Getting Started With AutoMod

By Jerry Banks

AutoSimulations, Inc.

655 Medical Drive
Bountiful, Utah 84010
(801) 298-1398

August 2000

Printed in the United States of America.

Contents

Chapter 1 Principles of Simulation

Chapter 2 Using the Software

Chapter 3 AutoMod Concepts

Chapter 4 Introduction to AutoMod Syntax

Chapter 5 Process System Basics

Chapter 7 Advanced Process System Features

Chapter 8 Basic Statistical Analysis Using AutoStat

Chapter 9 Modeling Complex Conveyor Systems

Chapter 10 Intermediate Statistical Analysis

Chapter 11 Introduction to Path Mover Systems

Chapter 12 Modeling Complex Material Handling Systems

Chapter 13 Indefinite Delays

Chapter 14 Additional Features

Chapter 15 Warmup Analysis Using AutoStat

Chapter 1

Principles of Simulation

Chapter 1

Principles of Simulation*

The purpose of this textbook is to provide an introduction to the AutoMod™ simulation software package. Prior to learning about the AutoMod program, it is useful to understand some of the basic concepts of simulation, which this chapter provides. If you have previously studied simulation, reading this chapter provides a useful review and update.

Chapter 1 is organized in three parts. The first part begins with a definition and an example of simulation, an introduction to modeling concepts, and four simulation modeling methods. The second part of the chapter discusses subjective topics, including the advantages and disadvantages of simulation, areas of application of AutoMod, and the steps in the simulation process. The third part of the chapter introduces issues related to random number and random variate generation, input data, model verification and validation, output interpretation, and the analysis of waiting lines.

Definition of simulation

Simulation is the imitation of a real-world process or system over time. Simulation involves the generation of an artificial history of the system and the observation of that artificial history to draw inferences concerning the operating characteristics of the real system being represented.

Simulation is an indispensable problem-solving methodology for the solution of many real-world problems. Simulation is used to describe and analyze the behavior of a system, ask “what if” questions about the real system, and aid in the design of real systems. Both existing and conceptual systems can be modeled with simulation.

* Source: *Handbook of Simulation*, ed. Jerry Banks, pp. 3–30. Copyright © 1998. Adapted by permission of John Wiley & Sons, Inc.

Example 1.1: Ad hoc simulation

Consider the operation of a tool crib where mechanics arrive between 1 and 10 minutes apart in time (integer values only, with each value equally likely; known as a discrete uniform distribution). A tool crib attendant serves the mechanics in a time between 1 and 6 minutes (also integer values and equally likely). Restricting the times to integer values is an abstraction of reality, because time is continuous, but the restriction helps present the example.

The objective is to manually simulate the tool crib operation until 20 mechanics are served. You will then compute measures of performance, such as the percentage of idle time of the attendant, the average waiting time per mechanic, and so on. Admittedly, 20 mechanics is far too few to draw conclusions about the operation of a real system's long-term behavior, but by following this example, the stage is set for further discussion in this chapter and subsequent chapters about using a computer for performing simulation.

Setting up an ad hoc simulation table

The following ad hoc simulation table manually simulates the tool crib operation. All times shown are in minutes.

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
Mechanic	**Time between arrivals**	**Arrival time**	**Length of service time**	**Time service begins**	**Time service ends**	**Time in system**	**Tool crib attendant idle time**	**Mechanic waiting time**
1	–	0	2	0	2	2	0	0
2	5	5	2	5	7	2	3	0
3	1	6	6	7	13	7	0	1
4	10	16	5	16	21	5	3	0
5	6	22	6	22	28	6	1	0
6	2	24	4	28	32	8	0	4
7	9	33	3	33	36	3	1	0
8	1	34	4	36	40	6	0	2
9	10	44	1	44	45	1	4	0
10	3	47	3	47	50	3	2	0
11	5	52	1	52	53	1	2	0
12	2	54	2	54	56	2	1	0
13	3	57	3	57	60	3	1	0
14	5	62	6	62	68	6	2	0
15	4	66	2	68	70	4	0	2
16	3	69	6	70	76	7	0	1
17	7	76	4	76	80	4	0	0
18	8	84	5	84	89	5	4	0
19	7	91	3	91	94	3	2	0
20	7	98	1	98	99	1	4	0
					Totals	79	30	10

Example 1.1 – Ad hoc simulation table (times in minutes)

The first column lists the 20 mechanics that arrive in the system.

To simulate the tool crib service process, random interarrival times need to be generated. Assume that the interarrival times are generated using a spinner that has possibilities for the values 1 through 10 (these values are recorded in column 2).

Assume that the first mechanic arrives at time 0 (indicated by an arrival time of “0”). The arrival time for each of the remaining mechanics is calculated by adding the time between arrivals to the previous mechanic’s arrival time. For example, because mechanic 1 is assumed to arrive at time 0, and there is a five-minute interarrival time for mechanic 2, mechanic 2 arrives at time 5. There is a one-minute interarrival time for mechanic 3, so the arrival occurs at time 6. By continuing this process, the arrival times of all 20 mechanics are determined (these values are recorded in column 3).

After determining the arrival times, the random service times are generated using a die that has possibilities for the values 1 through 6 (these values are recorded in column 4). Now, the simulation of the service process begins.

At time 0, mechanic 1 arrives and immediately begins service. The service time is two minutes, so the service period ends at time 2. The total time in the system for mechanic 1 is two minutes. The tool crib attendant is not idle, because the simulation begins with the arrival of the mechanic.

At time 5, mechanic 2 arrives and immediately begins service, as indicated in column 5. The service time is two minutes so the service period ends at time 7, as indicated in column 6. The tool crib attendant is idle from time 2 until time 5, so three minutes of idle time occur and mechanic 2 spends no time waiting for the attendant.

Mechanic 3 arrives at time 6, but service cannot begin until time 7 (mechanic 2 is serviced until time 7). The service time is six minutes, so service is completed at time 13. Mechanic 3 is in the system from time 6 until time 13, or for seven minutes, as indicated in column 7. Mechanic 3 must wait one minute for service to begin.

This process continues for all 20 mechanics. The totals for columns 7, 8, and 9 are shown.

Analyzing the simulation results

After completing the ad hoc simulation table, some performance measures can be calculated:

$$\text{Average time in system} = \frac{79}{20} = 3.95 \text{ minutes}$$

$\text{Percent idle time} = \left[\frac{30}{99}\right](100) = 30\%$, where 99 is the length of the simulation (based on the end of the last service time)

$$\text{Average waiting time per mechanic} = \frac{10}{20} = 0.5 \text{ minutes}$$

$$\text{Fraction of mechanics having to wait} = \frac{5}{20} = 0.25$$

$$\text{Average waiting time of those who waited} = \frac{10}{5} = 2 \text{ minutes}$$

This very limited simulation indicates that the system is functioning well. Only 25 percent of the mechanics had to wait. Some 30 percent of the time the tool crib attendant is idle. Whether a slower tool crib attendant should replace the current tool crib attendant depends on the cost of delaying the mechanics versus any savings from having a slower attendant.

This small simulation can be calculated manually, but there is a limit to the complexity of problems that can be solved in this manner. Also, the number of mechanics that must be simulated could be much larger than 20 and the number of times that the simulation must be run for statistical purposes could be large. Hence, using the computer to solve real simulation problems is almost always appropriate.

Example 1.1 raises several questions, including:

1. How is the form of the input data determined?
2. How are random variates generated if they follow statistical distributions other than the discrete uniform distribution?
3. How does the user know that the simulation imitates reality?
4. What other kinds of problems can be solved by simulation?
5. How long does the simulation need to run?
6. How many different simulation runs should be conducted?
7. What statistical techniques should be used to analyze the output?

Each of these questions raises a host of issues that many textbooks and thousands of technical papers have addressed. While this introductory chapter cannot treat all of these questions in detail, it will provide enough information to help you understand the framework of the AutoMod software. More information is available from Banks, Carson, Nelson, and Nicol (2000).

Modeling concepts

There are several concepts about simulation and the AutoMod software that you need to understand, including: models, events, system state variables, entities and attributes, resources, queues, activities, delays, and discrete-event simulation.

Additional information about these topics is available from Banks, Carson, Nelson, and Nicol (2000) and Law and Kelton (2000). The discussion in this section follows that of Carson (1993).

Models and events

A **model** is a representation of an actual system. A model should contain enough detail to answer the questions you are interested in, without containing more details than necessary.

Consider an **event** as an occurrence that changes the state of the system. In example 1.1, events include the arrival of a mechanic for service at the tool crib, the beginning of service for a mechanic, and the completion of service.

There are both internal and external events, also called **endogenous events** and **exogenous events**, respectively. For example, an endogenous event in example 1.1 is the beginning of service of the mechanic, because that occurrence is within the system being simulated. An exogenous event is the arrival of a mechanic for service, because that occurrence is from outside of the system being simulated. However, the arrival of a mechanic for service impinges on the system and must be taken into consideration.

This textbook focuses on AutoMod, which uses a discrete-event simulation model. A **discrete-event model** represents the components of a system and their interactions, as opposed to **mathematical models**, which consider systems through the use of formulas, such as the mathematical model from physics that $\text{Force} = \text{Mass} \times \text{Acceleration}$. Mathematical models, as well as **descriptive**, **statistical**, and **input-output** models, represent a system's inputs and outputs explicitly, but represent the internals of the model with mathematical or statistical relationships, in effect "solving" the model. Discrete-event models are time-based and are "run," not solved, with the results based on the interactions of system components.

Discrete-event models are **dynamic**, in that the passage of time plays a crucial role. Most mathematical and statistical models are **static** in that they represent a system at a fixed point in time. Consider the annual budget of a firm, which is kept in a spreadsheet. Changes can be made in the budget, and the spreadsheet can be recalculated, but the passage of time is usually not a critical issue.

System state variables

System state variables are the collection of all information needed to define what is happening within the system (based on the desired output) at a given point in time. For example, state variables may include time in system, percentage of idle time, and so on. The state information you are interested in may vary from model to model (or from different analyses of the same model), depending on what you are trying to learn from the simulation. Determining the system state variables is as much an art as a science. During the modeling process, it will become clear which system variables you may have forgotten about (and which ones you do not need that you thought you would).

The system state variables in a discrete-event model remain constant over intervals of time and change value only at certain well-defined points called **event times**.

In contrast, **continuous models** are models in which system state is represented by dependent variables that change continuously over time, as defined by differential or difference equations. For example, when modeling liquid-based systems, such as bottling facilities, the changes in the system are not a series of individual events, but a constant change over time.

Some models are mixed discrete-event and continuous. There are also continuous models that are treated as discrete-event models after some reinterpretation of system state variables, and vice versa. Although AutoMod is capable of modeling continuous systems, continuous systems are not discussed in this textbook. Instead, this textbook focuses on discrete-event simulations.

Entities and attributes

An **entity** represents an object that requires explicit definition. An entity can be dynamic in that it “moves” through the system (such as products or people), or it can be static in that it serves other entities (such as equipment or people). In example 1.1, the mechanic who arrives for service is a dynamic entity, whereas the tool crib attendant who performs the service is a static entity. In AutoMod, the dynamic entities that move through the system are called **loads**.

A load can have **attributes** that contain information about that load. All loads have the same user-defined attributes. For example, you may define load attributes such as a shift schedule and certifications when using loads to model people. When modeling products, you might define load attributes such as color, part type, time in system, and so on. Although each load has the same attributes, each load’s attribute might contain a different value. For example, assume all loads have an attribute called “A_color” that is used to track the color of the product. One load’s A_color attribute might contain the value “Red,” while another load’s A_color attribute might contain the value “Blue.”

Attributes of interest in one simulation may not be of interest in another simulation. For example, if red parts and blue parts are being manufactured, color might be an important attribute. However, if the time in the system for all parts is the issue, the color attribute may not be of importance.

Resources

A **resource** is an entity that provides service to dynamic entities, or loads. The tool crib attendant from the ad-hoc simulation is an example of a resource. Resources can serve one or more loads at the same time (that is, operate as a parallel server). A load can request one or more units of a resource. If denied, the requesting load waits for the resource or takes some other action (for example, diverts to another resource or leaves the system). If permitted to use the resource, the load remains for a time, then releases the resource.

There are many possible states of a resource. At a minimum, a resource has two states: idle and busy. You can also define states such as failed, blocked, or starved.

Queues

Queues in AutoMod are places where loads reside physically and graphically. Queues are used as places where loads wait (hence the name “queue”). Loads can wait in a queue for storage, while they are waiting for a resource, or while being processed by a resource.

In AutoMod, loads cannot be “in” or “on” a resource; they must be in a queue (or on a movement system, such as a conveyor or vehicle). To represent a machine, you generally use both a queue and a resource. In fact, often you use two queues: one queue to represent the line for the resource (the real-world queue) and one where the load resides physically while being processed by a resource (called a processing queue in this textbook). This textbook and its exercises usually use separate queues for the waiting line and for processing.

Activities and delays

An **activity** is a period of time with a duration that is known prior to commencement of the activity. Thus, when the duration begins, its end can be scheduled. The duration can be a constant value, a random value from a statistical distribution, the result of an equation, data read from a file, or can be calculated based on the event state. For example, a service time may be a constant 10 minutes for each load; it may be a random value from an exponential distribution with a mean of 10 minutes; it could be 0.9 times a constant value from clock time 0 to clock time 4 hours and 1.1 times the constant value after clock time 4 hours; or it could be 10 minutes when the preceding waiting line contains at most four loads, but 8 minutes when there are five or more in the preceding waiting line.

A **delay** is an indefinite duration that is caused by some combination of system conditions. When a load waits for a resource, the time that it remains in the waiting line may be unknown initially because that time may depend on other events, such as the arrival of a rush order that preempts the resource or a machine failure that requires repair before the resource can continue processing.

An AutoMod simulation contains activities that cause time to advance. A simulation also contains delays that cause loads to wait. The beginning and ending of an activity or delay is an event.

Discrete-event simulation models

Using the modeling concepts discussed so far, you can define a discrete-event simulation model as one in which the state variables change only at the discrete points in time at which events occur. Events occur as a consequence of activities and delays. Loads compete for system resources, possibly needing to wait for an available resource. Activities and delays “hold” loads for durations of time. A discrete-event simulation model is conducted over time (“run”) by a mechanism that moves simulated time forward.

Simulation modeling methods

There are four major simulation methods used by the simulation community:

- Process-interaction method
- Event-scheduling method
- Activity scanning method
- Three-phase method

The descriptions in this chapter are brief; readers requiring greater explanation are referred to Balci (1988) or Pidd (1998).

Process-interaction method

The simulation structure that has the greatest intuitive appeal is the **process-interaction method**. In this method, the computer program emulates the flow of an object (for example, a load) through the system. The load moves as far as possible in the system until it is delayed, enters an activity, or exits from the system. When the load's movement is halted, the clock advances to the time of the next movement of any load. This is the method that AutoMod uses.

This flow, or movement, describes in sequence all of the states that the object can attain in the system. For example, in a model of a self-service laundry, a customer may enter the system, wait for a washing machine to become available, wash his or her clothes in the washing machine, wait for a basket to become available, unload the washing machine, transport the clothes in the basket to a drier, wait for a drier to become available, unload the clothes into a drier, dry the clothes, unload the drier, and then leave the laundry. Each state and event is simulated.

Event scheduling method

The basic concept of the **event scheduling method** is to advance time to the moment when something happens next (that is, when one event ends, time is advanced to the time of the next scheduled event). An event usually releases a resource. The event then reallocates available objects or entities by scheduling activities in which they can now participate. For example, in the self-service laundry, if a customer's washing is finished and there is a basket available, the basket could be allocated immediately to the customer, who would then begin unloading the washing machine. Time is advanced to the next scheduled event (usually the end of an activity) and activities are examined to see whether any can now start as a consequence.

Activity scanning method

The third simulation modeling structure is **activity scanning**. Activity scanning is also known as the **two-phase approach**. Activity scanning produces a simulation program composed of independent modules waiting to be executed. In the first phase, a fixed amount of time is advanced, or scanned. In phase two, the system is updated (if an event occurs). Activity scanning is similar to rule-based programming (if the specified condition is met, then a rule is executed).

Three-phase method

The fourth simulation modeling structure is known as the **three-phase method**. In the first phase, time is advanced until there is a state change in the system or until something happens next. The system is examined to determine all of the events that take place at this time (that is, all the activity completions that occur). The second phase is the release of those resources scheduled to end their activities at this time. The third phase is to start activities, given a global picture of resource availability.

Possible modeling inaccuracies may occur with the activity scanning and three phase modeling methods, because discrete time slices must be specified. If the time interval is too wide, detail is lost. This type of simulation will become less popular as computing power continues to increase and computing costs continue to decrease.

Advantages and disadvantages of simulation*

Competition in the computer industry has led to technological breakthroughs that are allowing hardware companies to continually produce better products. It seems that every week another company announces its latest release, each with more options, memory, graphics capability, and power.

What is unique about new developments in the computer industry is that they often act as a springboard for related industries. One industry in particular is the simulation software industry. As computer hardware becomes more powerful, faster, and easier to use, simulation software does, too.

The number of businesses using simulation is rapidly increasing. Many managers are realizing the benefits of using simulation for more than just a one-time remodeling of a facility. Now, because of advances in software, managers are incorporating simulation in their daily operations on an increasingly regular basis.

Advantages

For most companies, the benefits of using simulation go beyond just providing a look into the future. These advantages are mentioned by many authors (Banks, Carson, Nelson, and Nicol (2000); Law and Kelton (2000); and Schriber (1991)), and include the following:

1. **Making correct choices** – Simulation lets you test every aspect of a proposed change or addition without committing resources to their acquisition. This is critical, because once the bricks have been laid or the material-handling systems have been installed, changes and corrections can be extremely expensive. Simulation allows you to test your designs without acquiring resources.
2. **Compressing and expanding time** – By compressing or expanding time, simulation allows you to speed up or slow down phenomena so that you can thoroughly investigate them. You can examine an entire shift in a matter of minutes if you desire, or you can spend two hours examining all the events that occurred during one minute of simulated activity.
3. **Understanding “Why?”** – Managers often want to know why certain phenomena occur in a real system. With simulation, you determine the answer to the “why” questions by reconstructing the scene and conducting a microscopic examination of the system. You cannot accomplish this with a real system because you cannot see or control it in its entirety.

* Source: Banks, J. and V. Norman (1995), “Justifying Simulation in Today’s Manufacturing Environment,” *IIE Solutions*, November.

4. **Exploring possibilities** – One of the greatest advantages of using simulation is that once you have developed a valid simulation model, you can explore new policies, operating procedures, or methods without the expense and disruption of experimenting with the real system. Modifications are incorporated in the model, and you observe the effects of those changes on the computer rather than on the real system.

5. **Diagnosing problems** – The modern factory floor or service organization is very complex—so complex that it is impossible to consider all the interactions taking place in a given moment. Simulation allows you to better understand the interactions among the variables that make up such complex systems. Diagnosing problems and gaining insight into the importance of these variables increases your understanding of their effects on the performance of the overall system.

6. **Identifying constraints** – Production bottlenecks give manufacturers headaches. It is easy to forget that bottlenecks are an effect rather than a cause. However, by using simulation to perform bottleneck analysis, you can discover the cause of delays in work in process, information, materials, or other processes.

7. **Developing understanding** – In many cases, designs are based on someone’s thoughts about the way the system operates rather than being based on analysis. Simulation studies help you design a system as it should operate, rather than how someone supposes it will.

8. **Visualizing the plan** – You can take your designs beyond CAD layouts using the animation features offered in many simulation packages. Animation allows you to see your facility or organization running, often from various angles and levels of magnification. In AutoMod, you can view your model in 3-D. Animation allows you to detect design flaws within systems that appear credible when seen on paper or in a 2-D CAD drawing.

9. **Building consensus** – Simulation provides an objective basis for decision-making. It is easier to approve or disapprove designs because you can simply select the designs and modifications that provided the most desirable results, whether it be increasing production or reducing the waiting time for service.

10. **Preparing for change** – We all know that the future will bring change. Answering “what-if” questions is useful for both designing new systems and redesigning existing systems. During the problem-formulation stage of a simulation study, you should discuss what scenarios are needed by everyone involved with the project so that you build the model and perform the study to be sure that it answers the correct questions. What if an automated guided vehicle (AGV) is removed from service for an extended period of time? What if demand for service increases by 10 percent? What if....? The options are unlimited.

11. **Making wise investments** – The typical cost of a simulation study is substantially less than one percent of the total amount being expended for the implementation of a design or redesign. Because the cost of a change or modification to a system after installation is so great, simulation is a wise investment.

12. **Training the team** – Simulation models can provide excellent training when designed for that purpose. Used in this manner, the team provides decision inputs to the simulation model as it progresses. The team, and individual members of the team, can learn from their mistakes, and learn to operate better. This is much less expensive and less disruptive than on-the-job learning.

13. **Specifying requirements** – Simulation can be used to specify requirements for a system design. For example, you may not know the specifications for a particular type of machine to achieve a desired goal in a complex system. By simulating different capabilities for the machine, the requirements can be established.

Disadvantages

The disadvantages of simulation can include:

1. **Model building requires special training** – Model building is an art that is learned over time and through experience. Furthermore, if two models of the same system are constructed by two competent individuals, they may have similarities, but it is highly unlikely that they will be identical.
2. **Simulation results may be difficult to interpret** – Because most simulation outputs are essentially random variables (they are usually based on random inputs), it may be hard to determine whether an observation is a result of system interrelationships or randomness.
3. **Simulation modeling and analysis can be time-consuming and expensive** – Skimping on resources for modeling and analysis may result in a simulation model and/or analysis that is not sufficient to the task.
4. **Simulation may be used inappropriately** – Simulation is used in some cases when an analytical solution is possible, or even preferable. This is particularly true in the case of small queuing systems and some probabilistic inventory systems, for which closed-form models (equations) are available. For examples of some queuing equations, see “Queueing theory” on page 1.30.

However, these four disadvantages can be offset as follows:

1. **Simulators make model building easier** – AutoSimulations has developed packages that contain models that only need input data for their operation. Such models have the generic tag “simulators,” templates, or run-time models.
2. **Statistical analysis tools make analyzing output easier** – the AutoStat™ statistical analysis software, which works with AutoMod, has output-analysis capabilities for performing very extensive analysis. Using AutoStat reduces the computational requirements on the part of the user, although the user must still understand the analysis procedure.
3. **Simulation is getting faster and faster** – Simulation can be performed faster today than yesterday and will be even faster tomorrow. Some speed improvements come from the advances in hardware that permit rapid running of scenarios. Other speed improvements come from simulation packages becoming easier to use. For example, AutoMod contains templates for modeling material handling systems such as conveyors, path movers (automated guided vehicles fork trucks, people, and so on), overhead cranes, power-and-free systems, kinematics, and tanks and pipes. The less work the simulation engineer must do, the faster the project can be completed.
4. **Limitations of closed-form models** – Although closed-form models are useful for small queuing and inventory problems, most real-world problems are too complex to be solved with these approaches. Simulation is necessary when there are a large number of events and interactions in a system, which is true of most manufacturing problems.

Applications of AutoMod

The AutoMod software can be used in almost any area of manufacturing and material handling. AutoMod has been used widely in the following applications, categorized by industry:

Automated material handling systems (AMHS)

- Optimizing existing material handling system equipment
- Designing new material handling layouts
- Modeling parcel and letter handling
- Evaluating mining automation

Automotive

- Simulating body and paint shops
- Analyzing repair lines in paint shops
- Modeling sortation systems in engine plants
- Resequencing vehicles
- Modeling operator shifts

Warehousing/distribution centers

- Simulating sorting strategies in distribution centers
- Determining warehouse layouts and performing operations modeling
- Consolidating multiple distribution centers

Airports

- Modeling airport baggage systems
- Modeling air cargo handling

Semiconductor

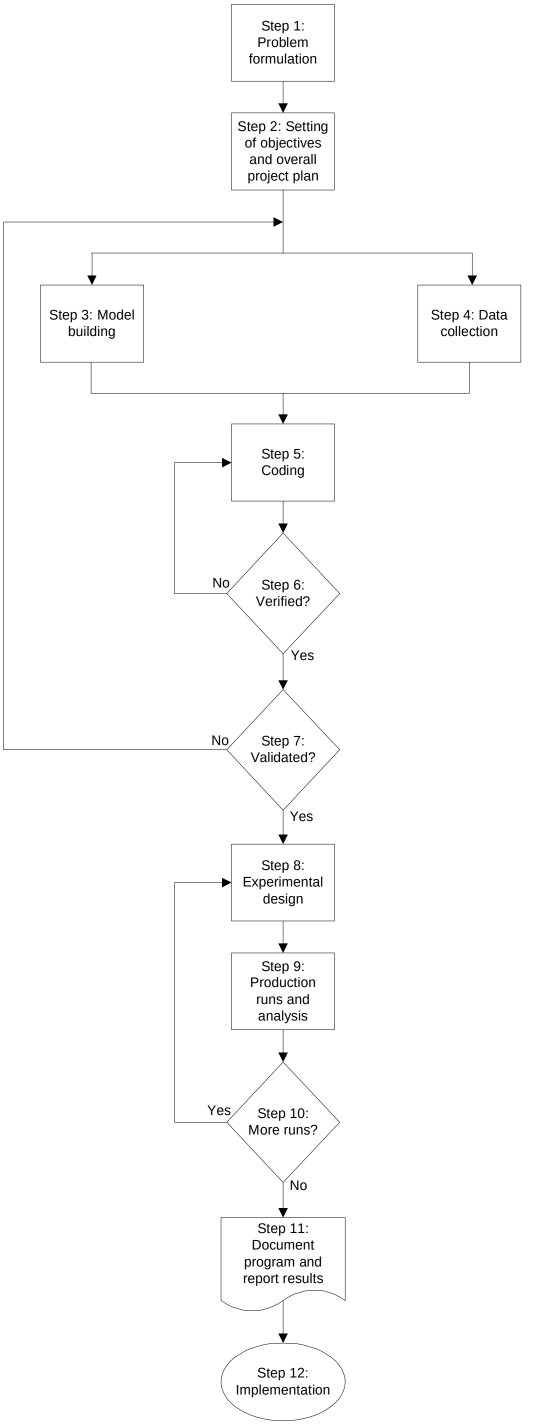
- Evaluating complex control logic for lot delivery
- Simulating 200mm and 300mm automated material handling systems
- Modeling cluster tool robots

Other

- Modeling fuselage assembly (aerospace)
- Modeling steel meltshops
- Performing buffer and downtime analysis
- Modeling customer service centers (home entertainment products, fast food stores)

Steps in a simulation study

The flowchart below shows a set of steps to guide a model builder in a thorough and sound simulation study. Similar illustrations and their interpretation can be found in other sources such as Law and Kelton (2000). This presentation is built on that of Banks, Carson, Nelson, and Nicol (2000).


Step 1: Problem formulation
Step 2: Setting of objectives and overall project plan
Step 3: Model building
Step 4: Data collection
Step 5: Coding
Step 6: Verified?
No
Yes
Step 7: Validated?
No
Yes
Step 8: Experimental design
Step 9: Production runs and analysis
Step 10: More runs?
Yes
No
Step 11: Document program and report results
Step 12: Implementation

Steps in a simulation study[*]

[*] Source: *Discrete Event System Simulation*, 3rd ed., by Banks, Carson, Nelson, and Nicol p. 16. Copyright © 2000. Reprinted by permission of Prentice-Hall, Upper Saddle River, New Jersey.

Step 1: Problem formulation. Every simulation study begins with a statement of the problem. If the statement is provided by those who have the problem (client), the simulation analyst must take extreme care to ensure that the problem is clearly understood. If a problem statement is prepared by the simulation analyst, it is important that the client understand and agree with its formulation. It is suggested that a set of assumptions be prepared by the simulation analyst and agreed to by the client. Even with all of these precautions, it is possible that the problem will need to be reformulated as the simulation study progresses.

Step 2: Setting of objectives and overall project plan. Another way to state this step is “prepare a proposal.” This step should be accomplished regardless of whether the analyst and client work for the same company or different companies. The objectives are the questions to be answered by the simulation study. The project plan should include a statement of the various scenarios that will be investigated. The plans for the study should be indicated in terms of time that will be required, personnel that will be used, hardware and software requirements (if the client wants to run the model and conduct the analyses), stages in the investigation, output at each stage, cost of the study, and billing procedures, if any.

Step 3: Model building. The real-world system under investigation is abstracted by a conceptual model, which is a series of mathematical and logical relationships concerning the components and the structure of the system. It is recommended that modeling begin simply and that the model grow until a model of appropriate complexity has been developed using AutoMod. First, model the material handling system(s). Then, add the basic process system. Next, add the resource cycles (maintenance, breakdowns, and shift schedules). Finally, add special features. It is not necessary to construct an unduly complex model. This will add to the cost of the study and the time for its completion without increasing the quality of the output. The client should be involved throughout the model construction process. This will enhance the quality of the resulting model and increase the client’s confidence in its use.

Step 4: Data collection. Shortly after the proposal is accepted, a schedule of data requirements should be submitted to the client. In the best of circumstances, the client has been collecting the necessary data, in the required format, and can submit this data to the simulation analyst electronically.

However, sometimes the delivered data is quite different than was anticipated. For example, in the simulation of an airline reservation system, the simulation analyst was told “we have every bit of data that you want over the last five years.” When the study commenced, the data delivered was the *average* “talk time” of the reservationist for each of the years. However, individual values were needed, not summary measures.

Model building and data collection are shown as contemporaneous in the flowchart (see “Steps in a simulation study” on page 1.14). This indicates that the simulation analyst can readily construct the model while the data collection is progressing.

Step 5: Coding. The conceptual model constructed in step 3 is written in a computer-recognizable form (that is, an operational model). In AutoMod, you write the model logic using the AutoMod language.

Step 6: Verified? Verification refers to the process of determining whether the operational model is performing as designed. Even for the models in the exercises in this textbook, it is possible to have verification difficulties. These models are orders of magnitude smaller than real models (the examples use about 50 lines of logic versus approximately 2,000 lines of logic for some real-world applications).

It is highly advisable that verification take place as a continuing process throughout the model-building process, rather than waiting until the model is complete. Verification is extremely important and is discussed further in this chapter (see “Verification and validation” on page 1.21).

Step 7: Validated? Validation is the determination that the conceptual model is an accurate representation of the real system. Can the model be substituted for the real system for the purposes of experimentation? If there is an existing system (called the base system), then an ideal way to validate the model is to compare the model's output to that of the base system. Unfortunately, there is not always a base system (such as when designing a new system). There are many methods for performing validation, and some of these are discussed further in this chapter (see "Verification and validation" on page 1.21).

Step 8: Experimental design. For each scenario that is to be simulated, decisions need to be made concerning the length of the simulation run, the number of runs necessary, and the manner of initialization, as required. AutoStat can help with this determination, and is discussed in detail in later chapters of this textbook.

Step 9: Production runs and analysis. Production runs, and their subsequent analysis, are used to estimate measures of performance for the scenarios that are being simulated. Again, AutoStat can be of tremendous help with this determination.

Step 10: More runs? Based on the analysis of runs that have been completed, the simulation analyst determines whether additional runs are needed and whether any additional scenarios need to be simulated. AutoStat can be of tremendous help with this determination.

Step 11: Document program and report results. Documentation is necessary for numerous reasons. If the simulation model is going to be used again, it may be necessary to understand how the simulation model operates. This will engender confidence in the simulation model so that the client can make decisions based on the analysis. Also, modifying a model is much easier with adequate documentation. One experience with an inadequately documented model is usually enough to convince a simulation analyst of the necessity of this important step.

The result of all the analysis should be reported clearly and concisely. This will enable the client to review the final formulation, the alternatives that were addressed, the criteria by which the alternative systems were compared, the results of the experiments, and analyst recommendations, if any.

Step 12: Implementation. The simulation analyst acts as a reporter rather than an advocate. The report prepared in step 11 stands on its merits and is provided as additional information that the client uses to make a decision. If the client has been involved throughout the study and the simulation analyst has followed all of the steps rigorously, it is likely that the implementation will be successful.

Random number and random variate generation

In a simulation model, there are many random events, including interarrival times, batch sizes, processing times, repair times, time until failure, and many others. To generate random numbers, example 1.1 used input values that were found using a spinner and a die. Almost all simulation models are constructed using a computer, so spinners and dice are not necessary. Instead, the computer generates independent random numbers that are distributed continuously and uniformly between 0 and 1 (represented by the notation U(0, 1)). These random numbers can then be converted to the desired statistical distribution, or **random variate**. The random variates are used to represent the random events in the model.

The importance of a good source of uniformly distributed random numbers is that all procedures for generating non-uniformly distributed random variates involve a mathematical transformation of uniform random numbers.

For example, the following formula converts uniformly distributed numbers into exponentially distributed random numbers. Suppose R_i is the ith random number generated from U(0, 1). Suppose further that the desired random variate is exponentially distributed with rate λ. These values can be generated from:

$$X_i = -\left(\frac{1}{\lambda}\right)\ln(1 - R_i)$$

Equation 1.1 - Random variate generator for the exponential distribution

where X_i is the ith random variate generated (for example, the time between the arrival of the ith and the (i + 1)st loads), and ln represents the natural logarithm. Suppose $\lambda = 1/10$ arrivals per minute. If $R_1 = 0.3067$:

$$X_i = -\left(\frac{1}{1/10}\right)\ln(1 - .3067) = 3.66$$

Note

This random variate generator uses the inverse-transform technique. Other distributions are generated using other techniques, such as convolution, acceptance-rejection, and composition. For more information about this subject, refer to Law and Kelton (2000).

AutoMod has two built-in random number generators that produce a sequence of random numbers. The first is the Tausworthe generator, which is not discussed in this textbook. The other is the **linear congruential generator (LCG)** that was documented by Knuth (1969). The LCG is defined by its parameters. The numbers generated by the LCG are actually “pseudorandom,” because they can be reproduced from the starting value so that model runs can be repeated and the results reproduced. The length of the sequence prior to repeating itself is very long (around two billion numbers on a 32-bit computer).

AutoMod also has the ability to generate a sample from an empirical distribution (a distribution of the raw input data) that is either discrete or continuous.

Input data

For each element in a system being modeled, the simulation analyst must decide upon a way to represent the associated random variables. The presentation of the subject that follows is based on Banks, Carson, and Goldsman (1998).

The techniques used may vary depending on:

- The amount of available data.
- Whether the data is real or just someone's best guess.
- Whether each variable is independent of other input random variables or whether it is related in some way to other input.

In the case of a variable that is independent of other variables, the choices are as follows:

- Assume that the variable is deterministic (constant).
- Fit a probability distribution to the data.
- Use the empirical distribution of the data.

These three choices are discussed in the next three subsections.

Assuming randomness away

Some simulation analysts may be tempted to assume that a variable is deterministic, and that the value can be obtained by averaging historic information. Or they may even guess at the value. However, if there is randomness in the model, this technique can invalidate the results.

