operation by feeding product units into a sequence of **Seize**, **Delay**, and **Release** modules (*Seize Process*, *Production Process*, and *Release Process*, respectively, in the Arena model), all drawn from the **Advanced Process** template panel. The actual processing (production in our case) takes place at the **Delay** module, called *Production Process*, where the production time of a batch is specified as *Unif*(10, 20) minutes.

The **Create** module *Raw Material* was initialized by populating it with a single product entity (at time 0), and therefore the interarrival time is irrelevant. Furthermore, by setting the Max Arrivals field to 1, the **Create** module will deactivate itself thereafter. The product entity circulates in the model repeatedly, with each cycle representing a production cycle.


Hold

Name: Shall we produce?

Type: Scan for Condition

Condition: Production==1

Queue Type: Queue

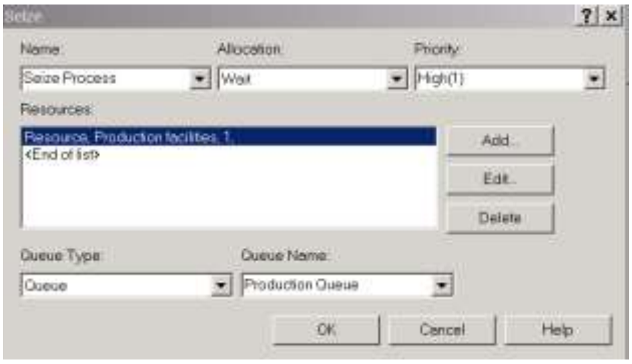
Queue Name: Shall we produce?.Queue

Figure 6.3 Dialog box of the **Hold** module *Shall We Produce?*

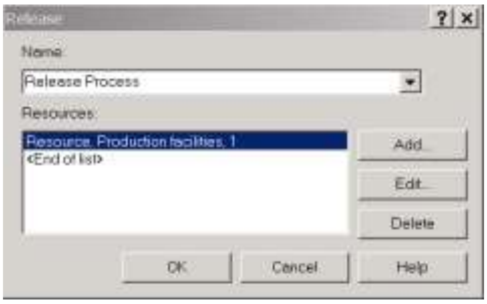
The circulating product entity then proceeds to the **Hold** module, called *Shall We Produce?*, to test if production is turned on. The state of production (*Off*= 0 or *On* = 1) is maintained in the variable *Production*, which is initially set to 0. Recall that the **Hold** module performs a gating function on an entity by scanning for the truth or falsity of a logical condition, as shown in the Hold dialog box in *Figure 6.3*. If the condition (in our case, *Production* = 1) is true, then the product entity proceeds to the next module. Otherwise, it waits in queue *Shall*

We Produce?.Queue until the condition becomes true before being allowed to proceed. The circulating product entity then proceeds to queue *Production Queue* in the *Seize Process* module and seizes the server immediately (being the only one in contention for the *Production facilities* resource).

Figure 6.4 depicts the dialog boxes of the **Seize** and **Release** modules, *Seize Process* and *Release Process*, which contain resource information. The **Seize** module has a queue called *Production Queue*, where product entities await their turn for one unit of resource *Production facilities* to perform the packing operation at module *Production Process*. Once the resource becomes available, it is seized by the highest-ranking product (in this case, rank 1). While the current process is in progress, the **Seize** module bars any additional product entities from entering it. On service completion, the **Release** module (which never denies entry) releases one unit of resource *Production Process*, as indicated by the Release dialog box in *Figure 6.4*. Note that an entity may seize a number of resources (when available) simultaneously. Further, an entity may release a number of owned resources simultaneously.



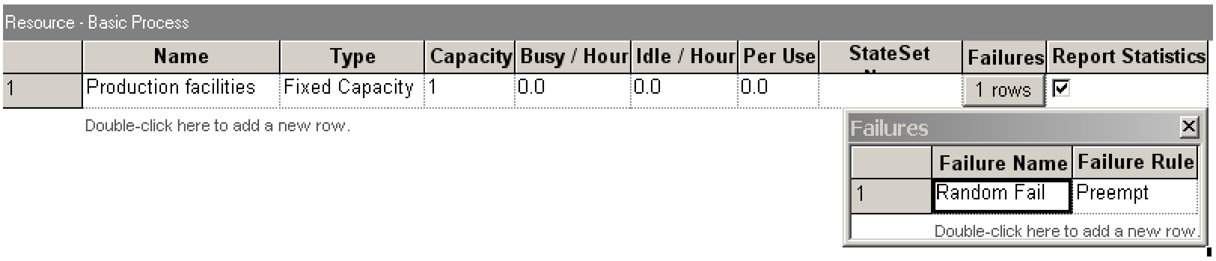
Name: Seize Process
Allocation: Wait
Priority: High(1)
Resources:
Resource, Production facilities, 1
<End of list>
Queue Type: Queue
Queue Name: Production Queue



Name: Release Process
Resources:
Resource, Production facilities, 1
<End of list>

Figure 6.4 Dialog spreadsheets of the **Seize** module *Seize Process* (left) and **Release** module *Release Process* (right).

A spreadsheet view of the **Resource** module with an entry for *Production Process* is shown at the bottom of *Figure 6.5* with its Failures field (associating resources to failures) popped up (top). All failure/repair data (uptimes and downtimes) are specified in the **Failure** spreadsheet module (see Chapter 4), as illustrated in *Figure 6.6*.



Resource - Basic Process

	Name	Type	Capacity	Busy / Hour	Idle / Hour	Per Use	StateSet N	Failures	Report Statistics
1	Production facilities	Fixed Capacity	1	0.0	0.0	0.0		1 rows	☑

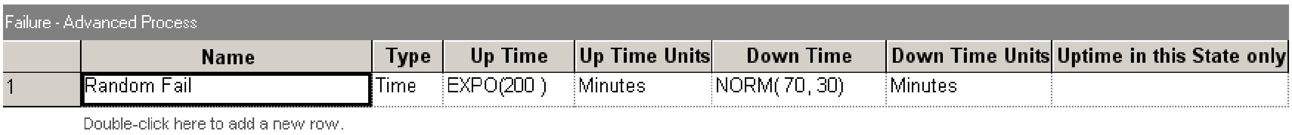
Double-click here to add a new row.

Failures

	Failure Name	Failure Rule
1	Random Fail	Preempt

Double-click here to add a new row.

Figure 6.5 Dialog spreadsheet of the **Resource** module showing resource *Production Process* (bottom) with its Failures dialog spreadsheet (top).



Failure - Advanced Process

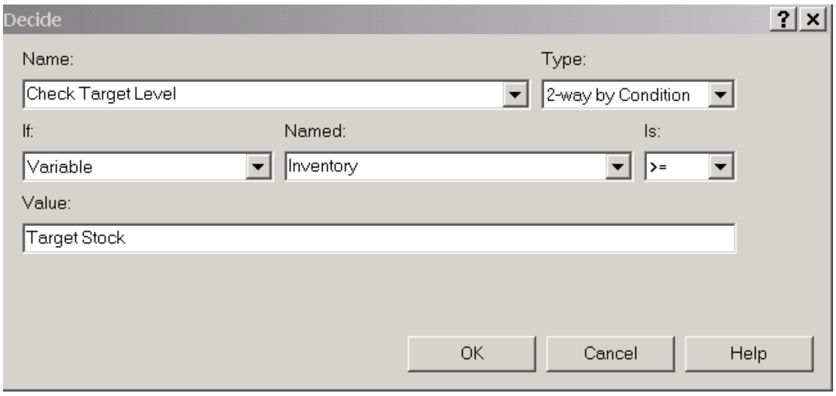
	Name	Type	Up Time	Up Time Units	Down Time	Down Time Units	Uptime in this State only
1	Random Fail	Time	EXPO(200)	Minutes	NORM(70, 30)	Minutes	

Double-click here to add a new row.

Figure 6.6 Dialog spreadsheet of the **Failure** module.

The inventory level at the warehouse is maintained by the variable *Inventory*, initially set to 250. Whenever the circulating product entity (batch of five units) enters the **Assign** module, called *Update Inventory*, it adds a batch of five finished units (variable *Batch Size*) to the warehouse inventory by just incrementing Inventory by the batch size.

The circulating product entity then proceeds to the **Decide** module, called *Check Target Level*, whose dialog box is shown in *Figure 6.7*. Here, the circulating product entity tests whether the inventory target level has been up-crossed. The test can result in two outcomes:



Decide

Name: Check Target Level

Type: 2-way by Condition

If: Variable

Named: Inventory

Is: >=

Value: Target Stock

Figure 6.7 Dialog box of the **Decide** module *Check Target Level*.

If the inventory target level has been up-crossed, then the circulating product entity moves on to the following **Assign** module, called *Stop Production*, and sets *Production* = 0 to signal that production is suspended.

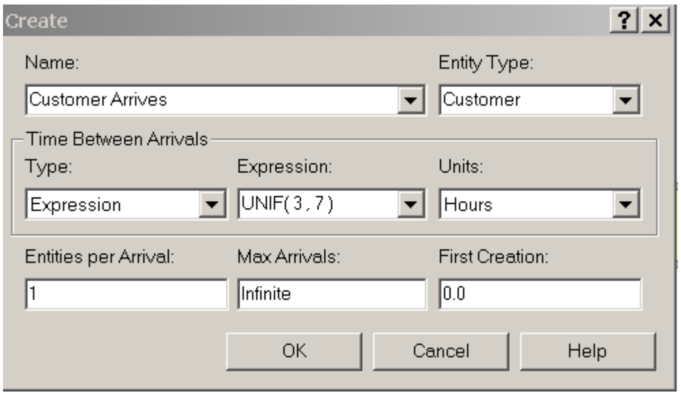
Otherwise, the circulating product entity does nothing.

Either way, the circulating product entity has completed its sojourn through the system and would normally be disposed of (at a **Dispose** module). However, since the *Production Process* module is never starved and no delay is incurred since its departure from the *Production Process* module, we can “recycle” the circulating product entity by always sending it back to the *Production Process* module to play the role of a new arrival. This modeling device is logically equivalent to disposing of a product entity and creating a new one. However, it is computationally more efficient, since it saves us this extra computational effort, so that the simulation will run faster. We recommend that model run optimizations by means of circulating entities (or any other logically correct modeling device) should be routinely sought to speed up simulation runs. The emphasis here is on “logically correct” optimizations. Note carefully that if the production operation were allowed to starve, then our previous computational “shortcut” would be invalid, and we would have to dispose of and create new entities throughout the simulation run.

6.1.4 Demand management segment

In this section, we continue with the demand management segment of the logic.

The source of customer demand at the warehouse is the **Create** module, called *Customer Arrives*, whose dialog box is depicted in *Figure 6.8*. The arrival pattern of customers is specified to be random with interarrival time distribution *Unif*(3, 7). Each arrival is an entity of type *Customer* (customer entity) with its private set of attributes (e.g., a demand quantity attribute, called *Demand*). On arrival, a customer entity first enters the **Assign** module, called *Customer Demand*, where its *Demand* attribute is assigned a random value from the *Unif*(50, 100) distribution. The customer entity then proceeds to the **Decide** module called *Check Inventory* to test whether the warehouse has sufficient inventory on hand to satisfy its demand. The test can result in two outcomes:



Create

Name: Customer Arrives

Entity Type: Customer

Time Between Arrivals

Type: Expression

Expression: UNIF(3 , 7)

Units: Hours

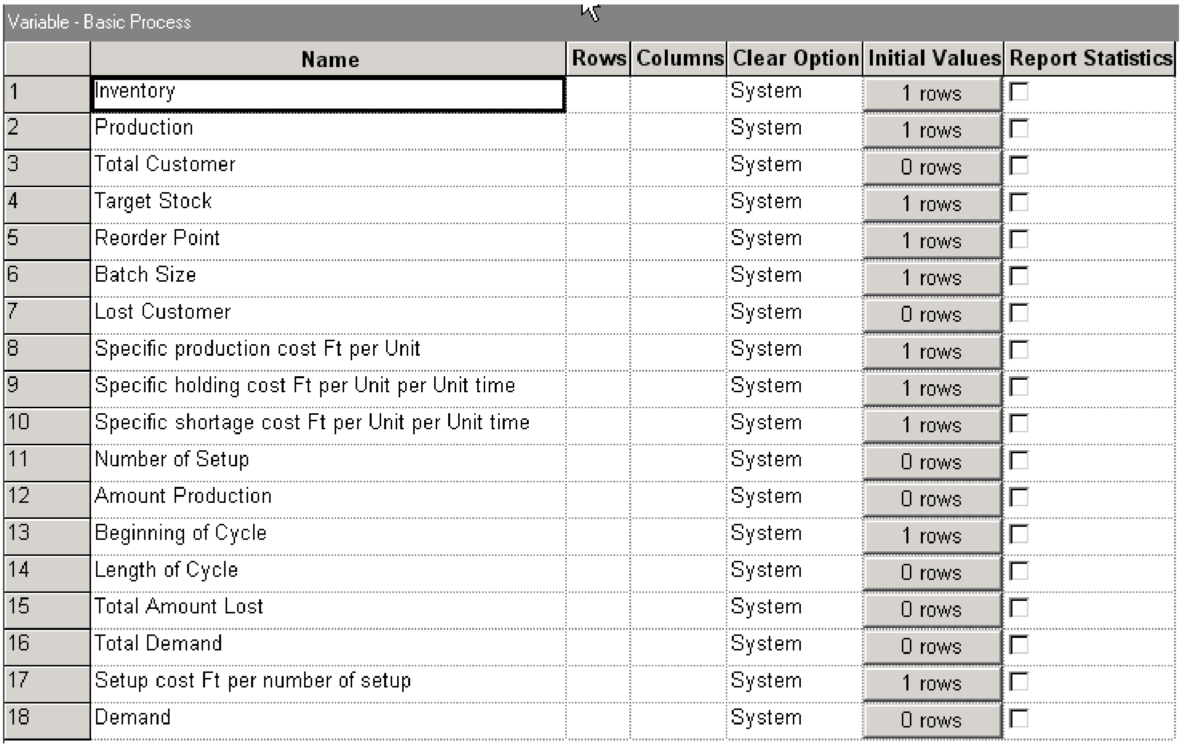
Entities per Arrival: 1

Max Arrivals: Infinite

First Creation: 0.0

Figure 6.8 Dialog box of the **Create** module *Customer Arrives*.

(1) If the value of variable *Inventory* is greater or equal to the value of attribute *Demand*, then the current demand can be satisfied and the customer entity takes the *True* exit to the **Assign** module, called *Decrease Inventory*, where it decrements the inventory by the demand amount. It next proceeds to the **Decide** module, called *Restart Production?*, to test whether the *Reorder Point* variable has just been down-crossed. If it has, the customer entity proceeds to the **Assign** module, called *Start Production*, to set *Production*=1, which would promptly release the circulating product entity currently detained in the **Hold** module *Shall We Produce?*, effectively resuming the production process. Either way, the customer entity proceeds to be disposed of at the **Dispose** module, called *Leave Customer*.



Variable - Basic Process

	Name	Rows	Columns	Clear Option	Initial Values	Report Statistics
1	Inventory			System	1 rows	□
2	Production			System	1 rows	□
3	Total Customer			System	0 rows	□
4	Target Stock			System	1 rows	□
5	Reorder Point			System	1 rows	□
6	Batch Size			System	1 rows	□
7	Lost Customer			System	0 rows	□
8	Specific production cost Ft per Unit			System	1 rows	□
9	Specific holding cost Ft per Unit per Unit time			System	1 rows	□
10	Specific shortage cost Ft per Unit per Unit time			System	1 rows	□
11	Number of Setup			System	0 rows	□
12	Amount Production			System	0 rows	□
13	Beginning of Cycle			System	1 rows	□
14	Length of Cycle			System	0 rows	□
15	Total Amount Lost			System	0 rows	□
16	Total Demand			System	0 rows	□
17	Setup cost Ft per number of setup			System	1 rows	□
18	Demand			System	0 rows	□

Figure 6.9 Dialog spreadsheet of the **Variable** module.

(2) If the value of variable *Inventory* is strictly smaller than the value of attribute *Demand*, then the current demand is either partially satisfied or not at all. Either way, the customer entity proceeds to the **Assign** module, called *Update Shortage*, where it sets the *Inventory* variable to 0. It also updates the variable *Lost Customer*, which keeps track of the number

of customers to-date whose demand could not be fully satisfied (*Lost Customer*=*Lost Customer*+1), and the attribute called *Amount Lost*, which keeps track of the demand lost by the current customer (*Amount Lost* =*Demand–Inventory*). The customer entity next enters the **Record** module, called *Tally Amount Lost*, to tally the lost quantity per customer whose demand was not fully satisfied. Finally, the customer entity proceeds to be disposed of at module *Leave Customer*.

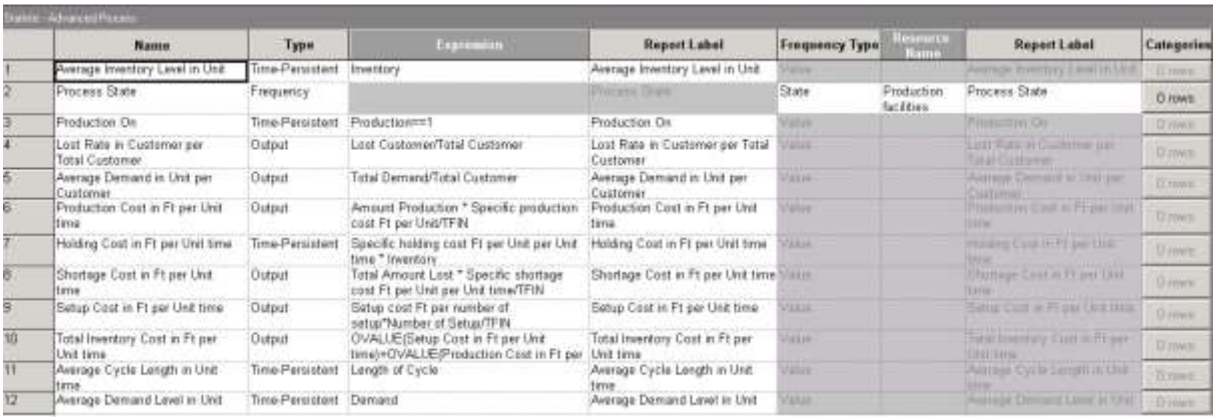
The user-defined variables involved in the model can be set or inspected by clicking on the **Variable** module of the **Basic Process** template panel. This yields the spreadsheet view of the module, as exemplified in *Figure 6.9*.

6.1.5 Statistics collection

The **Statistic** module is used to define nonstandard additional statistics (it is called *User Specified Statistic*) that are to be collected during the simulation and also to specify output data files.

Figure 6.10 displays the dialog spreadsheet of the **Statistic** module for the production-inventory model. It includes 4 Time-Persistent (DSTAT) type statistic, called *Average Inventory Level in Unit*, for the *Inventory* variable, *Average Demand Level in Unit*, for the *Demand* variable, *Average Cycle Length in Unit time*, for the *Length of Cycle* variable, and *Production On*, for the expression *Production*=1. The Time-Persistent type statistical outputs consist of the average value, 95% confidence interval, and minimal and maximal values of the variables ever observed during the replication. For example, the statistical output for *Production*=1 is the rate of time when this expression is true, or that is the probability that the production process is set to active. (Keep in mind, however, that the production process may experience downtimes.) Recall that this method of time averaging expressions can be used to estimate any probability of interest, including joint probabilities. Statistics collection can also be affected by checking the requisite boxes under the Report Statistics column in the **Variable** module (see *Figure 6.9*).

Figure 6.10 also contains a *Frequency* type statistic, called *Process States*, which estimates the state probabilities of the production process, namely, the probabilities that the production process is busy or down (recall that, in fact, the production process is never idle).

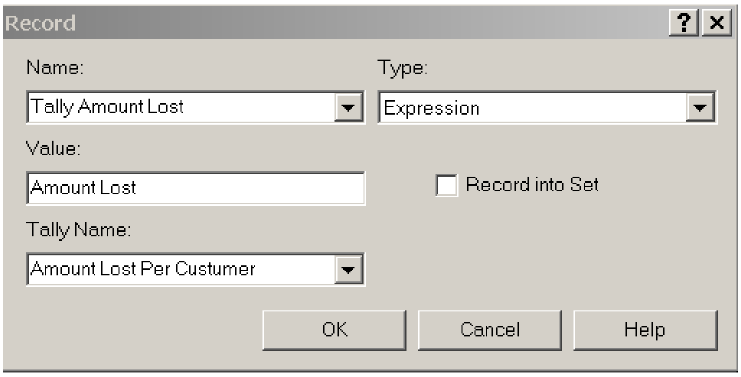


Statistic - Advanced Process

	Name	Type	Expression	Report Label	Frequency Type	Resource Name	Report Label	Categories
1	Average Inventory Level in Unit	Time-Persistent	Inventory	Average Inventory Level in Unit	Value		Average Inventory Level in Unit	0 rows
2	Process State	Frequency		Process State	State	Production facilities	Process State	0 rows
3	Production On	Time-Persistent	Production==1	Production On	Value		Production On	0 rows
4	Lost Rate in Customer per Total Customer	Output	Lost Customer/Total Customer	Lost Rate in Customer per Total Customer	Value		Lost Rate in Customer per Total Customer	0 rows
5	Average Demand in Unit per Customer	Output	Total Demand/Total Customer	Average Demand in Unit per Customer	Value		Average Demand in Unit per Customer	0 rows
6	Production Cost in Ft per Unit time	Output	Amount Production * Specific production cost Ft per Unit/TFIN	Production Cost in Ft per Unit time	Value		Production Cost in Ft per Unit time	0 rows
7	Holding Cost in Ft per Unit time	Time-Persistent	Specific holding cost Ft per Unit per Unit time * Inventory	Holding Cost in Ft per Unit time	Value		Holding Cost in Ft per Unit time	0 rows
8	Shortage Cost in Ft per Unit time	Output	Total Amount Lost * Specific shortage cost Ft per Unit per Unit time/TFIN	Shortage Cost in Ft per Unit time	Value		Shortage Cost in Ft per Unit time	0 rows
9	Setup Cost in Ft per Unit time	Output	Setup cost Ft per number of setup*Number of Setup/TFIN	Setup Cost in Ft per Unit time	Value		Setup Cost in Ft per Unit time	0 rows
10	Total Inventory Cost in Ft per Unit time	Output	OVALUE(Setup Cost in Ft per Unit time)+OVALUE(Production Cost in Ft per	Total Inventory Cost in Ft per Unit time	Value		Total Inventory Cost in Ft per Unit time	0 rows
11	Average Cycle Length in Unit time	Time-Persistent	Length of Cycle	Average Cycle Length in Unit time	Value		Average Cycle Length in Unit time	0 rows
12	Average Demand Level in Unit	Time-Persistent	Demand	Average Demand Level in Unit	Value		Average Demand Level in Unit	0 rows

Figure 6.10 Dialog spreadsheet of the **Statistic** module.

The first from the six *Output* type statistics in *Figure 6.10* used to calculate the lost rate of customers that suffered some lost demand. The corresponding Expression field there indicates that when the replication terminates, the variable *Lost Customer* is divided by the variable *Total Customer* to yield the requisite rate. The other *Output* type statistics calculate the elements of inventory cost and the total inventory cost.



Name: Tally Amount Lost

Type: Expression

Value: Amount Lost

□ Record into Set

Tally Name: Amount Lost Per Custumer

Figure 6.11 Dialog box of the **Record** module *Tally Lost Amount*.

Figure 6.11 displays the dialog box of the **Record** module, called *Tally Amount Lost*. Whenever a customer entity enters this module, the expression (in this case, the variable) *Amount Lost* is evaluated and the resultant value is tallied. When the replication terminates, the output report will contain a *Tallies* section summarizing the average amount lost per customer. Note carefully that since we tally a loss only when it occurs, this average is conditional on the event that a customer actually experienced a loss. If we were to include in the tally a value of 0 (no loss) for every customer with a fully satisfied demand, then the resulting average would provide us with the unconditional average of lost demand per customer. Obviously, the conditional average loss must be larger than the unconditional average loss.

6.1.6 Simulation output

Figure 6.12 displays reports of the results of a simulation run of length 20,000 hours (more than 2 years). The results in *Figure 6.12* give quite instructive information about performance of the production-inventory model.

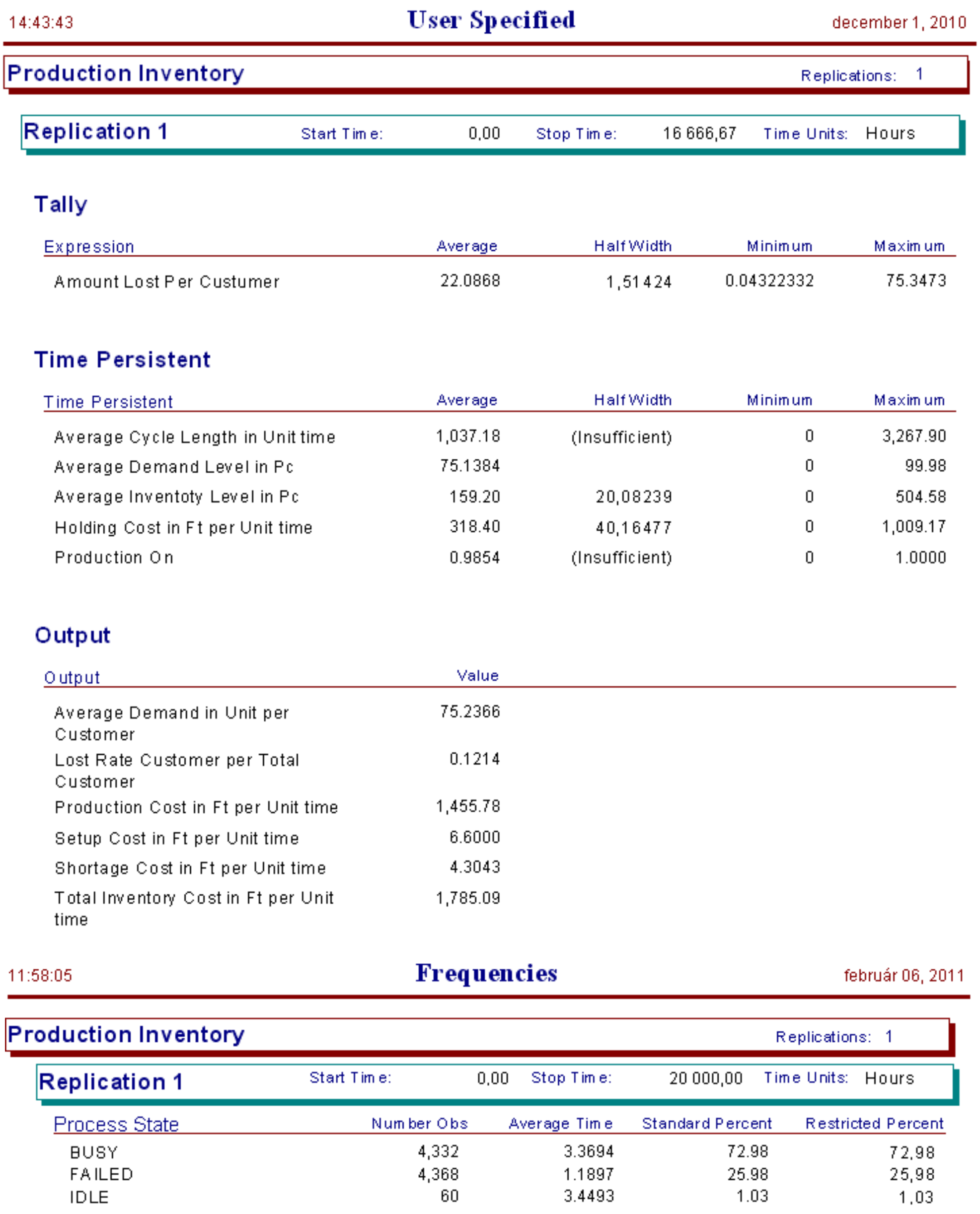
The expression *Amount Lost Per Customer* in the Tally section indicates that the average lost demand per customer who experienced some loss was about 22 units. The Half Width column estimates the half-length of respective confidence intervals at the 95% confidence level.

The *Average Inventory Level in Unit* statistic in the Time Persistent section shows that occasionally the inventory up-crossed its target level. However, this happened only rarely, because the *Production* variable in the Time Persistent section assumed the value *On*=1 about 98.54% of the time. The average inventory level is 160.54 units. It means that often the inventory level was lower, than the reorder level, *s*=150. It seems in *Figure 6.13* which shows the changing of the inventory and demand level in time.

The Output statistic *Lost Rate in Customer per Total Customer* in the Output section reveals that 12.02% of the customers had their demand either partially satisfied or not satisfied at all.

The *Total Inventory Cost in Ft per Unit time* also in the Output statistic is 1790.84 Ft/hour. Its components: the *Setup Cost in Ft per Unit time* is 6 Ft/hour, the *Production Cost in Ft per Unit time* is 1455.78 Ft/hour, the *Holding Cost in Ft per Unit time* is 321.09 Ft/hour, and the *Shortage Cost in Ft per Unit time* is 4.27 Ft/hour.

Finally, the Frequencies section estimates the state probabilities (BUSY, FAILED, and IDLE) of the production process. These are, respectively, 72.98% (resource utilization), 25.98% (downtime probability), and 1.03% (idle probability). Note that the average uptime and the average downtime are very close to their theoretical values (in this case, the downtime probability is equal to the ratio of expected downtime to expected cycle length).



14:43:43 **User Specified** december 1, 2010

Production Inventory Replications: 1

Replication 1 Start Time: 0,00 Stop Time: 16 666,67 Time Units: Hours

Tally

Expression	Average	Half Width	Minimum	Maximum
Amount Lost Per Custumer	22.0868	1,51424	0.04322332	75.3473

Time Persistent

Time Persistent	Average	Half Width	Minimum	Maximum
Average Cycle Length in Unit time	1,037.18	(Insufficient)	0	3,267.90
Average Demand Level in Pc	75.1384		0	99.98
Average Inventoty Level in Pc	159.20	20,08239	0	504.58
Holding Cost in Ft per Unit time	318.40	40,16477	0	1,009.17
Production On	0.9854	(Insufficient)	0	1.0000

Output

Output	Value
Average Demand in Unit per Customer	75.2366
Lost Rate Customer per Total Customer	0.1214
Production Cost in Ft per Unit time	1,455.78
Setup Cost in Ft per Unit time	6.6000
Shortage Cost in Ft per Unit time	4.3043
Total Inventory Cost in Ft per Unit time	1,785.09

11:58:05 **Frequencies** február 06, 2011

Production Inventory Replications: 1

Replication 1 Start Time: 0,00 Stop Time: 20 000,00 Time Units: Hours

Process State	Number Obs	Average Time	Standard Percent	Restricted Percent
BUSY	4,332	3.3694	72.98	72,98
FAILED	4,368	1.1897	25.98	25,98
IDLE	60	3.4493	1.03	1,03

Figure 6.12. Simulation results for the production/inventory model.

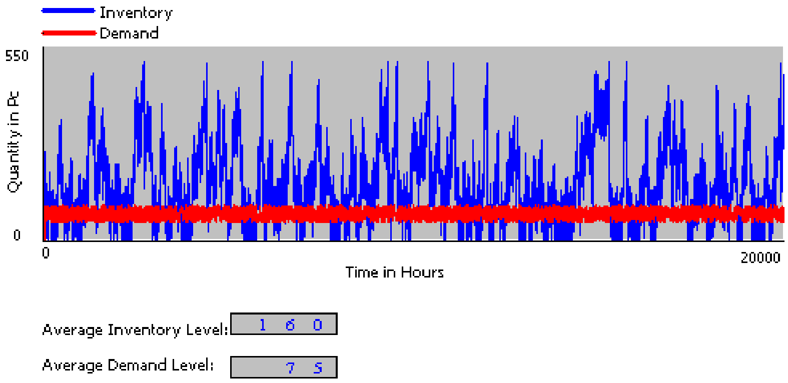
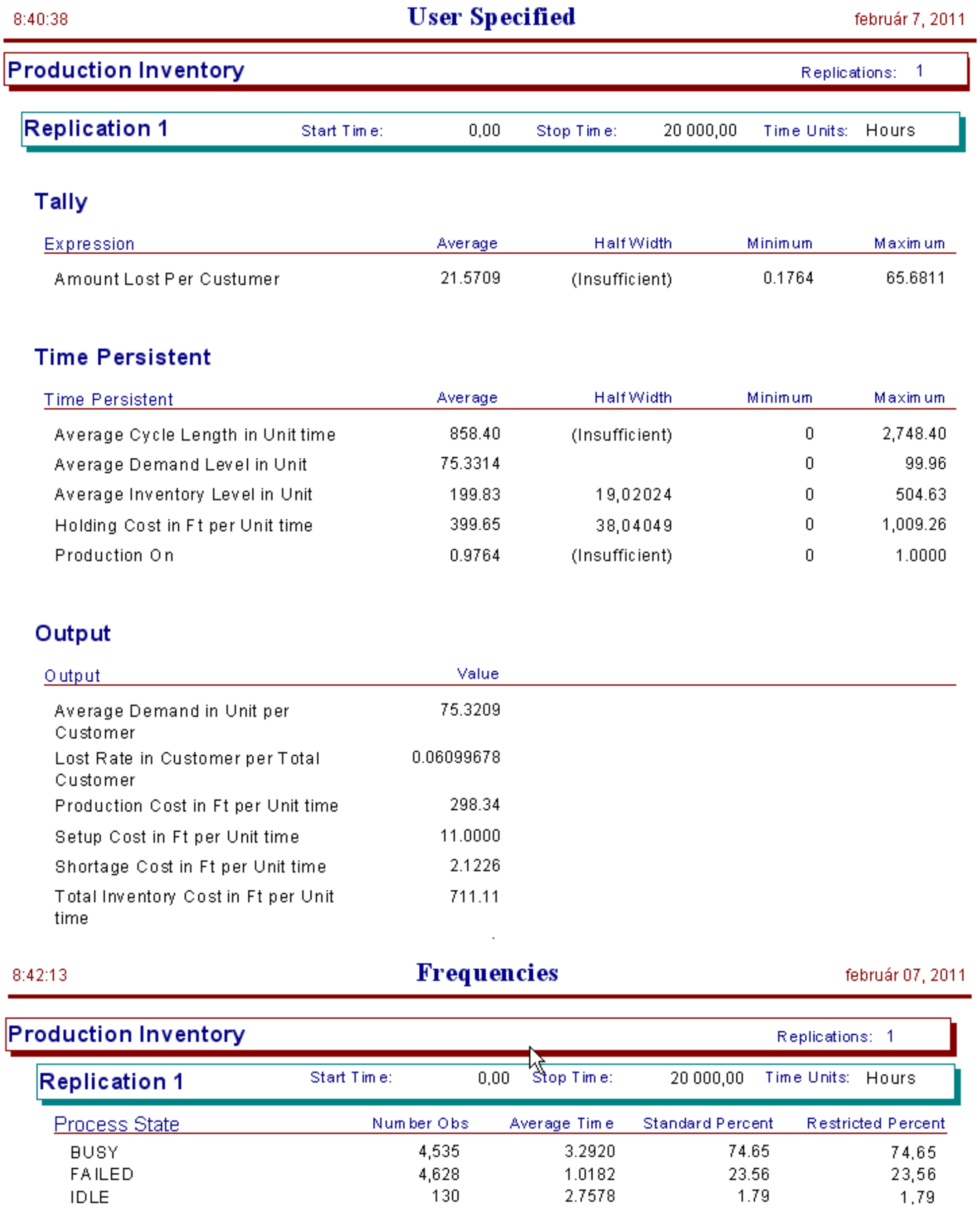


Figure 6.13. *Inventory* and *Demand* level as function of the simulation time.

6.1.7 Experimentation and analysis

In this section, we will experiment with selected model parameters, so as to improve the customer service level of fill rate (probability that the demand of an arriving customer is fully satisfied) and to decrease the inventory total cost. Clearly, the fill rate is quite low. Our performance metric of interest is the complementary probability of partially or fully unsatisfied demands. *Figure 6.12* estimates this probability as 12.14% (the value of Lost Rate Customer per Total Customer in the *Output* section). How can we modify the system to decrease this probability to an "acceptable" level? In inventory-oriented systems such as the one under consideration, the only usually way to increase the fill rate is to increase the level of inventory on hand. In spite of this in our case, we may attempt to achieve this goal by other way.



8:40:38 **User Specified** február 7, 2011

Production Inventory Replications: 1

Replication 1 Start Time: 0,00 Stop Time: 20 000,00 Time Units: Hours

Tally

Expression	Average	Half Width	Minimum	Maximum
Amount Lost Per Custumer	21.5709	(Insufficient)	0.1764	65.6811

Time Persistent

Time Persistent	Average	Half Width	Minimum	Maximum
Average Cycle Length in Unit time	858.40	(Insufficient)	0	2,748.40
Average Demand Level in Unit	75.3314		0	99.96
Average Inventory Level in Unit	199.83	19,02024	0	504.63
Holding Cost in Ft per Unit time	399.65	38,04049	0	1,009.26
Production On	0.9764	(Insufficient)	0	1.0000

Output

Output	Value
Average Demand in Unit per Customer	75.3209
Lost Rate in Customer per Total Customer	0.06099678
Production Cost in Ft per Unit time	298.34
Setup Cost in Ft per Unit time	11.0000
Shortage Cost in Ft per Unit time	2.1226
Total Inventory Cost in Ft per Unit time	711.11

8:42:13 **Frequencies** február 07, 2011

Production Inventory Replications: 1

Replication 1 Start Time: 0,00 Stop Time: 20 000,00 Time Units: Hours

Process State	Number Obs	Average Time	Standard Percent	Restricted Percent
BUSY	4,535	3.2920	74.65	74,65
FAILED	4,628	1.0182	23.56	23,56
IDLE	130	2.7578	1.79	1,79

Figure 6.14 Simulation results for the production/inventory model with reduced downtimes.

We increase the fill rate is to modify the original production/ inventory system by investing more in maintenance activities. This will reduce downtimes and make the process more available for production. *Figure 6.14* displays the estimated performance changes in the modified production/inventory model where the average repair time is reduced to 60 minutes with a standard deviation of 25 minutes. The reduction in repair time in *Figure 6.14* represents an improvement of about 14% in maintenance activities, as compared with the downtime probabilities in *Figure 6.6*. The resulting reduction in the percentage of partially or fully unsatisfied demands has dropped significantly from 12.14% to 6.09%. This reduction represents a more than 49% improvement in the fill rate.

This analysis leads us to conclude that a significant improvement in the fill rate can be achieved by improving the production process, rather than by modifying our inventory replenishment policy.

As we mentioned before, goal of SMC is to achieve economic balance between system costs and customer satisfaction. It known that simulation itself does not solve this problem directly, however, we can vary the input parameters of the system and we can observe the changes of the outputs. For example, we vary the value of *Reorder point* between 50 and 150 units, because we would like to decrease the inventory cost and to improve the service level (probability that the demand of an arriving customer is fully satisfied) and the utilization of the production facility. *Table 6.1* consists of the simulation results. Increasing *Reorder point* until 100 units results decreasing costs, improving facility utilization and the lost rate does not change significantly. *Reorder point*=100 units can be considered as an optimum, because above 100 units the cost increases and the facility utilization marginally falls down. This simple analysis leads us to conclude that a significant cost decrease can be achieved by choosing the appropriate reorder point.

Table 6.1

Cost analysis of the production-inventory system

Reorder Point in Unit	Holding cost in Ft/hour	Shortage cost in Ft/hour	Setup Cost in Ft/hour	Total Inventory Cost in Ft/hour	Lost Rate in %	Avg. Cycle Time in hour	Production on in %
50	349.87	3.21	6.0	1819,30	9.89	1314.49	98.25
75	288.09	4.20	4.0	1753,72	12.80	1532.36	98.91
100	296.68	3.75	3.5	1763,49	11.63	2477.92	99.02
125	303.74	4.35	4.0	1769,84	12.64	2261.50	98.94
150	321.09	4.27	6.0	1790,84	12.02	980.64	98.68

Much more information can be gleaned from the statistical output of *Figure 6.12*, but these items will not be pursued further (we encourage you to complete the analysis of the simulation results).

12.2 Multiproduct production/inventory system (Model 6-2)

This section extends the production/inventory system studied in Section 6.1 from a single product to multiple ones. The new system again operates under the lost sales discipline.

6.2.1 Problem statement

The extended system is depicted in *Figure 6.15*. Accordingly, the production facility produces product types 1,2, and 3, and these are supplied to three distinct incoming customer streams, denoted by types 1, 2, and 3, respectively. The production facility produces batches of products, switching from production of one product type to another, depending on inventory levels. However, products have priorities in production, with product 1 having the highest priority and product 3 the lowest.

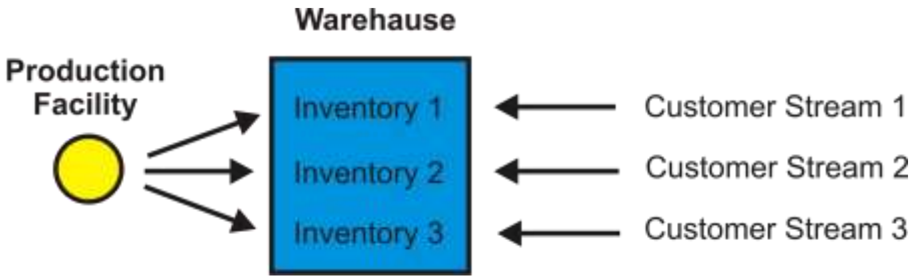


Figure 6.15 A production/inventory model with three products.

The raw-material storage (Production Facility) feeds the production process, and finished units are stored in the warehouse. Customers arrive at the warehouse with product demands, and if a demand cannot be fully satisfied by inventory on hand, the unsatisfied portion represents lost sales. Each product has its own parameters as per *Table 6.2*. The following assumptions are made:

There is always sufficient raw material in storage, so the production process never starves.

Processing of each product type is carried out in lots of five units, and finished lots are placed in the warehouse. Lot processing time is deterministic as per *Table 6.2*.

The production process experiences random failures, which may occur only while the production facility is busy. Times between failures are exponentially distributed with a mean of 200 minutes, while repair times are normally distributed, with a mean of 70 minutes and a standard deviation of 30 minutes (recall that if a negative repair time is sampled, another sample is generated until a non-negative value is obtained).

Table 6.2

Parameters of the production/inventory model with three product types

Product Type	Target Level	Reorder Point	Initial Inventory	Processing Time (hours)	Demand Interarrival Time (hours)	Demand Quantity
1	100	50	75	1	Exp(16)	UNIF(4,10)
2	200	100	150	0.6	Exp(8)	UNIF(10,15)
3	300	150	200	0.3	Exp(4)	UNIF(20,30)

The warehouse implements a separate (*s*, *S*) inventory control policy for each product type. Note that this is actually a convenient policy when a resource needs to be shared among multiple types of products. For instance, when the production process becomes blocked, it may be assigned to another product. In our case, the production facility may switch back and forth between the two highest priority products a few times before it turns to product type 3. It should be pointed out that other production-switching policies may also be employed, such as switching after the current product unit is produced, rather than the current production run.

Assume that we want to simulate the system for 100,000 hours, and estimate the following statistics:

Production-facility utilization, by product
Production-facility downtime probability
Average inventory level, by product
Percentage of customers whose demand is not completely satisfied, by product
Average lost demand quantity given that it is not completely satisfied, by product

6.2.2 Arena model

Since the current model is an extension of the earlier single-product version, its structure is similar to that of *Figure 6.2*. Accordingly, it has two main segments: inventory management and demand management. The inventory management segment is modified to model multiple inventory types with shared storage. It again circulates a product entity to produce batches of each product type, monitor inventories, and stop production when necessary. The demand management segment is modified to generate multiple types of customers and their demands. It also models the drawing down on inventories, and initiates production orders for each product type as necessary. In the next section, we examine the model logic of each segment in some detail.

6.2.3 Replenishment management segment

Figure 6.16 depicts the modified inventory management segment that models the production facility. The **Create** module, called *Raw Material*, operates precisely as in the previous example: It creates a single product entity that controls the Production operation (see Section 6.1.3) of all product types.

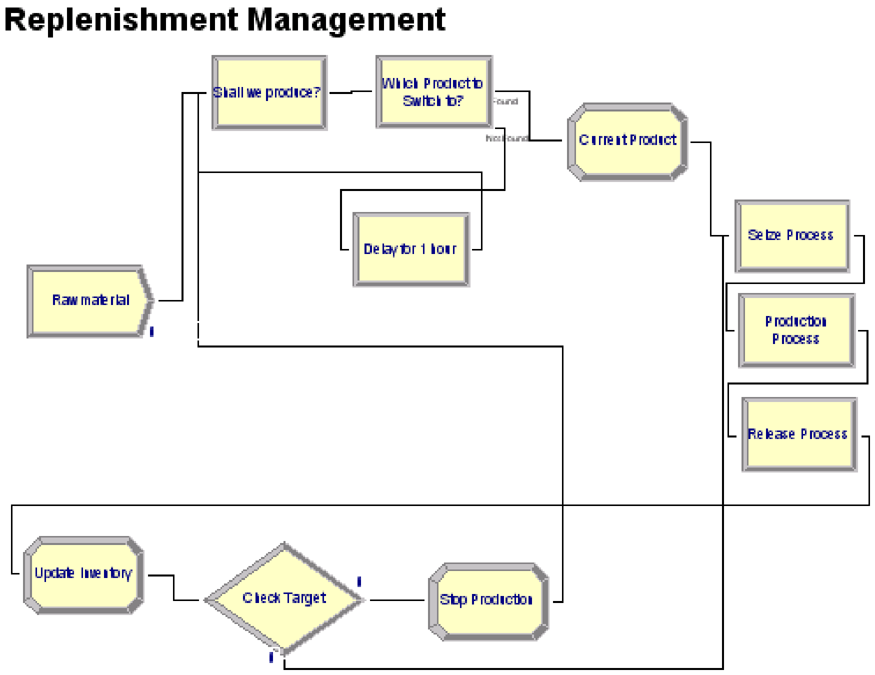


Figure 6.16 Arena model of the replenishment management segment with three product types.

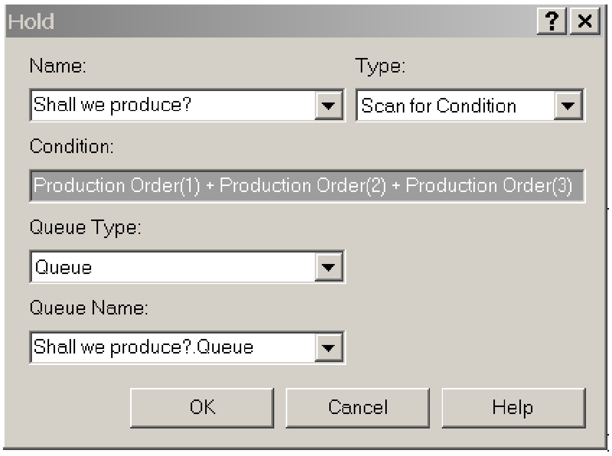
The circulating product entity is detained in the **Hold** module, called *Shall We Produce?*, until the following condition becomes true:

$$\textit{Production Order}\ (1) + \textit{Production Order}\ (2) + \textit{Production Order}\ (3) > 0,$$

as shown in the Condition field of the dialog box in *Figure 6.17*. Here, the quantities *Production Order*(*k*), *k* = 1, 2, 3, are vector elements that assume each a value of 1 or 0 representing the status of pending orders by product type. More specifically, *Production Order* (*k*) = 1 indicates that a production order is present for product k, while *Production Order*(*k*) = 0 indicates absence thereof. When the logical expression in the Condition field is true, this indicates that at least one product order is present.

When multiple product orders are present, it is necessary to decide which product type to produce next. To this end, the circulating product entity proceeds to the **Search** module, called

Which Product to Switch to?, whose dialog box is displayed in *Figure 6.18*. An incoming entity into this module triggers a search. Using an index variable that ranges from the Starting Value field contents to the Ending Value field contents, the **Search** module can perform various searches over the specified range, guided by a logical condition specified in the Search Condition field.


Hold

Name: Shall we produce?

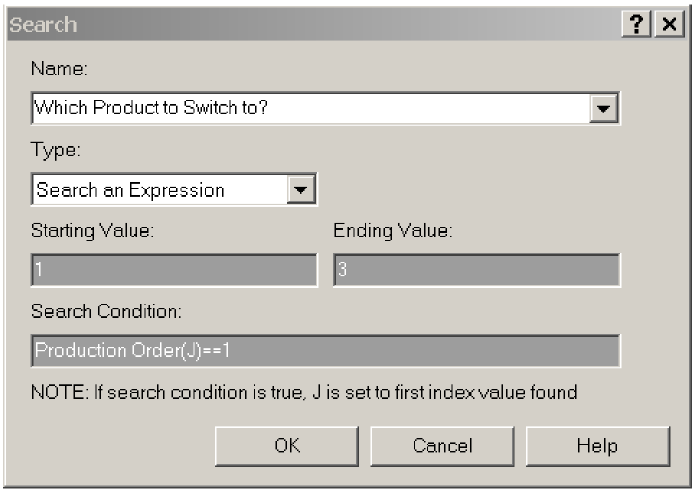
Type: Scan for Condition

Condition: Production Order(1) + Production Order(2) + Production Order(3)

Queue Type: Queue

Queue Name: Shall we produce?.Queue

Figure 6.17 Dialog box of the **Hold** module *Shall We Produce?*.


Search

Name: Which Product to Switch to?

Type: Search an Expression

Starting Value: 1

Ending Value: 3

Search Condition: Production Order(J)==1

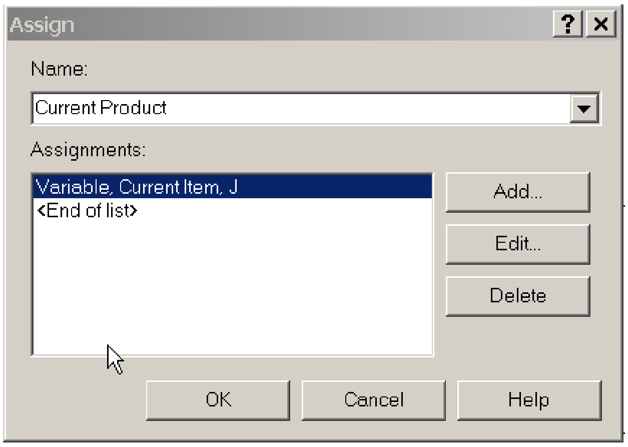
NOTE: If search condition is true, J is set to first index value found

Figure 6.18 Dialog box of the **Search** module *Which Product to Switch to?*

Various types of search can be selected in the Type field. For example, the modeler can search a range of expressions for the first one with a prescribed value; it can also search an entity queue or entity batch for the first entity with a prescribed attribute value, or one that satisfies a prescribed condition. The search result is returned in the Arena variable *J*, as shown in the Search Condition field; a positive *J* value returns the result of a successful search, while a returned value of *J* = 0 signals a failed search.

In our case, the search is for the first product k in the range {1, 2, 3}, which satisfies the condition *Production Order*(*k*)==1. Recall that in our model, product types have production priorities with lower-numbered products having higher priority. Consequently, the search for the highest-priority pending order starts with product type 1 and proceeds in ascending product type number.

Following search completion, the circulating product entity enters the **Assign** module, called *Current Product*, whose dialog box is displayed in *Figure 6.19*. Here, the circulating product entity notes which product type to switch production to. This is accomplished by setting the variable *Current Item* to variable *J*, which now contains the product type index returned from the just completed search. Note that it is appropriate here for *Current Item* to be a variable, since only one entity circulates in the inventory management segment. If multiple entities were to roam this segment, then Current Item would have to be an attribute in order to maintain the integrity of its value.

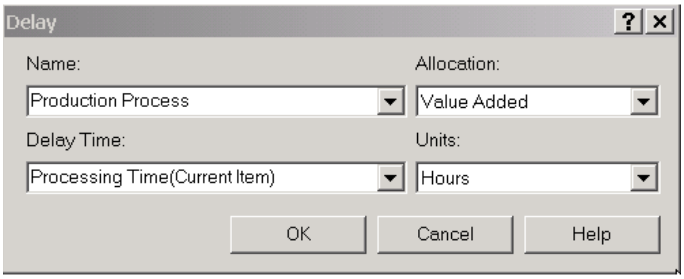


Assign

Name: Current Product

Assignments:
Variable, Current Item, J
<End of list>

Figure 6.19 Dialog box of the **Assign** module *Current Product*.



Delay

Name: Production Process

Allocation: Value Added

Delay Time: Processing Time(Current Item)

Units: Hours

Figure 6.20 Dialog box of the **Delay** module *Production Process*.

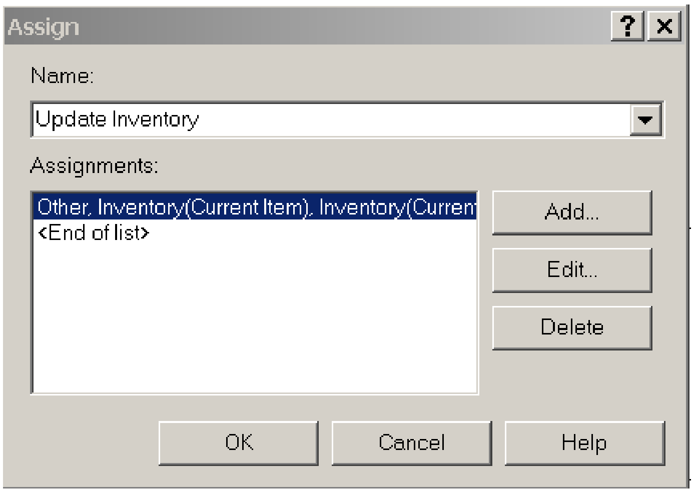
Next, the circulating product entity proceeds to traverse a standard Seize-Delay-Release sequence of three modules (*Seize Process*, *Production Process*, and *Release Process*, respectively), which model the production of the current product type, as indicated by the *Current Item* variable. While the **Seize** and **Release** modules are identical to their counterparts in Section 6.1, the **Delay** module implements here a product-dependent processing time (using the values from *Table 6.1*), as shown in *Figure 6.20*. The requisite processing times are stored in the vector Processing Time, indexed by product types.

When processing of the current batch is done, that batch should be placed in the warehouse. To this end, the circulating product entity proceeds to the **Assign** module, called Update Inventory, whose dialog box is shown in *Figure 6.21*. The inventory level of each product is modeled by a vector, called Inventory, which is indexed by product type. In a similar vein, the batch size of each product type is stored in the vector Batch Size of the same dimension. To update the inventory level of the current product type, the corresponding inventory level is simply incremented by the just-produced batch size (see the assignment in the first line of the Assignments field).

Clicking the Edit button in *Figure 6.21* pops up the Assignments dialog box for that line, displayed in *Figure 6.22*. The corresponding Assignments expression is:

Inventory (*Current Item*) = *Inventory*(*Current Item*) + *Batch Size* (*Current Item*)

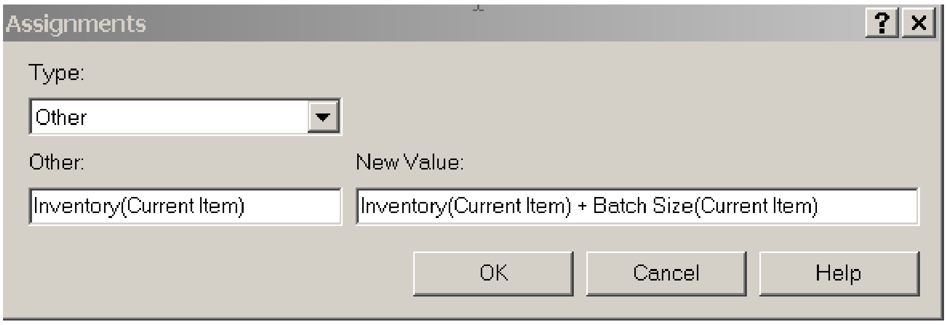
where the Other option in the Type field indicates an assignment to a vector element. Another style of assignment to vectors will be introduced in Section 6.2.4.



Name:
Update Inventory

Assignments:
Other, Inventory(Current Item), Inventory(Curren
<End of list>

Figure 6.21 Dialog box of the Assign module Update Inventory.

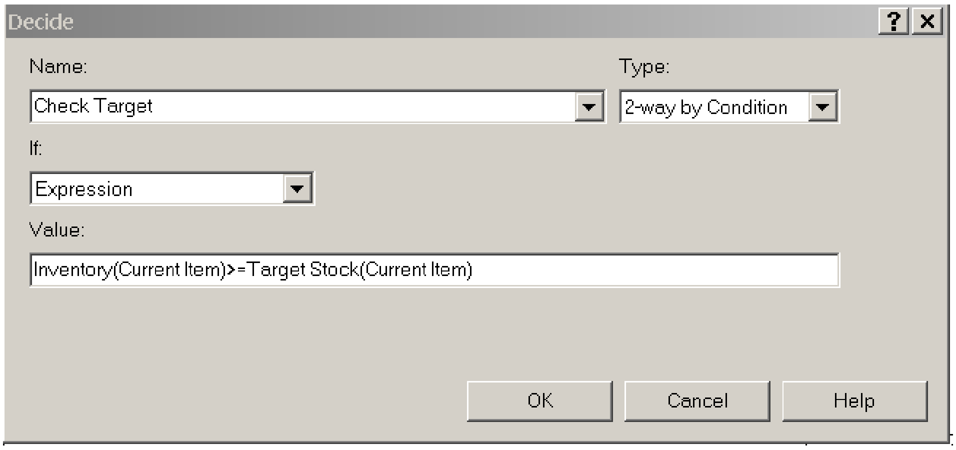


Type: Other

Other: Inventory(Current Item)

New Value: Inventory(Current Item) + Batch Size(Current Item)

Figure 6.22 Dialog box of the Assignments field from the Assign module Update Inventory.



Name: Check Target

Type: 2-way by Condition

If: Expression

Value: Inventory(Current Item)>=Target Stock(Current Item)

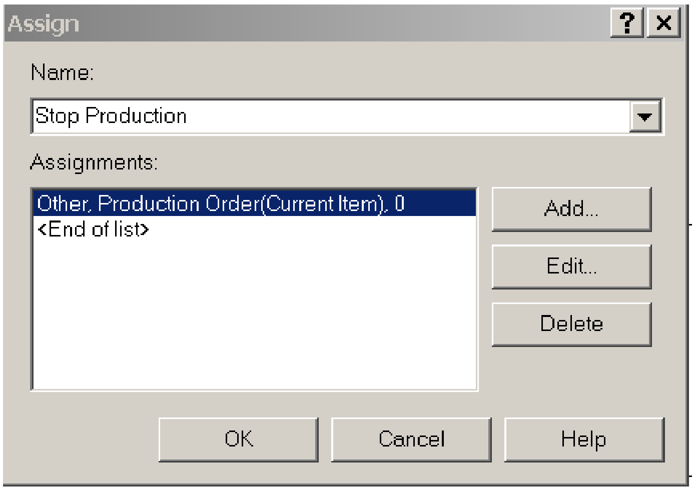
Figure 6.23 Dialog box of the **Decide** module *Check Target*.

At this point, the circulating product entity needs to check whether the target value of the current product type has been reached, so it proceeds to the **Decide** module, called *Check Target*, whose dialog box is displayed in *Figure 6.23*. Note that inventory target values for product types 1, 2, and 3 are stored in the vector elements *Target Stock*(k), k = 1, 2, 3, respectively, as evidenced by the logical expression in the Value field. The circulating product entity checks if the target level of the current product type has been reached. If it has, the product entity would take the exit branch from the *Check Target* module to the **Assign** module, called *Stop Production*, whose dialog box is shown in *Figure 6.24*. Here, the circulating product entity stops the production of the current production type by executing the assignment

$$\textit{Production Order}(\textit{current item}) = 0.$$

It then loops back to the **Hold** module *Shall We Produce?* (see *Figure 6.16*) to identify the next product type (if any) to which production is to be switched.

If, however, the target level of the current product type has not been reached, the circulating product entity would take the exit branch from the *Check Target* module to the **Seize** module *Seize Process* (see *Figure 6.16*) to process the next unit of the current product type.

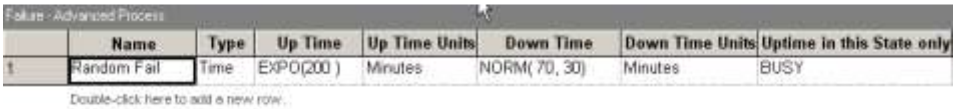


Assign

Name: Stop Production

Assignments:
Other, Production Order(Current Item), 0
<End of list>

Figure 6.24 Dialog box of the **Assign** module *Stop Production*.



Failure - Advanced Process

	Name	Type	Up Time	Up Time Units	Down Time	Down Time Units	Uptime in this State only
1	Random Fail	Time	EXPO(200)	Minutes	NORM(70, 30)	Minutes	BUSY

Double-click here to add a new row.

Figure 6.25 Dialog spreadsheet of the **Failure** module.

Finally, recall that the production facility experiences failures only in the busy state. This constraint is specified in the **Failure** module dialog spreadsheet, as shown in *Figure 6.25*. Here, the Uptime in this State Only field is set to BUSY to indicate that the production facility does not "age," unless it is busy producing products. In contrast, leaving this field blank would allow failures to occur any time, even when the production facility is idle, since "aging" takes place continually.

6.2.4 Demand management segment

Figure 6.26 depicts the modified demand management segment, which models demand arrivals at the inventory facility. In this model, the arrival processes of the three product types are modeled by a tier of three **Create** modules on the left side.

Figure 6.27 depicts the dialog box of the **Create** module, which generates customer entities that carry demand for the first product type.

In a similar vein, the demand processes of the three product types are modeled by a tier of three **Assign** modules, where the demand-product type and quantity are assigned to an incoming customer entity. *Figure 6.28* depicts the dialog box of the **Assign** module, which generates customer demand quantities for the first product type. In this module, a product type is assigned to the customer entity's attribute *Type*, and then a demand quantity is sampled and assigned to the customer entity's attribute *Demand*.

In addition, the model keeps track of the total number of arrivals of each customer type in a vector of counters, called *Total Customers*, by incrementing the corresponding counter. The vector *Total Customers* in the third assignment was declared and assigned in the corresponding **Assignments** modules, whose dialog box is shown in *Figure 6.29*. Here, the Type field was set to the *Variable Array* (1D) option, although the modeler can alternatively use the *Other* option, as illustrated in *Figure 6.22*.

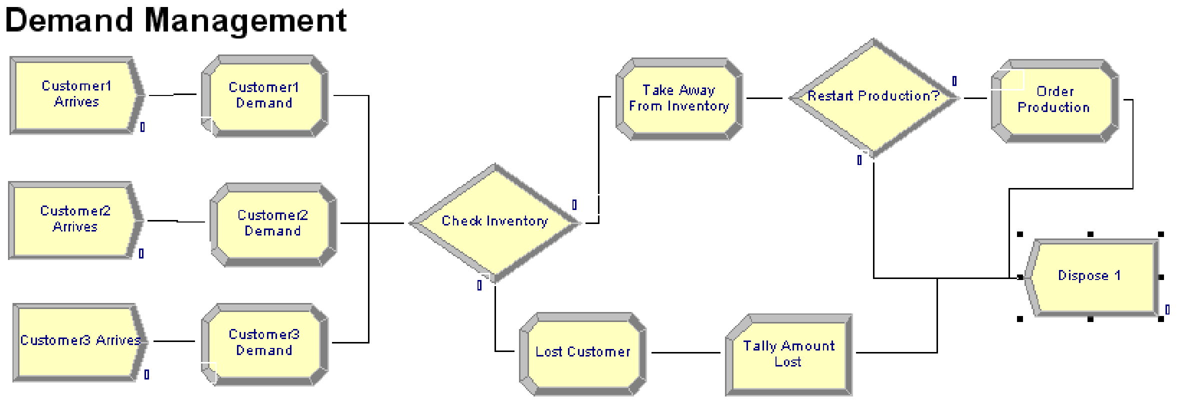
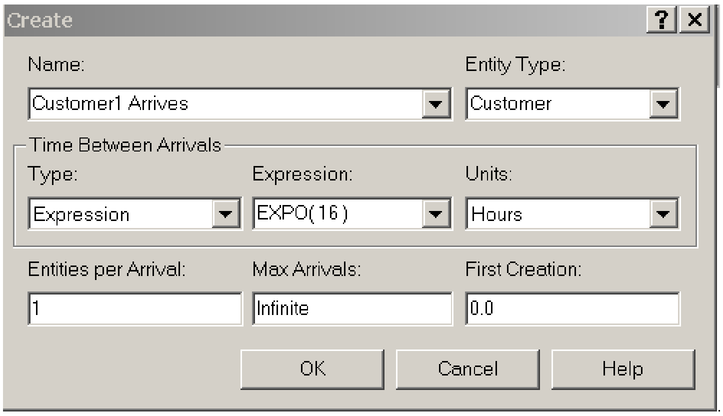


Figure 6.26 Arena model of the demand management segment for the production/inventory system with three product types.



Name: Customer1 Arrives

Entity Type: Customer

Time Between Arrivals

Type: Expression

Expression: EXPO(16)

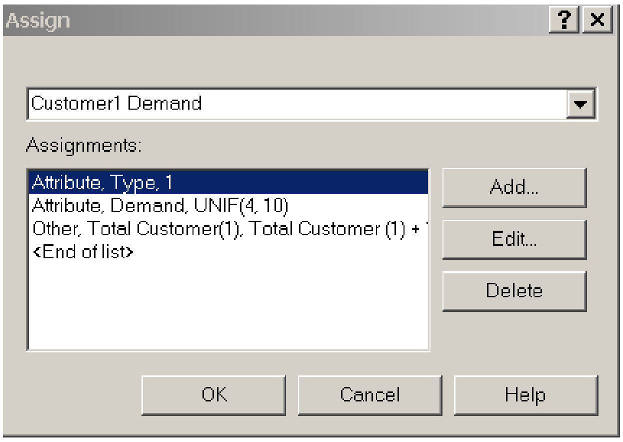
Units: Hours

Entities per Arrival: 1

Max Arrivals: Infinite

First Creation: 0.0

Figure 6.27 Dialog box of the **Create** module that generates type 1 customer entities.



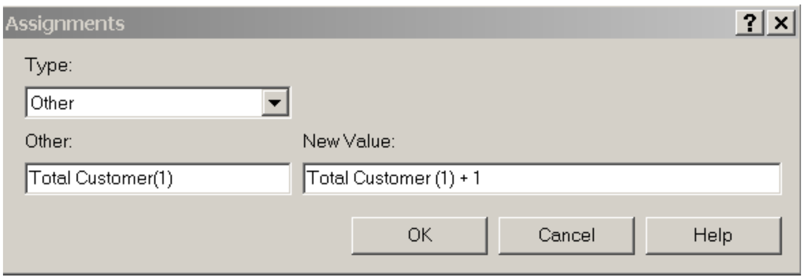
Customer1 Demand

Assignments:

Attribute, Type, 1
Attribute, Demand, UNIF(4, 10)
Other, Total Customer(1), Total Customer (1) +
<End of list>

Figure 6.28 Dialog box of the **Assign** module that generates demand quantities for type 1 customer entities.

Next, the customer entity needs to check whether there is sufficient inventory in the warehouse to satisfy the requested quantity of the requisite product type. To this end, all customer entity proceeds to the **Decide** module, called *Check Inventory*, whose dialog box is depicted in *Figure 6.30*.

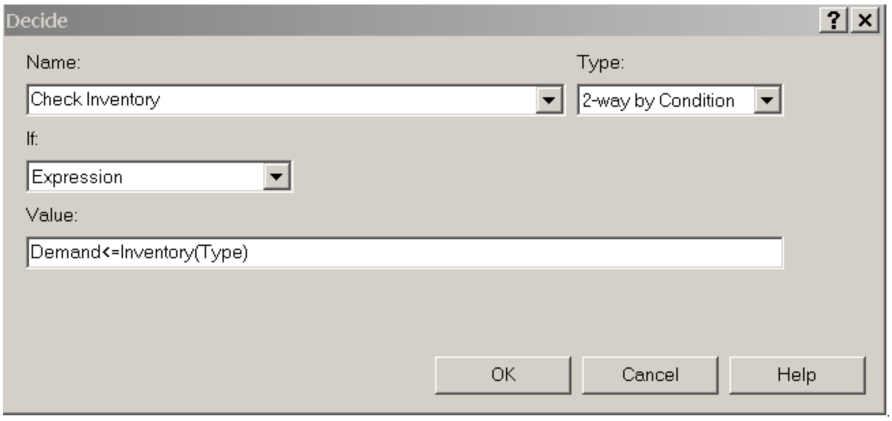


Type: Other

Other: Total Customer(1)

New Value: Total Customer (1) + 1

Figure 6.29 Dialog box of Assignments for declaring and assigning the Total Customers vector.



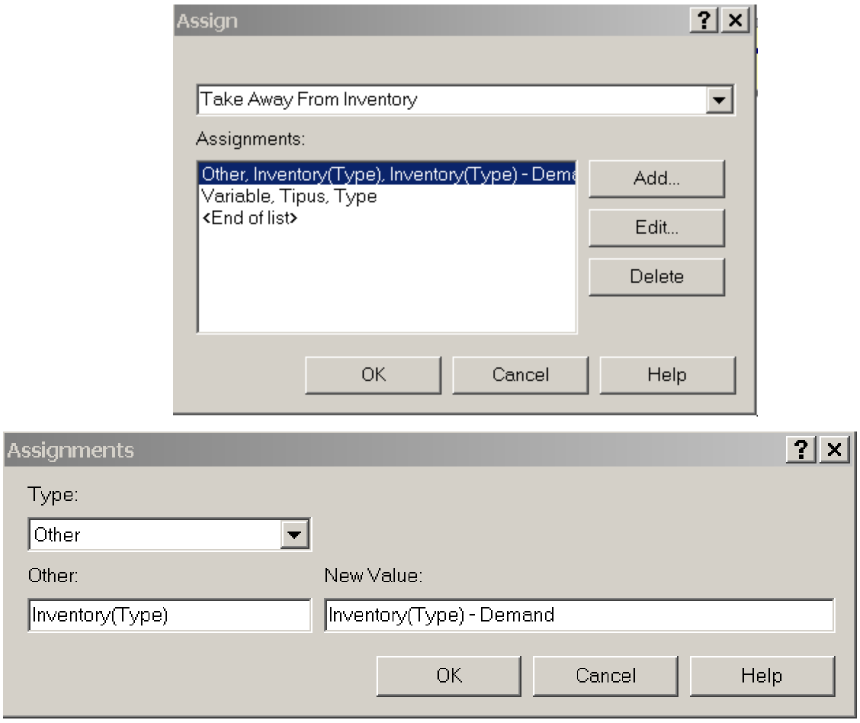
Name: Check Inventory

Type: 2-way by Condition

If: Expression

Value: Demand<=Inventory(Type)

Figure 6.30 Dialog box of the **Decide** module *Check Inventory*.



Take Away From Inventory

Assignments:

Other, Inventory(Type), Inventory(Type) - Dem

Variable, Tipus, Type

<End of list>

Type: Other

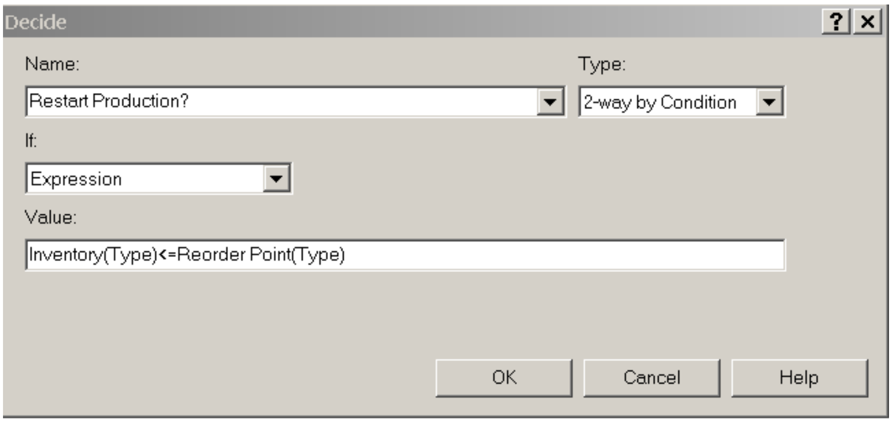
Other: Inventory(Type)

New Value: Inventory(Type) - Demand

Figure 6.31 Dialog spreadsheets of the **Assign** module *Take Away From Inventory* (bottom) and its associated Assignments dialog box (top).

The availability of sufficient product is expressed by the logical expression in the Value field, keeping in mind that the product type information is stored in the *Type* attribute of the customer entity. If there is sufficient inventory in stock, then the demand of the current customer entity can be satisfied. In this case, the customer entity takes the exit branch to the **Assign** module, called *Decrease Inventory*, where the level of the corresponding inventory type is decremented by the demand quantity. *Figure 6.31* displays the dialog box of this module (bottom) and its associated Assignments dialog box (top), which pops up when the Edit button is clicked.

The customer entity then enters the **Decide** module, called *Restart Production?*, to check whether the reorder level is subsequently down-crossed as shown in the logical expression in the dialog box of *Figure 6.32*. Note that the logical expression in the Value field compares two vector elements: the inventory level of the currently requested product type and the reorder level of the same product type. Two cases are then possible: (1) there was sufficient inventory on hand to satisfy the incoming demand, or (2) there was insufficient inventory on hand. We next describe the scenarios that result from each case.


Decide

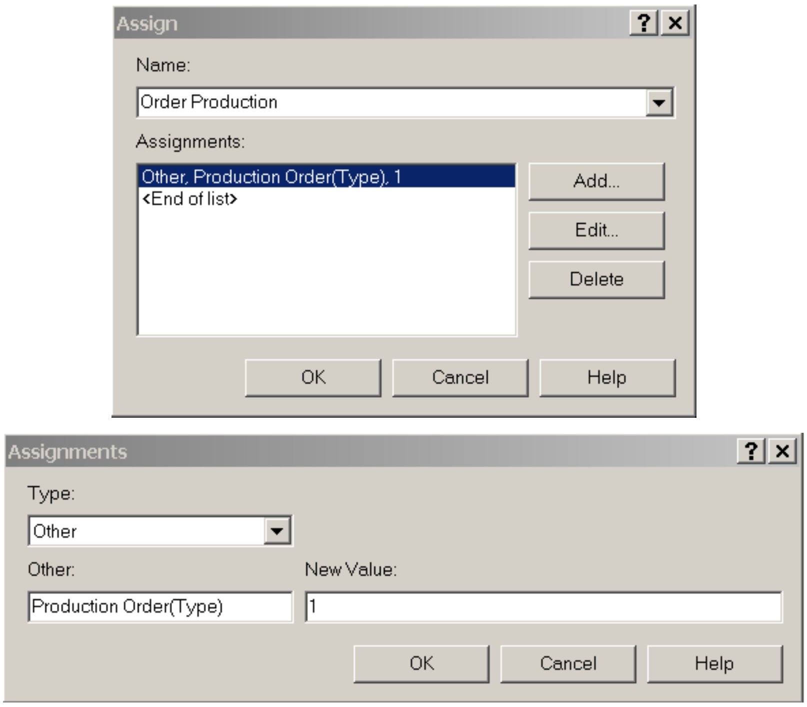
Name: Restart Production?

Type: 2-way by Condition

If: Expression

Value: Inventory(Type)<=Reorder Point(Type)

Figure 6.32 Dialog box of the **Decide** module *Restart Production?*


Assign

Name: Order Production

Assignments:
Other, Production Order(Type), 1
<End of list>

Assignments

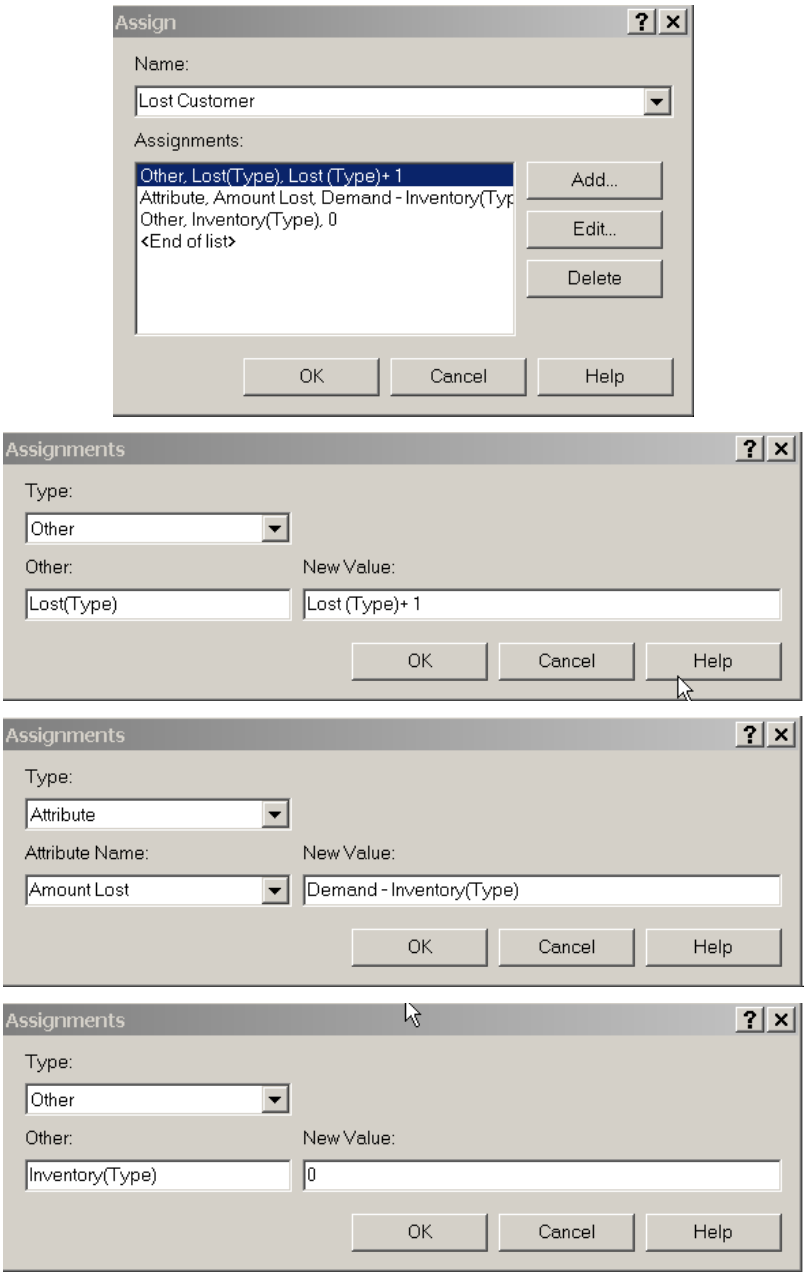
Type: Other

Other: Production Order(Type)

New Value: 1

Figure 6.33 Dialog box of the **Assign** module *Order Production* (bottom) and its associated Assignments dialog box (top).

Consider first the case of sufficient on-hand inventory. In this case, the customer entity proceeds to the **Assign** module, called *Order Production*, to initiate a production order, as shown in its dialog box (bottom) and associated **Assignments** modules (top) in *Figure 6.33*. Note that production is initiated by simply assigning a value of 1 to the variable in vector element *Production Order*(*Type*), which indicates shortage of that product type. Consequently, this shortage will be attended to in due time by the production facility that attends to production orders according to their priority structure. Note further that these variables may be set to 1 multiple times by customer entities entering this module during periods of shortage. While these assignments may be redundant, they are harmless in that the correctness of the models state is maintained; moreover, the models Arena logic is greatly simplified. Finally, having finished all its tasks in the model, the customer entity enters the dispose module, called *Dispose*, to be removed from the model.



Name: Lost Customer

Assignments:
Other, Lost(Type), Lost (Type)+ 1
Attribute, Amount Lost, Demand - Inventory(Typ
Other, Inventory(Type), 0
<End of list>

Type: Other
Other: Lost(Type)
New Value: Lost (Type)+ 1

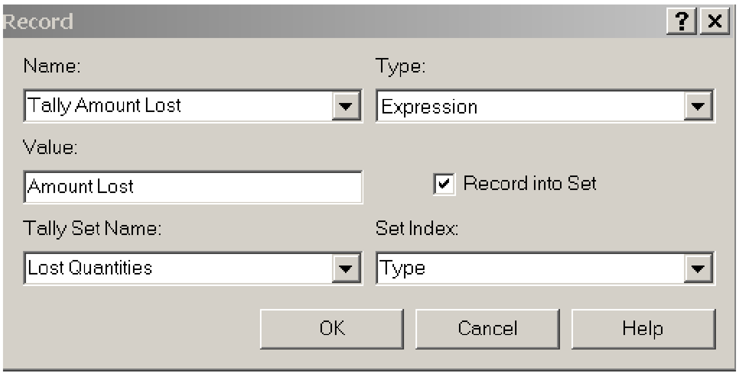
Type: Attribute
Attribute Name: Amount Lost
New Value: Demand - Inventory(Type)

Type: Other
Other: Inventory(Type)
New Value: 0

Figure 6.34 Dialog boxes of the **Assign** module *Lost Customer* (bottom) and its two associated **Assignments** modules (middle and top).

Consider next the case of insufficient on-hand inventory, which results here in the customer demand being lost. The customer entity then takes the other exit branch from the **Decide** module *Check Inventory* to the **Assign** module *Lost Customer*, whose dialog box (bottom) and two associated **Assignments** modules (middle and top) are depicted in *Figure 6.34*. Three assignments are executed in the Assignments field of the **Assign** module. The first assignment (middle dialog box) increments by 1 the appropriate element of the vector Lost, in order to track the number of lost customers by the type of product requested. The second assignment (top dialog box) records the quantity lost in the *Amount Lost* attribute of the current customer entity. Finally, the third assignment sets the inventory level of this product to 0. Note carefully that the order of assignments is important.

Next, the customer entity proceeds to the **Record** module, called *Tally Amount Lost*, to tally the current lost amount in the set Lost Quantities, as shown in the dialog box of *Figure 6.35*. Finally, the customer entity enters the **Dispose** module, called *Dispose*, to be removed from the model.



Record

Name: Tally Amount Lost

Type: Expression

Value: Amount Lost

☑ Record into Set

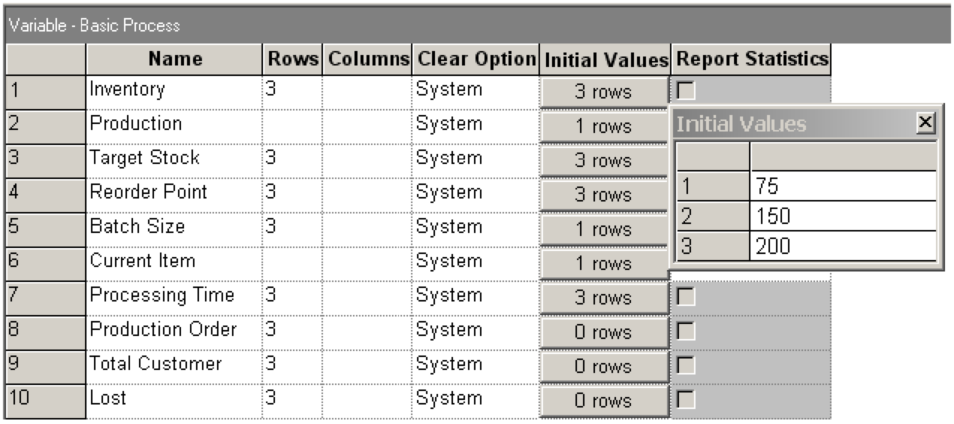
Tally Set Name: Lost Quantities

Set Index: Type

Figure 6.35 Dialog box of the **Record** module *Tally Amount Lost*.

6.2.5 Model input parameters and statistics

This section details the input parameters of the model and its statistics. Recall that all input parameters, including vector-valued ones, are declared in the **Variable** module from the **Basic Process template**.



Variable - Basic Process

	Name	Rows	Columns	Clear Option	Initial Values	Report Statistics
1	Inventory	3		System	3 rows	□
2	Production			System	1 rows	
3	Target Stock	3		System	3 rows	
4	Reorder Point	3		System	3 rows	
5	Batch Size	3		System	1 rows	
6	Current Item			System	1 rows	
7	Processing Time	3		System	3 rows	□
8	Production Order	3		System	0 rows	□
9	Total Customer	3		System	0 rows	□
10	Lost	3		System	0 rows	□

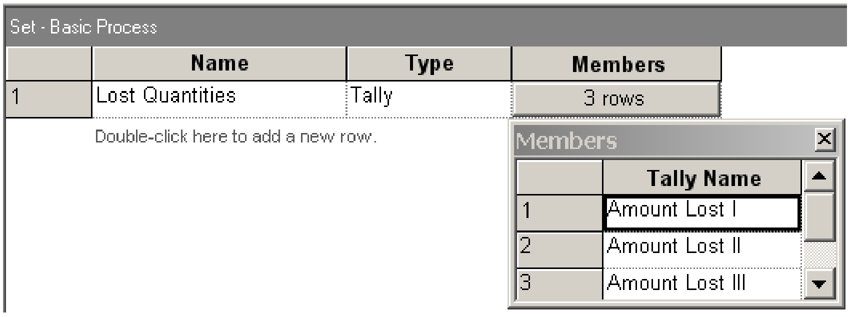
Initial Values

1	75
2	150
3	200

Figure 6.36 Dialog spreadsheet of the **Variable** module (bottom) and the Initial Values dialog spreadsheet of variable *Inventory* (top).

Figure 6.36 displays the *Variable dialog spreadsheet* (bottom), as well as the initial values of the *Inventory* variable (top). The dimension of a variable is indicated by the number of rows indicated in the button labels under the *Initial Values* column in *Figure 6.36*. Accordingly, the label 0 rows specifies no initial values, the label 1 rows specifies a single initial value for an ordinary (one-dimensional) variable (or a vector of identical values), whereas a label of the form n rows (for *n*> 1) specifies a vector-valued variable with the indicated dimension. The *Clear* Option column specifies the simulation time (if any) to reset to the initial value(s) specified. Typically, the System option is selected to activate initialization only at the beginning of each replication. Statistics collection of one-dimensional variables is enabled by checking the corresponding boxes under the *Report Statistics* column. However, statistics of vector-valued variables can be collected through the **Statistic** module spreadsheet, and will be illustrated next.

Recall that we used the *Set* element to track lost demand by product type (1, 2, or 3) in the **Record** module called *Tally Amount Lost*. To this end, we declare a three-dimensional Set, called *Lost Quantities*, in the **Set** module spreadsheet (from the **Basic Process template panel**), as shown in *Figure 6.37*.



Set - Basic Process

	Name	Type	Members
1	Lost Quantities	Tally	3 rows

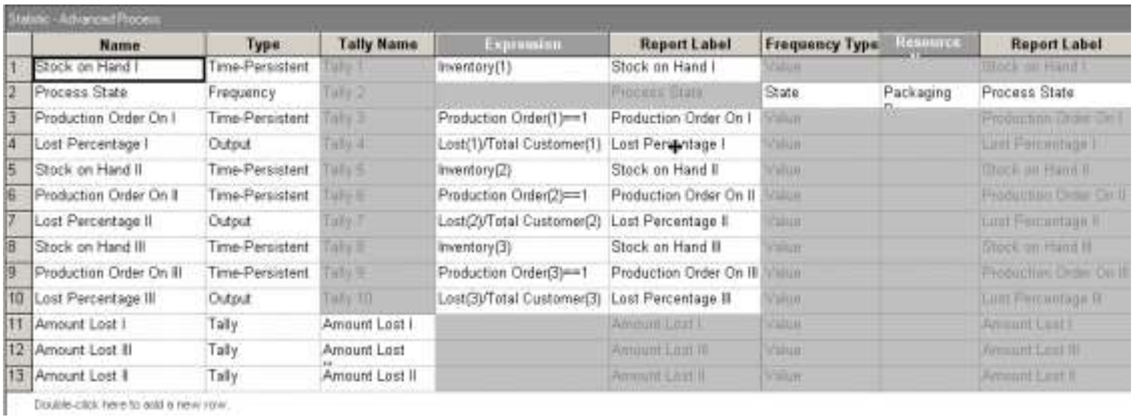
Double-click here to add a new row.

Members

	Tally Name
1	Amount Lost I
2	Amount Lost II
3	Amount Lost III

Figure 6.37 Dialog spreadsheet of the **Set** module (bottom) and the Members dialog spreadsheet for Tally set Amount Lost (top).

Finally, the specification of statistics collection can also be made in the **Statistic** module spreadsheet, as illustrated in *Figure 6.38*. Note that statistics collected on vector-valued variables are specified separately for each vector element. The *Stock on Hand and Production Order* statistics (by product type) are time averages, and as such of type *Time-Persistent*, while *Process States* statistics are of type Frequency. The *Lost Percentage* statistics are obtained after the simulation stops, and as such are of type *Output*. Finally, the *Amounts Lost* statistics are customer averages, and as such of type *Tally*.



Statistic - Advanced Process

	Name	Type	Tally Name	Expression	Report Label	Frequency Type	Resource	Report Label
1	Stock on Hand I	Time-Persistent	Tally 1	Inventory(1)	Stock on Hand I	Value		Stock on Hand I
2	Process State	Frequency	Tally 2		Process State	State	Packaging P...	Process State
3	Production Order On I	Time-Persistent	Tally 3	Production Order(1)==1	Production Order On I	Value		Production Order On I
4	Lost Percentage I	Output	Tally 4	Lost(1)/Total Customer(1)	Lost Percentage I	Value		Lost Percentage I
5	Stock on Hand II	Time-Persistent	Tally 5	Inventory(2)	Stock on Hand II	Value		Stock on Hand II
6	Production Order On II	Time-Persistent	Tally 6	Production Order(2)==1	Production Order On II	Value		Production Order On II
7	Lost Percentage II	Output	Tally 7	Lost(2)/Total Customer(2)	Lost Percentage II	Value		Lost Percentage II
8	Stock on Hand III	Time-Persistent	Tally 8	Inventory(3)	Stock on Hand III	Value		Stock on Hand III
9	Production Order On III	Time-Persistent	Tally 9	Production Order(3)==1	Production Order On III	Value		Production Order On III
10	Lost Percentage III	Output	Tally 10	Lost(3)/Total Customer(3)	Lost Percentage III	Value		Lost Percentage III
11	Amount Lost I	Tally	Amount Lost I		Amount Lost I	Value		Amount Lost I
12	Amount Lost III	Tally	Amount Lost ...		Amount Lost III	Value		Amount Lost III
13	Amount Lost II	Tally	Amount Lost II		Amount Lost II	Value		Amount Lost II

Double-click here to add a new row.

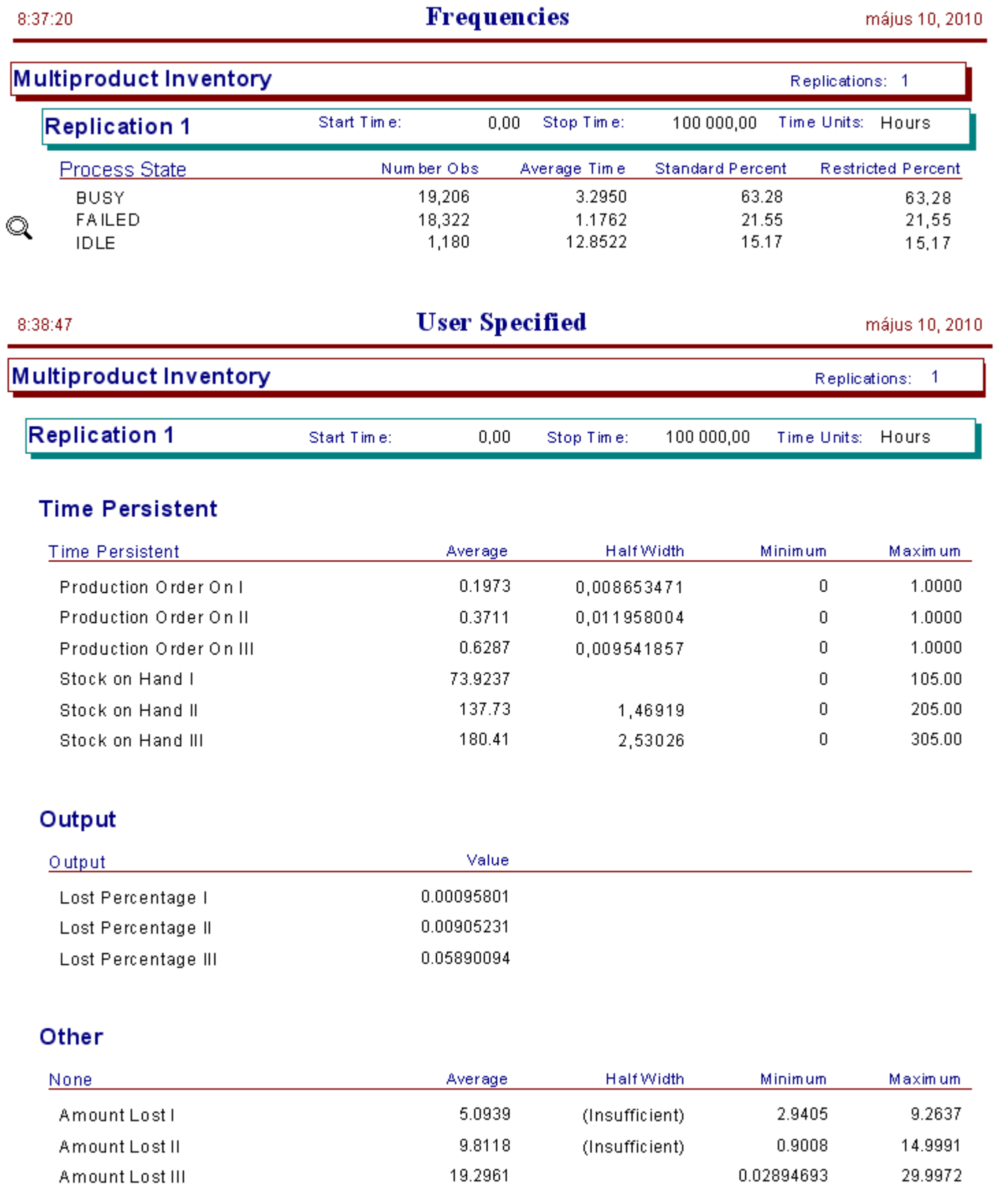
Figure 6.38 Dialog spreadsheet of the **Statistic** module for specifying statistics collection.

6.2.6 Simulation results

Figure 6.39 displays the results of a simulation run of length 100,000 hours. An inspection of the Frequencies report shows that the production facility is busy about 64% of the time, idle about 14% of the time, and down about 22% of the time.

Note that if the system, were to "age" continually (so that failures could occur at any state), then the downtime probability would increase and the idle-time probability would decrease. However, the busy-time probability would not change (within simulation noise), since the work load does not change.

The fraction of time that a product type was pending varies in accordance to the priority structure of product types: about 20% for type 1, 38% for type 2, and 63% for type 3. As expected, the lowest-priority product type received significantly less attention from the production facility.



8:37:20 **Frequencies** május 10, 2010

Multiproduct Inventory Replications: 1

Replication 1 Start Time: 0,00 Stop Time: 100 000,00 Time Units: Hours

Process State	Number Obs	Average Time	Standard Percent	Restricted Percent
BUSY	19,206	3.2950	63.28	63,28
FAILED	18,322	1.1762	21.55	21,55
IDLE	1,180	12.8522	15.17	15,17

8:38:47 **User Specified** május 10, 2010

Multiproduct Inventory Replications: 1

Replication 1 Start Time: 0,00 Stop Time: 100 000,00 Time Units: Hours

Time Persistent

Time Persistent	Average	Half Width	Minimum	Maximum
Production Order On I	0.1973	0,008653471	0	1.0000
Production Order On II	0.3711	0,011958004	0	1.0000
Production Order On III	0.6287	0,009541857	0	1.0000
Stock on Hand I	73.9237		0	105.00
Stock on Hand II	137.73	1,46919	0	205.00
Stock on Hand III	180.41	2,53026	0	305.00

Output

Output	Value
Lost Percentage I	0.00095801
Lost Percentage II	0.00905231
Lost Percentage III	0.05890094

Other

None	Average	Half Width	Minimum	Maximum
Amount Lost I	5.0939	(Insufficient)	2.9405	9.2637
Amount Lost II	9.8118	(Insufficient)	0.9008	14.9991
Amount Lost III	19.2961		0.02894693	29.9972

Figure 6.39 Simulation results for the multiproduct production/inventory system.

The statistics of stock on hand by product type reveal that the average inventory levels of all product types are below their reorder points, which indicates that the production facility is having a hard time keeping up with demand. This is largely due to failures as evidenced by the high downtime probability. Further evidence for this phenomenon is furnished by the minimum stock level statistics, all of which hit 0, indicating episodes of stock-out. Stock-outs are quantified by the percentage of loss entries in the Output section and the amount lost by product type in the *Other* section of the report.

Similarly to the previous example, system performance may be improved in a number of ways. Again, enhanced maintenance of the production facility would decrease its downtime. Finally, to avoid excessive neglect of orders with low-priority product types, the number of production switches among product types could be restricted.

6.3 Multi-stage supply chain (Model 6-3)

This section extends the production/inventory system studied in Section 6.1 by adding several stages to it. Specifically, this example studies a four-stage supply chain consisting of a supplier, manufacturing plant, distribution center, and retailer. The system uses so-called installation stock policies (replenishment policies based only on local information at the supplied installation). Note that such policies utilize only inventory position data (inventory on hand, outstanding orders, and backorders). In contrast, so-called stage stock policies require additional information in the form of inventory positions at the current installation, as well as at all downstream stages.

6.3.1 Problem statement

Consider a single-product, Multi-stage supply chain consisting of a retailer (**R**), distribution center (**DC**), manufacturing plant (**P**), and supplier (**S**). The manufacturing plant interacts with two buffers: an input buffer (**IB**) storing incoming raw material, and an output buffer (**OB**) storing outgoing finished product. The system is depicted schematically in *Figure 6.40*.

The retailer faces a customer demand stream, and to manage inventory, it uses a continuous-review (s_R, Q_R) control policy, based only on information at the retailer. Recall that under this policy, a replenishment of quantity Q_R is ordered from the distribution center whenever the inventory position (inventory on-hand plus outstanding orders minus backorders) down-crosses level s_R. If the distribution center has sufficient inventory on hand, then the order lead time consists only of transportation delay. Otherwise, it experiences additional delays due to additional transportation delays and possible stock-outs upstream of the distribution center. Any excess demand at the retailer that cannot be immediately satisfied from on-hand inventory is lost.

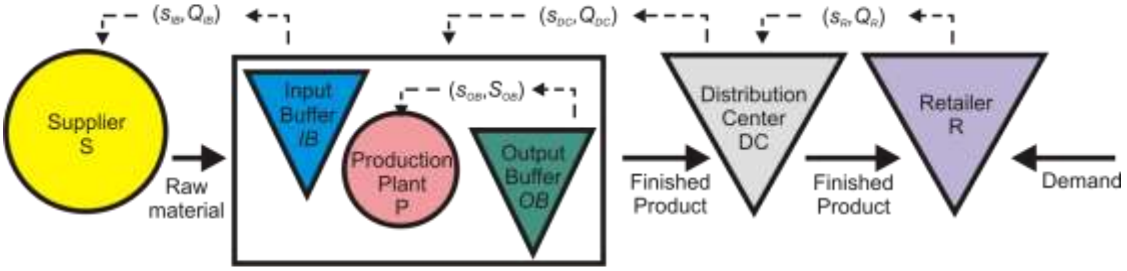


Figure 6.40 Multi-stage supply chain system.

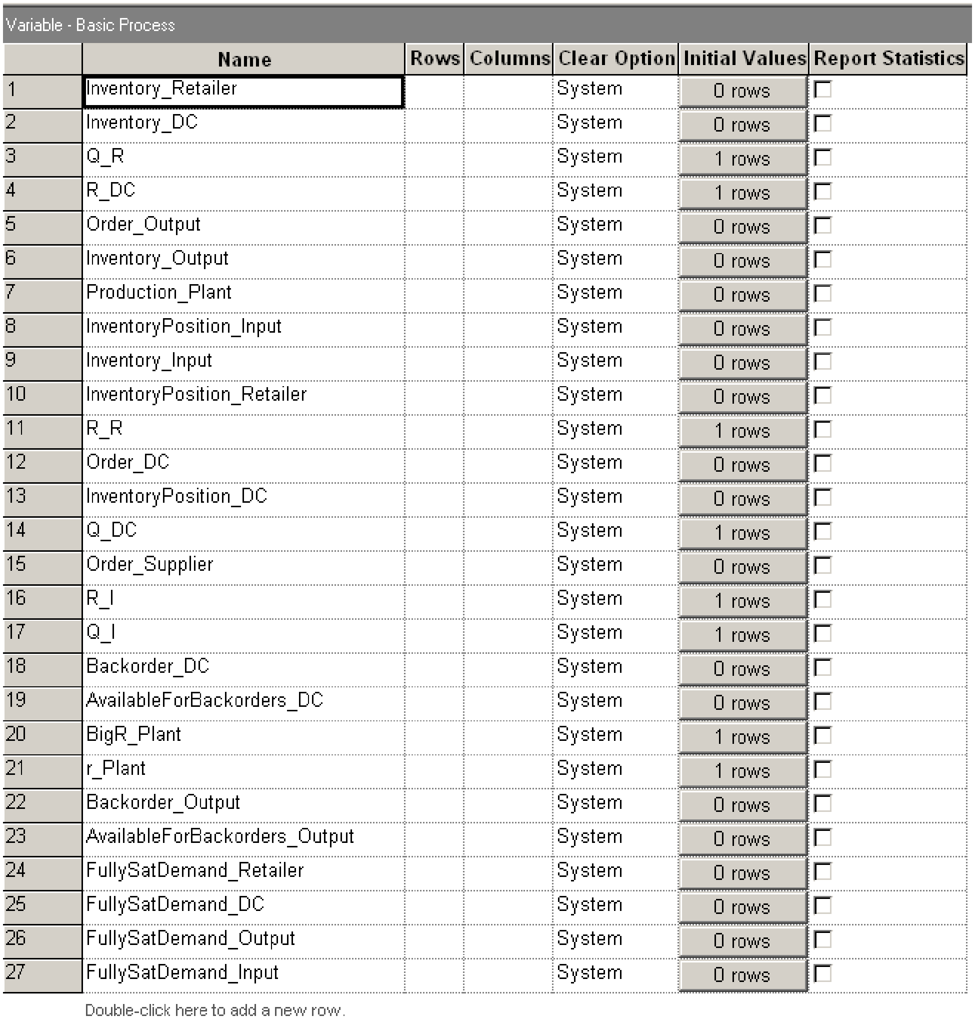
The demand stream at the distribution center consists of orders from the retailer. However, unlike the retailer, unsatisfied demand is backordered. In a similar vein, the distribution center replenishes its stocks from the output buffer (**OB**) of the upstream plant, based on a (s_{DC}, Q_{DC}) continuous-review inventory policy. The unsatisfied portions of orders placed with the plant are backordered.

The plant's manufacturing policy is a continuous-review (s_{OB}, S_{OB},) policy. The plant manufactures one product unit at a time, having consumed one unit of raw material from the input buffer. Note that shortages of raw material in the input buffer (starvation) will cause production stoppages. The input buffer, in turn, orders from an external supplier assumed to have unlimited inventories at all times, so the raw-material lead time is limited to transportation delay, and the plant's orders are always fully satisfied. The corresponding inventory control policy is a continuous-review (s_{IB}, Q_{IB}) policy. *Table 6.3* displays the values of the model's inventory control parameters.

Table6.3

Inventory control parameters

Input Buffer	Output Buffer	Distribution Center	Retailer
s_{IB} =10 Q_{IB}= 13	s_{OB} = 10 S_{OB} = 30	s_{DC} = 10 Q_{DC} = 20	s_R= = 5 Q_R = 10



Variable - Basic Process

	Name	Rows	Columns	Clear Option	Initial Values	Report Statistics
1	Inventory_Retailer			System	0 rows	□
2	Inventory_DC			System	0 rows	□
3	Q_R			System	1 rows	□
4	R_DC			System	1 rows	□
5	Order_Output			System	0 rows	□
6	Inventory_Output			System	0 rows	□
7	Production_Plant			System	0 rows	□
8	InventoryPosition_Input			System	0 rows	□
9	Inventory_Input			System	0 rows	□
10	InventoryPosition_Retailer			System	0 rows	□
11	R_R			System	1 rows	□
12	Order_DC			System	0 rows	□
13	InventoryPosition_DC			System	0 rows	□
14	Q_DC			System	1 rows	□
15	Order_Supplier			System	0 rows	□
16	R_I			System	1 rows	□
17	Q_I			System	1 rows	□
18	Backorder_DC			System	0 rows	□
19	AvailableForBackorders_DC			System	0 rows	□
20	BigR_Plant			System	1 rows	□
21	r_Plant			System	1 rows	□
22	Backorder_Output			System	0 rows	□
23	AvailableForBackorders_Output			System	0 rows	□
24	FullySatDemand_Retailer			System	0 rows	□
25	FullySatDemand_DC			System	0 rows	□
26	FullySatDemand_Output			System	0 rows	□
27	FullySatDemand_Input			System	0 rows	□

Double-click here to add a new row.

Figure 6.41 Dialog spreadsheet of the **Variable** module for the Arena model of the multi-stage supply chain system.

The system is subject to the following assumptions:

1. The retailer faces customer demand that arrives according to a Poisson process. The demand quantity of each arrival is one product unit, and all unsatisfied demands are lost.
2. For modeling generality and versatility, we assume that the manufacturing time distribution of a product unit and all transportation time distributions are of the *Erlang* type (see Section 3.8.7). Specifically, all transportation delays are drawn from the *Erl*(*k* = 2, λ= 1) distribution, while the manufacturing time distribution is *Erl*(*k*=3, λ=5).
3. At all stages, orders are received in the order they were placed (no overtaking takes place). In fact, an order shipment is launched only after the previous one is received at its destination.
4. When a stock-out occurs in the **DC** or plant and the unsatisfied portion of the order is

backordered, order fulfillment (shipment) is deferred until the full order becomes available. In brief, there is no shipping of partial orders.

We are interested in the following performance metrics:

The long-run time averages of all inventory levels in the system
The average number of backorders at the distribution center and the output buffer
Customer service levels in each stage

These performance metrics would guide the modeler in making suitable choices for parameters of the inventory control policies.

6.3.2 Arena model

The system under study is more complex by far than its counterparts in the previous two examples. Accordingly, its Arena model is composed of five segments, each associated with an inventory-holding buffer in a system stage. Each such buffer is subjected to the following events: order arrival, inventory updating, replenishment order triggering, and order shipment. Additional supply chain activities are modeled at the endpoints of the supply chain, namely, demand arrival on the downstream end, and product manufacturing on the upstream end. *Figure 6.41* displays all the variables of the Arena model.

Next, we describe each model segment in some detail, starting with the extreme downstream stage and moving upstream the supply chain.

6.3.3 Inventory management segment for retailer

Figure 6.42 depicts the retailer inventory management segment of the Arena model. This segment generates the demand stream, handles demand fulfillment, and triggers replenishment orders from the distribution center.

The Poisson stream of customer arrivals with single-unit demand quantities are generated by the **Create** module, called *Customer Demand Arrival At Retailer*. On arrival, a customer entity first enters the **Record** module, called *Tally Retailer Demand*, to tally its demand.

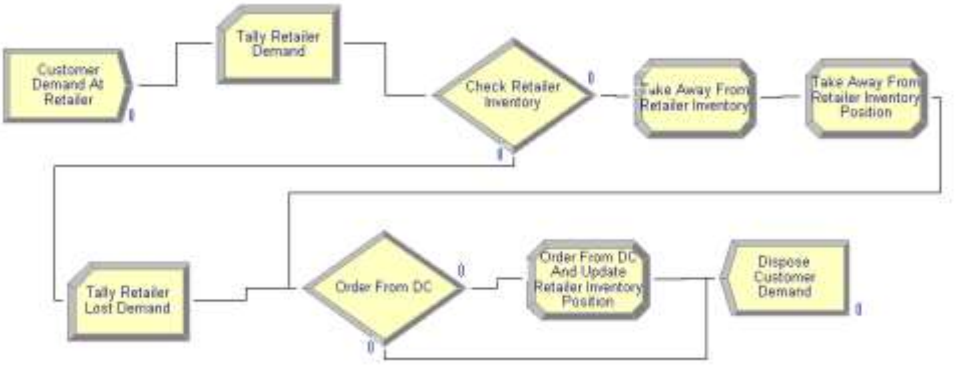
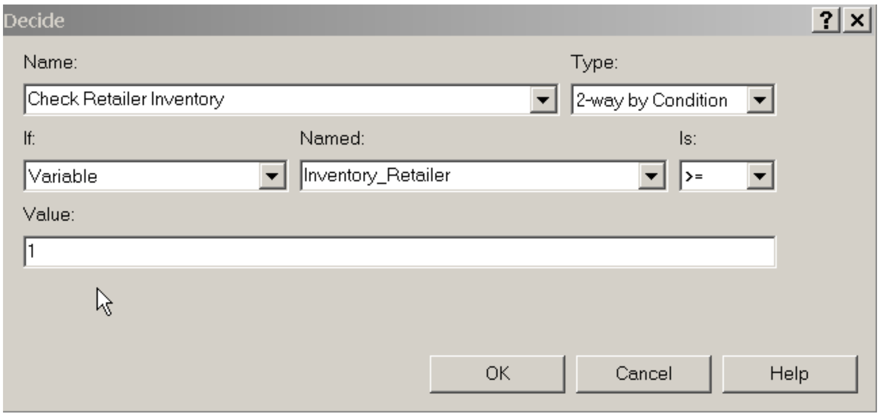

Customer Demand At Retailer
0
Tally Retailer Demand
Check Retailer Inventory
0
0
Take Away From Retailer Inventory
Take Away From Retailer Inventory Position
Tally Retailer Lost Demand
Order From DC
0
0
Order From DC And Update Retailer Inventory Position
Dispose Customer Demand
0

Figure 6.42 Arena model of the inventory management segment for the retailer.

The customer entity then proceeds to test whether the retailer has sufficient inventory on hand by entering the **Decide** module, called *Check Retailer Inventory*, whose dialog box is shown in *Figure 6.43*. The test has two possible outcomes. First, if the condition *Inventory_Retailer* ≥ 1 holds, then the customer entity takes the *True* exit to the **Assign** module, called *Take Away From Retailer Inventory*, where it decrements the on-hand inventory by 1. It then proceeds to another **Assign** module, called *Take Away From Retailer Inventory Position* to decrement by 1 the variable *InventoryPosition_Retailer*. Second, if the condition *Inventory_Retailer* = 0 holds, then the customer entity takes the *False* exit and the current demand is lost. In this case, the customer entity proceeds to the **Record** module, called *Tally Retailer Lost Sales*, to tally the lost demand.



Decide

Name: Check Retailer Inventory

Type: 2-way by Condition

If: Variable

Named: Inventory_Retailer

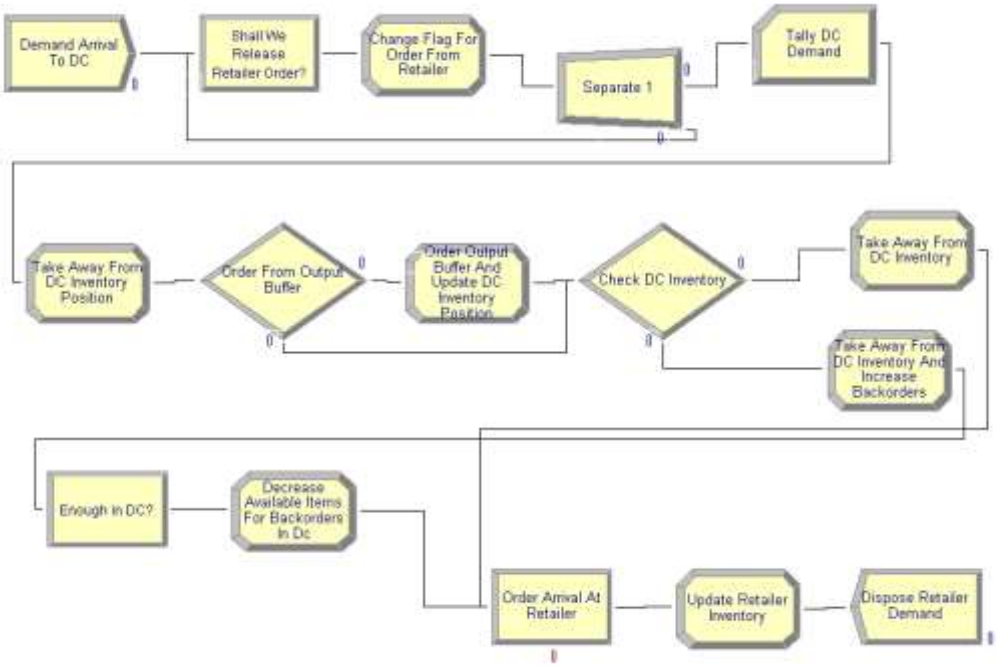
Is: >=

Value: 1

Figure 6.43 Dialog box of the **Decide** module *Check Retailer In*ventory.

Note that the two **Assign** modules make use of two variables to keep track of inventory information: *Inventory_Retailer*, which tracks the on-hand inventory level, fluctuating between 0 and $s_R + Q_R$, and *InventoryPosition_Retailer*, which tracks the inventory position, fluctuating between s_R and $s_R + Q_R$. The former is used to satisfy a customer's demand, while the latter is used when triggering a replenishment order. Recall that whenever the inventory position at the retailer down-crosses R_R, a replenishment order of quantity Q_R is placed with the distribution center, and the inventory position is immediately updated to $s_R + Q_R$.

Both paths converge at the **Decide** module, called *Order From DC*, where the customer entity checks whether the variable *InventoryPosition_Retailer* has just down-crossed s_R. If it has, then the customer entity proceeds to the **Assign** module, called *Order From DC And Update Retailer Inventory Position*, and performs two assignments. The first assignment sets *Order_DC* = 1, which would promptly release a pending order entity currently detained in the **Hold** module, called *Shall We Release Retailer Order?*, in the distribution-center inventory management segment (see *Figure 6.44*). The second assignment sets



Demand Arrival To DC
Shall We Release Retailer Order?
Change Flag For Order From Retailer
Separate 1
Tally DC Demand
Take Away From DC Inventory Position
Order From Output Buffer
Order Output Buffer And Update DC Inventory Position
Check DC Inventory
Take Away From DC Inventory
Take Away From DC Inventory And Increase Backorders
Enough In DC?
Decrease Available Items For Backorders In Dc
Order Arrival At Retailer
Update Retailer Inventory
Dispose Retailer Demand

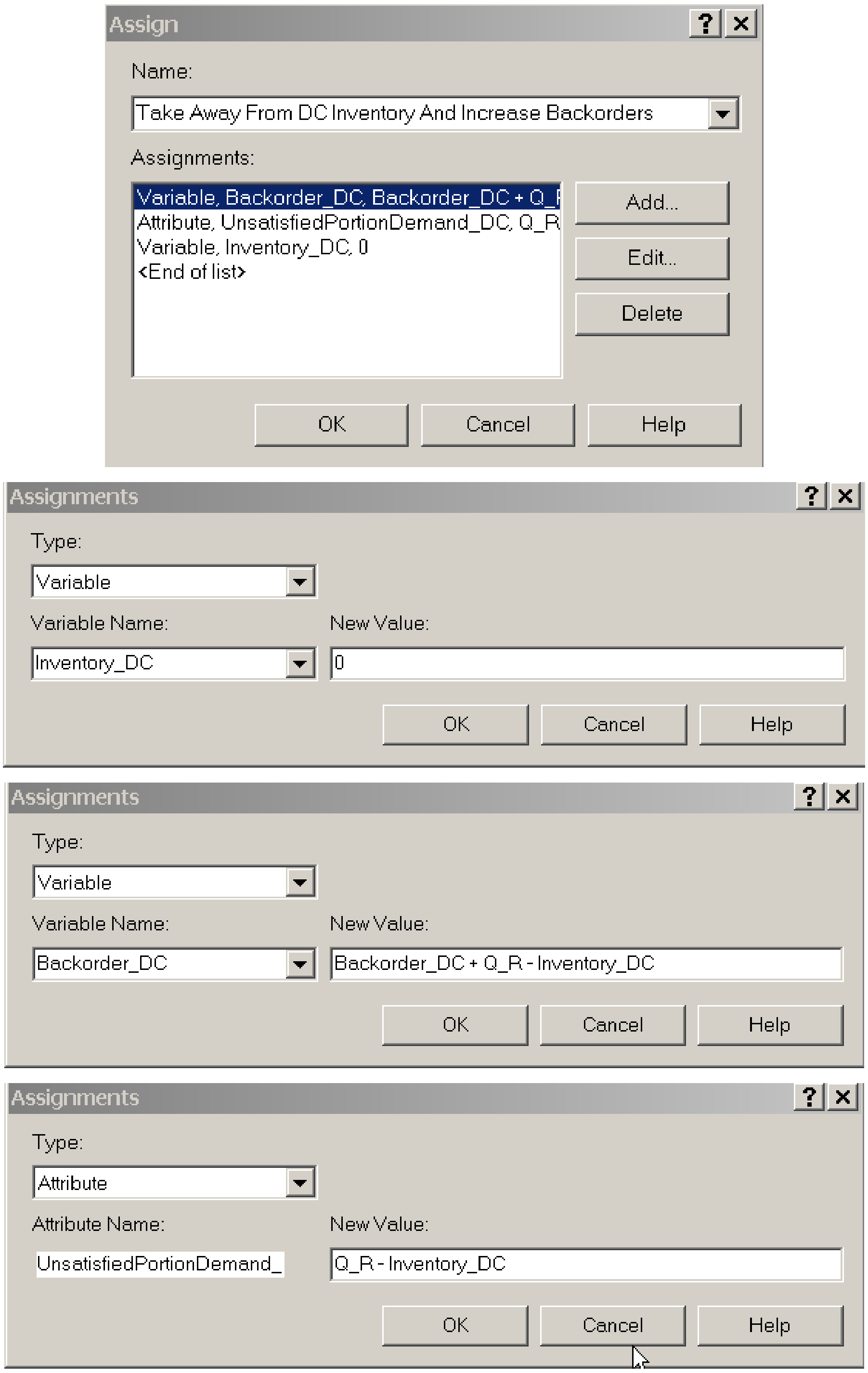
Figure 6.44 Arena model of the inventory management segment for the distribution center.

InventoryPosition_Retailer= InventoryPosition_Retailer+Q_R

to immediately update the retailer inventory position. Either way, the customer entity proceeds to be disposed of in module *Dispose Customer Demand*.

6.3.4 Inventory management segment for distribution center

Figure 6.44 depicts the Arena model's inventory management segment for the distribution center. This model segment generates incoming retailer orders, updates distribution-center inventory levels, triggers replenishment orders from the output buffer at the manufacturing plant, sends shipments to the retailer, and updates the retailer inventory level.



Assign

Name: Take Away From DC Inventory And Increase Backorders

Assignments:
Variable, Backorder_DC, Backorder_DC + Q_R
Attribute, UnsatisfiedPortionDemand_DC, Q_R
Variable, Inventory_DC, 0
<End of list>

Assignments

Type: Variable
Variable Name: Inventory_DC
New Value: 0

Assignments

Type: Variable
Variable Name: Backorder_DC
New Value: Backorder_DC + Q_R - Inventory_DC

Assignments

Type: Attribute
Attribute Name: UnsatisfiedPortionDemand_
New Value: Q_R - Inventory_DC

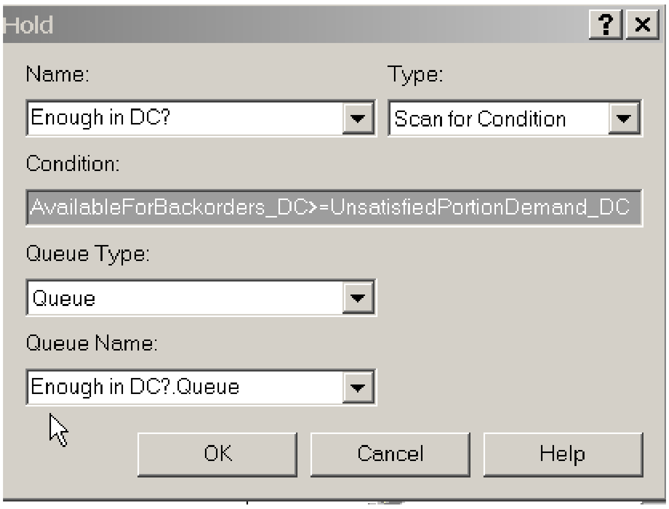
Figure 6.45 Dialog boxes of the **Assign** module *Take Away From DC Inventory And Increase Backorders* (top) and its associated Assignments dialog boxes (bottom).

The **Create** module, called *Demand Arrival To DC*, creates a single order entity at time 0, which later generates a pending order to be shipped from the distribution center to the retailer. The order entity enters the **Hold** module, called *Shall We Release Retailer Order?*, where it is held until the variable *Order_DC* equals 1, whereupon the order entity proceeds to the **Assign** module, called *Change Flag For Order From Retailer*, and sets *Order_DC* to 0. The order entity then proceeds to the **Separate** module, called *Separate 1*, where it duplicates itself. The order entity itself proceeds to enter the system as a retailer order, while its duplicate loops back to the **Hold** module *Shall We Release Retailer Order?* to generate the next retailer pending order.

The retailer order entity is next tallied in the **Record** module, called *Tally DC Demand*, and then enters the **Assign** module, called *Take Away From DC Inventory Position*, where the variable *InventoryPosition_DC* is decremented by the value of the variable *Q_R*. The order entity then enters the **Decide** module, called *Order From Output Buffer*, to check whether the variable *InventoryPosition_DC* has down-crossed level $R_{DC.}$ If it has, then the order entity proceeds to the **Assign** module, called *Order From Output Buffer And Update DC Inventory Position*, and performs two assignments. The first assignment sets *Order_Output* = 1; this assignment promptly releases the order entity currently detained in the **Hold** module *Shall We Release DC Order?* in the output-buffer inventory management segment (see *Figure 6.47*). The second assignment increments the distribution-center inventory position variable *Inventory-Position_DC* by the value of the variable *Q_DC*.

The order entity next enters the **Decide** module, called *Check DC Inventory*, to test whether the retailer has sufficient inventory on hand. The test has two possible outcomes: (1) If the condition *Inventory_DC≥Q_R* holds, then the order entity takes the *True* exit to the **Assign** module, called *Take Away From DC Inventory*, where it decrements the on-hand inventory by *Q_R*. (2) If the condition *Inventory_DC* < *Q_R* holds, then the order entity takes the *False* exit. In this case, the demand is not fully satisfied, and should be backordered from the manufacturing plant. To this end, the order entity proceeds to the **Assign** module, called *Take Away From DC Inventory And Increase Backorders*, where three assignments take place:

1. The backorder level is incremented by the shortage.
2. The attribute *UnsatisfiedPortionDemand_DC* of the order entity is assigned the unsatisfied portion of the demand.
3. The inventory level is decremented by 1.



Name: Enough in DC?
Type: Scan for Condition
Condition: AvailableForBackorders_DC>=UnsatisfiedPortionDemand_DC
Queue Type: Queue
Queue Name: Enough in DC?.Queue

Figure 6.46 Dialog box of the **Hold** module *Enough In DC?*

Figure 6.45 depicts these assignments by displaying the dialog box of the **Assign** module *Take Away From DC Inventory And Increase Backorders* (top) and its associated *Assignments dialog boxes* (bottom).

The order entity next proceeds to the **Hold** module, called *Enough In DC?*, where it is detained until sufficient inventory accumulates in the distribution center to satisfy the shortage, as shown in the Condition field of the dialog box in *Figure 6.46*.

The variable *AvailableForBackorders_DC* is used to track the number of inventory product units on hand that are currently available to satisfy backorders during stock-out periods. That is, the order entity is detained in this **Hold** module until sufficient inventory becomes available to satisfy its unsatisfied portion. Note that multiple order entities may be simultaneously detained in this **Hold** module, but those will be satisfied in their order of arrival at this module, since the **Hold** module's queue discipline is *FCFS*. Now, if the condition

$$AvailableForBackorders\_DC \geq UnsatisfiedPortionDemand\_DC$$

holds, then the order entity is released and proceeds to the **Assign** module, called *Decrease Available Items For Backorders In DC*, where the variable *Available-ForBackorders_DC* is decremented by the value of the variable *UnsatisfiedPortion-Demand_DC.*

The retailer order entity is now ready for shipment to the retailer. Recall that we assumed that product units are processed sequentially in the transportation system, so as to preclude overtaking. To enforce this rule, we use the **Process** module, called *Order Arrival At Retailer*, to model the transportation delay from the distribution center to the retailer. The order entity next enters the **Assign** module, called *Update Retailer Inventory*, to update the retailer inventory level by incrementing variable *Inventory_Retailer* by *Q_R*. Keep in mind that the inventory position at the retailer was updated when the replenishment order was placed, but the inventory level at the retailer is only updated after the shipment is actually received there. Finally, the order entity proceeds to be disposed of in the **Dispose** module, called *Dispose Retailer Demand*.

6.3.5 Inventory management segment for output buffer

Figure 6.47 depicts the output-buffer inventory management segment of the Arena model. This model segment generates distribution-center orders, updates the output-buffer inventory level, triggers resumption of suspended manufacturing as necessary, sends shipments to the distribution center, and updates the distribution-center inventory level.

The logic of generating distribution center orders placed at the output buffer is virtually identical to the generation logic used in the previous segment, and therefore will not be described in detail. Recall that a pending order entity (bound for the distribution center) is detained in the **Hold** module, called *Shall We Release DC Order?*.

This order entity is released once the variable *Order_Output* is set to 1. Similarly to the logic in the previous segment, the order entity proceeds to the **Separate** module, called *Separate 2*, where it duplicates itself. The order entity itself proceeds to enter the system as a distribution center order, while the duplicate loops back to the **Hold** module, called Shall We Release DC Order?, to generate the next pending order of the distribution center.

The order entity next enters the **Record** module, called *Tally Output Buffer Demand*, to tally the order quantity. It then proceeds to the **Decide** module, called *Check Output Buffer Inventory*, to check whether the output buffer has sufficient inventory on hand to satisfy its demand. Again, two outcomes are possible. (1) If the condition *Inventory_Output* ≥ *Q_DC* holds, then the order entity takes the *True* exit to the **Assign** module, called *Take Away from Output Inventory*, where it decrements the on-hand inventory by *Q_DC*. (2) If the condition

Inventory_Output < *Q_DC* holds, then the order entity takes the *False* exit. In this case, the demand is not fully satisfied and is backordered from the output buffer, and the order entity enters the **Assign** module, called *Take Away From Output Inventory And Increase Backorders*, to perform three assignments:

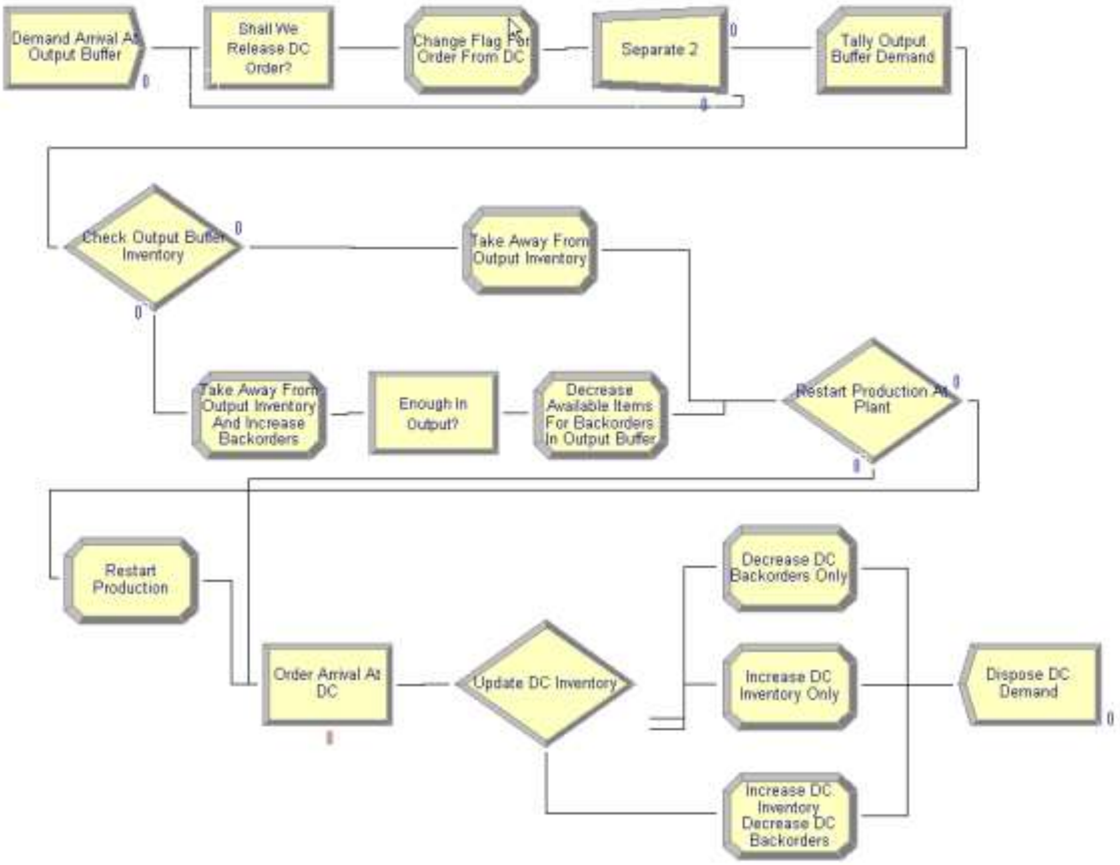

Demand Arrival At Output Buffer
Shall We Release DC Order?
Change Flag For Order From DC
Separate 2
Tally Output Buffer Demand
Check Output Buffer Inventory
Take Away From Output Inventory
Take Away From Output Inventory And Increase Backorders
Enough In Output?
Decrease Available Items For Backorders In Output Buffer
Restart Production At Plant
Restart Production
Order Arrival At DC
Update DC Inventory
Decrease DC Backorders Only
Increase DC Inventory Only
Increase DC Inventory Decrease DC Backorders
Dispose DC Demand

Figure 6.47 Arena model of the inventory management segment for the output buffer.

1. The backorder level variable *Backorder_Output* is incremented by the shortage amount.
2. The *UnsatisfiedPortionDemand_Output* attribute of the order entity is assigned the unsatisfied portion of the demand.
3. The inventory level *Inventory_Output* at the output buffer is set to 0.

The order entity next proceeds to the **Hold** module, called *Enough In Output?*, where it is detained until sufficient inventory accumulates in the output buffer, that is, until the condition

$$AvailableForBackorders\_Output \geq UnsatisfiedPortionDemand\_Output$$

becomes true. Once this happens, the order entity is released and proceeds to the **Assign** module, called *Decrease Available Items For Backorders In Output Buffer*, where the variable *AvailableForBackorders_Output* is decremented by the shortage portion, *UnsatisfiedPortionDemand_Output*.

The order entity then proceeds to the **Decide** module, called *Restart Production At Plant*, to check whether the variable *Inventory_Output* has just down-crossed the reorder level *r_Plant*. If it has, then the order entity enters the **Assign** module, called *Restart Production*, where it sets *Production_Plant* = 1, which would promptly release the pending production entity currently detained in the **Hold** module, called *Shall We Produce?, i*n the input-buffer inventory/production management segment (see *Figure 6.48*), effectively resuming the production process.

The order entity is now ready for shipment to the distribution center. To this end, it proceeds to the **Process** module, called *Order Arrival At DC*, to model the transportation delay from the output buffer to the distribution center. The order entity next proceeds to the **Decide** module, called *Update DC Inventory*, where three outcomes are possible:

1. If the condition *Backorder_DC* ≥ *Q_DC* holds, then the order entity takes the exit for the **Assign** module, called *Decrease DC Backorders Only*, where it decrements *Backorder_DC* by *Q_DC* and increments *AvailableForBackorders_DC* by *Q_DC*.
2. If the condition *Backorder_DC* = 0 holds, then the order entity takes the exit for the **Assign** module, called *Increase DC Inventory Only*, where it increments *Inventory_DC* by *Q_DC*.
3. If the condition 0 < *Backorder_DC* < *Q_DC* holds, then the order entity takes the exit for the **Assign** module, called *Increase DC Inventory Decrease DC Backorders*, where it sets *Inventory_DC* = *Inventory_DC* + *Q_DC* – *Backorder_DC*, increments *AvailableForBackorders_DC by Backorder_DC* by 1, and sets *Backorder_DC* to 0.

Finally, the order entity proceeds to be disposed of in the **Dispose** module, called *Dispose DC Demand*.

6.3.6 Production/inventory management segment for input buffer

Figure 6.48 depicts the input-buffer production/inventory management segment of the **Arena** model. This model segment manages raw-material consumption and finished goods production by keeping track of a circulating control entity that modulates the suspension and resumption of production.

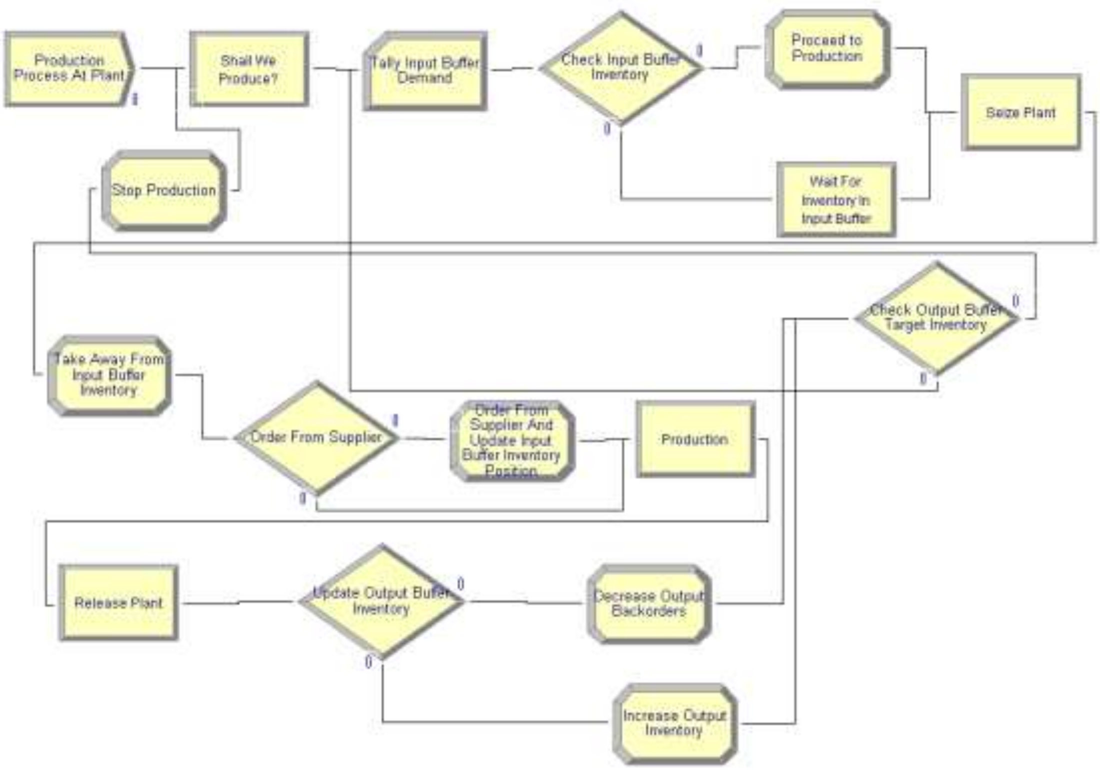

Production Process At Plant
Shall We Produce?
Tally Input Buffer Demand
Check Input Buffer Inventory
Proceed to Production
Seize Plant
Stop Production
Wait For Inventory In Input Buffer
Check Output Buffer Target Inventory
Take Away From Input Buffer Inventory
Order From Supplier
Order From Supplier And Update Input Buffer Inventory Position
Production
Release Plant
Update Output Buffer Inventory
Decrease Output Backorders
Increase Output Inventory

Figure 6.48 Arena model of the production/inventory management segment for the input buffer.

To manufacture a product unit, the plant removes one unit of raw material from the input buffer, processes it, and adds the resulting finished product to the output-buffer inventory and updates its level. If the target level is subsequently reached, then further production is suspended; production is re-started later when the reorder point is down-crossed. Production may also be stopped due to starvation resulting from depletion of raw material in the input buffer, until it is replenished from the supplier.

The **Create** module, called *Production Process At Plant*, generates a single control entity at time 0, which cycles in the segment such that each cycle represents a production cycle. The control entity first enters the **Hold** module, called *Shall We Produce?*, and is detained there until the production is allowed to re-start (recall that this happens when the inventory level in the output buffer down-crosses the reorder level there). Once production is allowed to resume, the control entity enters the **Record** module, called *Tally Input Buffer Demand*, to tally the next product unit.

The control entity then proceeds to the **Decide** module, called *Check Input Buffer Inventory*, to check if there is raw material in the input buffer. If true, it proceeds to the **Assign** module, called *Proceed to Production*, to update the number of satisfied demands. If false, starvation is in effect and the control entity takes the *False* exit to the **Hold** module, called *Wait For Inventory In Input Buffer*, until the condition *Inventory_Input* ≥ 1 becomes true. Once this condition holds, the control entity proceeds to the **Seize** module, called *Seize Plant*, where it immediately seizes the Plant resource. To model the consumption of one unit of raw material, the control entity enters the **Assign** module, called *Take Away From Input Buffer Inventory*, and decrements by 1 both the inventory on-hand variable *Inventory_Input* and the inventory position variable *InventoryPosition_ Input*.

Similarly to the previous segments, the control entity next proceeds to the **Decide** module, called *Order From Supplier*, to check whether the reorder point at the input buffer has been down-crossed. If it has, a raw material replenishment is promptly triggered by releasing the order entity currently detained in the **Hold** module, called *Shall We Release Plant Order?*, in the supplier inventory management segment (see *Figure 6.49*). The control entity next enters the **Assign** module, called *Order From Supplier And Update Input Buffer Inventory Position*, where it updates the input buffer inventory position (variable *InventoryPosition_Input*) and triggers a supplier order by setting the variable *Order_Supplier* to 1 of the variable *Q_Input*.

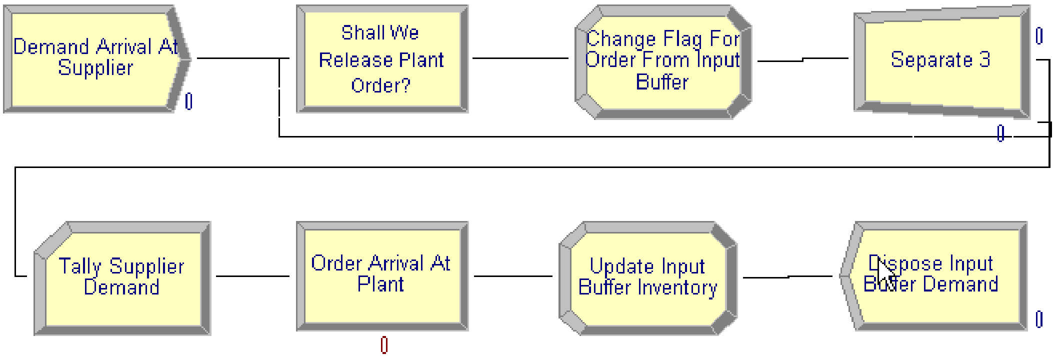

Demand Arrival At Supplier
0
Shall We Release Plant Order?
Change Flag For Order From Input Buffer
Separate 3
0
0
Tally Supplier Demand
Order Arrival At Plant
0
Update Input Buffer Inventory
Dispose Input Buffer Demand
0

Figure 6.49 Arena model of the production/inventory management segment for the supplier.

In all cases, the control entity eventually enters the **Delay** module, called *Production*, to model the manufacturing time delay of one product unit, following which it proceeds to the **Release** module, called *Release Plant*, where it releases the *Plant* resource. Before adding the finished product unit to inventory, the control entity enters the **Decide** module, called *Update Output Buffer Inventory*, to check if there are any pending backorders in the output buffer (i.e., whether the condition *Backorder_Output* ≥ 1 holds). If there are pending backorders, the control entity takes the *True* exit for the **Assign** module, called *Decrease Output Backorder*, where a pending backorder is satisfied by decrementing the variable *Backorder_Output* by 1 and incrementing the variable *AvailableForBackorders_Output* by 1. Otherwise, if no backorders are pending, the control entity takes the *False* exit for the **Assign** module, called *Increase Output Inventory*, where the output buffer inventory level is updated by incrementing the variable *Inventory_Output* by 1.

The control entity then proceeds to the **Decide** module, called *Check Output Buffer Target Inventory*, to check whether the inventory level at the output buffer has reached its target level. If it has, then the control entity takes the *True* exit for the **Assign** module, called *Stop Production*, and sets *Production_Plant* = 0 to suspend production, after which it cycles back to the **Hold** module, called *Shall We Produce?*, to wait until the next production cycle is resumed. Otherwise, the control entity takes the *False* exit and cycles back to the **Record** module, called *Tally Input Buffer Demand,* to start the next production cycle.

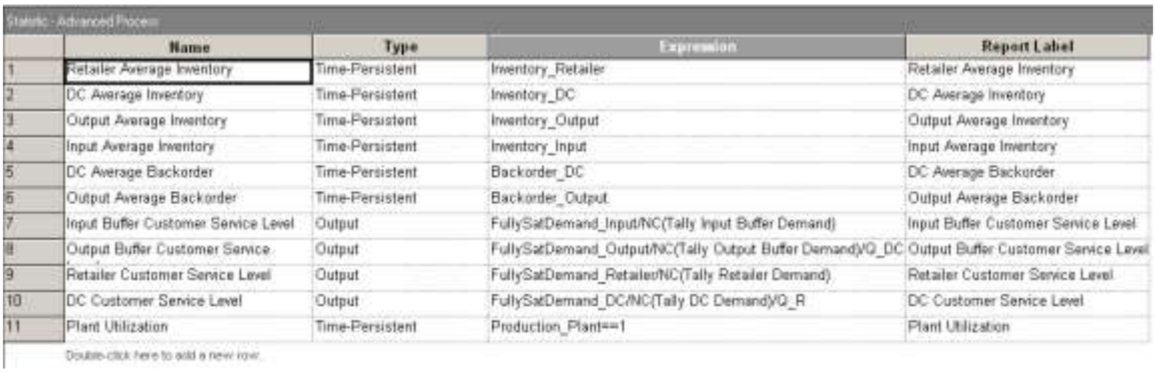
6.3.7 Inventory management segment for supplier

Figure 6.49 depicts the supplier inventory management segment of the Arena model. This model segment generates input buffer orders, sends shipments from the supplier to the input buffer, and updates the input buffer inventory level.

The logic of generating input buffer orders to the supplier is virtually identical to the generation logic used in the previous segments, and therefore will not be repeated. Note, however, that this model segment is a bit simpler than its counterparts; because the supplier always has sufficient inventory on hand, replenishment delays reduce to transportation delays.

6.3.8 Statistics collection

Figure 6.50 displays the spreadsheet view of the **Statistics** module for the multi-stage supply chain model. The spreadsheet includes *Time-Persistent* statistics of on-hand inventory levels at the retailer, distribution center, output buffer, and input buffer, as well as *Time-Persistent* statistics of backorder levels at the distribution center and output buffer. It also includes the *Time-Persistent* statistic of plant utilization, that is, the percentage of time the plant is busy producing. Finally, the *Output* statistics estimate customer service levels at each stage in terms of the fill rate, namely, the probability (fraction) of orders that were satisfied from on-hand inventory, without experiencing backordering.



Statistic - Advanced Process

	Name	Type	Expression	Report Label
1	Retailer Average Inventory	Time-Persistent	Inventory_Retailer	Retailer Average Inventory
2	DC Average Inventory	Time-Persistent	Inventory_DC	DC Average Inventory
3	Output Average Inventory	Time-Persistent	Inventory_Output	Output Average Inventory
4	Input Average Inventory	Time-Persistent	Inventory_Input	Input Average Inventory
5	DC Average Backorder	Time-Persistent	Backorder_DC	DC Average Backorder
6	Output Average Backorder	Time-Persistent	Backorder_Output	Output Average Backorder
7	Input Buffer Customer Service Level	Output	FullySatDemand_Input/NC(Tally Input Buffer Demand)	Input Buffer Customer Service Level
8	Output Buffer Customer Service	Output	FullySatDemand_Output/NC(Tally Output Buffer Demand)/Q_DC	Output Buffer Customer Service Level
9	Retailer Customer Service Level	Output	FullySatDemand_Retailer/NC(Tally Retailer Demand)	Retailer Customer Service Level
10	DC Customer Service Level	Output	FullySatDemand_DC/NC(Tally DC Demand)/Q_R	DC Customer Service Level
11	Plant Utilization	Time-Persistent	Production_Plant==1	Plant Utilization

Figure 6.50 Dialog spreadsheet of the **Statistic** module for the Multi-stage supply chain system.

12.3.9 Simulation results

The simulation study explored six supply-chain scenarios with varying demand rates. To attain reliable statistical outputs, each simulation run consisted of a single replication simulating 50,000,000 product-unit departures.

Simulation outputs for the six parameter settings of the demand arrival rate, l, are displayed in *Table 6.4*. The computed statistics are average inventory levels, average backorder levels, and customer fill rates (table columns) at four stages: input buffer, output buffer, distribution center, and retailer (table rows). The notation N/A stands for "not applicable."

Table 6.4

Simulation outputs for a replication of Multi-stage supply chain

		λ=1.0			λ=1.1	
	Inventory Level	**Backorder Level**	**Fill Rate (%)**	**Inventory Level**	**Backorder Level**	**Fill Rate (%)**
Input Buffer	15.0203	N/A	99.88	14.8334	N/A	99.87
Output Buffer	23.6452	0.0003	99.85	22.9043	0.0013	99.58
DC	23.0257	0.0000	100.00	22.8343	0.0000	100.00
Retailer	8.5209	N/A	98.61	8.3328	N/A	98.14
		λ=1.2			λ=1.3	
	Inventory Level	**Backorder Level**	**Fill Rate (%)**	**Inventory Level**	**Backorder Level**	**Fill Rate (%)**
Input Buffer	14.6480	N/A	99.86	14.4680	N/A	99.85
Output Buffer	22.0530	0.0052	98.88	21.0038	0.0199	97.23
DC	22.6296	0.0000	100.00	22.3842	0.0005	99.98
Retailer	8.1465	N/A	97.58	7.9646	N/A	96.97
		λ=1.4			λ=1.5	
	Inventory Level	**Backorder Level**	**Fill Rate (%)**	**Inventory Level**	**Backorder Level**	**Fill Rate (%)**
Input Buffer	14.2924	N/A	99.84	14.1222	N/A	99.82
Output Buffer	19.6205	0.0731	93.57	17.6510	0.2507	86.11
DC	22.0235	0.0031	99.90	21.3356	0.0169	99.56
Retailer	7.7828	N/A	96.26	7.5964	N/A	95.44

An examination of *Table 6.4* reveals that for a parameter setting of l = 1:0, the average inventory level in the input buffer is 15.0203 and its fill rate is 99.88%. Furthermore, the average inventory level at the retailer is 8.5209 and its fill rate is 98.61%. This means that 1.39% of the demand at the retailer is lost. Varying the demand-arrival rate parameter for each given inventory shows an expected pattern: a parameter increase results in concomitant decreases of the corresponding average inventory levels and fill rates, and concomitant increases in the corresponding average backorder levels.

Note that as the system load increases, the customer service level deteriorates the most at the output buffer of the manufacturing plant. This is due to the fact that the plant has a fixed production rate that must cope with an increasing demand rate. Thus, the highest supply chain stage (excluding its supplier) is most affected by the increasing load. This phenomenon is consistent with the bullwhip effect explained in the beginning of this chapter.

Exercises

1. Inventory management. A computer store sells and maintains an inventory of fax machines (units). On average, one unit is sold per day, but the store experiences a burst of customers on some days. A marketing survey suggests that customer interarrival times are iid exponentially distributed with rate 1 per day.

When the inventory on hand drops down to 5 units, an order of 20 units is placed with the supplier, and the lead time is uniformly distributed between 5 and 10 days. Unsatisfied customers check back with the store and wait for the order to arrive (backordering case). Keep in mind that another order is placed immediately upon order backorder arrival, should the inventory level fall below the reorder level after satisfying all backorders.

a. Develop an Arena model of the computer store, and simulate it for 2 years of daily (9:00 A.M. to 5:00 P.M.) operation.

b. What is the average stock on hand?

c. What is the average backorder level?

d. What percentage of the time does the store run out of stock? Note that this is the probability that a customer’s order is not satisfied and is subsequently backordered.

e. What is the percentage of time that the stock on hand is above the reorder level?

f. Modify the customer arrival rate to 0.05 customers per hour during the first 4-hour period, and 0.25 customers per hour during the second 4-hour period of any day. Repeat items a through e for the modified system. Compare the performance measures of items a through e in the original and modified systems.

2. Procurement auction. Procurement auctions (also known as reverse auctions) constitute a common purchasing method designed to lower purchasing costs by inducing competition among suppliers. In this type of auction, there is a single buyer of goods, and multiple potential suppliers bidding to sell the requisite goods; winner selection is usually based on the lowest price. The procurer sends a “request for bid” to all available suppliers, and the suppliers respond immediately with bids consisting of two attributes, price and lead time. The lowest bid winner ships the ordered quantity, which reaches the procurer after the stated lead time. Consider a procuring company that operates as follows:

There are three independent suppliers, each of which generates iid prices from the *Unif*(10, 20) distribution, and iid lead times (in hours) from the *Tria*(2, 4, 5) distribution.

The company follows a (s, Q) inventory control policy, with order quantity Q = 80 and reorder point s = 20, where all shortages are backordered. An auction is initiated whenever the reorder point is reached.

Customer interarrival times (in hours) follow the exponential distribution with rate λ= 1.

Demand quantities of arriving customers are iid and drawn from the discrete distribution

$$\Pr(D= d)= \left|\begin{array}{ll} 0.3, & d=1 \\ 0.2, & d=2 \\ 0.3, & d=3 \\ 0.2, & d=4 \end{array}\right.$$

The initial on-hand inventory is 50 units.

Develop an Arena model of the company operations, and simulate it for 100,000 hours. Compute the following performance metrics of the system:

a. Average price of winning bids

b. Average lead time of winning bids

c. Average on-hand inventory

3. Multiple products with rapid switchover. Consider a production/inventory system similar to the one described in Section 6.2. The production facility produces product types 1, 2, and 3,

and these are moved to a warehouse to supply three distinct incoming customer streams, denoted by types 1, 2, and 3, respectively. A raw material storage feeds the production process.

The production facility produces products in single units, switching from the production of one product type to another, depending on inventory levels. How-ever, product types have priorities in production, with product 1 having the highest priority and product 3 the lowest. Lower-priority products are subject to preemption by high-priority ones. More specifically, if the inventory level of a higher-priority product down-crosses its reorder level while the production of a lower-priority product is in progress, then the production process switches to the higher-priority product as soon as the current lower-priority product unit is completed. This policy provides a faster response to higher-priority products at the expense of more frequent setups. Clearly, this is a desirable policy in scenarios with low setup costs. However, in this problem setup times are assumed to be 0.

Each product has its own parameters, as shown in the following table.

Product Type	Reorder Point r	Target Level R	Initial Inventory	Processing Times(minutes)	Demand Arrivals (hours)	Demand Quantity
1	10	30	5	15	Exp(2)	Unif(2, 8)
2	25	50	40	10	Exp(4)	Unif(6, 10)
3	50	80	30	5	Exp(6)	Unif(5, 10)

The system is subject to the following assumptions and operating rules:

There is always sufficient raw material in storage, so the production process never starves.

Lot processing time is deterministic as shown in the table above.

The warehouse implements the (*s*, *S*) inventory control policy for each product (see Section 6.2).

On customer arrival, the inventory is immediately checked. If there is sufficient stock on hand, the incoming demand is promptly satisfied. Otherwise, the unsatisfied portion of the demand is backordered.

Develop an Arena model for the system and simulate it for 1 year (24-hour operation) and estimate the following statistics:

a. Probability of an outstanding order, by product type

b. Production-facility downtime probability

c. Average inventory level, by product type

d. Percentage of customers whose demand is not completely satisfied, by product type

e. Average backorder level, by product type

f. Average backordered quantity per backorder, by product type

3. Multiple products with finite raw material. Consider the previous problem with the following extension. The system now has separate raw-material storage areas for each product type at the production facility. The production process uses one unit of raw material to process one unit of finished product. Raw-material storage is maintained using the (*s*, *Q*) policy with lead times. Ordering and lead-time data are given in the table below.

Product Type	Reorder Point	Order Quantity	Initial Inventory	Lead Time (days)
1	20	10	25	Unif(1, 2)
2	30	20	40	Unif(1, 3)
3	40	30	50	Unif(0.5, 4)

If the facility runs out of raw material while producing a product unit, it switches to the highest-priority product that needs attention and has raw material in its storage.

Develop an Arena model for the system and simulate it for 1 year (24-hour operation). Estimate the following statistics in addition to those in the previous problem:

a. Average raw material inventory levels, by product type

b. Average length of stock-out periods, by product type

c. Probability of stock-outs, by product type

5. Optimal inventory policy. Home Needs (**HN**) sells garden sprinklers through two stores. It supplies both stores from a single distribution center (**DC**), and the **DC** is supplied in turn from a single factory. Both stores operate around the clock (24-hour business days), and the unsatisfied portion of customers demand at the stores is lost rather than backordered. All lead times are deterministic.

HN is interested in implementing order-up-to-*S* inventory control policies at its stores and **DC**. To this end, **HN** needs to forecast demand quantities arriving over a lead-time period (lead-time demand, for short), and excess inventory quantities needed to guard against excessive future stock-outs (safety stock). Let the lead-time demand forecast at the end of day *n* be given by

$$LTD_n = D_n \text{ x } LT,$$

where D_n is the daily demand observed on day *n*, and *LT* is a given deterministic lead time. Let the safety-stock forecast at the end of business day *n* be given by

$$SS_n = SSF \text{ x } LTD_n,$$

where *SSF* is a prescribed safety stock factor. Finally, let IP_n be the inventory position at the end of business day *n*.

At the end of each 24-hour business day, a review is performed at both stores and the DC, and an order is placed for each store, if necessary. The decision to place an order is based on the following rule at the end of each business day n:

Do not order if $IP_n - LTD_n > SS_n$.

Otherwise, place an order of quantity $Q_n = SS_n - IP_n + LTD_n$.

The system is subject to the following assumptions:

HN observed that total daily customer demand at the first store is uniform between 5 and 30 sprinklers, and at the second store it is uniform between 10 and 40 sprinklers. Daily demand becomes known at the 8th hour into the business day, (recall that a business day is 24 hours long).

The unsatisfied portion of the demand is lost.

Except for the first day, if the daily demand at the **DC** is 0, then the *LTD* and *SS* forecasts carry over from the previous day.

The factory operates as a single-stage process, and takes 2 minutes to produce each product unit.

Orders from both stores take 1 hour to prepare and reach the **DC**.

Lead times for orders are provided in the following table:

Origination/Destination	Lead Time (days)
DC to store 1	2
DC to store 2	2
Factory to DC	3 + manufacturing lead time

The initial on-hand inventory at the stores and **DC** and the corresponding *SSF* parameters are given in the following table:

Location	Initial Inventory	SSF
Store 1	35	1
Store 2	50	1
DC	255	0.5

Develop an Arena model for the system, and estimate the minimal value of *s* that results in a 90% fill rate based on 20-year replications. Estimate the following statistics:

Percentages of fully satisfied and partially satisfied customers at each store and orders at the **DC**

Average inventory levels at each store and at the **DC**

Average lost portion of sales at each store

CHAPTER 7

Modeling an Automotive Maintenance and Repair Shop

In Chapter 4, we showed you the kinds of modeling you could do with modules from the Basic Process panel. They are relatively high-level, easy-to-use modules that will usually take you a long way toward building a model at a sufficient level of detail. Sometimes it's all you'll need.

But sometimes it isn't. As you gain experience in modeling, and as your models become bigger, more complex, and more detailed, you might find that you'd like to be able to control or model things at a lower level, in more detail, or just differently from what the modules of the Basic Process panel offer. Arena doesn't strand you at this level or force you to accept a limited number of "canned" modeling constructs, nor does it force you to learn a programming language or some pseudo-programming syntax to capture complicated system aspects. Rather, it offers a rich and deep hierarchy of several different modeling levels to get the flexibility you might need to model some specific logic appropriately. It's probably a good idea to start with the high-level modules, take them as far as they'll go (may be that's all the way), and when you need greater flexibility than they provide, go to a lower, more detailed level. This structure allows you to exploit the easy, high-level modeling constructs to the extent possible, yet allows you to drill down when you need to. And because standard Arena modules provide all of this modeling power, you'll be familiar with how to use them. To put them to work, you'll simply need to become familiar with what they do.

This chapter explores some (certainly not all) of the detailed, lower-level modeling constructs available in the Advanced Process and Blocks panels; the latter panel provides the lowest-level model logic where modules correspond to the blocks in the SIMAN simulation language that underlies Arena. The example we'll use for this is a fairly complex model of an automotive repair and maintenance shop. We'll also touch on the important topics of non-stationary (time-dependent) arrival processes, model debugging, and a greater level of customization in animation.

Section 7.1 describes the system and Section 7.2 talks about how to model it using some new Arena modeling concepts. Section 7.3 describes our basic modeling strategy. The model logic is developed in Section 7.4. The unhappy (but inevitable) issue of debugging is taken up in Section 7.5. Corresponding to the more detailed modeling in this chapter, Section 7.6 indicates some ways you can fine-tune your animations to create some nonstandard effects. In Sections 7.7 and 7.8, we embellish our model and present several new Arena modeling concepts. In Section 7.9, we show you how to modify the original model to create the embellished model. We close the chapter in Section 7.10 with an altogether different kind of model, an inventory system, and take this opportunity to illustrate use of one of Arena's lowest and most detailed modeling levels, the Blocks panel that composes the underlying SIMAN simulation language.

After reading this chapter, you should be able to build very detailed and complex models and be able to exploit Arena's rich and deep hierarchy of modeling levels.

7.1 Model 7-1: An Automotive Maintenance and Repair Shop

A large automotive dealership located in the downtown area of a medium-sized city has determined that its maintenance and repair shop has outgrown its present facility and must be expanded. Due to limitations of their current location, they are considering building an additional three-bay facility in a suburb. This expansion facility would provide not only the additional repair and maintenance capacity, but would locate the facility near many of its current customers. Appointments for both locations would be scheduled from the downtown facility to allow for better control of the workflow; and all of the major repair work would continue to be performed at the downtown location.

According to the plan, customers will be permitted to make appointments for service at the new suburban facility up to three working days in advance (no appointments are made for service to be performed on the call-in day). For example, customers who call in on Wednesday may be allowed to make appointments only for Thursday, Friday, or Monday. If a customer is unable to schedule an appointment during the given three-day period, they may either call back the next day or may schedule the work to be performed at the downtown location. The dealership has extensive data on customer service requests and analysis of the data indicates that the average number of customer request calls is 29 per day and follows a (stationary) Poisson process throughout the day (that is, inter-call times are independent and identically exponentially distributed). Analysis of the data also reveals that 55% of the callers would like to schedule an appointment for the next day; 30%, for the day after; and the remaining 15%, for three days out. If an appointment cannot be scheduled for the chosen day, there is a 90% chance that they will want to schedule for the following day.