Suppose that a machine manufactures parts in exactly 1.5 minutes. The machine requires a tool change according to an exponential distribution with a mean of 12 minutes between occurrences. The tool change time is also exponentially distributed with a mean of 3 minutes.

It would be very inaccurate to add up the mean processing and change times and use that value. For example, in this system, you could simplify and say that 12 minutes per hour are devoted to tool changes, leaving 48 minutes of each hour for manufacturing. Therefore there are 32 parts made per hour (48/1.5), with a total processing time per part of 1.875 minutes (60/32). These values are not accurate.

In reality, the exponentially distributed change times and change rates would cause the time per part to vary much more widely, affecting measures such as the average number of parts in the system or the time that parts spent waiting before the machine. Using an average or a guess can invalidate your simulation.

Fitting a distribution to data

If there are sufficient data points (100 or more), it may be appropriate to fit a probability distribution to the data using conventional methods. When there is a small amount of data, the tests for goodness-of-fit, such as the chi-squared test, offer little guidance in selecting one distribution form over another.

There are also underlying processes that give rise to distributions in a rather predictable manner. For example, if arrivals:

1. Occur one at a time, and
2. Are completely at random without rush or slack periods, and
3. Are completely independent of one another,

then it is a Poisson process, which means that the number of arrivals in a given time period follows a Poisson distribution and the time between arrivals follows an exponential distribution.

Several vendors provide software to perform input data analysis. However, if a goodness-of-fit test is being conducted without the aid of input data analysis software, the following three-step procedure is recommended:

Step 1 ***Hypothesize*** a candidate distribution. First, ***ascertain*** whether the underlying process is discrete or continuous. Discrete data arises from counting processes. Examples include the number of customers that arrive at a bank each hour, the number of tool changes in an eight-hour day, and so on. Continuous data arises from measurement (time, distance, weight, etc.). Examples include the time to produce each part, the time to failure of a machine, and so on.

Discrete distributions frequently used in simulation include the Poisson, binomial, and geometric distributions. Continuous distributions frequently used in simulation include the uniform, exponential, normal, triangular, lognormal, gamma, and Weibull distributions.

Help

Information about these distributions, as well as the syntax required for using distributions in AutoMod logic, is provided in the AutoMod Syntax Help.

Step 2 ***Estimate*** the parameters of the hypothesized distribution. For example, if the hypothesis is that the underlying data is normal, then the parameters to be estimated from the data are the mean and the variance.

Step 3 ***Perform*** a goodness-of-fit test. If the test rejects the hypothesis, that is a strong indication that the hypothesis is not true. In that case, return to step 1, or use the empirical distribution of the data, as discussed in the next section.

The three-step procedure is described more extensively in engineering statistics texts and in many simulation texts, such as Banks, Carson, Nelson, and Nicol (2000) and Law and Kelton (2000). Even if software is being used to aid in the development of an underlying distribution, understanding the three-step procedure is recommended.

Using the empirical distribution of the data

When only a small amount of data is available, an attempt to fit a distribution is inappropriate, as indicated previously. Also, when all possibilities have been exhausted for fitting a distribution using conventional techniques, then the empirical distribution (actual data values) can be used.

Consider a real-world example in which the times to repair a conveyor system after a failure, denoted by x, for the previous 100 occurrences were given as follows:

Intervals (hours)	Frequency of Occurrence
$0 < x \leq 1.0$	27
$1.0 < x \leq 2.0$	13
$2.0 < x \leq 3.0$	31
$3.0 < x \leq 4.0$	18
$4.0 < x \leq 8.0$	11

No distribution could acceptably be fit to the data using conventional techniques. It was decided to use the data, as generated, for the simulation. That is, samples were drawn, at random, from the continuous distribution shown above. Linear interpolation was used so that simulated values might be in the form 2.89 hours, 1.63 hours, and so on.

What to do when no data is available

There are many cases where no data is available. This is particularly true in the early stages of a study, when the data is missing, when the data is too expensive to gather, or when the system being modeled does not yet exist.

One possibility in such a case is to obtain a subjective estimate concerning the system. Thus, if the estimate is that the time to repair a machine is between 3 and 8 minutes, a crude assumption is that the data follows a uniform distribution with a minimum value of 3 minutes and a maximum value of 8 minutes. The uniform distribution is referred to as the “distribution of maximum ignorance,” because it assumes that every value is equally likely. A better “guess” occurs if the most likely value can also be estimated. This would take the form “the time to repair the machine is between 3 and 8 minutes with a most likely time of 5 minutes.” Now, a triangular distribution can be used with a minimum of 3 minutes, a maximum of 8 minutes, and a most likely value (mode) of 5 minutes.

As indicated previously, there are naturally occurring processes that give rise to distributions. For example, if the time to failure follows the (reasonable) assumptions of the Poisson process, indicated previously, and the machine operator says that the machine fails about once every two hours of operation, then an exponential distribution for time to failure could be assumed, initially with a mean of two hours.

Estimates made on the basis of guesses and assumptions are strictly tentative. If, and when, data, or more data, becomes available, both the parameters and the distributional forms should be updated.

Verification and validation

When using simulation, the analyst must abstract information about the real system to make a conceptual model. The conceptual model is then coded into the operational simulation model. There is a two-step process to ensure that the operational model is an accurate representation of the real-world system. The process involves verification and validation of the simulation model.

Verification A determination of whether the model built in the simulation package (the operational model) is a correct representation of the conceptual model.

Validation A determination of whether the simulation model can be substituted for the real system for the purposes of experimentation.

This process is iterative. If there are discrepancies between the operational and conceptual models, or between the operational model and the real-world system, the operational model must be examined for errors, or the conceptual model must be modified in order to better represent the real-world system (with subsequent changes in the operational model). The verification and validation process should then be repeated.

Verification

The verification process involves examination of the simulation program to ensure that the operational (simulation) model accurately reflects the conceptual model. There are many common-sense ways to perform verification. Balci (1998) presents more detailed information on the topic.

1. **Follow the principles of structured programming.** The first principle is top-down design (that is, construct a detailed plan of the simulation model before coding). The second principle is program modularity (that is, break the simulation model into submodels). As you will see, AutoMod models follow the modularity principle.

   Write the simulation model in a logical, well-ordered manner. It is highly advisable to prepare a detailed flow chart indicating the macro activities that are to be accomplished. This is particularly true for large, real-world problems. It is possible to think through all of the logic needed for the exercises in this textbook. However, the amount of computer logic required for these exercises is small compared to that of real-world problems.

2. **Make the model as self-documenting as possible.** This requires comments on most lines, and sometimes between lines, of logic. Imagine that one of your colleagues is trying to understand the computer logic that you have written, but that you are not available to offer any explanation.

3. **Have the model code checked by more than one person.** There are several software engineering techniques used to review code that can be applied to simulation models, including: code reviews to highlight design deficiencies; audits to verify that the development of the code is proceeding logically and that the requirements are being met; and code inspection. A code inspection involves the designer, the modeler, a tester, and a moderator. The team meets and reviews the design and the model line by line. The documentation is also reviewed. Errors are written up, classified, and fixed, and another inspection occurs to make sure all issues have been addressed. Any of these methods can be used to verify your model.

4. **Ensure that the values of the input data are being used appropriately.** For example, if the interarrival times are in minutes, but the model is using seconds, the model is inaccurate.

5. **For a variety of input data values, ensure that the outputs are reasonable.** Many simulation analysts are satisfied when they receive output, but that is far from enough. If there are 100 loads in a waiting line, but you only expect 10, there is probably something wrong, such as modeling the capacity of a resource incorrectly.
6. **Use the AutoMod Debugger to check that the program operates as intended.** The Debugger is a very important verification tool that should be used for all real-system models. An example of one of the capabilities of the Debugger is a trace, which permits following the execution of the computer logic step-by-step.Using the AutoMod Debugger is not described in this textbook. For information about using the Debugger, see the “Running a Model” chapter in volume 1 of the *AutoMod User’s Manual,* online.
7. **Watch the model’s animation.** Using animation, the simulation analyst can detect actions that are illogical. For example, you may observe that when a resource fails, its graphic is supposed to turn red on the screen. While watching the animation, the resource never turns red. This could signal an error in the way that down times were modeled.

Validation

A variety of subjective and objective techniques can be used to validate the conceptual model. Balci (1998) and Sargent (1992) offer many suggestions for validation.

Subjective techniques include the following:

1. **Face validation –** A conceptual model of a real-world system must appear reasonable “on its face” to those that are knowledgeable about the real-world system. For example, the system experts can validate that the model assumptions are correct. Such a critique aids in identifying deficiencies or errors in the conceptual model. Eliminating these errors enhances the credibility of the conceptual model.
2. **Sensitivity analysis –** As model input is changed, the output should change in a predictable direction. For example, if the arrival rate increases, the time loads spend in queues should increase (barring other modifications, such as increased capacity).
3. **Extreme-condition tests** – Does the model behave properly when input data is at the extremes? If the arrival rate is set extremely high, does the output reflect this change with increased numbers in the queues, increased time in the system, and so on?
4. **Validation of conceptual model assumptions** – There are two types of conceptual model assumptions: structural assumptions (concerning the operation of the real-world system) and data assumptions. **Structural assumptions** can be validated by observing the real-world system and by discussing it with the appropriate personnel. Because no one person knows everything about the entire system, you must consult a variety of people to validate structural assumptions.

   Data assumptions should also be validated. For example, suppose it is assumed that the arrival times between customers at a bank during peak periods are independent and in accordance with an exponential distribution. To validate these assumptions:

   - Consult with appropriate personnel to determine when peak periods occur.
   - Collect interarrival data from these periods.
   - Conduct statistical tests to ensure that the assumption of independence is reasonable.
   - Estimate the parameter of the assumed exponential distribution.
   - Conduct a goodness-of-fit test to ensure the exponential distribution is reasonable.

   Information from intermediaries should also be questioned. For example, if you are subcontracting, make sure you understand the information the contractor is giving you. One simulation consultant was working through another consulting firm on an extremely large model of a distant port operation. It was only after a site visit that the simulation consulting firm discovered that one of the major model assumptions concerning how piles of iron ore are formed was wrong.

5. **Consistency checks** – Continue to examine the operational model over time. For example, if you use a simulation model annually, before using the model, make sure that there are no changes in the real system that must be reflected in the model. Similarly, the data should also be validated. For example, a faster machine may have been installed, but the processing time data was not updated.
6. **Turing tests** – Persons knowledgeable about system behavior can be used to compare model output to system output. For example, suppose that five reports of actual system performance over five different days are prepared, and five simulated outputs are generated. These 10 reports should be in the same format. The 10 reports are randomly shuffled and given to an expert on the system, such as an engineer. The engineer is asked to distinguish between the two kinds of reports, actual and simulated. If the engineer identifies a substantial number of simulated reports, then the model is inadequate. If the engineer cannot distinguish between the two, then there is less reason to doubt the adequacy of the model.

Objective techniques include the following:

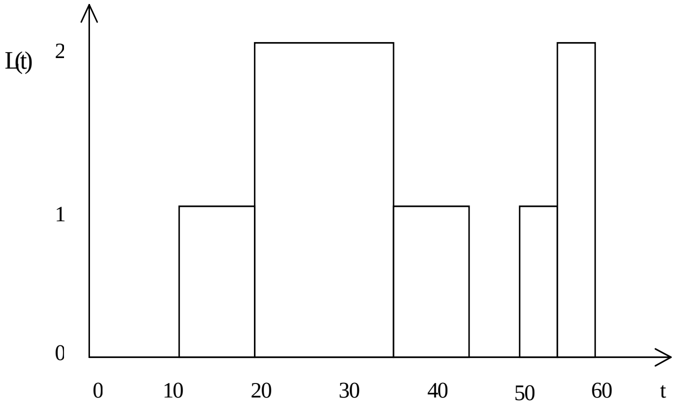
1. **Validating input-output transformations** – The basic principle of this technique is the comparison of output from the operational model to data from the real system. Input-output validation requires that the real system currently exists. One method of comparison uses the familiar *t*-test, discussed in most statistics texts.
2. **Validation using historical input data** – Instead of running the operational model with artificial input data, we could drive the operational model with the actual historical record. It is reasonable to expect the simulation to yield output results within acceptable statistical error of those observed from the real-world system. The paired *t*-test, discussed in most statistics texts, is one method for conducting this type of validation.

Experimentation and output analysis

The analysis of simulation output begins with the selection of performance measures. Performance measures can be time weighted, based on counting the occurrences of an event, or arise from the tabulation of expressions including means, variances, and so on.

An example of a time-weighted statistic is the average number of loads in a system over a time period of length "t." The graph below shows the number of loads in the system, L(t), at time t, from $t = 0$ to $t = 60$. The time-weighted average number of loads in the system, $\bar{L}$, at $t = 60$, is given by the sum of the areas of the rectangles divided by t. Thus,

$$\bar{L} = \frac{[(0 \times 10) + (1 \times 10) + (2 \times 15) + (1 \times 10) + (0 \times 5) + (1 \times 5) + (2 \times 5)]}{60} = 1.08$$



Number of loads in system, L(t), at time t

An example of a statistic based on counting the number of occurrences of an event is the number of acceptable loads completed in 24 hours of simulated time. A statistic based on the tabulation of expressions is the patent royalties from three different part types, each with a different contribution per load, for a 24-hour period.

The simulation of a random, or **stochastic**, system results in performance measures that contain random variation. Proper analysis of the output is required to obtain sound statistical results from these replications. Questions that must be addressed when conducting output analysis are:

- What is the appropriate run length of the simulation (unless the system dictates a value)?
- How do we interpret the simulated results?
- How do we analyze the differences between different model configurations?

Statistical confidence

A confidence interval for the performance measure being estimated by the simulation model is a basic component of output analysis.

Note

The AutoStat statistical analysis package can determine confidence intervals for you, as discussed in chapter 8, “Basic Statistical Analysis Using AutoStat.”

A confidence interval is a numerical range that has a probability $1 - \alpha$ of including the true value of the performance measure, where $1 - \alpha$ is the confidence level (such as 95 percent) for the interval. For example, let us say that the performance measure of interest is the mean time in the queue, μ, and a $100(1 - \alpha)$ percent confidence interval for μ is desired. If many replications are performed and independent confidence intervals on μ are constructed from those replications, then approximately $100(1 - \alpha)$ percent of those intervals will contain μ. Consider the following example.

Example 1.2: Confidence intervals

For the data in the table below, we want to calculate both a 95 percent $(\alpha = 0.05)$ and a 99 percent $(\alpha = 0.01)$ two-sided confidence interval. Assuming that the values for X are normally distributed, a $1 - \alpha$ confidence interval for the mean, μ, is given by $(\overline{X} - h, \overline{X} + h)$ where $\overline{X}$ is the sample mean and h is the half width.

Replication Number (i)	Time in Process (X_i) (in seconds)
1	752.23
2	785.49
3	645.13
4	639.96
5	610.13
6	661.42
7	645.28
8	606.32
9	677.74
10	584.53

Time in process data

The equation for $\overline{X}$ is given by:

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

Equation 1.2 - Sample mean

where n = the total number of replications.

The half width, h, of the confidence interval is computed as follows:

$$h = t_{1-\alpha/2,\, n-1} \frac{(S)}{\sqrt{n}}$$

Equation 1.3 - Half width

where $t_{1-\alpha/2,\, n-1}$ is the upper $1 - \alpha/2$ critical value of the t distribution with $n - 1$ degrees of freedom, and S is the sample standard deviation.

To compute S, first use equation 1.4 to compute S^2, the sample variance, as follows:

$$S^2 = \frac{\sum_{i=1}^{n} (X_i - \overline{X}^2)}{(n-1)}$$

Equation 1.4 - Sample variance

Taking the square root of S^2 yields S.

Because we want a two-sided confidence interval, we use $\alpha/2$ to compute the half width. Using equations 1.2 and 1.4, we obtain $\overline{X}$ = 660.82 seconds and S = 63.655 seconds.

In addition, when using equation 1.3, you can substitute a constant value for $1 - \alpha/2$ based on the confidence level you want (see Banks, Carson, Nelson, and Nicol (2000) for a table of t statistics). Constants for two commonly used confidence levels are shown below:

$t_{1 - .05/2, 9} = 2.262$ (95 percent confidence)

$t_{1 - .01/2, 9} = 3.250$ (99 percent confidence)

Therefore, using equation 1.3, we obtain a half width of:

$$h = 2.262\frac{63.655}{\sqrt{10}} \text{ so } h = 45.53 \text{ (95 percent confidence)}$$

and

$$h = 3.250\frac{63.655}{\sqrt{10}} \text{ so } h = 65.42 \text{ (99 percent confidence)}$$

The confidence interval is given by

$$(\overline{X} - h, \overline{X} + h)$$

Therefore, the 95 percent confidence interval is (615.29, 706.35) seconds, while the 99 percent confidence interval is (595.40, 726.24) seconds.

As demonstrated in example 1.2, the size of the interval depends on the confidence level desired, the sample size, and the inherent variation (measured by S). The higher level of confidence (99 percent) requires a larger interval compared with the lower confidence level (95 percent). In addition, the number of replications, n, and their standard deviation, S, are used in calculating the confidence interval. In simulation, each replication is considered one data point. Therefore, the factors that influence the width of the confidence interval are:

- Total number of replications (n)
- Level of confidence $(1 - \alpha)$
- Standard deviation of performance measure (S)

The relationship between these factors and the confidence interval is:

- As the number of replications increases, the width of the confidence interval decreases.
- As the level of confidence increases, the width of the interval increases. In other words, a 99 percent confidence interval is wider than the corresponding 95 percent confidence interval.
- As the standard deviation increases, the width of the interval increases.

Terminating versus non-terminating systems

The procedure for output analysis differs based on whether the system is terminating or non-terminating.

Terminating systems

In a **terminating system**, the duration of the simulation is fixed as a natural consequence of the model and its assumptions. The duration can be fixed by specifying a finite length of time to simulate or by limiting the number of loads created or disposed. An example of a terminating system is a bank that opens at 9:00 A.M. and closes at 4:00 P.M. Some other examples of terminating systems include a check processing facility that operates from 8:00 P.M. until all checks are processed, a ticket booth that remains open until all the tickets are sold or the event begins, and a manufacturing facility that processes a fixed number of jobs each day and then shuts down.

By definition, a terminating system is one that has a fixed starting condition and an event definition that marks the end of the simulation. The system returns to the fixed initial condition, usually “empty and idle,” before the system begins operation again. The objective of the simulation of terminating systems is to understand system behavior for a “typical” fixed duration. Because the initial starting conditions and the length of the simulation are fixed, the only controllable factor is the number of replications.

Therefore, the analysis procedure for terminating systems is to simulate a number of replications, compute the sample variance of the selected estimate, and determine whether the width of the resulting confidence interval is within acceptable limits.

For example, to analyze a model in which the average number of parts in a queue is of interest:

Step 1 ***Conduct*** a pilot run of “n” replications.

Step 2 ***Compute*** the confidence interval for the expected average number of parts in the queue using the observations recorded from each replication.

Step 3 If the confidence interval is too large, ***determine*** the number of additional replications required to bring it within limits.

Step 4 ***Conduct*** the additional replications and recompute the new confidence interval using all of the data. ***Iterate*** steps 3 and 4 until the confidence interval is of satisfactory size.

Non-terminating systems

In a **non-terminating system**, the duration is not finite; the system is in perpetual operation. An example of a non-terminating system is an assembly line that operates 24 hours a day, 7 days a week. Another example of this type of system is the manufacture of glass fiber insulation for attics. If operation of the system is stopped, the molten glass solidifies in the furnace, needing to be tediously chipped away before restarting the system. The objective in simulating a non-terminating system is to understand the long run, or “steady-state” behavior. To accurately study steady-state behavior, the effects of the initial conditions (that is, the transient phase), must be removed from the simulation results. The transient phase can be removed using one of the following methods:

- Swamping
- Pre-loading
- Deletion

Swamping

Swamping suppresses the initial-condition effects by conducting a very long simulation run — so long that any initial conditions have only a minuscule effect on the long-run value of the performance measure. For example, if the initial conditions last for 100 hours, you would simulate for 10,000 hours. A problem with the swamping technique is that the bias from starting empty and idle will always exist, even if it is small.

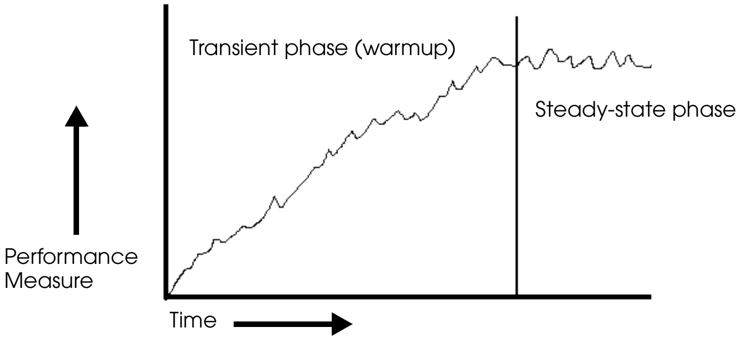
Pre-loading

Pre-loading primes the system before the simulation starts by placing loads at resources or on transportation devices, or in a waiting line for one of these. In other words, pre-loading attempts to make the initial conditions match steady-state conditions. This requires some rough knowledge of how the system looks in steady-state.

For example, if we are simulating a tool crib that has one line forming before three tool crib attendants, we need to observe the tool crib in operation in order to obtain information about the usual situation. We may find that the three tool crib attendants are usually busy, and that there are about four people in line. This is how the simulation would begin when using the pre-loading technique. The tool crib is a very simple system to observe. However, for more complex systems, this initialization procedure becomes somewhat difficult, especially if the system is still in the design phase.

Deletion

Deletion excludes the transient (warmup) phase, in which the system is influenced by the initial conditions. Data is collected from the simulation only after the transient phase has ended. This idea is demonstrated in the illustration below:



Deletion of initial observations for a non-terminating system

The difficulty with the deletion method is the determination of the length of the transient phase. Although elegant statistical techniques have been developed, a satisfactory method is to plot the output of interest over time and visually observe when steady-state is reached. Welch (1983) provides a formal description of this method. AutoStat, which can be used to determine a model's transient warmup period, uses Welch's method. Using AutoStat to determine the warmup time of a model is discussed in chapter 15, "Warmup Analysis Using AutoStat."

Queueing theory

We often need to simulate queues, or waiting lines. When we talk about a queue, we mean the line that forms before either a machine or a person (called a server). When we talk about a server, we mean both the queue and the server. Queues include work-in-process (WIP) that is waiting for a machine, WIP that is waiting for a transportation device, resources that are waiting to be repaired, and so on. In just a few paragraphs, we want to provide some very basic notions from the study of steady-state queues or waiting lines. Further information is available in Banks, Carson, Nelson, and Nicol (2000).

When we say a queue has reached "steady state," we mean that the waiting line has passed the transient phase (when it started empty and idle), and reached a situation in which there is random variation in a finite range. If you went to your neighborhood bank when it opened at 9:00 A.M., you would see the tellers and other service people idle. It would take a few minutes (the warmup period) to reach the normal hum of activity (the steady state). That's because the bank started empty and the tellers and other service people were idle at 9:00 A.M.

You can use the following queuing equations to determine how many servers are needed in a system, whether a server is fully utilized, and so on. The following symbols are used:

λ = arrival rate

μ = service rate

ρ = server utilization

c = number of servers

When there is **one server**, the utilization coefficient is:

$$\rho = \lambda/\mu$$

We usually talk about queueing in terms of distributions of time between arrivals and distributions of service time. For example, we can say that if the arrival rate is a random variable that averages four customers per hour and the service rate is also a random variable with an average of five customers per hour, the server utilization average is:

$$\rho = \frac{4}{5} = 0.8$$

On the average, the server is busy 80 percent of the time.

When there is **more than one server**, the utilization coefficient is:

$$\rho = \frac{\lambda}{(c\mu)}$$

For example, if the mean time between arrivals is 6.67 minutes (for an average rate of nine arrivals per hour) with two servers each able to serve a customer in 10 minutes (an average of six customers per hour), then the utilization rate would be

$$\rho = \frac{9}{[(2)(6)]} = 0.75$$

For any number of servers, if $\rho \geq 1$, the system is explosive. An **explosive system** is a system that grows without bound over time. Therefore, ρ must be less than one to maintain a stable system.

Note

Even if $\rho \leq 1$, the system might be explosive if there are reentrant queues (see Banks and Dai, 1997).

Another important part of queuing theory uses the following symbols to discuss time in system and number of items in a queue:

L = average number of loads in the system

L_Q = average number of loads in the queue

w = average time that a load is in the system

w_Q = average time that a load is in the queue

A very important result in queueing theory attributed to Little (1961) is that:

$$L = \lambda w$$

Thus, if w is known, then L is also known, because λ is just a parameter. Conversely, if L is known, w is known. It is also the case that:

$$L_Q = \lambda w_Q$$

Thus, if w_Q is known, then L_Q is known, because λ is just a parameter. Conversely, if L_Q is known, w_Q is known.

Consider the possible states of a load in the system; the load can be in service or in the queue. So, the time in the system is composed of waiting in the queue plus being serviced, or

$$w = w_Q + \frac{1}{\mu}$$

Multiplying both sides by λ gives:

$$\lambda w = \lambda w_Q + \frac{\lambda}{\mu}$$

or:

$$L = L_Q + \rho$$

So, if any of the four measures (L, L_Q, w, w_Q) is known, all of them are known.

Summary

This chapter began with a definition and example of simulation. We presented the underlying concepts of simulation and discussed the four simulation modeling methods used by the simulation community: process-interaction, event scheduling, activity scanning, and the three-phase method. We presented both the advantages and disadvantages of simulation, along with mitigating factors for the disadvantages. We then briefly outlined some of the applications of AutoMod and discussed the steps for completing a simulation study.

Our discussion of simulation included a presentation of the manner in which random numbers and random variates are generated and three ways that might be used for generating input data: assuming randomness away, fitting a distribution to data, and using the empirical distribution of the data. However, the first method, assuming randomness away, is discouraged. We described the important concepts of verification and validation, both of which are necessary for a simulation study. We also addressed the all-important topic of experimentation and output analysis. Lastly, we discussed some of the important elements of queueing theory.

Exercises

Exercise 1.1

Given the following numbers that represent the average time that loads spent in a process (in minutes) for 10 replications, find the 95 percent and 99 percent confidence intervals (see “Example 1.2: Confidence intervals” on page 1.25):

```
122.91   125.55   92.44   29.82   145.57   139.58   131.17   159.13   99.29   163.89
```

Exercise 1.2

Rework the ad hoc simulation table in example 1.1 (see “Example 1.1: Ad hoc simulation” on page 1.4) using the following service times (in order of their appearance from left to right):

```
2  5  1  3  6  5  3  2  1  6  3  4  1  6  3  3  2  4  4  5
```

Use the same random numbers for time between arrivals that were used in example 1.1.

Exercise 1.3

Using a spreadsheet package such as Excel, use the equation for generating random variates to generate 1000 random values with $\lambda = 1/10$ per minute (see “Equation 1.1 - Random variate generator for the exponential distribution” on page 1.17 for more information). Create a histogram to display these values.

Tip ☞ Steps for creating a histogram in Excel are included below (if you are using another spreadsheet package, consult that package’s documentation for information about how to create a histogram):

Step 1 ***Calculate*** the random values.

Step 2 In an empty column, ***define*** a column of numbers that begins with 0 and increments to 100 in intervals of 10. (These values are used later to defined the bin range for the histogram.)

Step 3 From the Tools menu, ***select*** Add-Ins. The Add-Ins window opens.

Step 4 If it is not already selected, ***select*** Analysis ToolPak and ***click*** Ok. You have now added histogram functionality to your spreadsheet.

Step 5 From the Tools menu, ***select*** Data Analysis. The Data Analysis window opens.

Step 6 ***Select*** Histogram and ***click*** Ok. ***Use*** the window to create the histogram (be sure to select Chart Output to create a histogram chart).

Exercise 1.4

Rework the ad hoc simulation table in exercise 1.2 by doubling service times (the first service time is now 4 minutes, the second is now 10 minutes, and so on). Also assume there are two tool crib attendants working independently. Continue using the same random numbers for time between arrivals.

Exercise 1.5

Using a spreadsheet program such as Excel, solve example 1.1 (see “Example 1.1: Ad hoc simulation” on page 1.4). Have the spreadsheet generate the required random numbers.

Exercise 1.6

People arrive at the division of motor vehicles to get license plates at a random rate with a mean time between arrivals of 5 minutes. An attendant can serve each person in an average time of 19 minutes. What is the minimum number of attendants needed so that the system is not explosive?

Exercise 1.7

Widgets at a facility are generated at a random rate with a mean time between arrivals of 20 minutes. The facility has four identical machines that process the widgets; a widget needs to be processed by only one machine. The amount of time that a machine takes to process a widget is randomly distributed with a mean of 90 minutes. Is this system stable or explosive? Why or why not?

Chapter 2

Using the Software

Chapter 2

Using the Software

In this chapter, you are going to learn how to open the AutoMod software and import and run a model. The chapter will also briefly describe some output from the simulation and show you how to edit a model.

Simulation and AutoMod are subjects that require both study and hands-on experience. You are going to need to spend time studying this textbook and working through the exercises at the end of each chapter to become proficient with the software. After completing this textbook, you will be able to build accurate and effective AutoMod simulation models.

Opening the AutoMod software

To open the AutoMod software, do the following:

Step 1 From the Start menu, ***select*** AutoMod 9.1 (Student Version) > AutoMod. The Work Area window opens.

or

Step 1 Using Explorer, ***navigate*** to the bin directory of the AutoMod installation.

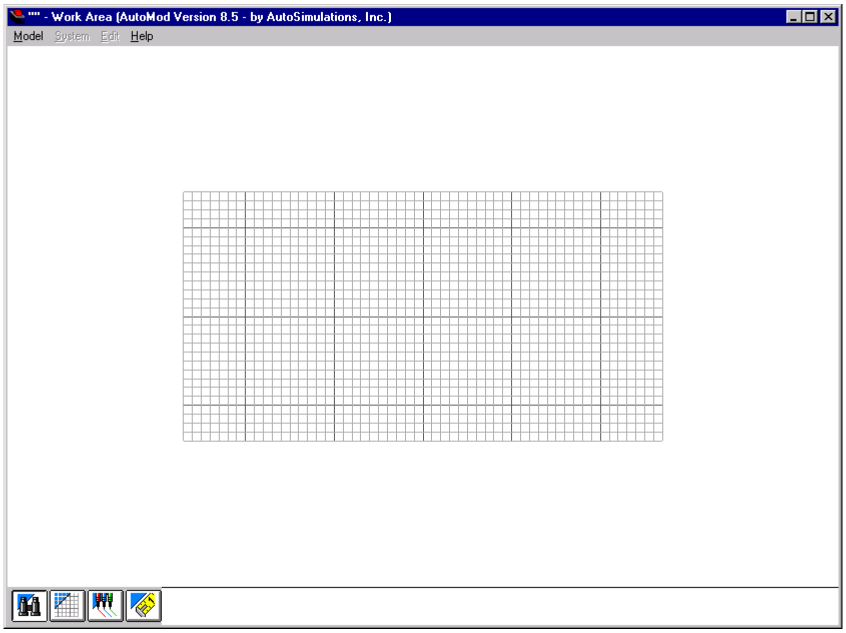
Step 2 ***Double-click*** the “amod.exe” file. The Work Area window opens.

The edit environment

There are two parts to the AutoMod software: the edit environment and the simulation environment.

The **edit environment**, also referred to as the build environment, is where you build your model and define model parameters. The **simulation environment** is where you run the model and view the results. You will learn more about the simulation environment later in this chapter (see “The simulation environment” on page 2.7).

When you first open the AutoMod software, you are in the edit environment and the Work Area window is displayed.


"" - Work Area (AutoMod Version 8.5 - by AutoSimulations, Inc.)
Model
System
Edit
Help

Work Area window

In this window, you can either create a new model or import an existing model. In this chapter, you will import an existing model.

Importing a model

Importing a model allows you to open an archived model. You will learn more about the difference between importing and opening models later in this textbook (see "Exporting versus saving a model" on page 3.5 of the "AutoMod Concepts" chapter for more information).

To import the example model for this chapter, complete the following:

Step 1 From the Model menu in the Work Area window, ***select*** Import.

Step 2 ***Navigate*** to the "demos\gswa\examp02_1" directory in the AutoMod installation directory ("gswa" is short for *Getting Started With AutoMod*).

Step 3 ***Double-click*** the "examp21.arc" folder.

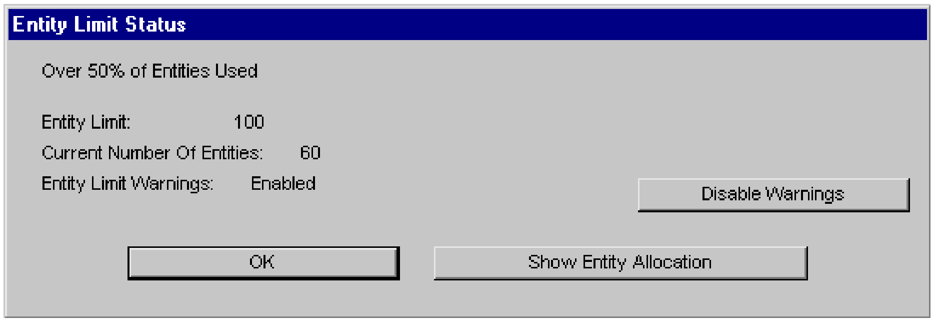
Step 4 ***Select*** the "model.amo" file and ***click*** Open. The Entity Limit Status window opens, which is discussed in the next section.

Step 5 ***Click*** OK to continue. The imported model is displayed in the Work Area window and the Process System palette appears.

Counting the number of entities in your model

The student version of the AutoMod software limits the number of entities (such as operators, queues, and equipment) in a model to 100. To check the number of entities defined in your model, do the following:

Step 1 From the Model menu, ***select*** Check Entity Limit. The Entity Limit Status window opens.



Entity Limit Status

Over 50% of Entities Used

Entity Limit: 100
Current Number Of Entities: 60
Entity Limit Warnings: Enabled

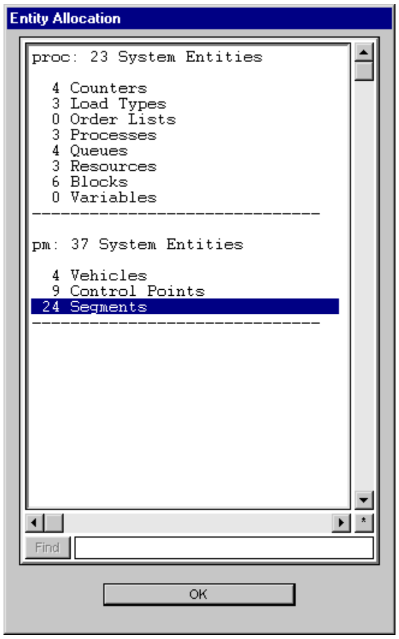
Entity Limit Status window

The Entity Limit Status window opens whenever you import a model, or whenever you try to build, save, or run a model containing over 100 entities. It also opens when you have placed fifty percent of the available entities, seventy-five percent of the available entities, and when you place the last ten entities. If you want to turn off the entity limit warning, click the Disable Warnings button.

Displaying entity allocation

Entities in a model are grouped into systems, such as a conveyor system, a vehicle system, and so on (systems are discussed in detail in “Systems” on page 3.7 of the “AutoMod Concepts” chapter). For more detailed information about the entities in your model, you can display the number and type of entities in each system:

Step 1 In the Entity Limit Status window, ***click*** the Show Entity Allocation button. The Entity Allocation window opens, displaying the number of each type of entity currently in your model.



```
proc:  23 System Entities

   4 Counters
   3 Load Types
   0 Order Lists
   3 Processes
   4 Queues
   3 Resources
   6 Blocks
   0 Variables
------------------------------

pm:  37 System Entities

   4 Vehicles
   9 Control Points
  24 Segments
------------------------------
```

Entity Allocation window

The Entity Allocation window lists each system and indicates the number of entities of each type within that system.

Step 2 ***Click*** OK to close the Entity Allocation window.

Step 3 ***Click*** OK to close the Entity Limit Status window.

Note You will be able to complete all of the exercises in this textbook using fewer than 100 entities.

You are now ready to run the model and watch the simulation.

Running a model

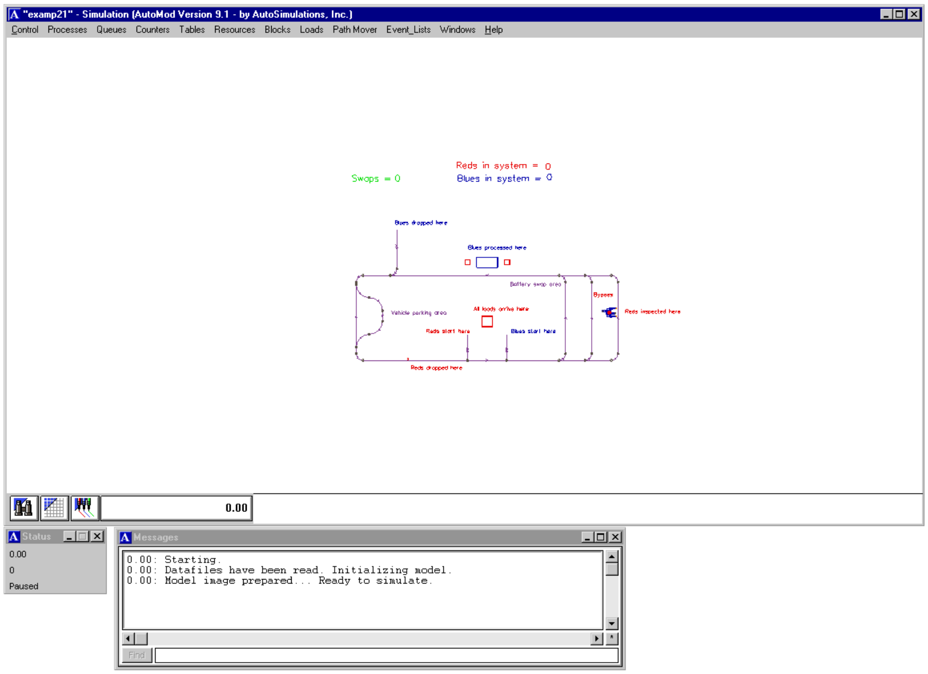
To run the example model and watch it the simulation:

Step 1 From the Model menu in the Work Area window, ***select*** Run Model. A confirmation window opens.

Step 2 ***Click*** Yes to build (compile) the example model. After the model has been compiled, the simulation environment opens.

The simulation environment

The **simulation environment** is where you run a model, watch the animation, and gather statistics, as shown below:



```
0.00
0
Paused
```

```
0.00: Starting.
0.00: Datafiles have been read. Initializing model.
0.00: Model image prepared... Ready to simulate.
```

Simulation environment

The simulation environment consists of three windows:

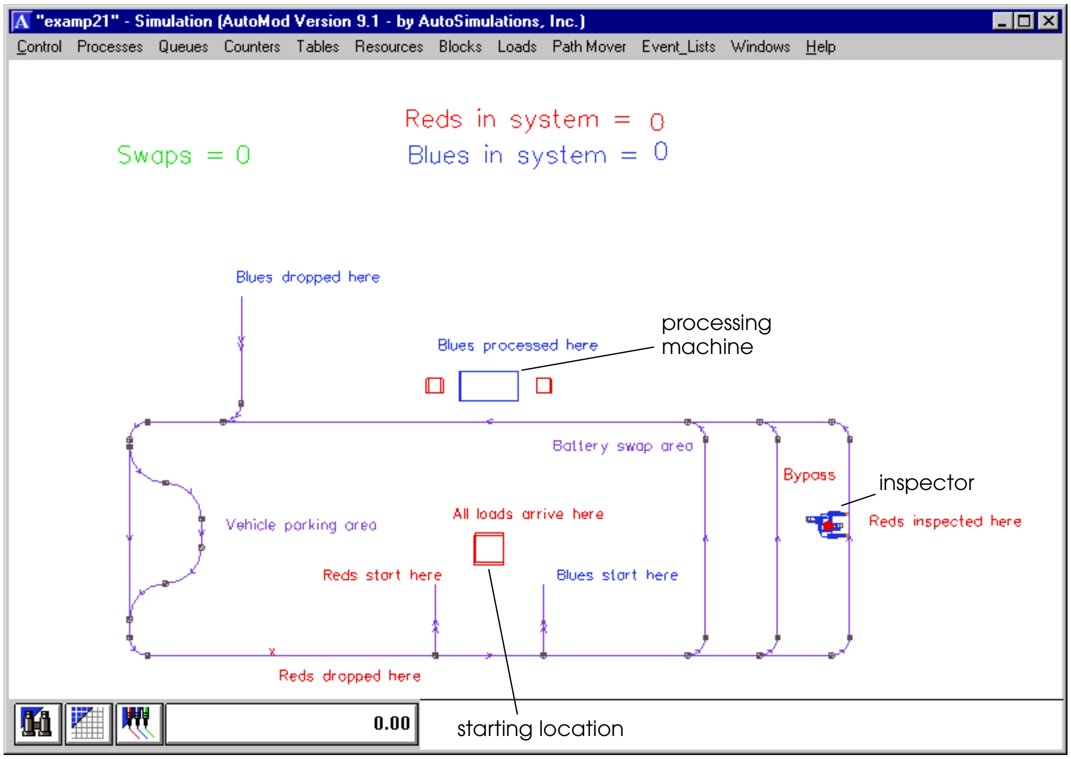
Simulation window Shows the model layout and animation.

Status window Shows the current simulation time and indicates whether the simulation has been paused or is continuing (running).

Message window Shows messages and errors for the model. (For more information, see “The print action” on page 4.12 of the “Introduction to AutoMod Syntax” chapter.)

Example model 2.1: Load inspection and processing in an AGV system

Take a moment to consider the layout for example model 2.1. An enlarged picture of the layout is shown below:



Example model 2.1 layout

The example model contains a system of paths on which automated guided vehicles (AGVs) can travel. Two types of **loads** (which represent products) are carried onboard vehicles in the facility: red loads and blue loads. Both types of loads first arrive in the queue (waiting area) near the center of the path system. The loads are then picked up by a vehicle on one of two spur paths located on either side of the arrival queue.

Red loads travel by vehicle to an inspector to the right. They are inspected while on the vehicle, and then travel to the drop-off point marked by an “X” on the lowest horizontal path.

Blue loads travel by vehicle to a processing area on the upmost horizontal path. The processing area consists of a machine with a queue on either side. Blue loads are unloaded into the queue on the right side, where they wait to use the machine. After using the machine, they are placed in the queue on the left side and wait to be picked up by a vehicle. Once on a vehicle, the blue loads are taken to the drop-off location at the end of the uppermost vertical path.

There is a battery swapping area on the first vertical path on the right, to which vehicles travel to have their batteries replaced when the charge is running low. Only AGVs requiring battery replacement travel on that path. The vertical path to the right of the battery swapping area is a bypass for vehicles that need to travel around the system, but are not replacing a battery or delivering a red load to be inspected. Finally, there is a parking area to which empty vehicles travel when there are no loads requiring delivery; the parking area gets vehicles out of the way so they do not block vehicles traveling on the path.

Pausing and continuing a simulation

When the simulation environment first opens, the simulation is paused. To continue the simulation:

Step 1 From the Control menu, ***Select*** Continue.

Take a moment to watch the simulation. There are two labels near the top of the screen that indicate the number of red and blue loads in the system. The green label (to the left) indicates the number of batteries that have been replaced.

Note

Notice the clock at the bottom of the Simulation window; the time is displayed in the following format: days:hours:minutes:seconds:hundredths of seconds.

After you have watched the simulation for a few minutes, pause the simulation by selecting Pause from the Control menu.

Tip

You can also toggle the simulation between paused or continued by pressing “p” (lowercase).

Changing the display step

The **display step** is the period of simulated time between animation updates. The longer the display step, the faster the simulation. Conversely, setting a shorter display step slows the simulation because graphics need to be redrawn more frequently. The display step at the beginning of a simulation is set to one second.

To change the display step:

Step 1 From the Control menu, ***select*** Display Step. The Change Display Step window opens.

Step 2 ***Type*** a number, then ***select*** a unit of time measurement from the drop-down list. For example, type “2” and select “seconds” to set the display step to 2 seconds.

Step 3 ***Click*** OK to close the Change Display Step window. The display step has now been changed.

If the model is currently paused, press “p” to continue the simulation so that you can see the effect of the new display step.

Tip

You can double the display step during a simulation by pressing “D” (uppercase). You can halve the display step during a simulation by pressing “d” (lowercase).

Toggling the animation on and off

You can greatly increase the speed of a simulation by turning off the animation. To turn off the animation:

Step 1 From the Control menu, ***select*** Turn off Animate.

If the model is currently paused, press “p” to continue the simulation.

Note

Because the animation is turned off, the clock is not updated and there is no change in the Simulation window. The Status window indicates whether the simulation is paused or is continuing.

The message “End of Run” appears in the Message window when the simulation is complete.

You can turn animation on by selecting Turn on Animate from the Control menu.

Tip

You can toggle the animation on and off during a simulation by pressing “g” (lowercase).

Changing the view

Because AutoMod animation is 3-dimensional, you can zoom in or out and rotate graphics to view the simulation from any position in 3-D space. Centering and zooming the screen is accomplished using the mouse.

Centering and zooming

On a PC with a two-button mouse, the *right* mouse button is used for centering and zooming. On a PC with a three-button mouse, the *middle* mouse button is used for centering and zooming.

To **center** the display at the position of the mouse pointer, move the pointer to the desired location and click the appropriate mouse button. Wherever you click becomes the center of the screen.

To **zoom** in on an area, hold down the button for centering and zooming (indicated above), drag the mouse to **“band”** the area, and release the mouse button. The banded area enlarges to fill the window.

Note

You can undo the last zoom (or other view change) with the keyboard command Control+Shift+U. You can return to the top view by pressing “v” (lowercase).

Rotating the picture

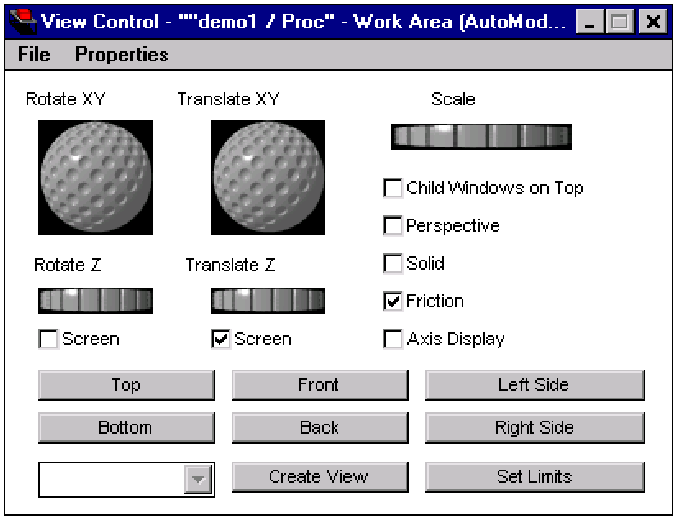
AutoMod models are 3-dimensional and contain an origin around which the graphics can be viewed. The View Control window allows you to change the view of your model.

Note The View Control works the same in both the edit environment and the simulation environment.

To open the View Control window:

Step 1 ***Click*** View Control in the lower-left corner of the Work Area window.

The View Control window opens.



□ Screen (Rotate Z)
☑ Screen (Translate Z)
□ Child Windows on Top
□ Perspective
□ Solid
☑ Friction
□ Axis Display

View Control window

Options in the View Control window are:

Rotate Rotating moves the model graphics ***around*** the different axes.

Translate Translating moves the model graphics ***along*** the different axes.

Scale Scaling the model makes the graphics bigger or smaller on the screen.

Child Windows on Top This option prevents the palette and dialog windows from moving behind other windows, such as the Simulation window.

Perspective Perspective controls the view of the model. When selected, **Perspective** shows all lines leading to a vanishing point; this is how we naturally view the world.

When the perspective box is cleared, the model is shown in **Orthogonal** view. All lines are at right angles to each other; this is a projection of a 2-D drawing into a 3-D view.

Solid Solid displays model entities as solid objects. When solid is off, only the wireframe outlines of shapes are displayed.

Friction Friction controls continuous movement, including translation, rotation, and scaling. When friction is on, the graphics move only when you indicate. When friction is off, any movement command causes the model to move continuously until you explicitly stop it, either by toggling friction on again or by pressing the space bar.

Axis Display Axis Display causes a triad (X, Y, Z) to be displayed at the model’s origin.

Screen The Screen check box controls the movement of the model in relation to screen coordinates or world coordinates. The **world** coordinates are in relation to the model’s origin (Axis Display). The **screen** coordinates are in relation to the current screen view.

The axes of the following views originate from the **world** coordinates:

Top View the entity from the positive Z axis.

Front View the entity from the negative Y axis.

Bottom View the entity from the negative Z axis.

Back View the entity from the positive Y axis.

Right Side View the entity from the positive X axis.

Left Side View the entity from the negative X axis.

Create Views It is possible to define views of your model and name them so that you can display that view later. To do this, adjust your model to the desired view, then click Create Views. Name the view. When you want to return to a view later, select the name of the view from the drop-down list in the View Control window.

Set Limits This button displays the Set Limits window, which allows you to adjust the size of the drawing grid.

Take a few minutes to familiarize yourself with the view control. When you are finished, do the following:

Step 1 In the View Control window, ***select*** Child Windows on Top.

Step 2 ***Close*** the view control.

Tip ☞ Remember that you can return to the top view by pressing “v” (lowercase).

Using keyboard shortcuts

Many of the viewing changes you have just made can also be accomplished using the keyboard.

Help

There is a help file that lists all of the AutoMod keyboard shortcuts. To open the help file, select Keyboard Help from the Help menu. The Keyboard Command Help opens.

Note

All keyboard commands are case-sensitive.

Take a few minutes to review the help file and become familiar with some of the basic keyboard shortcuts, such as:

h = Help file display

X = rotate X axis clockwise

x = rotate X axis counterclockwise

Y = rotate Y axis clockwise

y = rotate Y axis counterclockwise

Z = rotate Z axis clockwise

z = rotate Z axis counterclockwise

G = grid toggle

w = solid

When you are finished, close the Keyboard Command Help.

Saving the configuration of windows and views

To save the current size and position of your windows, as well as the angle and view of the graphics, save the startup configuration. Every time you run this model, the view appears as you have saved it.

For example, you can organize the placement of the windows on your screen in any manner you desire: resize the Simulation window, reposition the Message and Status windows, rotate the picture, and set options from the View Control.

To save the Startup Configuration:

Step 1 Once the window locations, views, and other options are set as you want, ***select*** Save Startup Config from the Control menu.

A new file called ".am2rc," is created in the model directory; it contains the instructions for setting up the model. When you open any model within the directory containing the ".am2rc" file, the screen appears as you have set it.

Displaying statistics

If you have not already done so, run the simulation to completion. The simulation runs for five days.

Tip ☞ Remember that you can speed up the simulation by turning off the animation.

Whenever you run a simulation model, statistics are recorded to help you understand the system's behavior. Statistics can be displayed in three formats:

Statistics summary Summarizes the statistics for a group of similar entities (for example, all queues, all resources, and so on).

Sorted Statistics Sorts entities in a statistic summary by a selected value, such as total, current, or average. Statistics can be sorted from low values to high, or from high to low.

Single statistic Displays statistics for a single entity (for example, a specific queue or resource).

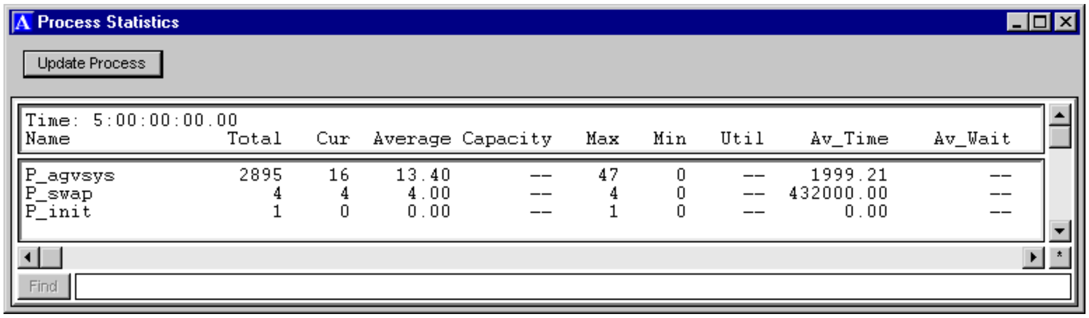
In this chapter, we are concerned only with displaying statistics summaries.

Note Statistics are dynamically updated; you can display them at any time during a simulation, as well as at the end of the run.

Displaying process system summary statistics

Processes are sets of instructions for the products being sent through the system (for more information, see “The process system” on page 3.7 of the “AutoMod Concepts” chapter). Process statistics provide information about each set of instructions. To display a statistic summary for processes, do the following:

Step 1 From the Processes menu, ***select*** Statistics Summary. The Statistics Summary window opens, as shown below:



```
Time: 5:00:00:00.00
Name                Total    Cur   Average Capacity    Max    Min    Util     Av_Time     Av_Wait

P_agvsys             2895     16     13.40       --     47      0      --     1999.21          --
P_swap                  4      4      4.00       --      4      0      --   432000.00          --
P_init                  1      0      0.00       --      1      0      --        0.00          --
```

Process summary statistics

Process statistics are defined as follows:

Name The name of the process.

Total The total number of loads that were sent to the process.

Cur The number of loads that are currently in the process.

Tip

You can calculate the number of loads that *completed* each process by subtracting the Cur statistic (the number of loads currently in the process) from the Total statistic (the number of loads that were sent to the process).

Average The average number of loads that were in the process at the same time.

Max The maximum number of loads that were in the process at the same time.

Min The minimum number of loads that were in the process at the same time.

Av_Time The average time that loads spent in the process.

Important

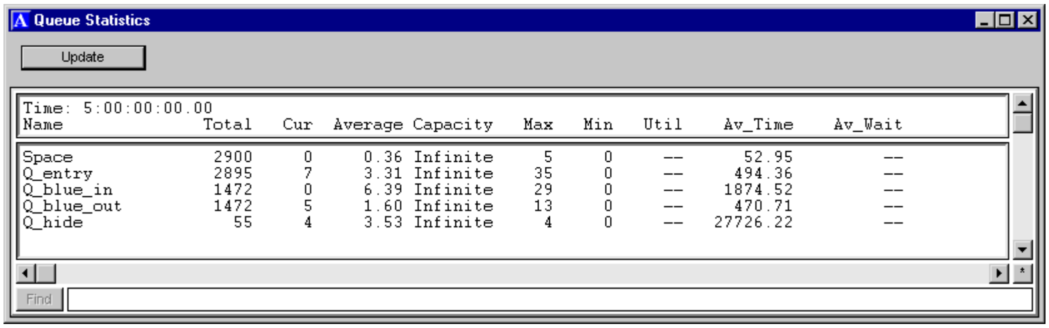
Throughout this textbook, you will frequently use process statistics to determine the number of loads that completed a process and the average time that loads spent in a process.

Displaying queue summary statistics

Queues are physical spaces where loads can wait or be processed (for more information, see “Queues” on page 5.12 of the “Process System Basics” chapter).

To display a statistic summary for queues:

Step 1 From the Queues menu, ***select*** Statistics Summary. The Queue Statistics window opens, as shown below:



```
Time: 5:00:00:00.00
Name              Total   Cur  Average Capacity   Max   Min   Util    Av_Time    Av_Wait

Space              2900     0     0.36 Infinite     5     0     --      52.95         --
Q_entry            2895     7     3.31 Infinite    35     0     --     494.36         --
Q_blue_in          1472     0     6.39 Infinite    29     0     --    1874.52         --
Q_blue_out         1472     5     1.60 Infinite    13     0     --     470.71         --
Q_hide               55     4     3.53 Infinite     4     0     --   27726.22         --
```

Queue summary statistics

Queue statistics are defined as follows:

Name The name of the queue.

Total The total number of loads that have entered the queue.

Cur The number of loads that are currently in the queue.

Average The average number of loads that have been in the queue at the same time.

Capacity The total number of loads allowed in the queue at the same time.

Max The maximum number of loads that have been in the queue at the same time.

Min The minimum number of loads that have been in the queue at the same time.

Util The fraction of the queue’s capacity that loads utilized. (In this example model, utilization statistics are not reported because every queue has infinite capacity.)

Av_Time The average amount of time that each load spent in the queue.

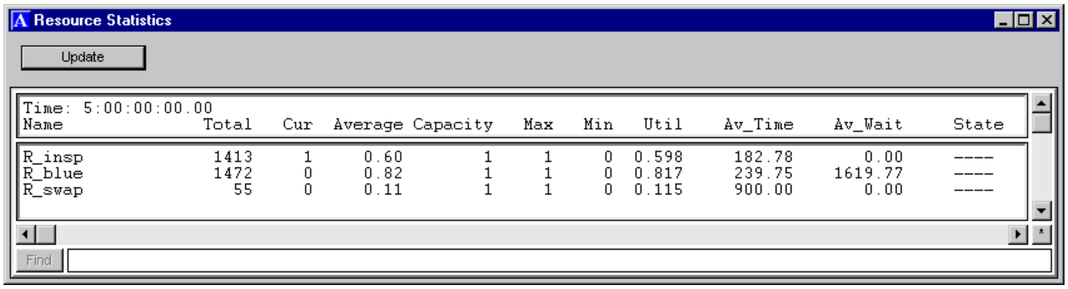
Av_Wait The average time that loads waited to enter a queue; this is the average of all loads, including loads that entered the queue without waiting. (In this example model, Av_Wait statistics are not reported, because every queue has an infinite capacity; an infinite number of loads could enter each queue at the same time, without needing to wait for available space.)

Displaying resource summary statistics

Resources represent machines, operators, tools, fixtures, and other entities that process loads (for more information, see “Resources” on page 5.6 of the “Process System Basics” chapter).

To display a statistic summary for resources:

Step 1 From the Resources menu, ***select*** Statistics Summary. The Resource Statistics window opens, as shown below:



```
Time: 5:00:00:00.00
Name                    Total   Cur  Average Capacity   Max   Min   Util    Av_Time    Av_Wait     State
R_insp                   1413     1     0.60        1     1     0  0.598     182.78       0.00     ----
R_blue                   1472     0     0.82        1     1     0  0.817     239.75    1619.77     ----
R_swap                     55     0     0.11        1     1     0  0.115     900.00       0.00     ----
```

Resource summary statistics

Resource statistics are defined as follows:

Name The name of the resource.

Total The total number of loads that have claimed the resource.

Tip You can calculate the number of loads that have *completed* using the resource by subtracting the loads that are currently (Cur) using the resource from the Total.

Cur The number of loads that are currently using the resource.

Average The average number of loads that used the resource at the same time.

Capacity The maximum number of loads allowed to use the resource at the same time.

Max The maximum number of loads that used the resource at the same time.

Min The minimum number of loads that used the resource at the same time.

Util The fraction of the resource’s capacity that loads utilized.

Av_Time The average time a load spent using the resource.

Av_Wait The average time a load waited to use the resource. This is the average of all loads that claimed the resource, including loads that claimed the resource without waiting.

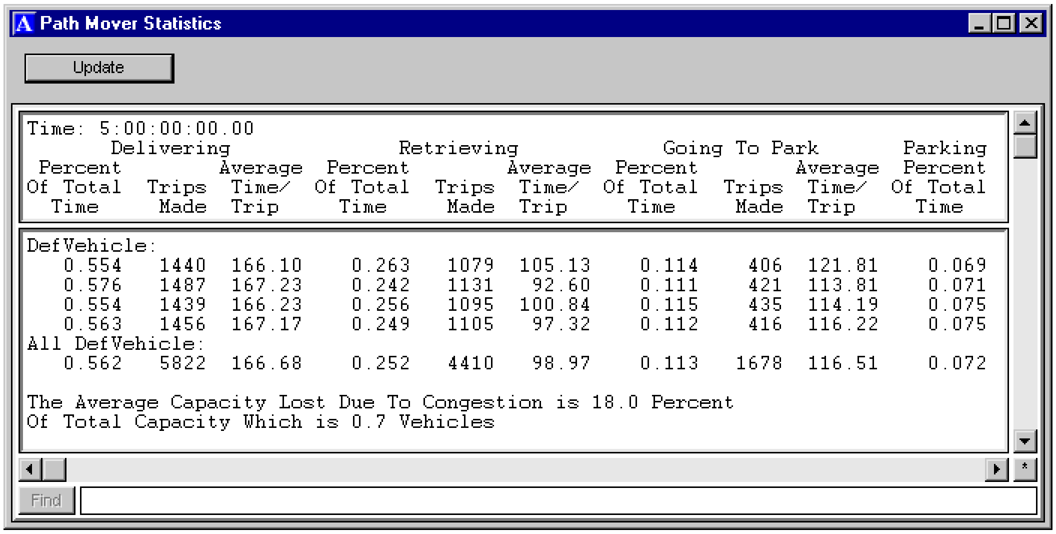
State The name of the current state of the resource.

Resources that have utilization close to 100 percent could indicate bottlenecks. Notice that utilization of resources is reasonable in this model. Battery swapping does not happen very often, which is why the resource R_swap’s utilization is low.

Displaying vehicle statistics

To display statistics for vehicles:

Step 1 From the Path Mover menu, ***select*** Vehicles. The Path Mover Statistics window opens, as shown below:



```
Time: 5:00:00:00.00
       Delivering              Retrieving            Going To Park       Parking
 Percent        Average  Percent        Average  Percent        Average  Percent
Of Total  Trips  Time/  Of Total  Trips  Time/  Of Total  Trips  Time/  Of Total
  Time     Made  Trip     Time     Made  Trip     Time     Made  Trip     Time
DefVehicle:
   0.554   1440  166.10    0.263   1079  105.13    0.114    406  121.81    0.069
   0.576   1487  167.23    0.242   1131   92.60    0.111    421  113.81    0.071
   0.554   1439  166.23    0.256   1095  100.84    0.115    435  114.19    0.075
   0.563   1456  167.17    0.249   1105   97.32    0.112    416  116.22    0.075
All DefVehicle:
   0.562   5822  166.68    0.252   4410   98.97    0.113   1678  116.51    0.072

The Average Capacity Lost Due To Congestion is 18.0 Percent
Of Total Capacity Which is 0.7 Vehicles
```

Vehicle statistics

Vehicle statistics are divided into four categories:

- Delivering
- Retrieving
- Going to park
- Parking

The categories list the following statistics for each vehicle and for all vehicles:

Percent Of Total Time The percent of total simulation time that vehicles spent delivering loads, retrieving loads, going to park, or parking.

Trips Made The number of trips made to deliver loads, retrieve loads, or go to park. (This statistic is not available for parking.)

Average Time/ Trip The average time spent per trip delivering loads, retrieving loads, or going to park. (This statistic is not available for parking.)

Vehicle congestion statistics

Vehicle statistics also provide the average capacity and total capacity lost due to vehicle congestion.

Average Capacity Lost The percentage of vehicle capacity lost due to congestion. The statistic represents the percent of time that vehicles were attempting to move but were blocked (for example, because they were waiting behind a stopped vehicle on the path).

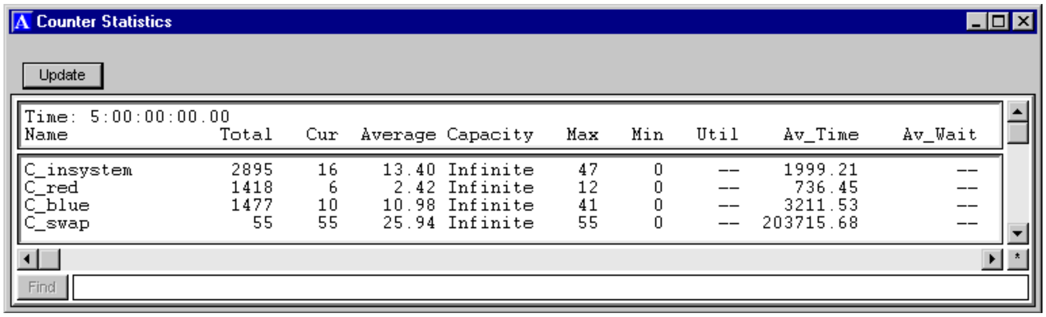
Total Capacity The total vehicle capacity lost due to congestion. This statistic represents the total amount of vehicle capacity that was available, but was not used because vehicles were blocked.

Displaying counter summary statistics

Counters are user-defined entities for tracking an integer value that increases or decreases during a simulation (for more information, see “Collecting custom statistics and controlling capacity with counters” on page 14.4 of the “Additional Features” chapter). In this example model, counters are used to track the total number of all loads in the system (C_insystem), the total number of red loads in the system (C_red), the total number of blue loads in the system (C_blue), and the total number of batteries that have been replaced (C_swap).

To display a statistics summary for counters:

Step 1 From the Counters menu, ***select*** Statistics Summary. The Counter Statistics window opens, as shown below:



```
Time:  5:00:00:00.00
Name                  Total    Cur  Average Capacity   Max    Min    Util     Av_Time     Av_Wait

C_insystem             2895     16    13.40 Infinite    47      0      --     1999.21          --
C_red                  1418      6     2.42 Infinite    12      0      --      736.45          --
C_blue                 1477     10    10.98 Infinite    41      0      --     3211.53          --
C_swap                   55     55    25.94 Infinite    55      0      --   203715.68          --
```

Counter summary statistics

Counter statistics in this model are defined as follows:

Name The name of the counter.

Total The total number of loads created or batteries replaced.

Cur The current number of loads in the system or batteries replaced.

Average The average number of loads in the system or batteries replaced.

Capacity The total number of loads allowed to claim the counter at the same time. The counters in this model have infinite capacity.

Max The maximum number of loads in the system at the same time or the maximum number of batteries replaced.

Min The minimum number of loads in the system at the same time or the minimum number of batteries replaced.

Util The fraction of the counter’s capacity that was utilized at the same time. In this model, counter utilization statistics are not reported because every counter has an infinite capacity, so loads do not need to wait to claim a counter.

Av_Time The average time that loads spent in the system or the average time between battery swaps.

Av_Wait The average time that a load waited to claim the counter. In this model, counter Av_Wait statistics are not reported because every counter has an infinite capacity.

You can define counters to track any integer value. Depending on what you are using the counters for, the statistics might mean different things than they do in this model.

As you can see, AutoMod provides a lot of statistical information. You will have an opportunity to examine model statistics again when you complete the exercises at the end of the chapter.

Closing the AutoMod software

To close the AutoMod software, you need to close both the simulation environment and the edit environment.

Closing the simulation environment

To close the simulation environment and return to the edit environment:

Step 1 From the Control menu in the Simulation window, ***select*** Edit Model. The simulation environment closes and the edit environment opens.

Closing the edit environment

To close the edit environment and quit the AutoMod software:

Step 1 From the Model menu in the Work Area window, ***select*** Quit.

or

Step 1 ***Click*** ☒ the close box.

Now you are going to make a copy of the example model so you can experiment with it.

Copying a model

Before continuing, create a working copy of the example model for this chapter. You will need the original example model to complete the exercises at the end of the chapter, so do not edit the original model; only edit your copy.

To copy the example model for this chapter:

Step 1 In Windows, ***navigate*** to the “demos\gswa” directory of the AutoMod installation.

Step 2 ***Select*** the example model’s parent directory. The parent directory of the example model for chapter 2 is named “examp02_1.”

Step 3 From the Edit menu, ***Select*** Copy.

Step 4 ***Navigate*** to a new destination directory where you want to place the copy of the model and ***select*** Paste from the Edit menu. A copy of the directory appears.

Important ⚠ AutoMod models should always reside in their own directory. If you have more than one model in the same directory, output from one model could overwrite the output from another model.

Step 5 ***Rename*** the copied directory to prevent you from confusing the original model with the copied model. For example, rename the copied directory “examp21copy.”

Editing a model

You are now ready to edit the copy of the example model. To open the model for editing:

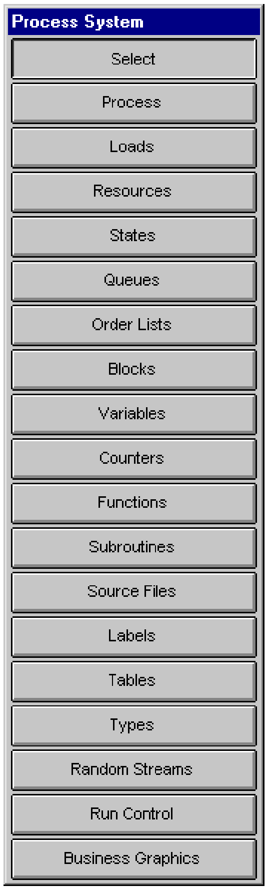
Step 1 ***Double-click*** the directory that you copied in the previous section.

Step 2 ***Double-click*** the model's .arc directory ("examp21.arc").

Step 3 ***Double-click*** the "model.amo" file. The AutoMod software opens and the copied model is imported.

You are now ready to make changes to the model.