Most customers (80%) elect to leave their vehicles for the entire day, while a few customers (20%) like to wait for the service to be completed. If customers choose to wait, they will be given an approximate wait time. This wait time is automatically calculated by taking the estimated service time for the work to be performed (referred to as Book Time) and adding an allowance factor (one hour). One of the goals in building this new facility is to provide a high level of customer satisfaction. Thus, it has been decided that no more than five appointments for customers who want to wait for their vehicles will be scheduled on any day.

The dealership uses a set of standard estimated service times (Book Times) to calculate the cost of service, regardless of the actual time. The predicted distribution for these Book Times for the types of service activities to be offered at this facility has been determined to be a scaled and shifted Beta distribution (44+90*BETA(2,3) in minutes, truncated down to the next smaller whole minute). The actual service time varies somewhat from the Book Time and follows a Gamma distribution, given by GAMM(Book Time/l.05, 1.05).

The dealership wants to yield a profit at the new facility and doesn't know how many appointments to schedule each day, so for now the number of appointments per day will be based on a maximum number of Book Time hours scheduled for each day. This value is based on three bays each with eight hours per day available for service. Since the actual service time tends to differ somewhat from the Book Time, a value of 24 hours has been chosen.

Assuming a five-day workweek of eight hours per day, the cost per hour for each bay is estimated at $45, including all capital and labor. The customer is charged based on the Book Time at $78 per hour. Since there is a fair amount of variability in the actual service time, the dealership will attempt to complete all scheduled service on the day repairs are started. To compensate for service-time variability, each service bay can stay open a maximum of three extra hours per day. However, the cost for overtime is $120 per hour per bay. If service cannot be completed in this time, the customer vehicle will be held overnight and service will be completed the next day. If this occurs, the dealership will provide the customer with a loaner vehicle, which costs the dealership $35 per day per vehicle. In cases where the shop load is such that service on a vehicle can't be started on the appointment day, we'll assume that the customer takes his or her vehicle home and returns it the next day and there is no charge for a loaner.

Some statistics of interest for this system are: daily profit, daily Book Time, daily actual service time, daily overtime, and the daily number of wait appointments not completed on time.

7.2 New Modeling Issues

From a simulation viewpoint, this problem is quite different from the ones we covered in Chapters 3 and 4. The most obvious difference is that this system is of a service nature whereas the previous systems were manufacturing-oriented. Although the original version of SIMAN (the simulation language on which Arena is based) was developed for manufacturing applications, the current Arena capabilities also allow for accurate modeling of service systems. Applications in this area include fast-food restaurants, banks, insurance companies, service centers, and many others. Although these systems have some special characteristics, the basic modeling requirements are largely the same as for manufacturing systems.

Now let's take a look at our automotive shop and explore the new requirements. As we proceed, it should become clear that the modeling constructs that we've covered up to this point are insufficient to model this system at the level of detail requested.

7.2.1 Multiple-Way Decisions

When a service appointment is made, we must direct it to the proper appointment day to wait for that day to arrive. To do this, we need the ability to send entities or appointments to five different parts of the model based on the day of the appointment.

We could do this by using a series of **Decide** modules, but it would get ugly and it's not necessary. You may not have been aware of it, but each **Decide** module used in the first three models of Chapter 4 is capable of branching in three or more directions.

7.2.2 Sets

As your models become more complex, you'll often find the need to model an entity arriving at a location or station and to select from one of several similar (but not quite identical) objects.

The most common situation is the selection of an available resource from a pool of resources. Let's assume you have three operators: Shelley, Lynn, and Charity. Any one of these operators can perform the required task, and you would like to select any of the three, as long as one is currently available. The Set module provides the basis for this functionality. Arena sets are groups of similar objects that can be referenced by a common name (the set name) and a set index. The objects that make up the set are referred to as members of the set. Members of a particular set must all be the same type of object, such as resources, queues, pictures, etc. You can collect almost any type of Arena objects into a set, depending on your modeling requirements. An object can also reside in more than one set. Let's assume in our Operators set that Lynn is also qualified as a setup person. Therefore, we might define a second resource set called Setup as Lynn and Doug (Doug's not an operator). Now, if an operator is required, we'd select from the set called Operators; if a setup person is required, we would select from the set called Setup. Lynn might be chosen via either scenario because she's a member of both sets. You can have as many sets as you want with as much or as little overlap as required.

For our repair center, we'll use sets to model service bays, entity pictures, and appointment queues.

7.2.3 Variables and Expressions

In many models, we might want to reuse data in several different places. For example, we might have several places in a model where we need to generate service times from the same distribution. It's possible to enter it into the modules where it is required, but if we decide to change the parameters of this distribution (or the distribution itself) during our experimentation, we'd have to open each dialog box that included a service time and change the value. There are other situations where we might want to keep track of the total number of entities in a system or in a portion of the system. In other cases, we might want to use complex expressions throughout the model. For example, we might want to base a processing time on the part type. Arena Variables and Expressions allow us to fulfill these kinds of needs easily.

The Variable module allows you to define your own global variables and their initial values. Variables can then be referenced in the model by their names. They can also be specified as one- or two-dimensional arrays. The Expression module allows you to define expressions and their associated values. Similar to variables, expressions are referenced in the model by their names and can also be specified as one- or two dimensional arrays. Although variables and expressions may appear to be quite similar, they serve distinctly different functions.

User-defined variables store some real-valued quantity that can be reassigned during the simulation run. For example, we could define a Variable called Wait Time with an initial value of 2 and enter the Variable name wherever a wait time was required. We could also define a Variable called Number in System with an initial value of 0, add 1 to this variable every time a new part entered the system, and subtract 1 from it every time a part exited the system. For our automotive shop, we'll use a Variable called Day to keep track of the current day of the week. We'll also use a number of variables to control how we take appointments and to collect information for calculating statistics at the end of a run.

On the other hand, user-defined expressions don't store a value. Instead, they provide a way of associating a name with some mathematical expression. Whenever the name is referenced in the model, the associated expression is evaluated, and its numerical value is returned. Typically, expressions are used to compute values from a distribution or from a complex equation based on the entity's attribute values or even current system variable values. If the mathematical expression is used in only one place in the model, it might be easier to enter it directly where it is required. However, if the expression is used in several places or the form of the expression to be used depends on an entity attribute, a user defined expression is often better. For our automotive shop, we'll use the Expression module (in the Advanced Process panel) to define expressions to generate the Book Time, the actual service time, and the type of appointment request (wait or leave).

Variables and expressions have many other uses that we hope will become obvious as you become more familiar with Arena.

7.2.4 Submodels

When developing large and complex models, it is often helpful to partition the model into smaller models, called submodels, which may or may not interact. This lets you organize the modeling and testing effort into manageable chunks that can then be linked together. For example, we might partition our automotive shop model into five obvious (well, we think they're obvious) submodels: generate appointment calls, make Appointment, service activity, update performance variables, and control logic.

Arena provides this capability in the form of submodels. This feature allows models to be separated formally into hierarchical views, called submodels, each with its own full workspace on your screen, which you view one at a time, as well as the overall view (called the Top-Level Model) of your model and any submodels. Each submodel can contain any object supported in a normal model window (such as spreadsheet modules, static graphics, and animation). Submodels themselves can contain deeper submodels; there is no limit to the amount of nesting that can occur. Submodels can be connected to other modules, to other submodels, or they can stand alone within an Arena model.

Within the Top-Level Model view and each submodel view, you can establish Named Views with associated hot keys to provide easy navigation among different areas of logic and animation (analogous to named ranges in a spreadsheet or bookmarks in a Web browser). The Project Bar's Navigate panel shows a tree listing the Top-Level Model, all of the submodels, and each of their named views. Clicking on a named or submodel view displays that region of the model, allowing easy navigation through the hierarchy and within a particular submodel view.

Although our automotive shop model is not as large or complex as many models you'll find (and build) in practice, we'll use submodels to organize the process and to illustrate the concept.

7.2.5 Duplicating Entities

As you start building more complex models, you'll often find the need to create duplicate entities. These entities are often used as control entities that can perform a variety of functions. A good example of this use is covered in Section 9.3 where we cover entity balking and reneging. There are also situations where you will want to model a batch of entities as a single entity (say a pallet of parts) in part of your model, then sp lit the batch into its individual parts for another section of your model. We’ll illustrate this concept in Section 9.4. In our automotive shop, we'll need the ability to create duplicate entities to represent the incoming calls requesting appointments. We'll use the **Separate** module from the Basic Process panel, which provides the ability to create or clone duplicate entities to accomplish this.

7.2.6 Holding Entities

You'll often need to hold entities at some place in your model until a specific system condition occurs that will allow these entities to proceed. You've already se en one form of this entities waiting in a queue for a resource to be come available. But we're thinking here in more general terms where entities are held based on activities occurring elsewhere in your model; e.g., a system time or a system condition. Two different methods for holding entities are provided by Arena. The first is the ability to hold entities until a signal to proceed is sent by another entity in the model. The second is the ability to hold entities until some system condition exists. We'll need to utilize the first method in our automotive shop model by including the Hold and Signal modules to hold the customer appointments until the day of the scheduled service. The second method will be presented in Section 9.4.

7.2.7 Statistics and Animation

The statistics we'll need are not unusual or greatly different from what we've collected in previous models. However, the type of system and the analysis needs are quite different. Let's start by looking at the analysis needs. A common go al when analyzing service systems is maximizing profit or customer satisfaction while minimizing costs (in the extreme, of course, these are incompatible goals). The key customer-satisfaction measure for our automotive shop would be the number of customers who must wait for their vehicles' service beyond the promised completion time. Clearly, we'd like to minimize this number. The key factor affecting these measures is the maximum number of customer wait jobs that we schedule on any day. Once we've created a model, we could increase or decrease this number easily and determine the impact. This requires that we give more thought to how long we run our simulation and what type of system we have. We'll deal with this in the next section.

Analyzing and improving the profit is a more difficult problem. Our appointments are based on the Book Time, but the actual service time required can vary significantly from one day to the next. Thus, if we're going to manipulate our model variables in an attempt to improve profit, we'd like to know the impact of changing these variables. Our norm al summary report won't directly give us this information. However, it's easy to add summary statistics that will provide the information we need.

It might be helpful to view the animation over several days, much like you might watch the real system. Unfortunately, and unlike our previous models, it's difficult to animate these types of systems so they show movement through the system. We could animate the appointment queues and the resources, but the- resulting animation would be very jumpy and wouldn't provide the time perspective we need. Although we often see animations of these types of systems, the animation is typically used for "show and tell" rather than for analysis. Having said that, we'll show you, how to animate this type of model in Section 7.6.

7.2.8 Terminating or Steady-State

Most (not all) simulations can be classified as either terminating or steady-state. This is primarily an issue of intent or the goal of the study, rather than having much to do with internal model logic or construction.

A terminating simulation is one in which the model dictates specific starting and stopping conditions as a natural reflection of how the target system actually operates. As the name suggests, the simulation will terminate according to some model-specified rule or condition. For instance, a store opens at 9 AM with no customers present, closes its doors at 9 PM, and then continues operation until all customers are "flushed" out. Another example is a job shop that operates for as long as it takes to produce a "run" of 500 completed assemblies specified by the order. The key notion is that the time frame of the simulation has a well-defined (though possibly unknown at the outset) and natural end, as well as a clearly defined way to start up.

A steady-state simulation, on the other hand, is one in which the quantities to be estimated are defined in the long run; that is, over a theoretically infinite time frame. In principle (though usually not in practice), the initial conditions for the simulation don't matter. Of course, a steady-state simulation has to stop at some point, and as you might guess, these runs can get pretty long, so you need to do something to make sure that you're running it long enough. This is an issue we'll take up in Section 7.2. For example, a pediatric emergency room never really stops or restarts, so a steady-state simulation might be appropriate. Sometimes people do a steady-state simulation of a system that actually terminates in order to design for some kind of worst-case or peak-load situation.

We now have to decide which to do for our automotive shop model. Although we'll lead you to believe that the distinction between terminating or non-terminating systems is very clear, that's seldom the case. Some systems appear at first to be one type, but on closer examination, they turn out to be the other. This issue is further complicated by the fact that some systems have elements of both types, and system classification may depend on the types of questions that the analyst needs to address. For example, consider a fast food restaurant that opens at II AM and closes at II PM. If we were interested in the daily operational issues of this restaurant, we'd use a non-stationary Poisson arrival process and analyze the system as a terminating system. If we were interested only in the operation during the rush that occurs for two hours over lunch, we might assume a stationary arrival process at the peak arrival rate and analyze the system as a steady-state system.

At first glance, our automotive shop definitely appears to be a terminating system. The system would appear to start and end empty and idle. However, there is the possibility that not all jobs will be completed on any given day (remember that we will allow only three hours of overtime). Jobs that are held overnight could significantly affect the starting conditions for the next day, so we might want to consider the system to be steady-state. We'll assume that this is not the case and will proceed to analyze our automotive shop in Chapter 6 as a terminating system.

7.3 Modeling Approach

In *Figure 1.2* of Chapter 1, we briefly discussed the Arena hierarchical structure. This structure freely allows you to combine the modeling constructs from any level into a single simulation model. In Chapters 3 and 4, we were able to develop our models using only the constructs found in the Basic Process panel (yes, we planned it that way), although we did require the use of several data constructs found in the Advanced Process panel for our failure and special statistics.

The general modeling approach that we recommend when creating your models is to stay at the highest level possible for as long as you can. However, as soon as you find that these high-level constructs don't allow you to capture the necessary detail, we suggest that you drop down to the next level for some parts of your model rather than sacrifice the accuracy of the simulation model (of course, there are elements of judgment in this kind of decision). You can mix modeling constructs from different levels and panels in the same model. As you become more familiar with the various panels (and modeling levels), you should find that you'll do this naturally. Before we proceed, let's briefly discuss the available panels.

The Basic Process panel provides the highest level of modeling. It's designed to allow you to create high-level models of most systems quickly and easily. Using a combination of the **Create**, **Process**, **Decide**, **Assign**, **Record**, **Batch**, **Separate**, and **Dispose** modules allows a great deal of flexibility. In fact, if you look around in these modules, you'll find many additional features we haven't yet discussed. In many cases, the use of these modules alone will provide all the detail required for a simulation project. These modules provide common functions required by almost all models, so it's likely that you'll use them regardless of your intended level of detail.

The Advanced Process panel augments the Basic Process panel by providing additional and more detailed modeling capabilities. For example, the sequence of modules Seize - Delay - Release provides basically the same fundamental modeling logic as a **Process** module. The handy feature of the Advanced Process panel modules is that you can put them together in al most any combination required for your model. In fact, many experienced modelers start at this level because they feel that the resulting model is more transparent to the user, who may or may not be the modeler.

The Advanced Transfer panel provides the modeling constructs for material-handling activities (like transporters and conveyors). Similar to the general modeling capabilities provided by the Advanced Process panel, the Advanced Transfer panel modules give you more flexibility in modeling material-handling systems.

The Blocks panel provides an even lower level of modeling capability. In fact, it provides the basic functionality that was used to create all of the modules found in the Basic Process, Advanced Process, and Advanced Transfer panels. In addition, it provides many other special-purpose modeling constructs not available in the higher-level modules. Examples would include "while" loops, combined probabilistic and logic branching, and automatic search features. You might note that several of the modules have the same names as those found in the three higher-level panels. Although the names are the same, the modules are not. You can distinguish between the two by the color and shape.

The difference between the Blocks panel on the one hand and the Basic Process, Advanced Process, and Advanced Transfer panels on the other hand is easy to explain if you've used SIMAN previously, where you define the model and experiment frames separately, even though you may do this all in Arena. The difference is perhaps best illustrated between the **Assign** modules on the two panels. When you use the Basic Process panel **Assign** module, you 're given the option of the type of assignment you want to make. If you make an assignment to a new attribute, Arena will automatically define that new attribute and add it to the drop-down lists for attributes everywhere in your model. One reason for staying at the highest level possible is that the Blocks **Assign** module only allows you to make the assignment-it doesn't define the new attribute. Even with this shortcoming, there are numerous powerful and useful features available only in the Blocks panel.

In addition, the Elements panel hosts the experiment frame modules. This is where, for example, you find the Attributes module to define your new attribute. You’ll rarely need these features since they're combined with the modules of the higher-level panels, but if you need this lowest level for a special modeling feature (you can go to Visual Basic, C, or FORTRAN if you're a real glutton for punishment), it's available via the same Arena interface as everything else.

The Blocks and Elements panels also provide modules designed for modeling continuous systems (as opposed to the discrete process we've been looking at). We'll see these in Chapter ll.

Now let's return to our automotive shop model, which does require features not found in the Basic Process panel modules. In developing our model, we'll use modules from the Basic Process and the Advanced Process panels (we'll use the Blocks and Elements panels to develop an inventory model at the end of this chapter). In some cases, we'll use lower-level constructs because they are required; in other cases, we’ll use them just to illustrate their capabilities. When you model with lower-level constructs, you need to approach the model development in a slightly different fashion. With the higher-level constructs, we tend to group activities and then use the appropriate modules. With the lower-level constructs, we need to concentrate on the actual activities. In a sense, your model becomes a detailed flowchart of these activities. Unfortunately, until you're familiar with the logic in the available modules, it's difficult to develop that flowchart.

7.4 Building the Model

At this point, let's divide our model into sections and go directly to their development where we can simultaneously show you the capabilities available. The seven sections, in the order in which they'll be presented, are:

Section 7.4.1: Defining the Data,

Section 7.4.2: Submodel Creation,

Section 7.4.3: Generate Appointment Calls,

Section 7.4.4: Make Appointment,

Section 7.4.5: Service Activity,

Section 7.4.6: Update Performance Variables, and

Section 7.4.7: Control Logic.

The data section will consist of data modules that we'll need for the model, the next section will show how to create submodels, and the remaining five sections will each be developed as submodels. Your final Top-Level Model window will look something like *Figure 7.1.* Since we'll animate our model later, we'll delete all animation objects that are included with the modules we place.

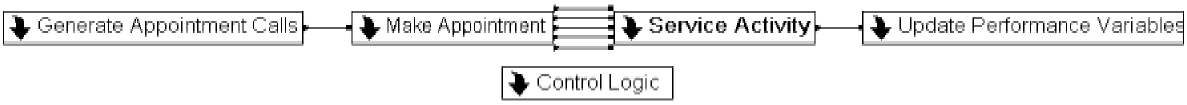


Figure 7.1. Model 7-1: Top-Level Model View of the Automotive Shop Model

As you read through the next seven sections on building the model, you might get the impression that this is exactly how we developed the model. Ideally one would like to be able to plan the construction of the model so it's that easy. In reality, though, you often end up going back to a previously developed section and adding, deleting, or changing modules or data. So as you start to develop more complex models, don't assume that you'll always get it right the first time (we certainly didn't).

7.4.1 Defining the Data

Let's start with the *Run>Setup* dialog box. Under the Project Parameters tab, you'll need to enter the Project Title. Under the Replication Parameters tab, we've somewhat arbitrarily requested ten replications, each of 20 days. Since we would like our report units to be in hours, we've also selected Hours for the base time units.

Since the real system starts empty, except for possible vehicles held overnight and appointments for the next three days, we'll specify a zero length Warm-up Period. Because we've requested multiple replications, we need to tell Arena what to do between replications. There are four possible options.

Option 1: Initialize System (yes), Initialize Statistics (yes)

> This will result in ten statistically independent and identical replications and reports, each starting with an empty system at time O and each running for 240 hours. The random-number generator (see Section 12.1) just keeps going between replications, making them independent and identically distributed (IID). Possible held-over vehicles carried to the next replication are lost.

Option 2: Initialize System (yes), Initialize Statistics (no)

> This will result in ten independent replications, each starting with an empty system at time O and each running for 240 hours, with the reports being cumulative. Thus, Report 2 would include the statistics for the first two replications, Report 3 would contain the statistics for the first three replications, and so on. The random-number generator behaves as in Option 1.

Option 3: Initialize System (no), Initialize Statistics (yes)

> This will result in ten runs, the first starting at time O, the second at time 240, the third at 480, etc. Since the system is not initialized between replications, the time continues to advance, and any held-over vehicles will be carried over to the next replication. The reports will contain the statistics for only a single replication, rather than being cumulative.

Option 4: Initialize System (no), Initialize Statistics (no)

> This will result in ten runs, the first starting at time O, the second at time 240, the third at 480, etc. Since the system is not initialized between replications, the time continues to advance, and any held-over vehicles will be carried to the next replication. The reports will

be cumulative. The tenth report will be the same as if we'd made a single replication of length 2400 hours.

Ideally, we don't want a lot of vehicles to be he ld overnight, and we'd like our replications to be independent. This would suggest either Option 1 or 2. We'll select Option 1 (more on this in Chapter 6).

We have a total of three resources (the three repair and maintenance bays). All three follow a schedule (discussed in Section 4.2). We could define a separate schedule for each resource; however, since they all follow the same schedule, you could simply reuse a single schedule (though doing so would make your modeless flexible in terms of possible future changes). In developing this schedule, we used the Graphical Schedule Editor, set the number of time slots to 12, and the maximum capacity on the y-axis of the graph to two (we could have made it one) in the Options dialog box. The Schedule is shown in *Figure 7.2*.

You might note that our schedule calls for the bays to be available 11 hours per day, the normal eight-hour day, plus the three hours of possible overtime. The resources are not available the first hour of each day. We'll use that first hour to schedule future appointments and allow the customers for the current day to arrive.

The next step is to define our resources. We selected the Preempt option for the Schedule type, because it is possible that a vehicle may still be in service after the normal eight-hour day and the allowed three-hour overtime. In this case, we want to stop work on the vehicle and pick up on the following day where we left off. The final spreadsheet view is shown in *Figure 7.3*.

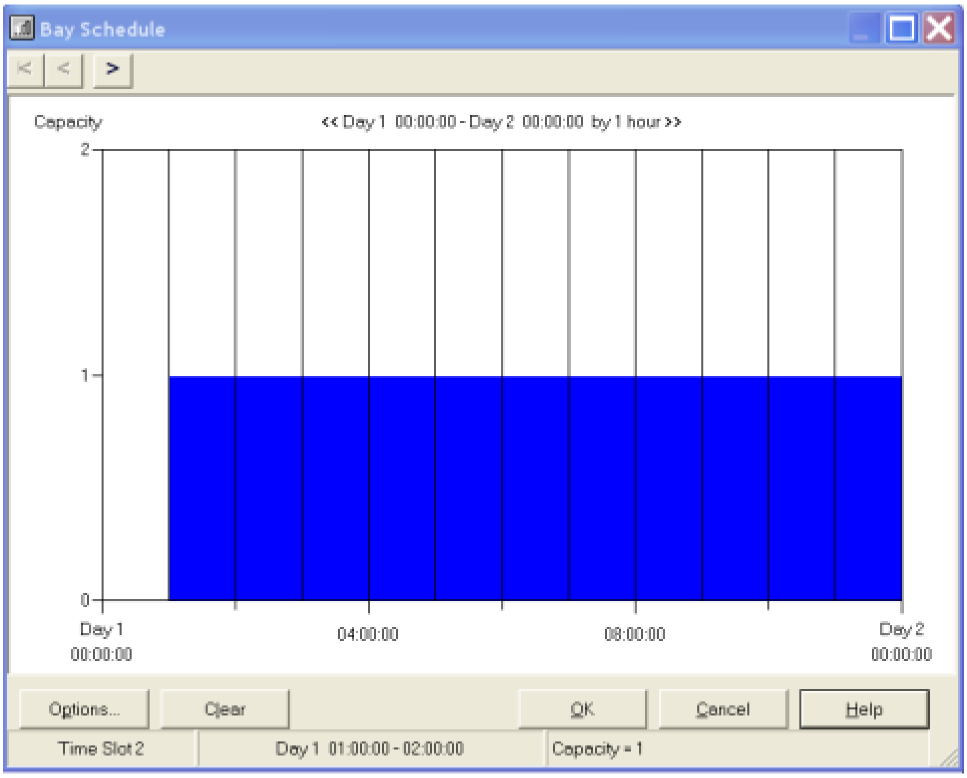
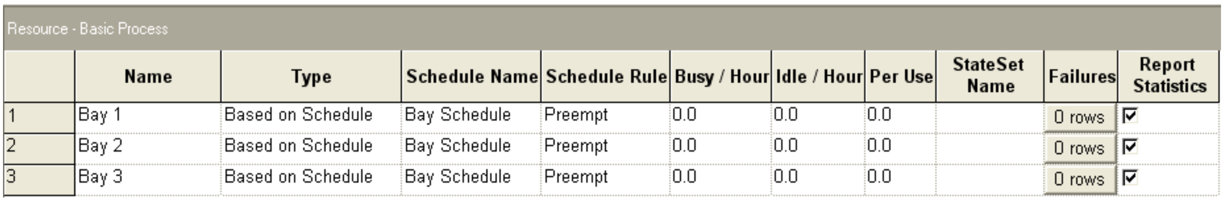


Figure 7.2. The Graphical Schedule Editor: The Service Bay Schedule

Having defined our resources, we can now define our set using the Set module found in the Basic Process panel.

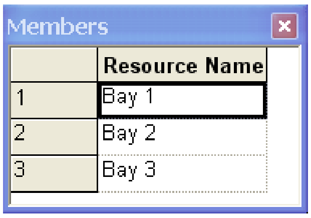
The Set module allows us to form sets of resources, counters, tallies, entity types, and entity pictures. You can also form sets of any other Arena objects by using the Advanced Set data module found in the Advanced Process panel. Form a set by clicking on the Set module and then double-clicking in the spreadsheet view to add a new entry. We'll walk you through the details for creating this set. Enter Bays in the Name cell, and select Resource from the drop-down list in the Type cell. To define the members of the set, click on "0 Rows" in the Members column to open a spreadsheet for listing the members, and then double-click to open a new blank row. Since we have already defined our resources, you can use the drop-down option to select the Resource Name for each member. The completed member spreadsheet view for the Bays set is shown in *Figure 7.4*.



Resource - Basic Process

	Name	Type	Schedule Name	Schedule Rule	Busy / Hour	Idle / Hour	Per Use	StateSet Name	Failures	Report Statistics
1	Bay 1	Based on Schedule	Bay Schedule	Preempt	0.0	0.0	0.0		0 rows	☑
2	Bay 2	Based on Schedule	Bay Schedule	Preempt	0.0	0.0	0.0		0 rows	☑
3	Bay 3	Based on Schedule	Bay Schedule	Preempt	0.0	0.0	0.0		0 rows	☑

Figure 7.3. The Resources Spreadsheet Window

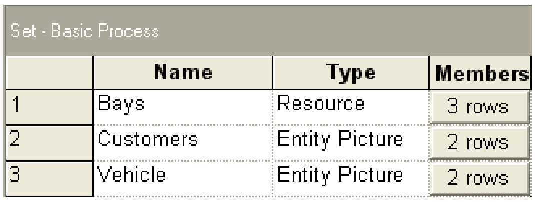


Members

	Resource Name
1	Bay 1
2	Bay 2
3	Bay 3

Figure 7.4. The Members Spreadsheet Window for the Bays Set

We also defined two sets for entity pictures, Customers and Vehicles. Both have two members, which represent the customers who wait and the customers who leave their vehicles. The completed Set data module is shown in *Figure 7.5*.

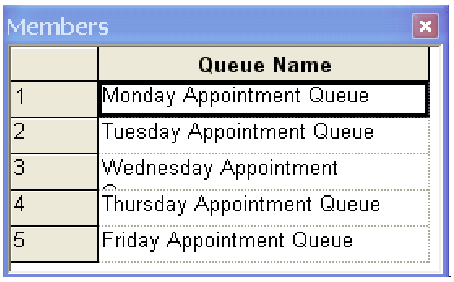


Set - Basic Process

	Name	Type	Members
1	Bays	Resource	3 rows
2	Customers	Entity Picture	2 rows
3	Vehicle	Entity Picture	2 rows

Figure 7.5. The Set Spreadsheet Window

We also need to create a queue set to hold our future appointments. First we need to define our five queues (Monday Appointment Queue through Friday Appointment Queue) using the Queue module from the Basic Process panel. Then we'll add the queue set using the Advanced Set module from the Advanced Process panel. Enter Appointment Queues in the Name cell, and select Queue from the drop-down list in the Type cell. Then enter the five queue member names in the spreadsheet, just like we did for the resource sets. The five members of this set are shown in *Figure 7.6*.



Members

	Queue Name
1	Monday Appointment Queue
2	Tuesday Appointment Queue
3	Wednesday Appointment
4	Thursday Appointment Queue
5	Friday Appointment Queue

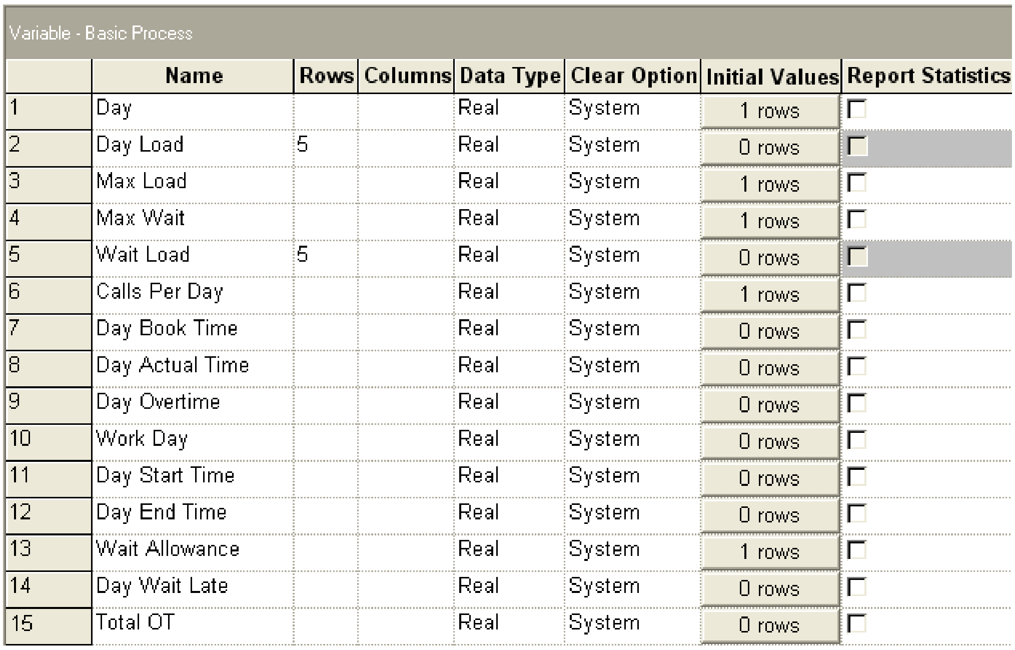
Figure 7.6. The Members Window from the Advanced Set Spreadsheet Module

For the next step, we'll use the Variable data module from the Basic Process panel to define our variables. We probably should point out that it's not always necessary to use this module to define variables. If you define a new variable when it's needed (say, in an **Assign** module), Arena will automatically enter the information for that new variable into the Variable data module. However, if you have variables that are defined as arrays or that you want to have non-zero initial values, this module should be used to set the required array sizes or initial values.

We'll define a total of 15 variables. The first six variables are used to control our model: a variable that keeps track of the current day (Day) which we'll initialize to 4 (more on this later); an array variable that we'll use to keep track of the current number of Book Time appointment hours assigned to that day (Day Load); the maximum number of Book Time hours we can assign to any day (Max Load) which we'll initialize to 24; the maximum number of customers who will wait for their service to be completed that we can schedule for any day (MaxWait), initialized to 5; an array variable that we'll use to keep track of the current number of wait jobs assigned to that day (Wait Load); and the average number of customers who call each day to request an appointment (Calls Per Day), initialized to 25.

The next three variables (and the last two variables) will be used to accumulate values that will be used to calculate outputs at the end of the simulation run: the current total number of Book Time hours processed to date (Day Book Time); the current total number of actual service hours processed to date (Day Actual Time); the current total number of overtime hours processed to date (Day Overtime); the current total number of days that vehicles were held overnight (Day OT); and the current number of wait jobs that have been late (Day Wait Late).

The four variables in between will be used in our control logic: the number of the current work day in the simulation (Work Day); the work starting time for the current day (Day Start Time); the normal work ending time for the current day (Day End Time), not including overtime; and the amount of allowance used to determine if a wait job is late (Wait Allowance), initialized to 1. The final spreadsheet view is shown in *Figure 7.7*.



Variable - Basic Process

	Name	Rows	Columns	Data Type	Clear Option	Initial Values	Report Statistics
1	Day			Real	System	1 rows	□
2	Day Load	5		Real	System	0 rows	□
3	Max Load			Real	System	1 rows	□
4	Max Wait			Real	System	1 rows	□
5	Wait Load	5		Real	System	0 rows	□
6	Calls Per Day			Real	System	1 rows	□
7	Day Book Time			Real	System	0 rows	□
8	Day Actual Time			Real	System	0 rows	□
9	Day Overtime			Real	System	0 rows	□
10	Work Day			Real	System	0 rows	□
11	Day Start Time			Real	System	0 rows	□
12	Day End Time			Real	System	0 rows	□
13	Wait Allowance			Real	System	1 rows	□
14	Day Wait Late			Real	System	0 rows	□
15	Total OT			Real	System	0 rows	□

Figure 7.7. The Variable Spreadsheet Window

We've also used the Expression data module in the Advanced Process panel to enter several of the expressions that we need for our model. We've chosen to define an expression for the

Book Time for each customer call, an expression to determine whether a customer wants to wait for his or her vehicle, and an expression for the actual service time. We could just as easily have entered these distributions directly into the model as needed.

Let's start with the first expressions discussed in the preceding paragraph. Open the Expression module and enter *Book Time Expression* in the Name cell for the first expression. Then enter AINT (44+90*BETA(2,3))/60 in the Expression Values cell. You can simply type this in or use the Expression Builder to make these entries. Since the Book Time used by most automotive repair shops is always expressed in integer minutes, we've used the Arena expression AINT to truncate the returned value to the next lower integer (AINT is really not poor grammar, but is one of Arena's many built-in math functions, a complete list of which can be found under the Help topic "Mathematical Expressions"). Note that we have also divided the resulting value by 60 to convert the time to hours.

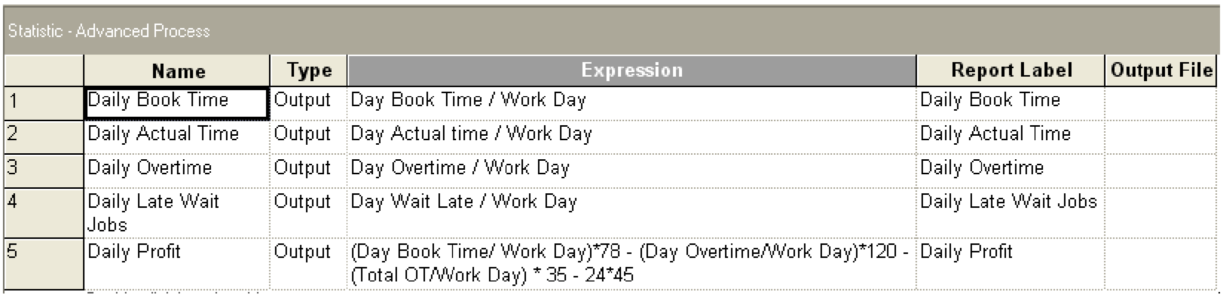
The second expression (*Wait Priority Expression*) is used to determine if the customer wants to wait for the service to be completed. The expression, DISC 0.20, 1, 1.0, 2), will return a 1, with probability of 0.20 and a 2 with probability of 0.80. In this case a 1 implies that the customer will wait for the service.

The last set of expressions is an equine of a dissimilar hue in that it requires the value generated by our first expression. When we generate the Book Time, we'll store it in an attribute of the entity called *Book Time*. We can then use the expression *Actual Service Time*, which will use the entity attribute value, to generate the actual service time. The expression we entered is GAMM (*Book Time*/1.05, 1.05).

Our final data items require the Statistic module from the Advanced Process panel. All of our entries are of the Output Type, which means that these values will only be calculated at the end of each replication. The first three entries calculate the daily average, in hours, of the Book Time processed, actual service time, and the overtime hours. We used the variables previously defined to calculate these values. The fourth expression is the average number of late wait jobs, per day.

The last statistic, composed of three terms, calculates the average daily profit. The first term, (*Day Book Time*/*Work Day*)*78, represents the income from completed jobs based on the Book Time charge of $78 per hour. The second term, (*Day Overtime*/*Work Day*)*120, represents the penalty incurred for overtime work. The first part of the last term, (*Total OT*/*Work Day*)*35, is the penalty cost due to the need to provide loaner vehicles to customers whose jobs were not completed on the scheduled day. The second part, 24*45, is the cost of capital and labor.

These expressions can be entered directly or you can use the Expression Builder. The final spreadsheet view is shown in *Figure 7.8*.



Statistic - Advanced Process

	Name	Type	Expression	Report Label	Output File
1	Daily Book Time	Output	Day Book Time / Work Day	Daily Book Time	
2	Daily Actual Time	Output	Day Actual time / Work Day	Daily Actual Time	
3	Daily Overtime	Output	Day Overtime / Work Day	Daily Overtime	
4	Daily Late Wait Jobs	Output	Day Wait Late / Work Day	Daily Late Wait Jobs	
5	Daily Profit	Output	(Day Book Time/ Work Day)*78 - (Day Overtime/Work Day)*120 - (Total OT/Work Day) * 35 - 24*45	Daily Profit	

Figure 7.8. The Statistic Spreadsheet Window

Having completed our data requirements, we're now ready to move on to our logic development.

7.4.2 Submodel Creation

To create a submodel, select *Object>Submodel>Add Submodel*. Your mouse pointer will change to cross hairs that can be moved in the model window to where you want to place the submodel. Click to place the newly created submodel as shown in *Figure 7.9*. The submodel object will have the name Submodel followed by a number that will be incremented as you add new submodels.

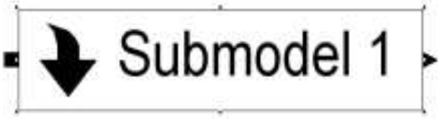
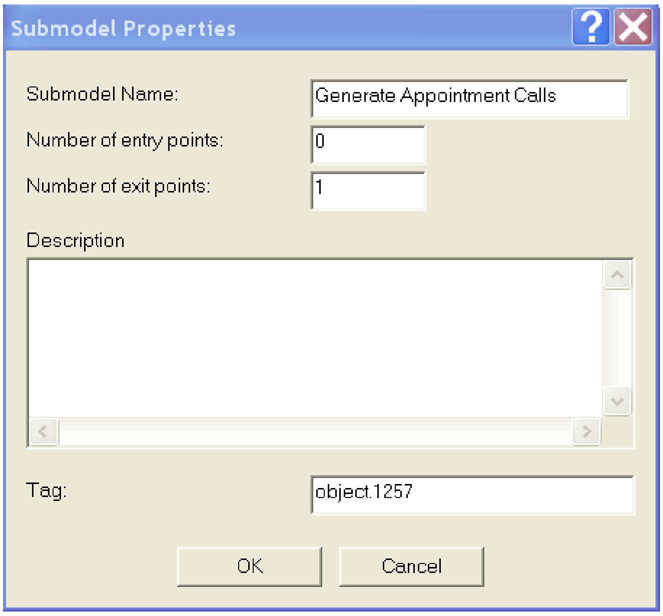


Figure 7.9. The Newly Created Submodel

To define your own name for the submodel object, right-click on it and select the *Properties* option, which will open the Submodel Properties window, as in *Display 7.1*. We'll call our first submodel *Generate Appointment Calls*. A new submodel has by default one entry point and one exit point. For the first submodel, we'll change the number of entry points to zero since this particular submodel will be used to generate calls, using a **Create** module, for the rest of our model. You can also enter a description, although we have chosen not to as our model is fairly simple and our names are descriptive enough.



Submodel Name: Generate Appointment Calls

Number of entry points: 0

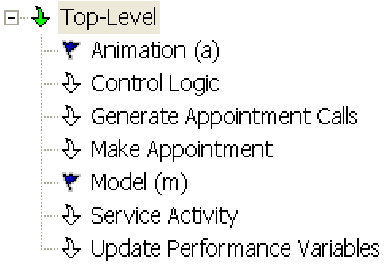
Number of exit points: 1

Tag: object.1257

Submodel Name	Generate Appointment Calls
Number of entry points	0
Number of exit points	1

Display 7.1. The Submodel Properties Window

You can repeat this process for the remaining four submodels with your final view looking something like *Figure 7.1*. If you do this, you might want to pay attention to the number of entry and exit points for each submodel in that figure. Once you have placed your submodels and given them the correct properties, you can then start to build your logic. To open a submodel and add or edit logic, just double-click it or use the Project Bar's Navigate panel to jump between submodels and the Top-Level Model, as depicted in *Figure 7.10*. To close a submodel window, you can use the Navigate section or point at a blank spot in the submodel window, right-click, and select *Close Submodel*. We're now ready to build the actual model.



- Top-Level
  - Animation (a)
  - Control Logic
  - Generate Appointment Calls
  - Make Appointment
  - Model (m)
  - Service Activity
  - Update Performance Variables

Figure 7.10. The Navigate Panel

7.4.3 Generate Appointment Calls

This submodel will generate the appointment calls each day and forward them to the next submodel where they will attempt to make an appointment. Except for the first day, the logic is fairly straightforward. We'll generate a single entity (one every 12 hours), determine the number of calls for that day, increment the Day variable, create enough duplicates so that the number of departing entities is equal to the number of calls for that day, and finally assign the necessary attributes to each departing entity.

Recall that a customer is allowed only to make an appointment for one of the next three working days. Since we decided that there would be no warm up for our simulation, we need to make sure that we have appointments at the start of the simulation of the first day. We'll do this by checking the time to see if we are at day 1. If so, we'll create three duplicate entities that will attempt to make appointments. Since the Day variable was initialized to 4 (Thursday), it will be incremented to 5 (Friday), and the equivalent of three days of customer calls will attempt to make appointments (limited to appointments on Monday, Tuesday, or Wednesday). We'll then reset the Day variable to Monday and allow the original entity to generate appointment calls.

The modules for generating the appointment calls and assigning variables and attributes are shown in *Figure 7.11*. We've already used most of these module types. The two new modules, the **Delay** and **Separate** modules, are from the Advanced Process and Basic Process panel, respectively.

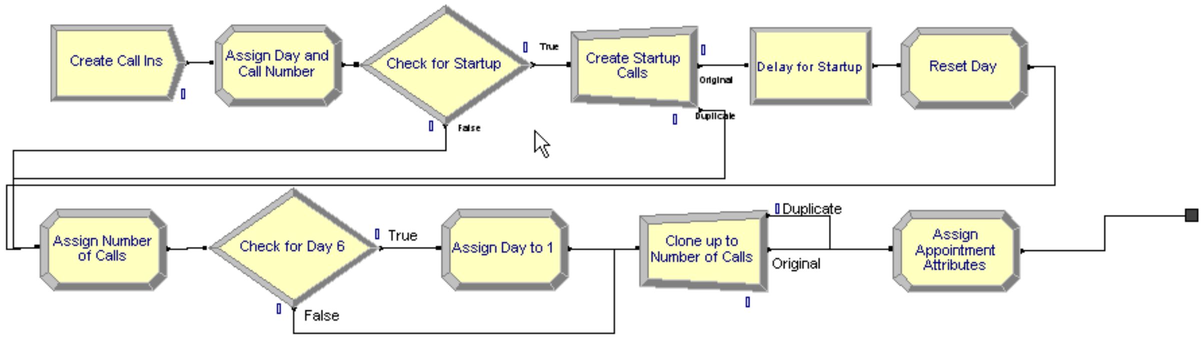


Figure 7.11. Generate Appointment Calls

The logic for this submodel can be summarized as follows:

```
Create one entity each day
Increment day and set call number to one
If start of replication
   Create 3 duplicate entities
```

```
        Assign number of calls (N)
        If day is 6
            Set day to 1
        Create N-1 duplicate entities
        Assign appointment attributes
        Delay original entity
        Assign number of calls (N)
        If day is 6
            Set day to 1
        Create N-1 duplicate entities
        Assign appointment attributes
    Else (not start of replication)
        Assign number of calls (N)
        If day is 6
            Set day to 1
        Create N-1 duplicate entities
        Assign appointment attributes
```

Create is the first module, described in *Display 7.2*, which is used to create an entity at the beginning of each day. We entered a name, an Entity Type, a constant 12 hours for the time between arrivals-the length of each day, and a First Creation at time 0.01 hours. We accepted the defaults on the remaining data. We hope the reason for the slight delay on the first creation will be come apparent after we present our Control Logic sub model. If not, maybe you'll have forgotten about this comment by the time we get there!

Name	Create Call Ins
Entity Type	Customer Calls
Time Between Arrivals	
Type	Constant
Value	12
Units	hours
First Creation	0.01

Display 7.2. The **Create** module

In the first **Assign** module, we increment the day variable and set the call number to one, as shown in *Display 7.3*.

Name	Assign Day and Call Number
Assignments	
Type	Variable
Variable Name	Day
New Value	Day+1
Assignments	
Type	***Attribute***
Variable Name	***Call Number***
New Value	***1***

Display 7.3. The First **Assign** module

The entity is then sent to the *Check for Startup* **Decide** module that provides entity branching based on chance or on a condition. Branch destinations are defined by graphical connections. An arriving entity examines each of the user-defined branch options and sends itself to the destination of the first branch whose condition is satisfied. If all specified conditions are false, the entity will automatically exit from the *False* exit at the bottom of the module.

In our **Decide** module, described in *Display 7.4*, we've selected the 2-*Way by Condition* option for Type to check whether we are at the start of the replication; that is, the current simulation time is less than 1. If *True*, we send the entity to the following **Separate** module. Otherwise, we send it to the **Assign** module where we assign the number of calls.

Name	Check for Startup
Type	2-Way by Condition
If	Expression
Value	TNOW < 1

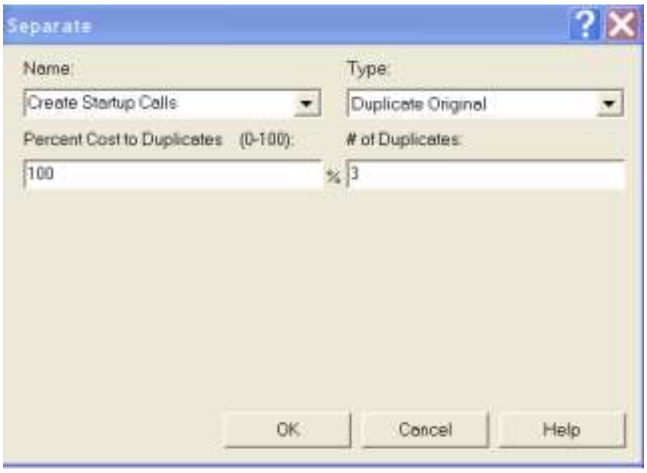
Display 7.4. The **Decide** module

Table 7.1.

The Decide Module Condition Notation

Description	Syntax	Options	Description	Syntax	Options
And	.AND.		Or	.OR.	
Greater than	.GT.	>	Greater than or equal to	.GE.	>=
Less than	.LT.	<	Less than or equal to	.LE.	<=
Equal to	.EQ.	==	Not equal to	.NE.	<>

A condition can be any valid expression but typically contains some type of comparison. Such conditions may include any of the notations shown in *Table 7.1*. Logical expressions can also be used.



Separate

Name: Create Startup Calls

Type: Duplicate Original

Percent Cost to Duplicates (0-100): 100 %

# of Duplicates: 3

Name	Create Startup Calls
Type	Duplicate Original
# of Duplicates	3

Display 7.5. The First **Separate** module

If we are at the start up, the entity is sent to the following **Separate** module (from the Basic Process panel), shown in *Display 7.5*. The **Separate** module allows us to make duplicates of the arriving entity. It also allows us to split an existing batch of entities into its original entities (see Section 9.4 for more on this topic). In our model, we want to create three duplicate entities that will allow us to generate appointments for the first three days of the simulation. To accomplish this, we selected the *Duplicate Original* option for the Type and entered a value of 3 for # of Duplicates. Note that there are two exit points for this module. The original entity will exit from the upper point and the three duplicate entities will exit from the lower point.

For now, let's follow the three duplicate entities that are sent to the following **Assign** module where we use a Poisson distribution with a mean taken from the previously defined variable *Calls per Day* to generate the *Number of Calls*, as shown in *Display 7.6*.

Name	Assign Number of Calls
Assignments	
Type	Attribute
Variable Name	Number of Calls
New Value	POIS (Calls Per Day)

Display 7.6. The Second **Assign** module

The entity is then forwarded to the second **Decide** module, where we'll check whether the current value of the variable Day exceeds 5, as shown in *Display 7.7*. This variable holds the numeric value of the current workday of the week, with a range from 1 to 6. We incremented this variable in our first **Assign** module, and now we need to check that it has not reached a value of 6.

Name	Check for Day 6
Type	2-Way by Condition
If	Variable
Named	Day
Is	>
Value	***5***

Display 7.7. The Second **Decide** module

If the variable has a current value of 6, the entity is directed to the following **Assign** module where it is reset to 1. The entity is then sent to a second **Separate** module. If the condition on the previous **Decide** module was *False*, the entity bypasses the following **Assign** module and is also sent to the second **Separate** module.

In the initial **Create** module, we created a single entity that represented all the calls for the day. Although we duplicate that entity at the start of the simulation run, the current entity still represents all the calls for one day. We use the second **Separate** module to duplicate the remaining calls. We need to duplicate only the number that we require, minus one, as the original entity will be treated as one of the calls, as shown in *Display 7.8*.

Name	Clone up to Number of Calls
Type	Duplicate Original
# of Duplicates	Number of Calls – 1

Display 7.8. The Second **Separate** module

These entities are then sent to the last **Assign** module where a series of attributes are assigned. The *Day Inc* attribute is the first day (out of the three possible days) that the customer would like to make an appointment. The *Priority* attribute is assigned a value of 1 for a wait customer or 2 for a customer who will leave his or her vehicle (using the *Wait Priority Expression* developed earlier). The expected service time is assigned to the attribute Book Time. The Day Inc attribute is added to the current Day value and assigned to the attribute Try Day, which is the first day of the week on which the customer would like to make an appointment (we'll check later for a value greater than 5). Finally, an entity picture is assigned from our previously defined picture set, and the entity is sent to the exit point of the submodel.

7.4.4 Make Appointment

As you learn more about the capabilities of Arena and develop more complex models, you'll find it necessary to plan your logic in advance of building the model. Otherwise, you may find yourself constantly moving modules around or deleting modules that were erroneously selected. Whatever method you adopt for planning your models will normally come about naturally, but you might want to give it some thought before your models become really complicated-and they will! A lot of modelers use pencil (a pen for the overconfident) and paper to sketch out their logic using familiar terminology. Others use a more formal approach and create a logic diagram of sorts using standard flowcharting symbols. Still another approach is to develop a sequential list of activities that need to occur (like we did in the previous section). These types of approaches will help you formalize the modeling steps and reduce the number of errors and model changes. As you become more proficient, you may even progress to the point where You'll start by laying out your basic logic using Arena modules (after All, they're similar to flowchart elements). However, even then we recommend that you lay out your complete logic before you start filling in the details.

We started to develop a list of logic, but decided quickly that it is just too complicated to present in a nice simple form. This is not uncommon in complicated models, so we decided to walk you through the logic.

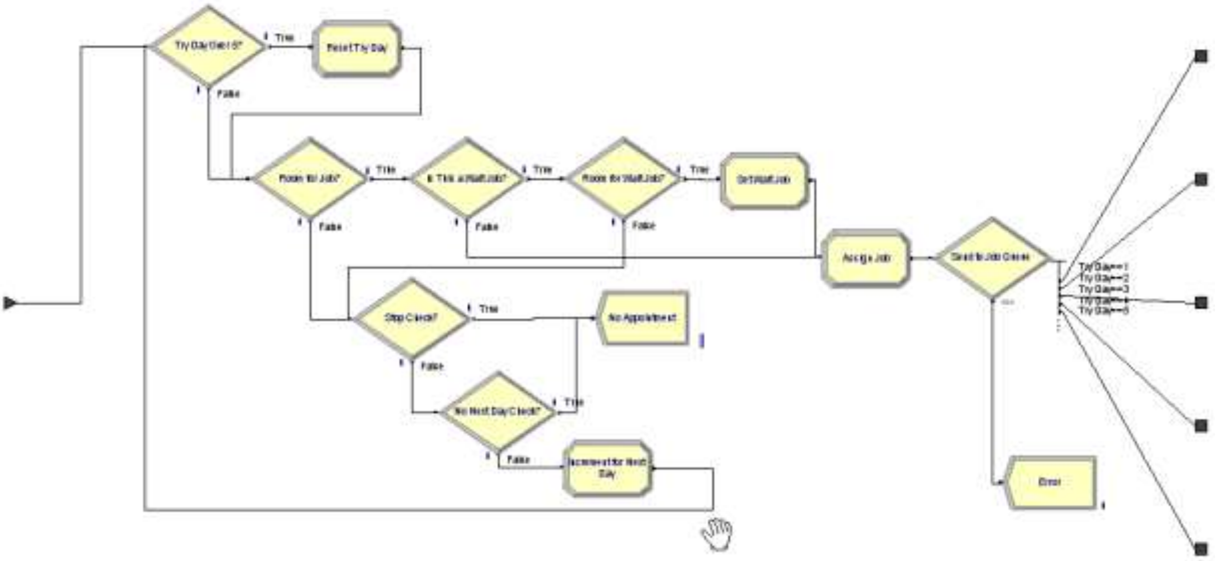


Figure 7.12. Make-Appointment Logic

You might want to refer to *Figure 7.12* as you read through this description. First, we check to make sure that the day on which the customer is trying to make the appointment is valid (with a value from 1 to 5). If it's not, we reset it to a valid day. Next we check the current appointment queue to see if there is room for our customer (based on the Book Time). If there is room, we check to see if it's a wait job. If it's a wait job, we check to see if there is space for a wait job. If so, we increment the wait jobs for that day and the amount of Book Time scheduled for that day. The entity is directed to one of the submodel exits where it will be placed in its appointment queue. If any of these conditions is false, the entity is sent to the section of logic where we try the next day. If the day for the requested appointment is more than three days out, we dispose of the customer call; remember that we only allow appointments for the next three days. If the requested day is already at its appointment limit, we check to see whether the customer wants to try for the next day (a 90% chance). If not, we dispose of the call. If the customer wants to try for the next day, we increment *Try Day* and send the entity back to the start of the logic to try again. Before we look at the detailed logic, you might take note that this submodel has one entry point and five exit points.

Now let's start over and walk through the logic detai1. Recall that we generated a value that represented the first day on which the customer would like to schedule his or her appointment and assigned that value to the attribute *Try Day*. At the first **Decide** module, we check to see if the value of *Try Day exceeds* 5. Since the current Day has a value from 1 to 5 and the increment we added could have a value from 1 to 3, *Try Day* could have a value from 2 to 8. If it has a value from 6 to 8 (these values represent Monday through Wednesday of the next week), we need to subtract a value of 5. We do this in the following **Assign** module.

Now that we have a valid *Try Day* value, we need to check whether there is room for the requested appointment. The Book Time attribute value represents the expected service time for this customer. Previously we defined an arrayed variable, *Day Load*, which is the total amount of Book Time hours currently scheduled for the requested day. We also defined a variable, *Max Load*, which is the maximum number of Book Time hours that we are allowed to schedule for any workday. Now we can use the following expression to see if there is room for this customer

```
    Book Time + Day Load (Try Day) <= Max Load.
```

Name	Send to Job Queue
Type	N-Way by Condition
If	Attribute
Named	Try Day
Is	==
Value	1
If	Attribute
Named	Try Day
Is	==
Value	2
If	Attribute
Named	Try Day
Is	==
Value	3
If	Attribute
Named	Try Day
Is	==
Value	4
If	Attribute
Named	Try Day
Is	==
Value	5

Display 7.9. The N-Way **Decide** module

If this expression evaluates as *True*, there is room and we send the entity to the next **Decide** module where we check to see if this customer wants to wait for service. If the value of the attribute Priority is equal to 1, that represents a wait customer, and we send the entity to the next **Decide** module where we check the number of wait customers currently scheduled for that day. We previously define an arrayed variable, *Wait Load*, which is the number of wait customers who currently have appointments for that day. We also defined a variable, *Max*

Wait, which is the maximum number of wait customers we can schedule on any day. We'll use the following express ion to see if we can schedule the customer appointment

```
    Wait Load (Try Day) < Max Wait.
```

If there is room, the number of wait customers for that day is incremented in the following **Assign** module.

We can schedule an appointment under two conditions: there is time available and it's not a wait customer; or there is time available, we have a wait customer, and there is room for another wait customer. If either of the se conditions is *True*, we send the entity to the *Assign Job* **Assign** module where we add the entity Book Time to our *Day Load* variable.

That entity is then sent to the **Decide** module at the far right where we will send it to the proper exit point based on the appointment day, as shown in *Display 7.9*. In this case, we select the *N-Way by Condition* Type and enter five conditions, one to check for each valid value of the attribute *Try Day*. The entity is then directed to the proper branch, shown at the lower right of the **Decide** module, where it is sent to one of five possible exit points. We'll pick up these entities in the next section where we'll discuss the next submodel.

You should note that the entity would exit from one of these points only if one of the expressions evaluates to True. There is still a possibility that none of the expressions will evaluate as *True*. Of course, this would imply that we made an error in our model. So we added a **Dispose** module labeled *Error* where we send any entities for which all the conditions evaluate to *False*. Luckily, we never had an entity use this module; of course, we would say that even if it were not true!

Name	No Next Day Check?
Type	2-Way by Chance
Percent True	10

Display 7.10. The Second **Decide** module

If the entity is unable to make an appointment for the Try Day, the entity is directed to the *Stop Check?* **Decide** module where we check to see if the value of *Try Day* is equal to 3, which implies it's not possible to make an appointment. In this case, the entity is sent to the *No Appointment* **Dispose** module where the entity is destroyed (maybe to call in the next day). If the *Try Day* value is 1 or 2, the entity is sent to the *No Next Day Check?* **Decide** module, as shown in *Display 7.10*. Here we selected the *2-Way by Chance* option and entered a value of 10 for the Percent. When an entity enters this module, Arena automatically generates a random number with range from 0 to 1. In this case, if the value is less than or equal to 0.10 (1 0% chance), the entity will exit the module on the right, labeled *True*. Otherwise, it will exit at the bottom, labeled *False*.

If the condition is *True*, the entity is sent to the same **Dispose** module connected to the previous **Decide** module. If the condition is False (90% of the time), the entity is sent to the following **Assign** module where the *Day Inc* and *Try Day* attributes are incremented. From here the entity is sent back to the very first **Decide** module where the entity will attempt to schedule an appointment on the next available day.

7.4.5 Service Activity

The logic for the Service Activity submodel is fairly simple (*Figure 7.13*), so we won't bother to describe the logic ahead of time, we'll just go at it! First, you should note there are five entry points. You might have noticed in *Figure 7.1* that the five exit points from the previous submodel are connected in the same sequence to this submodel. The top point represents a value of 1 for a Monday appointment. The bottom point represents a value of 5 for a Friday appointment. Each of these entry points is connected to its own Hold module, labeled *Monday List* through *Friday List*.

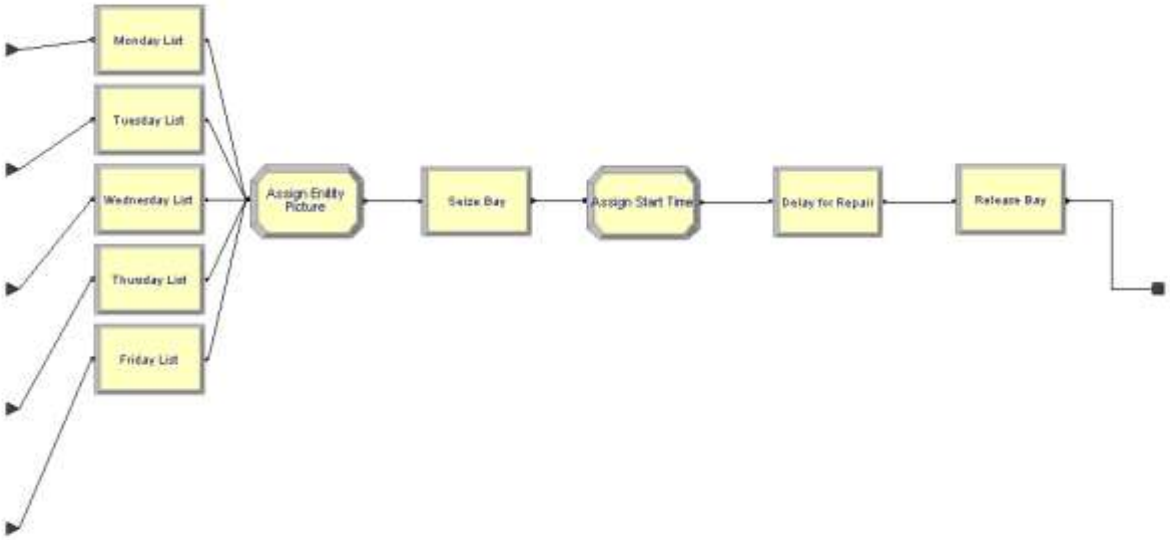
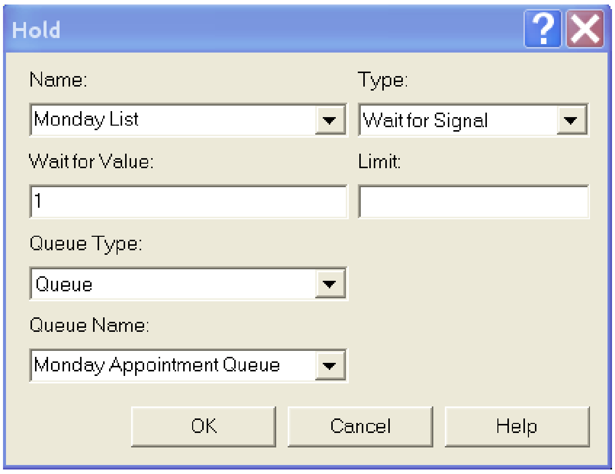


Figure 7.13. Service Activity Logic

The Hold module labeled *Monday List* is shown in *Display 7.11*. The Hold module allows you to hold entities in a queue until some system condition is met. In this case, it is a hold until a signal is received with a value of 1. You'll see where we send this signal when we cover the Control Logic submodel. For the *Monday List* module, we selected the *Wait for Signal* option and accepted the default value of 1 for the signal value. We then selected the Queue Name as *Monday Appointment Queue*. You might remember from Section 7.4.1 that we defined five queues and put them into an advanced set. This is the first place in our model that we use the queues.



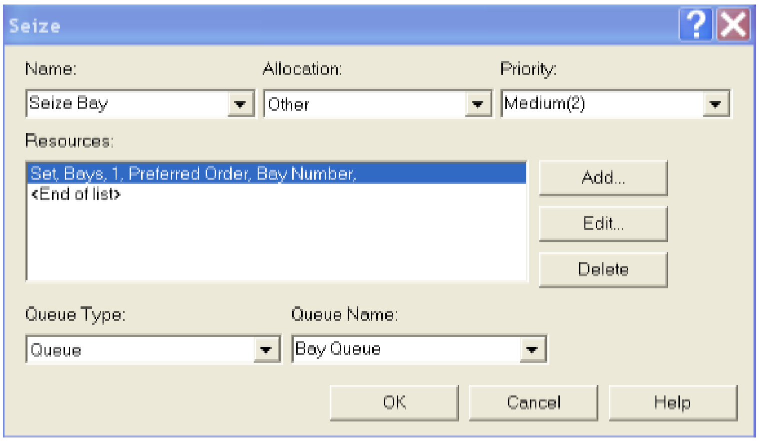
Name: Monday List
Type: Wait for Signal
Wait for Value: 1
Limit:
Queue Type: Queue
Queue Name: Monday Appointment Queue

Name	Monday List
Type	Wait for Signal
Wait for Value	1
Queue Name	Monday Appointment Queue

Display 7.11. The Monday Hold Module

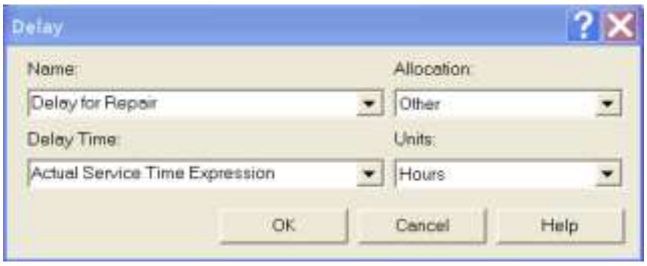
The remaining four Hold modules differ only in their names, the signal value, and the queue name. The signal values for these four modules range from two to five, corresponding to the days of the workweek from Tuesday to Friday. At the start of each Monday workday, a signal of one is sent to release all the entities (customer appointments) to the following module. A signal of two is sent at the start of Tuesday, etc.

The released entities are sent to the following **Assign** module where they are assigned a new picture from our Vehicle picture set and the current simulation time to attribute *Arrive Time*. These entities enter the Seize module (from the Advanced Process panel), as shown in *Display 7.12*. Normally, we would have used a **Process** module, but in this case, we need to separate the seize part of the **Process** module from the delay part (you'll see why shortly). In the Resources section, we have selected the Set option for the Type, entered Bays for the Set name (we defined this set earlier), accepted the *Preferred Order* default for the Selection Rule, and entered *Bay Number* for the Save Attribute. Finally, we entered *Bay Queue* for the Queue Name.



Name	Seize Bay
Resources	
Type	Set
Set Name	Bays
Selection Rule	Preferred Order
Save Attribute	Bay Number
Queue Name	Bay Queue

Display 7.12. The Seize Module



Name	Delay for Repair
Delay Time	Actual Service Time Expression
Units	Hours

Display 7.13. The Delay Module

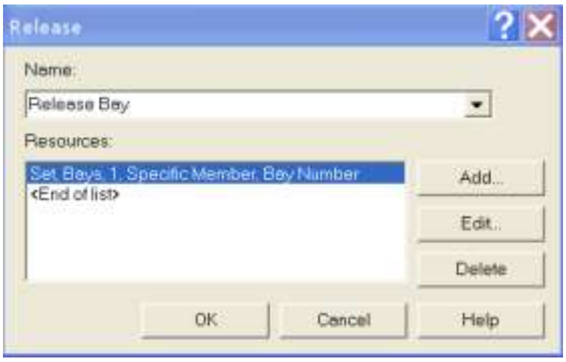
Now we open the Queue data module (the new Bay Queue name is added automatically) and select *Lowest Attribute Value* option for the Type and enter *Priority* for the Attribute Name. The result is that the arriving entities will enter the Bay Queue and be ranked by their value of attribute Priority. This means that the wait customers will be put at the front of the queue and serviced first, before the other customers. Remember that we promised these customers that we would try to get their service completed quickly so their wait would be minimized. We selected the Preferred Order option in the Seize module because we wan ted to seize the bay resources in the order they are listed in the set (Bay 1, Bay 2, and Bay 3). You'll see why we did this when we embellish the model later in the chapter.

When an entity is allocated a resource bay, the entity is sent to the following **Assign** module where we assign the current simulation time to the new attribute *Start Time*. We need this value because we would like to keep statistics on the actual service times, and we'll use this when we update our performance parameters.

The entity then enters the following Delay module from the Advanced Process panel, shown in *Display 7.13*. Here we've entered a Name and the previously defined expression, *Actual Service Time Expression*. We also selected the *Hours* option for the Units.

After the delay has been completed, the entity enters the Release module (from the Advanced Process panel), as shown in *Display 7.14*. In the Resource section, we selected the *Set* option for the Type, entered *Bays* for the Set Name, selected the *Specific Member* option for the Release rule, and entered *Bay Number* for the Set Index. Basically, we are releasing the same resource that this entity was allocated in the previous Seize module. We did this by keeping track of the set index of the allocated resource in our attribute *Bay Number* to make sure we release the correct resource.

These entities are then directed to the submodel exit point. Now might be a good time to point out that the combination of the Seize, Delay, and Release modules result in the same capabilities as a **Process** module. If it were not necessary to keep track of the time the service started we could have used the **Process** module.



Name: Release Bay

Resources:

Set, Bays, 1, Specific Member, Bay Number
<End of list>

Name	Release Bay
Resources	
Type	Set
Set Name	Bays
Release Rule	Specific Member
Set Index	Bay Number

Display 7.14. The Release Module

7.4.6 Update Performance Variables

The entities arriving to this sub model represent customers who have completed their service and are about to depart the system. Back in Section 7.4.1, we defined five variables that we are using to define the performance of our system; *Day Book Time*, *Day Actual Time*, *Day Overtime*, *Day OT*, and *Day Wait Late*. At first glance, this should be a fairly easy task. However, the fact that it is possible that one or more customer vehicles may not have completed service greatly complicates this task. The Arena modules required for this section are shown in *Figure 7.14*.

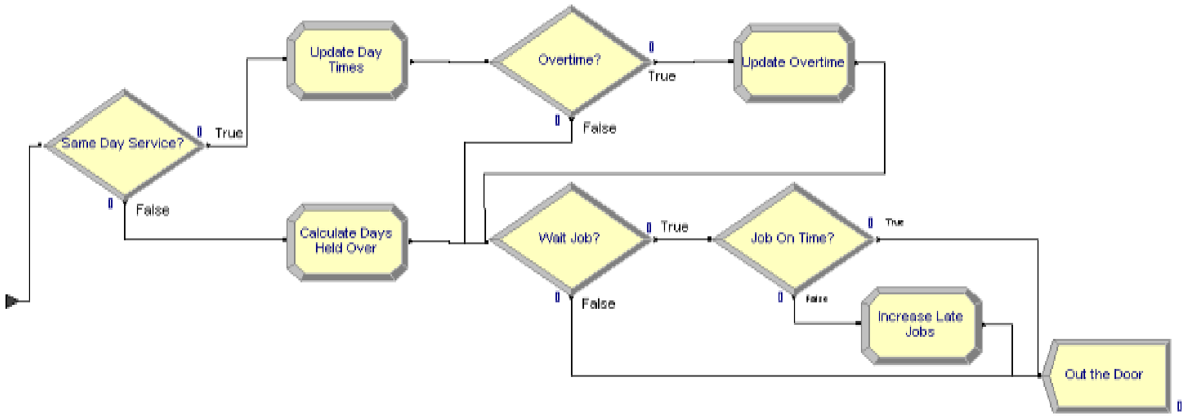


Figure 7.14. Update Performance Logic

The arriving entities are directed to the first **Decide** modu1e to determine whether the customer was held overnight. In our Control Logic submodel (to be discussed in the next section), we'll compute the simulation time that represents the start of the current workday (e.g., 1 for Day 1, 13 for Day 2, and so on) and assign that time to the variable *Day Start Time*. We'll also compute the simulation time of the normal end of the current day (e.g., 9 for Day 1, 21 for Day 2, etc.) and assign that value to the variable *Day End Time*. By comparing the value of the attribute Start Time to the value of the Day Start Time, we can determine whether the vehic1e was he ld overnight. We do this with the expression

```
Start Time >= Day Start Time.
```

The idea is that if the service was started on or after the start of the current day, then it was not held overnight. So, if this expression evaluates to True, we have same-day service. If it's False, then the customer vehic1e was held overnight.

Let's follow an entity where this expression evaluates to *True* a same-day service customer. The entity is sent to the *Update Day Times* **Assign** modu1e where we add the value of the entities attribute *Book Time* to the variable *Day Book Time*. We a1so add the value of the entities actual time (TNOW - *Start Time*) to the variable *Day Actual time*. This takes care of two of the required updates.

In the next **Decide** module, we determine whether overtime was required to complete this service. We do this by comparing the value of the variable *Day End Time* to the current simulation time (the end of service time) in the expression

```
Day End Time < TNOW.
```

If this evaluates to *True*, the service was completed after the normal workday. In this case, we need to update our *Day Overtime* variable. We need to be careful here, as we want to add only that portion of the service that was performed after the end of the normal workday the overtime. We compute this value using the expression

```
TNOW - MX (Start Time, Day End Time).
```

MX is the Arena function that returns the maximum of its arguments. If the service was started before the end of the normal day, then the *Day End Time* will yield the maximum value. If the service was started after the end of the normal day, then the *Start Time* will yield the maximum value.

This entity is then sent to the *Wait Job?* **Decide** module, which is the same place that the entity is sent if there were no overtime. At this module, we check to see whether this is a wait job. If it's not a wait job, the entity exits from the False branch and is sent to the **Dispose** module where it exits the system. If it is a wait job, it is sent to the following *Job On Time?* **Decide** module where we'll check to see if the service was completed by the promised time. We do this with the expression

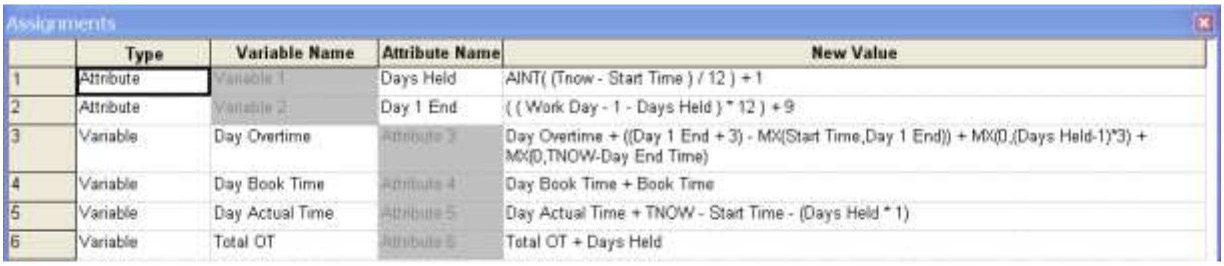
```
TNOW<=(Arrive Time+Book Time+Wait Allowance).
```

The promise time is calculated by adding the Book Time to the Wait Allowance (initialized to 1 hour). If this expression evaluates as True, the job is late and the entity is sent to the following **Assign** module where the variable *Day Wait Late* is incremented. The entity then exits the system. If the previous expression evaluates to False, the job is not late and the entity also exits the system.

Now let's go back and consider the case where the service on a vehicle is not completed by the end of the day and the vehicle must be held overnight. If this occurs, the entity is sent to the *Calculate Days Held Over* **Assign** module. The assignments for this module are shown in *Figure 7.15*. We need to be careful here as it is possible that a vehicle could be held over more than one night. It may not be likely, but it is possible. So we first calculate the number of days held over and assign it to the attribute *Days Held*. That value is computed as

```
AINT((TNOW - Start Time)/12)+1.
```

Remember that we already know that the job was held over one night and that a simulation day is 12 hours. You should be able to figure out why this expression yields the correct value.



	Type	Variable Name	Attribute Name	New Value
1	Attribute	Variable 1	Days Held	AINT((Tnow - Start Time) / 12) + 1
2	Attribute	Variable 2	Day 1 End	((Work Day - 1 - Days Held) * 12) + 9
3	Variable	Day Overtime	Attribute 3	Day Overtime + ((Day 1 End + 3) - MX(Start Time,Day 1 End)) + MX(0,(Days Held-1)*3) + MX(0,TNOW-Day End Time)
4	Variable	Day Book Time	Attribute 4	Day Book Time + Book Time
5	Variable	Day Actual Time	Attribute 5	Day Actual Time + TNOW - Start Time - (Days Held * 1)
6	Variable	Total OT	Attribute 6	Total OT + Days Held

Figure 7.15. The Assignments for Vehicles Held Over

In the next assignment, we'll calculate the simulation time that represents the normal end time of the day that the service was started and assign that value to the attribute *Day 1 End*. The expression is

```
((Workday–1–Days Held)*12)+9.
```

If you're not sure why this works, you might try creating some examples to check it out.

In the next assignment, we update the overtime hours, variable *Day Overtime*. This one is a little complicated, so we'll explain it in sections. The first two terms (not including the Day Overtime) compute the number of overtime hours on the first day of service as

```
((Day 1 End+3)–MX(Start Time, Day 1 End)).
```

The first term is the simulation time that represents the end of the first day of service (e.g., 12,24, etc.). We subtract from that the second term, which we explained earlier. This results in a value between zero and three hours. We then add to that the expression

```
MX(0,(Days Held–1)*3).
```

This term is positive only if the vehicle was held over for more than one day. If that is the case, we incur three hours of overtime for each additional day beyond the first day. We then add to this the last term

```
MX (0,TNOW–Day End Time).
```

This term yields any overtime that might have occurred on the day that the service was completed. If you remember that the current time is when the service was completed, you should be able to figure this out.

In the next assignment, we update the *Day Book Time* variable; this one should be obvious. Following that, we update the *Day Actual Time* variable by adding the value of the expression

```
    TNOW–Start Time–(Days Held*1).
```

Here we need to make sure that we do not include the first hour of each day, thus the reason for the last part of the above expression. Remember that the bays are not scheduled to be available for the first hour of each day and that we specified the preempt option for the schedule. This means that at the end of the 12-hour day, any service activities will be preempted and restarted when the bays become available, one hour after the start of the day.

The last assignment is used to update the total number of days that vehicles were held over. The departing entity is sent to the **Decide** module that checks to see whether this was a wait job. The logic for this and the remaining modules was discussed earlier.

The last module in this submodel destroys entities, so there is no exit from the submodel (which would have pleased Jean-Paul Sartre).

7.4.7 Control Logic

The logic for this submodel is fairly simple (the modules representing these logic steps are given in *Figure 7.16*). Up to now, we've usually considered an entity to have a physical counterpart. In this case, however, these entities have no physical meaning and are normally referred to as logic entities as they are included in the model for purposes of implementing logic or changing system conditions. We'll create one entity each day (every 12 hours) that will release the appointments to the shop, reset the appointment load variables for this day, calculate a few times, and dispose of the entity. You might note that this submodel has no entry or exit points.

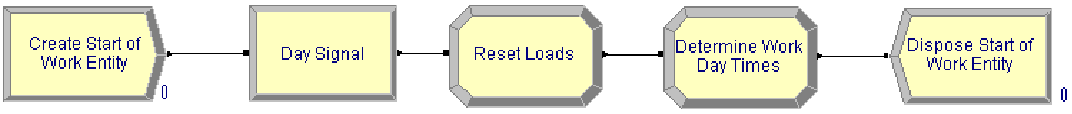
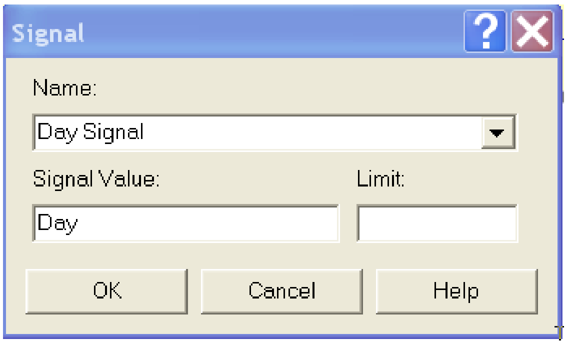


Figure 7.16. The Control Logic Submodel

The **Create** module is used to create our one entity each day. There is one interesting note about this module that we'll explain in more detail. It's the entry for the First Creation (0.999999). This means that these entities will be created just before the end of the first hour of each day. In the next module, we'll send a signal that will release all the entities from the current day's appointment queue and allow them to enter the queue where they will attempt to seize a service bay. Hopefully, you'll remember that the bays are not available during the first hour of each day. So you should be asking, "Why not a first creation value of 0 or 1?" If you didn't ask yourself that question, you should have! If you use a value of 0, the arrival time of each vehicle would be assigned in the Service Activity submodel as the start of the day, not the start of the workday. If you use a value of 1 and we assume that all three bays are available at the start of the day, then the first entity will be allocated Bay 1; the second, Bay 2; and the third, Bay 3. The remaining entities would be placed in the queue. This sounds right, but remember that we wanted to service our wait customers first. In this case, the first three customers in that day's appointment queue, regardless of their priority, would be serviced first.

As the remaining customers arrive to the queue, they would be ranked by their priority, which is what we want. So we timed the creation of our entity so the signal to release the current day's appointments would allow a very, very, very, small amount of time for the contents of the queue to be ranked properly, before the bays be come available.