AutoMod models can contain several systems. For example, a process system, where the model logic is defined, and one or more movement systems, where loads are transported from one location to another. Each type of movement system, such as a conveyor or path mover system, is defined separately. In this example model, there are two systems: the process system and a path mover system. A path mover system is a vehicle system in which vehicles and paths are defined. When the model opens, the process system is opened by default and the Process System palette is displayed, as shown below:



Process system palette

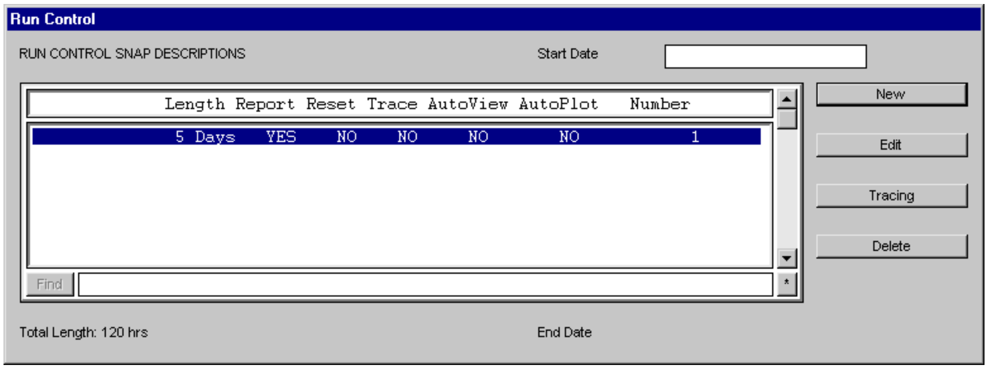
You will use the palette to make changes in the process system.

Changing the length of a simulation

The first thing you will edit in this model is the length of the simulation. You need to make the run shorter.

To change the length of the simulation:

Step 1 ***Click*** Run Control on the Process System palette. The Run Control window opens, as shown below:



Run Control

RUN CONTROL SNAP DESCRIPTIONS

Start Date

```
Length Report Reset Trace AutoView AutoPlot   Number
5 Days  YES    NO    NO     NO       NO          1
```

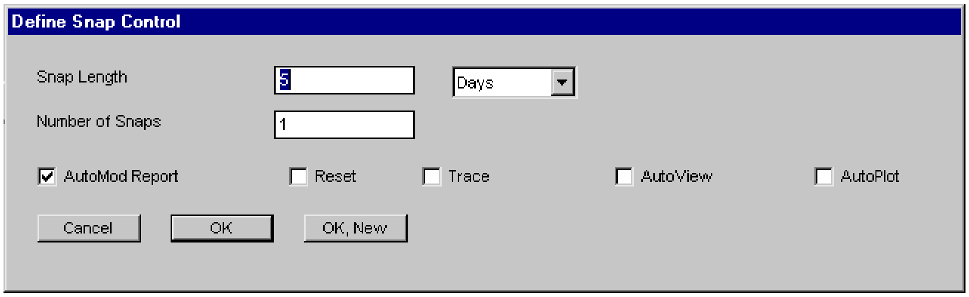
Total Length: 120 hrs

End Date

Run Control window

Currently, the run length is five days, as defined by the snap. A **snap** is a period of time after which statistics are written to reports. The length of the simulation depends on the length of all defined snaps. You will only define one snap for the models in this textbook.

Step 2 ***Select*** the snap, then ***click*** Edit. The Define Snap Control window opens.



Define Snap Control

Snap Length: 5 Days

Number of Snaps: 1

☑ AutoMod Report ☐ Reset ☐ Trace ☐ AutoView ☐ AutoPlot

Define Snap Control window

Step 3 In the Snap Length text box, ***type*** “2.5” to reduce the snap length to 2.5 days.

Step 4 ***Click*** OK to close the Define Snap Control window.

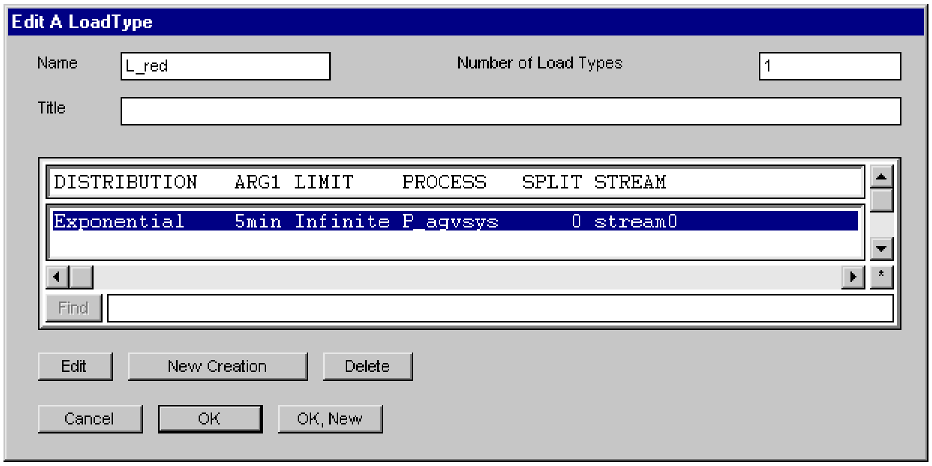
Changing the load creation rate

Now increase the rate at which red loads arrive in the system.

To change the creation rate of red loads:

Step 1 ***Click*** Loads on the Process System palette. The Loads window opens.

Step 2 In the Load Types select list, ***select*** L_red, then ***click*** Edit. The Edit A Load Type window opens, as shown below:



Edit A LoadType

Name: L_red

Number of Load Types: 1

Title:

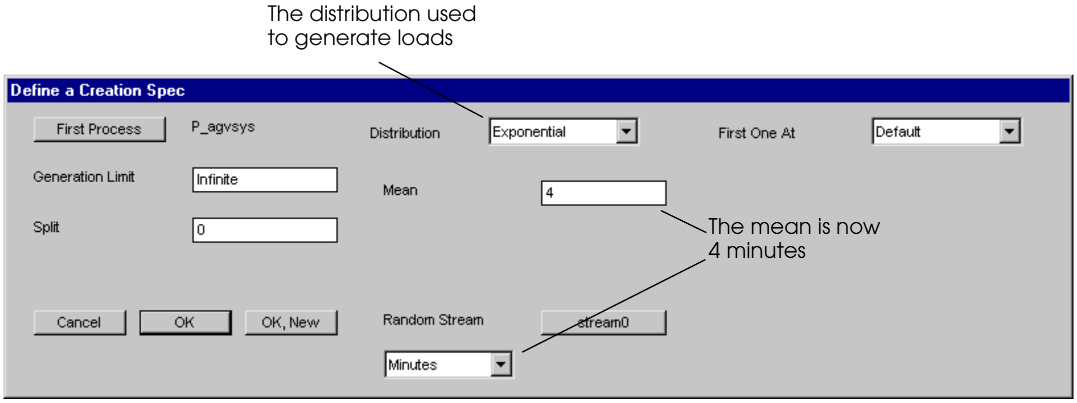
```
DISTRIBUTION   ARG1 LIMIT    PROCESS   SPLIT STREAM
Exponential    5min Infinite P_agvsys      0 stream0
```

Edit A Load Type window

In the window, there is one load creation defined. We can see that red loads are being created according to an exponential distribution with a mean of 5 minutes. The limit is "Infinite," indicating that loads are created continuously until the simulation ends.

Step 3 ***Select*** the load creation, then ***click*** Edit. The Define a Creation Spec window opens.

Step 4 In the Mean text box, ***change*** the "5" to a "4," as shown below:



The distribution used to generate loads

Define a Creation Spec

First Process: P_agvsys

Distribution: Exponential

First One At: Default

Generation Limit: Infinite

Mean: 4

Split: 0

Random Stream: stream0

Minutes

The mean is now 4 minutes

Define a Creation Spec window

Note It is possible to change the distribution used to generate the loads, although you do not want to do so for this model. Distributions are discussed in more detail in chapter 4, "Introduction to AutoMod Syntax."

Step 5 ***Click*** OK to close the Define a Creation Spec window.

Step 6 ***Click*** OK to close the Edit A Load Type window.

You are now ready to export (save) and run the model.

Exporting a model

At this point, save your changes by exporting the model. **Exporting** a model does the following:

- Saves the model.
- Creates an archive of the model, which allows you to move the model from one computer to another.

Tip ☞ Exporting in AutoMod is equivalent to saving in other applications. Therefore, when you are making changes to a model, export frequently.

Step 1 From the Model menu, ***select*** Export.

Step 2 ***Click*** Yes twice to confirm that you want to export the model.

Running the revised model

To verify the effect of the shorter run length and the increase in the number of red loads entering the system:

Step 1 ***Run*** the model (see “Running a model” on page 2.7 if you need more information).

The arrival rate for red loads has increased. The run length has been decreased, as well. If you turn off the animation and let the simulation run to completion, you will see that the simulation stops after 2 days and 12 hours (instead of the previous length of 5 days).

Once the simulation is complete and you have looked at the results, edit the model:

Step 2 ***Select*** Edit Model from the Control menu to return to the edit environment.

Now you are ready to take your first look at the model logic.

Editing a source file

Source files are text files in which you write the processes, or instructions, for a model. Processes are written using the AutoMod language, which is discussed later in this textbook.

In this example, you will edit the model's source file to lengthen the amount of time that it takes the inspector to inspect red loads.

Step 1 ***Click*** Source Files on the Process System palette. The Source Files window opens.

Step 2 ***Select*** logic.m in the Source Files select list, then ***click*** Edit. The AutoMod editor, BEdit, opens to display the model logic.

The syntax for line 11 is:

```
use R_insp for exponential 3 min
```

which means "Use the inspector for a time that is exponentially distributed with a mean of three minutes." We will comment this line of code and replace it with a line that includes a new inspection time.

Commenting model logic

You can add **comments** to model logic using the symbols `/*` at the beginning of the comment and `*/` at the end of the comment. Comments can span multiple lines of text. Any text between the comment symbols is *not* executed when the model is run. Commenting logic is useful for providing explanations of the model logic or for making temporary changes (without deleting logic).

To comment the current logic for line 11 and write a new line that lengthens the inspection time, do the following:

Step 1 Edit line 11 of the model logic by ***typing*** `/*` at the beginning of the line and `*/` at the end of the line, as shown below:

```
/* use R_insp for exponential 3 min */
```

Step 2 ***Position*** the cursor at the end of line 11, then ***press*** Enter to insert a new line.

Step 3 On the new line, ***type***:

```
use R_insp for exponential 4 min
```

The inspection process will now take a time that is exponentially distributed with a mean of four minutes (instead of three).

Step 4 From the File menu, ***select*** Save and Quit. BEdit closes.

The next change you need to make to the model is to decrease the number of vehicles in the simulation from four to three.

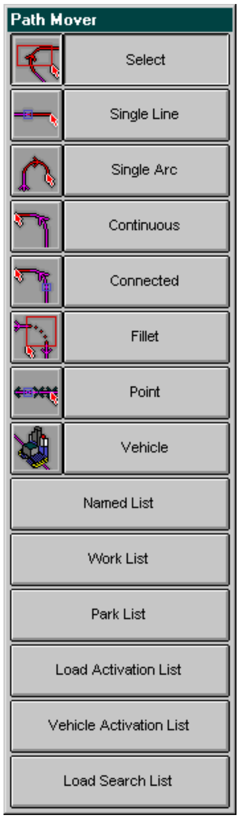
Opening a path mover system

To make changes to the paths or vehicles in the model, you must open the path mover system.

To open the path mover system:

Step 1 From the System menu, ***select*** Open. The Open A System window opens.

Step 2 In the System select list, ***select*** "pm" (for path mover), then ***click*** Open. The path mover system opens and the Path Mover palette appears, as shown below:



Path Mover palette

You will use the palette to make changes in the path mover system.

Changing the number of vehicles

The number of vehicles in the model is currently four.

To change the number of vehicles from four to three:

Step 1 ***Click*** Vehicle on the Path Mover palette. The Vehicles window opens.

Step 2 In the Vehicles select list, ***select*** DefVehicle, then ***click*** Edit. The Edit A Vehicle Definition window opens.

Step 3 In the Number of Vehicles text box, ***change*** the "4" to a "3".

Step 4 ***Click*** Done to close the Edit A Vehicle Definition window.

Step 5 ***Export*** and ***run*** the model.

Look at the area in the middle of the layout labeled "All loads arrive here." As the simulation continues, the stack of loads in this queue grows quickly, because the inspection times have increased and the number of vehicles has decreased, creating delays in the system.

Summary

After completing this chapter, you now know the basic steps for running a simulation:

Step 1 ***Import*** a model in the AutoMod software.

Step 2 ***Edit*** the model (if desired).

Step 3 ***Export*** the model to save changes.

Step 4 ***Build*** and ***run*** the model.

Step 5 ***Continue*** the simulation.

Tip ☞ ***Turn off*** the animation or ***increase*** the display step to accelerate the simulation.

Step 6 ***Display*** simulation statistics.

You have also learned how to change views in the AutoMod software and how to make some basic changes to a model, including commenting and editing the model’s logic.

Exercises

Important

The following exercises ask you to copy example model 2.1; **copy the original example model (not the copied model that you edited while reading the chapter)**. Remember to export the model after each assignment.

Exercise 2.1

Copy example model 2.1 to a new directory. Run the copied model and record the maximum number of loads in process P_agvsys. Edit the copied model. Change the number of AGVs to three.

Edit the source file logic.m. Change line 11 from `use R_insp for exponential 3 min` to `use R_insp for exponential 2 min` so that the mean service time of red loads is two minutes.

Run the model using the changed values. What is the difference in the maximum number of loads in process P_agvsys between the runs?

Exercise 2.2

Copy example model 2.1 to a new directory and edit the source file logic.m. Notice that on line 49, the time between battery swaps is normally distributed with a mean of 480 minutes and a standard deviation of 60 minutes (`wait for normal 480, 60 min`). On line 51, the swap time is a constant 15 minutes (`use R_swap for 15 min`). Run the model. View the resource statistics and record the Total and Av_Time for the resource R_swap with this configuration.

Edit the source file and change the time between battery swaps to a time that is normally distributed with a mean of 240 minutes and a standard deviation of 30 minutes (`wait for normal 240, 30 min`). Change the time required to replace the batteries to a constant 7.5 minutes (`use R_swap for 7.5 min`)

Run the model using the changed values and record the resource R_swap's Total and Av_Time statistics for the new configuration.

Exercise 2.3

Copy example model 2.1 to a new directory. Run the copied model and record the average time that loads spend in process P_agvsys.

Edit the copied model and comment out procedure P_swap (lines 44 to 56). Run the model.

What is the difference in the average time that loads spend in process P_agvsys between the two simulations?

Exercise 2.4

Copy example model 2.1 to a new directory. Run the copied model and record the maximum number of loads in the queue Q_entry.

Edit the copied model and change the interarrival time of part L_blue to a constant 5 minutes. Run the model.

What is the difference in the maximum number of loads in queue Q_entry between the two simulations?

Exercise 2.5

Copy example model 2.1 to a new directory. Run the model for 25 days.

What is the maximum number of blue loads in the system during the 25 day simulation?

Chapter 3

AutoMod Concepts

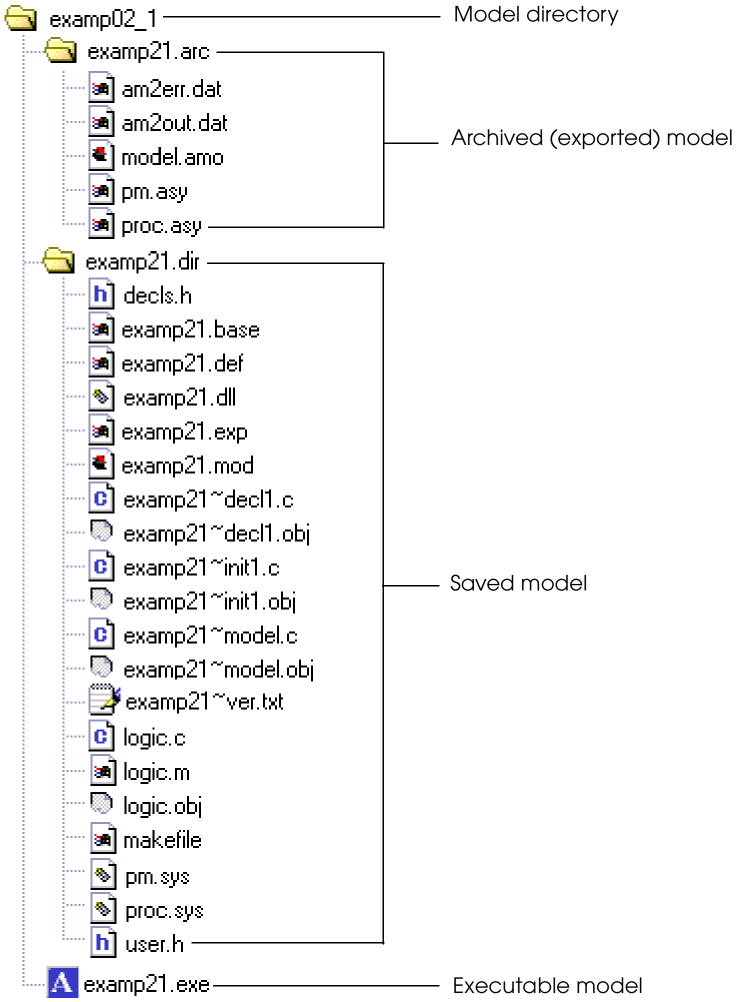
Chapter 3

AutoMod Concepts

In chapter 2, “Using the Software,” you gained some hands-on experience using the AutoMod software. This chapter provides you with an overview of the concepts needed to understand how the AutoMod software works and how it can be used as an effective tool for creating your own simulation models. The concepts discussed in this chapter include: the AutoMod file structure, systems, loads, territories, and archiving models and submitting them to your instructor.

The AutoMod file system

In order to work with example models and complete the assignments in this textbook, it is necessary to understand the files and directories that get created for each AutoMod model. The illustration below shows a model directory, which contains an exported model, a saved model, and an executable (built) model:



```
examp02_1
    examp21.arc
        am2err.dat
        am2out.dat
        model.amo
        pm.asy
        proc.asy
    examp21.dir
        decls.h
        examp21.base
        examp21.def
        examp21.dll
        examp21.exp
        examp21.mod
        examp21~decl1.c
        examp21~decl1.obj
        examp21~init1.c
        examp21~init1.obj
        examp21~model.c
        examp21~model.obj
        examp21~ver.txt
        logic.c
        logic.m
        logic.obj
        makefile
        pm.sys
        proc.sys
        user.h
    examp21.exe
```

An AutoMod model's file hierarchy

The **model directory** is a folder that you create to store a model; you can give the directory any name that you want.

Important Each model must be stored in its own model directory. If you store more than one model in the same model directory, the output from one model will overwrite the output from another model.

The model files are stored in a directory that has one of two extensions:

.arc An archived, ASCII text representation of a model; the <modelname>.arc directory is created when you *export* a model.

.dir A binary version of a model; the <modelname>.dir directory is created when you *save* or *export* a model.

Important Depending on your operating system, the model name may need to be DOS compatible (that is, no more than eight alphanumeric characters with no spaces or special characters). All of the example models that are installed with the AutoMod software are archived and have DOS-compatible names.

Exporting versus saving a model

There are two ways to “save” a model in AutoMod: exporting and saving.

Exporting a model creates an archived copy of the model. Think of an exported model as a backup that you frequently update throughout the process of building the model. The exported model is also the distributable version that you can send to other people when you are finished with the model.

Note

When you export a model, the model is also automatically saved. Therefore, develop the habit of exporting your model often, just as you would save often in other applications.

When you save a model, a version-specific copy of the model is stored on your machine. Think of a saved model as a working version of the model. When you make changes that you are not certain you want to keep, you can save and run the model to test the effect of the changes. If you decide you do not want to keep the changes, you can revert to the last exported version by importing the archived model.

Archived (exported) models

An **archived model** consists of a <modelname>.arc directory and its contents. The directory and the files it contains are created automatically when you export a model from within the AutoMod software.

To open an exported model within the AutoMod software, **import** the “model.amo” file.

Archived models have two advantages over saved models. First, archived models are smaller than saved models. Second, archived models provide upward compatibility with newer versions of the AutoMod software and can be moved to another computer. Therefore, whenever you need to need to send a model to another party (such as your instructor) or copy a model to a different computer, you should use the archived version of the model.

Saved models

A **saved model** consists of a <modelname>.dir directory and its contents. The directory and many of the model files it contains are created automatically when you save or export a model from within the AutoMod software.

To use a saved model within the AutoMod software, **open** the “<modelname>.mod” file.

Saved models are version-specific; they can only be opened using the same version of the AutoMod software that created them.

Executable models

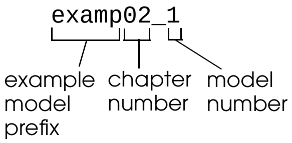
An **executable model** consists of a <modelname>.exe file and a saved model. When you build or run a model within the AutoMod software, the saved model expands. Several additional files are automatically created in the <modelname>.dir directory, and an executable file is created in the model directory. Because of the size of an executable model, you cannot run most models on a diskette; the models must be saved on a hard drive.

Tip

You can run a model by double-clicking on the <modelname>.exe file.

Using the example models

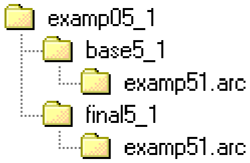
Several example models are installed with the AutoMod software for use with this textbook. An archived version of these models can be found in the “demos\gswa” directory in the AutoMod installation directory. The model directories are named using the following convention:



Example model naming convention

The name above indicates that the directory contains the first example model in chapter 2.

In some cases, we may ask you to make changes to an example model as you read through a chapter. In these cases, the model directory contains a “base” version of the model and a “final” version of the model, both in separate subdirectories, as shown below:



examp05_1
base5_1
examp51.arc
final5_1
examp51.arc

“Base” and “final” versions of an example model

The **base** version of a model is the version you should copy and modify as you read the chapter.

The **final** version of a model is the version you should copy and use for the exercises at the end of the chapter. The chapter modifications have already been made in the final version of the example model.

Important ⚠

Always copy an example model before using it. If you accidentally make changes to an original model, you will need to reinstall the AutoMod software to restore the example model.

Systems

AutoMod models can contain several **systems**, or collections of entities. AutoMod models must have one process system, and can optionally have one or more movement systems.

The process system

The **process system** defines the model logic that controls how products (**loads**) are processed in a model. There can be only one process system in each model. However, the process system can contain an infinite number of processes.

A **process** is a logical subsystem that defines a set of activities for loads during a simulation. Loads are sent to a process to perform one or more actions, such as using a resource or moving into a queue. The actions that a load performs when sent to a process are defined in the process' **arriving procedure,** which is a "to do" list for loads.

For example, a model might contain the following processes:

Process name	Description
P_inspection	An inspection process.
P_paint	A paint shop process.
P_grinding	A grinding operation.

Note ✎ We recommend beginning each process name with "P_". For a complete list of entity naming conventions, see "Entity naming conventions" on page 3.12.

Each of the processes in a model has an arriving procedure that tells a load how long each step of the procedure takes, which resources the load needs to use, where the load needs to travel, and so on. The logic for each arriving procedure is defined in a source file (for more information about source files, see "Source files" on page 3.11).

There can be multiple loads in a process at the same time, and multiple loads may have actions they need to do at the same instant in simulated time.

Individual processes have no physical constraints or graphical representation in the AutoMod software; their purpose is solely to provide logical control of products (loads).

The process system also allows you to define many other entities within a model, including resources (machines or operators) and queues (waiting lines).

Movement systems

A **movement system** contains components that can be used to simulate the transportation of loads, such as the components of a **conveyor system**. A **path mover** system in AutoMod is a flexible path-based system that can be used to simulate automated guided vehicles (AGVs), fork trucks, or people who carry loads through a facility along a predetermined path.

Note

In the first few chapters of this text, you will use and create only the process system in your models. Beginning in chapter 6, “Introduction to Conveyors,” you will learn how to create conveyor systems. After that, you will learn how to create path mover systems, as well.

You can have any number and combination of movement systems in a model. For example, a model could contain a process system and two conveyor systems, or a process system, one conveyor system, and one path mover system.

This text discusses only two movement systems, conveyors and path movers, because they are the only movement systems available in the student version of AutoMod. However, the fully-licensed version of the AutoMod software has other movement systems, including:

- Power & free (widely used in the automotive industry)
- AS/RS (Automated Storage/Retrieval systems)
- Bridge cranes
- Kinematics (used for robotic modeling)
- Tanks & pipes

You can also create static systems, which add stationary graphics to a model (for example, graphics imported from a CAD drawing). Static systems add graphical realism to a model.

System naming conventions

When naming systems in a model, we recommend using the following naming conventions:

System type	System name
Process system	proc
Conveyor system	conv
Path mover system	pm

If you define multiple movement systems of the same type, such as two conveyor systems, you can enumerate them to distinguish them (such as conv1 and conv2, for example).

Using naming conventions ensures consistency among models and makes systems easily recognizable. All of the example models for this textbook are named using these conventions.

Loads

Loads are used to represent physical entities that move through a system, such as products or people. *Loads are the active entity in the AutoMod software, meaning that they execute logic and cause events to happen.* When writing logic, you write a set of instructions for loads to follow.

Loads must be created in each model. You will learn how to define a **creation specification**, which causes loads (products) of a certain type to enter the model at a certain rate. For example, you may define a specification that starts 10 product A's per day into the system. The creation specification also defines which process the loads go to first. For example, product A's begin at P_sort, while product B's begin at P_sand.

After loads are created, they move logically from one process to another, executing actions for the process, which are contained in arriving procedures. When loads are ready to leave the system, they are sent "to die," meaning they disappear from the simulation.

Each load has a user-defined description called a **load type**. Load types contain specifications about a group of loads, such as a product or a group of operators. For example, for a given type of product, the load type determines such characteristics as the product's color, shape, and size. By default, a load is square; you can change the shape as desired, making it larger, making it a rectangle, and so on.

Help

For a complete list of valid load colors, see "color expression" in the AutoMod Syntax Help. (For more information on opening the help file, see "AutoMod Syntax Help" on page 3.11.)

For example, the following load types might be defined in a model:

Load type name	Description
L_init	A logical load that initializes values in the model.
L_productA	A load that represents a manufactured part.
L_productB	A load that represents a different manufactured part.

Note

We recommend beginning each load type name with "L_". For a complete list of entity naming conventions, see "Entity naming conventions" on page 3.12.

Territories and space

Loads occupy physical places known as territories. A **territory** is a physical location where a load is shown graphically. For example, while a load is being processed at a machine, you want the load to be physically (graphically) in front of the machine, so you would move the load into a queue in front of the machine's graphic. While a product is being stored, you want to stack it on a rack, which can also be modeled using a queue. In AutoMod, loads can be located in the following physical territories:

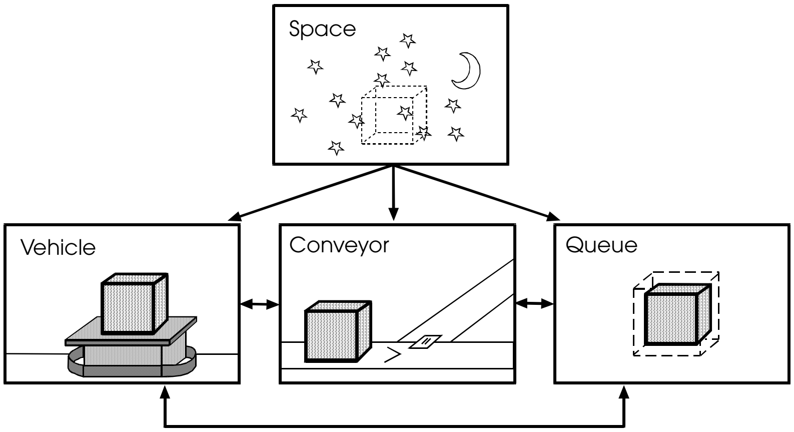
- Queues
- Path mover vehicles
- Conveyor sections

Loads must be in one of these territories at all times from the moment they are created. The first territory that all loads go to is a default queue called **Space**. When you send loads to their first location, such as a queue, they leave Space.

Note

*Loads cannot return to Space***.** Once you move a load into its first location in a process, you are in control of the load's physical location. (If you never move the load out of Space, the load remains there physically throughout the simulation while it executes its processes.)

Loads are not visible in Space. When loads are in a user-defined territory, such as on a vehicle or in a queue, they appear graphically in the model. You move loads between territories through actions in process procedures. Loads can move from any type of territory to any other, as shown below:



Territories in the AutoMod software

If there is no room in a destination territory due to limited capacity, a load must wait in its current territory until it can physically move to its destination. Waiting to move to a new territory delays the load's processing.

Source files

As mentioned in “The process system” on page 3.7, processes contain procedures that describe the manufacturing process. You use the AutoMod syntax to write arriving procedures, or instructions, for loads.You write arriving procedures in a **source file**, which is a text file that is part of your model.

Important

Source files *must* be named with a “.m” extension.

In each of the example models for this text, the model logic is saved in a source file named “logic.m.”

AutoMod syntax

An example of an arriving procedure for a sorting process, written using AutoMod syntax, is shown below:

```
begin P_sort arriving procedure
   use R_sorter for exponential 8 min
   send to die
end
```

Procedures begin with the syntax `begin` and end with the syntax `end`. The process, for which this procedure is defined, is named P_sort. The words `arriving procedure` after the process name indicate that loads execute this procedure when they arrive at the process.

The second line of the procedure tells loads to use a resource named R_sorter for a time that is exponentially distributed with a mean of 8 minutes. After using the resource, the loads are sent to die (the loads leave the simulation).

Note

The logic shown in all examples of this textbook is indented to help make the logic more readable. Indenting is not required, but it is highly recommended.

You cannot tell by looking at this procedure how loads are being sent to the P_sort process. Loads might be sent to this process from a creation specification, or they may be sent to P_sort from another procedure. You will learn more about sending loads to procedures later in this textbook.

AutoMod Syntax Help

This textbook discusses basic AutoMod syntax requirements as you learn about new concepts. However, if you need help remembering how to write a certain command, you can consult the AutoMod Syntax Help while writing your logic.

Help

The AutoMod Syntax Help contains diagrams that show the required and optional syntax in the AutoMod language, and the order in which syntax must appear.

To open the help system from within the AutoMod software:

Step 1 From the Help menu in the Work Area window, ***select*** Syntax Help. The AutoMod Syntax Help opens.

To open the help system from outside the AutoMod software:

Step 1 From the Start menu, ***select*** AutoMod 9.1 (Student Version) > Documentation > Help > AutoMod Syntax. The index for the AutoMod Syntax Help opens. Use the Contents and Index tabs to locate a topic.

Entity naming conventions

In the logic you have seen so far, the names of entities have always begun with a prefix that consists of a capital letter followed by an underscore (for example, P_sort and R_sorter). Like the system naming conventions discussed previously (see “System naming conventions” on page 3.8), it is important to use conventions when naming model entities. Using naming conventions makes it easier for you and others to understand your model and ensures consistency between models.

Because AutoMod syntax is always lowercase, using capital letters in entity names prevents you from accidentally using a keyword, which is a reserved word in AutoMod syntax, as a name. We recommend using the following prefixes when naming entities (each of these entities is discussed later in this textbook):

Prefix	Entity type
A_	Attribute
B_	Block
C_	Counter
F_	Function
L_	Load type
LBL_	Label
OL_	Order list
P_	Process
Q_	Queue
R_	Resource
RC_	Resource cycle
S_	Subroutine
T_	Table
V_	Variable

Using BEdit

BEdit is the built-in text editor in which you write and edit logic in AutoMod models. In order for your model to work correctly, you must both write logic and define the entities that are used in the logic. You can perform these two actions in either order; you can either define entities first and then reference them in logic, or you can write logic that uses undefined entities, then define the entities when you save and quit the source file.

BEdit prompts you to define all undefined entities when you save and quit a source file. For example, if you created a new model and typed the P_sort arriving procedure shown in “AutoMod syntax” on page 3.11 in BEdit, you would be prompted to define the process P_sort and the resource R_sorter when you saved the source file, because the names are unrecognized syntax.

The order in which you do these two steps is up to you. Referencing an entity before defining it means that you do not need to remember what you named an entity when you are writing the model logic.

However, you may want to define an entity and place its graphics before writing your logic so that you can see a graphical layout of the system as you work. If you define entities first, you can look up the name of an entity that you created earlier (from the Process system palette) by using the Entity Help to bring up a list of all defined Process system entities.

Help

To open the Entity Help, select Entity Help from the Help menu in BEdit.

You may find that there are times when you do the steps in one order and times when you do them in the other order. You will develop a personal style of working after you have built several models on your own.

When you save and close a file in BEdit, the software checks for syntax errors in addition to prompting you to define any undefined entities. If the model logic contains any syntax errors, a window opens with an explanation of the error. You must resolve all syntax errors before you can save and quit the editor.

Tip

If you want to quit the editor without correcting an error (for example, because you need to check the name of an entity in a movement system), you can comment the procedure containing the error (see “Commenting model logic” on page 2.25). Commented logic is *not* checked for errors. When you later return to the source file to correct the error, remember to remove the comment markers so that the logic is used when you run the simulation.

Submitting exercise solutions

If you purchased this textbook for use in a simulation class, you may be required to submit your solutions to exercises at the end of each chapter to an instructor. In later chapters, exercises are solved by creating or editing an AutoMod model. We will discuss two methods for submitting exercise solutions:

- Archiving a model in the Zip file format
- Printing the model logic

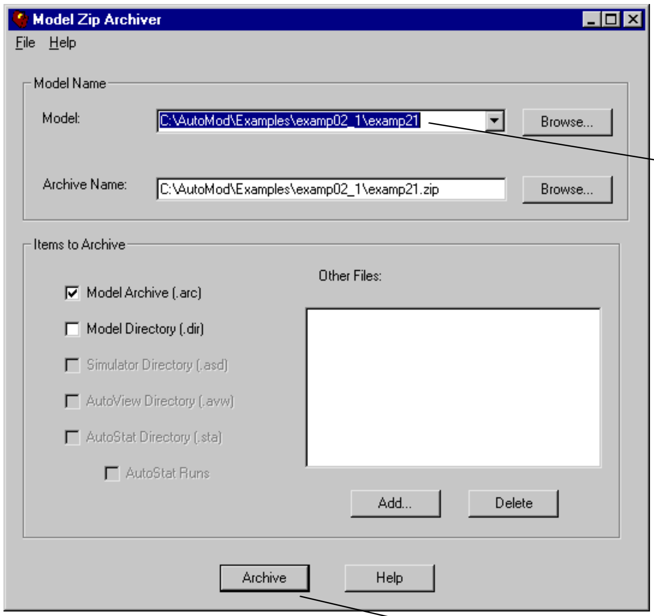
Archiving a model in the Zip format

Archiving your solution models in the Zip file format allows you to compress your models so that they can be submitted on a diskette or emailed to your instructor. The AutoMod software includes a utility for easily creating a zip file containing your solution model.