This control entity then enters the following Signal module, shown in *Display 7.15*. This will cause a signal with a value of the variable *Day* to be sent. Remember that we included five Hold modules in the Service Activity submodel. When the signal is sent, it will release all of the entities currently in that hold queue that are waiting for a matching signal.



Signal

Name: Day Signal

Signal Value: Day

Limit:

Name	Day Signal
Signal value	Day

Display 7.15. The Signal Module

After sending the signal releasing all the appointments for the current day, the entity enters the following **Assign** module. This module will reset the values of the variables *Wait Load* and *Day Load* to 0 for the current day, since the queue for this day is now empty. In the second **Assign** module, we first calculate the value of the current day as

```
AINT((TNOW/12)+1).
```

We next calculate value for the variable *Day Start Time* based on the value of the current variable *Work Day*. That expression is

```
((Work Day–1)*12)+1.
```

Finally, we add eight hours to the value computed for the *Day Start Time* to obtain the value for the variable Day End Time. We won't provide detailed discussions on why these expressions give the correct values. By now we assume that you can figure them out on your own. The entity is then sent to the **Dispose** module where it is destroyed.

This completes the development of our model logic, and we're afraid that it's now the time to address a rather delicate issue errors!

7.5 Finding and Fixing Model Errors

If you develop enough models, particularly large ones, sooner or later you'll find that your model either will not run or will run incorrectly. It happens to all of us. If you're building a model and you've committed an error that Arena can detect and that prevents the model from running, Arena will attempt to help you find that error quickly and easily. We suspect that this has probably already happened to you by this time, so consider your self lucky if that's not the case.

Typical errors that prevent a model from running include: undefined variables, attributes, resources, unconnected modules, duplicate use of module names, misspelling of names (or, more to the point, inconsistent spelling whether it's correct or not), and so on. Although Arena tries to prevent you from committing these types of errors, it can't give you maximum modeling flexibility without the possibility of an occasional error occurring. Arena will find most of these errors when you attempt to check or run your newly created model. To illustrate this capability, let's intentionally insert some errors into the model we've been developing. Unless you're really good, we suggest that you save your model under a different name before you start making these changes.

We'll introduce two errors. First, open the Service Activity submodel and delete the connector between the first entry point and the first Hold module, *Monday List*. Next, open the dialog box for the second **Assign** module, *Assign Start Time*. Change the New Value of the assignment from *TNOW* to *TNO*. These two errors are typical of the types of errors that new (and sometimes experienced) modelers make. After making these two changes, use the *Run> Check* option or the *Check* button (✓) on the Run Interaction toolbar to check your model. An Arena Errors/Warnings window should open with the message

```
ERROR:
Unconnected submodel entrance point
```

This message is telling you that there is a missing connector (the one we just deleted). Now look for the *Find* and *Edit* buttons at the bottom of the window. If you click the *Find* button, Arena will take you to and highlight the offending module. If you click the *Edit* button, Arena will take you directly to the dialog box of that offending module. Thus, Arena attempts to help you find and fix the discovered error. So, click *Find* and add the connector that we deleted.

Having fixed this problem, a new check of the model will expose the second error:

```
ERROR:
A linker error was detected at the following block:
*54 37$ ASSIGN:Start Time=TNO:NEXT(35$);
Undefined symbol: TNO
Possible cause: A variable or other symbol was used
                without first being defined.
Possible cause: A statistic was animated, but statistics collection
                was turned off.
To find the problem, search for the above symbol name using
Edit-Find from the menu.
```

This message tells you that the symbol *TNO* (the misspelled Arena variable *TNOW*) is undefined. Click *Edit* and Arena will take you directly to the dialog box of the **Assign** module where we planted the error. Go ahead and fix the error. During checking, if you find that you've misused a name or just want to know where a variable is used, use the *Edit> Find* option to locate all occurrences of a string of characters. You might try this option using the string *TNOW*. If for some reason you forgot what the error was, use *Run>Review Errors* to reopen the error window with the last error message.

We'll now introduce you to the Arena command-driven Run Controller, which can be used to find and eradicate faulty logic or runtime errors. Before we go into the Run Controller, let's define three Arena variables that we'll use: NQ, MR, and NR. You've already seen all three of these variables. There are many Arena variables that can be used in developing model logic and can be viewed during runtime. The three of interest are:

NQ(Queue Name)	Number in queue
MR(Resource Name)	Resource capacity
NR(Resource Name)	Number of busy resource units

Note that these will generally change during the simulation so their values refer to the status at the moment.

You can think of Arena variables as magic words in that they provide information about the current simulation status. However, they're also reserved words so you can't redefine their meanings or use them as your own names for anything. We recommend that you take a few minutes to look over the extensive list of Arena variables, which can be found in the Help system. You might start by looking at the variables summary and then look at the detailed definition if you need more information on a specific variable.

We're now going to use the Run Controller to illustrate just a few of the many commands available. We're not going to explain these commands in any detail; we leave it up to you to explore this capability further, and we recommend starting with the Help system.

Enter the Run Controller by the *Run>Run Control>Command* option or the *Command* button (0.>) on the Run Interaction toolbar to open the command window. At this point, the model is ready to run, but it has not started yet. Notice that the current time is 0.0. Now let's use the VIEW command to look at the first four lines of the model. Enter the command

```
   0.0 > view source 1-4
```

The response will be

```
   Model name: Model 07-01
      1 52$ CREATE,l,HoursToBaseTime(0.01) ,Customer Ca1ls;
            HoursToBaseTime(12) :
            NEXT(53$);
      2 53$ ASSIGN:
            Create Call Ins.NumberOut=Create Call Ins.NumberOut+1:
            NEXT(4$) ;
      3 4$ ASSIGN:Day=Day+1:Call Number=1:NEXT(5$);
      4 5$ BRANCH,l:
            If,TNOW<l,56$,Yes:
            Else,57$,Yes;
```

The Run Controller has listed the first four lines of SIMAN code generated by our Arena model. This code corresponds to the sequence of the first three modules in our Generate Appointment Calls submodel, Create - Assign - Decide, that we used to create the appointment entity, make the initial assignment, and check for startup. Note that Arena has inc1uded an extra **Assign** block that is part of the module we placed. Now that we know what the first four lines of code are, let's watch what happens when we execute that code. First, we need to change the Run Controller settings so it will display what happens. We do this with the SET TRACE command, so enter

```
   0.0>set trace *
```

This will activate a trace of all actions performed by Arena/SIMAN when we run our model. Let's let it run for only these four lines of code. Use the STEP command to do this by requesting that Arena step through the first, or next, four blocks of the model. We enter the command as

```
   0.0>step 4
```

Arena/SIMAN responds with

```
SIMAN System Trace Beginning at Time: 0.01
Seq#     Label          Block        System Status Change
----------------------------------------------------------------
Time:    0.01 Entity:   5
         1 52$          CREATE
```

```
                                  Entity Type set to Customer Calls Next
                                  creation scheduled at time 12.01
                                  Batch of 1 Customer Calls entities cre-
                                  ated
2 53$         ASSIGN
                                  Create Call Ins.NumberOut set to 1.0
3 4$          ASSIGN
                                  Day set to 5.0
                                  Call Number set to 1.0
4 5$          BRANCH, 1:
                                  If,TNOW<1,56$,Yes:
                                  Else,57$,Yes;
```

A single entity is created at time 0.01, and the next creation is scheduled to occur 12 hours later. It enters the first **Assign** block (the one included by Arena) and makes an assignment. The entity is then sent to our **Assign** module where Day is incremented to a value of 5 and *Call Number* is assigned a value of 1. The entity is then directed to our second **Decide** module where the condition is checked.

Now let's turn off the trace, using the CANCEL command, and advance the simulation time to 1.1 using the GO UNTIL command.

```
0.0>cancel trace *
*** All trace options canceled.
0.0>go until 1.1
Break at time: 1.1
```

We're now at simulation time 1.1, so let's see what we have in the queue waiting for a service bay (resource Bay 1, Bay 2, or Bay3). We'll use the SHOW command to find the number in queue, the queue number, and the status of all other queues in the model, as follows

```
1.1   Hours>show NQ(Bay Queue)
      NQ(Bay Queue)   =   15
1.1   Hours>show Bay Queue
      Bay Queue =     3
1.1   Hours>show NQ(*)
      NQ(   1)     0
      NQ(   2)     16
      NQ(   3)     15
      NQ(   4 )     18
      NQ(   5)     19
      NQ(   6)     0
```

As shown, there are 15 entities currently in the *Bay Queue*, and the number assigned by Arena to that queue is 3. When Arena checks a model, it assigns every resource, queue, etc., a number that it uses internally to reference these items during the run. The number that Arena assigns to each item can change from run to run, so a word of caution-if you edit and save your model, Arena may write it out slightly differently; thus the next time you run the model, the *Bay Queue* may not be number 3. There are ways to force the *Bay Queue* always to be number 3, but we will hot discuss them here.

Now let's look at the entities in the queue by using the VIEW QUEUE command as follows (we'll show only the first two entities here):

```
1.1 Hours>view queue Bay Queue
*** Queue 3 contains 15 Entities ***
*** Rank 1: Entity number 13
Entity.SerialNumber   = 1
Entity.Type           = 3
Entity.Picture        = 4
Entity.Station (M)    = 0
Entity.Sequence (NS)   = 0
Entity.JobStep (IS)   = 0
Entity.CurrentStation = 0
Entity.PlannedStation = 0
Entity.CreateTime     = 0.01
Entity.VATime         = 0.0
Entity.NVATime        = 0.989999
Entity.WaitTime       = 0.0
Entity.TranTime       = 0.0
Entity.OtherTime      = 0.0
Day Inc = 1.0
Book Time = 1.45
Day 1 End = 0.0
Priority = 2.0
Number Of Calls 22.0
Try Day = 1.0
Bay Number = 0.0
Start Time = 0.0
Days Held = 0.0
Call Number 1.0
Arrive Time = 0.999999
```

(View Queue listing continued on next page)

```
*** Rank 2: Entity number 17
Entity.SerialNumber    = 1
Entity.Type            = 3
Entity.Picture         = 4
Entity.Station (M)     = 0
Entity.Sequence (NS)   = 0
Entity.JobStep (IS)    = 0
Entity.CurrentStation  = 0
Entity.PlannedStation  = 0
Entity.CreateTime      = 0.01
Entity.VATime          = 0.0
Entity.NVATime         = 0.0
Entity.WaitTime        = 0.989999
Entity. TranTime       = 0.0
Entity.OtherTime       =0. 0
Day Inc = 1.0
Book Time 1.1333333
Day 1 End = 0.0
Priority = 2.0
Number Of Calls 22.0
Try Day = 1.0
Bay Number = 0.0
Start Time = 0.0
Days Held = 0.0
Call Number = 1.0
Arrive Time = 0.999999
```

This displays the internal entity number and all of the attribute values for each entity.

As you can see, Arena has defined a large number of attributes for its own use. You might also note that the attribute Arrive Time is defined by one of the modules we placed.

Since there are calls waiting in the queue for a Bay resource, all units of the resource currently available must be allocated. We can check this by using the SHOW command as follows:

```
1.1 Hours>show NR(Bay 1), MR(Bay 1)
NR(Bay 1) = 1
MR(Bay 1) = 1
```

Just for fun, let's add some resources to the pool using the ASSIGN command

```
1.1 Hours>assign MR(Bay 1)=2
Resource Bay 1 capacity change by 1.0 to 2.0
Entity 13 removed from queue Bay Queue
Tally Bay Queue.WaitingTime recorded 0.100001
Resource allocated to entity 13
Seized 1.0 unit(s) of resource Bay 1
Bay Number set to 1.0
```

This increases the number of Bay 1 resources from 1 to 2. Since there is now an additional resource available, Arena allocates it to the next entity in the queue. Now let's check the status of the queue and resource.

```
1.1 Hours>show NR(Bay 1),NQ(3)
NR(Bay 1) = 2
NQ(3) = 14
```

As expected, the additional resources were allocated and there are now 14 entities in the queue. It may surprise you that we can change the value for the number of resources on the fly, but that's actually what a schedule does. Just be aware that there are some Arena variables you can't change; e.g., TNOW, NQ, and NR. Now close out the command window and terminate the simulation.

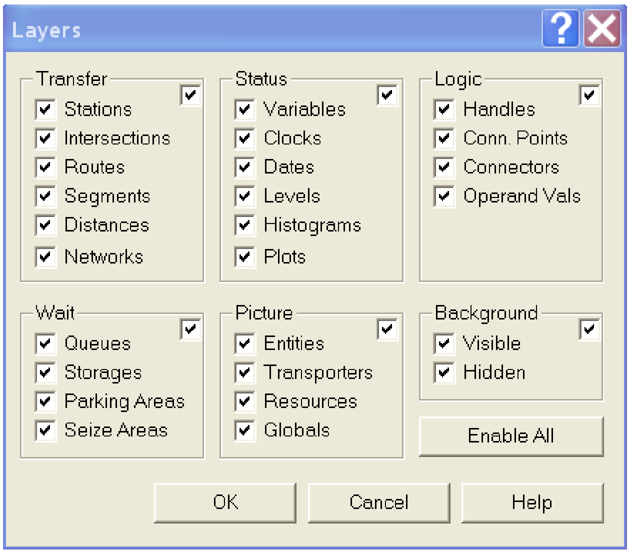
This should give you a good idea of how the Run Controller works. It really allows you to get at the SIMAN model that's running underneath; however, it's not for the timid, and you really need to spend some time with it and also have at least a cursory understanding of the SIMAN language.

There are easier, or at least less frightening, ways to check the model accuracy or find logic errors. The most obvious is through the use of an animation that shows what the model logic is doing. However, there are times when you must dig a little deeper in order to resolve a problem. The Run Interaction toolbar provides some of the tools you may need. We've already looked at the Check and Command options, so now let's look at the Highlight Active Module option by selecting *Run>Run Control>Highlight Active Module*.

Now click on the Navigate bar and run your model. If you're in the Top-Level Model view, you'll see the submodels highlighted as entities pass through them. With your model running, double-click on the Service Activity submodel and you'll see the modules highlighted as entities pass through. However, on some computers, this is happening so fast you'll typically see only the modules highlighted where the entity stops; e.g., Delay and **Dispose** modules. If you don't believe us, use the zoom option to zoom in so that only one or two modules fill the entire screen. Now click the *Go* or *Step* button and you'll see that Arena jumps around and tries to show you only the active modules. Again, it's generally working so fast that not all the modules are highlighted, but it does at least try to show them.

If you've selected these settings and your window doesn't contain all the logic modules in the submodel, Arena will change the window view each time it needs to show a module that isn't currently visible (but only within the current submodel). It basically puts that module in the center of your window, and that view remains until it encounters another module that's not in the current view. Although this allows you to see more of the active modules, you may get dizzy watching the screen jump around, so let's pause your run and go back to the view showing all the logic modules. When you finally get bored, stop your model and clear the Highlight Active Module option.

You also have some options as to what you see when the model is running. Select *View>Layers* to bring up the dialog box shown in *Figure 7.17*. You can do this while you're still in Run mode. This dialog box will allow you to turn on (check) or turn off (clear) different items from showing during run time of your model.



Layers

Transfer ☑
- ☑ Stations
- ☑ Intersections
- ☑ Routes
- ☑ Segments
- ☑ Distances
- ☑ Networks

Status ☑
- ☑ Variables
- ☑ Clocks
- ☑ Dates
- ☑ Levels
- ☑ Histograms
- ☑ Plots

Logic ☑
- ☑ Handles
- ☑ Conn. Points
- ☑ Connectors
- ☑ Operand Vals

Wait ☑
- ☑ Queues
- ☑ Storages
- ☑ Parking Areas
- ☑ Seize Areas

Picture ☑
- ☑ Entities
- ☑ Transporters
- ☑ Resources
- ☑ Globals

Background ☑
- ☑ Visible
- ☑ Hidden

Figure 7.17. The Layers Dialog Box

Now let's assume that you have a problem with your model and you're suspicious of the Release module in the Service Activity logic. It would be nice if the simulation would stop when an entity reaches this module, and there are three ways to cause this to happen. You could enter the command window and figure out the right setting, or take the easy way out and use the *Run>Run Control>Break on Module* option. You first need to highlight the module or modules on which you want to break. Then select *Run>Run Control>Break on Module*. This will cause the selected options to be outlined in a red box. Now click the Go button. Your model will run until an entity arrives at the selected module-it may take a while! The module will pause when the next entity arrives at the selected module. You can now attempt to find your error (for instance, by repeatedly clicking the *Step* button until you notice something weird) or click *Go*, and the run will continue until the next entity arrives at this module. When you no longer want this break, highlight the module and use *Run>Run Control>Break on Module* to turn off the break.

Break on Time	select
Value	198
Break on Condition	select
Value	NQ (Bay Queue) >17

Display 7.16. The Break Option

You can also use the *Run> Run Control> Break* option or the *Break* button ✋) to set breaks on the Simulation time, on a condition, or on an entity. Break on Time causes the model to run but then pause at the entered time, much like the GO UNTIL command in the Run Controller. Break on Condition allows you to enter a condition (such as *NQ* (*Bay Queue*)>17) that would cause the run to pause whenever this condition is satisfied. Break on Entity would cause the run to pause whenever the selected entity is about to become active. For this option, you need to know the entity ID number, which can be found using the Run Controller or by double-clicking on the entity picture during a paused simulation run. *Display 7.16* shows a break at time 198 and a break on the example condition.

Another good way to monitor what's going on during a run is to define *watches*. Before we do this, we suggest that you turn off all your Module breaks. Now use the *Run*> *Run Control*>*Watch* option or the *Watch* button (👓) to open the Watch window. Let's add two watches one for the number in the bay queue and another for the status of Bay 1, as shown in *Display 7.17* (which also illustrates the kind of output you'll see in the Watch window). You can add as many watches as you want, and if you default the label, Arena will use the expression as the default label.

```
Number in Bay Queue   2.000000
Status of Bay 1       1.000000
```

Expression Label	NQ (Bay Queue) Number in Bay Queue
Expression Label	NR(Bay 1) Status of Bay 1

Display 7.17. The Watch Option

Before you close this window, select *Window*>Tile. This will display both the Model window and the Watch window at the same time. (Make the Watch window active by clicking in it.) As the model runs, if you click *Go* with the Watch window active, you'll see the values change for the two expressions we requested. If you had an animation, it would also be visible. However, be aware that the values for the watch expressions are updated only when Watch is the active window.

Finally, during a run, you might want to view reports on the status of the model. You can view reports at any time during a run. Simply go to the Project Bar and click on the Reports panel. You can request a report when the model is running or when the model is in Pause mode. Just like the Watch window, you could have an animation running and periodically request a report and never stop the run. Just remember to close the report windows when you're done viewing them.

Even if you don't use the Run Controller, the options available on the Run Interaction toolbar provide the capability to detect model logic errors without your needing to become a SIMAN expert. We recommend that you take a simple animated model and practice using these tools now so you'll understand how they work. You could wait until you need them, but then not only would you be trying to find an error, but also trying to learn several new tools! Now that you know how to use these tools, let's animate the automotive shop model.

7.6 Animating the Automotive Shop Model

Earlier we talked about how this model is different from previous models in that it's not obvious what should be animated. There are basically four parts to our model: appointment calls, service activities, performance, and control logic. Most of these do not lend themselves to animation. However, we can animate the status of the appointments and the service activities.

Let's start with the status of the appointment calls. Our first two submodels, Generate Appointment Calls and Make Appointment, contain the logic used to make our appointments. Recall that we used several key variables when we were making appointments and then placed the appointment entity in a queue to wait for the scheduled service day. So the first part of our animation will contain these key variables and the five appointment queues. When completed, that section of our animation will look like *Figure 7.18*.

We decided that there are three variables of interest-the Book Time hours currently scheduled for any day, the current number of appointments for a day, and the number of wait customers currently scheduled for the day. The first and last can be obtained using the two arrayed variables that we defined for this purpose, *Day Load* and *Wait Load*. Although we did not define a variable for the number of appointments for a given day, we place these appointments in a queue. Thus, we can use the Arena variable NQ for our animation. We used the *Variable* button (0.0) from the Animate toolbar to place the variables on our animation.

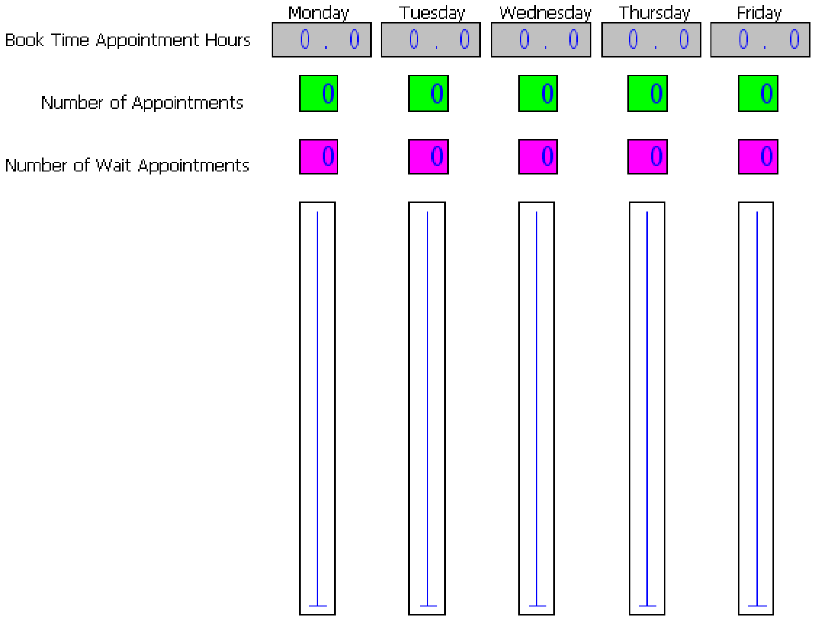


Figure 7.18. The Appointment Calls Status Animation

Here are some words of advice if you decide to attempt to animate this model yourself. If you have multiple variables or queues in the same animation, you generally want all of them to have the same basic look and feel. Place your first variable, enter the data, choose your colors, and size the variable. While you're developing this look and feel, you might want to consider activating the *Snap* option ( ). This approach will easily allow you to reproduce additional variables with the same exact size and spacing. Once you have completed your first variable, use *Copy* and *Paste* to make four duplicates and edit the duplicates to develop the additional variables you want. For our animation, we added a variable identifier using the *Text* option (A) from the Draw toolbar.

You can repeat this sequence of activities for the next row of variables or use the *Select* feature to capture all the objects (variable, text, etc.) that compose the row, then use *Copy* and *Paste* to make duplicates. Position these duplicates or copies wherever you want them and edit the duplicate. This approach will eliminate a lot of busy work and also ensure that the appearance of your animated variables is consistent.

In addition to the variables, we also added the five appointment queues to our animation. Again we suggest that you place the first queue, *Monday Appointment Queue*, add any background you want, and then use *Copy* and *Paste* to create the remaining four queues. Although there will be no movement on this portion of our animation, it will give us a clear view of the status of our appointments.

For the service activity portion of our animation, shown in *Figure 7.19*, we developed and placed pictures for our three bay resources. We also added text below each resource picture to identify the bay number. Again we recommend that you use the *Copy* and *Paste* features. We then placed the *Bay Queue* using the *Queue* button ()from the Animate toolbar. You might note that we chose the point type of queue and specified 24 points. We then positioned each point as shown in *Figure 7.19*. Here you really want to have the *Snap to Grid* option turned on. We also suggest that you initially place the first three or four points and then run your model to check to see if you have your spacing right before you place all 24 points.

For our entity pictures, we started with basic pictures that come with Arena. For the pictures that represent our appointments, we used the Blue Page and Red Page pictures. We did modify them by cutting off the bottom of each of the pages so we could shorten the length of our appointment queues. When you modify these pictures, they are saved with the .doe file for that model. If you bring up a new model window, it includes the original pictures, not the revised pictures. For the entity pictures of the vehicles that arrive for service, we opened *Vehicles.plb* and copied a picture of a red car to our picture library for our model. We used that picture for our wait customers. We then made a copy of the picture and changed the color of the car from red to blue for our normal customer.

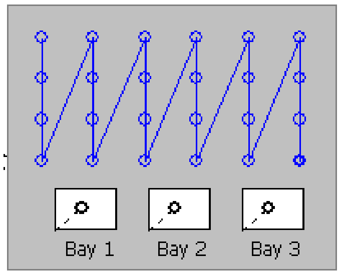


Figure 5·19. The Service Activity Animation

Finally, we decided to add a clock and a way to show the day of the week, as shown in *Figure 7.20*. We wanted to add a clock so the viewer can accurately relate the current model status being displayed by the animation to the time of day. Recall that our day is 12 hours long. We handle the appointment calls during the first hour, our service activity during the next eight hours, and allow for overtime during the last three hours. So we'll assume our clock runs from 0 to 12. You could use the *Clock* button () on the Animate toolbar to get the feature we need since our day is 12 hours and there is a choice between a 12- or 24-hour clock. However, if our day length were any other value than 12 or 24, this feature would not work, so we'll show you how to build your own clock.

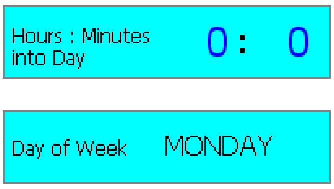


Figure 7.20. The Clock and Day Indicator

To do this, we separated our clock display into two animated variables separated by a colon. The first variable represents the current hour, and the second represents the current minute. Now we just have to figure out how to calculate these values.

Let's start with the calculation for the current hour. The expression

```
    AINT(TNOW-AINT(TNOW/12) *12)
```

will return the current hour using our Arena variable *TNOW* This function returns the integer portion of the enclosed expression; that is, it *truncates* the value defined by the expression. We'll leave it to you to figure out the expression. This displays the current hour based on a 12-

hour clock. To display the hour on the animation, we simply use the *Variable* option from the Animate toolbar and enter the above expression rather than a simple variable name.

Calculating the current minute is easier, although it's certainly not obvious. We use the expression

```
    (TNOW–AINT(TNOW))*60
```

which will return the value needed.

If you want to be assured that both the hour and the minute variable display are exactly the same size, we suggest that you place and size the hour variable first. Then copy, paste, and edit the expression for the minute variable.

Finally (since we're already being cute), we decided to add a display for the current day of the week. We used the *Global* button (■) from the Animate toolbar to accomplish this, which opens the Global Picture Placement window, as seen in *Display 7.18*. This window is very similar to the entity and resource picture windows. You first enter an expression (in our case, the variable Day) and then associate a picture with a *Trigger value* for that expression. You can have as many (trigger value, picture) pairs as you need to cover the range of pictures you need to display. You create each picture just like we did for entities and resources, but the picture will be based on the current value of the entered expression. In our model, the expression *Day* is set to 4 at the start of each replication. It's immediately incremented to a value of 5 and then to 1, and every 12 hours thereafter it's again incremented by 1 (if it reaches a value of 6, it's reset to 1). The initial symbol that will be displayed is the *THURSDAY* symbol. When the *Day* variable changes to 5, the symbol will change to *FRIDAY* and then to *MONDAY* and remain that way until the end of the day.

Expression	Day
Trigger Value	1
Trigger Value	2
Trigger Value	3
Trigger Value	4
Trigger Value	5

Display 7.18. The Global Picture Placement for the Current Day

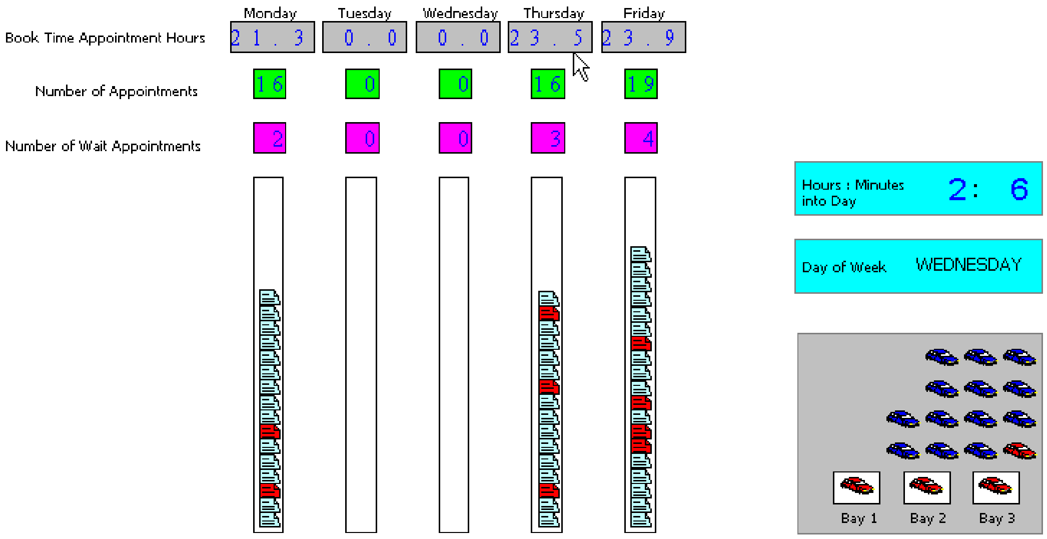
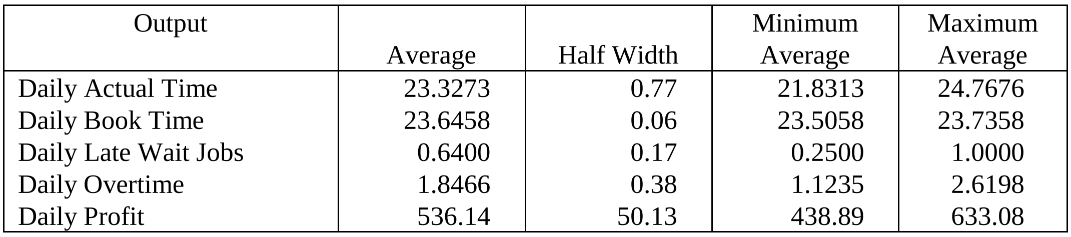


Figure 7.21. The Automotive Shop Animation

To ensure that all the symbols were the same size, we created the first symbol, *MONDAY*, and then used *Copy* to make duplicates to edit for the remaining symbols. When you close the window, you need to place your global symbol very much like you place text. After placing the symbol, you may have to change its size. You can do this in the model window, or you can reopen the Global Picture Placement window and change the Size Factor, which will rescale your symbol. We also added a label and put a box around our variables and symbol; the final animation is shown in *Figure 7.21*, which is from the first day of the first replication.

If you run your model to completion, the Category Overview report will be a summary across the ten replications. (You can look at the data for each replication via the Category by Replication report in the Reports panel.) Although we're not going to show the Category Overview report, we'll point out some statistics of interest. Due to the unique performance measures that we chose for this model, we are only interested in two parts of this report. The first statistics of interest are the bay resource utilizations. All three bays are utilized for approximately 7.8 hours each day. This might indicate that we should increase the number of Book Time hours that we allow to be scheduled each day, the *Max Load* variable. We'll do this in Section 6.4 (and will do a proper statistical analysis there of the effect of this change).

The second part of the report in which we're interested is the output statistics, as shown in *Figure 7.22*. The daily overtime average is 1.8 hours and the daily profit is $536.14. Given the fact that our bay resources are used only an average of 7.8 hours per day, and that we also have an average of 1.8 hours of overtime each day, it appears that this system has a high degree of variability. Again, we'll take this up in Chapter 6.



Output			Minimum	Maximum
	Average	Half Width	Average	Average
Daily Actual Time	23.3273	0.77	21.8313	24.7676
Daily Book Time	23.6458	0.06	23.5058	23.7358
Daily Late Wait Jobs	0.6400	0.17	0.2500	1.0000
Daily Overtime	1.8466	0.38	1.1235	2.6198
Daily Profit	536.14	50.13	438.89	633.08

Figure 7.22. The Automotive Shop Output Statistics

If you haven't been building your own model as you've been reading through this chapter, now would be a good time to open the model we built (Model 07-01. doe) and poke around to make sure you understand the concepts and features we've presented.

7.7. Model 7.2. Enhancing the Automotive Shop Model

Now, let's assume that you've finished your model and have taken it for a test ride (pun intended) for the customer and they've noticed a few missing elements. This is a very common occurrence in the real world. In their initial description of the problem, the customer neglected to tell you about two key issues that are missing from the model as it stands. The first issue is that not all jobs can be done in every bay. In order to reduce the cost of the facility, they have equipped Bays 2 and 3 so they can handle all jobs. Bay 1 can only handle a subset of the jobs (40%). The second problem is that not all customers arrive at the start of the day. In reality, some of the customers never show up for their appointments. Analysis of their data indicates that between 60% and 70% of the customers arrive on time. We'll assume that this proportion will follow a uniform distribution with parameters 0.6 and 0.7. The remaining customers tend to trickle in randomly over the next two hours, with an occasional customer not showing up. We also analyzed these data, and we'll show you the results later.

7.8 New Modeling Issues for Model 7-2

From a modeling viewpoint, we have several new concepts. To solve the first problem, we'll need to add an additional set to our model and figure out how to structure our logic so we make the most efficient use of our resource bays. For our second problem, we'll have to modify our customer arrival process and figure out a way to model our late arrivals.

7.8.1 Sets and Resource Logic

In our new model, we'll still have our pool of bay resources (the set *Bays*), but only a portion of the service will be performed by this set. We must create a second resource set that contains only Bays 2 and 3 to service the remaining customers' vehicles. We then need to structure our priorities so that we use these resources efficiently. Basically, we need to have our two different types of customers waiting in two different queues, so we must acquaint you with how Arena allocates resources to waiting entities.

We hope by now that this allocation process is fairly clear if all entities requesting a resource are resident in the same queue. However, if there are several places within a model (different Queue - Seize combinations) where the same resource could be allocated, some special rules apply. Let's first consider the various circumstances under which a resource could be allocated to an entity:

> an entity requests a resource and the resource is available,
> a resource becomes available and there are entities requesting it in only one of the queues, or
> a resource becomes available and there are entities requesting the resource in more than one queue.

It should be rather obvious what occurs under the first two scenarios, but we'll cover them anyway.

If an entity requests a resource and the resource is available, you guessed it, the resource is allocated to the entity. In our second case, when a resource becomes available and there are entities in only one of the queues, then the resource is allocated to the first entity in that queue. In this case, the determining factor is the queue-ranking rule used to order the entities. Arena provides four ranking options: First In, First Out (FIFO); Last In, First Out (LIFO); Low Value First; and High Value First. The default, FIFO, ranks the entities in the order that they entered the queue. LIFO puts the most recent arrival at the front of the queue (like a push/pop stack). The last two rules rank the queue based on an expression you define as an attribute of each of the entities in the queue. For example, as each entity arrives in the system, you might assign a due date to an attribute of that entity. If you selected Low Value First based on the due-date attribute, you'd have the equivalent of an earliest-due-date queue-ranking rule. As each successive entity arrives to the queue, it is placed in a position based on increasing due dates.

The case where entities in more than one queue request the resource is a bit more complicated. Arena first checks the seize priorities; if one of the seize priorities is a smaller number (higher priority) than the rest, the resource is allocated to the first entity in the queue preceding that seize. If all priorities are equal, Arena applies a default tiebreaking rule, and the resource is allocated based on the entity that has waited the longest regardless of the queue it's in. Thus, it's essentially a FIFO tie-breaking rule among a merger oft he queues involved. This means that if your queues were ranked according to earliest due date, then the entity that met that criterion might not always be allocated the resource. For example, a late job might have just entered a queue, whereas an early job may be at the front of another queue and has been waiting longer.

Arena provides a solution to this potential problem in the form of a shared queue. A *shared queue* is just what its name implies a single queue that can be shared by two or more seize activities. This allows you to define a single queue where all entities requesting the resource will be placed regardless of where their seize activities are within your model. Arena performs the bookkeeping to ensure that an entity in a shared queue continues in the proper place in the model logic when it's allocated a resource. Although this solves the problem when you have multiple places in your model where you seize the same resource, it doesn't solve the problem when you are seizing resources from different sets that have some common resources, which is the case we have for our model. We'll show you how to overcome this problem shortly.

7.8.2 Non-stationary Arrival Process

Changing our model to limit the number of customers who arrive on time is fairly simple and we'll show you how to do this in Section 7.9.2. The differences start with the arrival process of the late customers, which has a rate that varies over time. This type of arrival process is fairly typical of service systems and requires a different approach, Arrivals at many systems are modeled as a *stationary Poisson process* in which arrivals occur one at a time, are independent of one another, and the average rate is constant over time. For those of you who are not big fans of probability, this implies that we have exponential interarrival times with a fixed mean. You may not have realized it, but this is the process we used to model arrivals in most of our previous models (with the exception of Model 4-5, which was contrived to illustrate a particular point). There was a slight variation of this used for the Part B arrivals in our Electronic and Test System modeled in Chapter 4. In that case, we assumed that an arrival was a *batch* of four; therefore, our arrivals still occurred one batch at a time according to a stationary Poisson process.

For this model, the mean late customer arrival rate is a function of time. These types of arrivals are usually modeled as a *non-stationary Poisson process*. An obvious, but incorrect, modeling approach would be to enter for the Time Between Arrivals in a **Create** module an exponential distribution with a user-defined variable as a mean Value, then change this Value based on the rate for the current time period. For our example, we'd change this Value every 15 minutes. This would provide an approximate solution if the rate change between the periods was rather small. But if the rate change is large, this method can give very misleading (and wrong) results. The easiest way to illustrate the potential problem is to consider an extreme example. Let's say we have only two periods, each 30 minutes long. The rate for the first period is 3 (average arrivals per hour), or an interarrival-time mean of 20 minutes, and the rate for the second period is 60, or an interarrival-time mean of 1 minute. Let's suppose that the last arrival in the first time period occurred at time 29 minutes. We'd generate the next arrival using an interarrival time mean Value of 20 minutes. Using an exponential distribution with a mean of 20 could easily return a value more than 31 for the time to the next arrival. (With probability $e^{-31/20}$ = 0.21, to be (almost) exact. Actually this figure is the *conditional* probability of no arrivals in the second period, *given* that there were arrivals in the first period and that the last of these was at time 29. This is not quite what we want, though; we want the *unconditional* probability of seeing no arrivals in the second period. It's possible to work this out, but it's complicated. However, it's easy to see that a lower bound on this probability is given by the probability that the first arrival after time 0, generated as exponential with mean 20 minutes, occurs after time 60. This is one way (not the only way) to have no arrivals in the second period, and has probability $e^{-60/20}= e^{-3}= 0.0498$. Thus, the incorrect method would give us at least a 5% chance of having no arrivals in the second period.)

This would result in no arrivals during the second period, when in fact there should be an expected value of 30 arrivals. (The probability of no arrivals in the second period should be

$e^{-60(1/2)}$= 0.000000000000093576.) In general, using this simplistic method causes an incorrect decrease in the number of arrivals when going from one period to the next with an increase in the rate, or a decrease in the interarrival time. Going from one period to the next with a decrease in the rate will incorrectly increase the number of arrivals in the second period.

Nevertheless, it's important to be able to model and generate such arrival processes correctly since they seem to arise all the time, and ignoring the non-stationarity can create serious model-validity errors since the peaks and troughs can have significant impact on system performance. Fortunately, Arena has a built-in ability to generate non-stationary Poisson arrivals (and to do so correctly) in the **Create** module. We'll show you how to set it up in Section 7.9.2. The underlying method used is described in Section 12.3.

7.9 Building Model 7-2

Since we have two different problems to resolve for our new model, we'll treat them as two different additions. First we'll alter our model to account for the fact that not all jobs can be serviced in all three bays. Once we've completed these changes, then we'll take care of the customer-arrival problem.

7.9.1 Modeling the Service Bays

Model 7-1 requires three basic logic changes to account for how jobs are serviced by the bay resources. First, we'll need to define a new attribute to tell us the new job type. Then we'll need to add a new set containing the two resources to handle all the jobs. Finally, we'll have to alter our service logic. We'll make these modifications in this order.

We'll start by adding an assignment to the last **Assign** module, *Assign Appointment Attributes*, in our Generate Appointment Calls submodel. The value is generated from a discrete distribution, *DISC*(0.60, 1, 1.0, 2), and assigned to a new attribute named Job Type. If the resulting value is 1, the job can be serviced in any bay. If the value is 2, the job can only be serviced in Bays 2 or 3. Next we'll define a new set with the name *Bays 2* or *3*. The members of this set, which can service any job, are the resources *Bay 2* and *Bay 3*.

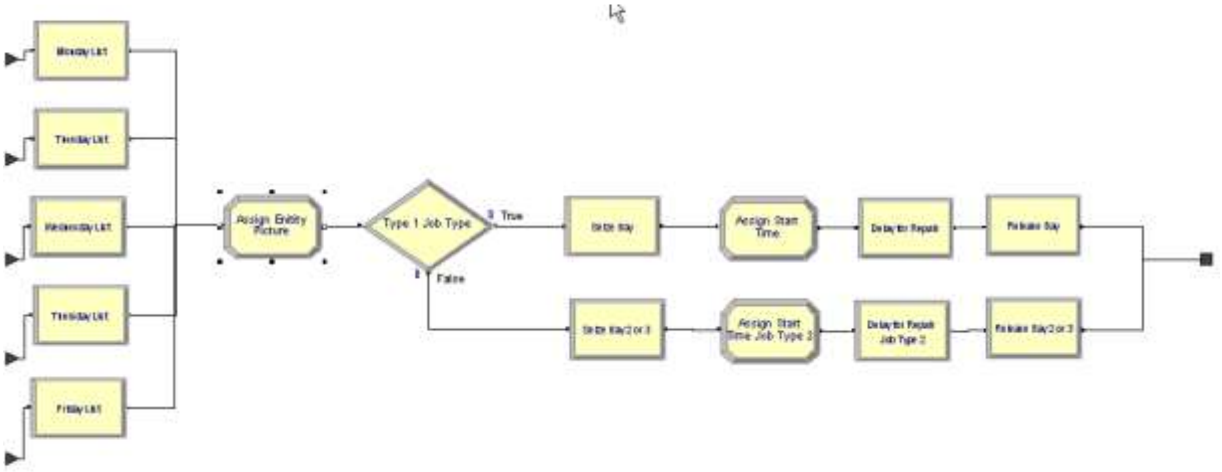


Figure 7.23. The New Service Activity Logic

We now need to modify the logic in our Service Activity submodel. The new logic for this submodel is shown in *Figure 7.23*. If you're looking at this new model, you'll find that the five Hold modules at the left are the same as in Model 7-1.

The first change occurs in the Assign Entity picture **Assign** module. Here we decided that we need to distinguish between the two new job types in our animation. To accomplish this, we added two new pictures to our Vehicle picture set. We copied the red and blue car pictures and changed the door colors to yellow. This picture set now contains the following pictures: Red Car, Blue Car, Red Yellow Car, and Blue Yellow Car. The Red Car is for a Priority 1 and Job Type 1 vehicle; the Blue Car is for a Priority 2 and Job Type 1 vehicle; the Red Yellow Car is for a Priority 1 and Job Type 2 vehicle; and the Blue Yellow Car is for a Priority 2 and Job Type 2 vehicle. Remember that a Priority 1 is a wait job and a Job Type 1 can go to all three bays. The expression for the new picture set is

```
    Vehicle(Priority+(AINT(Job Type/2))*2).
```

When you watch the new animation, you'll now be able to tell the difference between the two job types.

The changes to the rest of the submodel involved the insertion of a new **Decide** module following this assignment and the addition of a new row of modules to handle our new job type. This new **Decide** module is used to send the entities with a Job Type value of 1 to the upper row of modules, the *True* exit. These are the jobs that can be serviced in any bay. The jobs that can only be serviced in Bays 2 or 3 (Job Type value of 2) are sent to the lower row of modules, the *False* exit. The upper row of four modules are the same ones we had in Model 7-1, with one minor change, which we'll point out shortly.

We added the lower row of four modules by using the *Copy* and *Paste* options, copying the upper row. We then modified these modules as needed. We changed the module names so they would be unique, and then opened the new Seize module (*Seize Bay 2* or *3*) and changed the resource set name to *Bays 2* or *3* (the new set we just defined). The next set of changes will take a little explaining!

We now have two Seize modules that are attempting to seize the bay resources. The upper module is able to use all three bays, while the lower module is only able to use Bays 2 or 3. We needed to develop a method to set our priorities in such a way that would result in the most effective use of our bay resources. There are really two levels of prioritization in this model. The first level is the queue ranking. When we changed the name of our new Seize module, *Seize Bay 2* or *3,* we also changed the name of the associated queue, *Bays 2* or *3 Queue*. We now have a queue for each Seize module. Both queues have a ranking of *Lowest Attribute Value* based on the *Priority* attribute. This means that the wait customer service jobs will be at the front of their queues and will be serviced first. The Bay 1 resource can only be allocated by the upper Seize module, so we need to be concerned only with the way we allocate Bays 2 and 3. Since the entities in the lower Seize module can use only Bays 2 or 3, we would like to give that Seize module priority over the upper Seize module.

Let's first explain our logic and then we'll show you how to implement it. We are only concerned about the first entity in each queue and the Bay 2 and Bay 3 resources. If both entities are wait entities and a Bay 2 or Bay 3 resource is available, we want the resource to be allocated to the Job Type 2 entity, because it can only use a Bay 2 or 3 resource and the other entity can use all three. The same logic holds if both entities are customers who will leave their vehicles, Priority 2. While we're doing all this, we always want to service our wait customers first, Priority 1.

We'll do this by developing expressions for the Priority fields in our two Seize modules. The expression for the upper Seize module is

```
    (Priority*10)+1.
```

While the expression for our lower Seize module is

```
    Priority*10.
```

This will result in the following priorities:

Priority 1 and Job Type 1 with a Seize priority of 11,
Priority 2 and Job Type 1 with a Seize priority of21,
Priority 1 and Job Type 2 with a Seize priority of 10, and
Priority 2 and Job Type 2 with a Seize priority of 20.

This prioritization method will give us the results we want. It will basically mean that the Job Type 2 customers generally will be serviced first if they have the same priority. When you watch the final animation, you'll see that this is what happens.

Except for the module names, no changes were required for the following **Assign** and **Delay** modules. The last change we made in this submodel was in the last new Release module, where we changed the resource set name to *Bays 2* or *3*. The final change was to add our new queue, *Bays 2* or *3 Queue*, to the animation.

Although the prioritization logic was a bit contorted, the remaining changes were fairly simple. This takes care of our first problem. Next, we'll deal with the more complicated problem-the customer arrival process.

7.9.2 Modeling the Customer Arrivals

To resolve our second problem; we'll have to modify our customer-arrival process. In Model 7-1, we released all the customer appointments from the appointment queue just before the start of the workday. For this model, we'll need to release only a portion of these customers. We'll then need to add logic to allow for late-arriving customers and finally remove any customers from the appointment queue who did not show up. All of this logic will be added to our Control Logic submodel, as shown in *Figure 7.24*.

Let's start by adding the logic to release only a portion of the customers at the start of the workday. We'll do this with two modifications to the logic included for Model 7-1 (the logic in the first row of modules in *Figure 7.24*). Following the **Create** module, we inserted a new **Assign** module, *Determine On Time Arrivals*. The assignments for this module are shown in *Figure 7.25*. In our first assignment, we determine the number of on-time arrivals and assign it to a new variable, *On Time*. We simply generate a random variable from our uniform distribution between 0.6 and 0.7, multiply it by the number of appointments in the current appointment queue, and round the result to the nearest integer (ANINT function). We then subtract that value from the number of appointments for the current day to get the number of potential late customer arrivals and assign that value to the new variable, *Late*. Finally, we update a new variable, *Total Late,* which will contain the total number of late-arriving customers. You might note that it is possible that some of these customers may not show up for their appointments. We've somewhat arbitrarily chosen to include them in this total.

Our second modification is to the following Signal module. In Model 7-1, we defaulted on the signal Limit, which allowed an infinite number of entities to be released by the Hold module waiting for that signal. In this new model, we entered the variable On Time for the Limit value. This will limit the number of entities that can be released by the Hold module to the value of that variable. If you run this model and watch the animation, you should notice that at the start of each day not all the appointments are released (to see this, you may have to slow the animation). You might also notice that the entities released are the first ones in queue. Their positions in the queue are determined by when they made their appointments. This appeared to be a reasonable assumption.

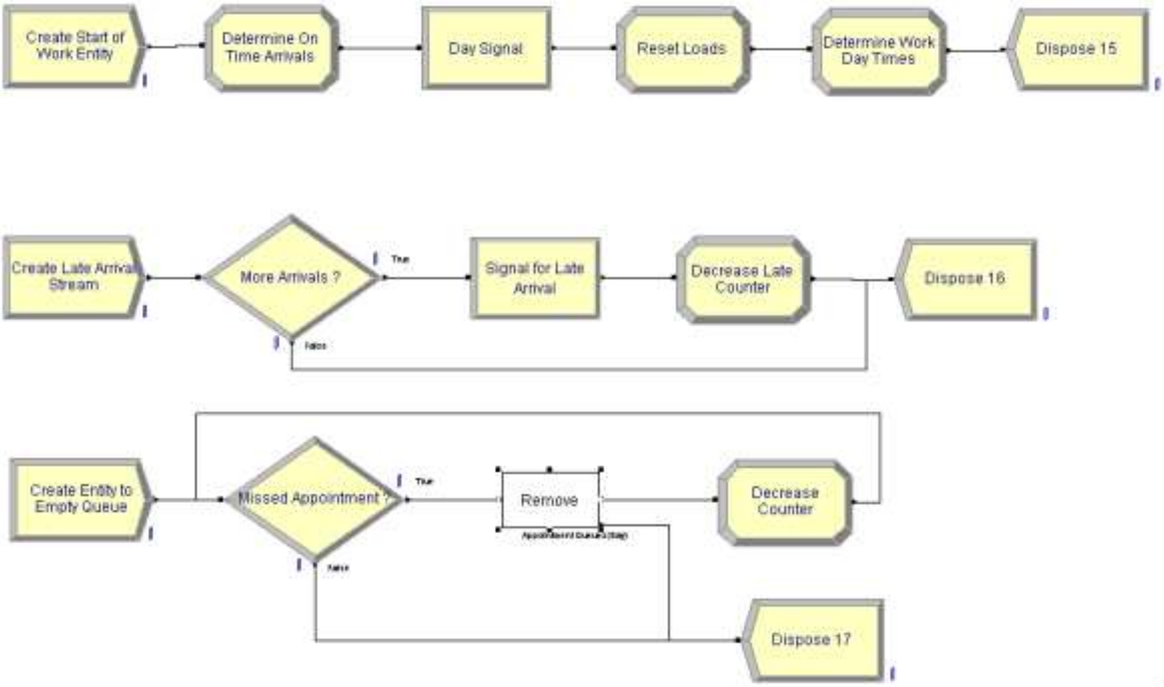
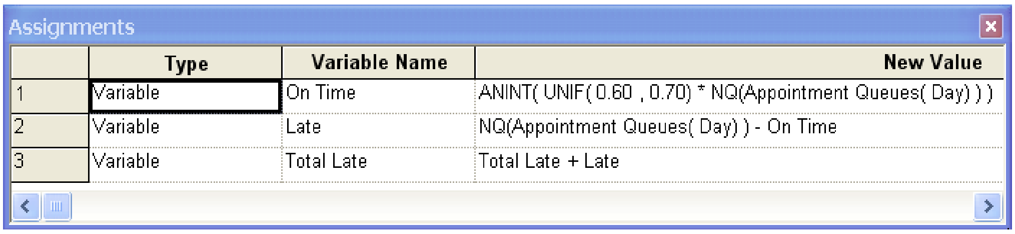


Figure 5·24. The New Control Logic Submodel

Now let's create the logic to allow for the late-arriving customers. This logic can be found in the second row of modules shown in *Figure 7.24*. After analyzing the available data, it was determined that late customers arrive only in the two hours immediately following the start of the workday. The data were analyzed by looking at each of the eight IS-minute periods during these two hours. The arrival rates for these periods were found to be 7, 6, 4, 3, 3, 2, 2, and 1 (in arrivals per hour). First, we need to enter the data into an Arena Schedule. To do so, we added a new schedule, *Late Arrival Schedule*, and selected the *Arrival* option for the Type.


Assignments

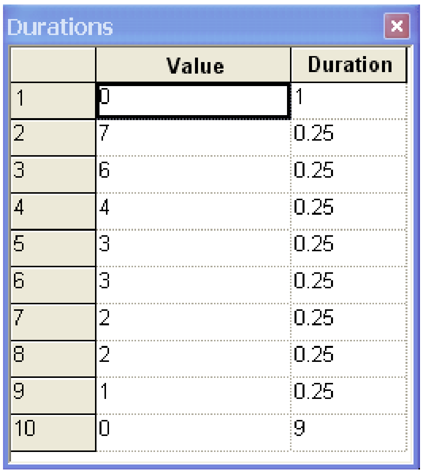
	Type	Variable Name	New Value
1	Variable	On Time	ANINT(UNIF(0.60 , 0.70) * NQ(Appointment Queues(Day)))
2	Variable	Late	NQ(Appointment Queues(Day)) - On Time
3	Variable	Total Late	Total Late + Late

Figure 7.25. The Determine On-Time Arrivals **Assign** module

If you're building this model along with us, be sure that the schedule covers the entire 12 hours of each day, even if you have to add zeroes at the end. If you use the Graphical Schedule Editor, you'll need to open the Options dialog and select 15 Minutes for the Time Slot Duration and enter 48 for the Range (four periods per hour for 12 hours). You can then create your schedule by entering the rates given above for periods five through 12. The remaining periods would have an arrival rate of zero. If you create your schedule in this manner, it may look like it covers the entire 48 time periods, however, it would really only cover the first 12 time periods (the first four periods with zero rate and the following eight periods with the positive rates).

The Graphical Schedule Editor does not assume that each schedule covers one day (they can be of any length). In this case, the schedule would start over in time period 13, which is only three hours into the 12-hour day. We could have checked the box under the Options button to Remain at arrival rate when at the end of the schedule and could have set that capacity to zero, which would have resulted in the correct schedule for the first day. The problem with this option, however, is that we chose to define a single replication of our model as being of length 20 days, which would have resulted in a zero capacity forever after the first day within each replication. Thus, we need to make sure that the last nine-hour time period (or the last 36 of the 15-minute periods) explicitly have a capacity of zero. You can do this by right-clicking on the Durations cell of the schedule and selecting *Edit via Dialog* or *Edit via Spreadsheet*. Selecting Edit via Spreadsheet opens the Durations window in *Figure 7.26*, which allows you to double-click to add a new row and then enter the Value, Duration combinations. For Arrival-type schedules, the default time units are always hours so we entered durations of 0.25 for the eight 15-minute periods when arrivals occur. The last pair of entries (0,9) represents a nine-hour period with a capacity of zero, thus filling out the entire 12 hours of the day explicitly.

Selecting Edit via Dialog opens the Durations dialog box, which allows you to enter the same values. If you try to reopen the Graphical Schedule Editor after adding these data, it will tell you that you have non-integer durations. If you had used the 15-minute durations, you could have viewed the schedule. A word of caution if you do this, be sure to exit the Graphical Schedule Editor without saving the data or the added data for zero capacity at the end will be deleted.



	Value	Duration
1	0	1
2	7	0.25
3	6	0.25
4	4	0.25
5	3	0.25
6	3	0.25
7	2	0.25
8	2	0.25
9	1	0.25
10	0	9

Figure 7.26. The Durations Spreadsheet Window

Name Entity Type	Create Late Arrival Stream Customer Calls
Time Between Arrivals Type Schedule Name	 Schedule Late Arrival Schedule

Display 7.19. Creating the Late Customer Arrivals

Now that we have created our arrival schedule, let's take a look at our logic. The **Create** module, which creates entities (arrivals) according to our previously defined non-stationary Poisson arrival schedule, is described in *Display 7.19*.

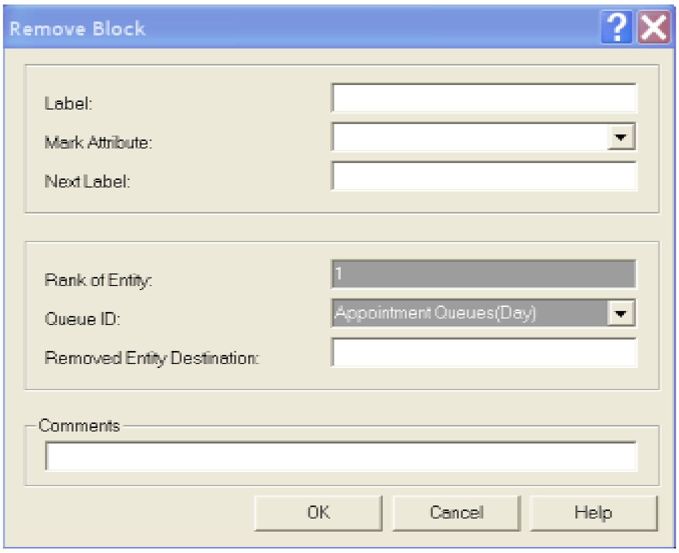
The **Create** module causes zero arrivals for the first hour of each day, arrivals according to the entered rates for hours two and three, and zero arrivals for the last nine hours. In the **Decide** module that follows, we determine whether there are customers who have not yet arrived for the current day by checking the current value of the variable *Late*. If it's positive, additional customers are still expected to arrive. If this expression evaluates to *True*, we send the entity to the next Signal module where we send a Signal Value of the variable *Day* with a limit of

1. This causes the next entity in the appointment queue for the current day to be released for service.

The control entity is then sent to the following **Assign** module where we decrement the value of the variable *Late* by one. This control entity is sent to the last module in this row and disposed. If we create a control entity and all the customers have already arrived (the expression in the **Decide** module evaluates to *False*), we send the entity directly to the **Dispose** module. This completes all the logic for our late-arriving customers.

We're not done yet, because it's possible that customer entities remain in the current day's appointment queue that did not arrive. We added the last row of logic to take care of this situation. The first module creates a control entity three hours into the first day and one entity every 12 hours thereafter. That entity then enters the following **Decide** module to check for a positive value of the variable *Late*. If so, it means that entities remain in the current day's appointment queue (we could have also checked for the number in the queue using the Arena variable *NQ*). Any remaining entities are assumed to be customers who did not show up for their appointments. If the expression evaluates to True, we send the entity to the next Remove module.

The Remove module removes an entity from a queue and sends it to another place in the model. It requires you to identify the entity to be removed by entering the queue identifier and the rank (position in the queue, with I being the front) of that entity. If you attempt to remove an entity from an undefined queue or to remove an entity with a rank that is greater than the number of entities in the specified queue, Arena terminates the run with an error. Although there are two Remove modules available in Arena, one in the Advanced Process panel and the other in the Blocks panel, we need the module from the Blocks panel because we need to enter an expression for the Queue ID that represents a specific queue from the set *Appointment Queues*. The Remove module from the Advanced Process panel allows you to enter an expression, but it results in an error when you check or try to run your model.



Rank of Entity: 1

Queue ID: Appointment Queues(Day)

Removed Entity Destination:

Rank of Entity Queue ID	1 Appointment Queues (Day)

Display 7.20. The Remove Module from the Blocks Panel

The Remove module for our logic is shown in *Display 7.20*. We entered a value of 1 for the Rank of Entity and the expression *Appointment Queues* (*Day*) for the Queue ID. The entity that is removed from the specified queue will be sent out of the lower exit point on the right side of the module. This removed entity will be sent to the **Dispose** module, where it will exit the system. The original control entity will be sent out of the upper exit point to the following **Assign** module.

In the **Assign** module, we decrement the variable *Late* by 1 and increment the new variable *Total Missed* by 1. This variable represents the total number of customers who did not arrive for their scheduled appointments. We probably should have also decremented our variable *Total Late* by 1. In our current model, the *Total Late* variable also includes the customers who did not show up for their appointments. Somehow we missed this small detail. That shows you that we're not always perfect in our model building (most of the time, but not always). We could easily have changed this by subtracting *Total Missed* from *Total Late* at the end of the simulation run. Since we are not using these values as part of our key performance measures, we didn't bother to make this change. The control entity is sent back to the **Decide** module where it checks to see whether there are any remaining customers in the appointment queue for the current day.

This completes the logic changes for our Control Logic submodel. We did make one additional change to our model; we added two outputs to our Statistic data module to output the average number of late appointments per day and the average number of customers per day who did not arrive for their appointments.

Output			Minimum	Maximum
	Average	Half Width	Average	Average
Daily Actual Time	23.2778	1.38	20.7947	26.3750
Daily Book Time	22.6435	0.21	22.0283	23.0492
Daily Late	6.2500	0.05	6.1500	6.3500
Daily Late Wait Jobs	0.6400	0.16	0.2500	1.0500
Daily Missed	0.7650	0.17	0.5000	1.3000
Daily Overtime	1.6332	0.36	0.7933	2.5454
Daily Profit	479.89	50.20	388.71	574.67

Figure 7.27. The Model 7-2 Automotive Shop Output Statistics

If you run this new model and check the results against those from Model 7-1, you'll see that our bay resources were utilized approximately 7.46 hours per day (compared to approximately 7.8 hours for Model 7-1). This should be expected as we have an average of 0.765 customers per day who did not show up for their appointments, which accounts for the approximately 0.34 fewer hours in our Model 7-2. The remaining results are shown in *Figure 7.27*.

If you compare these results to those from Model 7-1, shown in *Figure 7.22*, you'll see that both our daily overtime average and our daily profits are lower from this new model. Again, since we had customers who did not show up for their appointments, this appears to be correct.

This completes the model development for our automotive shop model. We'll use Model 7-2 as the basis for our statistical analysis of terminating simulations in Chapter 6.

7.10 Model 7-3: An (s, S) Inventory Simulation

We close this chapter by considering a completely different kind of system, an inventory, to indicate how such operations might be modeled and to illustrate the breadth of simulation in general and of Arena in particular. (It's not just for queueing type models As pointed out by the late Carl M. Harris, "queueing" is evidently the only word in English with five or more consecutive vowels; see Gass and Gross (2000).) We'll use modules from the **Blocks** and **Elements** panels only, primarily to demonstrate their use and make you aware that they're there if you need them (so, in effect, we're using the SIMAN simulation language). We will you recreate this model using the higher-level modeling constructs of the **Basic Process** and **Advanced Process** panels in Chapter 6. This model is essentially the same as the one in Section 1.5 of Law and Kelton (2000).

7.10.1 System Description

Widgets by Bucky, a multi-national holding company, carries inventory of one kind of item (of course, they're called widgets). Widgets are indivisible, so the inventory level must always be an integer, which we'll denote as $I(t)$ where t is time (in days) past the beginning of the simulation. Initially, 60 widgets are on hand: $I(0) = 60$.

Customers arrive with interarrival times distributed as exponential with mean 0.1 day (it's a round-the-clock operation), with the first arrival occurring not at time zero but after one of these interarrival times past time zero. Customers demand 1, 2, 3, or 4 widgets with respective probabilities 0.167, 0.333, 0.333, and 0.167. If a customer's demand can be met out of on-hand inventory, the customer gets the full demand and goes away happy. But if the on-hand inventory is less than the customer's demand, the customer gets whatever is on hand (which might be nothing), and the rest of the demand is **backlogged** and the customer gets it later when inventory will have been sufficiently replenished; this is kept track of by allowing the inventory level $I(t)$ to go negative, which makes no sense physically but is a convenient accounting artifice. Customers with backlogged items are infinitely patient and never cancel their orders. If the inventory level is already negative (i.e., we're already in backlog) and more customers arrive with demands, it just goes more negative. We don't keep track of specifically which widgets arriving in the future will satisfy which backlogged customers (they're also infinitely polite, so it doesn't matter).

At the beginning of each day (including at time zero, the beginning of day 1), Bucky "takes inventory" to decide whether to place an order with the widget supplier at that time. If the inventory level (be it positive or negative) is (strictly) less than a constant s (we'll use s=20), Bucky orders "up to" another constant S (we'll use S= 40). What this means is that he orders a quantity of widgets so that, if they were to arrive instantly, the inventory level would pop up to exactly S. So if t is an integer and thus $I(t)$ is the inventory level at the beginning of a day (could be positive, negative, or zero) and $I(t)<s$, Bucky orders $S–I(t)$ items; if $I(t)\leq s$, Bucky does nothing, lets the day go by, and checks again at the beginning of the next day, that is, at time t+1. Due to the form of this review/replenishment policy, systems like this are often called (s, S) **inventory models**.

However, an order placed at the beginning of a day does not arrive instantly, but rather sometime during the last half of that day, after a **delivery lag** (a.k.a. **lead time**) distributed uniformly between 0.5 and 1 day. So when the order arrives, the inventory level will pop up by an amount equal to the original order quantity but, if there were any demands since the order was placed, it will pop up to something less than S when the order is finally delivered. Note that the relative timings of the inventory evaluations and delivery lags are such that there can never be more than one order on the way, since an order placed at the beginning of a day will arrive, at the very latest, just before the end of that day, which is the beginning of the next day, the first opportunity to place another order; see Exercise 7-18 for modeling implications of this particular numerical situation.

Bucky is interested in the average total operating cost per day of this system over 120 days, which will be the sum of three components:

Average ordering cost per day. Every time an order is placed, there's a cost incurred of $32 regardless of the order quantity, plus $3 per item ordered; if no order is placed there's no ordering cost, not even the fixed cost of $32. The $3 is not the (wholesale) price of a widget, but rather the administrative operational cost to Bucky of ordering a widget (we're not considering prices at all in this model). At the end of the 120-day simulation, the total of all the ordering costs accrued is divided by 120 to get the average ordering cost per day.

Average holding cost per day. Whenever there are items actually physically in inventory (i.e., *I*(*t*)>0), a holding cost is incurred of $1 per widget per day. The total holding cost is thus

$$1\int_0^{20} \max(I(t),0)\,dt$$

(think about it), and the average holding cost per day is this total divided by the length of the simulation, 120 days.

Average shortage cost per day. Whenever we're in backlog (i.e., *I*(*t*)<0), a shortage cost of $5 per widget per day is incurred, a harsher penalty than holding positive inventory. The total shortage cost is thus

$$5\int_0^{20} \max(-I(t),0)\,dt$$

(think about it a little harder), and the average shortage cost per day is this total divided by the simulation length.

Note that for periods when we have neither backlog nor items physically in inventory (i.e., *I*(*t*) =0) there is neither shortage nor holding cost - cost-accountant nirvana. Also, you might notice that we're not accounting for the wholesale or retail price of the widgets anywhere; in this model, we assume that these prices are fixed and induce this demand, which will happen regardless, so the revenues and profits are fixed and it's only the operating cost that we can try to affect.

One final fine point before we build our (rather simple) simulation. Inventory evaluations occur at the beginning of each day i.e., when the clock is an integer, and any ordering cost is incurred at that time. It happens that the run is supposed to end at an integer time (120), so normally there would be an inventory evaluation then, and possibly an order placed .that would not arrive until after the end of the world, so we'd never get it but we'd have to pay the ordering cost. So we should prevent an inventory evaluation from happening at time 120, which we'll do by actually stopping the run at time 119.9999.

7.10.2 Simulation Model

As we mentioned earlier, we'll build this model using only modules from the Blocks and Elements panels. It would have been a lot easier to do this with the higher-level (and less ancient) modules from the Basic Process and Advanced Process panels, but we leave-that joy to you.

Figure 7.28 Shows the completed model, including the animation at the top right. The modules from the Blocks and Elements panels at the bottom are divided into three sections, as indicated by the outline boxes behind the modules.

Let's start with the data structure for the model. The modules in the bottom section of *Figure 7.28* are from the **Elements** panel, so are themselves called **elements**. Note that they're not connected to anything, since they define various objects for the whole model. As we go through these, you'll already be familiar with many of the terms, since many of the same constructs and functions in the higher levels of Arena are available here as well.

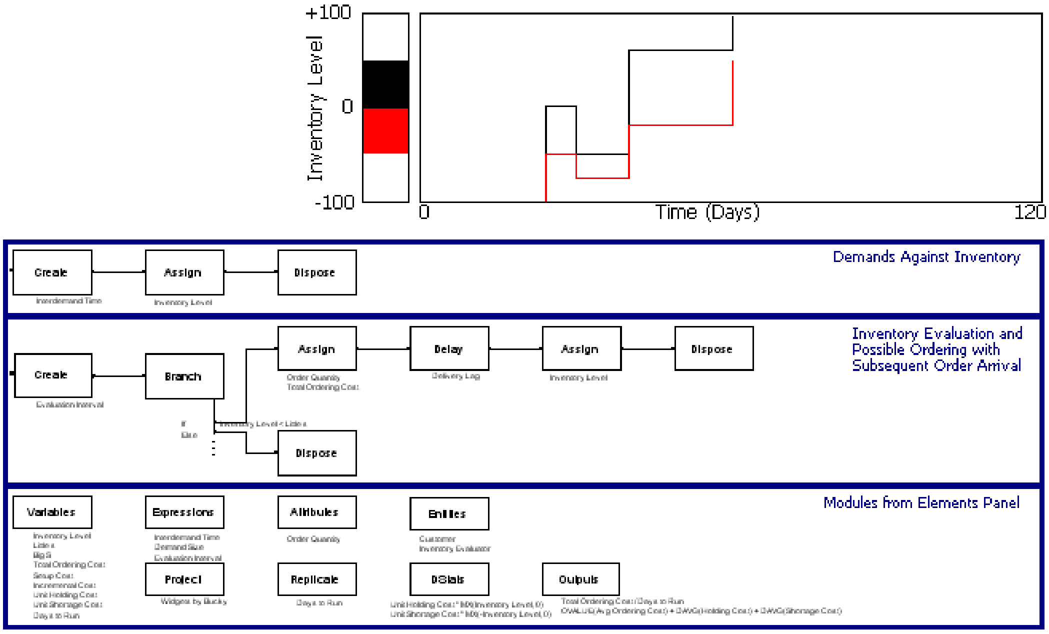
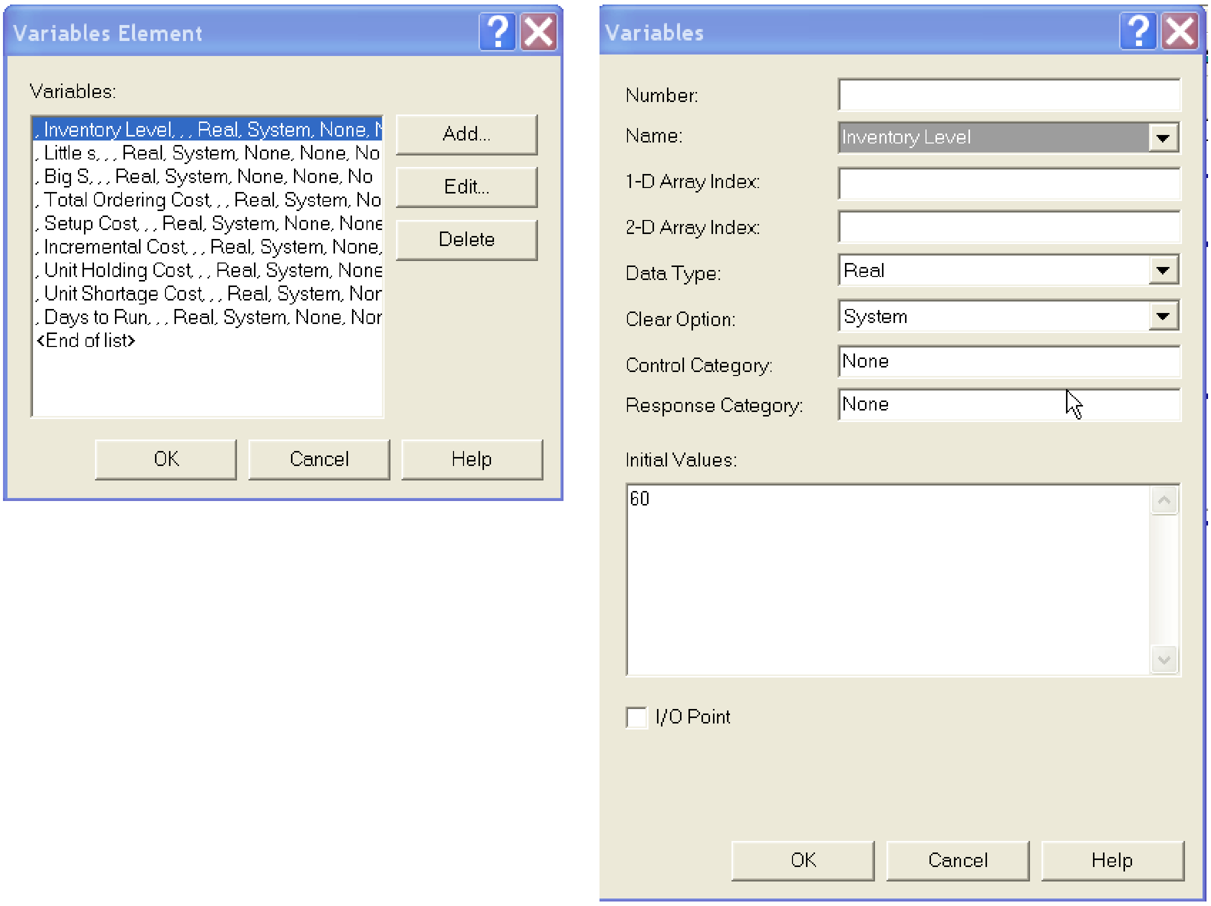


Figure 7.28. The Completed Inventory Model


Variables Element

Variables:

, Inventory Level, , , Real, System, None, N
, Little s, , , Real, System, None, None, No
, Big S, , , Real, System, None, None, No
, Total Ordering Cost, , , Real, System, No
, Setup Cost, , , Real, System, None, None
, Incremental Cost, , , Real, System, None,
, Unit Holding Cost, , , Real, System, None
, Unit Shortage Cost, , , Real, System, Nor
, Days to Run, , , Real, System, None, Nor
<End of list>

Variables

Number:
Name: Inventory Level
1-D Array Index:
2-D Array Index:
Data Type: Real
Clear Option: System
Control Category: None
Response Category: None
Initial Values:
60

□ I/O Point

Figure 7.29. The Variables Element

The Variables element, shown in its completed form with the entry for *Inventory Level* visible in *Figure 7.29*, defines and optionally initializes Arena variables (much like the Variable module from the Basic Process panel). If there's no initialization for a variable, it defaults to zero. We'll let you poke through the entries in this module on your own, which are defined as follows:

Inventory Level: At any time in the simulation, this is the inventory level (positive, zero, or negative), and is initialized to 60 here. This variable is the function $I(t)$.

Little s: This is *s*, initialized to 20.

Big S: This is *S*, initialized to 40.

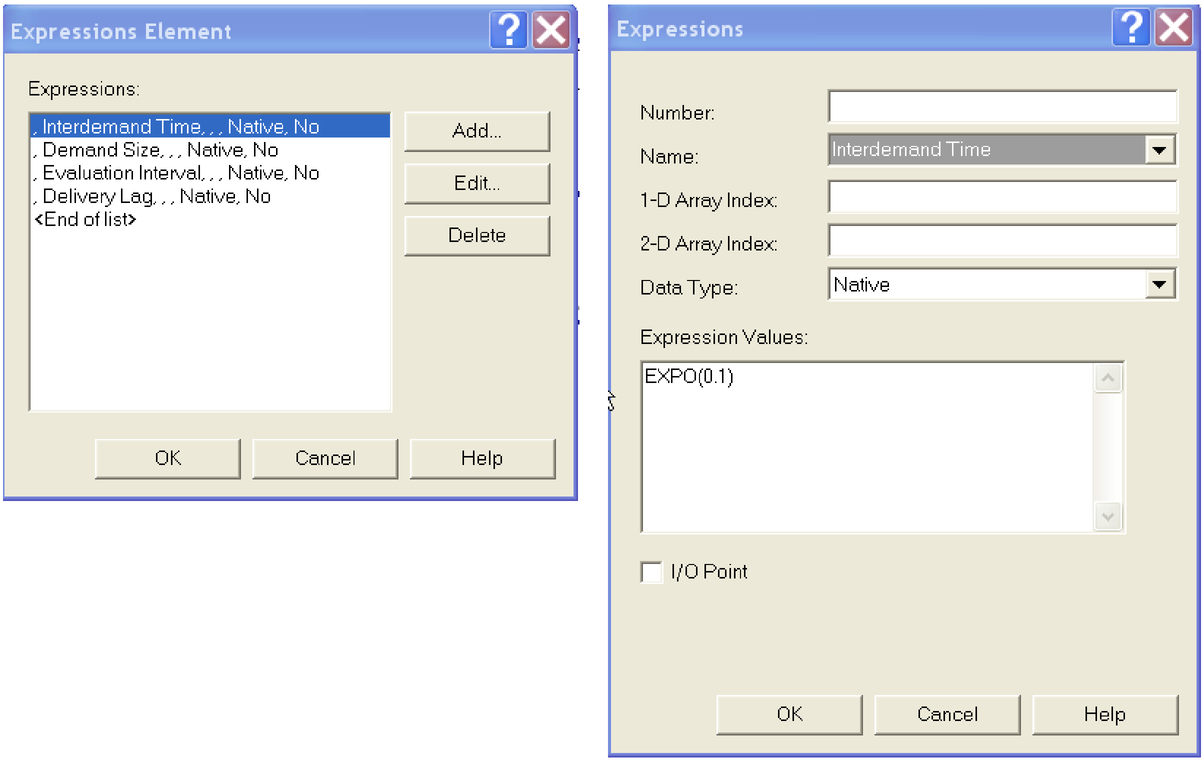
Total Ordering Cost: A statistical-accumulator variable to which all ordering costs are added; not initialized (so is implicitly initialized to 0).

Setup Cost: The fixed cost of ordering, initialized to 32.

Incremental Cost: The variable (per-widget) ordering cost, initialized to 3,

Unit Holding Cost: The cost of holding one widget in inventory for one day, initialized to 1.

Unit Shortage Cost: The cost of having one widget in backlog for one day, initialized to 5.



Expressions:

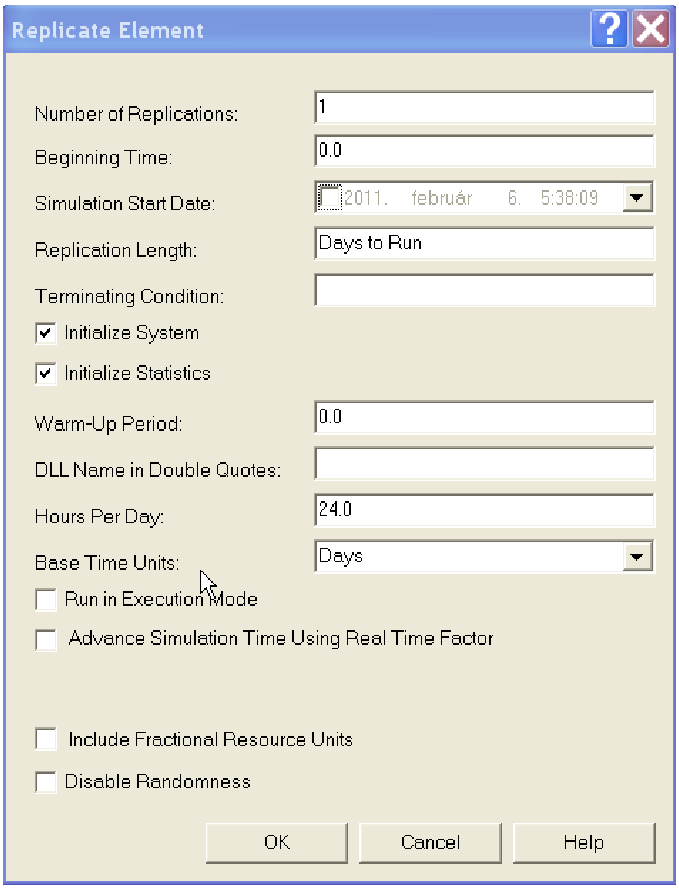
, Interdemand Time, , , Native, No
, Demand Size, , , Native, No
, Evaluation Interval, , , Native, No
, Delivery Lag, , , Native, No
<End of list>

Number:
Name: Interdemand Time
1-D Array Index:
2-D Array Index:
Data Type: Native
Expression Values:
EXPO(0.1)
□ I/O Point

Figure 7.30. The Expressions Element

We defined four Expressions via the Expressions element, in *Figure 7.30* with the entry for *Demand Size* visible. These Expressions define the probability distributions for the quantities indicated by their names and should be self-explanatory as you look through them. We decided to define *Evaluation Interval* as an Expression rather than a variable, even though it's just a constant (l), for reasons of model generality should we want to try out some kind of random evaluation interval in the future (but, as seen in Exercise 6-12, there's also an advantage to defining it as a Variable).