To compress a model into a Zip file:

Step 1 ***Export*** the model from within the AutoMod software.

Step 2 ***Open*** the Model Zip utility by selecting AutoMod 9.1 (Student Version) > Utilities > Model Zip from the Start menu. The Model Zip Archiver window opens, as shown below:



Model: C:\AutoMod\Examples\examp02_1\examp21

Archive Name: C:\AutoMod\Examples\examp02_1\examp21.zip

☑ Model Archive (.arc)
☐ Model Directory (.dir)
☐ Simulator Directory (.asd)
☐ AutoView Directory (.avw)
☐ AutoStat Directory (.sta)
☐ AutoStat Runs

Select a model in the Model drop-down list

Click Archive to create the zip file

Model Zip Archiver

Step 3 In the Model drop-down list, ***select*** the model you want to submit or ***click*** Browse to locate the desired model.

Step 4 ***Click*** Archive. The Zip file is created and saved in the model directory.

You can now copy the Zip file to a diskette or email the file to your instructor.

Additional information about using the Model Zip Archiver can be accessed from the Help menu in the Model Zip Archiver window.

Printing the model logic

Printing the logic for a model allows you to submit a printout of your model's source file to an instructor. You can print a model's logic by first saving your source file as a text file and then printing the text file from any text editor, such as Notepad or WordPad.

To print the logic for a model:

Step 1 ***Open*** the model.

Step 2 ***Edit*** the model's source file. The file is opened in BEdit.

Step 3 From the File menu, ***select*** Save As. The Save window opens.

Step 4 ***Navigate*** to the location where you want to save the file.

Step 5 ***Name*** the file with a .txt extension (for example, "johndoe02_1.txt").

Step 6 ***Click*** Save. A text file containing the model logic is saved in the defined location.

Step 7 ***Open*** the text file in a text editor and ***print*** the file.

Summary

This chapter provides an overview of some of the basic concepts necessary for using the AutoMod software, including systems, loads, territories, source files, and naming conventions. This chapter also discussed methods for submitting your chapter exercise solutions to an instructor. These concepts are used throughout the rest of this textbook.

Exercises

Exercise 3.1

Answer the following questions:

1. What is the difference between saving and exporting a model?
2. What is the difference between opening and importing a model?
3. How many process systems can a model have?
4. Can a model have more than one conveyor system?
5. Can a person carrying products from one location to another be modeled using a path mover system?
6. What is the limit to the number of processes in a model?
7. Why would you name a process with the prefix P_?
8. What is the correct extension of an AutoMod source file?
9. How many examples of the “use” action are provided in the AutoMod Syntax Help?
10. Is P_init a word that is reserved syntax in the AutoMod language?
11. Can loads be magenta?
12. Can loads be longer than they are wide?
13. In what territory do loads begin?
14. Is it possible to move from the territory of a queue to the territory of Space?
15. Can a load move from the territory of Space to the territory of a conveyor system without first moving into the territory of a queue?

Chapter 4

Introduction to AutoMod Syntax

Chapter 4

Introduction to AutoMod Syntax

This chapter presents the AutoMod simulation language in more detail. The chapter introduces the distributions available for generating random numbers in the AutoMod software, and the syntax for using distributions in processes. The chapter also discusses syntax for writing procedures, defining time delays, getting time values from the simulation clock, printing information, and sending loads to processes. After learning basic AutoMod syntax, you will recreate the example model in this chapter.

Example 4.1: A two-process model

Example model 4.1 is a very basic manufacturing system that can be modeled in two processes. In this hypothetical facility, loads are created with an interarrival time that is exponentially distributed with a mean of 12 minutes. Loads first go to a sorting process, where each load is sorted for a time that is uniformly distributed between 8 and 12 minutes. The sorted loads then go to a processing station that processes loads in three steps for the amount of time shown in the table below:

Step	Time
1 - Preparation	Constant 100 seconds
2 - Processing	Uniformly distributed between 3 and 13 minutes
3 - Unloading	Constant 20 seconds

After the unloading step, loads leave the system.

In the model, after each load is sorted but before it travels to the processing station, a message is printed to the Message window that indicates the load's ID number and type. The current simulation time is also printed whenever a load leaves the system.

The model is defined to run for one eight-hour shift.

Logic for example model 4.1

Recall that the logic in an AutoMod model is executed by loads. Loads are sent to a process and execute that process' arriving procedure. The logic to simulate the example model is shown below:

```
begin P_sort arriving procedure   /* All procedures start with begin */
   wait for uniform 10, 2 min    /* Delay load for 8 to 12 minutes */
   print this load “ was just sorted.” to message
            /* Load ID and text are printed to the Message window */
   send to P_procstation         /* Sends load to another process */
end                              /* All procedures end with an end */

begin P_procstation arriving     /* The word “procedure” is optional */
   wait for 100 sec              /* Time units (seconds) are specified */
   wait for uniform 8, 5 min     /* Delay load for 3 to 13 minutes */
   wait for 20                   /* Default time units are seconds */
   print this load “ left at ” (absolute clock/3600) as .2 “ hours.”
      to message
      /* Arithmetic operations can be performed in many places */
   send to die                   /* The load leaves the system */
end
```

A detailed explanation of the example model's logic is provided throughout the remainder of this chapter.

Arriving procedures

An arriving procedure is a set of instructions at a process that tell loads what to do during that operation. Arriving procedures are performed as soon as a load arrives at the process.

All procedures begin with the syntax `begin` and end with the syntax `end`. The statement

```
begin P_sort arriving procedure
```

includes the name of the procedure, in this case it is P_sort, and the syntax `arriving procedure` to identify the type of procedure that is being defined.

Tip

The syntax `procedure` is optional, so `begin P_sort arriving` is the same as `begin P_sort arriving procedure`.

Loads in example model 4.1 are sent to two processes: P_sort and P_procstation. An arriving procedure is defined for each process.

The wait action

Example model 4.1 simulates the length of time that loads are sorted and processed. The delays are modeled in the P_sort and P_procstation arriving procedures using the `wait` action.

The `wait` action causes a load to pause for a specific period of time. The first `wait` action in the example model is:

```
wait for uniform 10, 2 min    /* Delay load for 8 to 12 minutes */
```

The action causes the load to delay for a uniformly distributed random time between 8 and 12 minutes, all values being equally likely. The uniform distribution is discussed in more detail in the following section.

Distributions

One of the challenges of simulation is determining how to correctly model events that occur randomly in the real world. “Input data” on page 1.18 discusses methods of handling random input data. The examples and exercises in this textbook provide the information you need to model random events using standard distributions within the AutoMod software.

The system in example model 4.1 involves several random events: the time between load creations, the length of time required to sort loads, and the length of time that loads spend processing. The AutoMod software supports several distributions that can be used to generate random numbers; this chapter discusses four of them:

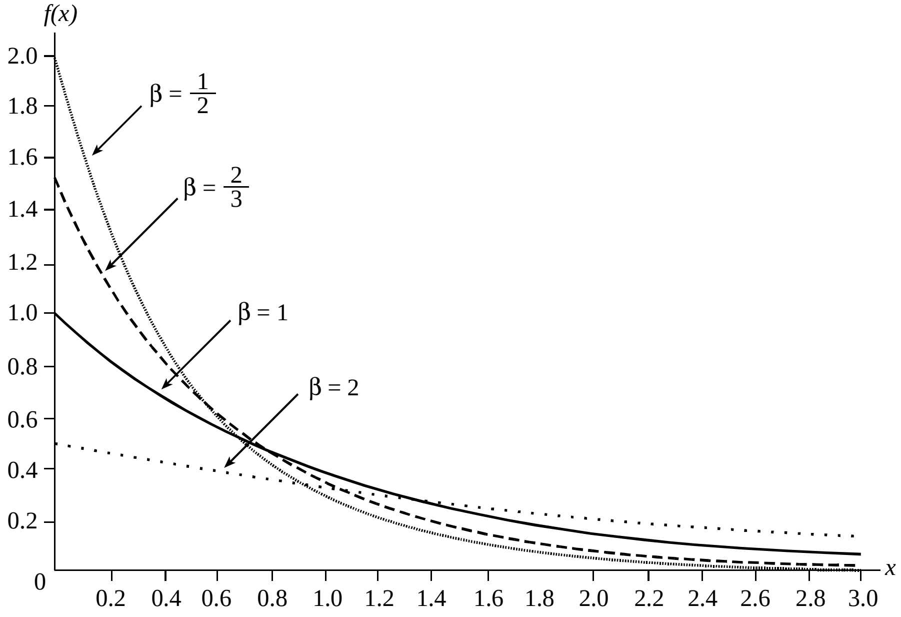
- Exponential
- Normal
- Triangular
- Uniform

This chapter also discusses the constant distribution, which is a non-random distribution.

Exponential distribution

An **exponential distribution** is typically used for modeling completely random events that have high variability.

The range of an exponential distribution is all positive real numbers. The probability of data values varies depending on the distribution’s mean, represented by β in the probability density functions below:



Exponential (β) probability density functions

The exponential distribution has the following properties:

- 63 percent of the data values are less than the mean.
- 37 percent of the data values are greater than the mean.
- 14 percent of the data values are greater than two times than the mean.
- 5 percent of the data values are greater than three times the mean.
- 2 percent of the data values are greater than four times the mean.

The syntax for generating a random number using an exponential distribution in the AutoMod software is:

```
exponential β
```

For example, to cause a load to delay for a time that is exponentially distributed with a mean of 2 minutes, you can use the `wait` action as follows:

```
wait for exponential 2 min
```

Tip ☞

You can abbreviate the syntax `exponential` to its first letter, `e`. For example,

```
wait for e 2 min
```

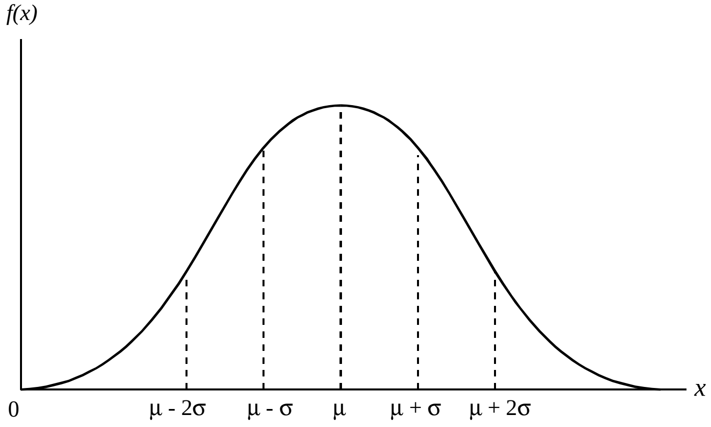
is the same as

```
wait for exponential 2 min
```

Normal distribution

A **normal distribution** is represented by a symmetrical, bell-shaped curve. Normal distributions are typically used for modeling events with limited variability (for example, processing times and repair times). Because, in practice, many of these types of events are described using values that are skewed rather than symmetrical, the normal distribution is the least commonly-used of the four distributions discussed in this chapter.

The range of the normal distribution is all real numbers. Virtually all values fall between plus or minus five standard deviations. If negative values are generated, they are automatically set to zero by the AutoMod software. The probability of data values varies depending on the distribution's mean (μ) and the standard deviation (σ).



Normal (μ, σ) probability density function

The normal curve is symmetrical about its mean, so values are equally likely to occur in the interval ranging from $\mu - \sigma$ to μ as they are from μ to $\mu + \sigma$.

The normal distribution has the following properties:

- 68.27 percent of the data values fall within ($\mu - \sigma$, $\mu + \sigma$).
- 95.45 percent of the data values fall within ($\mu - 2\sigma$, $\mu + 2\sigma$).
- 99.73 percent of the data values fall within ($\mu - 3\sigma$, $\mu + 3\sigma$).
- >99.99 percent of the data values fall within ($\mu - 4\sigma$, $\mu + 4\sigma$).

The syntax for generating a random number using a normal distribution in the AutoMod software is:

```
normal μ, σ
```

For example, to cause a load to delay for a time that is normally distributed with a mean of 60 seconds and a standard deviation of 5 seconds, you can use the `wait` action as follows:

```
wait for normal 60, 5 sec
```

Tip ☞

You can abbreviate the syntax `normal` to its first letter, `n`. For example,

```
wait for n 60, 5 sec
```

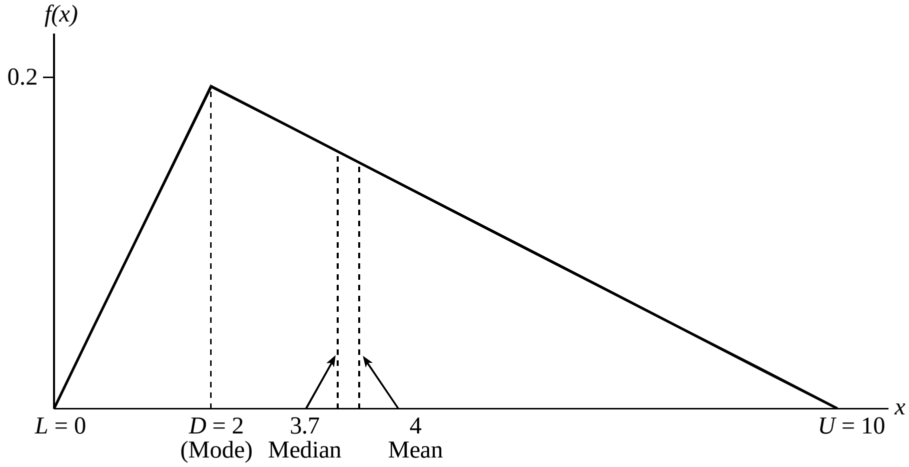
is the same as

```
wait for normal 60, 5 sec
```

Triangular distribution

A **triangular distribution** is used to approximate any unimodal, skewed distribution that has a bounded range. When exact data is not available, but the most likely value and rough estimates of the minimum and maximum values can be obtained, a triangular distribution can provide an approximation for the desired random numbers.

The probability of data values varies depending on the triangular distribution's minimum (lower) value (*L*), most probable value, or mode (*D*), and the maximum (upper) value (*U*). The range of the distribution is from *L* to *U*.



Triangular (0, 2, 10) probability density function

The syntax for generating a random number using a triangular distribution in the AutoMod software is:

```
triangular L, D, U
```

For example, to cause a load to delay for a time that is triangularly distributed with a minimum value of 3 minutes, a mode of 5 minutes, and a maximum value of 10 minutes, you can use the `wait` action as follows:

```
wait for triangular 3, 5, 10 min
```

Tip ☞

You can abbreviate the syntax `triangular` to its first letter, `t`. For example,

```
wait for t 3, 5, 10 min
```

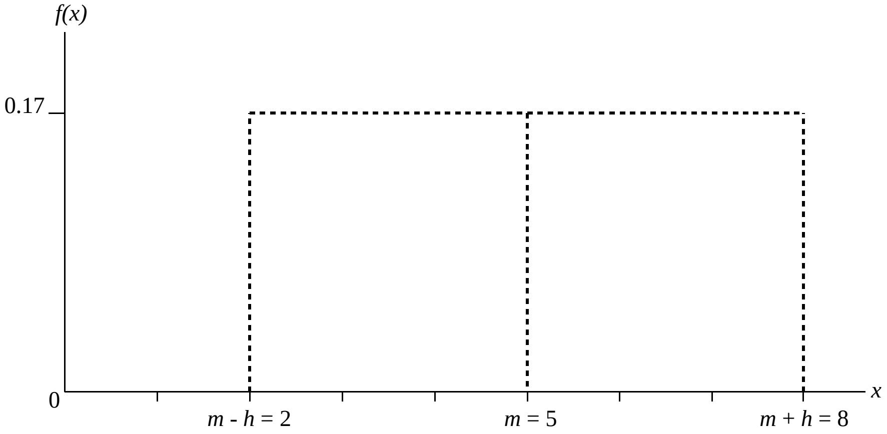
is the same as

```
wait for triangular 3, 5, 10 min
```

Uniform distribution

A **uniform distribution** is used when all values within the range of the distribution are equally probable.

The values that are generated from a uniform distribution depend on the distribution’s mean (*m*) and offset (*h*). The range of the distribution is from (*m* – *h*) to (*m* + *h*), as shown below:



Uniform (5,3) probability density function

The syntax for generating a random number using a normal distribution in the AutoMod software is:

```
uniform m, h
```

For example, to cause a load to delay for a time that is uniformly distributed with a mean of 10 minutes and an offset of 2 minutes, you can use the `wait` action as follows:

```
wait for uniform 10, 2 min
```

The delay has an equal probability of being any value between 8 and 12 minutes.

Tip ☞ You can abbreviate the syntax `uniform` to its first letter, `u`. For example,

```
wait for u 10, 2 min
```

is the same as

```
wait for uniform 10, 2 min
```

Calculating a uniform distribution’s mean and offset

Often, the exercises in this textbook provide a range of values and require you to calculate the mean and offset to use in the distribution. For example, suppose you want to delay a load for a time that is uniformly distributed between 6 and 12 minutes.

To find the *offset*, subtract the minimum value from the maximum value and divide by two:

$$\frac{(12-6)}{2} = 3$$

To find the *mean*, subtract the offset from the maximum value:

$$12 - 3 = 9$$

Thus, the syntax for delaying the load is defined as:

```
wait for uniform 9, 3 min
```

Constant distribution

A **constant distribution** is used when a constant time value is needed to represent a single, recurring event. For example, you may wish to model the morning break for a typical factory worker as a constant value of 15 minutes.

Unlike the other distributions, the syntax for using a constant distribution consists only of the constant value (you do not need to type the name of the distribution). For example, to cause a load to delay for a constant time of 100 seconds, you can use the `wait` action as follows:

```
wait for 100 sec
```

Units of time measurement

The AutoMod software supports several units of time measurement:

AutoMod syntax	Time unit description
`sec`	seconds (default)
`min`	minutes
`hr`	hours
`day`	days

You can use any of these units of time measurement with the wait action to specify the length of the time delay. For example, the syntax

```
wait for 5 hr
```

causes the load that executes the action to wait for 5 hours.

The default time unit is seconds. Therefore, if you do not specify any units of time measurement, the delay is in seconds. For example, the syntax

```
wait for 5
```

causes the load that executes the action to wait for 5 seconds.

Obtaining the current simulation time

Example model 4.1 prints a message containing the current simulation time whenever a load leaves the system. You can access the current simulation time in model logic by using the syntax `absolute clock`, which provides the simulation time in seconds (for example 532.09).

Tip ☞

The syntax `absolute clock` can be abbreviated as `ac`.

Performing mathematical calculations in logic

The AutoMod software supports the standard arithmetic operators for performing mathematical calculations, as shown in the table below:

Arithmetic Operator	Description
–	Subtraction
+	Addition
*	Multiplication
/	Division
%	Modulo division (the result is the absolute value of the remainder, after integer division).

You can perform a mathematical calculation anywhere in the model logic that a numeric value can be used. The order in which operations are performed follows the standard rules of operator precedence (multiplication, division, and modulo division have a higher precedence than addition and subtraction). You can use parentheses to cause operations to be performed in an order other than the default. For example,

```
(2 + 6) * 8
```

evaluates to 64, whereas

```
2 + 6 * 8
```

evaluates to 50.

In example model 4.1, the syntax

```
(absolute clock/3600)
```

causes the current simulation time to be divided by 3600, which converts the time value from seconds to hours (there are 3600 seconds in an hour).

Mathematical calculations are often used when converting time values. For example, to delay loads for a time that is exponentially distributed with a mean of 3 minutes 16 seconds, you can use the following syntax to convert the minutes to seconds:

```
wait for e (3*60)+16 sec
```

The print action

In example model 4.1, messages are printed to the Message window after each load is sorted and whenever a load leaves the system. You can print messages to the Message window during a simulation using the `print` action. By default, the `print` action prints to a file named "modelname.print," which is in your model directory, but you can print to the Message window or to another file, as explained in the following sections.

Printing constant strings

A **constant string**, sometimes called a literal string, is any sequence of alpha-numeric characters enclosed in quotes. The syntax for printing the constant string "Hello world." to the Message window is:

```
print “Hello world.” to message
```

You can print any number of constant strings in the same `print` action. For example,

```
print “This message ” “consists of 3 ” “constant strings.” to message
```

Note

When typing print statements, include spaces as shown or words in the Message window will run together.

Of course, the message above could easily be printed in one constant string. However, when you begin printing other values (for example, mathematical calculations) in a printed message, it will become clear why printing multiple strings is useful (see "Printing the result of a mathematical calculation" on page 4.13 for an example).

Printing a load's ID number and load type

Each load that is created during a simulation is assigned a load ID number. Load ID numbers begin with 1 (for the first load in the simulation) and increment by 1 for each additional load that is created.

You can print the ID number and type of the load that is executing an arriving procedure using the syntax `this load`. In example model 4.1, the ID and type of each load that executes the arriving procedure for process P_sort is printed to the Message window using the following syntax:

```
print this load “ was just sorted.” to message
```

During a simulation, the following message is printed to the Message window when the first load completes sortation:

```
Load 1 (L_loads) was just sorted.
```

The load's ID number is "Load 1" and the load's type is "L_loads" (the load's type is automatically printed in parentheses).

Printing the result of a mathematical calculation

You can print constant numeric values within a constant string. However, to print the result of a mathematical calculation, the calculation must occur outside of a constant string.

For example, the syntax

```
print “There are 2+6 containers” to message
```

prints the text within the quotes exactly as it appears:

```
There are 2+6 containers
```

To perform the mathematical calculation and print the sum, the following syntax is used:

```
print “There are ” 2+6 “ containers” to message
```

Because the calculation occurs outside of the quotes, the following message is printed:

```
There are 8 containers
```

Rounding printed values

When you print a calculation involving numeric values containing decimals, the AutoMod software prints the solution to the sixth decimal place (the solution is rounded if necessary). For example, the syntax

```
print 2.0/3.0 to message
```

prints the following message during a simulation:

```
0.666667
```

You can control how many decimal places are printed using the syntax `as` followed by a decimal and the number of decimal places you want to print. For example, the syntax

```
print 2.0/3.0 as .2 to message
```

prints the following message during a simulation:

```
0.67
```

The quotient is rounded to the second decimal place.

In example model 4.1, the following syntax is used to print a message whenever a load leaves the simulation:

```
print this load “ left at ” (absolute clock/3600) as .2 “ hours.”
   to message
```

When the first load leaves the simulation, the following message is printed:

```
Load 1 (L_loads) left at 0.43 hours.
```

The time value is rounded to the second decimal place.

Printing to a file

You can print to a file during a simulation by replacing the syntax `to message` with the syntax `to` followed by the path and filename of the destination file, enclosed in quotes.

For example to print “Hello world.” to a file named “Hello.txt” on your C drive, use the following syntax:

```
print “Hello world.” to “C:/Hello.txt”
```

Important

Use forward slashes “/” when specifying a path to the destination file.

If the file Hello.txt does not exist on your C drive, the file is created during the simulation. If the file already exists, the file is replaced during the simulation.

The file created by AutoMod is always an ASCII text file, but you can give it any extension. For example, you can name it .dat for data or .rep for report.

Printing to a file in the model’s archive directory

It is often useful to print to a file in the model’s archive (.arc) directory. Printing to the model’s archive directory is advantageous because it makes your model more portable. Because you are using a relative path, you do not need to worry about changing the file location if you run the model on a different computer.

To print to a model’s archive directory, use “arc/” as the file path. For example, the syntax for printing “Hello world” to a text file named “Hello.txt” in the model’s archive directory is shown below:

```
print “Hello world.” to “arc/Hello.txt”
```

The send action

The `send` action is used to send loads from one process to another. The `send` action can also be used to remove loads from a simulation.

Sending loads to a process

To send a load to a process, use the syntax `send to` followed by the name of the process to which you are sending the load.

For example, in example model 4.1, loads are sent from process P_sort to process P_procstation using the following syntax:

```
send to P_procstation
```

When the `send` action is executed, the load leaves process P_sort, enters process P_procstation, and begins executing the arriving procedure for the new process.

Sending loads to die

When loads have completed processing, you can remove them from a simulation using the following syntax:

```
send to die
```

Example model 4.1 uses this syntax to remove loads that have completed processing in process P_procstation.

Creating example model 4.1

Now that you are familiar with the logic in example model 4.1, you are ready to create the example model in the AutoMod software using the following steps:

Step 1 ***Create*** a new model.

Step 2 ***Create*** the process system.

Step 3 ***Write*** the model logic.

Step 4 ***Create*** new loads.

Step 5 ***Define*** the length of the simulation.

Step 6 ***Run*** the model.

Each of these steps is discussed in detail in the following sections.

Creating a new model

To create a new model, do the following:

Step 1 ***Open*** the AutoMod software. The Work Area window opens.

Step 2 From the Model menu, ***select*** New. A navigation window opens.

Step 3 ***Navigate*** to an empty directory in which you want to store the model.

Tip You can create a new directory by clicking 📁 in the navigation window.

Step 4 In the File Name text box, ***type*** “examp41” and ***click*** Save. The new model is created.

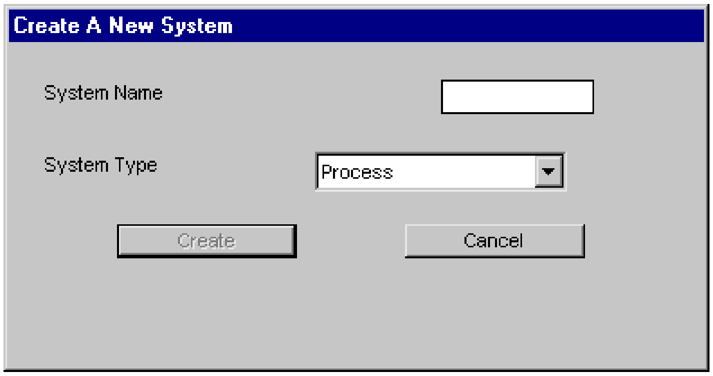
Important Model names cannot contain special characters, except for underscores. A period is automatically added before the extension. For example, you can name a model example4_1, but not example4.1.

You are now ready to create the process system.

Creating the process system

Every model must contain a process system. To create the process system, do the following:

Step 1 ***Select*** New from the System menu. The Create a New System window opens.



Create A New System

System Name

System Type: Process

Create a New System window

Step 2 In the System Name text box, ***type*** “proc” and ***click*** Create. The process system is created and the Process System palette appears.

The next step is to write the model logic in a source file.

Writing the model logic

To write the model logic, you must define a source file:

Step 1 ***Click*** Source Files on the Process System palette. The Source Files window opens.

Step 2 ***Click*** New. The Define a Source File window opens.

Step 3 In the Name text box, ***type*** “logic.m” and ***click*** Edit. The BEdit window opens.

Important ⚠ Source files must end in a .m extension.

Step 4 ***Type*** the logic shown below:

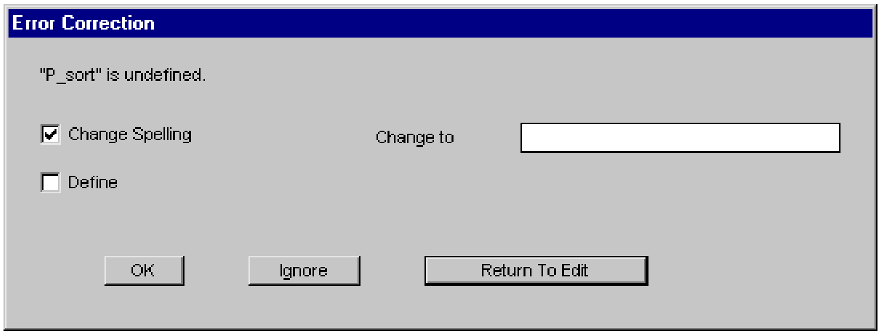
```
begin P_sort arriving procedure  /* All procedures start with begin */
   wait for uniform 10, 2 min    /* Delay load for 8 to 12 minutes */
   print this load “ was just sorted.” to message
             /* Load ID and text are printed to the Message window */
   send to P_procstation         /* Sends load to another process */
end                              /* All procedures end with an end */

begin P_procstation arriving     /* “procedure” syntax is optional*/
   wait for 100 sec              /* Time units (seconds) are specified */
   wait for uniform 8, 5 min     /* Delay load for 3 to 13 minutes */
   wait for 20                   /* Default time units are seconds */
   print this load “ left at ” (absolute clock/3600) as .2 “ hours.”
      to message
      /* Arithmetic operations can be performed in many places */
   send to die                   /* The load leaves the system */
end
```

When you are finished typing the logic, you are ready to save the source file and define the model’s entities.

Defining unknown entities

When you save and quit a source file, BEdit prompts you to define entities that have been named in the model logic, but which have not been defined. In this model, we need to define the processes P_sort and P_procstation. To define the processes, do the following:

Step 1 ***Select*** Save and Quit from the File menu in BEdit. The Error Correction window opens, indicating that P_sort is undefined.



"P_sort" is undefined.

☑ Change Spelling Change to

□ Define

Error Correction window

Step 2 ***Select*** Define. The “Change to” field changes to the “Define as” drop-down list.

Step 3 You want to define P_sort as a process, which is the type selected in the drop-down list, so ***click*** Define As. The Array Size window opens (arrays are discussed later in this textbook).

Step 4 ***Click*** OK to define the process with an array size of 1. The Error Correction window opens, indicating that P_procstation is undefined.

Step 5 ***Repeat*** steps 3 and 4 to define P_procstation as a process. When you are finished, BEdit closes.

Note If you get any other error messages, they are caused by typing errors. To fix typing errors, click the Return To Edit button in the Error Correction window to return to fix the mistakes on the lines specified. When you are finished, select Save and Quit from the File menu.

Step 6 ***Click*** OK to close the Define a Source File window. You have now defined the model logic.

Step 7 ***Export*** your model before continuing.

Creating new loads

Now that you have defined two processes and their arriving procedures, you are ready to define a new load type in the model. In the AutoMod software, you can define as many load types as necessary to distinguish between different types of parts in a system. In this model, we want to process only one type of load in the system.

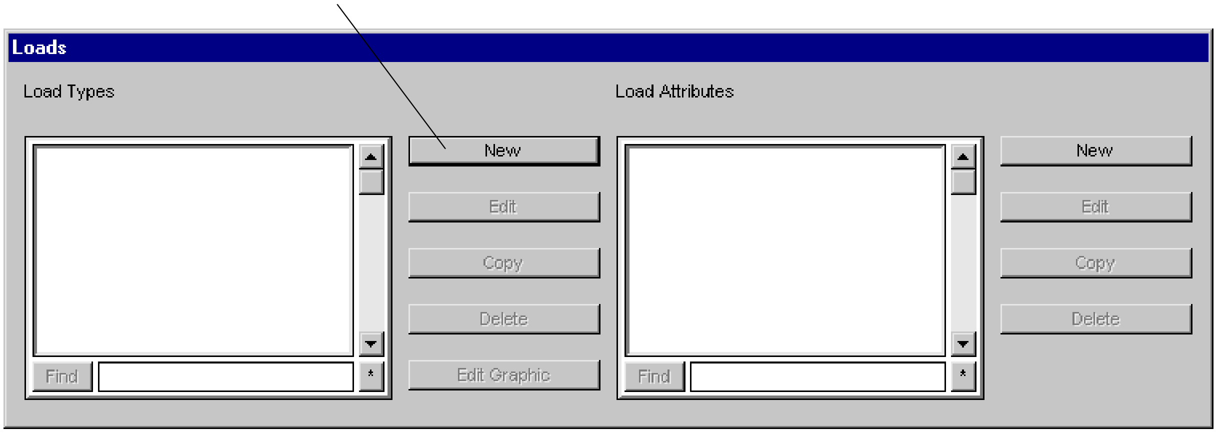
When you define a load type, you can also define a load creation specification that defines the number of loads of that type that are created during a simulation and the time between load arrivals. The load creation specification also specifies which process the loads execute first.

To define the load type and load creation specification, do the following:

Step 1 ***Click*** Loads on the Process System palette. The Loads window opens.

Step 2 To the right of Load Types, ***click*** New to define a new load type, as shown in the window below:

Click New to create a new load type



Loads window

The Define A Load Type window opens.

Step 3 In the Name text box, ***type*** “L_loads”.

You have now defined a new load type named L_loads. Now you need to define a load creation specification to indicate the quantity and arrival rate of the loads in the simulation.

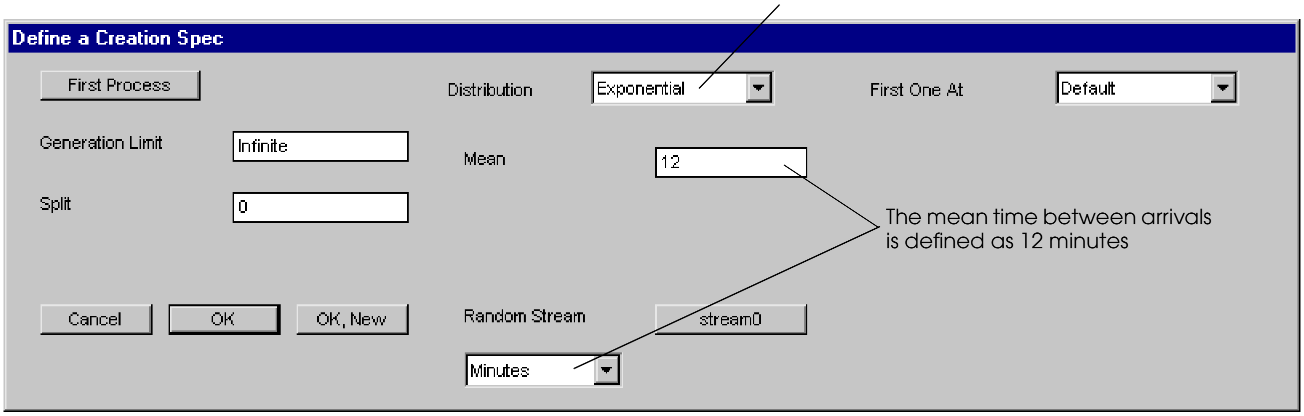
Step 4 To define the load creation specification, ***click*** New Creation. The Define a Creation Spec window opens.

In this model, loads are created with an interarrival time that is exponentially distributed with a mean of 12 minutes.

Step 5 ***Select*** Exponential in the Distribution drop-down list.

Step 6 In the Mean text box, ***type*** "12" and ***select*** minutes in the time units drop-down list, as shown below:

The time between load arrivals is exponentially distributed



Define a Creation Spec

Distribution: Exponential

First One At: Default

Generation Limit: Infinite

Mean: 12

Split: 0

Random Stream: stream0

Minutes

The mean time between arrivals is defined as 12 minutes

Define a Creation Spec window

Step 7 ***Click*** the First Process button to define the first process that loads execute during a simulation. The Pick A First Process window opens.

Step 8 ***Select*** P_sort in the list and ***click*** OK. The name of the first process is displayed in the Define a Creation Spec window.

Step 9 ***Click*** OK to close the Define a Creation Spec window, then ***click*** OK to close the Define A Load Type window. You have now created a new load type that will be generated during the simulation.

Limiting the number of loads created

When you are defining a load creation specification, you can define a **generation limit,** which controls how many loads of that specification are created during the simulation. By default, loads for each specification are created continuously throughout the simulation (there is no limit), so the generation limit is Infinite. However, if you want to create a finite number of loads of a certain type, set the generation limit to the desired value.

Tip

Generation limits are useful for controlling a simulation's length. Rather than specify a run length, you can limit the simulation using a generation limit. If you do not define a run control and you limit the number of loads generated, the model stops automatically once all loads have finished their process procedures. If you do not define a run control, you will get a warning at runtime that no run control is defined and the model will not stop without intervention, but you can ignore the message because the generation limit will stop the model.

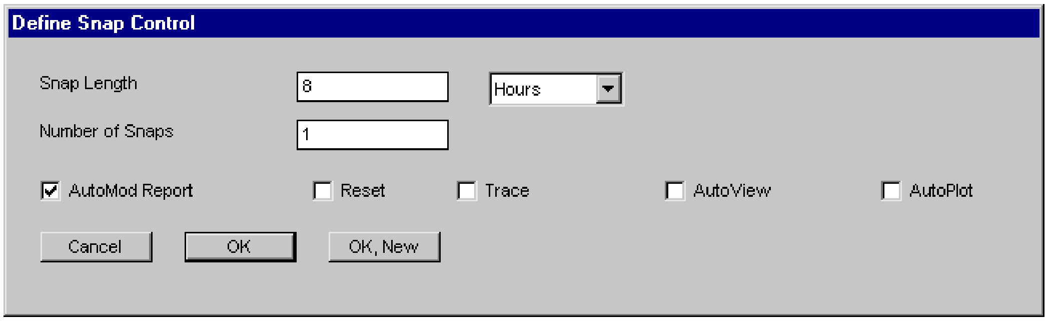
Defining the length of a simulation

Before running your model, you need to define the length of the simulation, which for this model is one eight-hour shift. To define the run length, do the following:

Step 1 *Click* Run Control on the Process System palette. The Run Control window opens.

Step 2 *Click* New to define a new snap. The Define Snap Control window opens.

Step 3 In the Snap Length text box, *type* “8” to define an eight-hour snap, as shown below:



Define Snap Control

Snap Length: 8 Hours

Number of Snaps: 1

☑ AutoMod Report □ Reset □ Trace □ AutoView □ AutoPlot

Define Snap Control window

Step 4 *Click* OK to close the Define Snap Control window.

You have now created your first model in the AutoMod software.

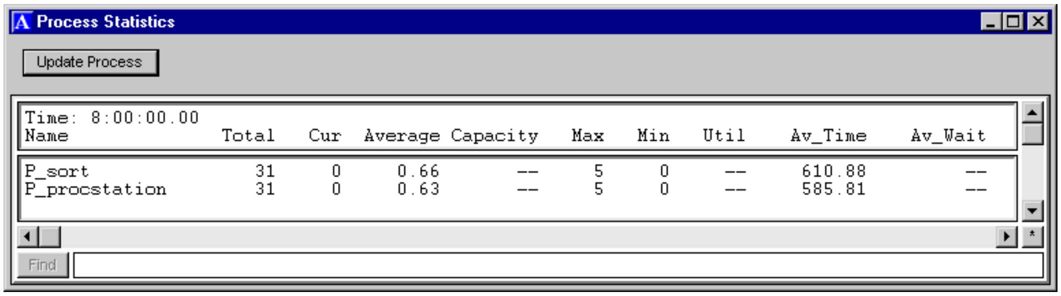
Step 5 *Export* the model.

Step 6 From the Model menu, *select* Run. *Build* the model. When the Simulation window opens, *continue* the model.

Displaying process system statistics

Step 1 As the simulation runs, *watch* the Message window. The print statements you wrote are sending messages to the window each time a load is sorted and each time a load leaves the system.

Step 2 At the end of the run, *select* Statistics Summary from the Processes menu. The Statistics Summary window opens, as shown below:



Process Statistics

```
Time: 8:00:00.00
Name                 Total   Cur   Average Capacity   Max   Min   Util     Av_Time     Av_Wait

P_sort                  31     0      0.66       --     5     0     --     610.88          --
P_procstation           31     0      0.63       --     5     0     --     585.81          --
```

Process Statistics window

You can calculate the number of loads that *completed* each process by subtracting the Cur statistic (the number of loads currently in the process) from the Total statistic (the number of loads that were sent to the process). In this model, there are no loads currently in either process, so the Total and the number completed are the same (31 loads). To review the definitions of these statistics, see “Displaying process system summary statistics” on page 2.15 of the “Using the Software” chapter.

Summary

After completing this chapter, you know some of the basic syntax for writing model logic. You know how to use the `wait`, `print`, and `send` actions, and you are familiar with several distributions. You also learned the basic steps for creating a model in the AutoMod software:

Step 1 ***Create*** a new model.

Step 2 ***Create*** the process system.

Step 3 ***Write*** the model logic and ***define*** the necessary entities.

Step 4 ***Create*** new loads.

Step 5 ***Define*** the length of the simulation.

Step 6 ***Run*** the model.

Exercises

Exercise 4.1

Create a new model to simulate the following sorting operation:

Loads are created with an interarrival time that is exponentially distributed with a mean of 10 minutes. The time required to sort each load is triangularly distributed with a minimum value of 7 minutes, a most-likely value of 8 minutes 22 seconds, and a maximum value of 10 minutes 22 seconds. After sorting, loads leave the system.

Run the model for eight hours.

What was the maximum number of loads in the sorting operation at any one time?

Exercise 4.2

Copy your solution model for exercise 4.1 to a new directory. Edit the copied model so that the load ID, the load type, and the simulation clock time are printed to the Message window each time a load arrives or departs. Use the following format:

> Time: <x.xx> hours -- load arriving: <load ID (load type)>

and

> Time: <x.xx> hours -- load departing: <load ID (load type)>

Note

Time values are rounded to the second decimal place.

Exercise 4.3

Copy your solution model for exercise 4.2 to a new directory. Edit the copied model and change the printed message when loads depart to each of the following:

a) print this load “ left at ” (absolute clock/3600) as .3 “ hours” to message
b) print this load “ left at ” absolute clock as .1 “ seconds” to message
c) print absolute clock to message

When you make the changes for b) and c), comment out your previous work so that the logic is still in the source file.

Run the model after each change to determine how the new print statements differ.

Exercise 4.4

Create a new model to simulate the following sorting operation:

Loads simultaneously enter a sorting operation with interarrival times that are determined by three different distributions, shown below:

Load arrival #	Time between arrivals
1	exponential with a mean of 15 minutes
2	exponential with a mean of 20 minutes
3	exponential with a mean of 30 minutes

The time required to sort each load is a constant 10 minutes. After sorting, loads leave the system.

Run the model for 1000 hours.

What was the maximum number of loads in the sorting operation at any one time?

Exercise 4.5

Create a new model to simulate the following process:

Loads are created with an interarrival time that is exponentially distributed with a mean of 4 minutes. The loads go through four sequential processes that take the following amounts of time:

Process #	Time in seconds
1	normal 150, 10
2	triangular 120, 200, 240
3	exponential 180
4	constant 160

Run the model for 100 days.

What was the average number of loads in each process?

Exercise 4.6

Create a new model that generates loads according to the following creation rate, then analyze the interarrival times using a spreadsheet package such as Excel:

Create 1001 loads with an interarrival time that is normally distributed with a mean of 60 minutes and a standard deviation of 10 minutes.

Tip ☞ To limit the number of loads that are created during the simulation, define the load creation specification using a generation limit (see “Limiting the number of loads created” on page 4.19 for more information).

Print the arrival time, in minutes, to a text file as each load arrives.

In the spreadsheet, calculate the 1000 interarrival times. Use your spreadsheet’s histogram feature to generate a histogram of the interarrival times.

Tip ☞ Steps for creating a histogram in Excel are included below: (If you are using another spreadsheet package, consult that package’s documentation for information about how to create a histogram.)

Step 1 ***Import*** the file you created into Excel.

Step 2 ***Calculate*** the time between arrivals (interarrival times).

Step 3 In an empty column, ***define*** a column of numbers that begins with 20 and increments to 100 in intervals of 10. (These values are used later to define the bin range for the histogram.)

Step 4 From the Tools menu, ***select*** Add-Ins. The Add-Ins window opens.

Step 5 If it is not already selected, ***select*** Analysis ToolPak, then ***click*** Ok. You have now added histogram functionality to your spreadsheet.

Step 6 From the Tools menu, ***select*** Data Analysis. The Data Analysis window opens.

Step 7 ***Select*** Histogram, then ***click*** Ok. ***Use*** the window to create the histogram (be sure to select Chart Output to create a histogram chart).

Chapter 5

Process System Basics

Chapter 5

Process System Basics

In chapter 4, "Introduction to AutoMod Syntax," you modeled load sortation and processing operations by creating time delays in arriving procedures. In this chapter, you will learn how to create resources (people or machines that process loads) and queues (locations where loads await processing). This chapter also discusses resource down times, which can be used to model machine failures or operator breaks.

In addition to learning about new process system entities, you will learn new AutoMod syntax, including how to select alternating values from a distribution. Finally, you will learn how to interpret AutoMod reports.

Example 5.1: Producing widgets at Acme, Inc.

Example model 5.1 demonstrates a simple factory that produces widgets. The factory produces widgets in three steps. In the first step, widgets are cut and shaped by an automatic lathe. In the second step, an operator drills each widget using a drilling machine. In the last step, the operator inspects each drilled widget for defects.

Widget blanks are started in the factory every 10 minutes. The blanks move into an infinite-capacity queue, where they wait to be cut and shaped. The lathe can cut and shape two blanks at the same time. The amount of time required to cut and shape each blank is exponentially distributed with a mean of 18 minutes.

After they have been cut and shaped, widgets are placed in another queue that can hold a maximum of six widgets at a time. The widgets wait in this queue until the drilling machine becomes available. An operator sets up the machine for each new widget. The time required for the operator to set up each widget for drilling is exponentially distributed with a mean of 2 minutes. The drill then runs (without operator assistance) for a time that is exponentially distributed with a mean of 4 minutes. Finally, the operator removes the completed widget from the drill. The removal takes a time that is exponentially distributed with a mean of 55 seconds.

After drilling, completed widgets are placed in an infinite-capacity queue to await inspection. When not setting up the drill, the operator of the drill inspects completed widgets for defects. Inspection takes a time that is exponentially distributed with a mean of 50 seconds.

The operator takes a 30 minute lunch break that occurs four hours into the shift. The operator takes two additional breaks per day. Both breaks last 15 minutes and occur two hours into the shift and six hours into the shift, respectively.

The factory operates 7 days a week, 24 hours a day. Operators are scheduled in 8-hour shifts. In addition, the drilling machine randomly breaks down according to an exponential distribution with a mean of 200 minutes. The repair time is also exponentially distributed with a mean of 10 minutes.

You will simulate this system for 100 days of operation.

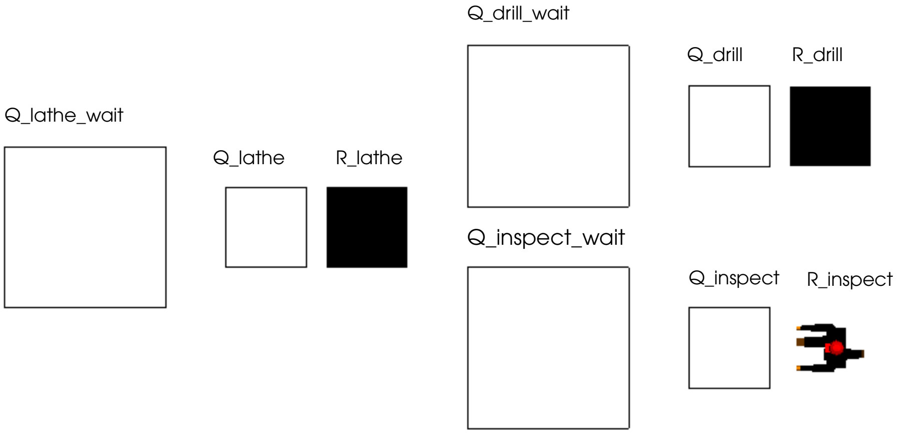
Running the example model

The directory for example model 5.1 contains both a base and final version of the model. The *final* version is a completed model that simulates the facility as it is described. The *base* version is a starting point from which you can add to the model throughout the chapter, learning how to define resources, queues, and so on. To begin, you will run the final version to view the system while it operates. Then you will edit the base version and define many of the components of the model to learn how to create resources and queues.

To become familiar with the processing of loads in the system:

Step 1 ***Import*** and ***run*** a copy of the *final* version of example model 5.1.

Step 2 To turn solids on, ***press*** “w”. The model is laid out as shown below:



Example model 5.1 layout

The solid shapes represent resources that process loads (the lathe, the drill, and the inspection operator). The other boxes represent queues. The large queues are waiting areas where loads accumulate while waiting to be processed by a resource. The smaller queues represent areas where loads are located while they are being processed.

Step 3 ***Quit*** the model.

After observing the system in operation, you are ready to learn how to use resources in a simulation.

Resources

Resources represent machines, operators, tools, fixtures, and other entities that process the items moving through the system (which are modeled as loads).

Note

A person can be modeled using either resources or loads, depending on the person’s role in the real system. If the person is an operator or attendant that services other entities moving through the system, model the person using a resource. If a person is serviced in a system, model the person as a load.

Resources in the AutoMod software have **capacity**, which is how many items they can process at the same time. A machine may be able to process more than one load at a time, for example. To allow a resource to process only one load at a time, you define its capacity as one.

As in the real world, when machines break or are unavailable for processing, they are considered “down.” In an AutoMod model, a machine goes down when it needs service, and an operator goes down when offshift or on break. Modeling resource down times is discussed later in this chapter (see “Modeling resource unavailability” on page 5.16 for more information).

Claiming and releasing resources

In the AutoMod software, loads execute the `get` action to claim a resource and the `free` action to release a resource. The `get` action claims one unit of a resource’s capacity, and the `free` action releases one unit of capacity. If a resource has no available capacity when a load tries to claim it, the load is delayed until a unit of the resource’s capacity becomes available.

For example, the syntax:

```
get R_resource
wait for 10 min
free R_resource
```

causes a load to claim one unit of R_resource’s capacity, wait for 10 minutes, and then release one unit of the resource’s capacity.

As an alternative to using the `get`, `wait`, and `free` actions, you can perform the same claim, delay, and release of a resource using a single `use` action, as shown below:

```
use R_resource for 10 min
```

Think of the `use` action as a shortcut for getting a resource, delaying, and then freeing a resource.

Determining which actions to use when claiming resources

The use action makes it easy to claim a resource, delay for a specified amount of time, and then release the resource. The use action is limited, however, because a load cannot execute any other actions in the model logic while the resource is claimed. Because of this limitation, when loads must perform more actions than a single wait action while claiming a resource, you must use the get and free actions to claim and release the resource.

For example, the following logic shows the drilling operation in the example model. Notice that in this procedure, loads use two resources: the drill and the operator who sets up and removes loads from the drill.

```
get R_drill                  /*claim the drill*/
use R_operator for e 2 min   /*setup time*/
wait for e 4 min             /*drill time*/
use R_operator for e 55 sec  /*remove time*/
free R_drill                 /*now drill can be freed*/
send to P_inspect
```

First, a load claims the drill. While the drill is claimed, the load also claims the operator to set up the drill. After the setup time (e 2 min), the operator is released, and the machine drills the load. The load then reclaims the operator, who removes the load from the machine (e 55 sec). After the removal time, both the operator and the drill are released.

While loads are claiming the drill, they must perform more than one action; consequently, the get and free actions are used to claim and release the drill. In contrast, only a time delay is required while loads are claiming the operator, so the use action is used to model the setup and removal times.

Defining a resource

The base model for example 5.1 does not have the lathe machine defined. Therefore, define and place the lathe resource in a copy of the base model.

To define a resource:

Step 1 ***Import*** a copy of the *base* version of example model 5.1.

Note

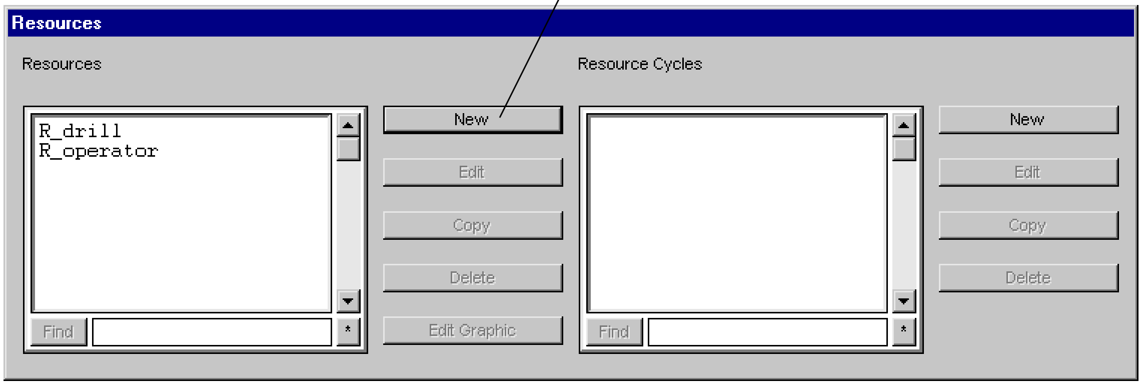
An attention window opens during import, indicating that there are errors in the model logic. The errors exist because the entities required for the lathe operation are currently missing from the model. You are going to add the necessary entities to the model, so click OK to close the window.

Step 2 To turn solids on, ***press*** “w”.

Step 3 ***Click*** Resources on the Process System palette. The Resource window opens.

Step 4 ***Click*** New to the right of the Resources list to define a new resource, as shown below:

Click New to define a new resource



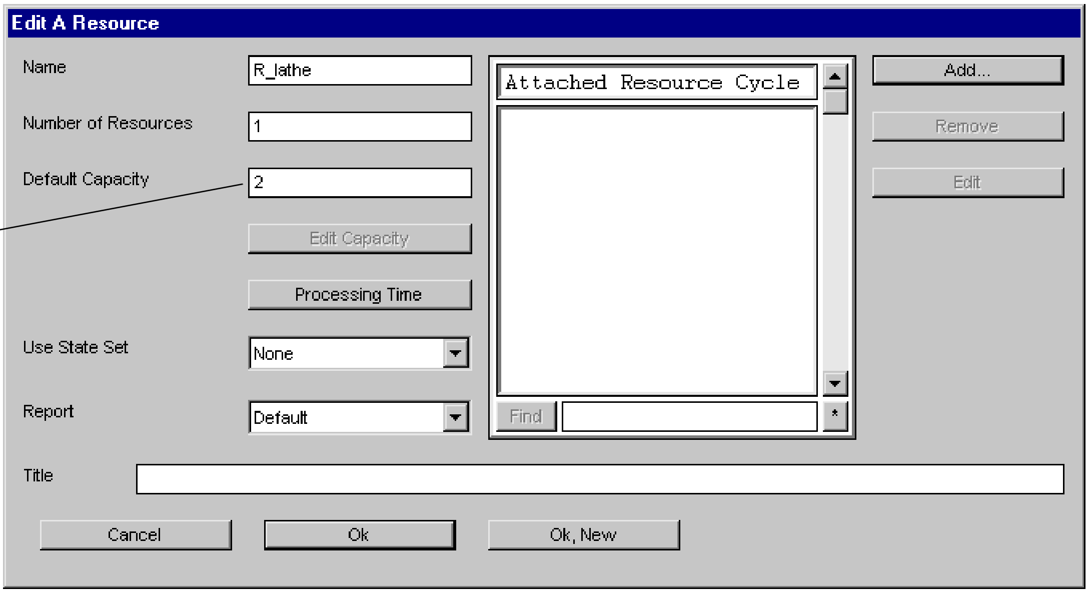
R_drill
R_operator

Resources window

The Define A Resource window opens.

Step 5 In the Name text box, ***type*** “R_lathe”.

Step 6 In the Default Capacity text box, ***type*** “2” (the lathe can cut and shape a maximum of two loads at the same time).



Set the capacity to 2

Edit A Resource

Name R_lathe

Number of Resources 1

Default Capacity 2

Use State Set None

Report Default

Title

Edit A Resource window

Step 7 ***Click*** OK to close the Edit A Resource window. You have now defined the lathe resource.

Placing resource graphics

Placing a resource's graphic allows you to visually see the state of a resource (idle, busy, or down) during a simulation. Resource graphics change color depending on their current activity, as defined in the following table:

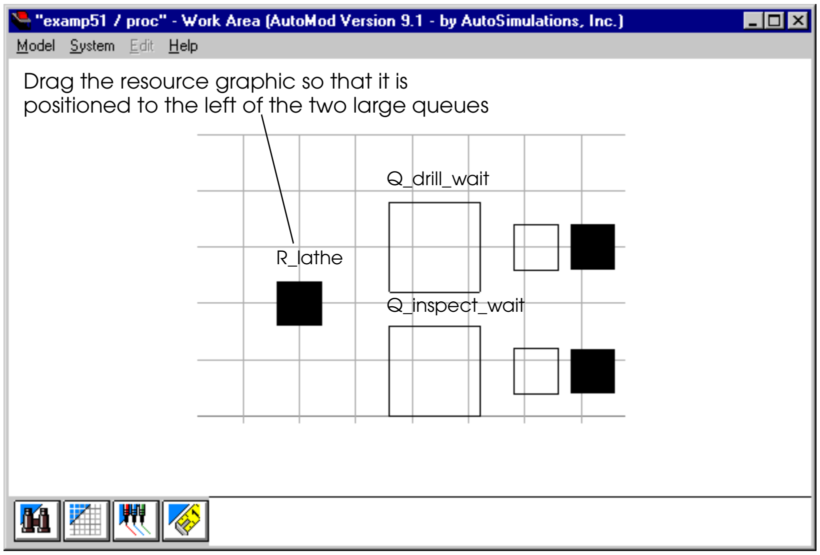
Resource's activity	Color
processing one or more loads	green
idle (not being used by any loads)	blue
down	red

To place the R_lathe resource's graphic:

Step 1 In the Resources window, ***select*** R_lathe, then ***click*** Edit Graphic. The Edit Resource Graphics window opens.

Step 2 ***Click*** Place.

Step 3 In the Work Area window, ***press and hold*** the mouse button; a box representing R_lathe appears. ***Drag*** the resource graphic until it is positioned to the left of the two large queues (see the illustration below), then ***release*** the mouse button.


"examp51 / proc" - Work Area (AutoMod Version 9.1 - by AutoSimulations, Inc.)
Model System Edit Help
Q_drill_wait
R_lathe
Q_inspect_wait

Drag the resource graphic so that it is positioned to the left of the two large queues

Placing resource R_lathe

Tip

If you need to move the resource, click Move in the Edit Resource Graphics window, then drag the resource graphic to a new location in the Work Area window.

Step 4 ***Click*** Done to close the Edit Resource Graphics window.

You have now placed the graphic for R_lathe.

Step 5 ***Export*** the model.

Importing graphics

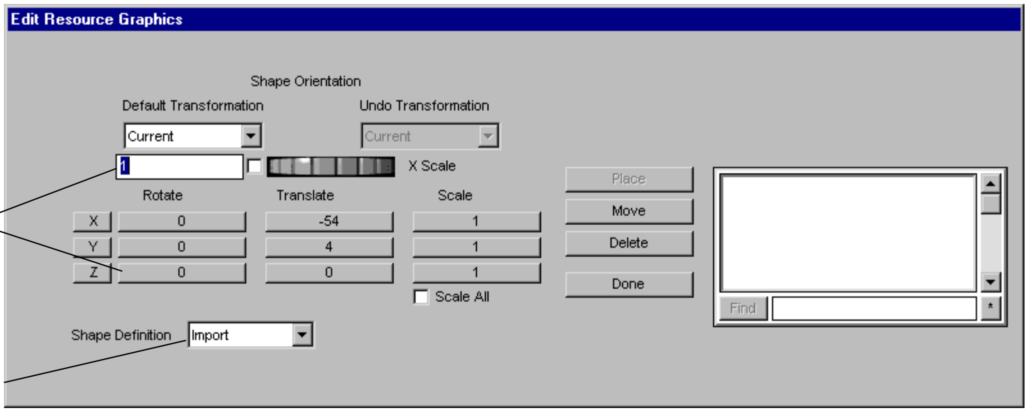
The student version of AutoMod includes several predefined graphics (called cell files) that you can use to add graphical realism to your model. These instructions show you how to import a cell file for a resource; you can also import cell files to increase the graphical realism of other entities, such as loads or vehicles.

In example model 5.1, you will import a cell file of a man to use for the operator.

To import the cell file:

Step 1 In the Resources window, ***select*** R_operator, then ***click*** Edit Graphic. The Edit Resource Graphics window opens.

Step 2 ***Select*** Import from the Shape Definition drop-down list. The Open a File window opens.



Click Z Rotate and type 180 to rotate the man

Select Import

Edit Resource Graphics window

Step 3 ***Navigate*** to the /demos/amodprims directory in the installation directory.

Step 4 ***Select*** “man.cell” and ***click*** Open. The graphic of a man replaces the box in the Work Area window.

Now rotate the man so that he is facing the queue Q_inspect:

Step 5 ***Click*** Z rotate.

Step 6 ***Type*** 180 in the Z Rotate box, then ***press*** Enter. The graphic is rotated in the Work Area window.

Step 7 ***Click*** Done to close the Edit Resource Graphics window.

Queues

Queues represent physical space where loads can be stored. Like resources, you can define the capacity of queues to limit the number of loads that are allowed to wait in a queue at the same time.

Note

Later in this textbook, you will learn how to move loads out of queues and onto conveyors and vehicles. Until then, loads in a simulation must always be located in a queue while waiting or processing.

In the AutoMod software, queues and resources are used together to represent a piece of equipment. A load cannot physically be on a resource, because a resource is not a territory (see “Territories and space” on page 3.10 of the “AutoMod Concepts” chapter). Because a load cannot be on a resource, you must use a queue on or near the resource to store the load while it is being processed by the resource. Using a queue with each resource also helps separate true queue (waiting) time from processing time, as discussed below.

Moving loads into queues

Loads move in and out of queues using the `move` action. When a load moves into a queue, the available capacity of the queue decreases by one. Conversely, when a load moves out of a queue, the queue’s available capacity increases by one. If a load attempts to move into a queue that has no remaining capacity, the load is delayed and must wait until a unit of the queue’s capacity becomes available before continuing.

The following logic is used in the example model to move loads into the inspection area, simulate the inspection time, and then remove loads from the system:

```
begin P_inspect arriving
   move into Q_inspect_wait       /*limit = infinity*/
   move into Q_inspect            /*limit = 1*/
   use R_operator for e 50 sec    /*inspection time*/
   send to die
end
```

The procedure first moves loads into the queue named Q_inspect_wait, which is defined with infinite capacity. Therefore, loads never wait to move into the waiting queue. Loads wait in the waiting queue until they can move into queue Q_inspect, which is defined with a capacity of one. When a load moves into queue Q_inspect, it uses resource R_operator for the required inspection time, and is then sent to die. Loads that are sent to die are automatically removed from queue Q_inspect, freeing the capacity for another load.

Separating waiting and processing queues

Notice that in the P_inspect arriving procedure, loads move sequentially into two queues: first Q_inspect_wait and then Q_inspect. The first queue is where loads wait for inspection, and the second queue is where loads are actually inspected by the operator.

In the examples and exercises in this textbook, loads that are waiting for a resource usually wait in a separate queue from the loads that are using a resource.

In most real-world systems, parts wait in a bin or rack until they can be moved to a machine or are held by an operator for processing. Therefore, modeling two queues is realistic, both graphically and statistically. By separating the waiting and processing queues, you also separate the waiting statistics from the processing statistics. The length of time that loads spend in a queue waiting for a resource is tracked separately from the length of time that loads spend in a queue using a resource, making it easier for you to verify that the model is accurate.

Defining a queue

You are now ready to define the waiting and processing queues for the lathe resource. Defining queues is very similar to defining resources.

To define a queue:

Step 1 ***Click*** Queues on the Process System palette. The Queues window opens.

Step 2 ***Click*** New to create a new queue. The Define A Queue window opens.

Step 3 In the Name text box ***type*** "Q_lathe_wait".

Step 4 In the Default Capacity text box, ***type*** "i" for infinite.

Step 5 ***Click*** OK/New to define another queue.

Step 6 In the Name text box, ***type*** "Q_lathe".

Step 7 In the Default Capacity text box, ***type*** "2" (the lathe can cut and shape two loads at the same time).

Step 8 ***Click*** OK to close the Define A Queue window.

You have now defined the two queues for the lathe resource.

Placing queue graphics

Placing queue graphics allows you to visually see whether loads are waiting or processing at various places during the simulation. Queue graphics change color depending on their current contents (either no loads in the queue or at least one load in the queue), as defined in the following table:

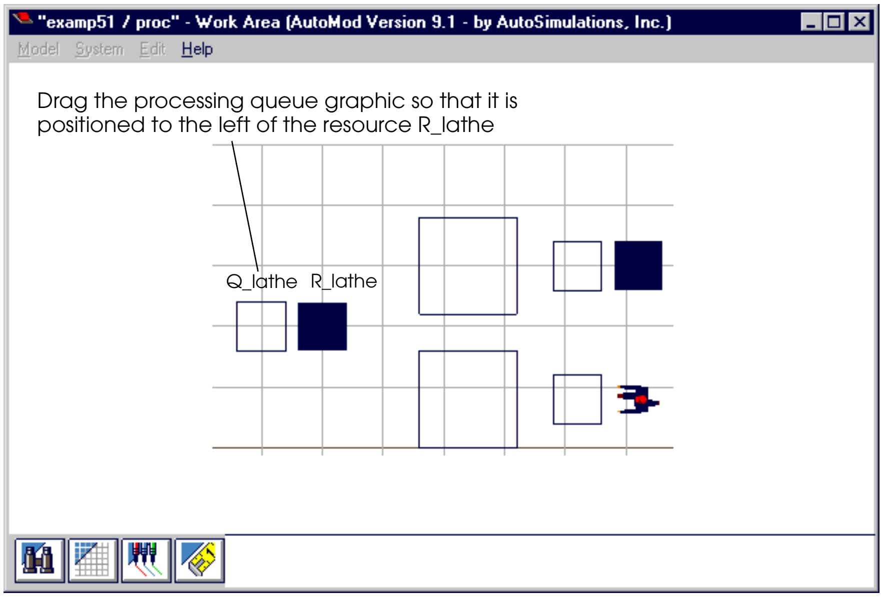
Queue's contents	Color
One or more loads	green
No loads	red

To place the graphic for the processing queue:

Step 1 In the Queues window, ***select*** Q_lathe, then ***click*** Edit Graphic. The Edit Queue Graphics window opens.

Step 2 ***Click*** Place.

Step 3 In the Work Area window, ***drag*** the queue graphic until it is positioned to the left of the lathe resource (see the illustration below).


"examp51 / proc" - Work Area (AutoMod Version 9.1 - by AutoSimulations, Inc.)
Model System Edit Help
Q_lathe
R_lathe

Drag the processing queue graphic so that it is positioned to the left of the resource R_lathe

Placing queue Q_lathe

Step 4 ***Click*** Done to close the Edit Queue Graphics window.

To place the graphic for the waiting queue:

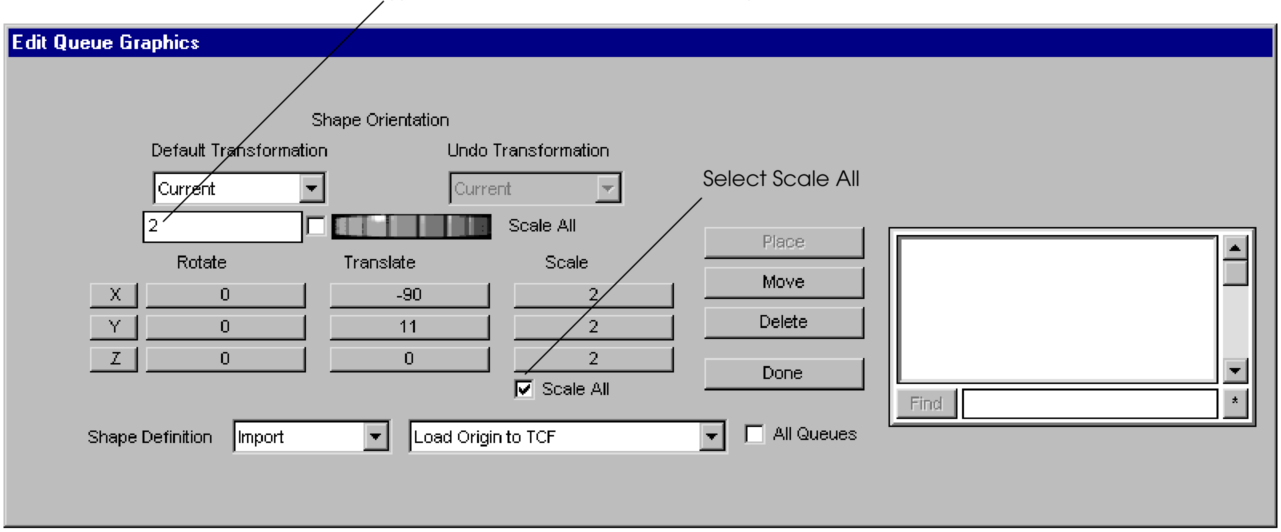
Step 5 In the Queues window, ***select*** Q_lathe_wait and ***click*** Edit Graphic.

Step 6 ***Click*** Place and ***click*** to the left of Q_Lathe. It is alright to place the graphic outside of the grid. A box representing the waiting queue appears.

Because the waiting queue has infinite capacity, make it larger than the processing queue.