The Attributes element has only one entry, used to declare that *Order Quantity* is an attribute to be attached to entities. The Entities element declares the two types of entities we'll be using, *Customer* and *Inventory Evaluator*. The Project element allows you to specify a Title, Analyst Name, and other documentation, as well as control some of the reporting, similar to the *Run>Setup>Project Parameters* option we've used before. There's not much to see in these modules, so we'll skip the figures and let you take a look at them on your own.



Number of Replications: 1

Beginning Time: 0.0

Simulation Start Date: □ 2011. február 6. 5:38:09

Replication Length: Days to Run

Terminating Condition:

☑ Initialize System

☑ Initialize Statistics

Warm-Up Period: 0.0

DLL Name in Double Quotes:

Hours Per Day: 24.0

Base Time Units: Days

□ Run in Execution Mode

□ Advance Simulation Time Using Real Time Factor

□ Include Fractional Resource Units

□ Disable Randomness

Figure 7.31. The Replicate Element

The Replicate element, shown in *Figure 7.31*, basically replicates what's available via *Run> Setup> Replication Parameters*. We have only two non-default entries. First, we changed the Base Time Units to Days, since all of our input time measurements are in days and the blocks from the Blocks panel have no provision for specifying input time units, assuming that they're all in the Base Time Units. Second, we specified the Replication Length to be *Days to Run*, the Variable we initialized to 119.9999, our cheap fudge to avoid the useless inventory evaluation at time 120.

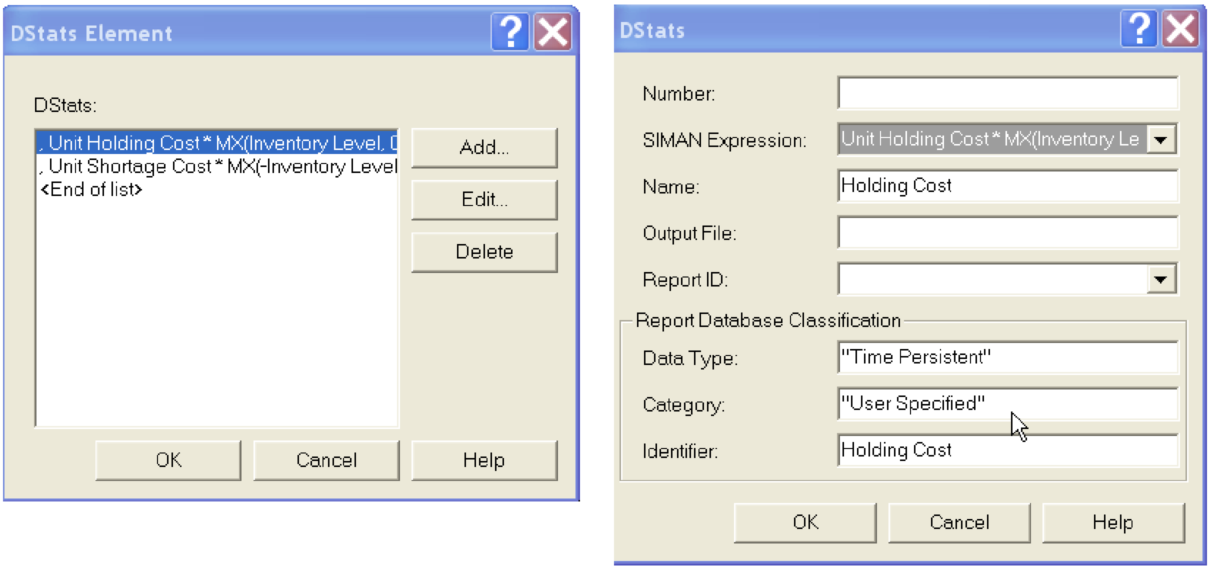
The DStats element, in *Figure 7.32* with the *Holding Cost* entry visible, does what Statistic module entries of type Time-Persistent do, in this case, saying that we want to accumulate the time-persistent values for the holding and shortage costs. The partly hidden SIMAN Expression in *Figure 7.32* is

```
Unit Holding Cost * MX(Inventory Level,0)
```

and is the instantaneous holding cost to be charged whenever *Inventory Level* is positive (recall that MX is the built-in Arena function that returns the maximum of its arguments). Similarly, the other entry in the DStats element, with *Name Shortage Cost*, accumulates the SIMAN Expression

```
Unit Shortage Cost * MX(-Inventory Level, 0)
```

for the shortage cost. What we want are the time-averages of these values, which we'll get next.



DStats Element

DStats:

, Unit Holding Cost * MX(Inventory Level, 0
, Unit Shortage Cost * MX(-Inventory Level
<End of list>

DStats

Number:

SIMAN Expression: Unit Holding Cost * MX(Inventory Le

Name: Holding Cost

Output File:

Report ID:

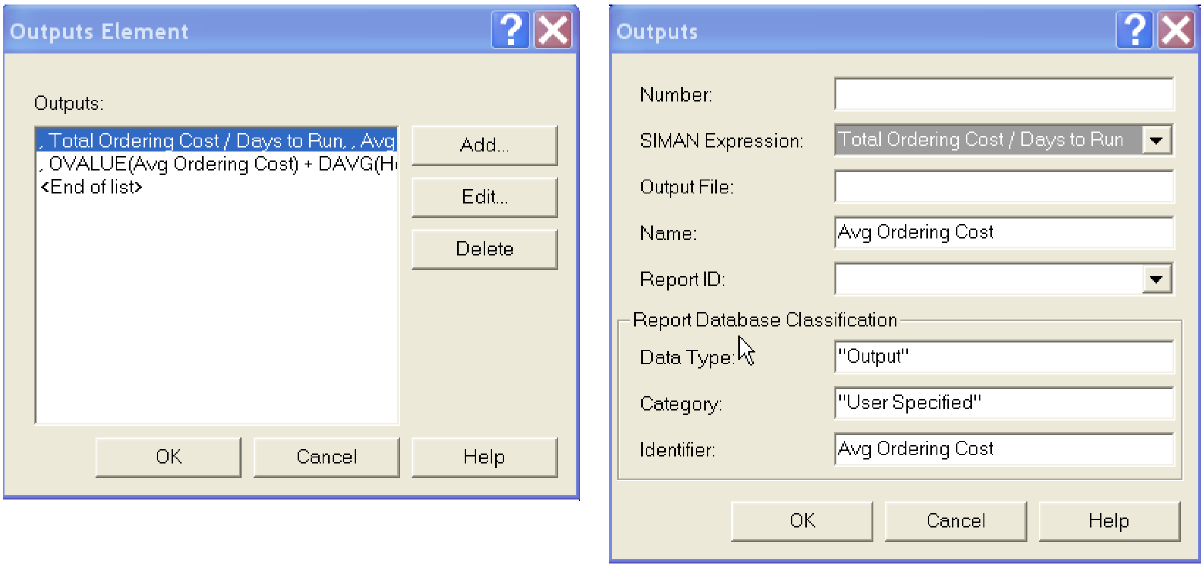
Report Database Classification

Data Type: "Time Persistent"

Category: "User Specified"

Identifier: Holding Cost

Figure 7.32. The DStats Element



Outputs Element

Outputs:

, Total Ordering Cost / Days to Run, , Avg
, OVALUE(Avg Ordering Cost) + DAVG(H
<End of list>

Outputs

Number:

SIMAN Expression: Total Ordering Cost / Days to Run

Output File:

Name: Avg Ordering Cost

Report ID:

Report Database Classification

Data Type: "Output"

Category: "User Specified"

Identifier: Avg Ordering Cost

Figure 7.33. The Outputs Element

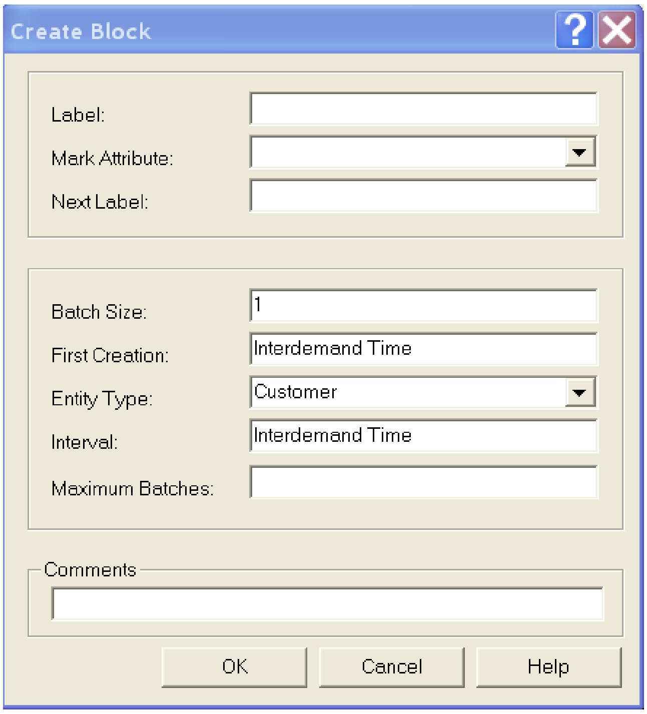
The Outputs element of *Figure 7.33* does what Statistic module entries of type Output do, and here we need to do two things. First (and visible in *Figure 7.33*), we'll get the average ordering cost per day by dividing *Total Ordering Cost* by *Days to Run*, the length of the run, and storing this in the output name *Avg Ordering Cost*. Second, we add up all three components of the overall output performance measure into what we call *Avg Total Cost*, via the expression

```
OVALUE(Avg Ordering Cost)+DAVG(Holding Cost)+DAVG(Shortage Cost)
```

The Arena function OVAL.UE returns the last (most recent) value of its argument, which in this case is what we defined in the preceding line of this module (we could have avoided this by entering *Total Ordering Cost*/*Days to Run* here instead, but doing it the way we did produces a separate output for the average-ordering-cost component of average total cost, which might be of interest on its own). And the Arena function DAVG returns the time-persistent average of its argument, so is used twice here to get the average daily holding and shortage costs (also produced separately in the reports).

Now let's turn to the logic modules from the Blocks panel, which are themselves called **blocks**. The top group of blocks in *Figure 7.28* represents customers arriving, making demands against the inventory, and leaving.

The **Create** block, shown in *Figure 7.34*, required three non-default entries. For both First Creation and Interval, we entered *Interdemand Time*, defined in the Expressions element as EXPO(0.1) . The Entity Type here we defined as *Customer*.



Label:

Mark Attribute:

Next Label:

Batch Size: 1

First Creation: Interdemand Time

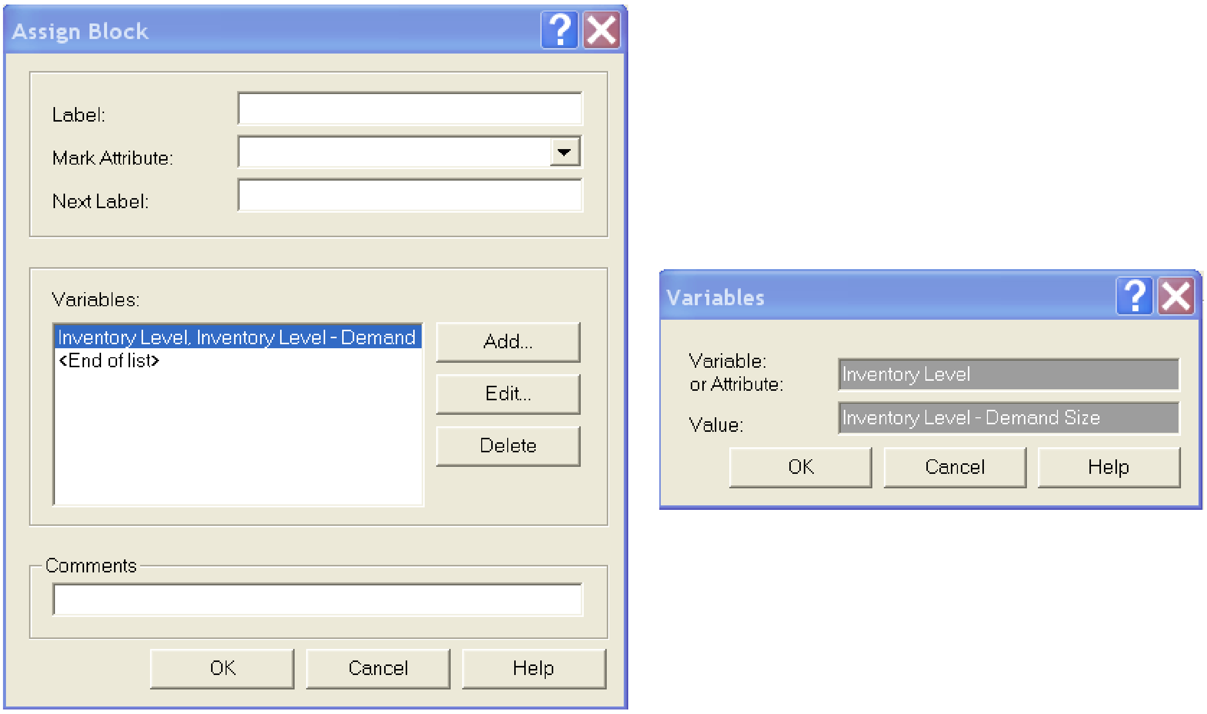
Entity Type: Customer

Interval: Interdemand Time

Maximum Batches:

Comments

Figure 7.34. The **Create** Block for Customer Arrivals



Label:

Mark Attribute:

Next Label:

Variables:

Inventory Level, Inventory Level - Demand
<End of list>

Comments

Variable: or Attribute: Inventory Level

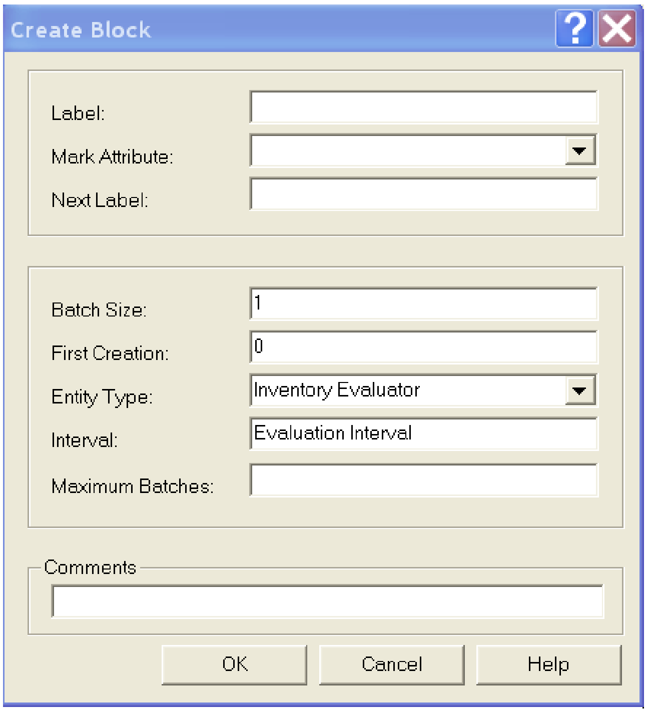
Value: Inventory Level - Demand Size

Figure 7.35. The **Assign** Block for Customer Demands Against Inventory

The next **Assign** block, in *Figure 7.35*, decrements the customer's demand from the *Inventory Level*. The Expression *Demand Size* was defined in the Expressions module as a discrete random variable to return demands of 1, 2, 3, or 4 with the appropriate probabilities.

The customer entity then goes away via the **Dispose** block, where the only thing we did was clear the box to Record Entity Statistics (we don't care about them in this model).

The center group of blocks in *Figure 7.28* represent the periodic inventory evaluation and decision about ordering. If an order is placed, we wait for it to arrive and then increment the inventory level; if no order is placed, there's nothing to do.



Label:

Mark Attribute:

Next Label:

Batch Size: 1

First Creation: 0

Entity Type: Inventory Evaluator

Interval: Evaluation Interval

Maximum Batches:

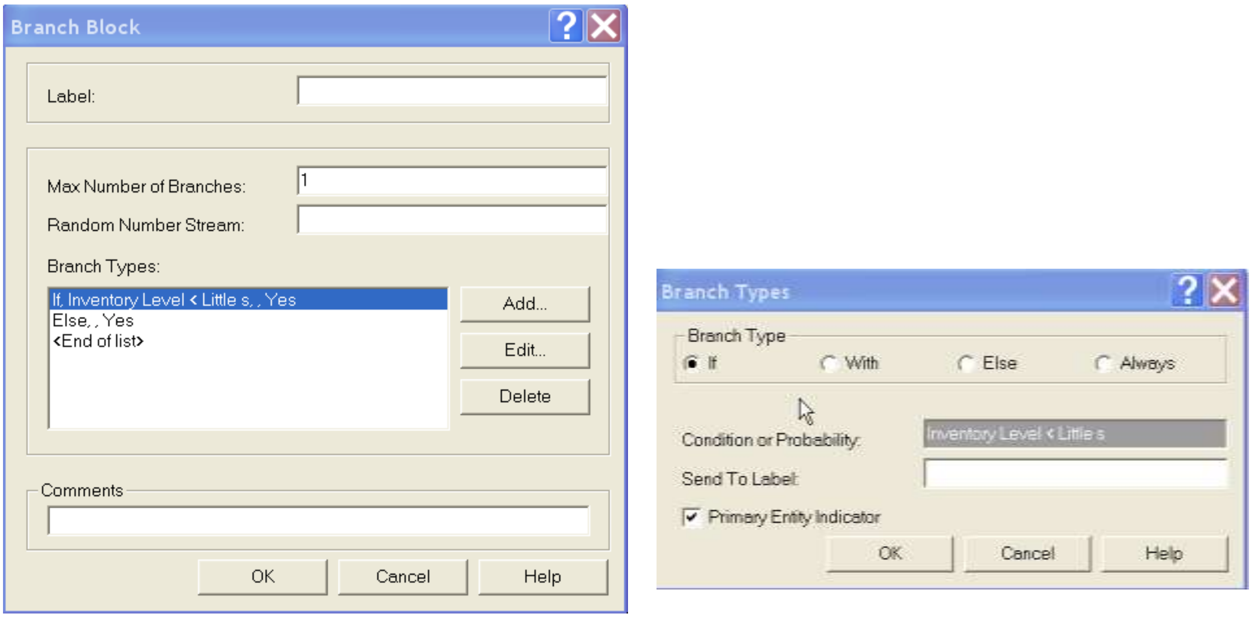
Comments

Figure 7.36. The **Create** Block for Inventory Evaluator

This logic starts with a **Create** block, in *Figure 7.36*, to insert into the model what you can think of as a widget (not bean) counter who will count the widgets, decide whether to place an order, and then, if an order is placed, wait around for it to be delivered, and then put the widgets on the shelf. We want the First Creation to be at time 0 since our system calls for an inventory evaluation at the beginning of the run. The Entity Type is *Inventory Evaluator*, and the Interval of time between successive creations is the Expression *Evaluation Interval*, which was specified to be the constant 1 in the Expressions element.

Once created, the *Inventory Evaluator* entity proceeds to the Branch block, in *Figure 7.37*, which does some of the same things as the Basic Process panel's **Decide** module, with which you're already familiar. In this case, we want to decide whether to place an order at this time. First we Add a branch of type. If with the condition *Inventory Level<Little s*, which, if true, indicates that we want to place an order now (the dialog box for this branch is visible in *Figure 7.37*). The other branch is of type Else and is the only other possibility, indicating that no order is to be placed right now. The exit points from the Branch block correspond to truth of the corresponding branches; in this case, the first exit point is followed if we are to place an order (and leads to the **Assign** block to the right of and just above the Branch block), and the second exit point corresponds to the Else branch and means that nothing is to be done (so the entity immediately goes to the **Dispose** block to the right of and just below the Branch block). An important part of the Branch block is the Max Number of Branches field, which defaults to infinity (and which we set instead to 1); in general, the Branch block evaluates each branch

in the list in sequence, and sends the incoming entity (or a duplicate of it) out through each branch that evaluates to *True*, until Max Number of Branches is used up. This is a powerful capability but one that can lead to errors if the entries in the block are not made and coordinated with care.



Branch Block

Label:

Max Number of Branches: 1

Random Number Stream:

Branch Types:

If, Inventory Level < Little s, , Yes
Else, , Yes
<End of list>

Comments

Branch Types

Branch Type: ● If ○ With ○ Else ○ Always

Condition or Probability: Inventory Level < Little s

Send To Label:

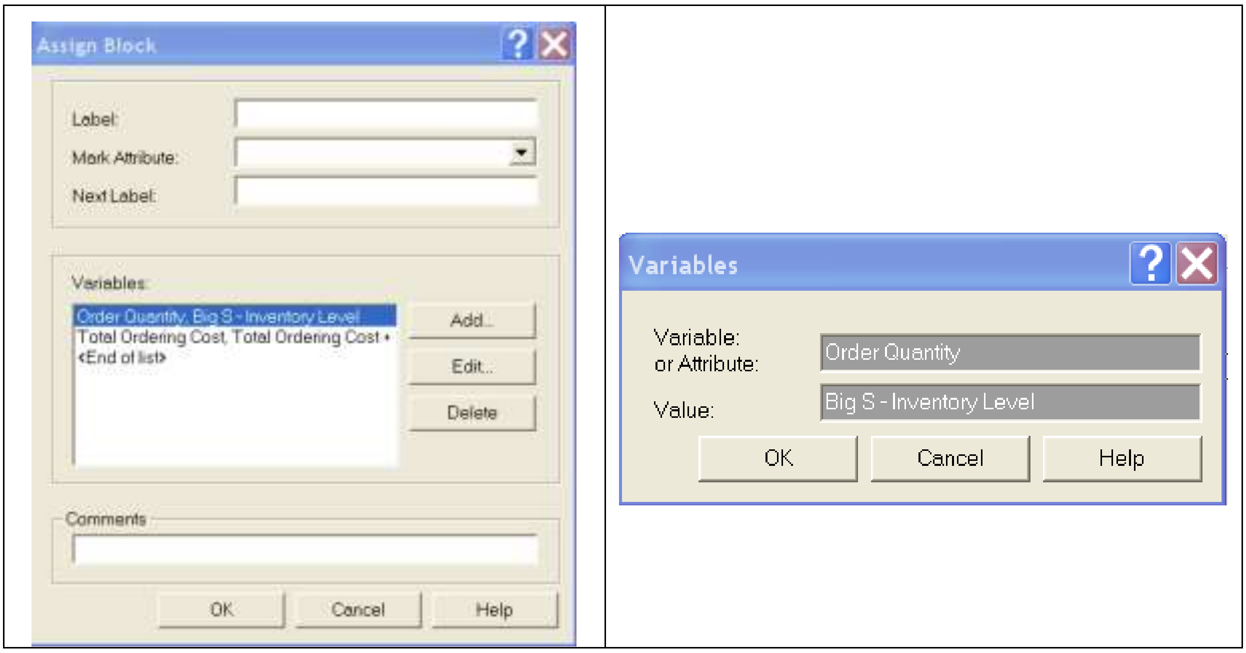
☑ Primary Entity Indicator

Figure 7.37. The Branch Block

If we need to place an order, the *Inventory Evaluator* entity next goes to the following **Assign** block, seen in *Figure 7.38*, which first computes the *Order Quantity* (an attribute attached to the entity) as *Big S–Inventory Level*. The next assignment in this block incurs the ordering cost for this order by replacing the Variable *Total Ordering Cost* by the expression

```
Total Ordering Cost + Setup Cost + Incremental Cost * Order Quantity
```

Note that it was important to do the assignments here in this order, since the result of the first is used in the second.



Assign Block

Label:

Mark Attribute:

Next Label:

Variables:

Order Quantity, Big S - Inventory Level
Total Ordering Cost, Total Ordering Cost +
<End of list>

Comments

Variables

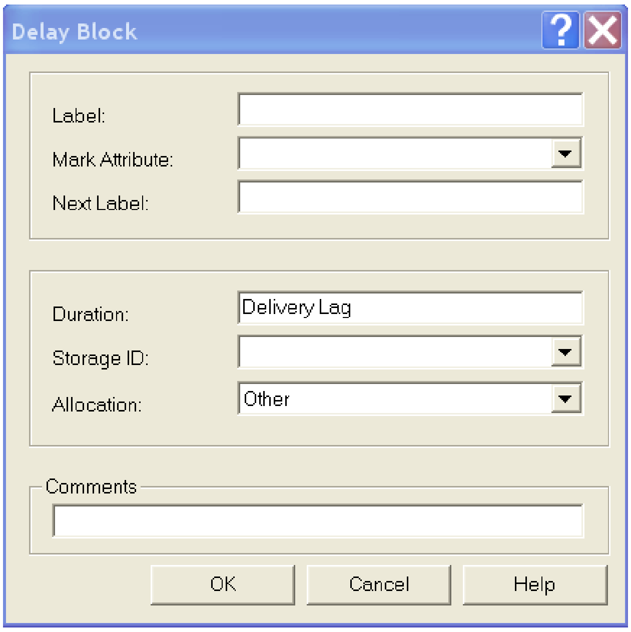
Variable or Attribute: Order Quantity

Value: Big S - Inventory Level

Figure 7.38. The **Assign** Block to Place an Order

Now it's time to wait for the order to arrive, which is accomplished by sending the *Inventory Evaluator* entity to the Delay block in *Figure 7.39*, where it simply sits for *Delivery Lag* time units (remember, only the Base Time Units are available in blocks). Recall that *Delivery Lag* was defined in the Expressions element to be UNIF (0.5, 1.0).

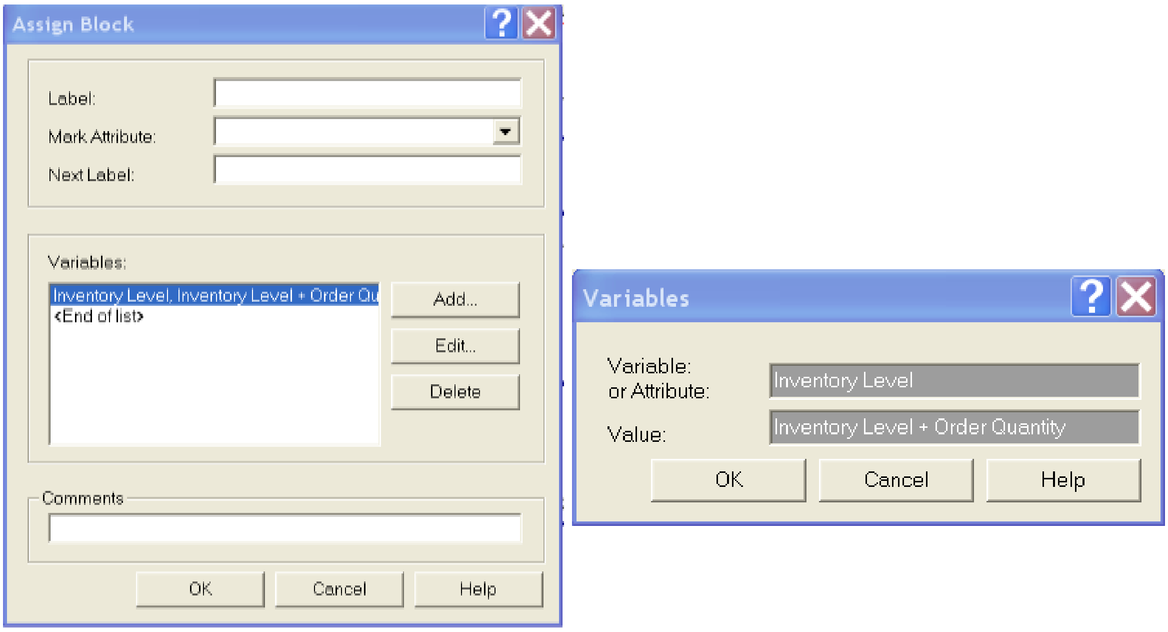
After the delivery lag, the *Inventory Evaluator* goes to the next **Assign** block, in *Figure 7.40*, where it increments *Inventory Level* by its attribute *Order Quantity*.



Duration: Delivery Lag

Allocation: Other

Figure 7.39. The Delay Block for the Delivery Lag



Variables:
Inventory Level, Inventory Level + Order Qu
<End of list>

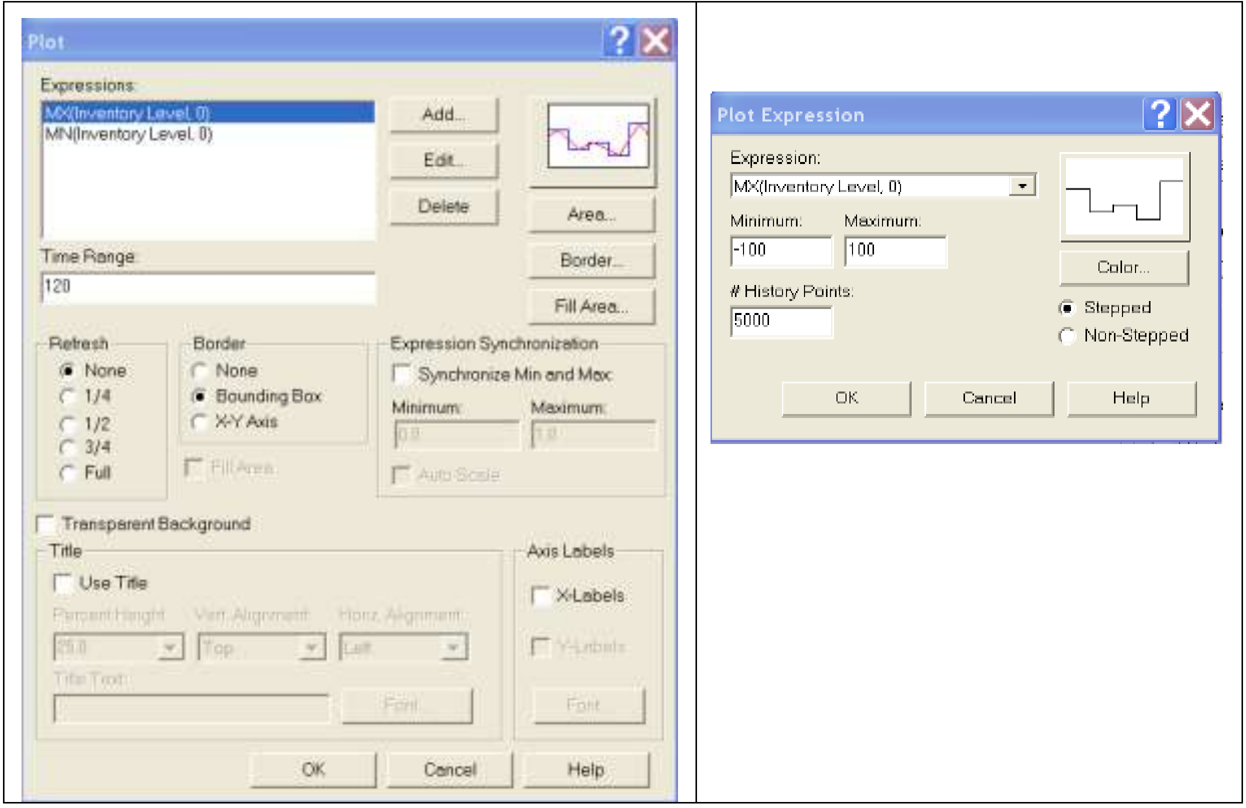
Variable: or Attribute: Inventory Level

Value: Inventory Level + Order Quantity

Figure 7.40. The **Assign** Block for the Order Arrival

After the order is delivered, this *Inventory Evaluator's* job is done, so it goes to the final **Dispose** block (don't be sad, though, because another one of these entities will be created soon).

Now let's add a little (just a little) animation. Our Plot dialog box is shown in *Figure 7.41*, which graphs the *Inventory Level* over time. We'd like to have different colors depending on whether the *Inventory Level* is positive (in the black) or negative (in the red), which we accomplish by plotting two separate curves. To show positive inventory levels, we'll plot the expression *MX*(*Inventory Level*, 0) in black,. and for negative inventory levels, we'll plot *MN* (*Inventory Level*, 0), which will be a negative value, in red (*MN* is Arena's built-in function to return the minimum of its arguments). Note that when 0 "wins" in either of these plots, we'll get a flat line at zero, which maybe isn't so bad since it will visually delineate where zero is (in a playful, multicolor fashion).



Plot

Expressions:
MX(Inventory Level, 0)
MN(Inventory Level, 0)

Time Range: 120

Refresh: ● None ○ 1/4 ○ 1/2 ○ 3/4 ○ Full

Border: ○ None ● Bounding Box ○ X-Y Axis

□ Fill Area

Expression Synchronization: □ Synchronize Min and Max; Minimum: 0.0; Maximum: 1.0; □ Auto Scale

□ Transparent Background

Title: □ Use Title; Percent Height: 25.0; Vert. Alignment: Top; Horiz. Alignment: Left; Title Text:

Axis Labels: □ X-Labels □ Y-Labels

Plot Expression

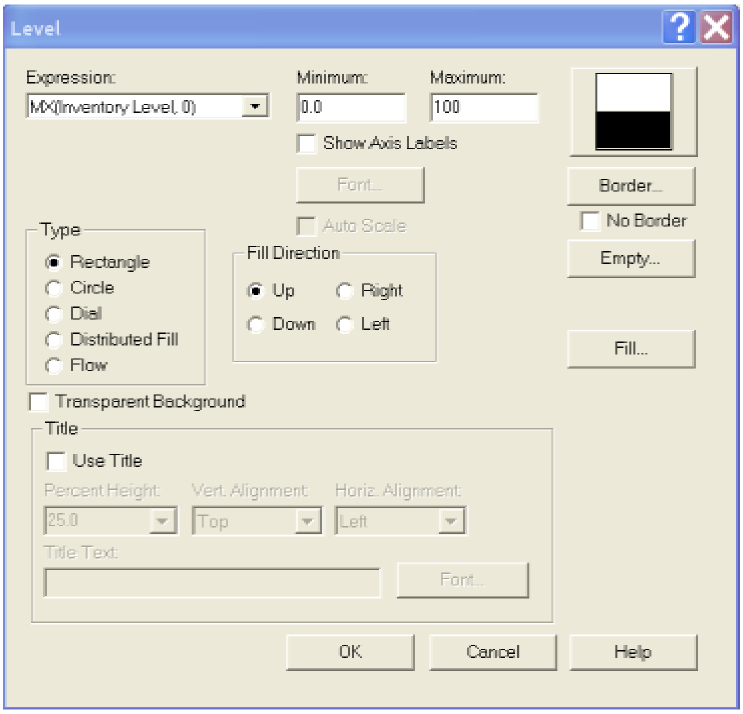
Expression: MX(Inventory Level, 0)

Minimum: -100 Maximum: 100

History Points: 5000

● Stepped ○ Non-Stepped

Figure 7.41. The Plot Dialog Box



Level

Expression: MX(Inventory Level, 0)

Minimum: 0.0 Maximum: 100

□ Show Axis Labels

□ Auto Scale

□ No Border

Type: ● Rectangle ○ Circle ○ Dial ○ Distributed Fill ○ Flow

Fill Direction: ● Up ○ Right ○ Down ○ Left

□ Transparent Background

Title: □ Use Title; Percent Height: 25.0; Vert. Alignment: Top; Horiz. Alignment: Left; Title Text:

Figure 7.42. The Level Animation Dialog Box

It might also be fun to see the inventory itself in some way (no, we're not going to animate bins of little widgets or ghosts of them). For this, we installed a pair of Level animations (sometimes called *thermometer animations*) just to the left of the plot. The top animation (the black one, seen in *Figure 7.42*) plots the number of widgets physically in the inventory, which is expressed by *MX*(*Inventory Level*, 0), and we'll plot it so that it goes up as the inventory becomes more positive. The bottom animation (the red one, for which we didn't bother making a figure here) plots the number of widgets in backlog, which is *MX*(*–Inventory Level*, 0); when we're in backlog, this is a positive number, but we'd like to plot it diving lower from a zero level as we go deeper into backlog, so we selected the Fill Direction to be Down.

After adding a few labels and such, we're ready to go. Watch the plot and the thermometers to see the inventory level drop as demands occur, then pop back up when orders arrive. The average daily costs (rounded to the nearest penny) were 9.37 for holding, 17.03 for shortage, and 100.39 for ordering, for a total of 126.79. Whether (*s*, *S*)=(20, 40) is the best policy is a good question and one that we'll ask-you to take up (in a statistically valid way) in Exercises 6-10 and 6-11 at the end of Chapter 6.

7.11 Chapter summary

This chapter has gone into some depth on the detailed lower-level modeling capabilities, as well as correspondingly detailed topics like debugging and fine-tuned animation. While we've mentioned how you can access and blend in the SIMAN simulation language, we've by no means covered it; see Pegden, Shannon, and Sadowski (1995) for the complete treatment of SIMAN. At this point, you should be armed with a formidable arsenal of modeling tools to allow you to attack many systems, choosing constructs from various levels as appropriate. In several of the following chapters, we'll continue to expand on Arena's modeling capabilities.

7.12 Exercises

7-1 Develop a model of the problem we described in Chapter 2 and modeled as Model 3-1, but this time only using modules from the Advanced Process panel to replace the **Process** module. Use the Plot and Variable features from the Animate toolbar to complete your model. Run it for 20 minutes and compare your results to what we got earlier.

7-2 Parts arrive at a two-machine system according to an exponential interarrival distribution with mean 20 minutes. Upon arrival, the parts are sent to Machine 1 and processed. The processing-time distribution is TRIA(4.5, 9.3, 11) minutes. The parts are then processed at Machine 2 with a processing-time distribution as TRIA(l6.4, 19.1, 21.8) minutes. The parts from Machine 2 are directed back to Machine 1 to be processed a second time (same processing time). The completed parts then exit the system. Run the simulation for a single replication of 20,000 minutes to observe the average number in the machine queues and the average part cycle time.

7-3 Stacks of paper arrive at a trimming process with interarrival times of EXPO(lO); all times are in minutes. There are two trimmers, a primary and a secondary. All arrivals are sent to the primary trimmer. If the queue in front of the primary trimmer is shorter than five, the stack of paper enters that queue to wait to be trimmed by the primary trimmer, an operation of duration TRIA(9, 12, 15). If there are already five stacks in the primary queue, the stack is balked to the secondary trimmer (which has an infinite queue capacity) for trimming, of duration TRIA(l7, 19, 21). After the primary trimmer has trimmed 25 stacks, it must be shut down for cleaning, which lasts EXPO(30). During this time, the stacks in the queue for the primary trimmer wait for it to become available. Animate and run your simulation for 5,000 minutes.

Collect statistics, by trimmer, for cycle time, resource utilization, number in queue, and time in queue. So far as possible, use modules from the **Advanced Process** panel.

CHAPTER 8

Modeling and Statistical Analysis of Flexible Manufacturing System

Many of the essential elements of modeling with Arena were covered in Chapters 4 and 5, including the basic use of some of the Basic Process and Advanced Process panel modules, controlling the flow of entities, Resource Schedules and States, Sets, Variables, Expressions, and enhancing the animation. In this chapter, we'll expand on several concepts that allow you to do more detailed modeling. As before, we'll illustrate things concretely by means of a fairly elaborate example. We'll first start by introducing you to the concept of Stations and Transfers using our Electronic Assembly and Test System from Chapter 4. The addition is described in Section 8.1. Section 8.1.1 discusses the new Arena concepts. We modify our original model in Section 8.1.2 and the animation in Section 8.1.3.

A new example is described in Section 8.2; expressing it in Arena requires the new ideas of entity-dependent Sequences, discussed in Section 8.2.1. Then in Section 8.2.2, we take up the general issue of how to go about modeling a system, the level of detail appropriate for a project, and the need to pay attention to data requirements and availability. The data portion required for the model is built in Section 8.2.3 and the logical model in Section 8.2.4. In Section 8.2.5, we develop an animation, including discussion of importing existing CAD drawings for the layout. We conclude this portion with a discussion on verifying that the representation of a model in Arena really does what you want, in Section 8.2.6.

Continuing our theme of viewing all aspects of simulation projects throughout a study, we resume the topic of statistical analysis of the output data in Section 8.3, but this time it's for steady-state simulations, using the model from Section 8.2.

By the time you read and digest the material in this chapter, you'll have a pretty good idea of how to model things in considerable detail You'll also be in a position to draw statistically valid conclusions about the performance of systems as they operate in the long run.

8.1 Model 7-1. A Small Flexible Manufacturing System

A layout for our small manufacturing system is shown in *Figure 8.1*. The system to be modeled consists of part arrivals, four manufacturing cells, and part departures. Cells 1, 2, and 4 each have a single machine; Cell 3 has two machines. The two machines at Cell 3 are not identical; one of these machines is a newer model that can process parts in 80% of the time required by the older machine. The system produces three parts types, each visiting a different sequence of stations. The part steps and process times (in minutes) are given in Table 7-1. All process times are triangularly distributed; the process times given in Table 7-1 at Cell 3 are for the older (slower) machine.

The interarrival times between successive part arrivals (all types combined) are exponentially distributed with a mean of 13 minutes. The distribution by type is 26%, Part 1; 48%, Part 2; and 26%, Part 3. Parts enter from the left, exit at the right, and move only in a clockwise direction through the system. For now, we'll also assume that the time to move between any pair of cells is two minutes.

We want to collect statistics on resource utilization, time and number in queue, as well as cycle time (time in system, from entry to exit) by part type. Initially, we'll run our simulation for 32 hours.

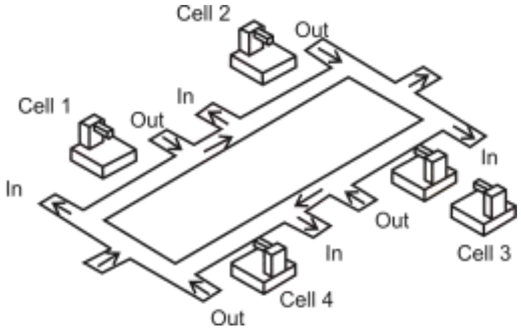


Figure 8.1. The Small Manufacturing System Layout

8.1.1 New Arena Concepts

There are several characteristics of this problem that require new Arena concepts. The first characteristic is that there are three part types that follow different process plans through the system. In our previous models, we simply sent all entities through the same sequence of stations. For this type of system, we need a process plan with an automatic routing capability.

The second characteristic is that the two machines in Cell 3 are not identical the newer machine can process parts faster than the old machine. Here we need to be able to distinguish between these two machines.

Table 8.1

Part Routings and Process Times

Part Type	Cell/Time	Cell/Time	Cell/Time	Cell/Time	Cell/Time
1	1/6, 8, 10	2/5, 8, 10	3/15, 20, 25	4/8, 12, 16	
2	1/11, 13, 15	2/4, 6, 8	4/15, 18, 21	2/6, 9, 12	3/27, 33, 39
3	2/7, 9, 11	1/7, 10, 13	3/18, 23, 28		

The third characteristic is the nature of the flow of entities through the system. In our previous models, the flow of entities through the system was accomplished using the direct Connect or direct Route option. When you use the Connect option, it results in an entity's being sent immediately to the next module, according to the connection, with no time advance in the simulation. If we used the Connect option in this new model, we would have to include a number of **Decide** modules to direct the parts to the correct next station in the sequence. It would result in the correct flow of parts; however, it wouldn't allow for the two-minute transfer delay, and it wouldn't allow us to animate the part flow. We could use a **Delay** module to represent the transfer time, but it wouldn't help us with animating part movement. Finally, we could use a number of **Decide** modules followed by **Route** modules both to model the two-minute transfer time and to show the part movement. As you might expect, there is an Arena concept, **Sequences**, that allows easy modeling of the flow of entities through the system while showing the part transfer time.

Many systems are characterized by entities that follow predefined, but different paths through the system. Most manufacturing systems have part or process plans that specify a list of operations each part type must complete before exiting the system. Many service systems have similar types of requirements. For example, a model of passenger traffic in an airport may require different paths through the airport depending on whether the passengers check baggage or have only carry-on pieces, as well as whether the passengers have domestic or international flights.

Arena can send entities through a system automatically according to a predefined *sequence* of station visitations. The **Sequence** data module, on the Advanced Transfer panel, allows us to define an ordered list of Stations that can include assignments of at-tributes or variables at each station. To direct an entity to follow this pattern of station visitations, we assign the sequence to the entity (using a built-in sequence attribute, described below) and use the Sequential option when we transfer the entity to its next destination (rather than the Connect tool).

As the entity makes its way along its sequence, Arena will do all the necessary bookkeeping to keep track of where the entity is and where it will go next. This is accomplished through the use of three special, automatically defined Arena attributes: Entity.Station (M), *Entity.Sequence* (NS), and Entity.JobStep (IS). Each entity has these three attributes, with the default values for newly created entities being 0 for each. The *Station* attribute contains the current station location of the entity or the station to which the entity is currently being transferred. The *Sequence* attribute contains the sequence the entity will follow, if any; you need to assign this to each entity that will be transferring via a sequence. The *Jobstep* attribute specifies the entity's position within the sequence.

We first define and name the list of stations to be visited for each type of entity (by part type, in our example) using the **Sequence** data module. Then, when we cause a new part to arrive into the system, we associate a specific Sequence with that entity by assigning the name of the sequence to the entity's Sequence attribute, NS.' When the entity is ready to transfer to the next station in its sequence, we select the Sequential option in the Destination Type field of the module we're using to transfer the entity to the next station. At this point during the run, Arena first increments the Jobstep attribute (IS) by 1. Then it retrieves the destination station from the Sequence based on the current values for the Sequence and Jobstep attributes. Any optional assignments are made (as defined in the Sequence) and the entity's Station attribute (M) is set to the destination station. Finally, Arena transfers the entity to that station.

Typically, an entity will follow a sequence through to completion and will then exit the model. However, this is not a requirement. The Jobstep attribute is incremented only when the entity is transferred using the Sequential option. You can temporarily suspend transfer via the sequence, transfer the entity directly to another station, and then re-enter the sequence later. This might be useful if some of the parts are required to be reworked at some point in the process. Upon completion of the rework, they might re-enter the normal sequence.

You can also reassign the sequence attributes at any time. For example, you might handle a part failure by assigning a new Sequence attribute and resetting the Jobstep attribute to 0. This new Sequence could transfer the part through a series of stations in a rework area. You can also back up or jump forward in the sequence by decreasing or in-creasing the Jobstep attribute. However, caution is advised as you must be sure to reset the Jobstep attribute correctly, remembering that Arena will first increment it, then look up the destination station in the sequence.

As indicated earlier, attribute and variable assignments can also be made at each jobstep in a sequence. For example, you could change the entity picture or assign a process time to a user-defined attribute. For our small manufacturing model, we'll use this option to define some of our processing times that are part- and station specific.

8.1.2 The Modeling Approach

The modeling approach to use for a specific simulation model will often depend on the system's complexity and the nature of the available data. In simple models, it's usually obvious what modules you'll require and the order in which you'll place them. But in more complex models, you'll often need to take a considerable amount of care developing the proper approach. As you learn more about Arena, you'll find that there are often a number of ways to model a system or a portion of a system. You will often hear experienced modelers say that there's not just a single, correct way to model a system. There are, however, plenty of wrong ways if they fail to capture the required system detail correctly.

The design of complex models is often driven by the data requirements of the model and what real data are available. Experienced modelers will often spend a great deal of time determining how they'll enter, store, and use their data, and then let this design determine which modeling constructs are required. As the data requirements become more demanding, this approach is often the only one that will allow the development of an ac-curate model in a short period of time. This is particularly true of simulation models of supply-chain systems, warehousing, distribution networks, and service networks. For example, a typical warehouse can have hundreds of thousands of uniquely different items, called SKUs (Stock-Keeping Units). Each SKU may require data characterizing its location in the warehouse, its size, and its weight, as well as reorder or restocking data for this SKU. In addition to specifying data for the contents of the warehouse, you also have customer order data and information on the types of storage devices or equipment that hold the SKUs. If your model requires the ability to change SKU locations, storage devices, restocking options, etc., during your experimentation, the data structure you use is critical. Although the models we'll develop in this book are not that complicated, it's always advisable to consider your data requirements before you start your modeling.

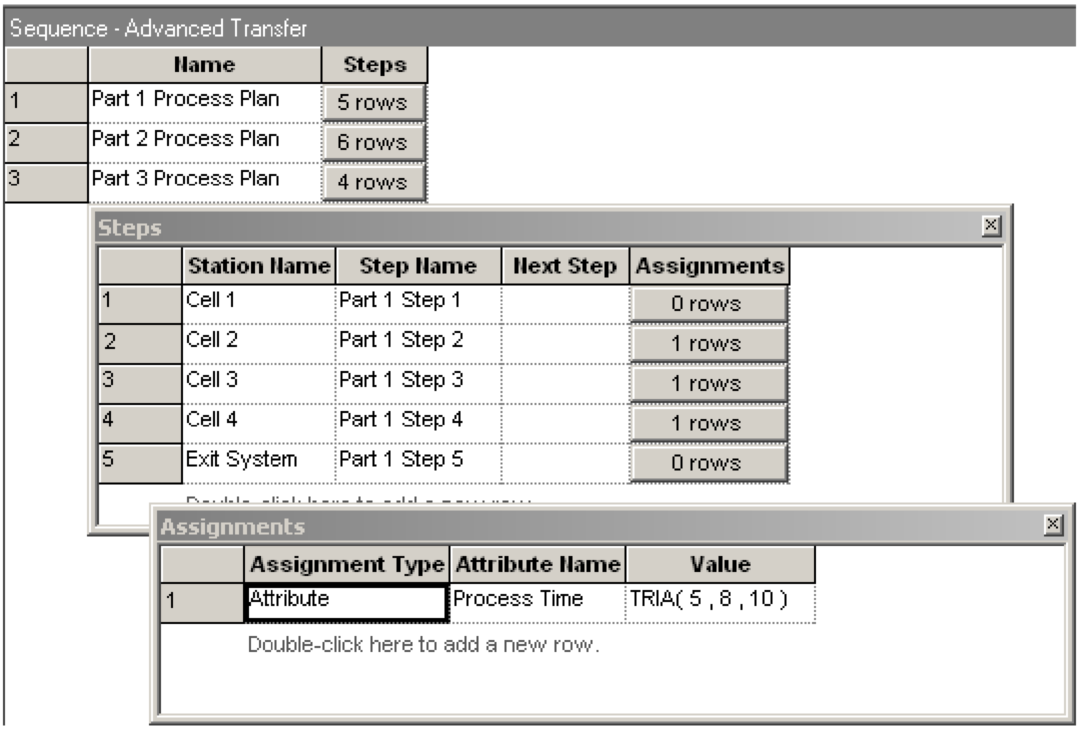
For our small manufacturing system, the data structure will, to a limited extent, affect our model design. We will use Sequences to control the flow of parts through the system, and the optional assignment feature to enter part process times for all but Cell 1. We'll use an Expression to define part process times for Cell 1. The part transfer time and the 80% factor for the time required by the new machine in Cell 3 will exploit the Variables concept. Although we don't normally think of Sets as part of our data design, their use can affect the modeling method. In this model, we'll use Sets, combined with a user-defined index, to ensure that we associate the correct sequence and picture with each part type.

First, we'll enter the data modules as discussed earlier. We'll then enter the main model portion, which will require several new modules. Next, we'll animate the model using a CAD or other drawing as a starting point. Finally, we'll discuss briefly the concept of model verification. By this time, you should be fairly familiar with opening and filling Arena dialogs, so we will not dwell on the mundane details of how to enter the information in Arena. In the case of modules and concepts that were introduced in previous chapters, we'll only indicate the data that must be entered. To see the "big picture" of where we're heading, you may want to peek ahead at *Figure 8.5*, which shows the complete model.

8.1.3 The Data Modules

We start by editing the **Sequence** data module from the Advanced Transfer panel. Double-click to add a new row and enter the name of the first sequence, `Part 1 Pro`*cess Plan.* Having entered the sequence name, you next need to enter the process steps, which are lists of Arena stations. For example, the `Part 1 Process Plan` requires you to enter the following Arena stations: *Cell 1, Cell 2, Cell 3, Cell 4,* and `Exit System`. We have arbitrarily given the Step Names as `Part 1 Step 1` through `Part 1 Step 5`. The most common error in entering sequences is to forget to enter the last step, which is typically where the entity exits the system. If you forget this step, you'll get an Arena run-time error when the first entity completes its process route and Arena is not told where to send it next. As you define the Sequences, remember that after you've entered a station name once, you can subsequently pick it from station pull-down lists elsewhere in your model. You'll also need to assign attribute values for the part process times for Cells 2, 3, and 4. Recall that we'll define an Expression for the part process times at Cell 1, so we won't need to make an attribute assignment for process times there.

Display 8.1 shows the procedure for Sequence *Part 1 Process Plan,* Step *Part 1 Step 2,* and Assignment of *Process Time*. Using the data in *Table 8.1*, it should be fairly straightforward to enter the remaining sequence steps. Soon we'll show you how to reference these sequences in the model logic.



Sequence - Advanced Transfer

	Name	Steps
1	Part 1 Process Plan	5 rows
2	Part 2 Process Plan	6 rows
3	Part 3 Process Plan	4 rows

Steps

	Station Name	Step Name	Next Step	Assignments
1	Cell 1	Part 1 Step 1		0 rows
2	Cell 2	Part 1 Step 2		1 rows
3	Cell 3	Part 1 Step 3		1 rows
4	Cell 4	Part 1 Step 4		1 rows
5	Exit System	Part 1 Step 5		0 rows

Assignments

	Assignment Type	Attribute Name	Value
1	Attribute	Process Time	TRIA(5 , 8 , 10)

Double-click here to add a new row.

Display 8.5. The **Sequence** data module

Table 8.2

The **Sequence** data module

Name	Part 1 Process Plan
Steps Station Name Step Name	 Cell 2 Part 1 Step 2

Assignments
Assignments Type
Attribute Name
Value
	Random(Expo)
	Attribute
	Process Time
	TRIA(5,8,10)

Next, we'll define the expression for the part process times at Cell 1, using the **Expression** data module in the Advanced Process panel. The expression we want will be called Cell 1 Times, and it will contain the part-process-time distributions for the three parts at Cell 1. We could just as easily have entered these in the previous **Sequence**s module, but we chose to use an expression so you can see several different ways to assign the processing times. We have three different parts that use Cell 1, so we need an arrayed expression with three rows, one for each part type. *Table 8.3* shows the data for this module.

Table 8.3

The Expression for Cell 1 Part Process Times

Name	Cell 1 Times
Rows	3
Expression Values	
Expression Value	TRIA(6,8,10)
Expression Value	TRIA(11,13,15)
Expression Value	TRIA(7,10,13)

Next, we'll use the **Variable** data module from the Basic Process panel to define the machine-speed factor for Cell 3 and the transfer time. Prior to defining our Factor variable, we made the following observation and assumption. The part-process times for Cell 3, entered in the **Sequence** data module, are for the old machine. We'll assume that the new machine will be referenced as I and the old machine will be referenced as 2. Thus, the first factor value is 0.8 (for the new machine), and the second factor is 1.0 (for the old machine). The transfer-time value (which doesn't need to be an array) is simply entered as 2. This allows us to change this value in a single place at a later time. If we thought we might have wanted to change this to a distribution, we would have entered it as an Expression instead of a Variable. *Table 8.4* shows the required entries.

Table 8.4

The Factor and Transfer Time Variables

Name	Factor
Rows	2
Initial Values	
Initial Value	0.8
Initial Value	1.0
Name	Transfer Time
Initial Values	
Initial Value	2

We'll use the **Set** data module from the Basic Process panel to form sets for our *Cell 3 machines*, part pictures, and entity types. The first is a Resource set, Cell 3 Machines, containing two members: *Cell 3 New* and *Cell 3 Old*. Our Entity Picture set is named *Part Pictures* and it contains three members: *Picture.Part 1*, *Picture.Part 2*, and *Picture.Part 3*. Finally, our Entity Type set is named *Entity Types* and contains three members: *Part 1*, *Part 2*, and *Part 3*.

As long as we're creating sets, let's add one more for our part sequences. If you at-tempt to use the **Set** module, you'll quickly find that the types available are Resource, Counter, Tally, Entity Type, and Entity Picture. You resolve this problem by using the **Advanced Set** data module found in the Advanced Process panel. This module list three types: Queue, Storage, and Other. The Other type is a catch-all option that allows you to form sets of almost any similar Arena objects. We'll use this option to enter our set named *Part Sequences*, which contains three members: *Part 1 Process Plan*, *Part 2 Process Plan*, and *Part 3 Process Plan*.

Before we start placing logic modules, let's open the *Run>Setup* dialog and set our Replication Length to 32 hours and our Base Time Units as minutes. We also selected the *Edit>Entity Pictures* menu option to open the **Entity Picture Placement** window, where we created three different pictures: *Picture.Part 1*, *Picture.Part 2*, and *Picture.Part 3*. In our example, we copied blue, red, and green balls; renamed them; and placed a text object with 1, 2, and 3, respectively, to denote the three different part types. Having defined all of the data modules, we're now ready to place and fill out the modules for the main model to define the system's logical characteristics.

8.1.4 The Logic Modules

The main portion of the model's operation will consist of logic modules to represent part arrivals, cells, and part departures. The part arrivals will be modeled using the four modules shown in *Figure 8.2*. The **Create** module uses a Random(Expo) distribution with a mean of 13 minutes to generate the arriving parts.

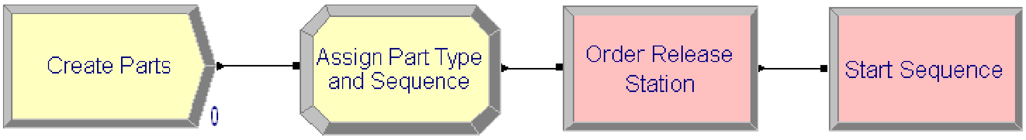


Figure 8.2 The Part Arrival modules

At this point, we have not yet associated a sequence with each arriving entity. We make this association in the **Assign** module as shown in *Display 8.8*. These assignments serve two purposes: they determine which part type has arrived, and they define an index, *Part Index*, for our sets that will allow us to associate the proper sequence with each arrival. We first determine the part index, or part type, with a discrete distribution. The distribution allows us to generate certain values with given probabilities. In our ex-ample, these values are the integers 1, 2, and 3 with probabilities 26%, 48%, and 26%, respectively. You enter these values in pairs-cumulative probability and value. The cumulative probability for the last value (in our case, 3) should be 1.0. In general, the values need not be integers, but can take on any values, including negative numbers. The part index values of 1, 2, and 3 allow us not only to refer to the part type, but in this case, they also allow us to index into the previously defined set *Part Sequences* so that the proper sequence will be associated with the part. To do so, we assign the proper sequence to the Arena Entity. Sequence attribute by using the *Part Index* attribute as an index into the *Part Sequences* set.

We also need to associate the proper entity type and picture for the newly arrived part. We do this in the last two assignments by using the *Part Index* attribute to index into the relevant sets. Recall that we created the *Entity Types* set (consisting of Part 1, Part 2, and Part 3). We also defined three pictures (*Picture.Part 1*, *Picture.Part 2*, and *Picture.Part 3*) and grouped them into a set called *Part Pictures*. You can think of these sets as an arrayed variable (one-dimensional) with the index being the *Part Index* attribute we just assigned. This is true in this case be-cause we have a one-to-one relationship between the part type (*Part Index*) and the sequence, picture, and entity type. An index of 1 implies *Part 1* follows the first sequence, etc. (Be careful in future models because this may not always be true. It is only true in this case because we defined our data structure so we would have this one-to-one relationship.)

In this case, the completed **Assign** module determines the part type, and then assigns the proper sequence, entity type, and picture. Note that it was essential to define the value of Part Index first in this **Assign** module, which performs multiple assignments in the order listed, since the value of Part Index determined in the first step is used in the later assignments (*Table 8.5*). We're now ready to send our part on its way to the first station in its sequence.

We'll accomplish this with the **Route** module from the Advanced Transfer panel. Before we do this, we need to tell Arena where our entity, or part, currently resides (its current station location). If you've been building your own model along with us (and paying attention), you might have noticed the attribute named *Entity.Station* on the pull-down list in the **Assign** module that we just completed. (You can go back and look now if you like.) You might be tempted to add an assignment that would define the current location of our part. Unfortunately if you use this approach, Arena will return an error when you try to run your model; instead, we use a **Station** module as described next.

Table 8.5

The **Assign** module. Assigning Part Attributes

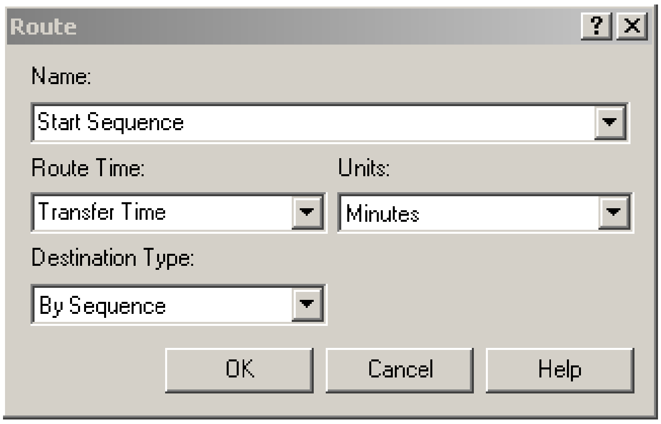
Name	Assign Part Type and Sequence
Assignments	
Type	Attribute
Attribute Name	Part Index
New Value	DISC(0.26,1,0.74,2,1.0,3)
Type	Attribute
Attribute Name	Entity.Sequence
New Value	Part Sequences (Part Index)
Type	Attribute
Attribute Name	Entity.Type
New Value	Entity Types (Part Index)
Type	Attribute
Attribute Name	Entity.Picture
New Value	Part Pictures(Part Index)

Our completed model will have six stations: Order Release, Cell 1, Cell 2, Cell 3, Cell 4, and Exit System. The last five stations were defined when we filled in the information for our part sequences. The first station, Order Release, will be defined when we send the entity through a **Station** module (Advanced Transfer panel), which will define the station and tell Arena that the current entity is at that location. Our completed **Station** module is shown in *Table 8.6.*

Table 8.6

The **Station** Module

Name	Order Release Station
Station Type	Station
Station Name	Order Release



Start Sequence

Transfer Time

Minutes

By Sequence

Display 8.6. The **Route** module

We're finally ready to send our part to the first station in its sequence with the **Route** module (Advanced Transfer panel). The **Route** module transfers an entity to a specified station, or the next station in the station visitation sequence defined for the entity. A delay time to transfer to the next station may be defined. In our model, the previously defined the variable *Transfer Time* is entered for the route time; see *Display 8.6.* We selected the Sequential option as the

Destination Type. This causes the Station Name field to disappear, and when we run the model, Arena will route the arriving entities according to the sequences we defined.

Table 8.7

Parameters of **Route** module

Name Route Time Units Destination Type	Start Sequence Transfer Time Minutes By Sequence

Now that we have the arriving parts being routed according to their assigned part sequences, we need to develop the logic for our four cells. The logic for all four cells is essentially the same. A part arrives to the cell (at a station), queues for a machine, is processed by the machine, and is sent to its next step in the part sequence. All four of these cells can be modeled easily using the **Station** - **Process** - **Route** module sequence shown in *Figure 8.3* (for Cell 1).

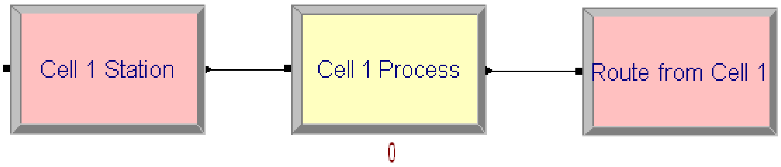


Figure 8.3. Cell 1 Logic Modules

The **Station** module provides the location to which a part can be sent. In our model we are using sequences for all part transfers, so the part being transferred would get the next location from the sequence. The data entries for the **Station** module for Cell 1 are shown in *Table 8.8***.**

Table 8.8

The Cell 1 **Station** module

Name Station Type Station Name	Cell 1 Station Station Cell 1

Table 8.9

The Cell 1 **Process** module

Name Action	Cell 1 Process Seize Delay Release
Resources Type Resource Name Quantity	 Resource Cell 1 Machine 1
Delay Type Units Expression	Expression Minutes Cell 1 Times(Part Index)

A part arriving at the Cell 1 station is sent to the following **Process** module (using a direct connect). For the Process Time, we've entered the previously defined expression *Cell 1 Times* using the *Part Index* attribute to reference the appropriate part-processing time. This expression will generate a sample from the triangular distribution with the parameters we defined earlier. The remaining entries are shown in *Table 8.9*.

Upon completion of the processing at Cell 1, the entity is directed to the **Route** module, described in *Table 8.10*., where it is routed to its next step in the part sequence. Except for the

Name, this **Route** module is identical to the **Route** module we used to start our sequence at the Order Release station in *Table 8.7*.

The remaining three cells are very similar to Cell 1, so we'll skip the details. To create their logic, we copied the three modules for Cell 1 three times and then edited the required data. For each of the new Station and **Route** modules, we simply changed all occurrences of Cell 1 to Cell 2, Cell 3, or Cell 4. We made the same edits to the three additional **Process** modules and changed the delay Expression for Cells 2 and 4 to Process Time. Recall that in the **Sequences** module we defined the part-processing times for Cells 2, 3, and 4 by assigning them to the entity attribute *Process Time*. When the part was routed to one of these cells, Arena automatically assigned this value so it could be used in the module.

Table8.10

The Cell 1 **Route** module

Name	Route from Cell 1
Route Time	Transfer Time
Units	Minutes
Destination Type	By Sequence

Table 8.11

The Cell 3 **Process** module

Name	Cell 3 Process
Action	Seize Delay Release
Resources	
Type	Set
Set Name	Cell 3 Machines
Quantity	1
Save Attribute	Index
Delay Type	Expression
Units	Minutes
Expression	Process Time * Factor(Machine Index)

At this point, you should be aware that we've somewhat arbitrarily used an Expression for the Cell 1 part-processing times, and attribute assignments in the sequences for the remaining cells. We actually used this approach to illustrate that there are often several different ways to structure data and access them in a simulation model. We could just as easily have incorporated the part-processing times at Cell 1 into the sequences, along with the rest of the times. We also could have used Expressions for these times at Cells 3 and 4. However, it would have been difficult to use an Expression for these times at Cell 2, because Part 2 visits that cell twice and the processing times are different, as shown in *Table 8.1*. Thus, we would have had to somehow include in our model the ability to know whether the part was on its first or second visit to Cell 2 and define our expression to recognize that. It might be an interesting exercise, but from the modeler's point of view, why not just define these values in the sequences? You should also recognize that this sample model is relatively small in terms of the number of cells and parts. In practice, it would not be uncommon to have 30 to 50 machines with hundreds of different part types. If you undertake a problem of that size, we strongly recommend that you take great care in designing the data structure since it may make the difference between a success or failure of the simulation project.

The **Process** module for Cell 3 is slightly different because we have two different machines, a new one and an old one, that process parts at different rates. If the machines were identical, we could have used a single resource and entered a capacity of 2. We made note of this earlier and grouped these two machines into a Set called Cell 3 Machines. Now we need to use this set at Cell 3. *Table 8.11* shows the data entries required to complete the **Process** module for this cell.

In the Resource section, we select Set from the pull-down list for our Resource Type. This allows us to select from our specified Cell 3 Machines set for the Set Name field. You can also Seize a Specific Member of a set; that member can be specified as an entity attribute. Finally, you could use an Expression to determine which resource was required. For our selection of Resource Set, we've accepted the default selection rule, Cyclical, which causes the entity to attempt to select the first available resource be-ginning with the successor of the last resource selected. In our case, Arena will try to select our two resources alternately; however, if only one resource is currently available, it will be selected. Obviously, these rules are only used if more than one resource is avail-able for selection. The Random rule would cause a random selection, and the Preferred Order rule would select the first available resource in the set. Had we selected this option, Arena would have always used the new machine, if available, because it's the first re-source in the set. The remaining rules would only apply if one or more of our resources had a capacity greater than 1.

The Save Attribute option allows us to save the index, which is a reference to the selected set member, in a user-specified attribute. In our case, we will save this value in the Attribute *Machine Index*. If the new machine is selected, this attribute will be assigned a value of 1, and if the old machine is selected, the attribute will be assigned a value of 2. This numbering is based on the order in which we entered our resources when we defined the set. The Process Time entry is an expression using our attribute *Process Time*, assigned by the sequence, multiplied by our variable Factor. Recall that our process times are for the old machine and that the new machine can process parts in 80% of that time. Although it's probably obvious by now how this expression works, we'll illustrate it. If the first resource in the set (the new machine) is selected, Machine Index will be assigned a value of I and the variable Factor will use this attribute to take on a value of 0.8. If the second machine (the old machine) is selected, the Machine Index is set to 2, the Factor variable will take on a value of 1.0, and the original process time will be used. Although this method may seem overly complicated for this ex-ample, it is used to illustrate the tremendous flexibility that Arena provides. An alternate method, which would not require the variable, would have used the following logical expression:

```
        Process Time * ((Machine Index == 1)*0.8) + (Machine Index == 2)
```

or

```
        Process Time * (1–Machine Index == 1) * 0.2))
```

We leave it to the reader to figure out how these expressions work (Hint: "a--b" evaluates to I if a equals h and to 0 otherwise).

Having completely defined all our data, the part arrival process, and the four cells, we're left only with defining the parts' exit from the system. The two modules that accomplish this are shown in *Figure 8.10*.

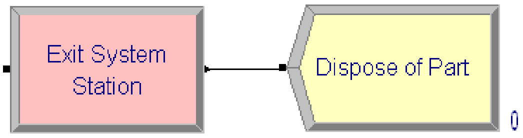


Figure 8.4. The Exit System Logic Modules

As before, we'll use the **Station** module to define the location of our Exit System Station. The **Dispose** module is used to destroy the completed part. Our completed model (although it is not completely animated) appears in *Figure 8.5*.

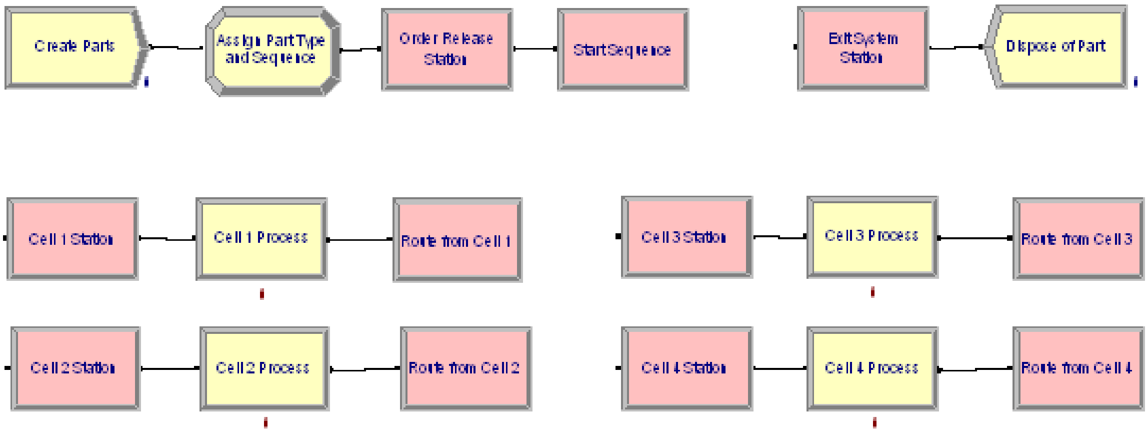


Figure 8.5. The Completed Model

At this point we could run our model, but it would be difficult to tell if our sequences arc working properly. So before we start any analysis, we'll first develop our animation to help us determine whether the model is working correctly. Let's also assume that we'll have to present this model to higher-level management, so we want to develop a some-what fancier animation.

8.1.5 Animation

We could develop our animation in the same manner as we did in Chapter 5 create our own entity pictures, resource symbols, and background based on the picture that was presented in *Figure 8.1*. However, a picture might exist that someone already developed that reflects an accurate representation of the system. In fact, it might even exist as a CAD file somewhere in your organization. For instance, the drawing presented in *Figure 8.1* was developed using the AutoCAD' package from AutoDesk. Arena supports integration of files from CAD programs into its workspaces. Files saved in the DXF format de-fined by AutoCAD can be imported directly into Arena. Files generated from other CAD or drawing programs (e.g., Visio®) should transfer into Arena as long as they adhere to the AutoCAD standard DXF format.

If the original CAD drawing is 2-D, you only need to save it in the DXF format and then import that file directly into Arena. Most CAD objects (polygons, etc.) will be rep-resented as the same or similar objects in Arena. If the drawing is 3-D, you must first convert the file to 2-D. Colors are lost during this conversion to 2-D, but they may be added again in AutoCAD or in Arena. This conversion also transforms all objects to lines, so the imported drawing can only be manipulated in Arena as individual lines, or the lines may be grouped as objects. We'll assume that you have a DXF file and refer you to online help (the DXF File Importation topic) for details on creating the DXF file or converting a 3-D drawing. One word of advice: a converted file is imported into Arena as white lines so if your current background is white, it may appear that the file was not imported. Simply change the background color (🎨▾ on the **Draw** toolbar) or select all the lines and changed the line color (🖊▾).

For our small manufacturing system, we started with a 3-D drawing and converted it to 2-D. We then saved the 2-D file in the DXF format (Model 07-01.dxf). A DXF file is imported into your current model file by selecting the File>DXF Import option. Select the file, and when you move your cursor to the model window, it will appear as cross hairs. Using the cursor, draw a box to contain the entire drawing to be imported. If the imported drawing is not the correct size, you can select the entire drawing and resize it as required. You can now use this drawing as the start of your animation.

For our animation, we first want highlight the entire drawing and use the Arrange>Ungroup menu option or *Ungroup* button ( ) from the **Arrange** toolbar to convert the drawing to individual lines. Next you should delete all the lettering and arrows. Then we'll move the Cell 1 queue animation object from the **Process** module (Cell 1 Process) to its approximate position on the drawing.

Now you want to position your cursor near the upper-left corner of a machine, left click-hold, move your cursor so the resulting box encloses the entire drawing and release the cursor. Use the *Copy* button to copy the drawing of the machine to the clipboard. Now we'll open the Resource Picture Placement window by clicking on the *Resource* button ( ) from the **Animate** toolbar. Double-click on the idle resource symbol and re-place the current picture with a copy of the contents of the clipboard. Delete the base of the machine and draw two boxes on top of the representation for the top part of the ma-chine. This is necessary as our drawing consists of all lines, and we want to add a fill color to the machine. Now delete all the lines from the original copy and fill the boxes with your favorite color. Copy this new symbol to your library and then copy that symbol to the Busy picture. Save your library, exit the resource window, and finally place the resource. You now have a resource picture that will not change, but we could go back later and add more animation effects.

For the next step, draw a box the same size as the base of the machine and then delete the entire drawing of the machine. Fill this new box with a different color and then place the resource symbol on top of this box (you may have to resize the resource symbol to be the correct size). Now move the seize point so that it is positioned at about the center of our new machine. Now make successive copies of our new resource and the machine base, and then change the names. If you follow this procedure, note that you will need to flip the resources for Cells 3 and 4.

To complete this phase of the animation, we need to move and resize the remaining queues. Once this is complete, you can run your animation and see your handiwork. You won't see parts moving about the system (there are no routes yet), but you will see the parts in the queues and at the machines. If you look closely at one of the machines while the animation is running, you'll notice that the parts sit right on top of the entire machine. This display is not what we want. Ideally, the part should sit on top of the base, but under the top part of our machine (the resource). Representing this is not a problem. Select a resource (you can do this in edit mode or by temporarily stopping the animation) and use the Bring to Front feature with the *Arrange>Bring to Front* menu option or the button on the **Arrange** toolbar ( ). Now when you run the animation, you'll get the desired effect. We're now ready to animate our part movement.

In our previous animations, we basically had only one path through the system so adding the routes was fairly straightforward. In our small manufacturing system, there are multiple paths through the system, so you must be careful to add routes for each travel possibility. For example, a part leaving Cell 2 can go to Cell 1, 3, or 4. The stations need to be positioned first; add station animation objects using the Station feature on the **Animate** toolbar. Next, place the routes. If you somehow neglect to place a route, the simulation will still (correctly) send the entity to the correct station with a transfer time of 2; however, that entity movement will not appear in your animation. Also be aware that routes can be used in both directions. For example, let's assume that you added the route from Cell 1 to Cell 2, but missed the route from 2 to 1. When a Part 2 completes its processing at Cell 2, Arena looks to route that part to Cell 1, routing from 2 to 1. If the route were missing, it would look for the route from 1 to 2 and use that. Thus, you would see that part moving from the entrance of Cell 2 to the Exit of Cell 1 in a counterclockwise direction (an animation mistake for this model).

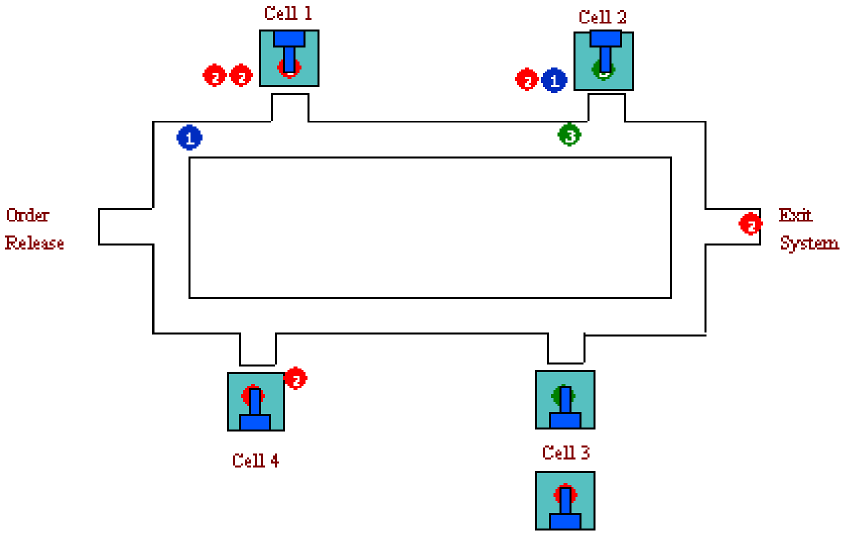


Figure 8.6. The Animated Small Manufacturing System

If you run and watch your newly animated model, you should notice that occasion-ally parts will run over or pass each other. This is due to a combination of the data sup-plied and the manner in which we animated the model. Remember that all part transfers were assumed to be two minutes. Arena sets the speed of a routed entity based on the transfer time and the physical length of the route on the animation. In our model, some of these routes are very short (from Cell 1 to Cell 2) and some are very long (from Cell 2 to Cell 1). Thus, the entities being transferred will be moving at quite different speeds relative to each other. If this was important, we could request or collect better transfer times and incorporate these into our model. The easiest way would be to delete the variable *Transfer Time* and assign these new transfer times to a new attribute in the **Sequences** module. If your *Transfer times* and drawing are accurate, the entities should now all move at the same speed. The only potential problem is that a part may enter the main aisle at the same time another part is passing by, resulting in one of the parts overlapping the other until their paths diverge. This is a more difficult problem to resolve, and it may not be worth the bother. If the only concern is for purposes of presentation, watch the animation and find a long period of time when this does not occur, show only this period of time during your presentation. The alternative is to use material-handling constructs, which we'll do in Chapter 8.

After adding a little annotation, the final animation should look something like *Figure 8.6* (this is a view of the system at time 541.28). At this point in your animation, you might want to check to see if your model is running correctly or at least the way you intended it. We'll take up this topic in our next section.

8.1.6 Verification

Verification is the process of ensuring that the Arena model behaves in the way it was intended according to the modeling assumptions made. This is actually very easy compared to model validation, which is the process of ensuring that the model behaves the same as the real system. We'll discuss both of these aspects in more detail in Chapter 12. Here we'll only briefly introduce the topic of model verification.

Verification deals with both obvious problems as well as the not-so-obvious. For ex-ample, if you had tried to run your model and Arena returned an error message that indicated that you had neglected to define one of the machine resources at Cell 3, the model is obviously not working the way you intended. Actually, we would normally call this the process of debugging! However, since we assume that you'll not make these kinds of errors, we'll deal with the not-so-obvious problems.

Verification is fairly easy when you're developing small classroom-size problems like the ones in this book. When you start developing more realistically sized models, you'll find that it's a much more difficult process, and you may never be 100% sure on very, very large models.

One easy verification method is to allow only a single entity to enter the system and follow that entity to be sure that the model logic and data are correct. You could use the *Step* feature (▶❚) found on the **Run** toolbar to control the model execution and step the entity through the system. For this model, you could set the Max Batches field in the **Arrive** module to 1. To control the entity type, you could replace the discrete distribution that determines the part type with the specific part type you want to observe. This would allow you to check each of the part sequences. Another common method is to replace some or all model data with constants. Using deterministic data allow you to predict the system behavior exactly.

If you're going to use your model to make real decisions, you should also check to see how the model behaves under extreme conditions. For example, introduce only one part type or increase/decrease the part interarrival times. If your model is going to have problems, they will most likely show up during these kinds of stressed-out runs. Also, try to make effective use of your animation-it can often reveal problems when viewed by a person familiar with the operation of the system being modeled.

It's often a good idea, and a good test, to make long runs for different data and observe the summary results for potential problems. One skill that can be of great use during this process is that of performance estimation. A long, long time ago, before the invention of calculators and certainly before personal computers, engineers carried around sticks called slide rules (often protected by leather cases and proudly attached to engineers' belts). These were used to calculate the answers to the complicated problems given to them by their professors or bosses. These devices achieved this magic feat by adding or subtracting logs (not tree logs, but logarithms). Although they worked rather well (but not as well, as easily, or as fast as a calculator), they only returned (actually you read it off the stick) a sequence of digits, say 3642. It was up to the engineer to figure out where to put the decimal point. Therefore, engineers had to become very good at developing rough estimates of the answers to problems before they used the sticks. For ex-ample, if the engineer estimated the answer to be about 400, then the answer was 364.2. However, if the estimate was about 900, there was a problem. At that point, it was necessary to determine if the problem was in the estimation process or the slide-rule process. We suspect that at this point you're asking two questions: 1) why the long irrelevant story, and 2) did we really ever use such sticks? Well, the answers are: 1) to illustrate a point, and 2) just one of the authors!

So how do you use this great performance-estimation skill? You define a set of conditions for the simulation, estimate what will result, make a run, and look at the summary data to see if you were correct. If you were, feel good and try it for a different set of conditions. If you weren't correct (or at least in the ballpark), find out why not. It may be due to bad estimation, a lack of understanding of the system, or a faulty model. Sometimes not-so-obvious (but real) results are created by surprising interactions in the model. In general, you should thoroughly exercise your simulation models and be comfortable with the results before you use them to make decisions. And it's best to do this early in your project and often.

Back in Chapter 2 when we mentioned verification, we suggested that you verify your code. Your response could have been. "What code?" Well, there is code, in fact we briefly showed you some of this code in Section 5.5. But you still might be asking, "Why Code?" To explain the reason for this code requires a little bit of background (yes, an-other history lesson). The formation of Systems Modeling (the company that originally developed Arena) and the initial release of the simulation language SIMAN (on which Arena is based and to which you can gain access through Arena) occurred in 1982. Personal computers were just starting to hit the market, and SIMAN was designed to run on these new types of machines. In fact, SI MAN only required 64 Kbytes of memory, which was a lot in those days. There was no animation capability (Cinema, the accompanying animation tool, was released in 1985), and you created models using a text editor, just like using a programming language. A complete model required the development of two files, a *model file* and an *experiment file*. The model file, often referred to as the *.mod* file, contained the model logic. The experiment file, referred to as the *.exp* file, defined the experimental conditions. It required the user to list all stations, attributes, resources, etc., that were used in the model. The creation of these files required that the user follow a rather exacting syntax for each type of model or experiment construct. In other words, you had to start certain statements only in column 10; you had to follow certain entries with the required comma, semicolon, or colon; all resources and attributes were referenced only by number; and only a limited set of key words could be used. (Many of your professors learned simulation this way.)

Since 1982, SIMAN has been enhanced continually and still provides the basis for an Arena simulation model. When you run an Arena simulation, Arena examines each option you selected in your modules and the data that you supplied and then creates SIMAN MOD and EXP files. These are the files that are used to run your simulations. The implication is that all the modules found in the Basic Process, Advanced Process, and Advanced Transfer panels are based on the constructs bond in the SIMAN simulation language. The idea is to provide a simulation tool (Arena) that is easy to use, yet one that is still based on the powerful and flexible SIMAN language. So you sec, it is still possible, and sometimes even desirable, to look at the SIMAN code. In fact, you can even write out and edit these files. However, it is only possible to go down (from Arena to SIMAN code), and not possible to go back up (from SIMAN code to Arena). As you become more proficient with Arena, you might occasionally want to look at this code to be assured that the model is doing exactly what you want it to- verification.

We should point out that you can still create your models in Arena using the base SIMAN language. If you build a model using only the modules found in the Blocks and Elements panels, you arc basically using the SIMAN language. Recall that we used several modules from the Blocks panel when we constructed our call center model in Chapter 5.

You can view the SIMAN code for our small manufacturing model by using the *Run*> *SIMAN*>*View* menu option. Selecting this option will generate both files, each in a separate window. *Figure 8.7* shows a small portion of the MOD file, the code written out for the **Process** module used at Cell 3. The SIMAN language is rather descriptive, so it is possible to follow the general logic. An entity that arrives at this module increments some internal counters, enters a queue, *Cell 3 Process.Queue*, waits to seize a resource from the set *Cell 3 Machines*, delays for the process time (adjusted by our factor), releases the resource, decrements the internal counters, and exits the module.

```
Model statements for module:  BasicProcess.Process 3 (Cell 3 Process)
;
10$           ASSIGN:        Cell 3 Process.NumberIn=Cell 3 Pro-
cess.NumberIn + 1:
                             Cell 3 Process.WIP=Cell 3 Process.WIP+1;
138$          QUEUE,         Cell 3 Process.Queue;
137$          SEIZE,         2,VA:
                             SELECT(Cell 3 Machines,CYC, Machine In-
dex),1:NEXT(136$);

136$          DELAY:         Process Time * Factor( Machine Index ),,VA;
135$          RELEASE:       Cell 3 Machines(Machine Index),1;
183$          ASSIGN:        Cell 3 Process.NumberOut=Cell 3 Pro-
cess.NumberOut + 1:
                             Cell 3 Process.WIP=Cell 3 Process.WIP-
1:NEXT(11$);
```

Figure 8.7. SIMAN Model File for the **Process** module

Figure 8.8 shows a portion of the EXP file that defines our three attributes and the queues and resources used in our model.

```
ATTRIBUTES:   Machine Index:
              Part Index:
              Process Time;
QUEUES:       Cell 1 Process.Queue,FIFO,,AUTOSTATS(Yes,,):
              Cell 2 Process.Queue,FIFO,,AUTOSTATS(Yes,,):
              Cell 3 Process.Queue,FIFO,,AUTOSTATS(Yes,,):
              Cell 4 Process.Queue,FIFO,,AUTOSTATS(Yes,,);
RESOURCES: Cell 2 Machine,Capacity(1),,,COST(0.0,0.0,0.0),
          CATEGORY(Resources),,AUTOSTATS(Yes,,):
              Cell 1 Machine,Capacity(1),,,COST(0.0,0.0,0.0),
          CATEGORY(Resources),,AUTOSTATS(Yes,,):
              Cell 3 Old,Capacity(1),,,COST(0.0,0.0,0.0),
          CATEGORY(Resources),,AUTOSTATS(Yes,,):
              Cell 3 New,Capacity(1),,,COST(0.0,0.0,0.0),
          CATEGORY(Resources),,AUTOSTATS(Yes,,):
              Cell 4 Machine,Capacity(1),,,COST(0.0,0.0,0.0),
          CATEGORY(Resources),,AUTOSTATS(Yes,,);
```

Figure 8.8. SIMAN Experiment File for the **Process** module

We won't go into a detailed explanation of the statements in these files; the purpose of this exercise is merely to make you aware of their existence. For a more comprehensive explanation, refer to Pegden, Shannon, and Sadowski (1995).

If you're familiar with SIMAN or you would just like to learn more about how this process works, we might suggest that you place a few modules, enter some data, and write out the MOD and EXP files. Then edit the modules by selecting different options and write out the new files. Look for the differences between the two MOD files. Doing so should give you a fair amount of insight as to how this process works.

8.2 Statistical Analysis of Output from Steady-State Simulations

In Sections 5.2.9 and 5.8.1, we described the difference between terminating and steady-state simulations and indicated how you can use Arena's reports, PAN, and the Output Analyzer to do statistical analyses in the terminating case. In this section, we'll show you how to do statistical inference on steady-state simulations.

Before proceeding, we should encourage you to be Yule that a steady-state simulation is appropriate for what you want to do. Often people simply assume that a long-run, steady-state simulation is the thing to do, which might in fact be true. But if the starting and stopping conditions are part of the essence of your model, a terminating analysis is probably more appropriate; if so, you should just proceed as in Section 5.8. The reason for avoiding steady-state simulation is that, as you're about to sec, it's a lot harder to carry out anything approaching a valid statistical analysis than in the terminating case if you want anything beyond Arena's standard 95`%, confidence intervals on mean performance measures; so if you don't need to get into this, you shouldn't.

One more caution before we wade into this material: As you can imagine, the run lengths for steady-state simulations need to be pretty long. Because of this, there are more opportunities for Arena to sequence its internal operations a little differently, causing the random-number stream (see Chapter 11) to be used differently. This doesn't make your model in any sense "wrong" or inaccurate, but it can affect the numerical results, especially for models that have a lot of statistical variability inherent in them. So if you follow along on your computer and run our models, there's a chance that you're going to gel numerical answers that differ from what we report here. Don't panic over this since it is, in a sense, to he expected. If anything, this just amplifies the need for some kind of statistical analysis of simulation output data since variability can come not only from "nature" in the model's properties, but also from internal computational issues.

In Section 8.3.1, we'll discuss determination of model warm up and run length. Section 8.3.2 describes the truncated-replication strategy for analysis, which is by far the simplest approach (and, in some ways, the best). A different approach called hatching is described in Section 8.3.3. A brief summary is given in Section 8.3.4, and Section 8.3.5 mentions some other issues in steady-state analysis.

8.2.1 Warm Up and Run Length

As you might have noticed, our examples have been characterized by a model that's initially in an empty-and-idle state. This means that the model starts out empty of entities and all resources are idle. In a terminating system, this might be the way things actually start out, so everything is fine. But in a steady-state simulation, initial conditions aren't supposed to matter, and the run goes forever.

Actually, though, even in a steady-state simulation, you have to initialize and stop the run somehow. And since you're doing a simulation in the first place, it's a pretty safe bet that you don't know much about the "typical" system state in steady state or how "long" a run is long enough. So you're probably going to wind up initializing in a state that's pretty weird from the viewpoint of steady state and just trying some (long) run lengths. If you're initializing empty and idle in a simulation where things eventually become congested, your output data for some period of time after initialization will tend to understate the eventual congestion; i.e., will be biased toward low values of typical performance measures.

To remedy this, you might try to initialize in a state that's "better" than empty and idle. This would mean placing, at time 0, some number of entities around the model and starting things out that way. While it's possible to do this in your model, it's pretty inconvenient. More problematic is that you'd generally have no idea how many entities to place around at time 0; this is, after all, one of the questions the simulation is supposed to answer.

Another way of dealing with initialization bias is just to run the model for so long that whatever bias may have been there at the beginning is overwhelmed by the amount of later data. This can work in some models if the biasing effects of the initial conditions wear off quickly.

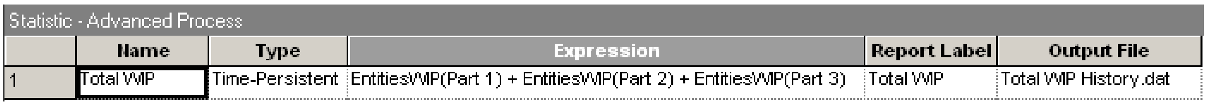
However, what people usually do is to initialize empty and idle, realizing that this is unrepresentative of steady state, but then let the model warm up for a while until it appears that the effect of the artificial initial conditions have worn off. At that time, you can clear the statistical accumulators and start afresh, gathering statistics for analysis from then on. In effect, this is initializing in a state other than empty and idle, but you let the model decide how many entities to have around when you start to watch your performance measures. The run length should still be long, but maybe not as long as you'd need to overwhelm the initial bias by sheer arithmetic.

It's very easy to specify an initial Warm-up Period in Arena. Just open the Run/Setup/ Replication Parameters dialog and fill in a value (be sure to verify the Time Units). Every replication of your model still runs starting as it did before, but at the end of the Warm-up Period, all statistical accumulators are cleared and your reports (and any Outputs-type saved data from the **Statistic** module of results across the replications) reflect only what happened after the warm-up period ends. In this way, you can "decontaminate" your data from the biasing initial conditions.

The hard part is knowing how long the warm-up period should be. Probably the most practical idea is just to make some plots of key outputs from within a run, and eyeball when they appear to stabilize. To illustrate this, we took Model 7-2 from Section 8.2 and made the following modifications, calling the result Model 7-3:

■ To establish a single overall output performance measure, we made an entry in the **Statistic** module to compute the total work in process (WIP) of all three parts combined. The Name and Report Label are both Total WIP, and the Type is Time-Persistent. To enter the Expression we want to track over time, we right-clicked in that field and selected **Build Expression**, clicked down the tree via *Basic Process Variables>Entity> Number in Process*, selected *Part 1* as the Entity Type, and got EntitiesWIP(*Part 1*) for the Current Expression which is part of what we want. Typing a + after that and selecting *Part 2* for the Entity Type, another +, then Type 3 for the Entity Type finally gives us EntitiesWIP(*Part 1*)+EntitiesWIP(*Part 2*)+EntitiesWlP(*Part 3*), which is the total WIP. This will create an entry labeled Total WIP in the reports (under User Specified) giving the time-average and maximum of the total number of parts in process.

■ However, we want to track the "history" of the Total WIP curve during the simulation rather than just getting the post-run summary statistics since we want to "eyeball" when this curve appears to stabilize in order to specify a reasonable Warm-up Period. You could place an animated Plot in your model, as we've done before; however, this will disappear as soon as you hit the *End* button, and will also be subject to perhaps-large variation, clouding your judgment about the stabilization point. We need to save the curve history and make more permanent plots and also to plot the curve for several replications to mitigate the noise problem. To do so we made a file-name entry, Total WIP History.dat, in the Total WIP History. dat, in the Output File field of the Total WIP entry in the **Statistic** module, which will save the required information into that file, which can be read into the Arena Output Analyzer (see Section 5.8.4) and plotted after the run is complete. Depending on how long your run is and how many replications you want, this file can get pretty big since you're asking it to hold a lot of detailed, within-run information (our run, described below, resulted in a file of about 176 KB). The complete entry in the **Statistic** module is in *Figure 8.9*.



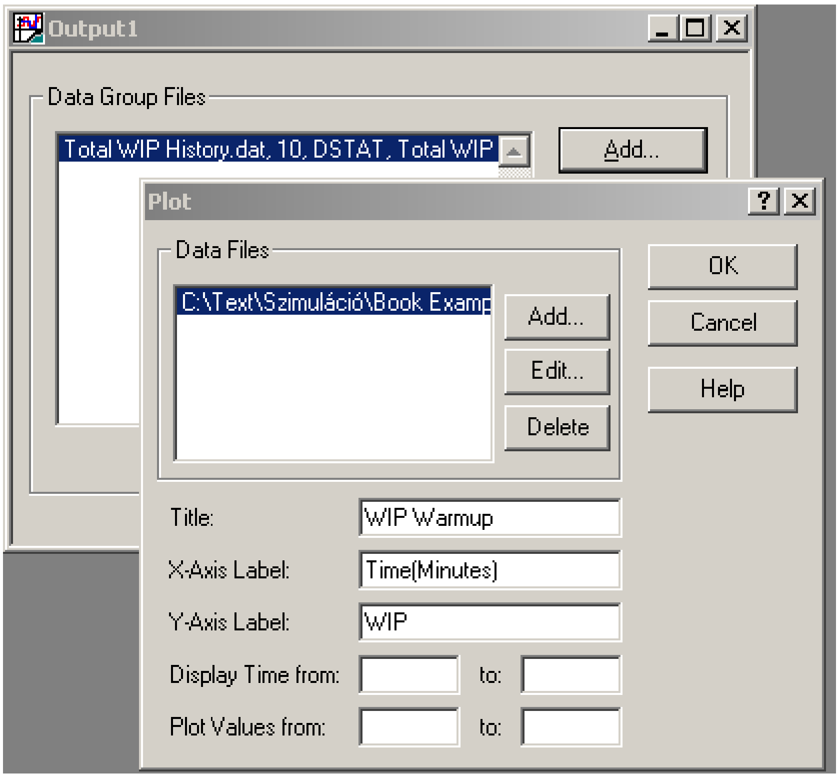
Statistic - Advanced Process

	Name	Type	Expression	Report Label	Output File
1	Total WIP	Time-Persistent	EntitiesWIP(Part 1) + EntitiesWIP(Part 2) + EntitiesWIP(Part 3)	Total WIP	Total WIP History.dat

Figure 8.9. The Completed Total WIP Entry in the **Statistic** module

Since we aren't interested in animation at this point, we pulled down the *Run>Run Control* menu and checked Batch Run (*No Animation*) to speed things up. We also unchecked everything under Statistics Collection in *Run>Setup>Project Parameters*, as well as Record Entity Statistics in the **Dispose** module, to increase speed further. To get a feel for the variation, we specified 10 for the Number of Replications in *Run>Setup>Replication Parameters*; since we're now interested in long-run steady-state performance, we increased the Replication Length from 32 Hours (1.33 Days) to 5 Days. We freely admit that these are pretty much arbitrary settings, and we settled on them after some trial and error. For the record, it only took about five seconds to run all this on a humble 366 MHz Pentium II note-book.

To make a plot of WIP vs. time in the Output Analyzer (see Section 5.8.4 for the basics of the Output Analyzer), we created a new data group and added the file Total WIP History . dat to it. We then selected Plot (📈 or Graph/Plot), Added the .dat file (selecting All in the Replications field of the Data File dialog), typed in a Title, and changed the Axis Labels; the resulting dialogs are shown in *Figure 8.9*.



Total WIP History.dat, 10, DSTAT, Total WIP

C:\Text\Szimuláció\Book Examp

Title: WIP Warmup

X-Axis Label: Time(Minutes)

Y-Axis Label: WIP

Display Time from: to:

Plot Values from: to:

Figure 8.10. The Output Analyzer's Plot Dialog

Figure 8.11 shows the resulting plot of WIP across the simulations, with the curves for all ten replications superimposed. From this plot, it seems clear that as far as the WIP output is concerned, the run length of five days (7,200 minutes, in the Base Time Units used by the plot) is enough for the model to have settled out, and also it seems fairly clear that the model isn't "exploding" with entities, as would be the case if processing couldn't keep up with arrivals. As for a Warm-up Period, the effect of the empty-and-idle initial conditions for the first several hundred minutes on WIP is evident and consistent across replications, but it looks like things settle down after about 2,000 minutes; rounding up a little to be conservative, we'll select 2 Days (2,880 minutes) as our Warm-up Period.

If we'd made the runs (and plot) for a longer time period, or if the model warm up occurred more quickly, it could be difficult to see the appropriate Warm-up Period on the left end of the curves. If so, you could "zoom in" on just the first part of each replication. via the "Display Time from ... to .. "area in the Plot dialog.

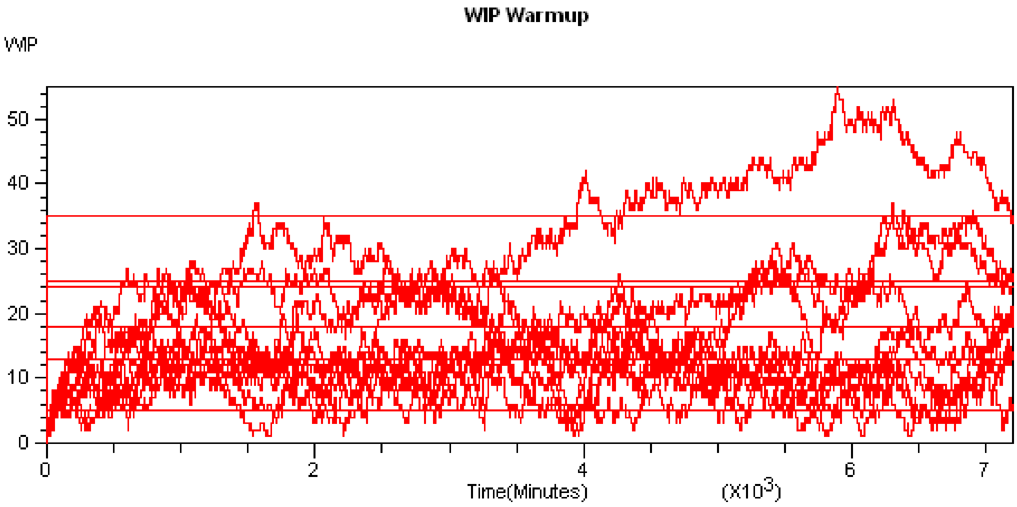


Figure 8.11. Within-Run WIP Plots for Model 7-2

In models where you're interested in multiple output performance measures, you'd need to make a plot like *Figure 8.11* for each output. There could be disagreement between the output measures on warm-up rate, in which case the safe move is to take the maximum of the individual warm-ups as the one to use overall.

8.2.2 Truncated Replications

If you can identify a warm-up period, and if this period is reasonably short relative to the run lengths you plan to make, then things are fairly simple for steady-state statistical analysis: just make IID (that's independent and identically distributed, in case you've forgotten) replications, as was done for terminating simulations in Section 5.8, except that you also specify a Warm-up Period for each replication in Run>Setup>Replication>Parameters. With these appropriate warm-up and run-length values specified, you just proceed to make independent replications and carry out your statistical analyses as in Section 5.8 with warmed-up independent replications so that you're computing steady-state rather than terminating quantities. Life is good.

This idea applies to comparing or selecting alternatives (Sections 5.8.4 and 5.8.5) and optimum seeking (Section 5.8.6). as well as to single-system analysis. However, there could be different warm-up and run-length periods needed for different alternatives; you don't really have the opportunity to control this across the different alternatives that **OptQuest** might decide to consider, so you should probably run your model ahead of time for a range of possible scenarios and specify the warm-up to be (at least) the maxi-mum of those you experienced.

For the five-day replications of Model 7-3, we entered in *Run>Setting>Replication Parameters* the 2-Day Warm-Up Period we determined in Section 8.3.1, and again asked for ten replications. Since we no longer need to save the within-run history, we deleted the Output File entry m the **Statistic** module. We called the resulting model Model 7-4 (note our practice of saving successive changes to models under different names so we don't cover up our tracks in case we need to backtrack at some point). The result was a 95% confidence interval for expected average WIP of 16.39 ± 6.51: from Model 7-3 with no warm up, we got 15.35 ± 4.42, illustrating the downward-biasing effect of the initial low-congestion period. To see this difference more clearly, since these confidence intervals are pretty wide, we made 100 replications (rather than 10) and got 15.45 ± 1.18 for Model 7-4 and 14.42 ± 0.86 for Model 7-3.

Note that the confidence intervals, based on the same number of replications, are wider for Model 7-4 than for Model 7-3; this is because each replication of Model 7-4 uses data from only the last three days whereas each replication of Model 7-3 uses the full five days, giving it lower variability (at the price of harmful initialization bias).

If more precision is desired in the form of narrower confidence intervals, you could achieve it by simulating some more. Now, however, you have a choice as to whether to make more replications with this run length and warm up or keep the same number of replications and extend the run length. (Presumably the original warm-up would still be adequate.) It's probably simplest to stick with the same run length and just make more replications, which is the same thing as increasing the statistical sample size. There is something to be said, though, for extending the run lengths and keeping the same number of replications; the increased precision with this strategy comes not from increasing the "sample size" (statistically, degrees of freedom) but rather from decreasing the variability of each within-run average since it's being taken over a longer run. Furthermore, by making the runs longer, you're even more sure that you're running long enough to "get to" steady state.

If you can identify an appropriate run length and warm-up period for your model, and if the warm-up period is not too long, then the truncated-replication strategy is quite appealing. It's fairly simple, relative to other steady-state analysis strategies, and gives you truly independent observations (the results from each truncated replication), which is a big advantage in doing statistical analysis. This goes not only for simply making confidence intervals as we've done here, but also for comparing alternatives as well as other statistical goals.

8.2.3 Batching in a Single Run

Some models take a long time to warm up to steady state, and since each replication would have to pass through this long warm-up phase, the truncated-replication strategy of Section 8.3.2 can become inefficient. In this case, it might be better to make just one really long run and thus have to "pay" the warm up only once. We modified Model 7-4 to make a single replication of length 50 days including a (single) warm up of two days (we call this Model 7-5); this is the same simulation "effort" as making the ten replications of length five days each, and it took about the same amount of computer time. Since we want to plot the within-run WIP data, we reinstated a file name entry in the Output File field in the **Statistic** data module, calling it *Total WIP History One Long Run.dat* (since it's a long run, we thought it deserved a long file name too). *Figure 8.12* plots Total WIP across this run. (For now, ignore the thick vertical bars we drew in the plot.)

The difficulty now is that we have only one "replication" on each performance measure from which to do our analysis, and it's not clear how we're going to compute a variance estimate, which is the basic quantity needed to do statistical inference. Viewing each individual observation or time-persistent value within a run as a separate "data point" would allow us to do the arithmetic to compute a within-run sample variance, but doing so is extremely dangerous since the possibly heavy correlation (see Section C.2.4 in Appendix C) of points within a run will cause this estimate to be severely biased with respect to the true variance of an observation. Somehow, we need to take this correlation into account or else "manufacture" some more "independent observations" from this single long run in order to get a decent estimate of the variance.

There are several different ways to proceed with statistical analysis from a single long run. One relatively simple idea (that also appears to work about as well as other more complicated methods) is to try to manufacture almost-uncorrelated "observations" by breaking the run into a few large batches of many individual points, compute the aver-ages of the points within each batch, and treat them as the basic IID observations on which to do statistical analysis (starting with variance estimation). These batch means then take the place of the means from within the multiple replications in the truncated-replication approach we've replaced the replications by batches. In the WIP plot of *Figure 8.12*, the thick vertical dividing lines illustrate the idea; we'd take the time-average WIP level between the lines as the basic "data" for statistical analysis. In order to obtain an unbiased variance estimate, we need the batch means to be nearly uncorrelated, which is why we need to make the batches big; there will still be some heavy correlation of individual points near either side of a batch boundary, but these points will be a small minority within their own large batches, which we hope will render the batch means nearly uncorrelated. In any case, Arena will try to make the batches big enough to look uncorrelated, and will let you know if your run was too short for it to manufacture batch means that "look" (in the sense of a statistical test) nearly uncorrelated, in which case, you'd have to make your run longer.

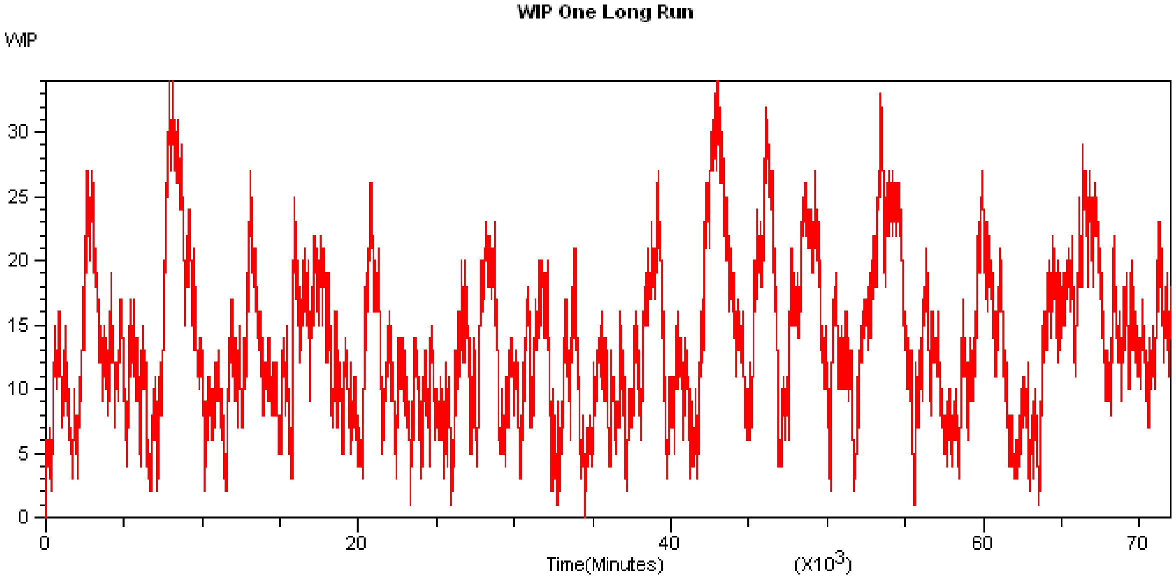


Figure 8.12. Total WIP Over a Single Run of 50 Days

As we just hinted, Arena automatically attempts to compute 95% confidence intervals via batch means for the means of all output statistics and gives you the results in the Half Width column next to the Average column in the report for each replication. If you're making just one replication, as we've been discussing, this is your batch-means confidence interval. On the other hand, if you've made several replications, you see this in the Category by Replication report for each replication; the Half Width in the Category Over-view report is across replications, as discussed in Sections 5.8 and 8.3.2. We say "attempts to compute" since internal checks are done to see if your replication is long enough to produce enough data for a valid batch-means confidence interval on an output statistic; if not, you get only a message to this effect, without a half-width value (on the theory that a wrong answer is worse than no answer at all) for this statistic. If you've specified a Warm-up Period, data collected during this period are not used in the confidence-interval calculation. To understand how this procedure works, think of Arena as batching "on the fly" (i.e., observing the output data during your run and throwing them into batches as your simulation goes along).

So what does "enough data" mean? There are two levels of checks to be passed, the first of which is just to get started. For a Tally statistic, Arena demands a minimum of 320 observations. For a *Dstat* statistic, you must have had at least five units of simulated time during which there were at least 320 changes in the level of the discrete-change variable. Admittedly, these are somewhat arbitrary values and conditions, but we need to get started somehow, and the more definitive statistical test, which must also be passed, is done at the end of this procedure. If your run is not long enough to meet this first check, Arena reports "(Insufficient)" in the Half-Width column for this variable. Just making your replication longer should eventually produce enough data to meet these getting-started requirements.

If you have enough data to get started, Arena then begins batching by forming 20 batch means for each *Tally* and *Dstat* statistic. For *Tally* statistics, each batch mean will be the average of 16 consecutive observations; for *Dstat* statistics, each will be the time average over 0.25 base time unit. As your run progresses, you will eventually accumulate enough data for another batch of these same "sizes," which is then formed as batch number 21. If you continue your simulation, you'll get batches 22, 23, and so on, until you reach 40 batches. At this point, Arena will re-batch these 40 batches by averaging batches one and two into a new (and bigger) batch one, batches three and four into a new (bigger) batch two, etc., so that you'll then be back to 20 batches, but each will be twice as "big." As your simulation proceeds, Arena will continue to form new batches (21, 22, and so on), each of this larger size, until again 40 batches are formed, when re-batching back down to 20 is once again performed. Thus, when you're done, you'll have between 20 and 39 complete batches of some size. Unless you're really lucky, you'll also have some data left over at the end in a partial batch that won't be used in the confidence-interval computation. The reason for this re-batching into larger batches stems from an analysis by Schmeiser (1982), which demonstrated that there's no advantage to the other option, of continuing to gather more and more batches of a fixed size to reduce the half width, since the larger batches will have inherently lower variance, compensating for having fewer of them. On the other hand, having batches that are too small, even if you have a lot of them, is dangerous since they're more likely to produce correlated batch means, and thus an invalid confidence interval.

The final check is to see if the batches are big enough to support the assumption of independence between the batch means. Arena tests for this using a statistical hypothesis test due to Fishman (1978). If this test is passed, you'll get the Half Width for this output variable in your report. If not, you'll get "(Correlated)" indicating that your process is evidently too heavily autocorrelated for your run length to produce nearly uncorrelated batches; again, lengthening your run should resolve this problem, though depending on the situation, you may have to lengthen it a lot.

Returning to Model 7-5, after making the one 50-day run and deleting the first two days as a warm-up period, Arena produced in the Category Overview report a 95% batch-means confidence interval of 13.6394 + 1.38366 on expected average WIP. The reason this batch-means confidence interval shows up here in the Category Overview report is that we've only made a single replication.

In most cases, these automatic batch-means confidence intervals will be enough for you to understand how precise your averages are, and they're certainly easy (requiring no work at all on your part). But there are a few considerations to bear in mind. First, don't forget that these are relevant only for steady-state simulations; if you have a terminating simulation (Section 5.8), you should be making independent replications and doing your analysis on them, so you should ignore these automatic batch-means confidence intervals. Secondly, you still ought to take a look at the Warm-up Period for your model, as in Section 8.3.1, since the automatic batch-means confidence intervals don't do anything to correct for initialization bias if you don't specify a Warm-up Period. Finally, you can check the value of the automatic batch-means half width as it is computed during your run, via the Arena variables THALF(Tally ID) for Tally statistics and DHALF(Dstat ID) for Dstat statistics; one reason to be interested in this is that you could use one of these for a critical output measure in the Terminating Condition field of your Simulate module to run your model long enough for it to become small enough to suit you; see Section 12.5 for more on this and related ideas.

We should mention that, if you really want, you can decide on your own batch sizes, compute and save the batch means, then use them in statistical procedures like confidence-interval formation and comparison of two alternatives (see Section 5.8). Briefly, the way this works is that you save your within-run history of observations just as we did to make our warm-up-determination plots, read this file into the Output Analyzer, and use its *Batch/Truncate* capability to do the batching and averaging, saving the batch means that you then treat as we did the cross-replication means in Section 5.8. When batching, the Output Analyzer performs the same statistical test for uncorrelated batches, and will let you know if the batch size you selected is too small. Some reasons to go through this are if you want something other than a 95% confidence level, if you want to make the most efficient use of your data and minimize waste at the end, or if you want to do a statistical comparison between two alternatives based on their steady-state performance. However, it's certainly a lot more work.

8.2.4 What To Do?

We've shown you how to attack the same problem by a couple of different methods and hinted that there are a lot more methods out there. So which one should you use? As usual, the answer isn't obvious (we warned you that steady-state output analysis is difficult). Sometimes there are tradeoffs between scientific results and conceptual (and practical) simplicity, and that's certainly the case here.

In our opinion (and we don't want to peddle this as anything more than opinion), we might suggest the following list, in decreasing order of appeal:

Try to get out of doing a steady-state simulation altogether by convincing yourself (or your patron) that the appropriate modeling assumptions really entail specific starting and stopping conditions. Go to Section 5.8 (and don't come back here).

If your model is such that the warm up is relatively short, probably the simplest and most direct approach is truncated replication. This has obvious intuitive appeal, is easy to do (once you've made some plots and identified a reasonable warm-up period), and basically winds up proceeding just like statistical analysis for terminating simulations, except for the warm up. It also allows you to take advantage of the more sophisticated analysis capabilities in PAN and OptQuest.

3. If you find that your model warms up slowly, then you might consider batch means, with a single warm up at the beginning of your single really long run. You could either accept Arena's automatic batch-means confidence intervals or handcraft your own. You cannot use the statistical methods in PAN or OptQuest with the batch-means approach, however (part of the reason this is last in our preference list).

8.2.5 Other Methods and Goals for Steady-State Statistical Analysis

We've described two different strategies (truncated replications and batch means) for doing steady-state statistical analysis; both of these methods are available in Arena. This has been an area that's received a lot of attention among researchers, so there are a number of other strategies for this difficult problem: econometric time-series modeling, spectral analysis from electrical engineering, regenerative models from stochastic processes, standardized time series, as well as variations on batch means like separating or weighting the batches. If you're interested in exploring these ideas, you might consult Chapter 9 of Law and Kelton (2000), a survey paper like Sargent, Kang, and Goldsman (1992), or peruse a recent volume of the annual *Proceedings of the Winter Simulation Conference*, where there are usually tutorials, surveys, and papers covering the latest developments on these subjects.

8.3 Chapter summary

Now you should have a very good set of skills for carrying out fairly detailed modeling and have an understanding of (and know what to do about) issues like verification and steady-state statistical analysis. In the following chapter, we'll expand on this to show you how to model complicated and realistic material-handling operations. In the chapters beyond, you'll drill down deeper into Arena's modeling and analysis capabilities to exploit its powerful and flexible hierarchical structure.

CHAPTER 9

Entity Transfer

Up to now, we've considered two different ways to direct an entity's flow through a model. We've had them move from module to module with no travel time via Connections. In other models, we've moved them by Routing between stations with some transit-time delay. In both cases, the entities proceed uninhibited, as though they all had their own feet and there was enough room in the transit ways for as many of them at a time as wanted to be moving.

Of course, things aren't always so nice. There could be a limit on the concurrent number of entities in transit, such as a communications system where the entities are packets of information and the bandwidth is limited to a certain number of packets in transit at a time. There could also be situations in which something like a forklift or a person needs to come pick up an entity and then transport it. There are also different kinds of conveyors where entities can be viewed as competing for space on the belt or line. We'll explore some of these modeling ideas and capabilities in this chapter. It is often important to model such entity transfer accurately since studies have shown that delays and inefficiencies in operations might be caused more by the need just to move things around rather than in actually doing the work.

Section 9.1 discusses in more detail the different kinds of entity movement and transfers and how they can be modeled. In Section 9.2, we'll indicate how you can use the Arena modeling tools you already have to represent a constraint on the number of entities that can be in motion at a time (though all entities still have their own feet). Transport devices like forklifts, pushcarts, and (of course) people are taken up in Section 9.3. Modeling conveyors of different types is described in Section 9.4.

After reading this chapter and considering the examples, you'll be able to model a rich variety of entity movement and transfer that can add validity to your model and realism to your animations.

9.1 Types of Entity Transfers

To transfer entities between modules, we initially used the Connect option (Chapter 3) to transfer entities directly between modules with no time delay. In Chapter 4, we introduced the concept of a Route that allows you to transfer entities between stations allowing a time delay in the transfer. We first showed how to use Routes for entity transfer to a specific station; then we generalized this concept in Chapter 8 by using Sequences.

Although this gives us the ability to model most situations, we sometimes find it necessary to limit or constrain the number of transfers occurring at any point in time. For example, in modeling a communications network, the links have a limited capacity. Thus, we must have a method to limit the number of simultaneous messages that are being transferred by each network link or for the entire network. The solution is rather simple; we think of the network links as resources with a capacity equal to the number of simultaneous messages allowed. If the capacity depends on the size of the messages, then we define the resource capacity in terms of this size and require each message to seize the required number of resources, determined by its size, before it can be transferred. Let's call this type of entity transfer resource constrained, and we'll discuss it in more detail in Section 9.2.

Using a resource to constrain the number of simultaneous transfers may work fine for a communications network, but it doesn't allow us to model accurate ly an entire class or category of entity transfers generally referred to as material handling. The modeling requirements for different material-handling systems can vary greatly, and the number of different types of material-handling devices is enormous. In fact, there is an entire handbook devoted to this subject (see Kulwiec, 1985). However, it's possible to divide the se devices into two general categories based on their modeling requirements.

> The first category constrains the number of simultaneous transfers based on the number of individual material-handling devices available. Material-handling devices that fall into this category are carts, hand trucks, fork trucks, AGVs, people, and so on. However, there is an additional requirement in that each of these devices has a physical location. If a transfer is required, we may first have to move the device to the location where the requesting entity resides before we can perform the actual transfer. From a modeling standpoint, you might think of these as moveable resources, referred to in Arena as Transporters.
>
> The second category constrains the ability to start a transfer based on space availability. It also requires that we limit the total number of simultaneous transfers between two locations, but this limit is typically based on the space requirement. Material-handling devices that fall into this category include conveyors, overhead trolleys, power-and-free systems, tow lines, and more. An escalator is a familiar example of this type of material-handling device. If a transfer is required, we first have to obtain the required amount of available or unoccupied space on the device, at the location where we are waiting, before we can initiate our transfer. These devices require a significantly different modeling capability, referred to in Arena as Conveyors.

The Arena Transporter and Conveyor constructs allow us to model almost any type of material-handling system easily. However, there are a few material-handling devices that have very unique requirements that can create a modeling challenge. Gantry or bridge cranes are classic examples of such devices. A single crane is easily modeled with the transporter constructs. If you have more than one crane on a single track, the method used to control how the cranes move is critical to modeling these devices accurately. Unfortunately, almost all systems that have multiple cranes are controlled by the crane operators who generally reside in the cabs located on the cranes. These operators can see and talk to one another, and their decisions are not necessarily predictable. In this case, it is easy to model the cranes; the difficult part is how to incorporate the human logic that prevents the cranes from colliding or gridlocking.

Thus, we've defined three types of constrained entity transfers: resource constrained, transporters, and conveyors. We'll first briefly discuss resource-constrained transfers in Section 9.2, then introduce transporters and conveyors by using them in our small manufacturing system from Chapter 8 (Model 8-1) in Sections 9.3 and 9.4.

9.2 Model 9-1. The Small Manufacturing System with Resource-Constrained Transfers

In our initial model (Model 9-1) of the small manufacturing system, we assumed that all transfer times were two minutes. If these transfer times depend on the availability of material-handling devices, the actual times may be quite different from each other during operation of the system. Because of this, the earlier model might give us reasonable estimates of the potential system capacity; however, it would most likely not provide very accurate estimates on the part cycle times. The simplest method is to include resource-constrained transfers. So let's assume that our transfer capacity is two, with the same two-minute transfer time in every case. This means that a maximum of two transfers can occur at the same time. If other entities are ready for transfer, they will have to wait until one of the ongoing transfers is complete.

Using a resource to constrain the number of simultaneous entity transfers is a relatively easy addition to a model. We need to define what we think of as a new kind of transferring resource, seize one unit of that resource before we initiate our route to leave a location, and release the resource when we arrive at the next station or location. This is an ordinary Arena resource, but we're thinking of it differently to model the transfers.

There are two different ways we could add this logic to our existing model. The most obvious (at least to us) is to insert a **Seize** module from the Advanced Process panel just prior to each of the existing **Route** modules (there are five in the current model). Each **Seize** module would request one unit of the transfer resource; e.g., Transfer. We would also need to change the capacity of this new resource to 2 in the **Resource** data module. There is one additional concept that you should consider while making these modifications, "Should each **Seize** module have a separate queue?" If you use a single *Shared Queue* for all of our newly inserted **Seize** modules, then the resource would be allocated based on a FIFO rule. If you specified a different queue at each **Seize** module, you would be able to apply priorities to the selection process. For example, you might want to give the new arriving entities top priority. This would allow the newly arriving parts to be sent to their first operation as soon as possible, in the hope that it would allow the processing of the part to start with a resulting reduction in the part cycle time. (If you spend much time thinking about this, you might conclude that this is faulty logic.) If all priorities are the same, you end up with the same FIFO rule as with the shared-queue approach. So it would appear that the two approaches are the same. This is not quite true if you plan to show the waiting parts in your animation (which we do). By using separate queues (the default) prior to each route, we can show the waiting parts easily on our animation.

Having seized the transfer resource prior to each part transfer via a route, we now need to release that resource once the part arrives at its destination. We accomplish this

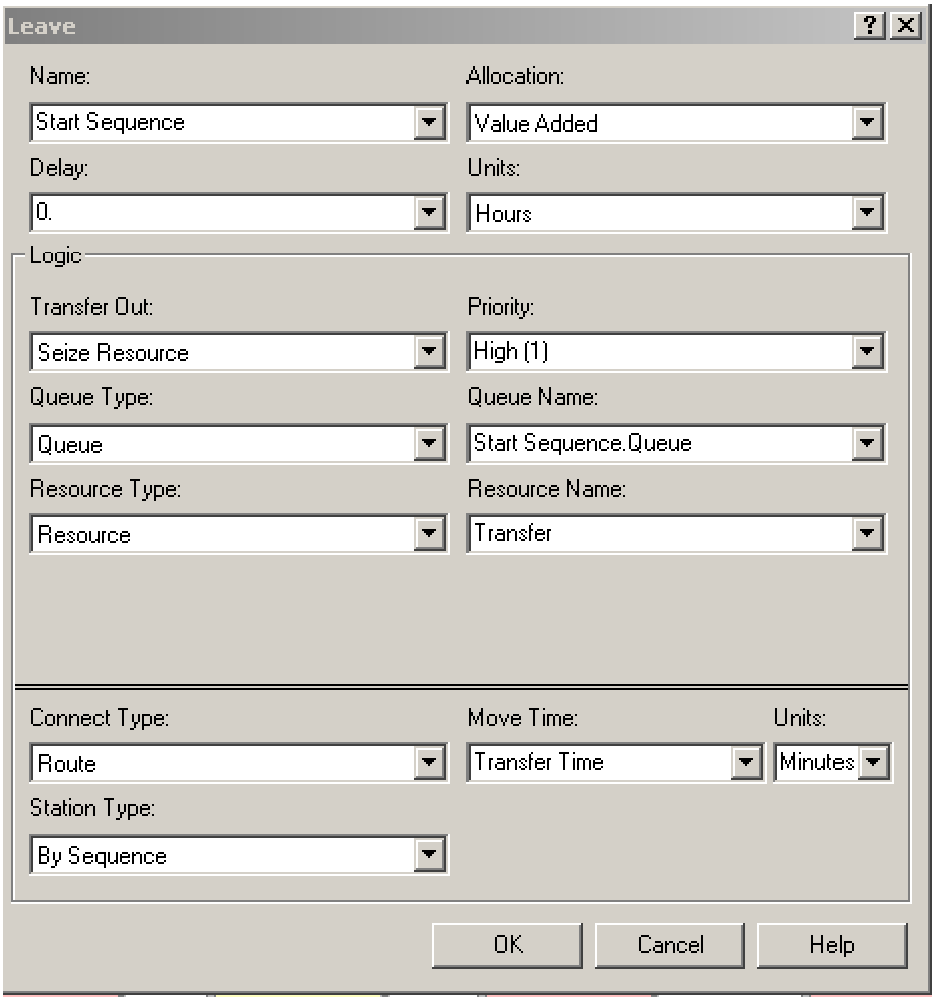
by inserting a **Release** module after each **Station** module in the model. (The single exception is Order Release Station, which is used only to tell Arena the location of the newly arrived parts.) In each of these modules, we simply release the Transfer resource.

The second method is to replace the **Route** modules with **Leave** modules (from the Advanced Transfer panel) and the **Station** modules with **Enter** modules (also from the Advanced Transfer panel). The **Leave** module allows us to seize the resource and also to route the part in the same module. Similarly, the **Enter** module allows us to combine the features of the **Station** and **Release** modules. We'll choose this second method because it allows us to introduce two new modules, and it facilitates the conversion of the model to include other transfer features.

The **Leave** module allows you to seize a transferring resource before leaving the station. Which resource to seize can be defined as a particular resource, a resource set, a specific member of a set, or a resource based on the evaluation of an expression. You can also specify a Priority in the Transfer Out logic that would allow you to control which entity would be given the transferring resource if more than one entity was waiting. The smaller the number, the higher the priority. For our system, you will only have to select the *Seize Resource* option and enter a transferring resource name. We'll also have to enter the additional routing information.

Let's start by replacing the *Start Sequence* **Route** module with the **Leave** module shown in *Display 9.1*. We'll use the same module name, *Start Sequence*, select the Transfer Out logic as *Seize Resource*, and enter the Resource Name as *Transfer*. Finally, we select Route as the Connect Type, enter Transfer Time for the Move Time, select Minutes, and select Sequence as the Station Type. We've defaulted on the remaining boxes. Note that this gives us our individual queues for parts waiting for the Transfer resource. Notice also that with the selection of the Transfer Out resource logic, a queue (*Start Sequence.Queue*) was added to the module when the dialog box was accepted. At this point, you might want to move this queue to the proper position on your animation.

Now you can replace the remaining four **Route** modules with **Leave** modules. With the exception of the Name (we used the same name that was on the replaced **Route** module), the data entries are the same as shown in *Display 9.1*.



Name: Start Sequence
Allocation: Value Added
Delay: 0.
Units: Hours
Logic
Transfer Out: Seize Resource
Priority: High (1)
Queue Type: Queue
Queue Name: Start Sequence.Queue
Resource Type: Resource
Resource Name: Transfer
Connect Type: Route
Move Time: Transfer Time
Units: Minutes
Station Type: By Sequence

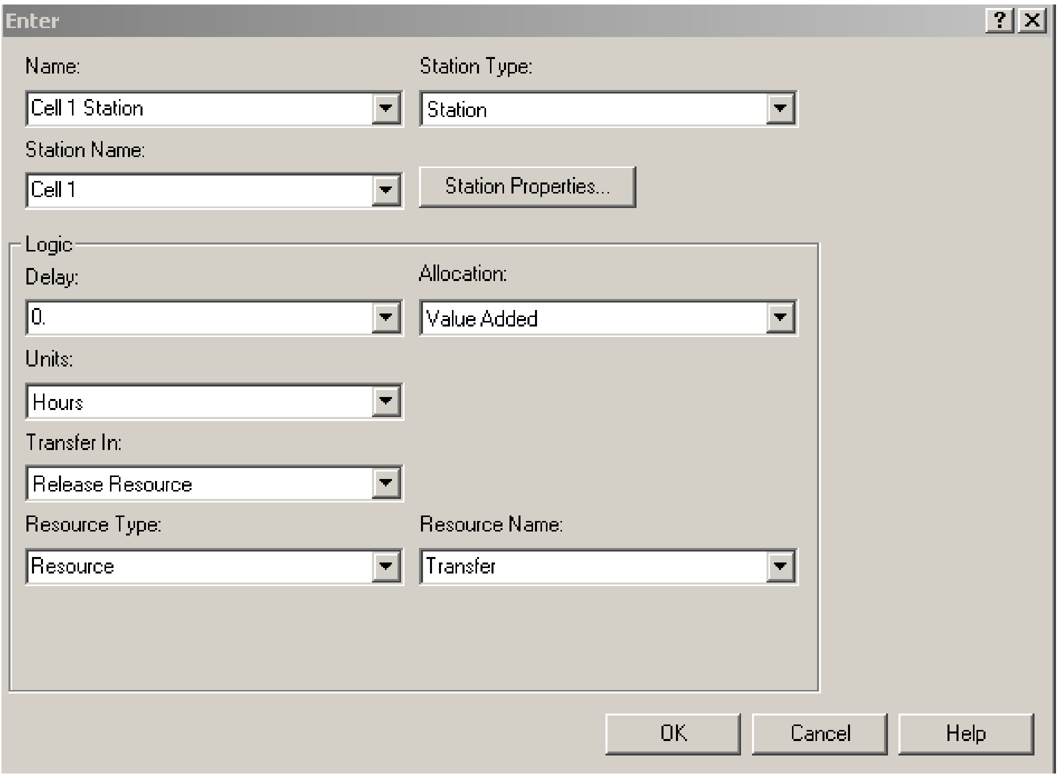
Name	Start Sequence
Logic TransferOut ResourceName	 Seize Resource Transfer
Connect Type Move Time Units Station Type	Route Transfer Time Minutes Sequence

Display 9.1. The **Leave** module for Resources

The Enter module allows you to release the transferring resource and provides the option of including an unload delay time. Let's start with the replacement of the *Cell 1 Station* **Station** module with an **Enter** module, as in *Display 9.2*. Delete the existing **Station** module and add the new **Enter** module. We'll use the same Name, *Cell 1 Station*, and Select *Cell 1* from the drop-down list for the Station Name. In the logic section, you need only select *Release Resource* for the Transfer In entry and select the *Transfer* resource for the Resource Name to be released.

Replace the remaining four **Station** modules with **Enter** modules (but not the Order Release Station). As before, the data entries are the same as shown in *Display 9.2*, with the exception of the Name (we used the same name that was in the replaced **Station** module).

Provided that you have already moved the new wait queues to the animation, we are almost ready to run our model. Before we do that, you should be aware that we have not yet defined the capacity of our constraining resource, which determines the maximum number of parts that could be in transit at one time. Although we have defined the existence of this resource, Arena defaults to a capacity of 1 for it. You'11 need to open the **Resource** data module (Basic Process panel) and change the capacity of the *Transfer* resource to 2. The resulting model looks identical to Model 9-1 because our replacements were one-for-one and we used the same names. We used the same animation, with the addition of the five new wait queues. The positions of these queues are shown in *Figure 9.1*.



Name: Cell 1 Station

Station Type: Station

Station Name: Cell 1

Logic

Delay: 0.

Allocation: Value Added

Units: Hours

Transfer In: Release Resource

Resource Type: Resource

Resource Name: Transfer

Name Station Name	Cell 1 Station Cell 1
Logic Transfer In Resource Name	Release Resource Transfer

Display 9.2. The **Enter** module for Resources

If you run this model and watch the animation, you'll quickly observe that there is seldom a part waiting for transfer. If you have doubts about the correctness of your model, change the capacity of the Transfer resource to 1 and watch the animation. If you compare the results of this model (with two Transfer resources) to the results of Model 9-1, you'll see a slight increase in the part total time in system due to time spent waiting in a queue for the Transfer resource. The difference is between 3 and 9 minutes, depending on part type.

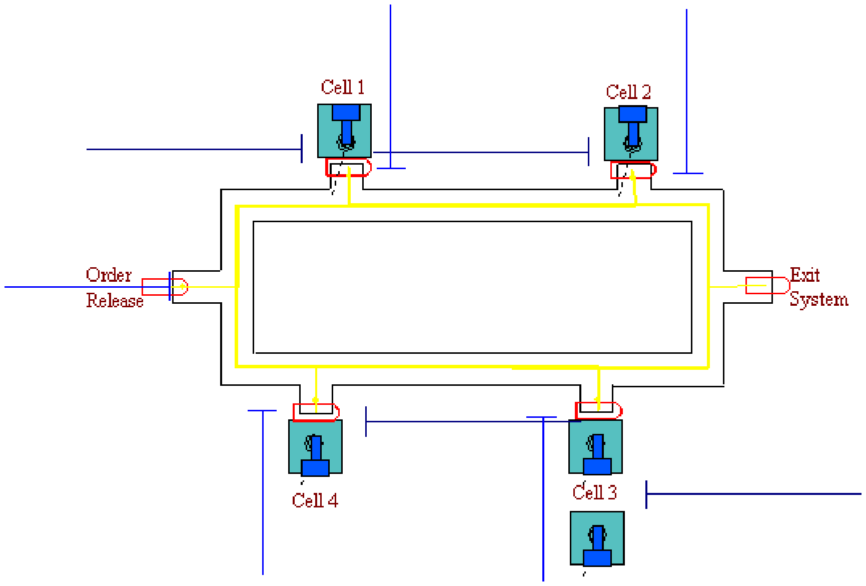


Figure 9.1. The Revised Small Manufacturing System Animation

9.3 The Small Manufacturing System with Transporters

Let's now assume that all material transfers are accomplished with some type of Transporters such as carts, hand trucks, or fork trucks (we'll call them carts). Let's further assume that there are two of these carts, each moving at 50 feet per minute, whether loaded or empty. Each cart can transport only one part at a time, and there are 0.25 minute load and unload times at the start and end of each transport. We'll provide the distances between stations after we've added the carts to our model.

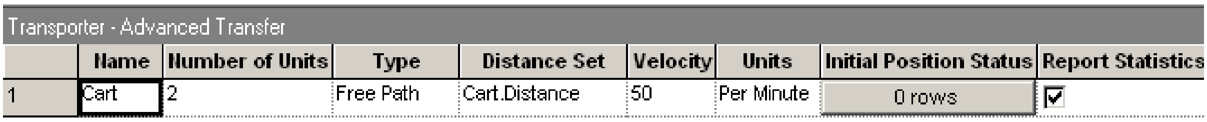
There are two types of Arena Transporters: *Free-Path* and *Guided*. Free-Path Transporters can move freely through the system without encountering delays due to congestion. The time to travel from one point to another depends only on the transporters velocity and the distance to be traveled. Guided Transporters are restricted to moving within a predefined network. Their travel times depend on the vehicles' speeds, the network paths they follow, and potential congestion along those paths. The most common type of guided vehicle is an *automated guided vehicle* (AGV). The carts for our system fall into the free-path category.

The transfer of a part with a transporter requires three activities: Request a transporter, Transport the part, and Free the transporter. The key words are Request, Transport, and Free. The Request activity, which is analogous to seizing a resource, allocates an available transporter to the requesting entity and moves the allocated transporter to the location of the entity, if it's not already there. The *Transport* activity causes the transporter to move the entity to the destination station. This is analogous to a route, but in the case of a Transport, the transporter and entity move together. The Free activity frees the transporter for the next request, much like the action of releasing a resource. If there are multiple transporters in the system, we face two issues regarding their assignment to entities. First, we might have the situation during the run where an entity requests a transporter and more than one is available. In this case, a *Transporter Selection Rule* dictates which one of the transporter units will fulfill the request. In most modeled systems, the *Smallest Distance* rule makes sense-allocate the transporter unit that's closest to the requesting entity. Other rules include Largest Distance, though it takes a creative mind to imagine a case where it would be sensible. The second issue concerning transporter allocation arises when a transporter is freed and there are multiple entities waiting that have requested one. In this case, Arena applies a priority, allocating the transporter to the waiting entity that has the highest priority (lowest priority number). If there's a priority tie among entities, the transporter will be allocated to the closest waiting entity.

9.3.1 Model 9-2. The Modified Model 9-1 for Transporters

To represent the carts in our small manufacturing system model, let's start by first defining the carts. We do this with the **Transporter** data module found on the Advanced Transfer panel. If you're building the model along with us, we suggest that you start with a copy of Model 9-1, which we just completed. In this case, we need only enter the transporter Name, Capacity (number of units), and Velocity (*Display 9.3*). Transporters defined using the modules from the Advanced Transfer panel are assumed to be free path. We'll accept the default for the Distance Set name (we'll return to this concept later), time Units, Initial Position of the carts, and the Report Statistics.

We need to take care when defining the Velocity that we enter a quantity that's appropriate for the time and distance units we're using in the model. We defaulted our time Units to Per Minute, the same units as for our stated velocity, but we'll need to make sure that our distances are specified in feet.



Transporter - Advanced Transfer

	Name	Number of Units	Type	Distance Set	Velocity	Units	Initial Position Status	Report Statistics
1	Cart	2	Free Path	Cart.Distance	50	Per Minute	0 rows	☑

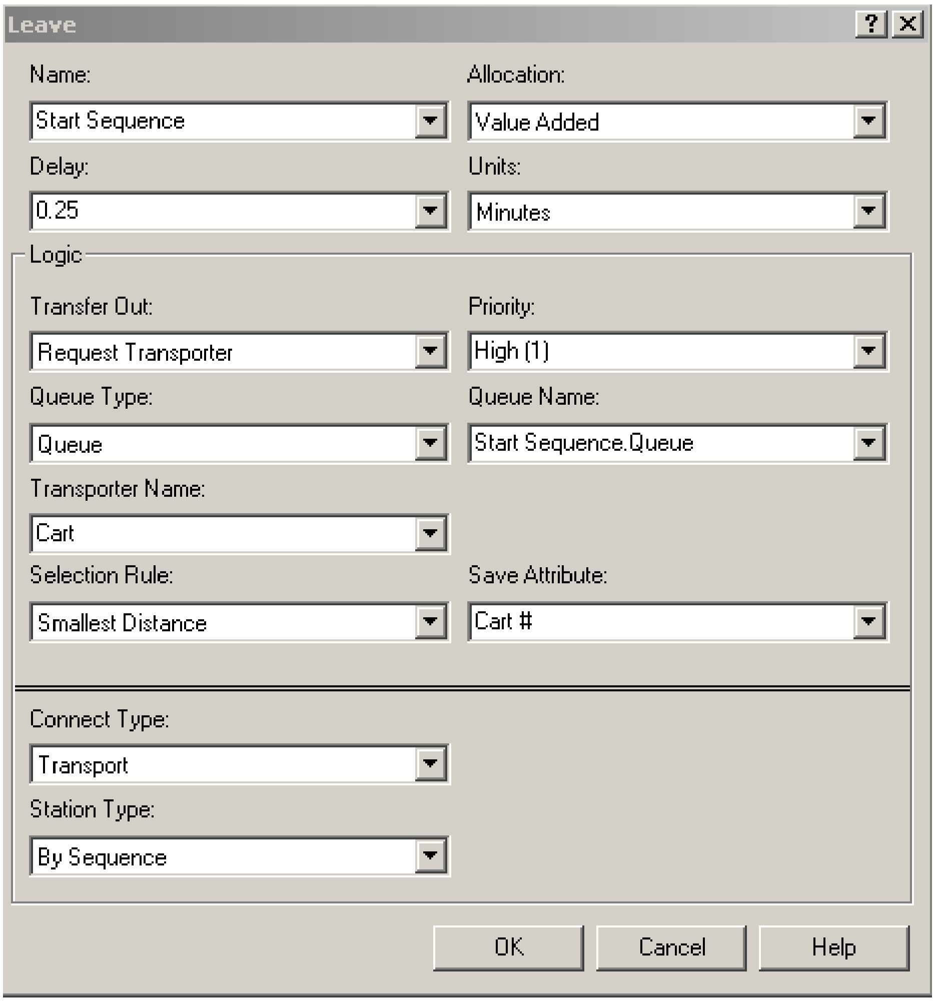
Name	Cart
Capacity	2
Velocity	50

Display 9.3. The **Transporter** data module

Having defined the transporter, we can now develop the picture that represents the cart. This is done in almost the same way as it was for an entity or resource. Clicking on the *Transporter* button (), found on the **Animate Transfer** toolbar, opens the Transporter Picture Placement window. Let's replace the default picture with a box (we used a 5-point line width) that is white, with a green line when the cart is idle and blue line when it's busy. When the cart is carrying a part, Arena also needs to know where on the cart's busy picture to position the part. This placement is similar to a seize point for a resource. In this case, it's called *a ride point* and can be found under the Object menu of the Arena Picture Editor window when you're editing the busy picture. Selecting *Object > Ride Point* changes your pointer to cross hairs. Move this pointer to the center of the cart and click. The ride point, which appears as ® on the busy cart, results in the entity being positioned so its reference point is aligned with this ride point. When you've finished creating your pictures, accept the changes to close the Transporter Picture Placement window your pointer becomes cross hairs that you should position somewhere near your animation and click. This action places the picture of your idle cart in the model window. You needn't worry about where you place the transporter as it is only placed in the model window in case you need to re-edit it later. During a run, this picture will be hidden, and replicas of it will move across the animation for each individual transporter unit.

To request the cart in the model logic, we'll use the same modules that we did for seizing a resource-the **Leave** modules. *Display 9.4* shows the entries for the Start Sequence **Leave** module. The other **Leave** module entries are identical, except for their names. We've entered a 0.25-minute delay to account for the load activity and selected the Request Transporter option from the lists for the Transfer Out type. The Selection Rule entry determines which transporter will be allocated if more than one is currently free: Cyclic, Random, Preferred

Order, Largest Distance, or Smallest Distance. The Cyclic rule attempts to cycle through the transporters, thus leveling their utilizations. The Preferred Order rule attempts always to select the available transporter with the lowest number. We've chosen the Smallest Distance rule, which results in an allocation of the cart closest to the requesting entity. A new Attribute, Cart #, is defined and used to save the number of the cart that was allocated (more on this later). We also select the Transport option for the Connect Type. Note that when you make this selection, the Move Time option disappears. The actual move time will be calculated from the distance traveled and the transporter velocity.



Name: Start Sequence
Allocation: Value Added
Delay: 0.25
Units: Minutes
Logic
Transfer Out: Request Transporter
Priority: High (1)
Queue Type: Queue
Queue Name: Start Sequence.Queue
Transporter Name: Cart
Selection Rule: Smallest Distance
Save Attribute: Cart #
Connect Type: Transport
Station Type: By Sequence

Name Delay Units	Start Sequence 0.25 Minutes
Logic Transfer Out Transporter Name Selection Rule Save Attribute	 Request Transporter Cart Smallest Distance Cart #
Connect Type Station Type	Transport Sequence

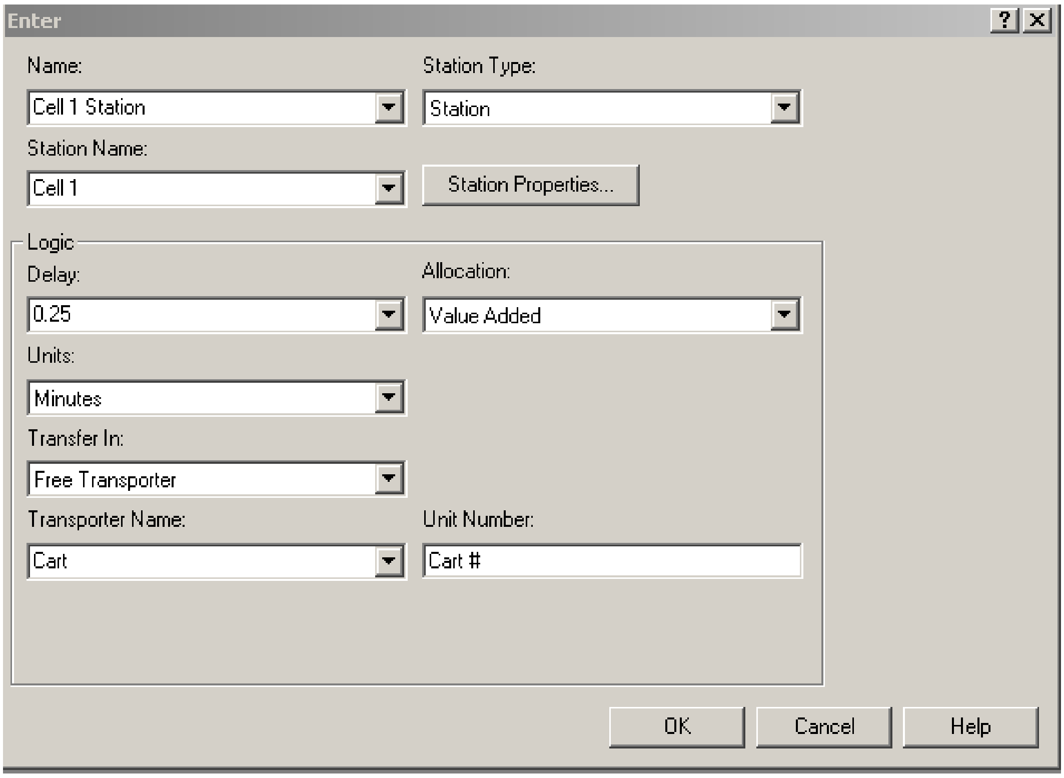
Display 9.4. The **Leave** module for Transporters

The **Leave** module performs four activities: allocate a transporter; move the empty transporter to the location of the requesting part, delay for the load time, and transport the part to its destination. There are alternate ways to model these activities. For example, we could have used a **Request** - **Delay** - **Transport** module combination to perform the same activities. The **Request** module (Advanced Transfer panel) performs the first two; the **Delay** module, the load activity; and the **Transport** module (Advanced Transfer panel), the last activity. Although this requires three modules, it does provide you with some additional modeling flexibility. You can specify different travel velocities for both moving the empty transporter and transporting the part to its destination location. In fact, you could specify the transporting velocity as an expression of an attribute of the part; for example, part weight. You could also replace the **Delay** module with a **Process** module, which could require the availability of an operator or material handler (modeled as a resource) to load the part. Finally, you could replace the **Request** module with an **Allocate** - **Move** module combination. The **Allocate** module (Advanced Transfer panel) allocates a transporter, while the **Move** module (Advanced Transfer panel) moves the empty transporter to the requesting part location. Again, this would allow you to insert additional logic between these two activities if such logic were required to model your system accurately.

When the part arrives at its next location, it needs to free the cart so it can be used by other parts. To free the cart, we'll use the same modules that we did for releasing the resource in the previous model-the **Enter** modules. *Display 9.5* shows the entries for the Enter Cell 1 **Station** module. The other **Enter** module entries are identical, except for their names. We've entered the unload time in minutes and selected the Free Transporter option for the Transfer In. In our model, we also entered the Transporter Name, Cart, and name of the attribute where we saved the transporter Unit Number, Cart #. We could have left both of these last two fields empty. Arena keeps track of which cart was allocated to the entity and frees that cart. However, if you enter the Transporter Name and there is more than one transporter, you also need to enter the Unit Number. If you only entered the Transporter Name, Arena would always try to free the first transporter.

As with the **Leave** module, the **Enter** module performs multiple activities: defining the station, delaying for the unload time, and freeing the transporter. Again there are alternate ways to model these functions. We could have used a **Station** - **Delay** - **Free** module combination to perform the same activities. The **Station** module defines the station; the **Delay** module, the unload activity; and the **Free** module (Advanced Transfer panel) frees the transporter. Separating these three activities would allow you to insert additional model logic, if required for your system. For example, you could replace the **Delay** module with a **Process** module, which could require the availability of an operator or material handler (modeled as a resource) to unload the part.

So far we've defined the carts, and we've changed the model logic to Request, Transport, and Free the carts to model the logic required to transfer the parts through the system. The actual travel time depends on the transporter velocity, which is defined with the carts, and the distance of the transfer. When we defined the carts (*Display 9.3*), we accepted a default reference to a Distance Set with the (default) name Cart Distance. We must now provide these distances so that Arena can reference them whenever a request or transport occurs. Let's start by considering only the moves from one station to the next station made by the parts as they make their way through the system (see *Table 7.1*). This information is provided in *Table 9.1*. These table entries include a pair of stations, or locations, and the distance (in feet) between those stations. For example, the first entry is for the move from Order Release to Cell 1, which is a distance of 37 feet. Blank entries in *Table 9.1* are for from-to pairs that don't occur (see *Table 7.1*).



Enter

Name: Cell 1 Station

Station Type: Station

Station Name: Cell 1

Logic

Delay: 0.25

Allocation: Value Added

Units: Minutes

Transfer In: Free Transporter

Transporter Name: Cart

Unit Number: Cart #

Name Station Name	Cell 1 Station Cell 1
Logic Delay Units Transfer In Transporter Name Unit Number	 0.25 Minutes Free Transporter Cart Cart #

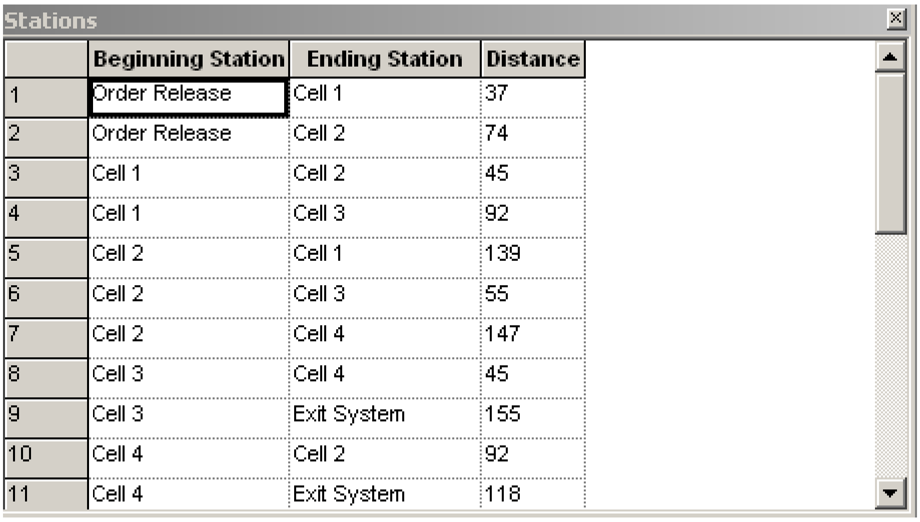
Display 9.5. The **Enter** mdule for Transporters

Table 9.1

The Part Transfer Distances

From/To	Cell 1	Cell 2	Cell 3	Cell 4	Exit System
Order Release	37	74			
Cell 1		45	92		
Cell 2	139		55	147	
Cell 3				45	155
Cell 4		92			118

We enter our distances using the **Distance** data module found on the Advanced Transfer panel. The 11 entries are shown in *Display 9.6*. You really only need to type the Distance since the first two entries can be selected from the lists. You should also note that a direction is implied, *from* Order Release *to* Cel1 1 in the case of the first row in *Display 9.6*. As was the case with the **Route** module, we can define the distance from Cel1 1 to Order Release if the distance or path is different (but no entity ever takes that route in this model, so it's unnecessary).

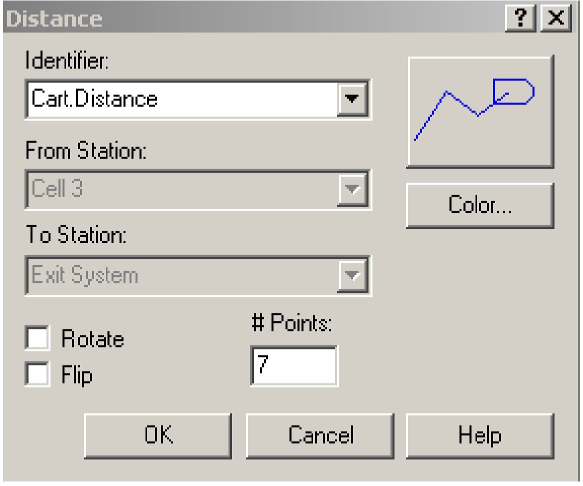


Stations

	Beginning Station	Ending Station	Distance
1	Order Release	Cell 1	37
2	Order Release	Cell 2	74
3	Cell 1	Cell 2	45
4	Cell 1	Cell 3	92
5	Cell 2	Cell 1	139
6	Cell 2	Cell 3	55
7	Cell 2	Cell 4	147
8	Cell 3	Cell 4	45
9	Cell 3	Exit System	155
10	Cell 4	Cell 2	92
11	Cell 4	Exit System	118

Display 9.6. The Part Transfer Distances

Now let's take a look at the animation component for these distances. We already defined the transporter picture, but we need to add our distances to the animation. If you're building this model along with us, now would be a good time to delete all the route paths (but leave the stations), before we add our distances. Add your distances by using the *Distance* button () found in the **Animate Transfer** toolbar. Click on this button and add the distances much as we added the route paths. When you transport a part, the cart (and the part) will follow these lines (or distances) on the animation. When you click on the *Distance* button, a dialog box appears that allows us to modify the path characteristics, as in *Display 9.7*. In this case, we requested that neither the cart nor the part riding on the cart be rotated or flipped as they move along the path. These features are not necessary as our pictures are all squares and circles and, for easy identification, we placed numbers on the parts (we'll leave it as a challenge to you to figure out which numbers go with which part type).



Distance

Identifier: Cart.Distance

From Station: Cell 3

To Station: Exit System

☐ Rotate

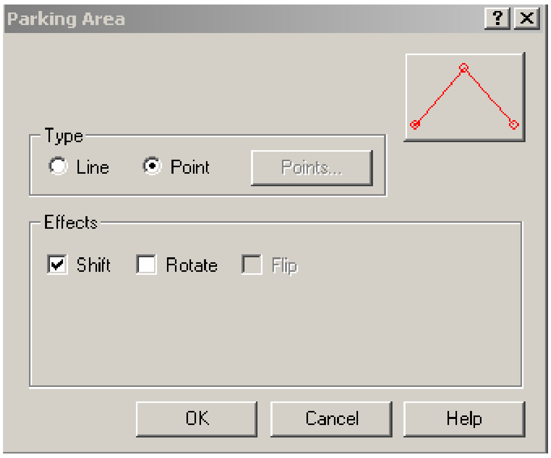
☐ Flip

Points: 7

Display 9.7. The Distance Dialog Box

As you add the 11 distances, you might also want to consider activating the *Grid* and *Snap* commands from the **View** toolbar. We suggest that you add these distances in the order in which they appear in the spreadsheet for the **Distance** data module. This will avoid confusion and reduce the possibility of missing one. If you find that your distances are not where you want them, they are easy to edit. Click on the distance line to select it. Pay close attention to the shape of the pointer during this process. When you highlight the line,' the handles (points) will appear on the line. When you move your pointer directly over a point, it will change to cross hairs; click and hold to drag the point. If the pointer is still an arrow and you click and hold, all the interior points on the line will be moved. If you accidentally do this, don't forget that you can use the *Undo* button. If you find you need additional points, or you have too many, simply double-click on the distance line to open the dialog box and change the number of points. When you add or subtract distance-line points, they're always added or subtracted at the destination-station end of the line.

If you ran the model and watched the animation now, you would see the transporter moving parts through the system, but once the transporter was freed, it would disappear from the animation until it started moving again. This is because we have not told Arena where the transporters should be when they are idle. We do this by adding parking areas to our animation. Clicking the *Parking* button ( ) from the **Animate Transfer** toolbar will open the Parking dialog box shown in *Display 9.8*. Accepting the dialog will change your pointer to cross hairs. Position your pointer near the lower left-hand corner of a Station on the animation and click. As you move your pointer, you should see a bounding outline stretching from the station to your current pointer position. If you don't have this click again until you're successful. Now position your pointer where you want the transporter to sit when it becomes idle, and click; move your pointer to a position where a second idle transporter would sit and double-click. This should exit you from the parking area activity and revert your pointer to its normal arrow. If you accidentally create too many (or too few) parking areas, you can double-click one of the parking areas you added to reopen the Parking Area dialog box. Use the *Points* button to edit the number of points or parking areas. We want two because it's possible to have as many as two carts at the same location at the same time. Repeat this action for each station in your model.


Parking Area

Type
○ Line ● Point [Points...]

Effects
☑ Shift □ Rotate □ Flip

[OK] [Cancel] [Help]

Display 9.8. The Parking Area Dialog Box

The final positioning for the Order Release and Cell 1 stations should look something like *Figure 9.2*.

You can run your model and animation at this point, but if you do, very quickly the execution will pause and the Arena Error/Warning window will appear. A warning will be displayed telling you that a distance between a pair of stations has not been specified, forcing Arena to assume a distance of 0, and it will recommend that you fix the problem. (Arena makes the rash assumption that you forgot to enter the value.) You can close this window and continue the run, but the message will reappear. The problem is most likely caused by a cart being freed at the Exit System, Cell 3, or Cell 4 locations when the cart is being requested at the Order Release location. Arena attempts to transfer the cart to Order Release and fails to find a distance; thus, we have a problem. In fact, there are more problems than just this. If a cart is freed at Cel1 1 or Cel1 2 and is requested at Order Release, it will travel backward rather than clockwise as we desire. To avoid this, we'll add distances for all possible loaded and empty transfers that might occur.