Step 7 To enlarge the waiting queue, ***select*** the Scale All check box in the Edit Queue Graphics window. ***Type*** “2” in the Scale text box, then ***press*** Enter, as shown below:

Type “2” in the Scale text box, then press Enter



Select Scale All

Default Transformation: Current
Undo Transformation: Current
2 □ Scale All

	Rotate	Translate	Scale
X	0	-90	2
Y	0	11	2
Z	0	0	2

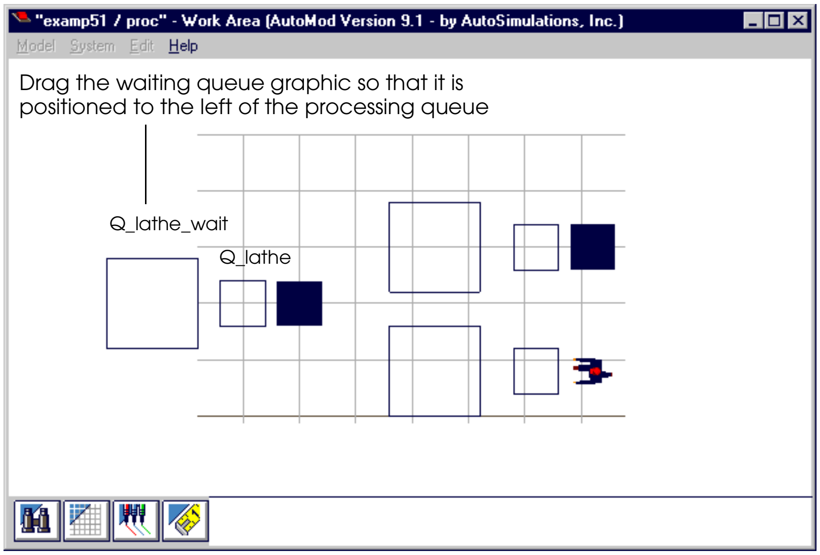
☑ Scale All

Shape Definition: Import, Load Origin to TCF □ All Queues

Edit Queue Graphics window

The queue graphic is scaled to 2 feet in every direction.

Step 8 If the queue you just placed is too close to R_lathe or is out of position, ***click*** Move in the Edit Queue Graphics window. In the Work Area window, ***drag*** the waiting queue so that it is positioned to the left of the processing queue, as shown below:


"examp51 / proc" - Work Area (AutoMod Version 9.1 - by AutoSimulations, Inc.)
Model System Edit Help
Drag the waiting queue graphic so that it is positioned to the left of the processing queue
Q_lathe_wait
Q_lathe

Placing queue Q_lathe_wait

Step 9 ***Click*** Done.

Step 10 ***Export*** and ***run*** the model.

Tip ☞ If an Error window opens when you try to run the model, verify that you have entered the names of the lathe resource and queues exactly as they appear in this textbook, because the entities have been defined using names that are spelled exactly as written (names are case-sensitive).

Step 11 ***Verify*** that loads are using the new queues and resource by watching the graphics and checking the resource and queue statistics.

Step 12 When you are ready to continue, ***edit*** the model.

Modeling resource unavailability

Resources become unavailable for several reasons, which vary based on the type of resource being modeled (a piece of equipment, an operator, and so on). Reasons that resources might be unavailable include: failures, scheduled maintenance, off-shift periods, breaks, and lunches. By default in AutoMod, all of these types of unavailability are modeled as down times (resources are either “down” or “up”). A resource that is taken down is unavailable to process loads. A resource that is up is able to process loads.

When a resource is down, loads that attempt to claim the resource are automatically delayed until the resource is brought up. When the resource is brought up, loads claim the resource in first-in first-out (FIFO) order (the load that has been waiting the longest is the first to claim the resource).

If a resource is taken down while a load is using the resource, the load is automatically delayed until the resource is brought up. When the resource is brought up, the load resumes processing for the remaining processing time. For example, assume a load is required to use a resource for 5 minutes. If the resource goes down 2 minutes into the load’s processing, when the resource is brought up, the load still has 3 minutes of processing time remaining, regardless of the length of time that the resource was down.

Resources can be taken down and brought up in two ways:

- Using logic in a procedure
- Using resource cycles

Modeling down times using logic

A load can execute the `take down` action in an arriving procedure to make a resource unavailable during a simulation. After a delay, the load can execute the `bring up` action to make a resource available again. For example, the syntax:

```
take down R_resource
wait for e 12 min
bring up R_resource
```

takes down resource R_resource, waits for a repair time that is exponentially distributed with a mean of 12 minutes, and then brings up resource R_resource.

Using “dummy” loads to execute down time processes

Until now, all processes in a model have related directly to the item being manufactured or serviced, and are therefore executed by the load being manufactured. With down time procedures, however, you cannot use the load that represents the product to control down times, because when a resource goes down, any loads that are claiming the resource are automatically delayed, and therefore could not bring the resource back up. Consequently, a process that models resource down times should be executed by a single **dummy** load (a load that is not one of the products moving through the system) that executes the `take down` and `bring up` actions in a continuously repeating loop throughout a simulation. The dummy load should have a generation limit of one and be sent to the process at time zero (for more information, see “Creating a dummy load at time zero” on page 5.18).

Repeating logic can be written in an arriving procedure using the syntax `while...do`.

Writing repeating logic to model down times

The syntax while...do indicates that as long as a condition is true, certain actions should be executed. For example, procedure P_break models the operator’s lunch and breaks in example model 5.1 using a while...do loop. In this case, the condition is while 1=1, and because 1 is always equal to 1, the loop repeats continuously for the entire simulation.

```
begin P_break arriving
   while 1=1 do                        /*do for entire simulation*/
   begin
      wait for 120 min                 /*works for 2 hours*/
      take down R_operator             /*first break*/
      wait for 15 min                  /*lasts for 15 min*/
      bring up R_operator              /*back to work*/
      wait for 105 min                 /*work until lunch*/
      take down R_operator             /*lunch time*/
      wait for 30 min                  /*30 minute lunch*/
      bring up R_operator              /*back to work*/
      wait for 90 min                  /*works for 1.5 hours*/
      take down R_operator             /*second break*/
      wait for 15 min                  /*lasts for 15 min*/
      bring up R_operator              /*back to work*/
      wait for 105 min                 /*work until shift ends*/
   end
end
```

The while...do syntax is followed by a series of actions that start with begin and end with end. The begin and end syntax defines a loop that executes as long as the condition in the while...do syntax remains true.

Look at how the operator’s down times are scheduled inside the loop. Each loop represents one shift (all of the time delays in the loop add up to 8 hours). The operator’s first break occurs two hours (120 minutes) into the shift, so the first take down action occurs after a 120 minute time delay. The break lasts 15 minutes, so the operator is brought up after a 15 minute delay.

After the morning break, the operator works until lunch, which begins 4 hours into the shift (105 minutes after the end of the morning break). Consequently, after delaying for 105 minutes, the operator is taken down for lunch and brought up again after 30 minutes.

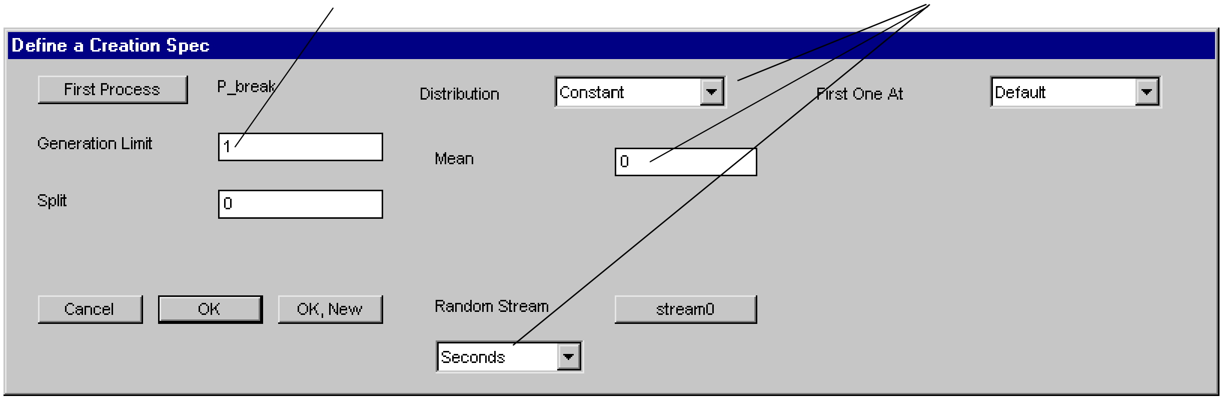
The operator’s afternoon break occurs 6 hours into the shift, which is 90 minutes after the lunch break ends. Consequently, the next take down action occurs after a time delay of 90 minutes. The break lasts 15 minutes, so the operator is brought up after a 15 minute time delay.

The last time delay in the loop is 105 minutes, which is the amount of time the operator works before the shift ends. The loop repeats for the next shift.

Note

A dummy load usually remains in Space throughout the simulation, as it does in this example (the load does not move into a queue or any other territory), because it does not represent a physical item in the facility.

The P_break procedure is executed by a single dummy load named L_break. The creation specification for the L_break load type has a generation limit of one (only one load of type L_break is created during the simulation), and the load is created at time zero, as shown below:



The Generation Limit is 1

The Distribution is constant 0 seconds

Define a Creation Spec

First Process: P_break

Generation Limit: 1

Split: 0

Distribution: Constant

Mean: 0

First One At: Default

Random Stream: stream0

Seconds

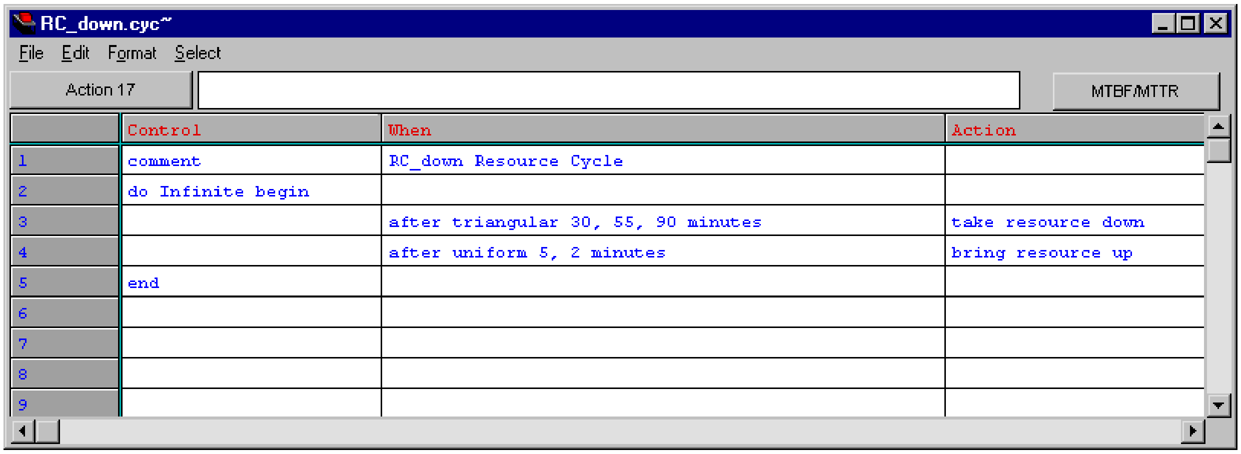
Creating a dummy load at time zero

The model contains a second down time procedure, named P_down, to model the drilling machine failures. The P_down procedure is executed by a single dummy load named L_down.

Another method for defining down times is using resource cycles, which is discussed next.

Modeling resource down times using resource cycles

A **resource cycle** is a list of actions that control one or more resources during a simulation. Resource cycles are an alternative to defining resource down times using arriving procedures. An example resource cycle is shown below:



	Control	When	Action
1	comment	RC_down Resource Cycle	
2	do Infinite begin		
3		after triangular 30, 55, 90 minutes	take resource down
4		after uniform 5, 2 minutes	bring resource up
5	end		
6			
7			
8			
9			

Resource cycle edit table

Resource cycles are defined in an edit table. The table contains three columns:

Control The Control column defines the beginning and end of a cycle (or loop) of repeating actions.

When The When column defines the times at which actions are performed. You can double-click a cell in the When column for a list of valid keywords. (You cannot use all AutoMod syntax in a resource cycle; only the keywords in the list are valid for resource cycles.)

Action The Action column defines one or more actions that control a resource during a simulation. You can double-click a cell in the Action column for a list of valid actions. (You cannot use all AutoMod syntax in a resource cycle; only the actions in the list are valid for resource cycles.)

Note You can define multiple actions in the same resource cycle. For example, you could model all the breaks in one shift in the same cycle, as long as you define the cycle's repeating time interval so that it is the same as the shift length (one complete cycle represents one complete shift).

The next section discusses how to create a resource cycle to replace the P_down arriving procedure for example model 5.1.

Replacing the P_down arriving procedure

Example model 5.1 currently uses the arriving procedure P_down to take down the drill. To replace the procedure with a resource cycle, you must comment out the procedure, delete the dummy load that executes the procedure, and define a resource cycle.

To comment the arriving procedure and delete the dummy load:

Step 1 ***Edit*** the logic.m source file, then ***highlight*** lines 47-55 containing the P_down arriving procedure.

Step 2 From the Edit menu, ***select*** Comment Block. Comment markers are automatically inserted before and after the procedure, as shown in bold below:

```
/* begin P_down arriving
   while 1=1 do
   begin
      wait for e 200 min      /*up time*/
      take down R_drill       /*down time begins*/
      wait for e 10 min       /*down time*/
      bring up R_drill        /*repair finished*/
   end
end */
```

Step 3 From the File menu, ***select*** Save and Quit.

Because the procedure is no longer used, you must delete the dummy load that is being sent to the procedure, or you will get an error when you run the model:

Step 4 ***Click*** Loads on the Process System palette. The Loads window opens.

Step 5 ***Select*** L_down in the load types list, then ***click*** Delete. A confirmation window opens. ***Click*** Yes to delete the load type.

You are now ready to define a resource cycle.

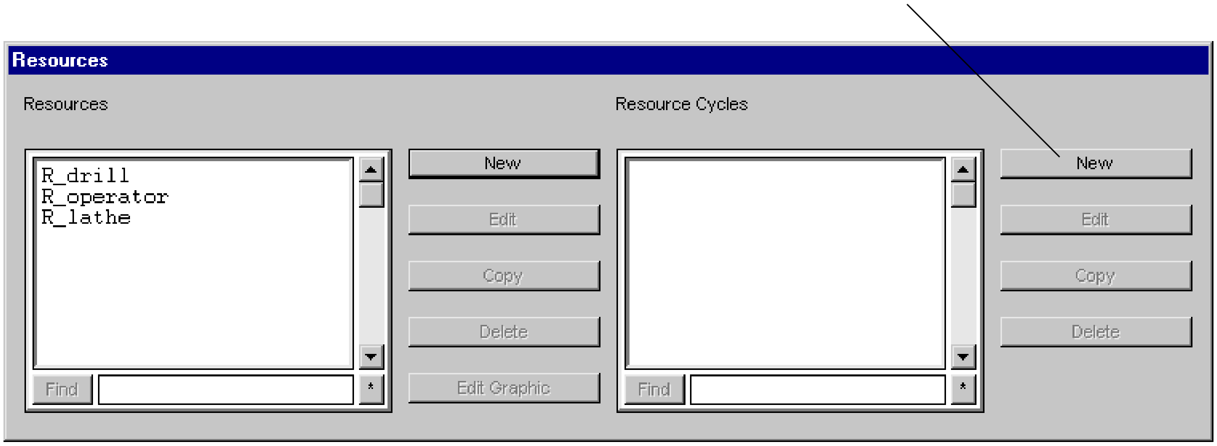
Creating a resource cycle

To create a resource cycle to model failures for the drill, do the following:

Step 1 *Click* Resources on the Process System palette. The Resources window opens.

Step 2 *Click* New to the right of the Resource Cycles list to create a new resource cycle, as shown below:

Click New to create a new resource cycle



R_drill
R_operator
R_lathe

Resources window

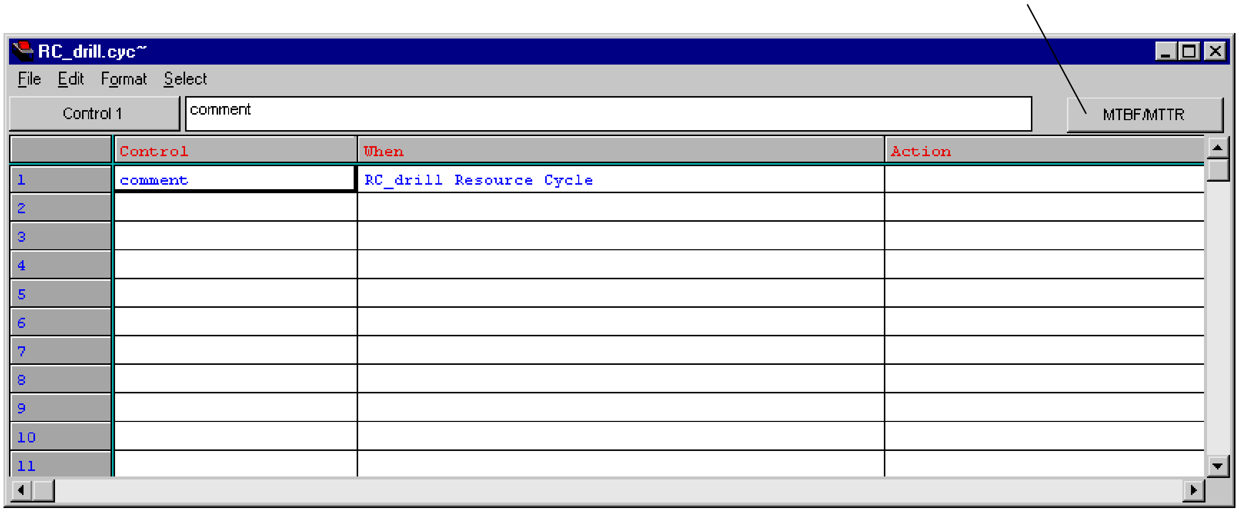
The New Resource Cycle window opens.

Step 3 In the Name text box, ***type*** "RC_drill" and ***click*** OK, Edit. A new resource cycle edit table opens.

In a resource cycle, you can automatically generate actions to take down and bring up resources by defining a resource's mean time between failures (MTBF) and mean time to repair (MTTR).

Step 4 *Click* MTBF/MTTR, as shown below:

Click MTBF/MTTR



comment

	Control	When	Action
1	comment	RC_drill Resource Cycle	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			

Resource cycle edit table

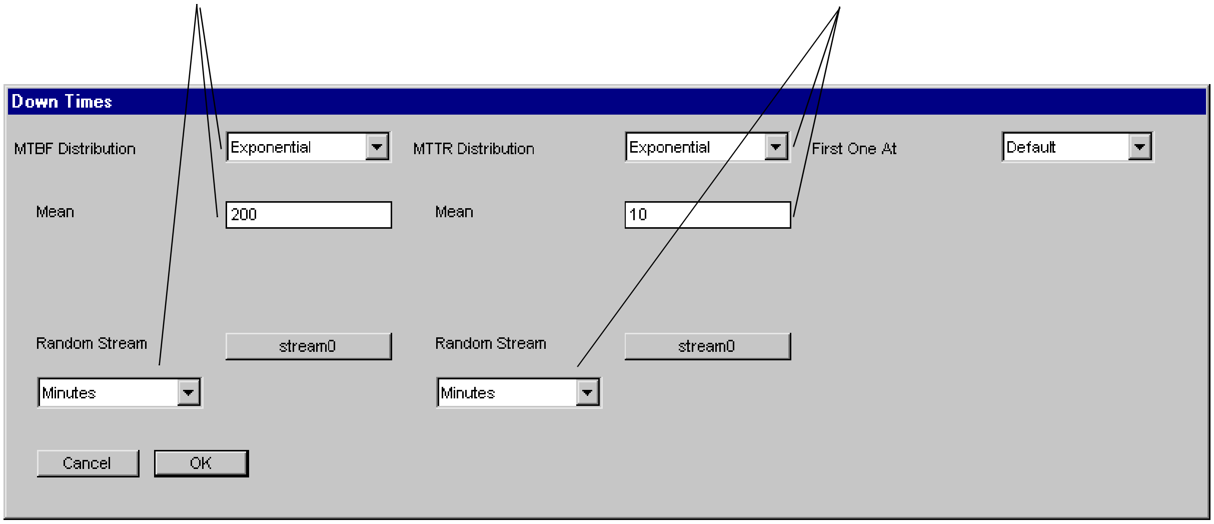
The Down Times window opens.

According to the example description (see “Example 5.1: Producing widgets at Acme, Inc.” on page 5.4), the time between failures is exponentially distributed with a mean of 200 minutes, and the time required to repair the machine is exponentially distributed with a mean of 10 minutes.

Step 5 ***Enter*** the distributions in the Down Times window, as shown below:

The mean time between failures is exponentially distributed with a mean of 200 minutes

The mean time to repair is exponentially distributed with a mean of 10 minutes



Down Times

MTBF Distribution: Exponential

Mean: 200

Random Stream: stream0

Minutes

MTTR Distribution: Exponential

Mean: 10

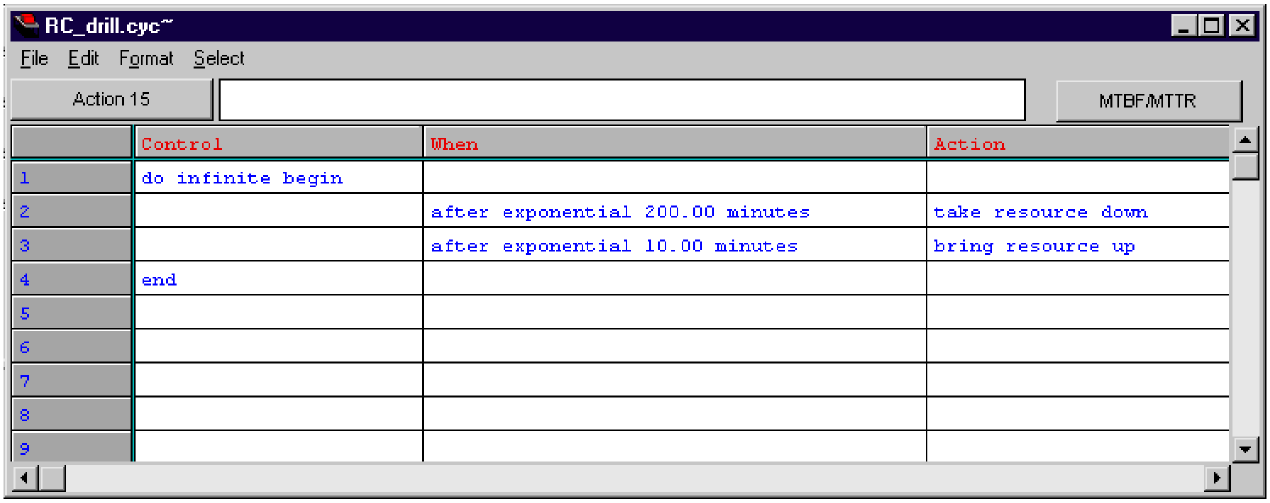
Random Stream: stream0

Minutes

First One At: Default

Defining a resource’s MTBF and MTTR

Step 6 ***Click*** OK to automatically generate the resource cycle in the edit table. The resource cycle opens, as shown below:



	Control	When	Action
1	do infinite begin		
2		after exponential 200.00 minutes	take resource down
3		after exponential 10.00 minutes	bring resource up
4	end		
5			
6			
7			
8			
9			

Completed resource cycle edit table

The resource cycle causes the same delays as the P_down arriving procedure that we commented in the model logic. Row 1 defines a loop that repeats an infinite number of times during a simulation. Row 2 waits for the time between failures and then takes down the resource. Row 3 waits for the amount of time that the resource is repaired and then brings up the resource. Row 4 defines the end of the loop.

Step 7 From the File menu, ***select*** Save.

Step 8 From the File menu, ***select*** Quit to close the edit table.

Now that you have defined the resource cycle, you need to attach it the drilling machine.

Attaching a resource cycle to a resource

In order for a resource to use the cycle you have just defined, you must attach the resource cycle to a resource, or the cycle will be ignored during the simulation.

Note

You can attach a resource cycle to multiple resources that share the same failure and repair rates. When a resource cycle is attached to more than one resource, the resources are taken down and brought up independently (unique random numbers are generated for each resource). If you need to model different failure rates, either create multiple cycles or use logic to model the down times.

You need to attach the resource cycle RC_drill to the resource R_drill.

To attach the resource cycle:

Step 1 In the Resources window, ***select*** R_drill from the Resources list and ***click*** Edit. The Edit A Resource window opens.

Step 2 ***Click*** Add to the right of the Attached Resource Cycle list. The Add Resource Cycle window opens.

Step 3 The resource cycle RC_drill is already selected, so ***click*** Add to attach the cycle to R_drill. The resource cycle appears in the Attached Resource Cycle list in the Edit A Resource window.

Step 4 ***Click*** OK to close the Edit A Resource window.

Step 5 ***Export*** the model and ***run*** the simulation.

To verify that the resource is being taken down correctly during the simulation, look at the single resource statistics, as described in the next section.

Verifying down times for resources

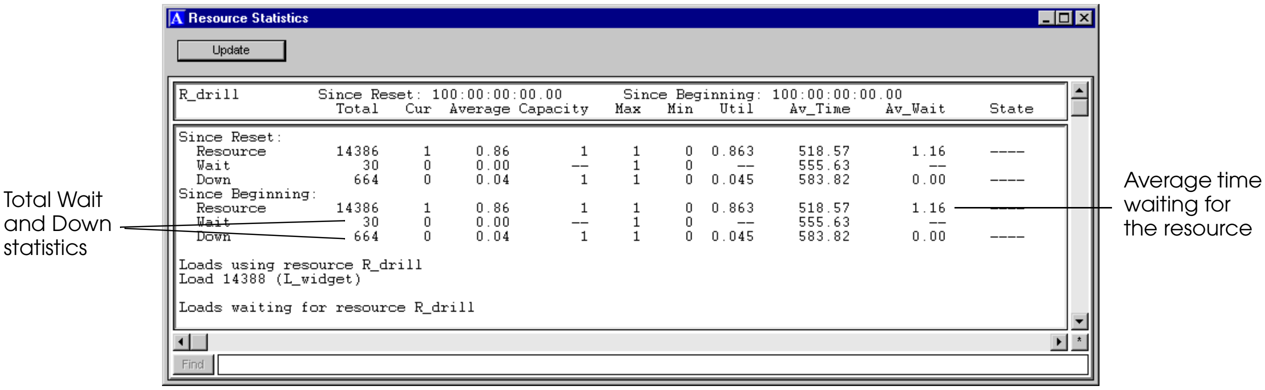
Until now, you have displayed statistic summaries when analyzing a simulation. Now you will learn how to display statistics for a single resource, which include information about the resource's down times. Single resource statistics also provide information about the utilization of the resource and whether loads had to wait to use the resource.

To display single statistics for resource R_drill:

Step 1 ***Run*** the simulation to completion.

Step 2 From the Resources menu in the Simulation window, ***select*** Single Resource. The Pick a Resource window opens.

Step 3 ***Select*** R_drill in the Resource list, then ***click*** OK. The Resource statistics window opens, as shown below:



```
R_drill         Since Reset: 100:00:00:00.00       Since Beginning: 100:00:00:00.00
                  Total   Cur  Average Capacity   Max   Min   Util     Av_Time    Av_Wait      State
Since Reset:
  Resource        14386     1     0.86        1     1     0  0.863      518.57       1.16      ----
  Wait               30     0     0.00       --     1     0     --      555.63         --
  Down              664     0     0.04        1     1     0  0.045      583.82       0.00      ----
Since Beginning:
  Resource        14386     1     0.86        1     1     0  0.863      518.57       1.16      ----
  Wait               30     0     0.00       --     1     0     --      555.63         --
  Down              664     0     0.04        1     1     0  0.045      583.82       0.00      ----

Loads using resource R_drill
Load 14388 (L_widget)

Loads waiting for resource R_drill
```

Total Wait and Down statistics

Average time waiting for the resource

Resource statistics for R_drill

Note Single resource statistics show two sets of statistics: statistics collected since the last reset (**relative** statistics), and statistics collected since the beginning of the simulation (**absolute** statistics. Resetting statistics is not discussed in this textbook, and no models use reset statistics. Therefore, both sets of statistics are the same and cover the entire simulation.

Look at the totals for the down and wait statistics. The total wait statistics indicate that 30 loads had to wait to claim R_drill because of capacity constraints (the drill can only process one load at a time). The total down statistics indicate that R_drill was taken down 664 times.

You can verify that the average wait (Av_Wait) statistics are accurate using a simple calculation. If 30 loads wait an average of 555.63 seconds to claim the resource, and 14356 loads (14386-30) wait zero seconds to claim the resource, the average is calculated as:

$$\frac{30(555.63) + 14356(0)}{14386} = 1.16$$

which is the value listed as the Av_Wait statistic for the resource.

The down statistics indicate that the resource was down an average time of 583.82 seconds, or 9.73 minutes. This is close to the mean time to repair of 10 minutes defined in the resource cycle. Similarly, the drill was down for 0.045 (or 4.5 percent of the time). This is close to the time that results from dividing the mean time to repair by the sum of the mean time to repair and the mean time between failures:

$$\frac{10}{(10 + 200)} = 0.048$$

The utilization of the drill was 0.863, and it was down for 0.045, so the drill's idle time is:

$1 - 0.863 - 0.045 = 0.092$, or 9.2 percent.

Creating business graphs to view statistics

So far, all the statistics you have looked at are presented in a textual form. AutoMod also has the ability to graph these statistics as pie charts, bar charts, and other types of graphs. Each graph can hold multiple statistics, and you can define multiple graphs for a model.

Suppose that you want to know how the current number of loads in Q_lathe_wait changes over the first 10 days of the simulation. To get this information, you would create a timeline business graph. Business graphs are defined in the process system using the Business Graphics option on the palette.

Note

Business graphs can also be defined while running a model; however, graphs defined during a run are temporary for that run. Graphs defined in the process system are saved and can be viewed during any run.

To define a timeline graph:

Step 1 ***Select*** Edit Model from the Control menu.

Step 2 ***Click*** Business Graphics on the Process System palette. The Business Graphics window opens.

Step 3 ***Click*** New. A graph, called “graph0,” is created.

Step 4 ***Name*** the graph “Lathe_Queue” (replace the name “graph0”).

Step 5 ***Select*** Timeline from the Graph Type drop-down list. The values for the X and Y axes of the graph appear.

Step 6 ***Define*** the following values for the X axis (time) and Y axis (the current number of loads in Q_lathe_wait):

```
Y Maximum =  20
Y Increment = 2
X Minimum =   0
X Maximum =  10
X Increment =.5
```

Tip

When creating your own graphs, you can estimate X and Y axis values by running the model and looking at the statistics you want to graph to get an idea of their values. Then you can define your X and Y values accordingly.

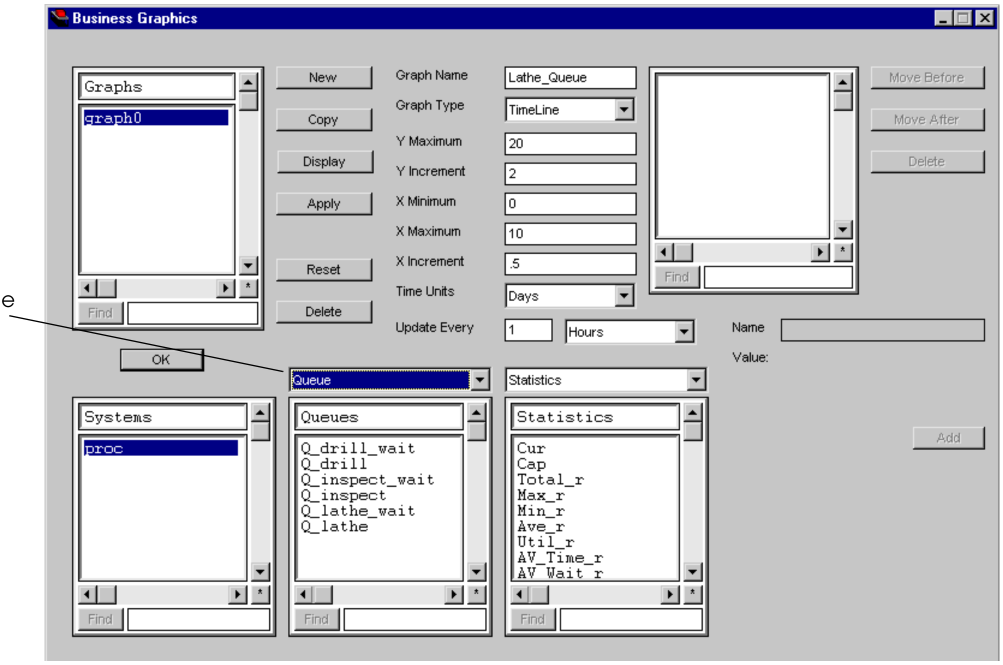
Step 7 ***Change*** the time units of the graph from Hours to Days using the drop-down list. This defines the X axis as going from zero to 10 days in half-day increments.

Step 8 ***Change*** the Update Every interval (how often a data point is graphed) to 1 hour.

Now you are ready to define which statistic you are graphing.

Step 9 To define which type of entity you want to graph, ***select*** Queue from the drop-down list above the middle select list, as shown below:

Select Queue to change the type of entity being graphed



Graphs: graph0

Graph Name: Lathe_Queue
Graph Type: TimeLine
Y Maximum: 20
Y Increment: 2
X Minimum: 0
X Maximum: 10
X Increment: .5
Time Units: Days
Update Every: 1 Hours

Queue

Statistics

Systems: proc

Queues:
Q_drill_wait
Q_drill
Q_inspect_wait
Q_inspect
Q_lathe_wait
Q_lathe

Statistics:
Cur
Cap
Total_r
Max_r
Min_r
Ave_r
Util_r
AV_Time_r
AV_Wait_r

Changing the entity type in a business graph

Step 10 ***Select*** Q_lathe_wait in the Queues list.

The third select list, Statistics, contains all of the statistics that you can graph for a queue. You are interested in graphing the current number of loads in the queue, which is abbreviated "Cur."

Note

Statistics that track values which change over time have two values: _a (absolute) and _r (relative, or reset, statistics). The Cur(rent) statistic, which you are graphing, only appears in the list once (without _a or _r), because it is not tracked over time. Resetting statistics is not covered in this textbook.

To define the current number of loads as the statistic to graph:

Step 11 ***Select*** Cur in the Statistics list, then ***click*** Add. Cur is added to the list of statistics being graphed.

Step 12 ***Click*** OK.

Step 13 ***Export*** the model.

Step 14 From the Model menu, ***select*** Run Model. ***Build*** the model. Before continuing the simulation, display your business graph, as explained in the next section.

Displaying a business graph

To display the timeline business graph:

Step 1 From the Control menu, ***select*** Business Graphics.

Step 2 ***Highlight*** Lathe_Queue in the list and ***click*** Display.