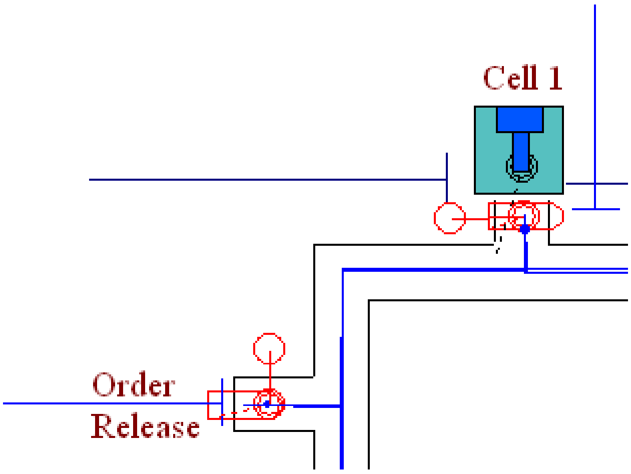


Figure 9.2. Place Distance and Parking Area Animation Objects

Table 9.2

Possible Cart Moves, Including Empty Moves

From/To	Order Release	Cell 1	Cell 2	Cell 3	Cell 4	Exit System
Order Release		37	74			
Cell 1	155		45	92	129	
Cell 2	118	139		55	147	
Cell 3	71	92	129		45	155
Cell 4	34	55	92	139		118
Exit Sytem	100	121	158	37	74	

In general, if you have *n* locations, there are *n*(*n*–1) possible distances. You can roughly think of this as an *n* by *n* matrix with 0's along the diagonal. This assumes that all moves are possible and that the distance between any two locations could depend on the direction of travel. For our example, we have six locations, or 30 possible distances (in our model, distance *does* depend on direction of travel). If the distances are not travel dependent, you only have half this number of distances. Also, there are often many moves that will never occur. In our example, the empty cart may be required to move from the Exit System location to the Order Release location, but never in the opposite direction. (Think about it!) The data contained in *Table 9.1* were for *loaded* cart moves only. If we enumerate all possible *empty* cart moves and eliminate any duplicates from the first set, we have the additional distances given in *Table 9.2*. The shaded entries are from the earlier distances in *Table 9.1*. Altogether, there are 25 possible moves.

If the number of distances becomes excessive, we suggest that you consider switching to guided transporters, which use a network rather than individual pairs of distances. For a review of the concepts of guided transporters, we refer you to Chapter 9, "Advanced Manufacturing Features," of Pegden, Shannon, and Sadowski (1995).

The final animation will look very similar to our first animation, but when you run the model, you'll see the carts moving around and the parts moving with the carts. It's still possible for one cart to appear on top of another since we're using free-path transporters. However, with only two carts in the system, it should occur much less frequently than in the previous model that used unconstrained routes. A view of the running animation at approximately time 643 is shown in *Figure 9.3*.

9.3.2 Model 9-3. Refining the Animation for Transporters

If you ran your model (or ours) and watched the animation closely, you might have noticed that the parts had a tendency to disappear while they were waiting to be picked up by a cart. This was only temporary as they suddenly reappeared on the cart just before it pulled away from the pickup point. If you noticed this, you might have questioned the correctness of the model. It turns out that the model is accurate, and this disappearing act is due only to a flaw in the animation. When a new part arrives, or a part completes its processing at a cell, it enters a request queue to wait its turn to request a cart. Once a cart is allocated, the entity is no longer in the request queue during the cart's travel to the entity location. (We don't need to go into detail about where the entity really is at this time.) The simplest way to animate this activity is with the Storage feature. An Arena storage is a place for the entity to reside while it's waiting for the transporter to move to its location so that it will show on the animation. Each time an entity enters a storage, a storage variable is incremented by 1, and when the entity exits the storage, this variable is decremented by 1. This also allows us to obtain statistics on the number in the storage.

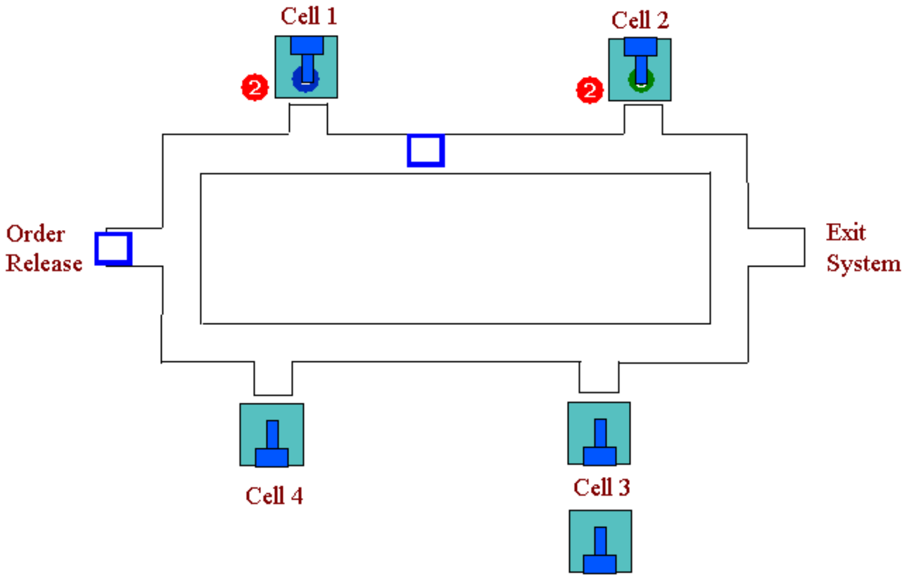


Figure 9.3. The Small Manufacturing System with Transporters

Unfortunately, the storage option is not available with the modules found in the Advanced Transfer panel. However, it is available if we use modules from the Blocks panel (the spreadsheet view is not available with the modules from the Blocks or Elements panels). We'll briefly show you how to make the necessary changes to our model. Let's start with Model 9-2 and save it as Model 9-3. The first thing you should do is delete all five **Leave** modules from the model and replace them with the **Queue - Request - Delay - Tansport** module combination shown in *Figure 9.4*. (The modules shown in *Figure 9.4* replace the Start Sequence **Leave** module.) All four of these modules were taken from the Blocks panel.

Next you'll need to select the **Queue** data module (Basic Process panel) and add the five queues where parts wait for a cart: Start Sequence.Queue, Route from Cell 1.Queue, Route from Cell 2.Queue, Route from Cell 3. Queue, and Route from Cell 4.Queue. At this point, you might be thinking, "We already added these queues!" You're correct, but they were added in the **Leave** modules that we just deleted. Once you've completed that task, you'll need to add ten storages to the **Storage** data module (Advanced Process panel), as shown in *Display 9.9*.

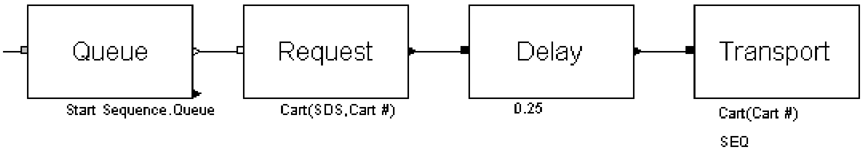
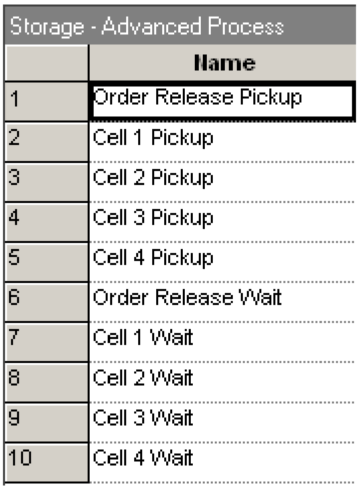


Figure 9.4. The **Queue-Request-Delay-Transport** modules Block Panel



Storage - Advanced Process

	Name
1	Order Release Pickup
2	Cell 1 Pickup
3	Cell 2 Pickup
4	Cell 3 Pickup
5	Cell 4 Pickup
6	Order Release Wait
7	Cell 1 Wait
8	Cell 2 Wait
9	Cell 3 Wait
10	Cell 4 Wait

Display 9.9. The **Storage** data module

As was the case with the missing queues, our attribute Cart # had also been defined in the deleted **Leave** modules. So let's, start by adding this attribute to our Assign Part Type and Sequence **Assign** module. The additional assignment is shown in *Display 9.10*. We've used this **Assign** module only to define the attribute so the value assignment has no meaning. We could have also used the **Attributes** module from the Elements panel to add this attribute.

Type	Attribute
Name	Cart #
Value	0

Display 9.10. Defining the Cart # Attribute

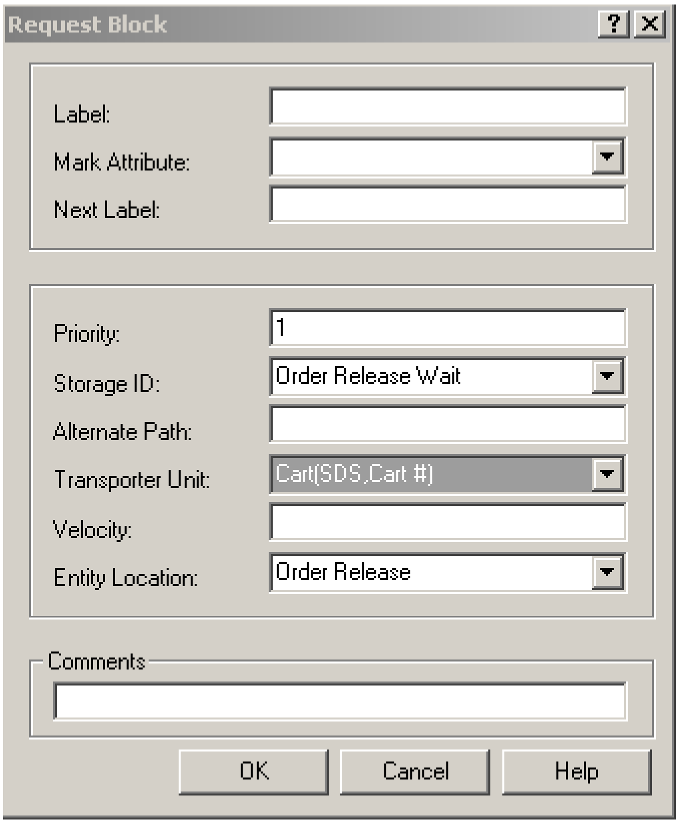
Finally, we can edit our four-module combination shown in *Figure 9.4*. We'll show you the data entries for the arriving parts and let you poke through the model to figure out the entries for the four cells. Let's start with the **Queue** data module where we only need to enter the queue name, shown in *Display 9.11*. This is the queue in which parts will wait until they have been allocated a cart. We could have omitted this queue and Arena would have used an internal queue to hold the waiting entities. This would have resulted in an accurate model, but we would not have been able to animate the queue or review statistics on parts waiting for transfer.

Queue ID	Start Sequence.Queue

Display 9.11. Defining *Start Sequence.Queue*: The **Queue** data module

The **Queue** Block module is followed by a **Request** Block module. This module allocates an idle Cart and moves it to the part location. It also allows us to specify a storage location, which we can use to animate the part during the time the cart is moved to its location. The data for the Storage ID and Entity Location entries can be selected from the list in *Display 9.12*. The Transporter Unit entry requires further explanation. First, you might have noticed that the box is shaded. This indicates that this is a required field for this module. There are three different forms that can be used to specify the transporter. The first form is *Transporter*

ID(Number), where *Transporter ID* is the transporter name; in this case, Cart. The *Number* is the unit number of the transporter being requested. In our example, there are two carts, so the unit number would be 1 or 2. Thus, we are, in effect, requesting a specific cart if we use this form. You should also be aware that if you omit the *Number,* Arena defaults it to 1. The second form is *Transporter ID(TSR),* where *TSR* is the transporter selection rule. For our example, this would be the shortest distance rule, SDS (which stands for Smallest Distance to Station). When using the **Request** Block module, you must use the Arena designation for these rules. The last form is *Transporter ID (TSR, AttributeID),* where *AttributeID* is the name of the attribute where Arena should store the unit number of the transporter selected. In our example, this would be the attribute Cart #. Thus, our entry is Cart (SDS , Cart #).



Label:
Mark Attribute:
Next Label:

Priority: 1
Storage ID: Order Release Wait
Alternate Path:
Transporter Unit: Cart(SDS,Cart #)
Velocity:
Entity Location: Order Release

Comments

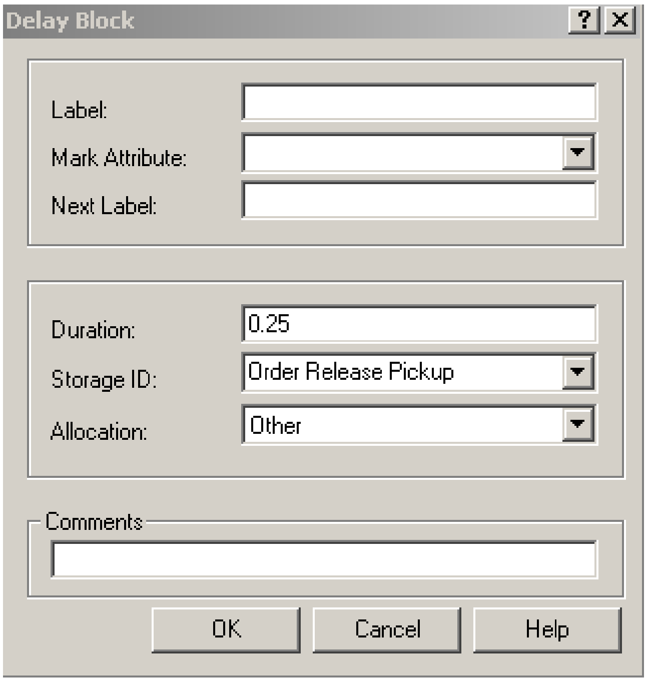
Storage ID Transporter Unit Entity Location	Order Release Wait Cart (SDS, Cart #) Order Release

Display 9.12. The **Request** Block module

When the Cart arrives at the pickup station, the part will exit the **Request** Block module and enter the **Delay** Block module, as illustrated in *Display 9.13*. We have entered the load delay, 0.25, and a second Storage ID, Order Release Pickup. We'll discuss why we added this second storage when we modify the animation.

Once the part has been loaded onto the Cart, it is sent to the following **Transport** Block module, seen in *Display 9.14*. There we entered the cart to be used, Cart (Cart #) , and the destination, SEQ (the shortened Arena name for Sequence). In this example, we've intentionally kept track of the Cart that was allocated so we could make sure that the correct cart was freed upon arrival at the destination station.

The remaining modules were filled with the same entries as these, except for the Entity Location and the Storage ID values (which are specific to the location).



Delay Block

Label:

Mark Attribute:

Next Label:

Duration: 0.25

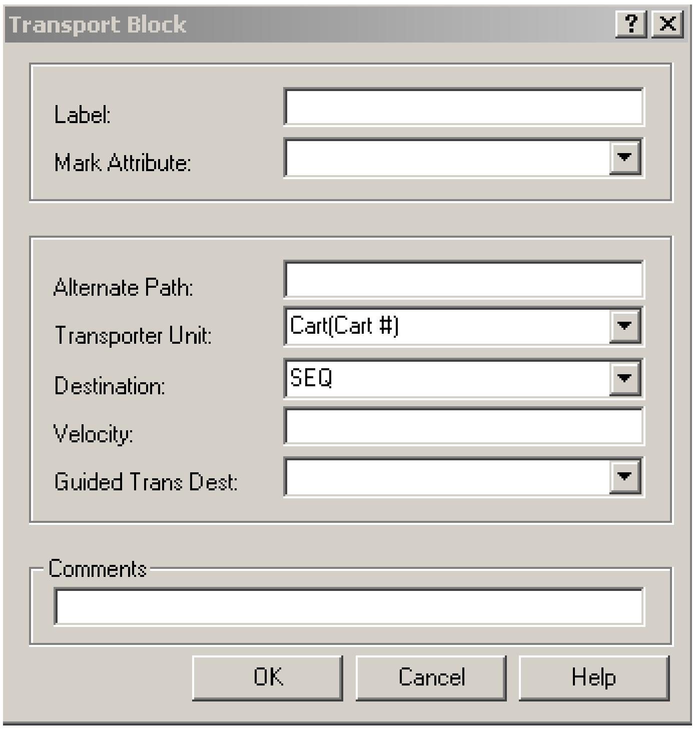
Storage ID: Order Release Pickup

Allocation: Other

Comments

Duration Storage ID	0.2 5 Order Release Pickup

Display 9.13. The **Delay** Block module



Transport Block

Label:

Mark Attribute:

Alternate Path:

Transporter Unit: Cart(Cart #)

Destination: SEQ

Velocity:

Guided Trans Dest:

Comments

Transporter Unit Destination	Cart (Cart #) SEQ

Display 9.14. The **Transport** Block module

Now let's modify our animation. You should first check to see if your wait queues have disappeared (ours did). The reason they disappeared is that they were part of the **Leave** modules that we replaced. We had simply moved the queues that came with our **Leave** modules to the proper place on the animation. Had you used the *Queue* button **()** found on the **Animate** toolbar to add your queues, they would still be there. If not, go ahead and use the *Queue* button to add the five wait queues.

Now we need to add our ten storages. We'll use the *Storage* button () from the **Animate Transfer** toolbar to make these additions. We placed the wait queues so there was room between the pickup point and the queue for us to place the pickup storages. We need two positions for each pickup, because there might be as many as two carts being moved to a location at the same time. We then placed the pickup storage directly over the parking position where the cart will sit during its loading. We used only one storage for the pickup position, even though it's possible to have two parts being picked up at the same station at the same time. If this were the case, the second cart and part would sit on top of the first one. If you want to see exactly how we did this, we suggest you take a look at our model, Model 9-3. If you run your (or our) simulation, you'll see the part initially in the wait queue. As soon as a cart becomes available and is allocated to the part, the part picture will move to the pickup storage and remain there until the cart arrives at the station. At that time, the part will appear on the transporter and remain there until the load delay is complete. The part will then move off with the cart.

There is a simpler way to create an animation where the parts don't disappear during the move and delay activity. This method would use a **Store - Request - Delay - Unstore - Transport** module combination in place of the **Leave** modules. The **Request** and **Transport** modules are from the Advanced Transfer panel, while the **Store** and **Unstore** modules are found on the Advanced Process panel. These last two modules allow you to increment and decrement storages. If you used this method, you would not animate the wait queue. All parts would be placed into a single storage while they waited for a cart to be allocated, for the cart to be moved to the station, and for the delay for the load time. We've chosen not to provide the details, but feel free to try it on your own.

Before you run your model, you should use the *Run > Setup* command and request that the transporter statistics be collected (the Project Parameters tab). If you watch the animation, you'd rarely see a queue of parts waiting for the carts, so you would not expect any major differences between this run and the original run (Model 9-1). The carts are in use only about 60% of the time. Again, we caution you against assuming there are no differences between these two systems. The run time is relatively short, and we may still be in a transient startup state (if we're interested in steady-state performance), not to mention the effects of statistical variation that can cloud the comparison.

If we change the cart velocity to 25 (half the original speed) and run the model, we'll see that there is not a huge difference in the results between the runs. If you think about this, it actually makes sense. The average distance traveled per cart movement is less than 100 feet, and when the carts become busier, they're less likely to travel empty (sometimes called *deadheading).* Remember that if a cart is freed and there are several requests, the cart will take the closest part. So the average part movement time is only about 2 minutes. Also, there was a load and unload time of 0.25 minute, which remains unchanged (remember the slide rule?). We might suggest that you experiment with this model by changing the cart velocity, the load and unload times, the transporter selection rule, and the number of carts to see how the system performs. Of course, you ought to make an appropriate number of replications and perform the correct statistical comparisons, as described in Chapter 6.

9.4. Conveyors

Having incorporated carts into our system for material movement, let's now replace the carts with a conveyor system. We'll keep the system fairly simple and concentrate on the modeling techniques. Let's assume we want a loop conveyor that will follow the main path of the aisle in the same clockwise direction we required for the transporters. There will be conveyor entrance and exit points for parts at each of the six locations in the system. The conveyor will move at a speed of 20 feet per minute. The travel distances, in feet, are given in *Figure 9.5*. We'll also assume that there is still a requirement for the 0.25-minute load and unload activity. Let's further assume that each part is 4 feet per side, and we want 6 feet of conveyor space during transit to provide clearance on the corners and to avoid any possible damage.

Arena conveyors operate on the concept that each entity to be conveyed must wait until sufficient space on the conveyor is available at its location before it can gain access to the conveyor. An Arena conveyor consists of cells of equal length that are constantly moving. When an entity tries to get on the conveyor, it must wait until a defined number of unoccupied consecutive cells are available at that point. Again, it may help to think in terms of a narrow escalator in an airport, with each step corresponding to a cell and different people requiring different numbers of steps. For example, a traveler with several bags may require two or three steps, whereas a person with no bags may require only one step. When you look at an escalator, the cell size is rather obvious. However, if you consider a belt or roller conveyor, the cell size is not at all obvious.

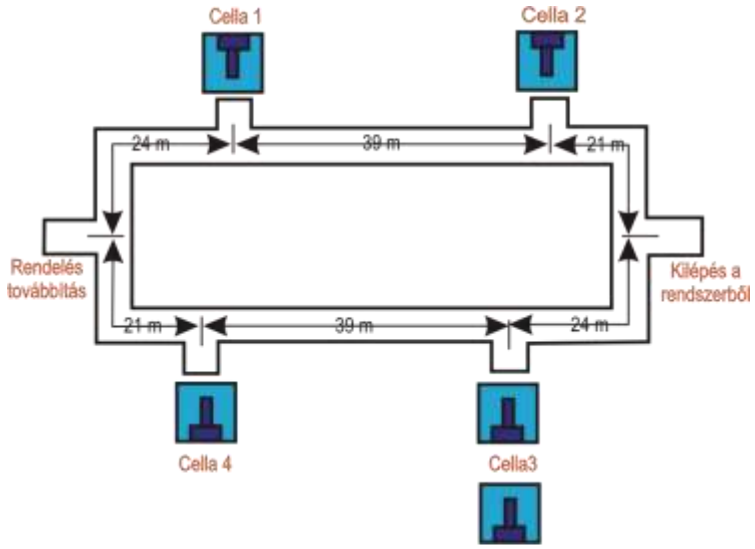


Figure 9.5. Conveyor Lengths

In order to make the conveyor features as flexible as possible, Arena allows you to define the cell size; that is, to divide the conveyor length into a series of consecutive, equal-sized cells. Each cell can hold no more than one entity. This creates a rather interesting dilemma because you would like the cells to be as small as possible to obtain the greatest modeling accuracy, yet you would also like the cells to be as large as possible to obtain the greatest computational efficiency. Let's use a simple example to illustrate this dilemma. You have a conveyor that is 100 feet long and you want to convey parts that are 2 feet long. Because we've expressed our lengths in feet, your first response might be to set your cell size to 1 foot. This means that your conveyor has 100 cells and each part requires two cells for conveyance. We could have just as easily set the cell size at 2 feet (50 cells), or 1 inch (1200 cells). With today's computers, why should we worry about whether we have 50 or 1200 cells? There should be a negligible impact on the computation speed of the model, right? However, we've seen models that have included over five miles of conveyors, and there is certainly a concern about the speed of the model. The difference between a cell size of 1 inch or 100 feet would have a significant impact on the time to run the model.

So why not always make your cells as large as possible? Well, when an entity attempts to gain access to a conveyor, it's only allowed on the conveyor when the end of the previous cell, or the start of the next cell, is lined up with the entity location. In our escalator analogy, this corresponds to waiting for the next step to appear at the load area. Consider the situation where an entity has just arrived at an empty conveyor and tries to get on. You have specified a cell size of 100 feet, and the end of the last cell has just passed that location, say by 1/2 inch. That entity would have to wait for the end of the current cell to arrive, in 99 feet 1 1 1/2 inches. If you had specified the cell size at 1 inch, the wait would only have been for 1/2 inch of conveyor space to pass by.

The basic message is that you need to consider the impact of cell size with respect to the specific application you're modeling. If the conveyor is not highly utilized, or the potential slight delay in timing has no impact on the system's performance, use a large cell size. If any of these are critical to the system performance, use a small cell size. There are two constraints to consider when making this decision. The entity size, expressed in number of cells, must be an integer so you cannot have an entity requiring 1.5 cells; you would have to use one or two cells as the entity size. Also, the conveyor segments (the length of a conveyor section from one location to another) must consist of an integral number of cells. So, if your conveyor were 100 feet long, you could not use a cell size of three.

Before we start adding conveyors to our model, let's go over a few more concepts that we hope will be helpful when you try to build your own models with conveyors. As with Resources and Transporters, Conveyors have several key words: *Access*, *Convey*, and *Exit*. To transfer an entity with Arena conveyors, you must first Access space on the conveyor, then *Convey* the entity to its destination, and finally *Exit* the conveyor (to free up the space). For representing the physical layout, an Arena conveyor consists of a series of segments that are linked together to form the entire conveyor. Each segment starts and ends at an Arena station. You can only link these segments to form a line or loop conveyor. Thus, a single conveyor cannot have a diverge point that results in a conveyor splitting into two or more downstream lines, or a converge point where two or more upstream lines join. However, you can define multiple conveyors for these types of systems. For example, in a diverging system, you would convey the entity to the diverge point, Access space on the appropriate next conveyor, Exit the current conveyor, and Convey the entity to its next destination. For our small manufacturing system, we'll use a single loop conveyor.

Arena has two types of Conveyors: *Non-accumulating* and *Accumulating*. Both types of conveyors travel in only a single direction, and you can't reverse them. These conveyors function very much as their name would imply. Examples of a non-accumulating conveyor would be a bucket or belt conveyor or our escalator. The spacing between entities traveling on these types of conveyors never changes, unless one entity exits and re-accesses space. Because of this constraint, non-accumulating conveyors operate in a unique manner. When an entity accesses space on this type of conveyor, the entire conveyor actually stops moving or is disengaged. When the entity is conveyed, the conveyor is re-engaged while the entity travels to its destination. The conveyor is stopped when the entity reaches its destination, the entity exits, and the conveyor is then restarted. If there is no elapsed time between the Access and Convey, or when the entity reaches its destination and then exits, then it's as if the conveyor was never stopped. However, if there is a delay, such as for a load or unload activity of positive duration (which is the case in our system), you'll see the conveyor temporarily stop while the load or unload occurs.

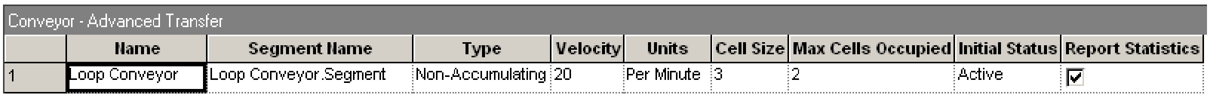
Accumulating conveyors differ in that they never stop moving. If an entity is stopped on an accumulating conveyor, all other entities on that conveyor will continue on their way. However, the stopped entity blocks any other entities from arriving at that location so that the arriving entities accumulate behind the blocking entity. When the blocking

entity exits the conveyor or conveys on its way, the accumulated entities are also free to move on to their destinations. However, depending on the spacing requirements specified in the model, they may not all start moving at the same time. You might think of cars accumulated or backed up on a freeway. When the cars are stopped, they tend to be fairly close together (mostly to prevent some inconsiderate driver from sneaking in ahead of them). When the blockage is removed, the cars start moving one at a time in order to allow for more space between them. We'll describe these data requirements later in this chapter.

9.4.1 Model 9-4. The Small Manufacturing System with Non-accumulating Conveyors

We're now ready to incorporate non-accumulating conveyors into our system. If you re building this model along with us (after all, that was the idea), you might want to consider making another change. After including conveyors and running our model, we're likely to find that the load and unload times are so small relative to the other times that it's hard to see if the conveyor is working properly. (Actually, we've already run the model and know that this will happen.) Also, you may want to do some experimentation to see what impact these activities have on system performance. So we suggest that you define two new Variables, Load Time and Unload Time, and set the initial values for both to 0.25 minute. Then replace the current constant times with the proper variable name; this will allow you to make global changes for these times in one place. You should know how to do this by now (if not, reread Section 7.2.3), so we'll not bore you with the details.

Let's start by taking our model, Model 9-1, and again deleting the route paths. Define the conveyor using the **Conveyor** data module from the Advanced Transfer panel as in *Display 9.15*.



Conveyor - Advanced Transfer

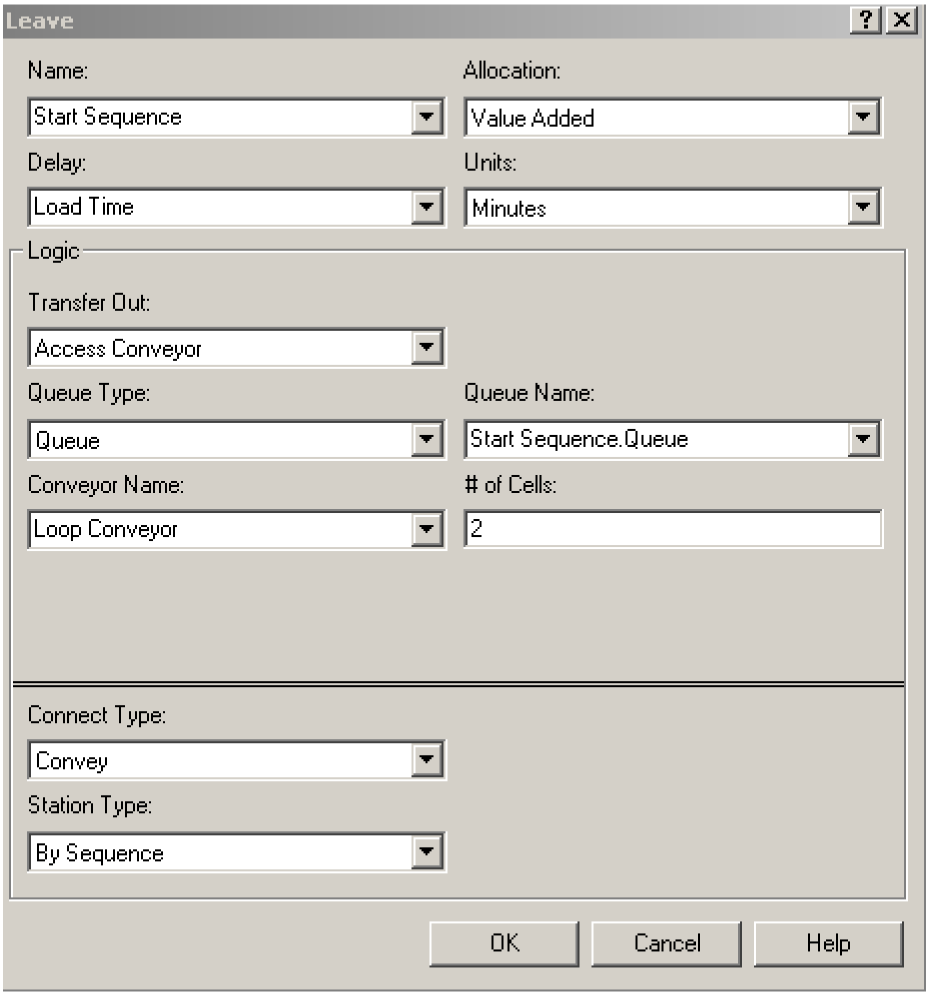
	Name	Segment Name	Type	Velocity	Units	Cell Size	Max Cells Occupied	Initial Status	Report Statistics
1	Loop Conveyor	Loop Conveyor.Segment	Non-Accumulating	20	Per Minute	3	2	Active	☑

Name	Loop Conveyor
Segment Name	Loop Conveyor.Segment
Type	Nonaccumulating
Velocity	20
Cell Size	3
Max Cells Occupied	2

Display 9.15. The **Conveyor** module

Two of these entries, Cell Size and Max Cells Occupied, require some discussion. Based on the results from the previous model, it's fairly clear that there is not a lot of traffic in our system. Also, if there's a slight delay before a part can Access space on the conveyor, the only impact would be to increase the part cycle time slightly. So we decided to make the conveyor cells as large as possible. We chose a cell size of 3 feet, because this will result in an integer number of cells for each of the conveyor lengths (actually, we cheated a little bit on these lengths to make this work). Since we require 6 feet of space for each part, there is a maximum of two cells occupied by any part. This information is required by Arena to assure that it has sufficient storage space to keep track of an entity when it arrives at the end of the conveyor. Since our conveyor is a loop and has no end, it really has no impact for this model. But, in general, you need to enter the largest number of cells that any entity can occupy during transit.

Incorporating conveyors into the model is very similar to including transporters, so we won't go into a lot of detail. We need to change every **Leave** and **Enter** module, much like we did for transporters. The entries for the *Start Sequence* **Leave** module are shown in *Display 9.16*. We've included only the changes that were required starting with Model 9-1. You might note that each part requires two cells (with a size of 3 feet each) to access the conveyor, and that we've used the variable Load Time that we suggested earlier.


Leave

Name: Start Sequence

Allocation: Value Added

Delay: Load Time

Units: Minutes

Logic

Transfer Out: Access Conveyor

Queue Type: Queue

Queue Name: Start Sequence.Queue

Conveyor Name: Loop Conveyor

# of Cells: 2

Connect Type: Convey

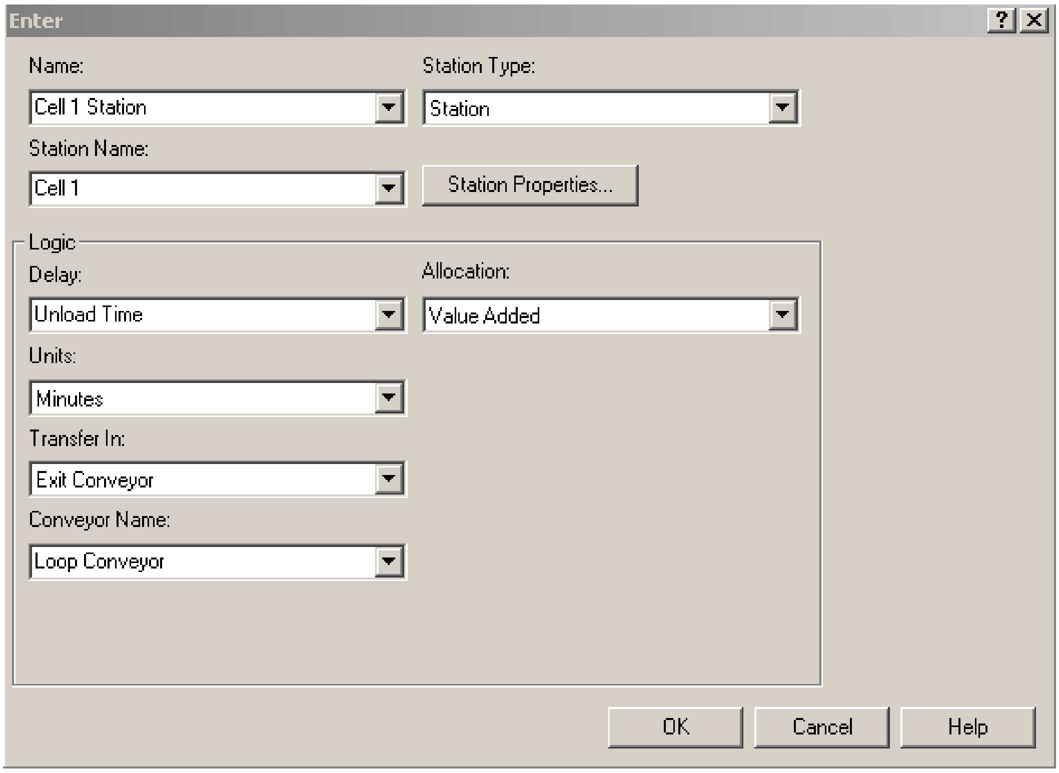
Station Type: By Sequence

Name Delay Units	Start Sequence Load Time Minutes
Logic Transfer Out Conveyor Name # of Cells	 Access Conveyor Loop Conveyor 2
Connect Type	Convey

Display 9.16. The **Leave** module for Conveyors

The entries for the *Cell 1 Station* **Enter** module are shown in *Display 9.17*. We're now ready to place the conveyor segments on our animation. As was the case with the Distances used in our last model, we'll first need to define the segment data required for both the model and animation. We define these segments using the **Segment** data module found on the Advanced

Transfer panel. *Display 9.18* shows the first entry for the main spreadsheet view of the **Segment** data module. We arbitrarily started our Loop Conveyor at the Order Release station.



Name: Cell 1 Station

Station Type: Station

Station Name: Cell 1

Logic

Delay: Unload Time

Allocation: Value Added

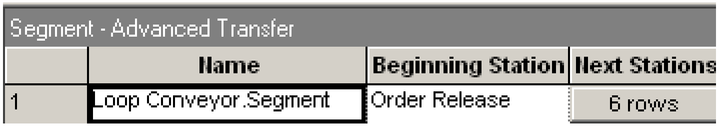
Units: Minutes

Transfer In: Exit Conveyor

Conveyor Name: Loop Conveyor

Name Station Type Station Name	Cell 1 Station Station Cell 1
Logic Delay Units Transfer In Conveyor Name	 Unload Time Minutes Exit Conveyor Loop Conveyor

Display 9.17. The **Enter** module for Conveyors

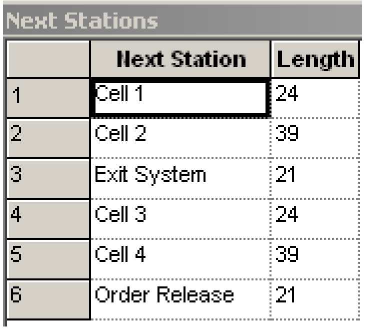


Segment - Advanced Transfer

	Name	Beginning Station	Next Stations
1	Loop Conveyor.Segment	Order Release	6 rows

Name Beginning Station	Loop Conveyor Segment Order Release

Display 9.18. The **Segment** data module



Next Stations

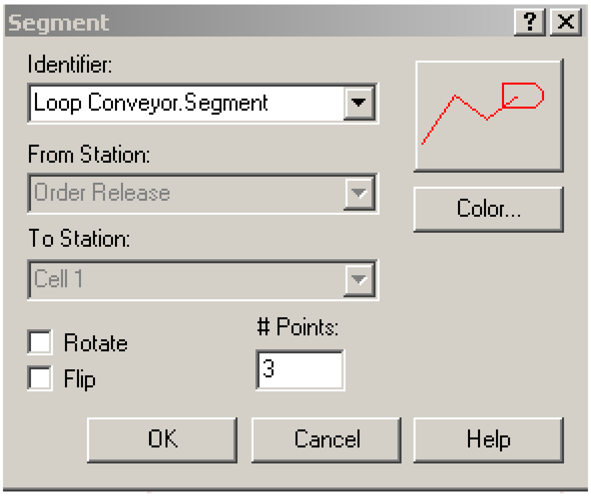
	Next Station	Length
1	Cell 1	24
2	Cell 2	39
3	Exit System	21
4	Cell 3	24
5	Cell 4	39
6	Order Release	21

Display 9.19. The **Segment** data module

Display 9.19 shows the spreadsheet data entries for the six segments. Notice that the Length is expressed in terms of the actual length, not the number of cells, for that segment. These lengths are taken directly from the values provided in *Figure 9.5*.

We're now ready to add our segments to the animation. You'll first need to move each station symbol to the center of the main loop directly in front of where the part will enter or exit the conveyor. To add the segments, we use the *Segment* button ( ) found on the **Animate Transfer** toolbar. We add segments much like we added distances, except in this model, we need to add only six segments. The Segment dialog box is shown in *Display 9.20*, where we've simply accepted the defaults.

For this model, you need to place only six segments, as shown in *Figure 9.6*. As you're placing your segments, you may find it necessary to reposition your station symbols in order to get the segments to stay in the center of the loop (we did!).



Identifier: Loop Conveyor.Segment

From Station: Order Release

To Station: Cell 1

☐ Rotate

☐ Flip

Points: 3

Display 9.20. The Segment Dialog Box

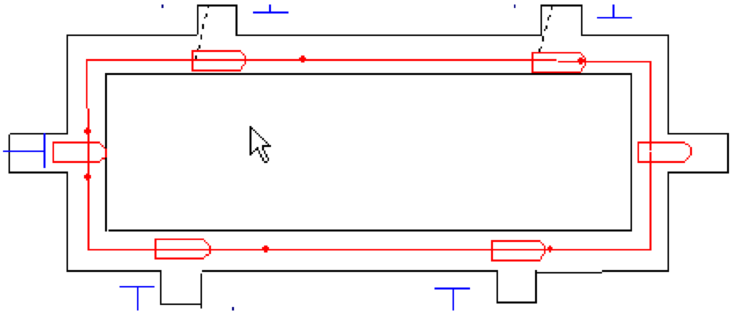


Figure 9.6. The Conveyor Segments

You're now almost ready to run your animated model. You should first use the Run > Setup command and request that the conveyor statistics be collected (the Project Parameters tab). If you run the model to completion, you'll notice that two new statistics for the conveyor are included in the report. The first statistic tells us that the conveyor was blocked, or stopped, about 17.65% of the time. Although this may initially appear to be a large value, you must realize that each time a part leaves or enters a station, it has a load or unload delay. During this time, the conveyor is stopped or blocked. If you consider the number of moves and the number of parts that have gone through the system, this value seems plausible. If you're still not convinced, the numbers are all there for you to approximate what this value should be.

The second statistic tells us that the conveyor was utilized only about 6.6% of the time. This is not the amount of time that a part was on the conveyor, but the average amount of space occupied by parts on the conveyor. This one is easy to check out. You can determine the total length (168 feet) of the conveyor using the lengths provided in *Figure 9.5*. Knowing that each cell is 3 feet, we can determine the number of cells (56). Each part requires two cells, meaning that the conveyor can only hold a maximum of 28 parts at any point in time. Thus, if we multiply the 28-part conveyor capacity by the average utilization (0.066), we know that on the average there were about 1.85 parts on the conveyor. If you watch the animation, this would appear to be correct.

If you compare the system time for the parts to those from Model 9-1, you'll find that they are slightly higher here. Given the conveyor blocking and speed, this also appears to be reasonable.

If you can't see the conveyor stop during the load and unload activity (even if you set the animation speed factor at its lowest setting), change these variables to 2 or 3 minutes. The stopping should be quite obvious with these new, higher values.

There are also several alternate ways to change variable values during a simulation. If you want to make such changes frequently during a simulation run, you might want to consider using the Arena interface for Visual Basic® for Applications (VBA) (see Chapter 10). The second method is to use the Arena command-driven Run Controller. Begin running your model, and after it has run for a while (we waited until about time 445), use the *Run*>*Pause* command or the *Pause* button (⏸) on the **Standard** toolbar to suspend the execution of the model temporarily. Then use *Run*>*Run Control*>*Command* or the *Command* button ( ) on the **Run Interaction** toolbar to open the command window. This text window will have the current simulation time displayed, followed by ">" as a prompt. Arena is ready for you to enter your commands. We used the SHOW command to view the current value of the variable Load Time and then the ASSIGN command to change that value-see *Figure 9.7*. We then repeated these two steps for the Unload Time. Now close this window and use the *Run* > *Go* menu option or the *Go* button (▶) on the **Standard** toolbar to run the simulation with the new variable values from the current time.

If you watch the animation, in a very short period of time, you'll see that the conveyor movement is quite j erky, and it quickly fills up with over ten parts. You can stop the simulation run at any time to change these values. It's worth noting that the changes you make in the Run Controller are temporary. They aren't saved in the model, and the next time you run your model, it will use the original values for these variables.

```
SIMAN Run Controller.
 449.03333>SHOW Load Time
LOAD TIME = 0.25
449.03333>ASSIGN Load Time = 2
449.03333>SHOW Unload Time
UNLOAD TIME = 0.25
449.03333>ASSIGN Unload Time = 2
449.03333>
```

Figure 9.7. Changing Variables with the Run Controller

Although viewing the animation and making these types of changes as the model is running are excellent ways to verify your model, you should not change the model conditions during a run when you finally are using the model for the purposes of evaluation. This can give you very misleading performance values on your results and will be essentially impossible to replicate.

9.4.2 Model 9-5. The Small Manufacturing System with Accumulating Conveyors

Changing our conveyor model so the conveyor is accumulating is very easy. We started by using the File > Save As command to save a copy of Model 9-4 as Model 9-5. We need to make only two minor changes in the **Conveyor** data module. Change the conveyor Type to Accumulating, and enter an Accumulation Length of 4, as in *Display 9.21*. Adding the Accumulation Length allows the accumulated parts to require only 4 feet of space on the conveyor. Note that this value, which applies only when an entity is stopped on the conveyor, does not need to be an integer number of cells. When the blockage is removed, the parts will automatically re-space to the 6 feet, or two cells, required for transit on the conveyor.

Type	Accumulating
Accumulation Length	4

Display 9.21. The Accumulating **Conveyor** data module Dialog Box

Having made these changes, run your model and you should note that very little accumulation occurs. You can confirm this by looking at the conveyor statistics on the summary report. The average accumulation is less than 1 foot and the average utilization is less than 6%. To increase the amount of accumulation, which is a good idea for verification, simply change the load and unload times to 4 or 5 minutes. The effect should be quite visible.

You can see that these results are very similar to those from the non-accumulating system except for the part system times, which are much higher. We strongly suspect that this is due to the short run time for the model. We'll leave it to the interested reader to confirm (or refute) this suspicion.

9.5 Chapter summary

Entity movement is an important part of most simulation models. This chapter has illustrated several of Arena's facilities for modeling such movement under different circumstances to allow for valid modeling.

This is our last general "tutorial" chapter on modeling with Arena. It has gone into some depth on the detailed lower-level modeling capabilities, as well as correspondingly detailed topics like debugging and fine-tuned animation. While we've mentioned how you can access and blend in the SIMAN simulation language, we've by no means covered it; see Pegden, Shannon, and Sadowski (1995) for the complete treatment of SIMAN. At this point, you should be armed with a formidable arsenal of modeling tools to allow you to attack many systems, choosing constructs from various levels as appropriate.

There are, however, more modeling constructs and, frankly, "tricks" that you might find handy. These we take up next in Chapter 9.

CHAPTER 10

A Sampler of Further Modeling Issues and Techniques

In Chapters 4-9, we gave you a reasonably comprehensive tour of how to model different kinds of systems by means of a sequence of progressively more complicated examples. We chose these examples with several goals in mind, including reality and importance (in our experience) of the application, illustration of various modeling issues, and indication of how you can get Arena to represent things the way you want-in many cases, fairly easily and quickly. Armed with these skills, you'll be able to build a rich variety of valid and effective simulation models.

But no reasonable set of digestible examples could possibly fathom all the nooks and crannies of the kinds of modeling issues (and, yes, tricks) that people sometimes need to consider, much less all of the features of Arena. And don't worry, we're not going to attempt that in this chapter either. But we would like to point out some of what we consider to be the more important additional modeling issues and techniques (and tricks) and tell you how to get Arena to perform them for you.

We'll do this by constructing more examples, but these will be more focused toward specific modeling techniques and Arena features, so will be smaller in scope. In Section 10.1, we'll refine the conveyor models we developed in Chapter 9; in Section 10.2, we'll discuss a few more modeling refinements to the transporters from Chapter 9. In service systems, especially those involving humans standing around in line, there is often consideration given to customer reneging (in other words, jumping out of line at some point); this is taken up in Section 10.3, along with the concept of balking. Section 10.4 goes into methods (beyond the queues you've already seen) for holding entities at some point, as well as batching them together with the possibility of taking the batch apart later. In Section 10.5, we'll discuss how to represent a tightly coupled system in which entities have to be allocated resources downstream from their present position before they can move on; this is called overlapping resources from the entity viewpoint. Finally, Section 10.6 briefly mentions a few other specific topics, including guided transporters, parallel queues, and the possibility of complex decision logic and looping.

This chapter is structured differently from the earlier ones in that the sections are not necessarily meant to be read through in sequence. Rather, it is intended to provide a sampler of modeling techniques and Arena features that we've found useful in a variety of applied projects.

10.1 Modeling Conveyors Using the Advanced Transfer Panel

In this section, we indicate some refinements to the basic conveyor models described in Chapter 9.

10.1.1 Model 10-1. Finite Buffers at Stations

In Chapter 8, we introduced you to Arena conveyors. In Section 9.4.1, we developed a model, Model 8-4, for our small manufacturing system using non-accumulating conveyors' as the method for transferring parts within our system. In developing that model, we assumed that there was an infinite buffer in front of each cell for the storage of parts waiting to be processed. This assumption allowed us to use the conveyor capabilities found in the Enter and **Leave** modules. We did, however, need to add the **Conveyor** and **Segment** data modules from the Advanced Transfer panel to define our conveyor.

Now let's modify that assumption and assume that there is limited space at Cells 1 and 2 for the storage of unprocessed parts. In fact, let's assume that there is only room for one unprocessed part at each cell. For this type of model, we need to define what happens to a part that arrives at Cell 1 or 2 and finds that there is already a part occupying the limited buffer space. Assuming that we could determine a way to limit the buffer using the **Enter** module (it can't be done), it would be tempting simply to let the arriving part wait until the part already in the buffer is moved to the machine in that cell. Of course, this could cause a significant logjam at these cells. Not only would parts not be able to enter the cell, but parts on their way to other cells would queue up behind them creating yet another problem. It turns out that processed parts trying to leave the cell need to access space on the conveyor before they can be conveyed to their next destination station. Yes, the space they're trying to access is the same space occupied by the part waiting to enter the cell. This would create what's sometimes called a deadlock, or gridlock.

So let's use the following strategy for parts arriving at Cell 1 or 2. If there is not a part currently waiting in the buffer for the cell, the arriving part is allowed to enter the buffet Otherwise, the arriving part is conveyed around the loop conveyor back to the same point to try a second (or third, or fourth, etc.) time. To do this, we need to alter our model so w can better control when the part exits the conveyor. The Advanced Transfer panel provides five new modules for conveyors (**Access**, **Convey**, **Exit**, **Start**, and **Stop**) that allow us to model conveyor activities at a more detailed level. The **Exit** module causes an entity to exit a conveyor, releasing the conveyor cell(s) it occupied. This is essentially what happens when you select the Exit Conveyor option in the Transfer In dialog box of the **Enter** module. The **Access** module causes an entity to request or access space on a conveyor at a specific location, normally its current station location. The **Convey** module is used to convey the entity to its destination once it has successfully accessed the required conveyor space. You are essentially requesting that Arena Access and Convey an entity when you select the Access Conveyor option in the Transfer Out dialog box of the **Leave** modules. The **Start** and **Stop** modules cause the conveyor to start and stop its movement, respectively. These two modules can be used to develop your own failure logic or generally control when the conveyor is idle or running.

To develop our new model, we will start with Model 8-4 and replace the **Enter** and **Leave** modules for Cell 1 with our new modules, as shown in *Figure 10.1*.

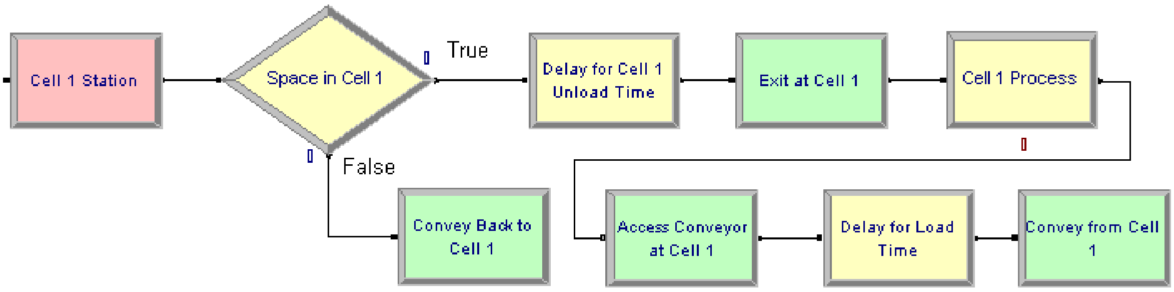


Figure 10.1. New Model Logic for Cell 1

In the **Enter** module we are replacing, we defined the station, with the name Cell 1, and also exited the conveyor. The definition of the station is critical because Arena must know where to send an entity that is being conveyed to Cell 1. Therefore, we started our replacement set of modules with a **Station** module from the Advanced Transfer panel with the specific purpose of defining the entry point to the station Cell l. We also could have used our **Enter** module for this purpose, but we wanted to show you the **Station** module. The **Station** module simply defines the logical entry for entities that are transferred to that station. In our case, the only value provided in the dialog box is the station name, shown in *Display 10.1*.

Name	Cell 1 Station
Station Name	Cell 1

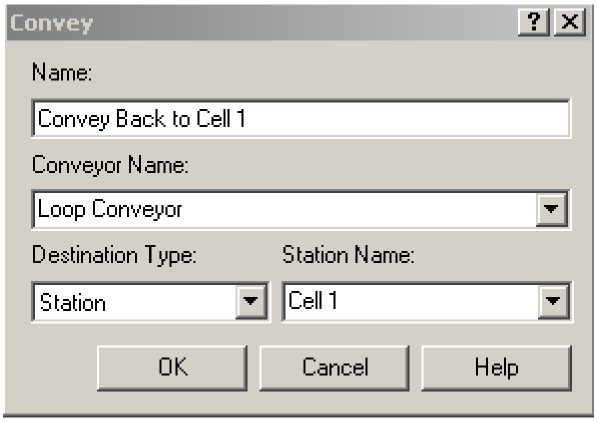
Display 10.1. The **Station** module

When an entity, or part, arrives at station *Cell 1*, it must check the status of the waiting queue before it can determine its fate. We cause this to happen by sending the entity to the **Decide** module where we check to see if the number of entities in queue *Cell 1 Process.Queue* is equal to 0, shown in *Display 10.2*. This is the same queue name used in Model 8-4, and it was defined in the **Process** module. If the queue is currently occupied by another entity, the arriving entity will take the *False* branch of our module.

Name	Space in Cell 1
Type	2-way by Condition
If	Expression
Value	NQ(Cel1 1 Process.Queue)= = 0

Display 10.2. The **Decide** module

In this case, we do not want to exit the conveyor; we instead want to leave the part on the loop conveyor and convey it around the loop and back to the same station. We do this by sending the entity to the **Convey** module, in *Display 10.3*, where we convey the entity around the loop and back to Cell 1.



Name: Convey Back to Cell 1

Conveyor Name: Loop Conveyor

Destination Type: Station

Station Name: Cell 1

Name	Convey Back to Cell 1
Conveyor Name	Loop Conveyor
Station Name	Cel1 1

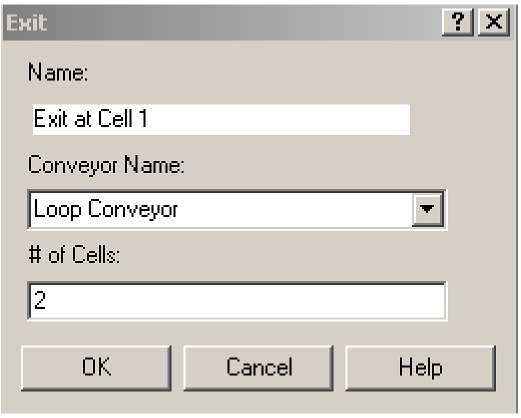
Display 10.3. The **Convey** module

At this point, you might be thinking, "But the entity is already at station Cell 1!" Arena assumes that your intention is to convey the entity and it will send the entity on its way. Of course, it also assumes that it is physically possible to convey the entity to the specified destination, which is the case. Thus, the **Convey** module will convey the entity on the designated conveyor to the specified destination. If the entity is not already on the conveyor, Arena will respond with a terminating error message.

If the queue at Cell 1 is unoccupied, the entity will satisfy the condition and be sent to the connected **Delay** module, in *Display 10.4*, where the entity is delayed in the Space in Cell 1 **Decide** module for the unload time. It is then sent to the **Exit** module, which removes the entity from the conveyor and releases the conveyor cells it occupied, shown in *Display 10.5*.

Name	Delay for Cell 1 Unload Time
Delay Time	Unload Time
Units	Minutes

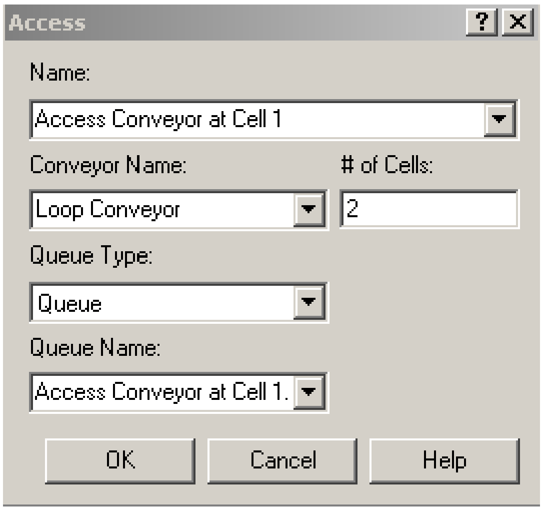
Display 10.4. The **Unload-Delay** module



Name:
Exit at Cell 1
Conveyor Name:
Loop Conveyor
# of Cells:
2

Name	Exit at Cell 1
Conveyor Name	Loop Conveyor
# of Cells	2

Display 10.5. The **Exit** module



Name:
Access Conveyor at Cell 1
Conveyor Name:
Loop Conveyor
# of Cells:
2
Queue Type:
Queue
Queue Name:
Access Conveyor at Cell 1.

Name	Access Conveyor at Cell 1
Conveyor Name	Loop Conveyor
# of Cells	2

Display 10.6. The **Access** module

The Entity is sent from the **Exit** module to the **Process** module that represents the actual machining operation, shown in *Figure 10.1*. Once the part finishes its operation, it exits the **Process** module and is sent to the following **Access** module to get space on the conveyor, as shown in *Display 10.6*.

The **Access** module allocates cells on the conveyor so the entity can then be conveyed to its next destination; it does not actually convey the entity. Thus, if you do not immediately cause the entity to be conveyed, it will cause the entire conveyor to stop until the entity is conveyed.

In the event that the required cells were not available, the entity would reside in the queue (Access Conveyor at Cell 1.Queue) until space was accessible. This provides a way for the entity to show up in the animation if it has to wait for available conveyor space.

Having accessed conveyor space, the entity delays for the Load Time, then is sent to the **Convey** module where it is conveyed according to its specified sequence. This completes the replacement modules for Cell 1 in Model 8-4.

You also need to perform the same set of operations for Cell 2. You can simply repeat these steps or make a copy of these modules and edit them individually. We chose to replace only the **Enter** module, retaining the **Leave** module, for Cell 2, as shown in *Figure 10.2*. We'll assume that by now you're comfortable making the required changes.

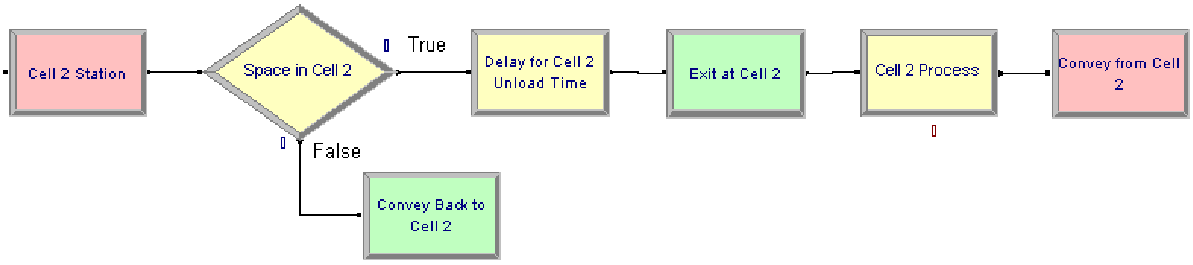


Figure 10.2. New Model Logic for Cell 2

*When you delete the **Enter** and **Leave** modules, you may find that part of your animation has been deleted too. This is because you received a "free" animated queue with **Leave** module. When you are editing or changing a model like this, it is frequently easier to animate the new features using the constructs from the **Animate** toolbar, so when developed our model, we deleted the added animation features that were provided by new modules and animated these new features ourselves. In our model, we deleted queue and the storage for Cell 1. We simply added the new access queue. When we ran our new model, we could see parts being blocked from entry to Cell 1 starting at about time 275.*

10.1.2 Model 10-2. Parts Stay on Conveyor During Processing

Now that you have conquered these new conveyor modules, let's examine another quick problem. Start with the accumulating conveyor model, Model 8-5, presented in Section 9.4.2. Assume that we're trying a new layout that requires that the operations at Cell 2 be performed with the part remaining on the conveyor. Specifically, parts conveyed to Cell 2 do not exit the conveyor, but the part stops at Cell 2, the actual operation is performed' while the part sits on the conveyor, and the part is then conveyed to its next destination. Since the conveyor is accumulating, other parts on the conveyor will continue to move unless they are blocked by the part being operated on at Cell 2.

We could implement this new twist by replacing the current **Enter** and **Leave** modules: for Cell 2 with a set of modules similar to what we did for Model 10-1. However, since the entities arriving at Cell 2 do not exit or access the conveyor, there is a far easier solution. We'll modify the Transfer In option on the **Enter** module by selecting the *None* option. In this case, the entity will not exit the conveyor. This will cause the entity to reside on the conveyor while the operation defined by the server takes place. However, if this is the only change we make, an error will occur when the entity attempts to leave Cell 2. In the Transfer Out dialog box of the **Leave** module, we had previously selected the Access Conveyor option that caused the entity to try to access space on the conveyor, and since the entity will remain on the conveyor, Arena would become confused and terminate with a runtime error. This is easily fixed by selecting the *None* option in the Transfer Out dialog box in the Logic section of the **Leave** module. These are the only model changes required.

You can test your new model logic by watching the animation. We suggest you fast-forward the simulation to about time 700 before you begin to watch the animation. At this point in the simulation, the effects of these changes become quite apparent.

10.2 More on Transporters

In Chapter 9, we presented the concepts for modeling Arena transporters and conveyors using the functionality available in the high-level modules found in the Basic Process panel. In Section 10.1, we further expanded that functionality for conveyors by using modules from the Advanced Transfer panel to modify and refine the models presented in Chapter 9. The same types of capabilities also exist for Transporters. Although we're not going to develop a complete model using these modules, we'll provide brief coverage of their functions and show the sequence of modules required to model several different situations. This should be sufficient to allow you to use these constructs successfully in your own models. Remember that on me help is always available.

Let's start with the basic capabilities covered in Section 9.3. The fundamental transporter constructs are available in the Transfer In and Transfer Out section of the **Enter** and **Leave** modules. Consider the process of requesting a transporter and the subsequent transfer of the transporter and entity to its next station or location. The **Request** and **Transport** modules found in the Advanced Transfer panel also provide this capability.

The **Request** module provides the first part of the Transfer Out section of the **Leave** module and is almost identical to it. You gain the ability to override the default Velocity, but you lose the ability to specify a Load Time. However, a Load Time can be included by then directing the entity to a **Delay** module. The **Request** module actually performs two activities: allocation of a transporter to the entity and moving the empty transporter to the location of that entity, if the transporter is not already there. The **Transport** module performs the next part of the Transfer Out activity by initiating the transfer of the transporter and entity to its next location. Now let's consider a modeling situation where it would be desirable to separate these two functions. Assume that when the transporter arrives at the entity location there is a loading operation that requires the assistance of an operator. If we want to model the operator explicitly, we'll need these new modules. The module sequence (**Request - Process - Transport**) required to model this situation is shown in *Figure 10.3*.

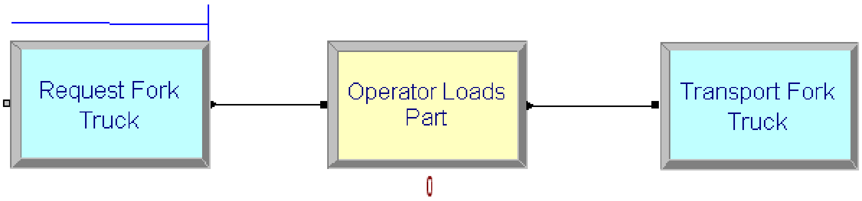


Figure 10.3. Operator-Assisted Transporter Load

You can also separate the two activities of the **Request** module by using the Allocate and **Move** modules. The **Allocate** module allocates a transporter to the entity, but leaves the transporter at its current location. The **Move** module allows the entity that has been allocated a transporter to Move the empty transporter anywhere in the model. When using the **Request** module, the transporter is automatically moved to the entity location. Consider the situation where the empty transporter must first pick up from a staging area a fixture required to transport the entity. In this case, we need to allocate the transporter, send it to the staging area, pick up the fixture, and finally send it to the entity's location. The module sequence (Allocate - Move - Process - Move) required to model these activities is shown in *Figure 10.4*.

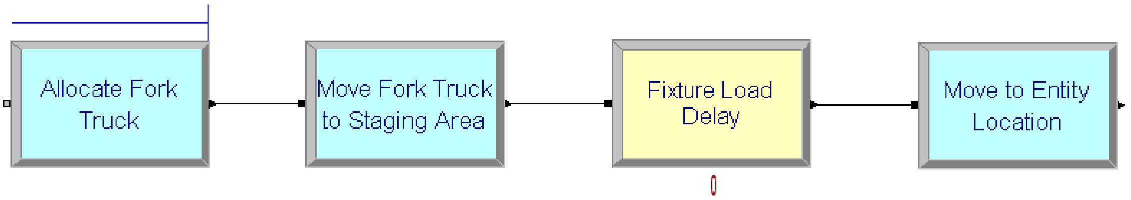


Figure 10.4. Fixture Required for Entity Transfer

Of course, you could include all kinds of embellishments for this activity. Suppose that only a portion of the entities need this fixture. We could easily include a **Decide** module between them to check for this condition, as shown in *Figure 10.5*.

Our two examples using the **Allocate** and **Move** modules both resulted in the transporter being moved to the location of the entity. Now let's assume that whenever your transporter is freed, you want it to be roving around the system looking for work. You have a predefined path that the transporter should follow. This path would be very similar to a sequence of stations, except it would form a closed loop. Each time the unallocated transporter reached the next station on its path, it would check to see if there is a current request from somewhere. If there is no request, the transporter continues on its mindless journey. If there is a request, the transporter proceeds immediately to the location of that request.

To model this, you would create a single entity (let's call it the loop entity) that would attempt to allocate the transporter with a very low priority (high number for the priority). Once allocated, the empty transporter is moved to the next station on its path. Upon arrival at this station, the transporter is freed so it can respond to any current request. The loop entity is directed to the initial **Allocate** module where it once again tries to allocate the transporter. This assumes that any current request for the transporter has a higher priority than the loop entity. Remember that the default priority is 1, with the lowest value having the highest priority.

Once a transporter arrives at its destination, it must be freed. The **Free** module provides this function. You only need to enter the transporter name in the dialog box, and in some cases, the unit number. Two additional modules, **Halt** and **Activate**, allow you to control the number of active or available transporters. The **Halt** module causes a single transporter unit to become inactive or unavailable to be allocated. The **Activate** module causes an inactive transporter to become active or available.

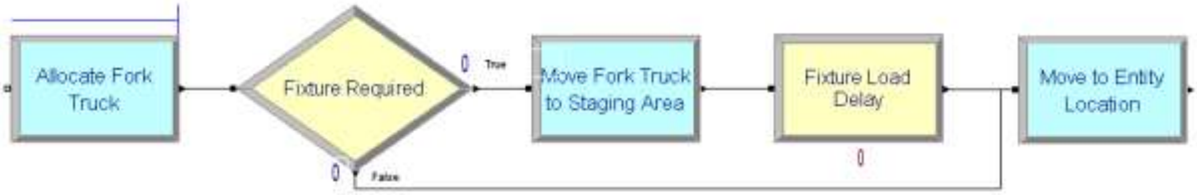


Figure 10.5. Checking for Fixture Requirement

10.3 Entity Reneging

10.3.1 Entity Balking and Reneging

Entity balking occurs when an arriving entity or customer does not join the queue because there is no room, but goes away or goes someplace else. Entity reneging occurs when an entity or customer joins a queue on arrival but later decides to jump out and leave, probably regretting not having balked in the first place. Let's consider the more complex case where it's possible to have both customer balking and customer reneging. First, let's define the various ways that balking and reneging can occur.

In most service systems, there is a finite system capacity based on the amount of waiting space. When all the space is occupied, a customer will not be able to enter the system and will be balked. This is the simplest form of entity balking. Unfortunately, most systems where balking can occur are much more complicated. Consider a simple service line where, theoretically, the queue or waiting-line capacity is infinite. In most cases, there is some finite capacity based on space, but a service line could possibly exit a building and wind around the block (say the waiting line for World Series or Super Bowl tickets). In these cases, there is no concrete capacity limit, but the customers' decisions to enter the line or balk from the system are based on their own evaluation of the situation. Thus, balking point or capacity is often entity-dependent. One customer may approach a long service line, decide not to wait, and balk from the system-yet the next customer may enter the line.

Entity reneging is an even more complicated issue in terms of both modeling and representation in software. Each entity or customer entering a line has a different tolerance threshold with respect to how long to wait before leaving the line. The decision often depends on how much each customer wants the service being provided. Some customers won't renege regardless of the wait time. Others may wait for a period of time and then leave the line because they realize that they won't get served in time to meet their needs. Often the customers' decisions to remain or leave the line are based on both the amount of time they have already spent in the line as well as their current place in the line. A customer may enter a line and decide to wait for ten minutes; if he is not serviced by that time, he plans to leave the line. However, after ten minutes have elapsed, the customer may choose to stay if he is the next in line for service.

Line switching, or jockeying, is an even more complicated form of reneging that often occurs in supermarket checkout lines, fast-food restaurants, and banks that do not employ a single waiting line. A customer selects the line to enter and later re-evaluates that decision based on current line lengths. After all, we invariably enter the slowest moving line, and if we switch lines, the line we left speeds up and the line we enter slows down. We won't cover the logic for jockeying.

10.3.2 Model 10-3. A Service Model with Balking and Reneging

Let's look at balking and reneging in the context of a very simple model. Customers arrive with *EXPO*(5) inter arrival times at a service system with a single server-service time is *EXPO*(4.25). All times are in minutes. Although the waiting line has an infinite capacity, each arriving customer views the current length of the line and compares it to

his tolerance for waiting. If the number in the line is greater than his tolerance, he'll balk away from the system. We'll represent the customer-balking tolerance by generating a sample from a triangular distribution, *TRIA*(3, 6, 15). Since our generated sample is from a continuous distribution, it will not be an integer. We could use one of the Arena math functions to convert it to an integer, but we're only interested if the number in the waiting line is greater than the generated tolerance.

There are two ways to model the balking activity. Let's assume that we create our arrivals, generate our tolerance value from the triangular distribution, and assign this value to an entity attribute. We could send our arrival to a **Decide** module and compare our sample value to the current number in the waiting line, using the Arena variable *NQ*. If the tolerance is less than or equal to the current number in queue, we balk the arrival from the system. Otherwise, we enter the waiting line. An alternative method is to assign our tolerance to a variable and also use this same variable for the server queue capacity. Then we send our arrival directly to the server. If the current number in the queue is greater than or equal to the tolerance, which is equal to the queue capacity, the arrival will be balked automatically. Using the second method, it's possible for the queue capacity to be assigned a value less than the current number in queue. This method works because Arena checks only the queue-capacity value when a new entity tries to enter the' queue. Thus, the current entities remain safely in the queue, regardless of the new queue-capacity value. We will use this second method when we develop our model.

To represent reneging, assume that arriving customers who decide not to balk are' willing to wait only a limited period of time before they renege from the queue. We will generate this renege tolerance time from an *ERLA*(15, 2) distribution (Erlang), which has a mean of 30, and assign it to an entity attribute in our **Create** module. Modeling the mechanics of the reneging activity can be a challenge. If we allow the arrival to enter the queue and the renege time is reached, we need to be able to find the entity and remove it from the queue. At this point in our problem description, you might want to consider alternative methods to handle this. For example, we could define a variable that keeps track of when the server will next be available. We generate the entity-processing time ' first and assign it to an attribute in an **Assign** module. We then send our entity to the **Decide** module where we check for balking. If the entity is not balked, we then check (in the same **Decide** module) to see if the entity will begin service before its renege time. If not, we renege the entity. Otherwise, we send the entity to an **Assign** module where we update our variable that tells us when the server will become available, then send the entity to the queue. This model logic may seem complicated, but can be summarized as follows:

```
Define Available Time = Time in the future when server will be available
Create arrival
    Assign Service Time
    Assign the Time in the future the activity would renege, which is
    equal to Tolerance Time + TNOW
    Assign Balk Limit
Decide
    If Balk Limit > Number in queue Balk entity
    If Renege Time < Available Time Renege entity
    Else
    Assign Available Time = MX(Available Time , TNOW) + Service Time
    Send entity to queue
```

Note the use of the "maximum" math function, MX.

There is one problem with this logic that can be fixed easily-our number in queue is not accurate because it won't contain any entities that have not yet reneged. We can fix this by sending our reneged entities to a **Delay** module where they are delayed by the renege time; we also specify a Storage (e.g., Renege Customers). Now we change our first Decide statement as follows:

```
    If Balk Limit > Number in queue + NSTO(Renege Customers)
```

We have to be careful about our statistics, but this approach will capture the reneging process accurately and avoid our having to alter the queue.

Let's add one last caveat before we develop our model. Assume that the actual decision of whether to renege is based not only on the renege time, but also on the position of the customer in the queue. For example, customers may have reached their renege tolerance limit, but if they're now at the front of the waiting line, they may just wait for service (i.e., renege on reneging). Let's call this position in the queue where the customer will elect to stay, even if the customer renege time has elapsed, the customer stay zone. Thus, if the customer stay zone is 3 and the renege time for the customer has expired, the customer will stay in line anyway if they are one of the next three customers to be serviced.

We'll generate this position number from a Poisson distribution, POIS(0.75). We've used the Poisson distribution because it provides a reasonable approximation of this process and also returns an integer value. For those of you with no access to Poisson tables (you mean you actually sold your statistics book?), it is approximately equivalent to the following discrete empirical distribution: DISC(0.472, 0, 0.827, 1, 0.959, 2, 0.993, 3, 0.999, 4, 1.0, 5). See Appendix D for more detail on this distribution.

This new decision process means that the above logic is no longer valid. We must now place the arriving customer in the waiting line and evaluate the reneging after the renege time has elapsed. However, if we actually go ahead and place the customer in the queue, there's no mechanism to detect that the renege time has elapsed. To overcome this problem, we'll make a duplicate of each entity and delay it by the renege time. The original entity, which represents the actual customer, will be sent to the service queue. After the renege-time delay, we'll have the duplicate entity check the queue position of the original entity. If the customer is no longer in the service queue (that is, the customer was served), we'll just dispose of the duplicate entity. If the customer is still in the queue, we'll check to see if that customer will renege. If the current queue position is within the customer stay zone, we'll just dispose of the duplicate entity. Otherwise, we'll have the duplicate entity remove the original entity from the service queue and dispose of both itself and the original entity. This model logic is outlined as follows:

```
Create Arrivals
    Assign Renege Time = ERLA(15,2)
    Assign arrival time: Enter System = TNOW Assign Stay Zone Number =
    POIS(0.75) Assign Balking Tolerance = TRIA(3,6,15)
Create Duplicate entity
    Original entity to server queue If Balk from queue Count balk Dispose
    Delay for Service Time = EXPO(4.25) Tally system time
    Dispose
Duplicate entity
    Delay by Renege Time
    Search queue for position of original entity If No original entity
    Dispose If Queue position <= Stay Zone Number Dispose
    Remove original entity from queue and Dispose Count Renege customer
    Dispose
```

In order to implement this logic, we obviously need a few new features, such as ability to search a queue and remove an entity from a queue. As you would suspect, Arena modules that perform these functions can be found in the Advanced Process and Block panels. Our completed Arena model (Model 10-3) is shown in *Figure 10.6*.

We start our model with a **Create** module that creates arriving customers, which then sent to the following **Assign** module where the values we'll need later are assigned, as in *Display 10.7*.

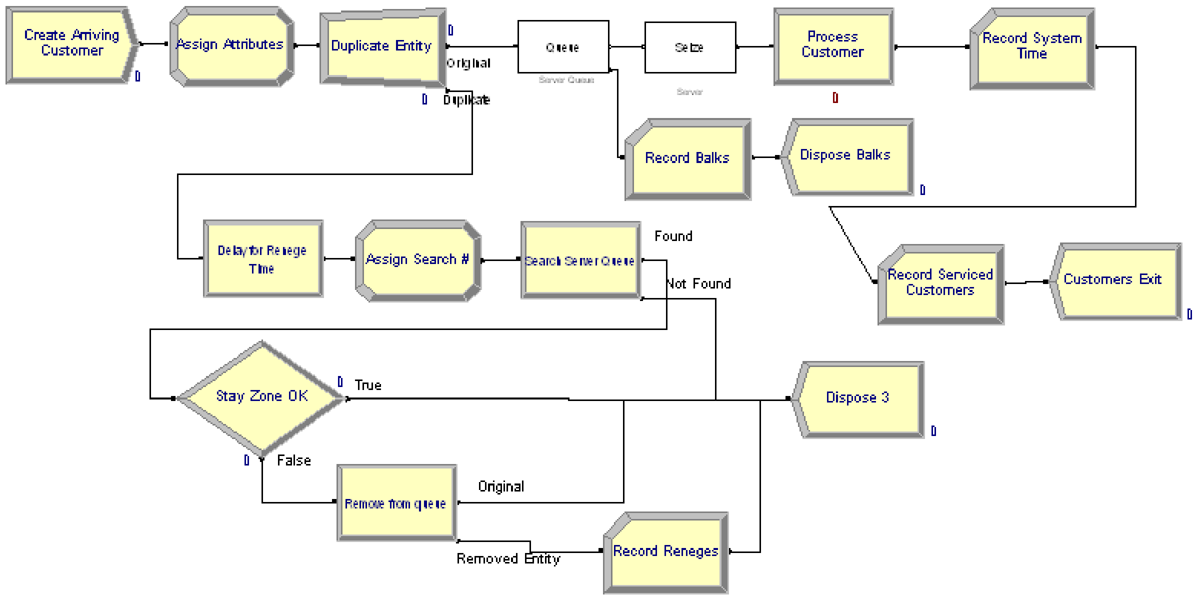


Figure 10.6. The Service Model Logic

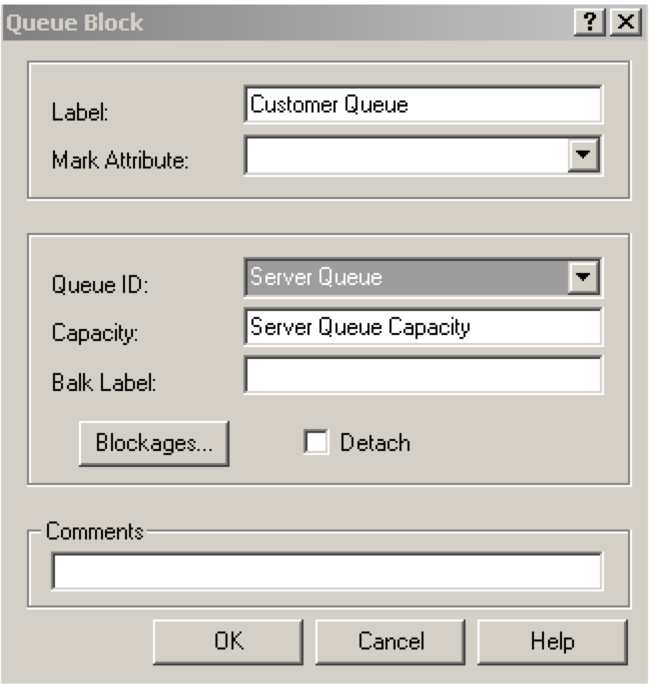
Name	Assign Attributes
Type	Attribute
Attribute Name	Enter System
New Value	TNOW
Type	Attribute
Attribute Name	Renege Time
New Value	ERLA(15, 2)
Type	Variable
Variable Name	Server Queue Capacity
New Value	TRIA (3, 6, 15)
Type	Attribute
Attribute Name	Stay Zone
New Value	POIS (0.75)
Type	Variable
Variable Name	Total Customers
New Value	Total Customers + 1
Type	Attribute
Attribute Name	Customer #
New Value	Total Customers

Display 10.7. The **Assign** module

We send the new arrivals to a **Separate** module. This module allows us to make duplicates (clones) of the entering entity. The original entity leaves the module by the exit point located at the right of the module. The duplicated entities leave the module by the exit points below the module. In this case, we only need to enter the name for our module, accepting the remaining data as the defaults.

The duplicates are exact replicas of the original entity (in terms of attributes and their values) and can be created in any quantity. If you make more than one duplicate of an entity, you can regard them as a batch of entities that will all be sent out of the same (bottom) exit. Note that if you enter a value *n* for # of duplicates, *n*+1 entities actually leave the module *n* duplicates from the bottom connection point and 1 original from the top.

The original entity (customer) is sent to a **Queue - Seize** module sequence (from the Blocks panel) where it tries to enter queue *Server Queue* to wait for the server. In order to implement this method, we need to be able to set the capacity of the queue that precedes the Seize to the variable *Server Queue Capacity*. We accomplish this by attaching the Blocks panel to our model and selecting and placing a **Queue** module. (Note that when you click on the **Queue** module, there is no associated spreadsheet view; this will be the case for any modules from the Blocks or Elements panels.) Before you open the Queue dialog box, you might notice that there is a single exit point at the right of the module. Now we double-click on the module and enter the label, name, and capacity, as shown in *Display 10.8*. After you close the dialog box, you should see a second exit point near the lower right-hand corner of the module. This is the exit that the balked entities will take.



Queue Block

Label: Customer Queue

Mark Attribute:

Queue ID: Server Queue

Capacity: Server Queue Capacity

Balk Label:

□ Detach

Comments

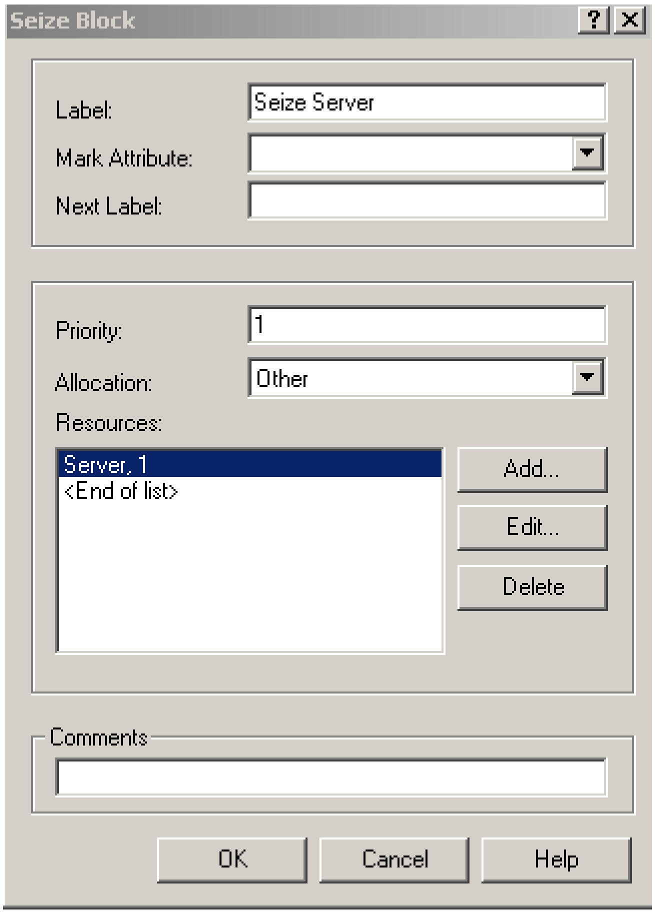
Label	Customer Queue
Queue ID	Server Queue
Capacity	Server Queue Capacity

Display 10.8. The **Queue** module from the Blocks Panel

The capacity of this queue was entered as the variable *Server Queue Capacity*, which was set by our arriving customer as his tolerance for not balking, as explained earlier. If the current number in queue is less than the current value of *Server Queue Capacity*, the customer is allowed to enter the queue. If not, the customer entity is balked to the **Record** module where he increments the balk count and is then sent to a **Dispose** module, where he exits the system (see *Figure 10.6*). A customer who is allowed to enter the queue waits for the resource *Server*.

We need to follow this **Queue** module with a **Seize** module. When you add your **Seize** module, make sure it comes from the Blocks panel and not from the Advanced Process panel. The module in the Advanced Process panel automatically comes with a queue and Arena would become quite confused if you attempted to precede a seize construct with two queues. We then double-click on the module and make the entries shown in *Display 10.9*.

When the customer seizes the server, it is sent to the following **Process** module. This **Process** module uses a *Delay Release* Action, which provides the service delay and releases the server resource for the next customer. A serviced customer is sent to a **Record** module where the system time is recorded, to a second **Record** module where the number is counted, and then to the following **Dispose** module.



Seize Block

Label: Seize Server

Mark Attribute:

Next Label:

Priority: 1

Allocation: Other

Resources:

Server, 1
<End of list>

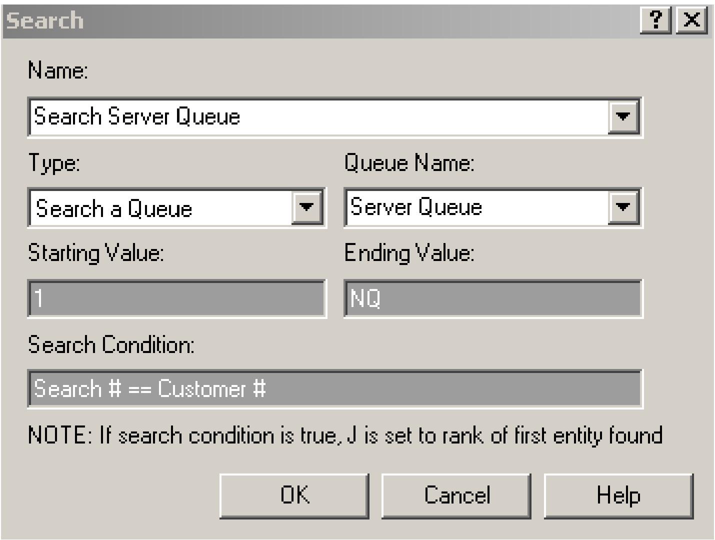
Comments

Label	Seize Server
Resource ID	Server
Number of Units	1

Display 10.9. The **Seize** module from the Blocks Panel

The duplicate entity is sent to a **Delay** module where it is delayed by the renege time that was assigned to the attribute *Renege Time*. After the delay, the entity enters an **Assign** module where the value of the attribute *Customer #* is assigned to a new variable named *Search #*. The attribute *Customer #* contains a unique customer number assigned when the customer entered the system. The entity is then sent to the following **Search** module, from the Advanced Process panel. A **Search** module allows us to search a queue to find the *rank*, or queue position, of an entity that satisfies a defined search condition. A queue rank of 1 means that the entity is at the front of the Queue (the next entity to be serviced). In our model, we want to find the original customer who created the duplicate entity performing the search. That customer will have the same value for its *Customer #* attribute as the variable *Search #* that we just assigned.

The **Search** module (*Display 10.10*) searches over a defined range according to a defined condition. Normally, the search will be over the entire queue contents, from 1 to NQ. (Note that the search can be performed backward by specifying the range as NQ to 1.)



Name: Search Server Queue

Type: Search a Queue

Queue Name: Server Queue

Starting Value: 1

Ending Value: NQ

Search Condition: Search # == Customer #

NOTE: If search condition is true, J is set to rank of first entity found

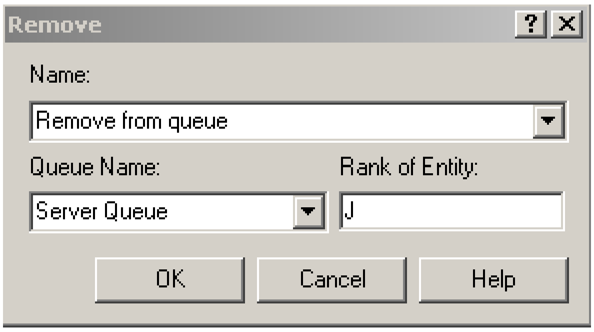
Name	Search Server Queue
Type	Search a Queue
Queue Name	Server Queue
Starting Value	1
Ending Value	NQ
Search Condition	Search # == Customer,#

Display 10.10. The **Search** module

However, you may search over any range that your model logic requires. If you state a range that exceeds the current number in the queue, Arena will terminate with a runtime error. Arena will assign to the variable J the rank of the first entity during the search that satisfies the condition and send the entity out the normal exit point (labeled Found). If the condition contains the math functions MX or MN (maximum or minimum), it will search the entire range. If attributes are used in the search condition, they will be interpreted as the attribute value of the entity in the search queue. If the queue is empty, or no entity satisfies the condition, the entity will be sent out the lower exit point (labeled Not Found). Ordinarily, you're interested in finding the entity rank so you can remove that entity from the queue (**Remove** module) or make a copy of the entity (**Separate** module) The **Search** module can also be used to search over entities that have been formed as a temporary group using a **Batch** module. In addition, you can search over any Arena expression.

For our model, we want to search over the entire queue range, from 1 to NQ, for the original entity that has the same *Customer #* value as the duplicated entity initiating the search. If the original entity, or customer, is no longer in the queue, the entity will exit via the *Not Found* exit point and be sent to the following **Dispose** module. If the original entity is found, its rank in the queue will be saved in the variable *J*. The entity is the sent to the following **Decide** module. The check at this module is to see if the value of *J*: less than or equal to the value of the attribute Stay zone. If this condition is True, implies that the position of the customer in the queue is good enough that he chooses 1 remain in the line. In this case, we dispose of the duplicate entity (see *Figure 10.6*). If the condition is False, we want to renege the original customer. Therefore, we send the duplicate entity to the following **Remove** module.

The **Remove** module allows us to remove an entity from a queue and send it to another place in our model. It requires that you identify the entity to be removed by entering the queue identifier and the rank of that entity. If you attempt to remove an entity from an undefined queue or to remove an entity with a rank that is greater than the number of entities in the specified queue, Arena will terminate the run with an error. In our model, we want to remove the customer with rank *J* from queue *Server Queue*, as shown in *Display 10.11*.



Name: Remove from queue

Queue Name: Server Queue

Rank of Entity: J

Name	Remove from queue
Queue Name	Server Queue
Rank of Entity	J

Display 10.11. The **Remove** module

If you look at the **Remove** module, you'll see two exit points on the right side. The entity that entered the **Remove** module will depart from the upper exit point; in our model, it is sent to the same **Dispose** module we used for the first branch of our **Decide** module. The customer entity removed from the server queue will depart by the lower exit point and is sent to a **Record** module to count the number of customers that renege. The entity is then sent to the **Dispose** module.

We set our Replication length to 2000 minutes. The result of the summary report for this model shows that we had 8 balking and 41 reneging customers with 332 serviced customers.

10.4 Holding and Batching Entities

In this section, we'll take up the common situation where entities need to be held up along their way for a variety of reasons. We'll also discuss how to combine or group entities and how to separate them later.

10.4.1 Modeling Options

As you begin to model more complex systems, you might occasionally want to retain or hold entities at a place in the model until some system condition allows these entities to progress. You might be thinking that we have already covered this concept, in that an entity waiting in a queue for an available resource, transporter, or conveyor space allows us to hold that entity until the resource becomes available. Here we're thinking in more general terms; the condition doesn't have to be based on just the availability of a resource, transporter, or conveyor space. The conditions that allow the entity to proceed can be based on any system conditions; for example, time, queue size, etc. There are two different methods for releasing held entities.

The first method holds the entities in a queue until they receive permission or a signal to proceed from another entity in the system. For example, consider a busy intersection with a policeman directing traffic. Think of the cars arriving at the intersection as entities being held until they are allowed to proceed. Now think of the policeman as an entity eventually giving the waiting cars a signal to proceed. There may be ten cars waiting, but the policeman may give only the first six permission to proceed. We used this method in Model 5-2 to hold the late arrivals.

The second method allows the held entities themselves to evaluate the system conditions and determine when they should proceed. For example, think of a car wanting to turn across traffic into a driveway from a busy street with oncoming traffic; unfortunately, there's neither a traffic light nor a policeman. If the car is the entity, it waits until conditions are such that there is no oncoming traffic within a reasonable distance and the driveway is clear for entry. In this case, the entity continuously evaluates the conditions until it is safe to make the turn. If there's a second car that wants to make the same turn directly behind the first, it waits until it is at the front of the line and then performs its own evaluation. We'll illustrate both methods in Model 10-4 below.

There are also situations where you need to form batches of items or entities before they can proceed. Take the simplest case of forming batches of similar or identical items. ` For example, you're modeling the packing operation at the end of a can line that produces beverages. You want to combine or group beverages into six-packs for the packing operation. You might also have a secondary operation that combines 4 six-packs into a case. In this illustration, the items or entities to be grouped are identical, and you would most likely form a permanent group (that is, one that you'd never want to take apart again later). Thus, six entities enter the grouping process and one entity, the six-pack, exits the process. However, if you're modeling an operation that groups entities that are later to be separated to continue individually on their way, you would want to ' form a temporary group. In the first case, you lose the unique attribute information attached to each entity. In the second case, you want each entity departing the operation ' to retain the same attribute information it held when it joined the group. So when you're modeling a grouping operation, you need to decide whether you want to form a temporary or permanent group. We'll discuss both options in Model 10-4 below.

10.4.2 Model 10-4. A Batching Process Example

Randomly arriving items are into batches before being processed. You might think of the process as an oven that cures the arriving items in batches. The maximum size of the batch that can be sent to the process depends on the design capacity. Let's assume that each item must be placed on a special fixture for the process, and these fixtures are very expensive. The number of fixtures determines the process capacity. Let's further assume that these fixtures are purchased in pairs. Thus, the process can have a capacity of 2, 4, 6, 8, etc. In addition, we'll assume that the process requires a minimum batch size of 2 before it can be started.

Arriving items are sent to a batching area where they wait for the process to become available. When the process becomes available, we must determine the batch size to process, or cause the process to wait for the arrival of enough additional items to make a viable batch. Here's the decision logic required:

```
Process becomes available
    If Number of waiting items >2 and < Max Batch Form batch of all items
        Set Number of waiting items to 0 Process batch
    Else if Number of waiting items > Max Batch Form batch of size Max
    Batch
        Decrement Number of waiting items by Max Batch Process batch
    Else if Number of waiting items < 2
        Wait for additional items
```

As long as there are items available, the items are processed. However, if there are insufficient items (<2) for the next batch, the process is temporarily stopped and requires an additional startup-time delay before the next batch can be processed. Because of this additional startup delay, we may want to wait for more than two items before we restart the process.

We want to develop a simulation model that will aid us in designing the parameters of this process. There is one design parameter (the process capacity or Max Batch) and one logic parameter (restart batch size) of interest. In addition, we would like to limit the number of waiting entities to approximately 25.

We'll design our simulation model and then use **OptQuest** for Arena to search for the best solution based on minimizing the number of restarts.

The completed model that we'll now develop is shown in *Figure 10.7*. You might note that there is one new module: **Batch** from the Basic Process panel. You might also note that we don't have a resource defined for the process; it's not required as the model will limit the number of batches in the oven to 1. Let's start with the **Variable** module and *Run > Setup > Replication Parameters* dialog box and then proceed to the model logic. We'll discuss the **Statistic** data module later.

We use the **Variable** module to define the two parameters that we'll use in **OptQuest**: the maximum batch size or process capacity, *Max Batch*; and the restart batch size, *Restart*. These variables are initially set to 10 and 4, respectively. The Run > Setup > Replication Parameters dialog box specifies a replication length of 10,000 (all times are in minutes).

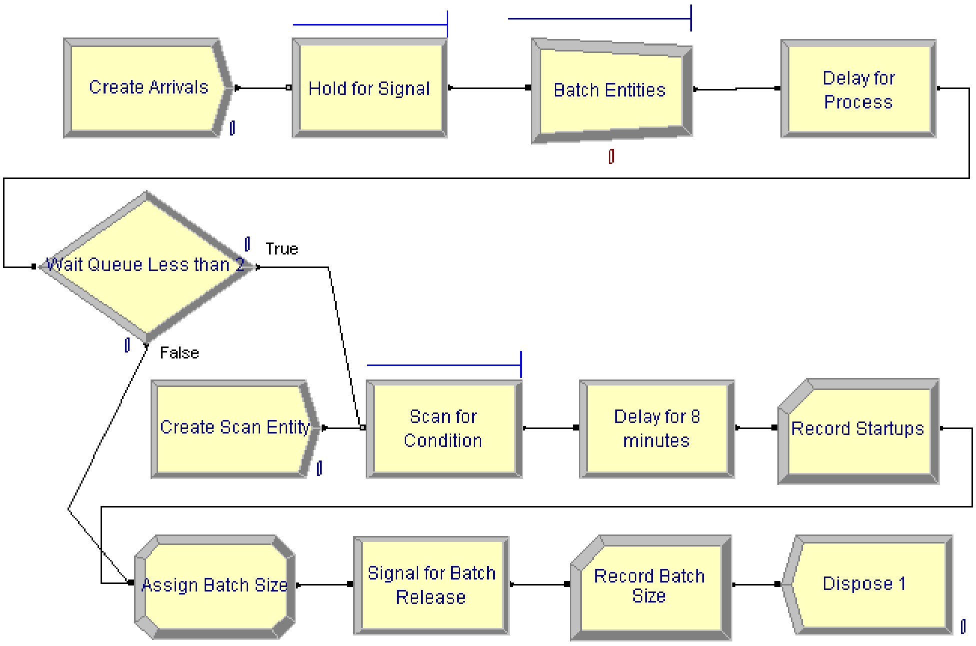


Figure 10.7. The Batch Processing Model

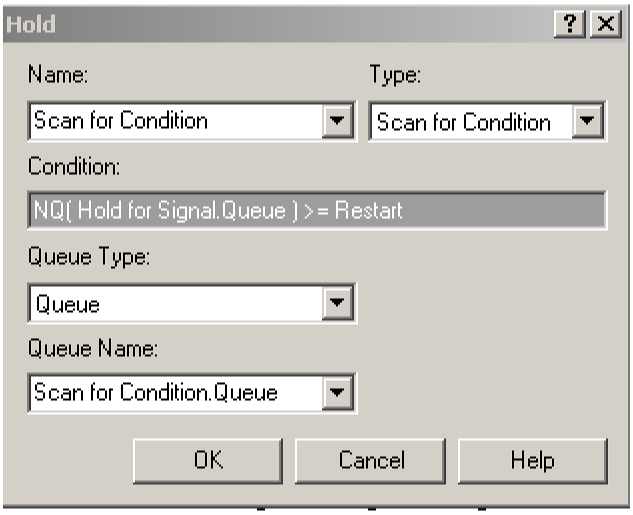
Now let's look at the item arrival process the **Create-Hold** modules at the upper left of *Figure 10.7*. The **Create** module is used to generate arrivals with exponential inter arrival times. We've used a value of 1.1 as the inter arrival mean. No other entries are required for this module. The arriving items are then sent to the **Hold** module from the Advanced Process panel. The **Hold** module holds entities until a matching *signal* is received from elsewhere in the model. The signal can be based on an expression or an attribute value. Different entities can be waiting for different signals, or they can all be waiting for the same signal. When a matching signal is received, the **Hold** module will release up to a maximum number of entities based on the Limit (which defaults to infinity), unless the signal contains additional release limits. This will be explained when we cover the **Signal** module.

In our example, all entities will wait for the same signal, which we have arbitrarily specified as Signal 1. We have defaulted the Limit to infinity since a limit will be set in the **Signal** module. We have also requested an Individual Queue, *Hold for Signal.Queue*, so we can obtain statistics and plot the number in queue for our animation.

Now let's consider the conditions at the start of the simulation. There is nothing being processed; therefore, nothing should happen until the arrival of the fourth item, based on the initial value of Restart. As arriving items do not cause a signal to be sent, some other mechanism must be put into the model to cause the start of the first batching operation and process.

This mechanism can be found in the Create - Hold - Delay - Record, etc., sequence of modules found toward the center of *Figure 10.7*. The *Create Scan Entity* **Create** module has Max Arrivals set to 1, which results in only a single entity being released at time 0. This entity is sent directly to the **Hold** module that follows.

The *Scan for Condition* **Hold** module, with the type specified as Scan for Condition, allows us to hold an entity until the user-defined condition is true; at that time, the entity is allowed to depart the module. The waiting entities are held in a user defined queue (the default) or in an internal queue. If an entity enters a **Hold** module that has no waiting entities, the condition is checked and the entity is allowed to proceed if the condition evaluates to true. If there are other entities waiting, the arriving entity joins the queue with its position based on the selected queue-ranking rule, defaulted to FIFO. If there are entities waiting, the scan condition is checked as the last operation before any discrete-event time advance is triggered anywhere in the model. If the condition is true, the first entity in the scan queue will be sent to the next module. Arena will allow that entity to continue until it's time for the next time advance. At this time, the condition is checked again. Therefore, it's possible for all waiting entities to be released from the **Hold** module at the same time, although each entity is completely processed before the next entity is allowed to proceed.



Name: Scan for Condition

Type: Scan for Condition

Condition: NQ(Hold for Signal.Queue) >= Restart

Queue Type: Queue

Queue Name: Scan for Condition.Queue

Name	Scan for Condition
Type	Scan for Condition
Condition	NQ (Hold for Signal . Queue) >= Restart

Display 10.12. The **Hold** module (Scan for Condition)

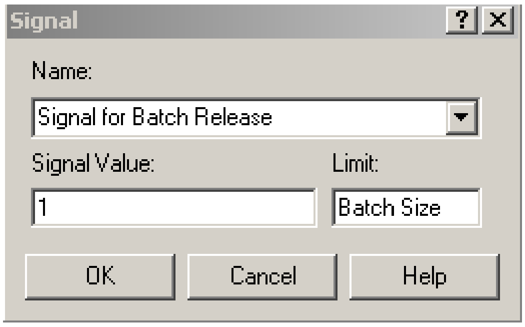
For our model, we've entered a condition that requires at least four items to be in the wait queue preceding our **Hold** module before the condition is satisfied (*Display 10.12*); note that you can use the Arena Expression Builder if you right-click in this field. At that time, the entity will be sent to the following Delay for 8 Minutes **Delay** module. As you begin to understand our complete model, you should realize that we've designed it so there will never be more than one entity in the scan queue.

Now, at the start of a simulation run, the first arriving item will be created and sent to the queue *Hold for Signal.Queue*. At the same time, the second **Create** module will cause a single entity to arrive and be placed in the queue *Scan for Condition.Queue*. Nothing happens until the first queue has four items. At that time, the entity is released from the second **Hold** module and is sent to the **Delay** module where it incurs an 8-minute delay, accounting for the process restart time. Be aware that during this delay additional items may have arrived. The entity is then sent to a **Record** module where the number of startups, counter *Startups*, is incremented by 1. The new process batch size, *Batch Size*, is calculated in the following **Assign** module, in *Display 10.13*. Recall that, at least for the first entity, we know there are at least four items in the wait queue. Thus, our process batch size is either the number in the wait queue or the maximum batch size if the number waiting is greater than the process capacity.

Name	Assign Batch Size
Type	Variable
Variable Name	Batch Size
New Value	MN (NQ (Hold for Signal.Queue), Max Batch)

Display 10.13. Assigning the Next Process Batch Size

Having calculated the next batch size and assigned it to a global variable, *Batch Size*, we send the entity to a **Signal** module, seen in *Display 10.14*. This module broadcasts a signal with a value of 1 across the whole model, which causes the entities in the wait queue, up to a maximum *Batch Size*, to be released. This entity then enters a **Record** module where the next batch size is tallied, and then the entity is disposed.



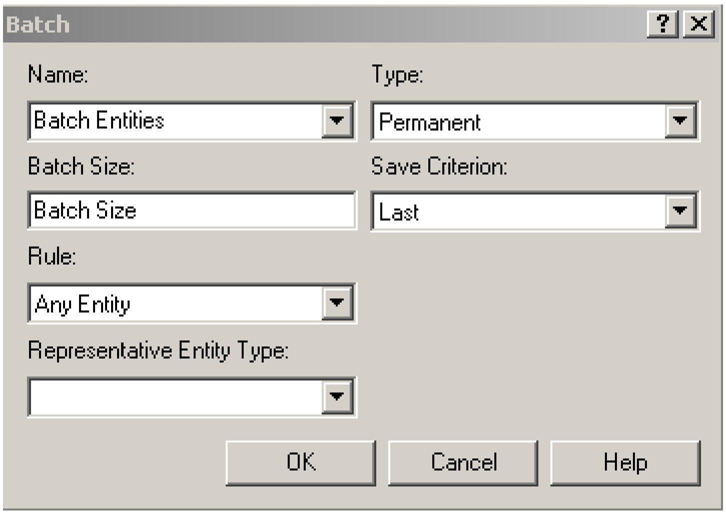
Name	Signal for Batch Release
Signal Value	1
Limit	Batch Size

Display 10.14. The **Signal** module

You can have multiple **Signal** and multiple **Hold** modules in your model. In this case, a **Signal** module will send a signal value to each **Hold** module with the Type specified as *Hold for Signal* and release the maximum specified number of entities from all **Hold** modules where the Signal matches.

Now the process has undergone a startup delay, and the first batch of items has been released to the *Batch Entities* **Batch** module following the *Hold for Signal* **Hold** module. The **Batch** module allows us to accumulate entities that will then be formed into a permanent or temporary batch represented by a single entity. In our example, we have decided to form a permanent batch. Thus, the unique attribute values of the batched entities are lost because they are disposed. The attribute values of the resulting representative entity can be specified as the Last, First, Product, or Sum of the arriving individual batched entities. If a temporary batch is formed, the entities forming the batch are removed from the queue and are held internally to be reinstated later using a **Separate** module. Entities may also be batched based on a match criterion value. If you form a permanent batch, you can still use the **Separate** module to recreate the equivalent number of entities later. However, you've lost the individual attribute values of these entities.

Normally, entities arriving at a **Batch** module will be required to wait until the required batch size has been assembled. However, in our model, we defined the batch size to be formed to be exactly equal to the number of items we just released to the module; see *Display 10.15*. Thus, our items will never have to wait at this module.



Name: Batch Entities

Type: Permanent

Batch Size: Batch Size

Save Criterion: Last

Rule: Any Entity

Representative Entity Type:

Name	Batch Entities
Batch Size	Batch Size

Display 10.15. Forming the Process Batch

The entity that now represents the batch of items is directed to the *Delay for Process* **Delay** module where the process delay occurs. Note that this process delay depends on the batch size being processed: 3 minutes plus 0.6 minute for each item in the batch.

The processed batch is sent to the following *Wait Queue Less Than 2* **Decide** module where we check whether there are fewer than two items in the wait queue. If so, we must shut down the process and wait for more items to arrive. We do this by sending the entity to the previously discussed *Scan for Condition* **Hold** module to wait for enough items to restart the process. If there are at least two waiting items, the entity is sent to the *Assign Batch Size* **Assign** module where we set the next batch size to start the next batch processing.

Since the preceding few paragraphs were fairly complex, let's review the logic that controls the batching of items to the process. Keep in mind that the arriving entities are placed in a Hold queue where they are held until a signal to proceed is received. The first process is initiated by the second **Create** module that creates only one control entity. This entity is held in the Scan queue until the first four items have arrived. The value 4 is the initial value of the Variable *Restart*. This control entity causes the first batch to be released for processing, and it is then disposed. After that, the last batch to complete processing becomes a control entity that determines the next batch size and when to allow the batch to proceed for processing.

Before we're ready to run our model in **OptQuest**, we need to define an output statistic in the **Statistic** data module that will allow us to limit the average number of entities waiting in the queue *Hold for Signal.Queue*. We've given it the name *Max Batch Value*. The expression is DMAX(*Hold for Signal.Queue.NumberInQueue*). The function DMAX will return the maximum value of an Arena time-persistent statistic. The statistic of interest is automatically generated as part of the output report. The name of the statistic is the queue name followed by *.NumberInQueue*. The interested reader can find this type of information in the help topic "Statistics (Automatically Generated by SIMAN)."

We set up **OptQuest** to vary the variable *Max Batch* from 4 to 14 in discrete increments of 2. The *Restart* variable was varied from 2 to 10 in discrete increments of 1. The objective was to minimize the number of startups with a requirement that our *Max Batch Value* statistic not exceed 25. The best solution found was:

Max Batch = 10
Restart = 8
Max Batch Value = 24.9
Startups = 30.6.

10.5 Overlapping Resources

In Chapters 4-8, we concentrated on building models using the modules available from the Basic Process panel, the Advanced Process panel, and the Advanced Transfer panel. Even though we used these modules in several different models, we still have not exhausted all the capabilities.

As we developed models in the earlier chapters, we were not only interested in introducing you to new Arena constructs, but we also tried to cover different modeling techniques that might be useful. We have consistently presented new material in the form of examples that require the use of new modeling capabilities. Sometimes the fabrication of a good example to illustrate the need for new modeling capabilities is a daunting task. We have to admit that the example that we are about to introduce is a bit of a stretch, not only in terms of the model description, but also the manner in which we develop the model. However, if you bear with us through this model development, we think that you will add several handy additions to your toolbox.

10.5.1 System Description

The system we'll be modeling is a tightly coupled, three-workstation production system. We have used the words "tightly coupled" because of the unique part-arrival process and because there is limited space for part buffering between the workstations.

We'll assume an unlimited supply of raw materials that can be delivered to the system on demand. When a part enters the first workstation, a request is automatically forwarded to an adjoining warehouse for the delivery of a replenishment part. Because the warehouse is performing other duties, the replenishment part is not always delivered immediately. Rather than model this activity in detail, we'll assume an exponential delivery delay, with mean of 25 (all times are minutes), before the request is acted upon. At that point, we'll assume the part is ready for delivery, with the delivery time following a UNIF(10, 15) distribution. To start the simulation, we'll assume that two parts are ready for delivery to the first workstation.

Replenishment parts that arrive at the first workstation are held in a buffer until the workstation becomes available. A part entering the first workstation immediately requests a setup operator. The setup time is assumed to be *EXPO*(9). Upon completion of the setup, the part is processed by the workstation, lasting *TRIA*(10, 15, 20). The completed part is then moved to the buffer between Workstations 1 and 2. This buffer space is limited to two parts; if the buffer is full, Workstation 1 is blocked until space becomes available. We'll assume that all transfer times between workstations are negligible, or occur in 0 time.

There are two almost-identical machines at Workstation 2. They differ only in the time it takes to process a part: The processing times are *TRIA*(35, 40, 45) for Machine 2A and *TRIA*(40, 45, 50) for Machine 2B. A waiting part will be processed by the first available machine. If both machines are available, Machine 2A will be chosen. There is no setup required at this workstation. A completed part is then transferred to Workstation 3. However, there is no buffer between Workstations 2 and 3. Thus, Workstation 3 must be available before the transfer can occur. (This really affects the system performance!) If Machines 2A and 2B both have completed parts (which are blocking these machines) waiting for Workstation 3, the part from Machine 2A is transferred first.

When a part enters Workstation 3, it requires the setup operator (the same operator used for setup at Workstation 1). The setup time is assumed to be *EXPO*(9). The process time at Workstation 3 is *TRIA*(9, 12, 16). The completed part exits the system at this point.

We also have failures at each workstation. Workstations 1 and 3 have a mean Up Time of 600 minutes with a mean Down Time for repair of 45 minutes. Machines 2A and 2B, at Workstation 2, have mean Up Times of 500 minutes and mean Down Times of 25 minutes. All failure and repair times follow an exponential distribution. One subtle but very important point is that the Up Time is based only on the time that the machines are processing parts, not the elapsed time.

Now to complicate the issue even further, let's assume that we're interested in the percent of time that the machines at each workstation are in different states. This should give us a great amount of insight into how to improve the system. For example, if the machine at Workstation 1 is blocked a lot of the time, we might want to look at increasing the capacity at Workstation 2.

The possible states for the different machines are as follows:

Workstation 1: Processing, Starved, Blocked, Failed, Waiting for setup operator, and Setup

Machines 2A and 2B: Processing, Starved, Blocked, and Failed

Workstation 3: Processing, Starved, Failed, Waiting for setup operator, and Setup.

We would also like to keep track of the percent of time the setup operator spends at Workstations 1 and 3. These states would be: WS 1 Setup, WS 2 Setup, and Idle.

These are typical measures used to determine the effectiveness of tightly coupled systems. They provide a great deal of information on what are the true system bottlenecks. Well, as long as we've gone this far, why not go all the way! Let's also assume that we'd like to know the percent of time that the parts spend in all possible states. Arranging for this is a much more difficult problem. First, let's define the system or cycle time for a part as starting when the delivery is initiated and ending when the part completes processing at Workstation 3. The possible part states are: Travel to WS 1, Wait for WS 1, Wait for setup at WS l, Setup at WS 1, Process at WS 1, Blocked at WS 1, Wait for WS 2, Process at WS 2, Blocked at WS 2, Wait for setup at WS 3, Setup at WS 3, and Process at WS 3. As we develop our model, we'll take care to incorporate the resource states. But, we'll only consider the part states after we have completed the model development. You'll just have to trust that we might know what we're doing.

10.5.2 Model 10-5. A Tightly Coupled Production System

In developing our model, we'll use a variety of different modules. Let's start with the modules for the arrival process and Workstation 1, which are shown in *Figure 10.8*.

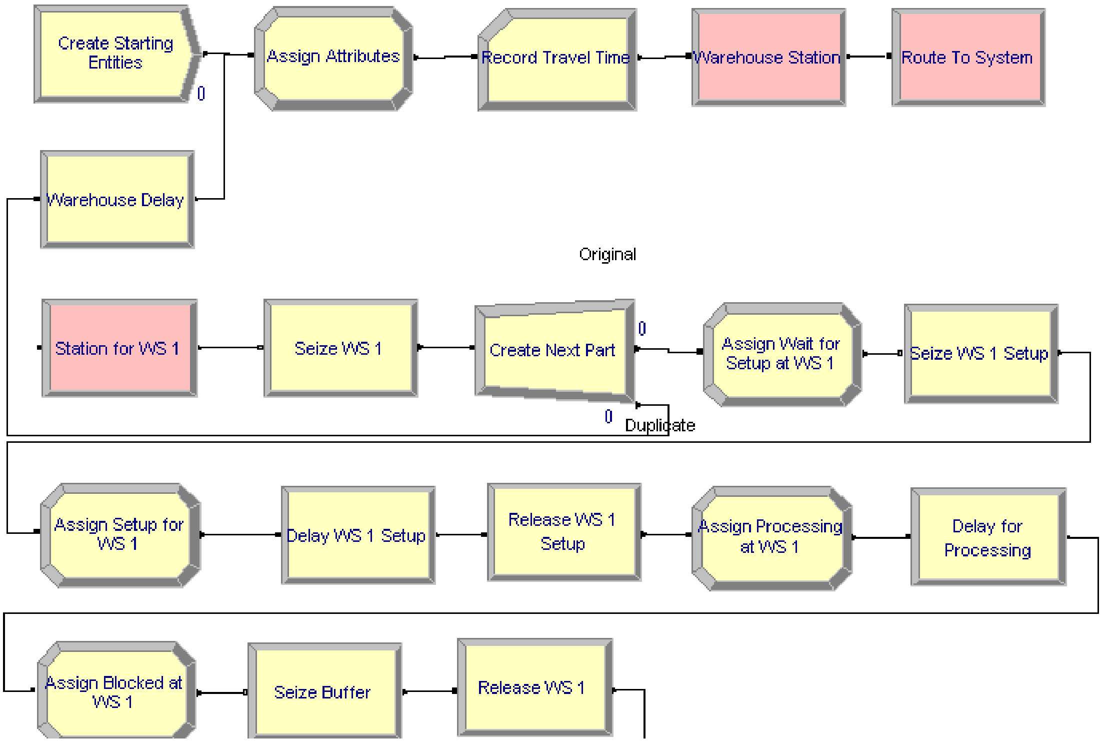


Figure 10.8. Part Arrival and Workstation 1

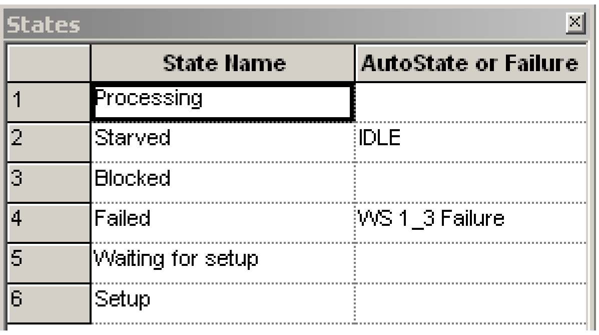
Since you have seen almost all the modules that we'll be using in this model, we won't provide all the displays. However, we will provide sufficient information for you to re-create the model on your own. Of course, you can always open Model 10-05.doe and follow along.

Let's start with the initial arrivals that provide the first two parts for the system. All the remaining arrivals will be based on a request issued when a part enters the machine at the first workstation. The single **Create** module found at the upper left of *Figure 10.8* has three entries; a Name, a Batch Size of 2, and Max Arrivals of 2. This causes two entities, or parts, to be created at time 0. The **Create** module then becomes inactive-no more entities are created by it for the rest of the simulation. These two parts are sent to an **Assign** module where we make two attribute assignments. We assign the value of TNOW to the attribute Enter System and assign a value generated from UNIF(10, 15) to the attribute *Route Time*. The value assigned to Enter System is the time the part entered the system, and the *Route Time* is the delivery time from the warehouse to the first workstation. Since we are interested in keeping detailed part-status information, we send these parts to a **Record** module where we tally the *Delivery Time* based on the expression *Route Time*, which we assigned in the previous **Assign** module. The part is then sent to the following **Station** module where we define the Station Warehouse. Next we use the **Route** module to route from the station *Warehouse* to the station *WS 1 Station* using the *Route Time* we previously assigned.

Upon completion of this transfer, the part arrives at the **Station** module, *Station for WS 1*. The following **Seize** module attempts to seize 1 unit of resource WS 1. We now need to take care of three additional requirements. First, we need to specify the resource states and make sure that the statistics are kept correctly. Second, we need to make a request for a replenishment part. Finally, we need to have a setup occur before the part is processed.

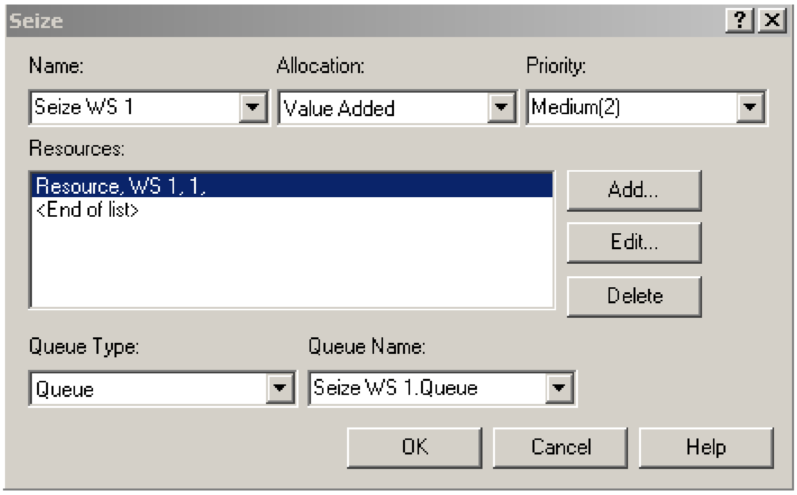
Let's start by defining our resources using the **Resource** data module. We enter six resources: *WS 1*, *WS 2A*, *WS 2B*, *WS 3*, *Setup Operator*, and *Buffer*. The first five have capacity of 1 and the *Buffer* resource has a capacity of 2.

Back in Section 4.2.4, we showed you how to use Frequencies to generate frequency statistics on the number in a queue. In order to get frequency data on our resources, we first need to define our StateSets. We do this using the **StateSet** data module from the Advanced Process panel. We entered five StateSets: *WS 1 Stateset*, *WS 2A StateSet*, *WS 2B StateSet*, *WS 3 StateSet*, and *Setup Operator StateSet*. The states for the *WS 1 StateSet* are shown in *Display 10.16*.



	State Name	AutoState or Failure
1	Processing	
2	Starved	IDLE
3	Blocked	
4	Failed	WS 1_3 Failure
5	Waiting for setup	
6	Setup	

Display 10.16. The WS 1 StateSet States



Name: Seize WS 1

Allocation: Value Added

Priority: Medium(2)

Resources:

Resource, WS 1, 1,
<End of list>

Queue Type: Queue

Queue Name: Seize WS 1.Queue

Name	Seize WS 1
Allocation	Value Added
Resources	
Type	Resource
Resource Name	WS 1

Display 10.17. The WS 1 **Seize** module

We then entered our two failures (*WS 1 3 Failure* and *WS 2 Failure*) using the **Failure** data module. Now we need to go back to the **Resource** data module and add our StateSets and Failures. When we added our failures, we selected *Ignore* as the Failure Rule.

Now let's go back to our model logic in *Figure 10.8*. Having dealt with the entry to WS 1, we need to seize the WS 1 resource, shown in *Display 10.17*.

Next we need to take care of the part-replenishment and the setup activity. The part that has just Seized the workstation resource enters the following **Separate** module, which creates a duplicate entity. The duplicate entity is sent to the *Warehouse Delay*. **Delay** module, which accounts for the time the part-replenishment request waits until the next part delivery is initiated, lasting *EXPO*(25) . We then send this entity to the same set of modules that we used to cause the arrival of the first two parts. The original entity exits the **Separate** module to an **Assign** module where we assign the state of the resource *WS 1* to *Waiting for Setup Operator*, in *Display 10.18*.

Name	Assign Wait for Setup at WS 1
Assignments	
Type	Other
Other	STATE (WS 1)
New Value	Waiting for Setup

Display 10.18. Resource State Assignment

It is then directed to the *Seize WS 1 Setup* **Seize** module where it requests the setup operator, in *Display 10.19*. Note that we specify the state of the setup resource as *WS 1 Setup*. This will allow us to obtain frequency statistics on the *Setup Operator* Resource.

Name	Seize WS 1 Setup
Resources	
Type	Resource
Resource Name	Setup Operator
Resource State	WS 1 Setup

Display 10.19. Seizing the Setup Operator

By now, you may be scratching your head and asking, "What are they doing?" Well, we warned you that this problem was a little contorted, and once we finish the development of Workstation 1, we'll provide a high-level review of the entire sequence of events. So let's continue.

In the following *Assign Setup for WS 1* **Assign** module, we set the *WS 1* resource to the *Setup* state and then we delay in the following *Delay WS 1 Setup* Delay block for the setup activity. Upon completion of the setup, we release the *Setup Operator* resource, assign the *WS 1* resource state to *Processing*, and undergo the processing delay. After processing, we assign the *WS 1* resource state to *Blocked*. At this point, we're not sure that there is room in the buffer at Workstation 2. We now enter the *Seize Buffer Seize* module to request buffer space, from the resource *Buffer*. Once we have the buffer resource, we release the WS 1 resource and depart for Workstation 2.

Now (we feel like we're out of breath after running through this logic), let's review the sequence of activities for a part at Workstation 1. We start by creating two parts at time 0; let's follow only one of them. That part is time stamped with its arrival time, and a delivery time is generated and assigned. The delivery time is recorded and the part is routed to Workstation 1. Upon entering Workstation l, it joins the queue to wait for the resource WS 1. Having seized the resource, it exits the **Seize** module, where it duplicates the replenishment part, which is sent back to where the first part started. The part then assigns the server resource state to *Waiting for setup* and queues for the setup operator. Having seized the setup operator, setting its state to *Setup*, it assigns the *WS 1* resource state to *Setup*, delays for the setup, releases the setup operator, assigns the *WS 1* resource state back to *Processing*, and delays for processing.

After processing, the part queues to seize one unit of the resource *Buffer* (capacity of 2), setting the server state to Blocked during the queuing time. After seizing the buffer resource, it releases the WS 1 resource and exits the Workstation 1 area with control of one unit of the resource *Buffer*.

The modules for Workstation 2, which has two machines, are shown in *Figure 10.9*. A part arriving at the Workstation 2 logic enters a queue in the *Seize WS 2* **Seize** module to wait for one of the two machines at this workstation. We first create a resource set (using the **Set** module) named *WS 2* Set containing the two resources at Workstation 2, *WS 2A* and *WS 2B*. We select our resource from this set saving the resource index in the attribute *WS 2 Resource*. We chose the Preferred Order rule so that if both machines were idle, the faster machine (*WS 2A*) would be selected.

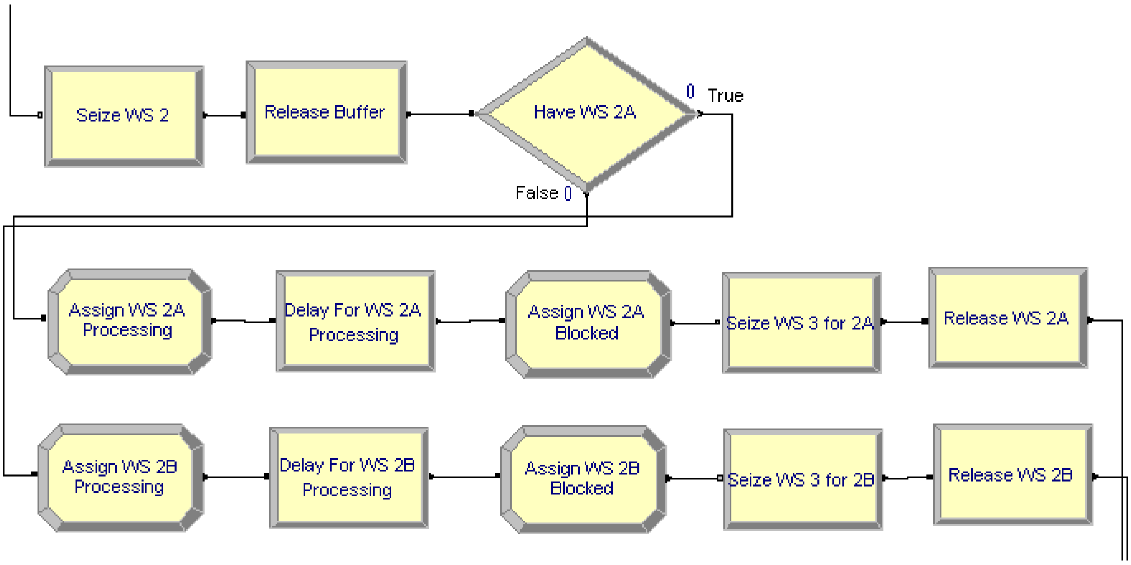


Figure 10.9. Workstation 2 modules

Recall that parts arriving at Workstation 2 have control of one unit of the buffer resource. Thus, as soon as the part seizes one of the two machines, the buffer must be released because there is now a free space in front of the workstation. This action is accomplished in the following *Release Buffer* **Release** module.

Upon seizing an available machine and releasing the buffer resource, the part enters the *Have WS 2A* **Decide** module, which is used to determine which machine resource has been seized, based on the *WS 2 Resource* attribute value assigned in the first **Seize** module when the resource was allocated to the part. If the part were allocated resource *WS 2A*, the first resource in the set, this attribute would have been assigned a value of 1. We use this logic to determine which machine we have been allocated and send the part to logic specifically for *WS 2A* or *WS 2B*. The middle five modules in *Figure 10.9* are for *WS 2A* and the bottom five are for *WS 2B*. Since there is no setup at Workstation 2, the first **Assign** module assigns the state of the machine resource to *Processing* and directs the part to the following **Delay** module, which represents the part processing. The last **Assign** module assigns the machine resource state to *Blocked*.

The part then attempts to seize control of resource *WS 3* in the following **Seize** module. If the resource is unavailable, the part will wait in queue *Seize WS 3.Queue*, which we defined in the **Queue** module as being a shared queue between the two **Seize** modules. Now, if there are two waiting parts, one for each machine, we want the part from *WS 2A* to be first in line. We accomplish this by selecting the ranking rule to be Low Attribute Value based on the value of attribute *WS 2 Resource*. This causes all entities in the queue to be ranked according to their value for the attribute *WS 2 Resource*. This assures that *WS 2A* will receive preference.

Once a part has been allocated the *WS 3* resource, it will be transferred to that machine in 0 time. The modules we used to model Workstation 3 are shown in *Figure 10.10*. You might notice that these modules look very similar to those used to model Workstation 1, and they are essentially the same. An entering part (it already has been allocated the resource WS 3) first assigns the workstation resource state to *Waiting for setup* and then attempts to seize the setup operator. After being allocated the setup operator resource, the resource *WS 3* state is set to *Setup*, the part is delayed for setup, the setup operator is released, the resource *WS 3* state is set to *Processing*, and the part is delayed for the process time. Upon completion, the *WS 3* resource is released, the part cycle time is recorded, and the entity is disposed.

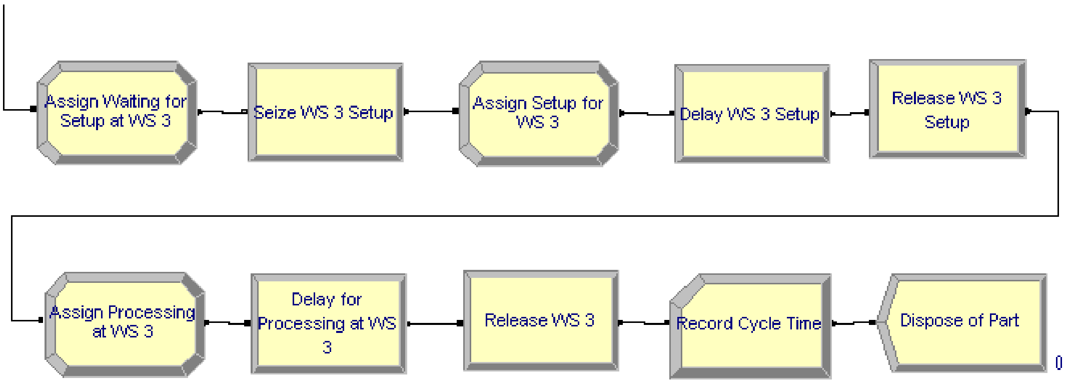
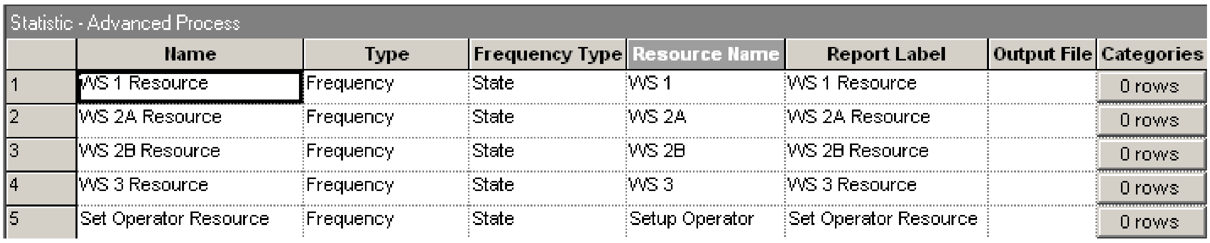


Figure 10.10. Workstation 3 modules

Now that we've completed our model logic, we need to request the frequency statistics for our workstations and setup operator. We want frequency statistics based on the resource states. The required inputs for the **Statistic** module to obtain these frequencies are shown in *Display 10.20*.

Recall that when we were describing our problem, we also wanted to output the percent of time that the parts spend in all possible states. Obtaining this information for resource states is fairly easy. We simply define the resource states and Arena collects and reports this information using the Frequencies feature. Unfortunately, this use of Frequencies is valid only for resources.


Statistic - Advanced Process

	Name	Type	Frequency Type	Resource Name	Report Label	Output File	Categories
1	WS 1 Resource	Frequency	State	WS 1	WS 1 Resource		0 rows
2	WS 2A Resource	Frequency	State	WS 2A	WS 2A Resource		0 rows
3	WS 2B Resource	Frequency	State	WS 2B	WS 2B Resource		0 rows
4	WS 3 Resource	Frequency	State	WS 3	WS 3 Resource		0 rows
5	Set Operator Resource	Frequency	State	Setup Operator	Set Operator Resource		0 rows

Display 10.20. The **Statistic** module for Frequency Statistics

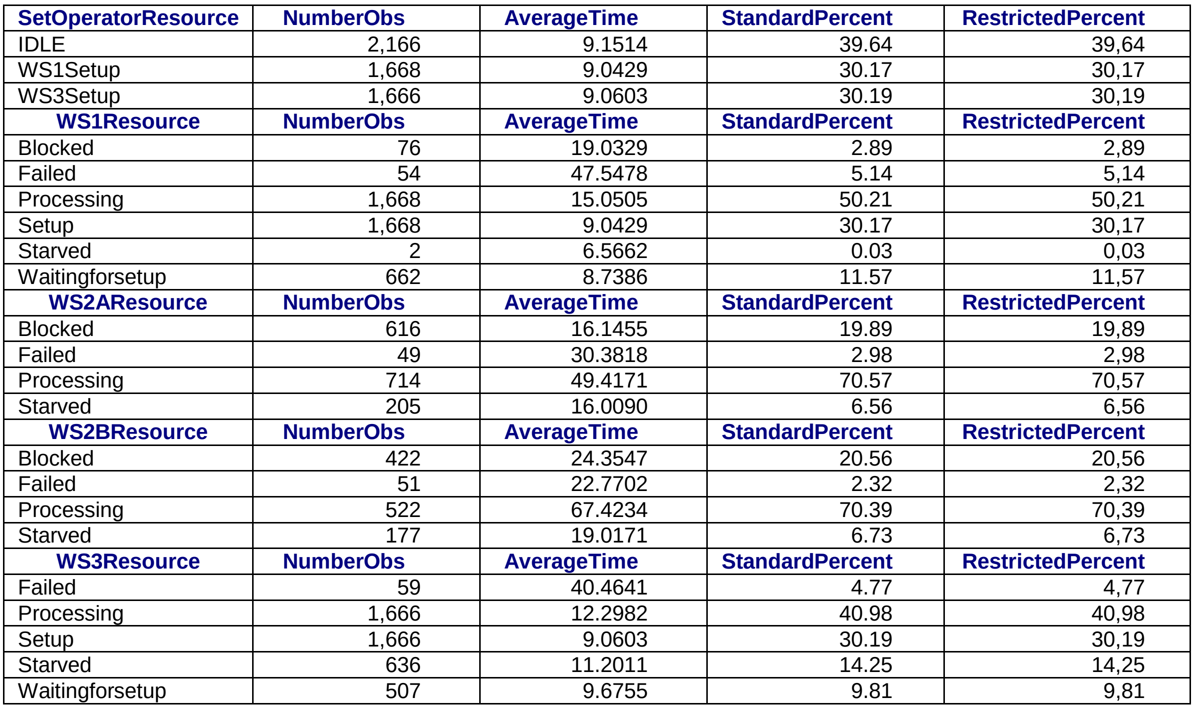
Collecting the same type of information for part or entity status is difficult because the part states span numerous activities over several resources. This creates a modeling problem: What is the best way to obtain this information? Before we show you our approach, we'll discuss several alternatives, including their shortcomings.

Our first thought was to define a variable that we could change based on the current part state. Then we could request frequency statistics on that variable, much as we did for the part-storage queue in the rework area of Model 4-2 in Section 4.2.4. Of course, this would require us to edit our model to add assignments to this variable whenever the part status changed. Although this sounds like a good idea, it falls apart when you realize that there can be multiple parts in the system at the same time. It would work just fine if we limited the number of parts in our system to one. Since this was not in the problem description, we decided to consider alternate ways to collect this information.

Our second idea was to assume that all the required information was already being collected (a valid assumption). Given this, we need only assemble this information at the end of the run and calculate our desired statistics. Arena automatically calls a wrap-up routine at the end of each run. We could write user code (see Section 10.2) that would perform this function. One drawback is that it would not give us this information if we decided to look at our summary report before the run ended. This looked like it could be a lot of work, so we explored other options.

For our third approach, we decided to consider adding additional output statistics to the **Statistic** data module. This would not allow you to collect additional statistics; however, this does allow you to calculate additional output values on statistics already being collected by the model. Since the model is currently collecting all the information we need, although not in the correct form, we will use this option to output information on part status. First we entered a Replication Length of 50,000 time units in the *Run>Setup>Replication Parameter* dialog box.

If we ran our model at this point, we'd get the results shown in *Figure 10.11*. Let's temporarily focus our attention on the frequency statistics for *WS 1 Resource*. It tells us that the workstation was processing parts only 50.21% of the time, with a large amount of non-productive time spent in the *Waiting for setup* and *Setup* states.



SetOperatorResource	NumberObs	AverageTime	StandardPercent	RestrictedPercent
IDLE	2,166	9.1514	39.64	39,64
WS1Setup	1,668	9.0429	30.17	30,17
WS3Setup	1,666	9.0603	30.19	30,19
WS1Resource	**NumberObs**	**AverageTime**	**StandardPercent**	**RestrictedPercent**
Blocked	76	19.0329	2.89	2,89
Failed	54	47.5478	5.14	5,14
Processing	1,668	15.0505	50.21	50,21
Setup	1,668	9.0429	30.17	30,17
Starved	2	6.5662	0.03	0,03
Waitingforsetup	662	8.7386	11.57	11,57
WS2AResource	**NumberObs**	**AverageTime**	**StandardPercent**	**RestrictedPercent**
Blocked	616	16.1455	19.89	19,89
Failed	49	30.3818	2.98	2,98
Processing	714	49.4171	70.57	70,57
Starved	205	16.0090	6.56	6,56
WS2BResource	**NumberObs**	**AverageTime**	**StandardPercent**	**RestrictedPercent**
Blocked	422	24.3547	20.56	20,56
Failed	51	22.7702	2.32	2,32
Processing	522	67.4234	70.39	70,39
Starved	177	19.0171	6.73	6,73
WS3Resource	**NumberObs**	**AverageTime**	**StandardPercent**	**RestrictedPercent**
Failed	59	40.4641	4.77	4,77
Processing	1,666	12.2982	40.98	40,98
Setup	1,666	9.0603	30.19	30,19
Starved	636	11.2011	14.25	14,25
Waitingforsetup	507	9.6755	9.81	9,81

Figure 10.11. The Tightly Coupled System Frequencies Report

9.5.3 Model 10-6. Adding Part-Status Statistics

Before we continue with our model development, let's describe what we need to do in order to obtain the desired information on part status. What we want is the percent of time that parts spend in each of the previously defined part states: Travel to WS 1, Wait for WS 1, Wait for Setup at WS 1, Setup at WS 1, Process at WS 1, Blocked at WS 1, Wait for WS 2, Process at WS 2, Blocked at WS 2, Wait for Setup at WS 3, Setup at WS 3, and Process at WS 3.

All the information we need to calculate these values is already contained in our summary output. Let's consider our first part state, *Travel to WS 1*. The average delivery time per part was 12.517, tallied for a total of 1672 parts. The cycle time was 225.07 for 1665 parts. You might note that there were still seven parts (1672-1665) in the system when the simulation terminated. We'll come back to this later. So if we want the percent of time an average part spent traveling to Workstation 1, we could calculate that value with the following expression:

```
        ((12.517 * 1672) / (225.07 * 1665)) * 100.0
```

or 5.5848%. We can use this approach to calculate all of our values. Basically, we compute the total amount of part time spent in each activity, divide it by the total amount of part time spent in all activities, and multiply by 100 to obtain the values in percentages. Since the last two steps of this calculation are always the same, we will first define an expression, Tot, to represent this value (**Expression** data module). Note that by using an expression, it will be computed only when required. That expression is as follows:

```
        TAVG(Cycle Time) * TNUM(Cycle Time)/100
```

TAVG and TNUM are Arena variables that return the current average of a Tally and the total number of Tally observations, respectively. The variable argument is the Tally ID. In this case, we have elected to use the Tally name as defined in our **Record** module. In cases where Arena defines the Tally name (for example, for time-in-queue tallies), we recommend that you check a module's drop-down list or Help for the exact name.

The information required to calculate three of our part states is contained in Tallies: *Travel to WS 1*, *Wait for WS 1*, and *Wait for WS 2*. The expressions required to calculate these values are as follows:

```
        TAVG(Delivery Time) * TNUM(Delivery Time)/Tot
        TAVG(Seize WS l.Queue.WaitingTime) *
             TNUM(Seize WS l.Queue.WaitingTime)/Tot
        TAVG(Seize WS 2.Queue.WaitingTime) *
             TNUM(Seize WS 2.Queue.WaitingTime)/Tot
```

The information for the remaining part states is contained in the frequency statistics. As you would expect, Arena also provides variables that will return information about frequencies. The Arena variable FRQTIM returns the total amount of time that a specified resource was in a specified category, or state. The complete expression for this variable is

```
        FRQTIM(Frequency ID, Category)
```

The Frequency ID argument is the frequency name. The Category arguxnent is the category name. Thus, our expression for *Setup at WS 1* becomes

```
        FRQTIM(WS 1 Resource, Setup)/Tot
```

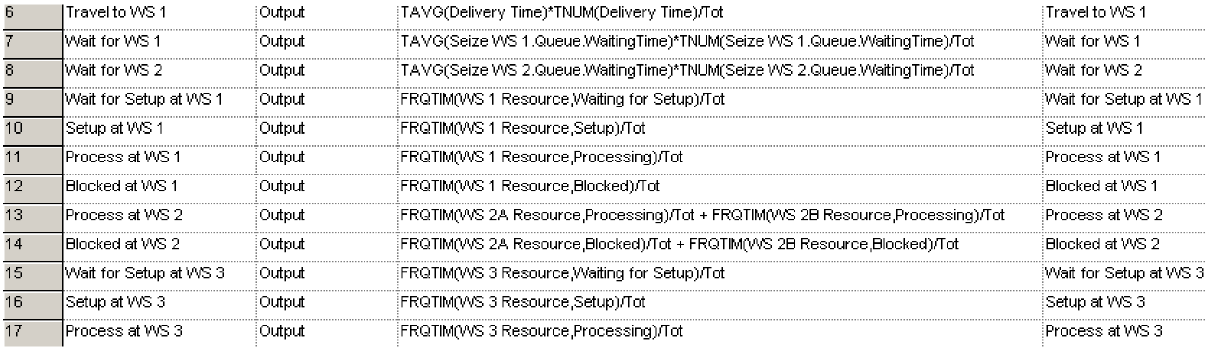
where *WS 1 Resource* is the name of our previously defined frequency statistic (defined in the **Statistic** data module) and *Setup* is the category name for our setup state for the *WS 1* resource (defined in the **StateSet** data module).

If we define all our expressions in this manner, the nine remaining expression are:

```
   Wait for Setup at WS 1     FRQTIM(WS 1 Resource, Waiting for Setup)/ Tot
   Setup at WS 1        FRQTIM(WS 1 Resource, Setup)/Tot
   Process at WS 1      FRQTIM(WS 1 Resource, Processing)/Tot
   Blocked at WS 1      FRQTIM(WS 1 Resource, Blocked)/Tot
   Process at WS 2      FRQTIM(WS 2A Resource, Processing)/Tot +
                        FRQTIM(WS 2B Resource, Processing)/Tot
   Blocked at WS 2      FRQTIM(WS 2A Resource, Blocked)/Tot +
                        FRQTIM(WS 2B Resource, Blocked)/Tot
   Wait for Setup at WS 3     FRQTIM(WS 3 Resource, Waiting for Setup)/ Tot
   Setup at WS 3        FRQTIM(WS 3 Resource, Setup)/Tot
   Process at WS 3      FRQTIM(WS 3 Resource, Processing)/Tot
```

You should note that there could be two parts being processed or blocked at Workstation 2. Thus, we have included terms for both the 2A and 2B resources at Workstation 2.

Now that we have developed a method, and the expressions, to calculate the average percent of time our parts spend in each state, we will use the **Statistic** data module to add this information to our summary report. The new statistics that were added to the **Statistic** data module are shown in *Display 10.21*.



6	Travel to WS 1	Output	TAVG(Delivery Time)*TNUM(Delivery Time)/Tot	Travel to WS 1
7	Wait for WS 1	Output	TAVG(Seize WS 1.Queue.WaitingTime)*TNUM(Seize WS 1.Queue.WaitingTime)/Tot	Wait for WS 1
8	Wait for WS 2	Output	TAVG(Seize WS 2.Queue.WaitingTime)*TNUM(Seize WS 2.Queue.WaitingTime)/Tot	Wait for WS 2
9	Wait for Setup at WS 1	Output	FRQTIM(WS 1 Resource,Waiting for Setup)/Tot	Wait for Setup at WS 1
10	Setup at WS 1	Output	FRQTIM(WS 1 Resource,Setup)/Tot	Setup at WS 1
11	Process at WS 1	Output	FRQTIM(WS 1 Resource,Processing)/Tot	Process at WS 1
12	Blocked at WS 1	Output	FRQTIM(WS 1 Resource,Blocked)/Tot	Blocked at WS 1
13	Process at WS 2	Output	FRQTIM(WS 2A Resource,Processing)/Tot + FRQTIM(WS 2B Resource,Processing)/Tot	Process at WS 2
14	Blocked at WS 2	Output	FRQTIM(WS 2A Resource,Blocked)/Tot + FRQTIM(WS 2B Resource,Blocked)/Tot	Blocked at WS 2
15	Wait for Setup at WS 3	Output	FRQTIM(WS 3 Resource,Waiting for Setup)/Tot	Wait for Setup at WS 3
16	Setup at WS 3	Output	FRQTIM(WS 3 Resource,Setup)/Tot	Setup at WS 3
17	Process at WS 3	Output	FRQTIM(WS 3 Resource,Processing)/Tot	Process at WS 3

Display 10.21. The Summary Statistics for Part States

Before we proceed, let's address the discrepancy between the number of observations for our Delivery Time and Cycle Time statistics on our summary report. Because of the methods we use to collect our statistics, and the fact that we start our simulation with the system empty and idle, this discrepancy will always exist. There are several ways to deal with it. One method would be to terminate the arrival of parts to the system and let all parts complete processing before we terminate the simulation run. Although this would yield statistics on an identical set of parts, in effect we're adding a shutdown period to our simulation. This would mean that we would have potential transient conditions at both the start and end of the simulation, which would affect the results of our resource statistics.

We could increase our run length until the relative difference between these observations would become very small, thereby reducing the effect on our results. Although this could easily be done for this small textbook problem, it could result in unacceptably long run times for a larger problem. If we wanted to be assured that all part-state statistics were based on the same set of parts, we could collect the times each part spends in each activity and store these times in entity attributes. When the part exits the system, we could then Tally all these times. Although this is possible, it would require that we make substantial changes to our model, and we would expect that the increased accuracy would not justify these changes.

If you step back for a moment, you should realize that the problem of having summary statistics based on different entity activities is not unique to this problem. It exists for almost every steady-state simulation that you might construct. So we recommend that in this case you take the same approach that we use for the analysis of steady-state simulations. Thus, we would add a Warm-up Period to eliminate the start-up conditions. Because our system is tightly coupled, it will not accumulate large queues, and the number of parts in the system will tend to remain about the same. Because the warehouse delay is modeled as exponential, it is possible that there could be no parts in the system, although this is highly unlikely. At the other extreme, there can be a maximum of only eight parts in the system. Thus, by adding a warm up to our model, we will start collecting statistics when the system is already in operation. This will reduce the difference between the number of observations for our Tallies. For now, let's just assume a Warm-up Period of 500 minutes and edit the *Run>Setup>Replication Parameters* dialog box to include that entry. If this were a much larger system that could accumulate large queues, you might want to reconsider this decision.

If you now run the model, the information shown in *Figure 10.12* will be added to the summary report.

Output	Value
Blocked at WS 1	0.3881
Blocked at WS 2	5.4040
Process at WS 1	6.6783
Process at WS 2	18.7632
Process at WS 3	5.4595
Setup at WS 1	4.0051
Setup at WS 3	4.0136
Travel to WS 1	5.5551
Wait for Setup at WS 1	1.5395
Wait for Setup at WS 3	1.3023
Wait for WS 1	36.5635
Wait for WS 2	10.3425

Figure 10.12. The Appended Summary Report

10.6 Few Miscellaneous Modeling Issues

Our intention in writing this tome was not to attempt to cover all the functions available in the Arena simulation system. There are still a few modules in the Basic Process, Advanced Process, and Advanced Transfer panels that we have not discussed. And there are many more in the Blocks and Elements panels that we have neglected entirely However, in your spare time, we encourage you to attach the panels that you seldom use and place some modules. Using the Help features will give you a good idea of what we have omitted. In most cases, we suspect that you'll never need these additional feature. Before we close this chapter, we'd like to point out and briefly discuss a few features we have not covered. We don't feel that these topics require an in-depth understanding only an awareness.

10.6.1 Guided Transporters

There is an entire set of features designed for use with guided transporters. These features are useful not only for modeling automated guided vehicle (AGV systems, but also for warehousing systems and material-handling systems that use the concept of a moving cart, tote, jig, fixture, etc. Interestingly, they're also great for representing many amusement-park rides. Because this topic would easily fill an additional chapter or two, we've chosen not to present it in this book. However, you can find a complete discussion of these features in Chapter 9 of Pegden, Shannon, and Sadowski (1995).

10.6.2 Parallel Queues

There are also two very specialized modules from the Blocks panel, **QPick** and **PickQ**, that are seldom used; but if you need them, they can make your modeling task much easier. The **QPick** module can be used to represent a process where you want to pick the next entity for processing or movement from two or more different queues based on some decision rule. Basically, the **QPick** module would sit between a set of detached **Queue** modules and a module that allocates some type of scarce resource (for example, Allocate, Request, Access, or **Seize** modules). Let's say that you have three different streams of entities converging at a point where they attempt to seize the same resource. Furthermore, assume that you want to keep the entities from each stream separate from each other. The modules required for this part of your model are shown in *Figure 10.13*. Each entity stream would end by sending the entity to its Detached queue (more on s Detached queue shortly). The link between the QPick and **Queue** modules is by module Labels. When the resource becomes available, it will basically ask the **QPick** module to ' determine from which queue to select the next entity to be allocated the resource. Also note that you must use the **Seize** module from the Blocks panel for this to work, not the **Seize** module from the Advanced Process panel. When using modules from the Blocks panel, Arena does not automatically define queues, resources, etc. Thus, you may have to place the corresponding modules from the Basic Process or Elements panels to define these objects.

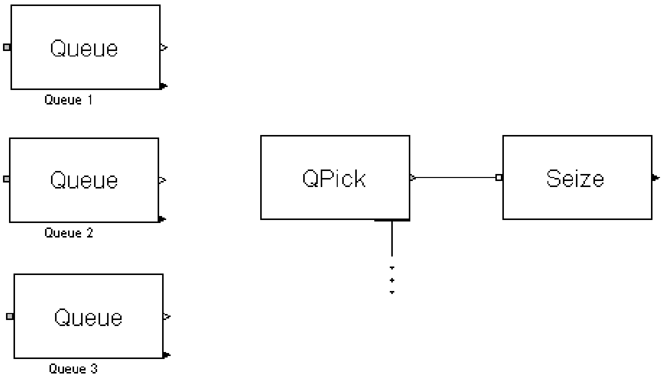


Figure 10.13. Using a **QPick** module

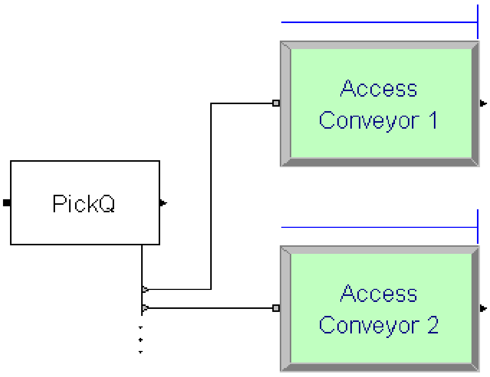


Figure 10.14. Using a **PickQ** module

The **PickQ** module can be used to represent a process where you have a single arrival stream, and you will use some decision rule to pick between two or more queues in which to place the entity. Let's assume that you have a stream of arriving entities that are to be loaded onto one of two available conveyors. The modules required for this part of your model are shown in *Figure 10.14* (the **Access** modules are from the Advanced Transfer panel). Note that you can't specify internal queues for the **Access** modules, and the decision as to which conveyor the entity is directed to is based on characteristics of the queue, not the conveyors. The **PickQ** module can be used to direct entities to **Access, Seize, Allocate, Request**, or **Queue** modules, or any module that is preceded by a queue.

Now let's address the issue of a Detached queue. If you use the **Queue** module from the Blocks panel, you're given the option of defining the queue as being Detached. This means that the queue is not directly linked to a downstream module. Such a queue can be indirectly linked, as was the case with the **QPick** module, or there might be no obvious link. For example, you may want to use a set of entities whose attribute values hold information that you might want to access and change over the course of a simulation. If this is all you want to do, it might be easier to use a Variable defined as a matrix, or perhaps an external database. The advantage of using a queue is that you can also change the ranking of the entities in the queue. You can access or change any entity attribute values using Arena variables. You can also use the **Search, Copy, Insert, Pickup**, and **Remove** modules from the Blocks panel (or the **Search, Remove, Pickup** and **Dropoff** modules from the Advanced Process panel) to interact with the entities in the queue. Note that if queue ranking is important, Arena only ranks an entity that enters the queue. Thus if you change an entity attribute that is used to rank the queue, you must remove the entity from the queue and then place it back into the queue.

10.6.3 Decision Logic

There are situations where you may require complex decision logic based on current system conditions. Arena provides several modules in the Blocks panel that might prove useful. The first is a set for the development of if then-else logic. The **If, ElseIf, Else**, and **EndIf** modules can be used to develop such logic. We will not explain these modules in detail here, but we encourage you to use Help if you implement logic with these modules. Although these modules will allow you to develop powerful logic, you need to be very careful to ensure that your logic is working correctly. These modules are designed to work only when all of the modules between an **If** and its matching **EndIf** (including any **ElseIf** or **Else** modules inside) are graphically connected in Arena. This allows you to use many of Arena's modules inside **If**/**EndIf** logic, such as **Assign**, **Seize**, and **Delay**, but precludes use of those that don't permit graphical connections, such as **Route**, **Convey**, and **Transport**. There is also a set of modules to implement do-while logic: the **While** and **EndWhile** modules. The same warnings given previously also apply for these modules.

If you really need to implement this type of logic, there are several options. The easiest is to use the Label and Next Label options to connect your modules rather than direct graphical connections. Although this works, it doesn't show the flow of logic. An alternative is to write your logic as an external SIMAN .mod file and use the **Include** module from the Blocks panel to include this logic in your module. Unfortunately, this option is not available for use with the academic version software. The safest and most frequently used option is to use a combination of **Decide** and **Branch** modules to implement your logic. This always works, although the logic may not be very elegant.

REFERENCES

1. **Altiok, T.- Melamed B.:** Simulation Modeling and Analysis with Arena. Academic Press is an imprint of Elsevier, ISBN 13: 978-0-12-370523-5, 2007.

2. Arena Standard Edition User's Guide. Rockwell Software.

3. Arena Professional Edition Reference Guide. Rockwell Software.

4. **Banks, J. and J. S. Carson**: Discrete-Event System Simulation, Prentice-Hall, 1984. 2. Conway, R. W., B. M. Johnson, and W. L. Maxwell, "Some Problems of Digital Systems Simulation," Management Science, Vol. 6, 1959, pp. 92-110.

5**. Barnes, R. M.:** Motion and Time Study: Design and Measurement of Work, Sixth Edition, John Wiley, 1968.

6. **Bartee, E. M.:** Engineering Experimental Design Fundamentals, Prentice-Hall, 1968.

7. **Bratley, P., B. L. Fox**, and **L. E. Schrage:** A Guide to Simulation, Springer-Verlag, 1983.

8. **Bratley, P.-Fox B. L.-Schrage L. E.:** A Guide to Simulation, 2d ed., Springer-Verlag, New York, NY., 1987.

9. **Benkő, J.:** Logisztikai tervezés. (mezőgazdasági alkalmazásokkal) Dinasztia Kiadó, Budapest, 2000.

10. **Benkő, J.:** A termény betakarítás és szállítás modellezése az Arena szimulátorral. Logisztikai évkönyv 2006 (Szerk.: Szegedi Z.), Magyar Logisztikai Egyesület, Budapest, 2006. 125-133 p.

11. **Benkő, J.:** Logisztika I. (A felsőfokú logisztikai tanfolyam tankönyve.), LOKA, Gödöllő, 2009.

12. **Benkő, J**.: Anyagmozgatási műveletek modellezése szimulációval Arena környezetben. Logisztikai évkönyv 2010. (Szerk: Szegedi Z.), MLE, Budapest 2010. 99-105 p., ISSN 1218-3849

13. **Benkő, J**.: Ellátási láncok modellezése szimulációval. Logisztikai évkönyv 2011. (Szerk: Bokor Z.), MLE, Budapest 2011. 13-18 p., ISSN 1218-3849

14. **Benkő, J.:** Kanban-rendszerű gyártás modellezése szimulációval. Magyar Minőség, XIX. évfolyam, 3. szám, 2010. március, ISSN 1789-5502 (CD-ROM), 16-31 p.

15. **Benkő, J.:** Modeling Supply Chain with Simulation. I. Central European Conference on Logistics, University of Miskolc 26. november 2010.

16. **Devroye, L.:** Non-Uniform Random Variate Generation, Springer-Verlag, New York, NY., 1986.

17. **Devore, J. L.:** Probability and Statistics for Engineering and the Sciences, 6th ed., Wadsworth Inc, Belmont, CA., 2003.

18. **Diananda, P. H.:** "Some Probability Limit Theorems with Statistical Applications." Proceedings, Cambridge Phil. Soc., Vol. 49, 1953, pp. 239-246.

19. **Emshoff, J. R. and R. L. Sisson**: Design and Use of Computer Simulation Models Macmillan, 1970.

20. **Feller, W.:** An Introduction to Probability Theory and Its Applications, John Wiley, Vol. II, 1972.

21. **Fishman, G. S. and P. J. Kiviat**: "Analysis of Simulated Generated Time Series," Management Science, Vol. 13, 1967, pp. 525-557.

22. **Fishman, G. S.:** Principles of Discrete Event Simulation, John Wiley, 1978.

23. **Giffin, W.**: Transform Techniques for Probability Modeling, Academic Press, 1975.

24. **Hogg, R. V. and A. T. Craig:** Introduction to Mathematical Statistics, Macmillan, 1970.

25. **Kelton, W. D.-Sadowski, R. P.-Sturrock, D. T.:** Simulation with Arena. McGraw Hill Higher Education, International Edition, ISBN 0-07-121933-1, 2004.

26. **Law, A. M.- Kelton, W. D.:** Simulation Modeling and Analysis, 3d ed., McGraw-Hill, New York, NY., 2000.

27. **Meszéna Gy.-Ziermann M.:** Valószínűségelmélet és matematikai statisztika. Közgazdasági és Jogi Könyvkiadó, Budapest, 1981.

28. **Mihram, G. A.:** Simulation: Statistical Foundations and Methodology, Academic Press, 1972.

29. **Papoulis, A.:** Probability, Random Variables, and Stochastic Processes, McGraw-Hill, 1965.

30. **Prezenszki, J.** (Ed.): Logisztika. (Bevezető fejezetek.) BME, Mérnöktovábbképző Intézet, Budapest, 1995.

31. **Prezenszki, J.**(Ed.): Logisztika II. (Módszerek, eljárások.), LFK, Budapest, 1999.

32. **Pritsker, A. A. B.:** The GASP IV Simulation Language, John Wiley, 1974.

33. **Pritsker, A. A. B.:** Modeling and Analysis Using Q-GERT Networks, Halsted Press and Pritsker & Associates, Inc., 1977.

34. **Pritsker, A. A. B.:** Introduction to Simulation and SLAM II. A Halsted Press Book, John Wiley & Sons, New York, 1986.

35. **Pritsker, A. A. B.:** "Compilation of Definitions of Simulation," SIMULATION, Vol. 33, 1979, pp. 61-63.

36. **Pritsker, A. A. B.:** "Models Yield Keys to Productivity Problems, Solutions," Industrial Engineering, October, 1983, pp. 83-87.

37. **Shannon, R. E.:** Systems Simulation: The Art and Science, Prentice-Hall, 1975.

38. **Szegedi Z.- Prezenszki J.:** Logisztika menedzsment. Kossuth Kiadó, Budapest, 2003.

39. **Van Horn, R. L.:** "Validation of Simulation Results," Management Science, Vol. 17, 1971, pp. 247-258.

40. **Wilson, J. R.:** Statistical Techniques for Simulation Practitioners. Course Notes, Pritsker & Associates, 1985.