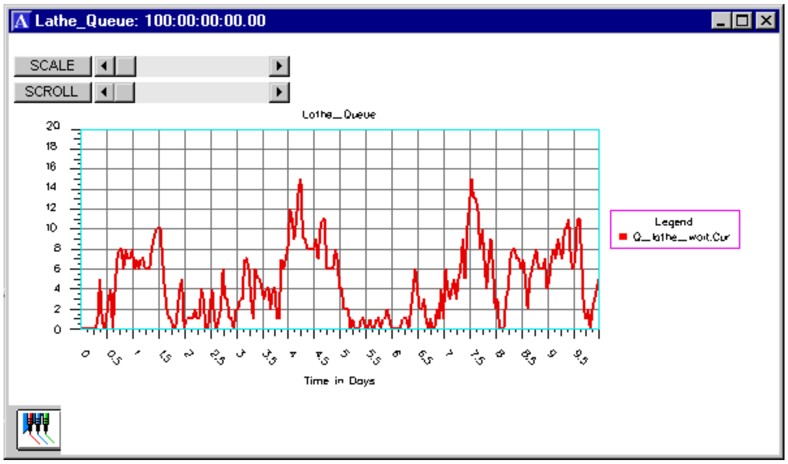
Step 3 ***Close*** the Business Graphics window.

Step 4 ***Run*** the simulation and ***watch*** the graph update as the simulation runs.

Tip

You can turn the model's animation off and the graph is still updated.

The timeline is shown below:



Timeline graph for Q_lathe_wait

Note

Notice that the values for the X and Y axes are reasonable. When creating your own graphs, if you have not estimated the axis values correctly, you can edit the X and Y values during the run by selecting Control > Business Graphics. Just remember that changes made to business graphs while running the model are not saved. You must make the changes in the process system to save them permanently.

Interpreting reports

You have already learned how to display summary statistics and single resource statistics in the Simulation window. In this chapter, you will learn how to display the statistics for an entire model by opening the model’s report. At the end of each run, the AutoMod software prints all summary statistics to a text file named “<modelname>.report”. The file is saved in the model directory (with the executable model) and can be opened using any text editor once the simulation has been closed.

To view the report for example model 5.1, finish running the model and do the following:

Step 1 If necessary, ***run*** the simulation to completion. ***Turn off*** the animation to speed up the run.

Step 2 From the Control menu, ***select*** Edit Model to close the simulation environment.

Important The simulation environment must be closed for the report file to be updated with the statistics for the current run.

Step 3 Using a text editor, such as Notepad, ***open*** the “examp51.report” file, located in the model directory. Change the file type from “.txt” to “All files” to see the .report file.

The report is opened. The next few sections explain how to interpret the report.

Version and clock information

The report begins with version and clock information about the simulation, as shown below:

```
*** AutoMod Version 9.1 - by AutoSimulations, Inc. ***
Model examp51
Statistics at Absolute Clock = 100:00:00:00.00, Relative Clock = 100:00:00:00.00
CPU time: Absolute: 3.104 sec, Relative: 3.104 sec
```

The first two lines of the report indicate the version of the AutoMod software that was used to run the model and the model name. The third line shows the length of the simulation, which is 100 days (recall that the simulation clock is displayed in the format days:hours:minutes:seconds.hundredths of seconds).

Note Reports display both absolute and relative times. In this model, the statistics are not reset, so both sets of statistics are the same and cover the entire simulation.

The fourth line measures the length of the simulation in CPU time (real time). On the computer that generated this example, the simulation took 3.104 seconds to run from start to finish. The length of time required to run the model varies depending on the speed of your computer’s processor, as well as whether or not animation is displayed, the display step setting if the animation *is* displayed, and whether or not you pause the model during a simulation. In this case, the model was run with animation turned off and without pausing the simulation.

Process statistics

Process statistics provide information about each process, such as how long loads were in the process and how many loads were in the process at the same time. (For definitions of these statistics, refer to “Displaying process system summary statistics” on page 2.15 of the “Using the Software” chapter.)

The process statistics for P_lathe, the first process in the model, are shown below (all times are in seconds):

```
Name              Total   Cur  Average Capacity   Max   Min   Util    Av_Time    Av_Wait
========================================================================================
P_lathe           14400     6    10.85       --    47     0     --    6512.76         --
```

A total of 14,400 loads were sent to P_lathe during the simulation. At the end of the simulation, there are six loads currently in P_lathe.

Now look at the queue statistics for the lathe:

```
Name              Total   Cur  Average Capacity   Max   Min   Util    Av_Time    Av_Wait
========================================================================================
    .
    .
    .
Q_lathe_wait      14400     6     9.09 Infinite    45     0     --    5452.51         --
Q_lathe           14394     2     1.92        2     2     0  0.958    1149.69    5452.51
```

Notice that Q_lathe_wait currently contains six loads and Q_lathe currently contains two loads. Initially, it may seem like there should be eight loads currently in the P_lathe process, not six. The reason there are only six is that the two loads in Q_lathe are in the process P_drill, as shown in the model logic below:

```
begin P_lathe arriving
   move into Q_lathe_wait /*limit = infinity*/
   move into Q_lathe      /*limit = 2*/
   get R_lathe
   wait for e 18 min      /*lathing time*/
   send to P_drill
end

begin P_drill arriving
   move into Q_drill_wait /*limit = 6*/
...
```

Loads do not move out of Q_lathe and into Q_drill_wait until they are in the process P_drill. The first action that loads execute in the procedure is to move into Q_drill_wait; however, this queue has limited capacity. Take a look at the statistics for Q_drill_wait:

```
Name              Total   Cur  Average Capacity   Max   Min   Util    Av_Time    Av_Wait
========================================================================================
Q_drill_wait      14392     6     2.93        6     6     0  0.489    1759.87      89.00
```

At the end of the simulation, the queue is currently filled to capacity. So, the two loads that are in Q_lathe are still in the process P_drill, waiting to move into Q_drill_wait. (You can also view the process statistics for P_drill to verify its current contents.)

The maximum number of loads that were in P_lathe at the same time reached 47 at least once over the 100 day simulation. The minimum number of loads was zero (there are zero loads in P_lathe at the beginning of the simulation, and perhaps at other times, as well).

Calculating the average time in system

One way of calculating the average time that loads spend in the system (cycle time) is to add up the time that loads spend in all of their processes. (Later in this textbook, you will learn an automatic way to track the time that loads spend in the system using load attributes.)

The statistics for the lathe, drill, and inspection processes are shown below:

```
Name               Total   Cur  Average Capacity   Max   Min   Util    Av_Time    Av_Wait
========================================================================================
P_lathe           14400     6    10.85 Infinite    47     0     --    6512.76         --
P_drill           14394     9     3.99 Infinite     9     0     --    2392.60         --
P_inspect         14385     0     0.10 Infinite     1     0     --      57.59         --
```

The average time that loads spent in the system can be calculated by adding the average time (Av_Time) that loads spent in P_lathe (6512.76), P_drill (2392.60), and P_inspect (57.59). Thus, the average time in system was 8962.95 seconds, or 149.38 minutes.

We can determine how much of the average time in the system was spent processing using the resource statistics:

```
Name        Total   Cur  Average Capacity   Max   Min   Util    Av_Time    Av_Wait   State
==========================================================================================
R_drill     14386     1     0.86        1     1     0  0.863     518.57       1.16   ----
R_operator  43156     0     0.39        1     1     0  0.385      77.11      28.70   ----
R_lathe     14394     2     1.92        2     2     0  0.958    1149.69       0.00   ----
```

The average time that loads spent processing can be calculated by adding the average time that loads spent using R_drill (518.57), R_operator (77.11), and R_lathe (1149.69). Thus, the average time that loads spent processing was 1656.38 seconds, or 27.60 minutes.

From these statistics, we can tell that loads spent the majority of their time waiting, as calculated by subtracting the time spent processing (27.60 minutes) from the total time in the system (149.38 minutes), which equals 121.78 minutes, or about two hours.

Now look at the processes that control the down times for resources. Look at the average time that the dummy loads spent in the P_break and P_down processes:

```
Name               Total   Cur  Average Capacity   Max   Min   Util    Av_Time    Av_Wait
========================================================================================
P_break               1     1     1.00 Infinite     1     0     -- 8640000.00         --
P_down                0     0     0.00 Infinite     0     0     --       0.00         --
```

The P_break average time is very high (100 days, or the entire length of the simulation). This is because the dummy load L_break remains in the P_break process (executing the `while 1 = 1 do` loop) throughout the entire simulation. The average time for P_down is zero, because we deleted the dummy load that was sent to the P_down process and replaced it with a resource cycle.

To determine the amount of down time for a resource, use the single resource statistics, as explained in “Verifying down times for resources” on page 5.24.

Queue statistics

Queue statistics indicate the total, average, and current number of loads in each queue, as well as the minimum and maximum loads that were in each queue at the same time. (For definitions of these statistics, refer to “Displaying queue summary statistics” on page 2.16 of the “Using the Software” chapter.)

The queues statistics for the example model are shown below (all times are in seconds):

```
Queue Statistics
Name               Total   Cur  Average Capacity   Max   Min   Util    Av_Time    Av_Wait
==========================================================================================
Space              14401     1     1.00 Infinite     2     0     --     599.96         --
Q_drill_wait       14392     6     2.93        6     6     0  0.489    1759.87      89.00
Q_drill            14386     1     0.91        1     1     0  0.906     544.27    1759.87
Q_inspect_wait     14385     0     0.00 Infinite     1     0     --       0.00         --
Q_inspect          14385     0     0.10        1     1     0  0.096      57.59       0.00
Q_lathe_wait       14400     6     9.09 Infinite    45     0     --    5452.51         --
Q_lathe            14394     2     1.92        2     2     0  0.958    1149.69    5452.51
```

From the queue statistics, we can see that a total of 14,401 loads were created in Space and one load was still there at the end of the simulation. The load that is still in Space is L_break, the dummy load that executes the P_break procedure; the L_break load remains in Space throughout the simulation. The number of loads in Space increases to two each time a new load is created, and then instantly decreases to one when the newly created load moves into the queue Q_lathe_wait (the queue has infinite capacity, so loads never have to wait to enter it). Consequently, the maximum number of loads that were in Space during the simulation is two.

There are currently six loads in Q_lathe_wait, which is below its average of 9.09 loads. The maximum number of loads that were in the queue at any one time was 45. That is understandable because the maximum number of loads in the P_lathe procedure was 47, and two of those loads were in the queue Q_lathe. The average time that loads spent in Q_lathe was 5452.51 seconds, or slightly more than 1.5 hours.

Notice that Q_inspect_wait has an average number of loads of 0.00 and an average time of 0.00. This is because as soon as the operator and drill resources are freed in the P_drill arriving procedure, the operator is claimed in the P_inspect arriving procedure, so loads never have to wait in Q_inspect_wait.

The average wait (Av_Wait) statistic for Q_lathe is 5452.51 seconds. That means that loads had to wait in Q_lathe_wait an average of 5452.51 seconds to enter Q_lathe.

Resource statistics

Resource statistics provide information about the utilization of each resource. (For definitions of these statistics, refer to “Displaying resource summary statistics” on page 2.17 of the “Using the Software” chapter.)

The resource statistics for the example model are shown below (all times are in seconds):

```
Resource Statistics
Name       Total   Cur  Average Capacity   Max   Min   Util    Av_Time    Av_Wait   State
==========================================================================================
R_drill     14386     1     0.86        1     1     0  0.863     518.57       1.16   ----
R_operator  43156     0     0.39        1     1     0  0.385      77.11      28.70   ----
R_lathe     14394     2     1.92        2     2     0  0.958    1149.69       0.00   ----
```

Resource R_lathe, which has a capacity of 2 loads, was claimed by an average of 1.92 loads during the simulation. The utilization is shown to be 0.958 (utilization is calculated by diving the average number of loads that claimed the resource by the resource’s capacity, in this case, 1.92/2). The average time that loads used resource R_lathe was 1149.69 seconds, or 19.16 minutes, slightly higher than the resource’s mean processing time of 18 minutes.

The average time that loads claimed R_operator was 77.11 seconds. To verify that this is a reasonable number, refer to the model logic. The operator is used three times with mean values of 2 minutes, 55 seconds, and 50 seconds, respectively. These values add up to 225 seconds. All loads use the operator three times, and 225/3 = 75 seconds, which is close to the average time of 77.11 seconds for this run.

Determining how many loads a resource has completed

For resources, the Total statistic shows how many loads have *claimed* the resource, including both loads that have finished using the resource and loads that are still (currently) using the resource. To calculate the number of loads a resource has *completed*, subtract the current (Cur) number from the total (Total) number.

For example, to determine how many loads R_drill has completed, subtract the loads that are currently using the resource (1) from the Total (14386), for 14385 completed loads.

Scheduling down times for when a resource is idle

In manufacturing, both unplanned and planned down times occur. Machine failures are unplanned down times, while preventative maintenance events are planned. Seldom does preventative maintenance interrupt processing; rather, the preventative maintenance is performed between jobs.

To model a planned unavailable activity, such as preventative maintenance, you can use a resource cycle with an unavailable time that occurs only when a resource is idle. A planned delay is created using the `wait until idle` action in the Action column of a resource cycle.

The following example shows how to model preventative maintenance and other planned delays that you want to occur only when a resource is idle.

Example 5.2: Modeling a planer

In example model 5.2, jobs are created with an interarrival time that is exponentially distributed with a mean of 10 minutes. Jobs first move into an infinite-capacity queue where they wait to be leveled and smoothed by a planer. When the planer becomes available, jobs move into a single-capacity processing queue, where they are leveled and smoothed by the planer. The planer requires a time that is exponentially distributed with a mean of 7 minutes to level and smooth each job.

The planer fails randomly; the time between failures is exponentially distributed with a mean of 120 minutes. The time required to repair the planer is also exponentially distributed with a mean of 10 minutes.

Also, after the planer has operated for 60 minutes, it is stopped and cleaned the next time it becomes idle. The cleaning takes a time that is normally distributed with a mean of 6 minutes and a standard deviation of 30 seconds.

You will simulate this system for 100 days.

Defining a resource cycle that delays until idle

The base version of example model 5.2 already has the necessary load creation specification, queues, and model logic defined for the planing operation. In addition, a resource cycle has already been created to model the planer's failures. But you need to create a second resource cycle to model the cleaning operation. In the new resource cycle, use the `wait until idle` action to delay the down time until the resource is idle.

To create the cleaning resource cycle:

Step 1 ***Import*** a copy of the *base* version of example model 5.2.

Step 2 ***Click*** Resources on the Process System palette. The Resources window opens.

Step 3 ***Click*** New to the right of the Resource Cycles list to define a new resource cycle. The New Resource Cycle window opens.

Step 4 In the Name text box, ***type*** "RC_clean" and ***click*** OK, Edit. A new resource cycle edit table opens.

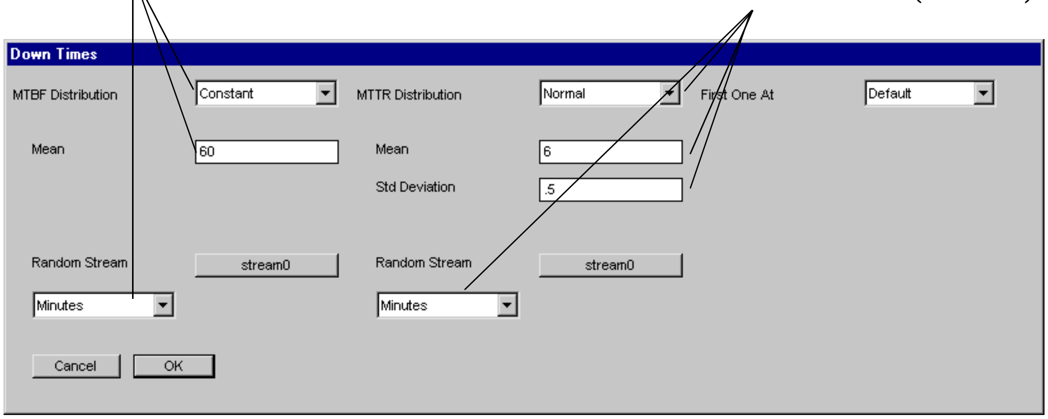
Step 5 ***Click*** MTBF/MTTR. The Down Times window opens.

The time between cleanings is a constant 60 minutes, and the time required to clean the planer is normally distributed with a mean of 6 minutes and a standard deviation of 30 seconds.

Step 6 ***Enter*** the distributions in the Down Times window, as shown below:

The mean time between cleanings is a constant 60 minutes

The time required to clean the planer is normally distributed with a mean of 6 minutes and a standard deviation of 30 seconds (.5 minutes)



Down Times

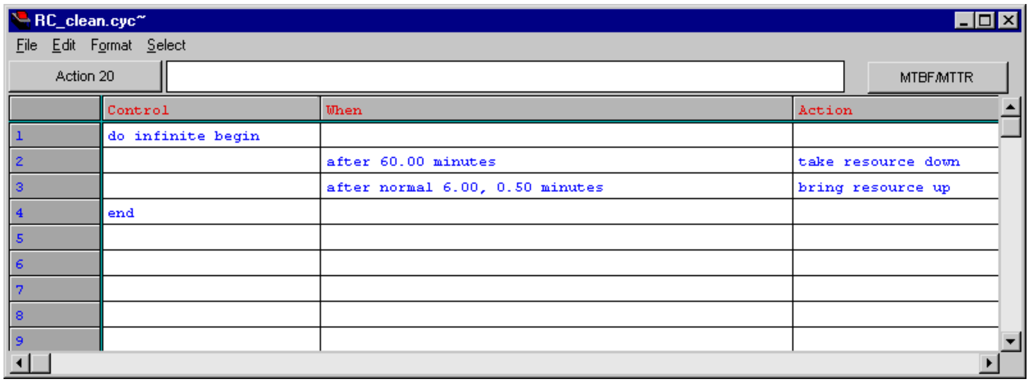
MTBF Distribution: Constant
Mean: 60
Random Stream: stream0
Minutes

MTTR Distribution: Normal
Mean: 6
Std Deviation: .5
Random Stream: stream0
Minutes

First One At: Default

Defining the planer's cleaning times

Step 7 *Click* OK to automatically generate the resource cycle in the edit table. The generated resource cycle edit table opens, as shown below:



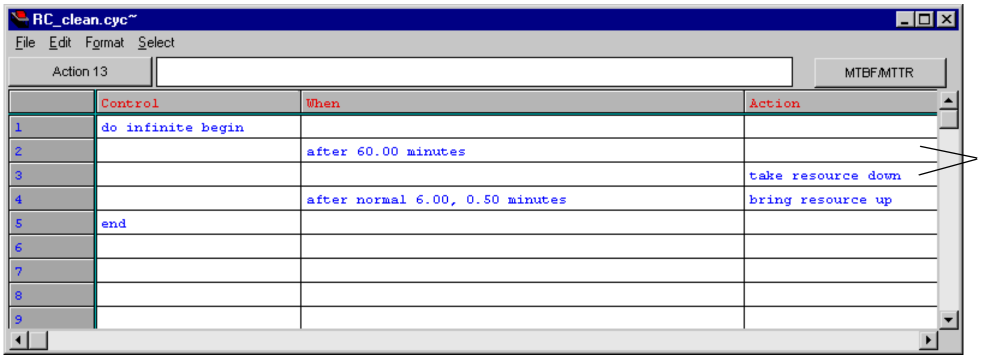
	Control	When	Action
1	do infinite begin		
2		after 60.00 minutes	take resource down
3		after normal 6.00, 0.50 minutes	bring resource up
4	end		
5			
6			
7			
8			
9			

Generated resource cycle edit table

Now you need to edit the resource cycle so that it only occurs when the resource is idle. To do this, you need to insert a new line between the two actions and move the `take resource down` action to the new line. You can then add a new action to line 2 that delays the down action until the resource is idle.

Step 8 ***Highlight*** row 3 and ***select*** Insert from the Edit menu. A new row is inserted.

Step 9 ***Click*** the Action in row 2 (“take resource down”) and ***select*** Cut from the Edit menu. ***Click*** the Action in row 3 and select ***paste*** from the Edit menu. The resource cycle should look like the one shown below:



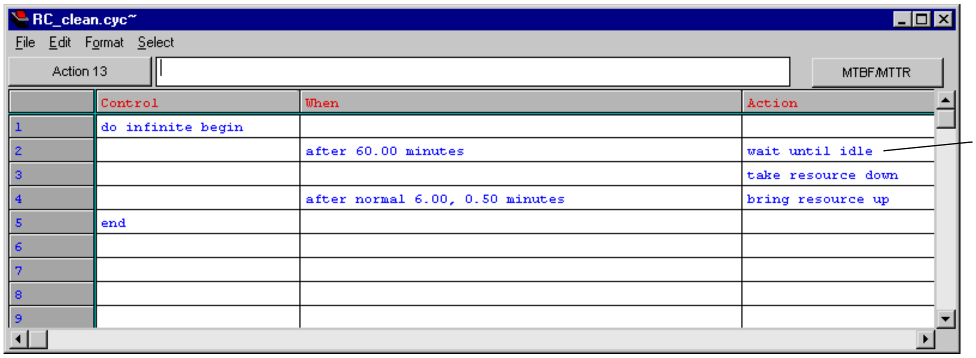
	Control	When	Action
1	do infinite begin		
2		after 60.00 minutes	
3			take resource down
4		after normal 6.00, 0.50 minutes	bring resource up
5	end		
6			
7			
8			
9			

The action is moved from row 2 to row 3

Cutting and pasting an action

Step 10 ***Double-click*** in row 2’s Action cell. The Action window opens.

Step 11 ***Select*** “wait until idle” in the Action list and ***click*** Set. The action is inserted in the resource cycle edit table, as shown below:



	Control	When	Action
1	do infinite begin		
2		after 60.00 minutes	wait until idle
3			take resource down
4		after normal 6.00, 0.50 minutes	bring resource up
5	end		
6			
7			
8			
9			

The wait until idle action is inserted in the resource cycle edit table

The completed resource cycle edit table

The resource cycle is now complete. After a constant 60 minutes, the resource cycle waits until the resource is idle and then takes down the resource. After a time that is normally distributed with a mean of 6 minutes and a standard deviation of 30 seconds, the resource cycle brings up the resource. The cycle repeats continuously throughout the simulation.

Step 12 From the File menu, ***select*** Save.

Step 13 From the File menu, ***select*** Quit to close the edit table and the list of actions.

Now that you have defined the resource cycle, you need to attach it the planer resource.

Attaching the resource cycle to the planer

To attach the resource cycle to the planer, do the following:

Step 1 In the Resources window, ***select*** R_plane and ***click*** Edit. The Edit A Resource window opens.

Step 2 ***Click*** Add to the right of the Attached Resource Cycle list. The Add Resource Cycle window opens.

Step 3 ***Click*** Add to attach the cycle RC_clean to the resource R_plane. The resource cycle appears in the Attached Resource Cycle list in the Edit A Resource window. (The resource is now controlled by two resource cycles, RC_fail and RC_clean.)

Step 4 ***Click*** OK to close the Edit A Resource window.

Step 5 ***Export*** the model.

Step 6 ***Select*** Run Model and ***build*** the model. Before continuing the simulation, you are going to set an alarm, as described in the next section, to help you verify that the resource is being taken down correctly.

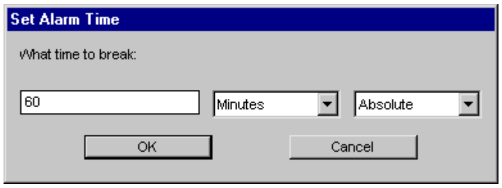
Setting an alarm to pause the simulation at a specific time

The first cleaning time in the model is scheduled to occur after a constant 60 minutes of simulation time. You can set an alarm to automatically pause the simulation after 60 minutes, allowing you to watch the animation from that point to see whether the down event occurs when the resource is idle.

To set the alarm for 60 minutes:

Step 1 From the Control menu, ***select*** Set Alarm. The Set Alarm Time window opens.

Step 2 In the “What time to break” text box, ***type*** “60” and ***select*** minutes from the drop-down list, as shown below:



Set Alarm Time

What time to break: 60 Minutes Absolute

Set Alarm Time window

Step 3 ***Click*** OK to close the Set Alarm Time window. A message is printed to the Message window indicating that an alarm has been set for one hour of simulation time.

Step 4 ***Press*** “g” to turn off the graphics.

Step 5 ***Press*** “p” to continue the simulation. At 60 minutes in the simulation, the alarm rings and the model is paused. The animation is also updated, so you can see that the resource R_plane’s graphic is green, indicating that it is currently processing a load.

Step 6 ***Press*** “g” to turn on the graphics, then ***press*** “w” to turn graphics solid.

Step 7 ***Press*** “u” to open the Display Speed window, and ***change*** the display step to 30 seconds. ***Press*** Enter to close the window.

Step 8 ***Press*** “p” to continue the simulation. The R_plane resource continues to process its current load. It is not until almost 24 minutes later that the resource finishes processing this load. Once the resource has finished and becomes idle, it is taken down by the resource cycle. The resource’s color changes to red, indicating that it is down.

Tip ☞ Another way to pause models automatically is to use a **breakpoint**, which stops the simulation whenever a certain condition becomes true (such as a resource going down, coming up, becoming idle, and so on). You could set a breakpoint for the resource R_plane to pause the simulation any time the resource is going down by selecting Breakpoint from the Resources menu and selecting Going down. The model automatically pauses whenever the resource goes down.

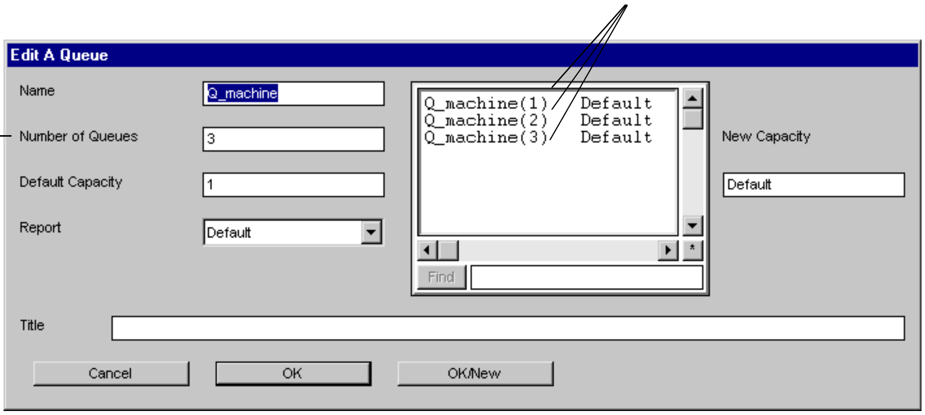
Modeling similar processes using arrayed entities

It is very common in manufacturing to have several people or pieces of equipment that can do the same type of work. Sometimes, those people or pieces of equipment form parallel lines or similar work cells. For example, suppose a facility has two assembly lines. Once a product starts down one line, it cannot jump back and forth between the lines—it must stay on the same line. But the two lines use the same types of equipment and operators, and the processes take the same amount of time. What is the best way to model this?

You could model each line individually, creating the necessary resources, queues, and operators, and writing two procedures that accomplish the same thing. But your model would contain a lot of duplication, and editing the model would be difficult, because every change would need to be made in two places (or more, if you added additional assembly lines).

A better approach is to create multiple “copies” of each entity using an array. An **array** is a group of entities (resources, queues, processes, and so on) that share similar characteristics, such as capacity, down times, or arriving procedures. Arrays also allow you to align multiple entities of different types to form assembly lines or similar work cells.

To define an arrayed queue, resource, process, or other entity, set the number of entities to greater than one when you define the entity. The arrayed entities are automatically created and named using the format “entityname(#),” as shown below:



The arrayed queues are automatically created and named

In an array, the number of queues is greater than one

Edit A Queue

Name: Q_machine

Number of Queues: 3

Default Capacity: 1

Report: Default

```
Q_machine(1)   Default
Q_machine(2)   Default
Q_machine(3)   Default
```

New Capacity: Default

Title:

Arrayed queues

You can then use the arrayed entities in your logic. For example, to move a load into the first queue in the array, use the syntax:

```
move into Q_Machine(1)
```

To move a load into the second queue in the array, use the syntax:

```
move into Q_Machine(2)
```

To illustrate how arrays can make modeling easier, the following example shows how to model a simple grinding operation two ways, with and without arrays.

Example 5.3: Grinding operation

Example model 5.3, which is a grinding operation, has loads arrive in the system with an interarrival time that is exponentially distributed with a mean of 6 minutes. The grinding operation consists of three grinding machines. Loads must be processed by all three grinding machines in sequential order.

Each grinder is preceded by an infinite-capacity waiting queue (where loads wait for the grinding machine to become available) and a single-capacity processing queue (where loads are processed by the grinding machine).

A single operator loads each grinder for a time that is normally distributed with a mean of 20 seconds and a standard deviation of 3 seconds. The same operator unloads each grinder for a time that is exponentially distributed with a mean of 30 seconds.

You will simulate this system for seven days.

Modeling example 5.3 using individual entities

The first approach shows how to model the system without using arrays. The model contains three processes, three grinding resources, an operator, and several queues:

```
begin P_FirstGrinder arriving procedure
   move into Q_FirstGrinder_wait    /* Move into the waiting queue */
   move into Q_FirstGrinder         /* Move into the processing queue */
   get R_FirstGrinder               /* Claim the grinder */
   use R_Operator for n 20, 3 sec   /* Loading */
   wait for 4 min                   /* Grinding */
   use R_Operator for e 30 sec      /* Unloading */
   free R_FirstGrinder              /* Free the grinder */
   send to P_SecondGrinder          /* Send to the next process */
end

begin P_SecondGrinder arriving procedure
   move into Q_SecondGrinder_wait   /* Move into the waiting queue */
   move into Q_SecondGrinder        /* Move into the processing queue */
   get R_SecondGrinder              /* Claim the grinder */
   use R_Operator for n 20, 3 sec   /* Loading */
   wait for 4 min                   /* Grinding */
   use R_Operator for e 30 sec      /* Unloading */
   free R_SecondGrinder             /* Free the grinder */
   send to P_ThirdGrinder           /* Send to the next process */
end

begin P_ThirdGrinder arriving procedure
   move into Q_ThirdGrinder_wait    /* Move into the waiting queue */
   move into Q_ThirdGrinder         /* Move into the processing queue */
   get R_ThirdGrinder               /* Claim the grinder */
   use R_Operator for n 20, 3 sec   /* Loading */
   wait for 4 min                   /* Grinding */
   use R_Operator for e 30 sec      /* Unloading */
   free R_ThirdGrinder              /* Free the grinder */
   send to die                      /* Remove load from the system */
end
```

The logic in the three processes is very similar. In fact, the only difference is which queues and resources are being used and the process to which the load is sent. Imagine how much logic you would need to model 50 grinding machines instead of only 3.

A more efficient approach is to model the system using arrays, as discussed next.

Modeling example 5.3 using arrayed entities

The following example shows how to model example 5.3 using arrayed processes, arrayed resources, and arrayed queues. There are far fewer lines of logic in this approach than the previous approach, and once written, the logic in this approach can be used to model any number of grinding machines without being changed.

This approach uses some new syntax (`procindex` and `if...then...else`), which is discussed in the next two sections.

```
begin P_grinder arriving                /*P_grinder is arrayed by 3*/
   move into Q_grinder_wait(procindex)
      /*Move into waiting queue 1, 2, or 3*/
   move into Q_grinder(procindex)
      /*Move into processing queue 1, 2, or 3 */
   get R_grinder(procindex)             /*Claim grinder 1, 2, or 3*/
   use R_operator for n 20, 3 sec       /*Loading*/
   wait for 4 min                       /*Grinding*/
   use R_operator for e 30 sec          /*Unloading*/
   free R_grinder(procindex)            /*Free the grinder*/
   if procindex = 3 then send to die    /*If 3, processing is finished*/
   else send to P_grinder(procindex+1)  /*If 1 send to 2, if 2 send to 3*/
end
```

Using procindex to align arrayed entities

When you array a process, such as P_grinder, you can define a single arriving procedure that is shared by each of the arrayed processes. For example, if you array the process P_grinder by three, you only need to define one arriving procedure, and it will be used by loads that are sent to P_grinder(1), P_grinder(2), and P_grinder(3). But how will you know which of the three processes a load is in while it is executing the arriving procedure? AutoMod automatically determines which arrayed process a load is in and stores it in the attribute **procindex**, which is an integer attribute.

For example, assume a load is sent to the process P_grinder(1). When the load executes the arriving procedure, the value of `procindex` is 1. The next time the load executes the procedure, it is in P_grinder(2), so `procindex` is 2, and so on.

The `procindex` attribute is frequently used to align arrayed entities of different types. In this grinding example, the resource R_grinder and the queues Q_grinder_wait and Q_grinder are all arrays. A load moves into queues and claims a resource based on the value of `procindex`. Loads in P_grinder(1) have a `procindex` of 1. Therefore, the loads move into Q_grinder_wait(1), then Q_grinder(1), and claim R_grinder(1). When the loads are in P_grinder(2), they use the second of each arrayed queue and resource, and then when in P_grinder(3), the loads use the third of each arrayed queue and resource. Although the loads are being processed by similar equipment and taking similar delays, they use the correct equipment for each process.

Writing conditional syntax using if...then...else

In example model 5.3, loads need to go through three processes. In the non-arrayed solution, this is accomplished using a send action at the end of each procedure that sends the load to the next process. In the arrayed approach, however, you cannot use three different send actions. You need to know which process you are in currently to know the process to which to send the load next. The solution is to use if...then...else syntax to perform actions conditionally:

```
if procindex = 3 then send to die    /*If 3, processing is finished*/
else send to P_grinder(procindex+1)  /*If 1 send to 2, if 2 send to 3*/
```

The if...then and else syntax says, “If a certain condition is true, then execute an action. Otherwise, execute a different action.” In the case of the send action, procindex is used to determine the current process and to increment the number of the process to which the load is sent. If the load is in process three, then the load is sent to die.

So when the load is in P_grinder(1), procindex is 1. The if condition is not true, so it is ignored. The next line, beginning with else, sends the load to process P_grinder (1+1), or P_grinder(2). The load is next sent to P_grinder(3). When the load in P_grinder(3) executes the if line, the condition is true, so the load is sent to die.

You can use the if...then syntax with or without the else clause. For example, if you only have one condition to check, you can use the following syntax:

```
move into Q_process
if this load type = L_oversized then
   wait for u 30, 15 sec /* oversized load takes extra setup */
use R_worker for e 10, 2 min
```

If the load that is executing the procedure is of type L_oversized, then the load takes a delay that other loads do not, representing an additional setup time due to the load’s size.

If you want to test for more than one condition, you can use else. You can also check multiple conditions using else...if syntax. For example:

```
begin P_sort arriving procedure
   if this load type = L_can then
      send to P_canner
   else if this load type = L_bottler then
      send to P_bottler
   else send to P_scrap
end
```

If the load that is executing the procedure is of type L_can, then the load is sent to the process P_canner. If not, the load is tested for another condition, that is, whether it is type L_bottler. If so, the load is sent to process P_bottler. If the load is neither of these types, it is scrapped.

You can use as many else...if statements as necessary to test for all conditions.

Creating example model 5.3

You are now ready to create example model 5.3 using arrays.

To create the example model:

Step 1 ***Open*** the AutoMod software (if it is not already open) and ***create*** a new model named “examp53”.

Step 2 ***Create*** a new process system named “proc”.

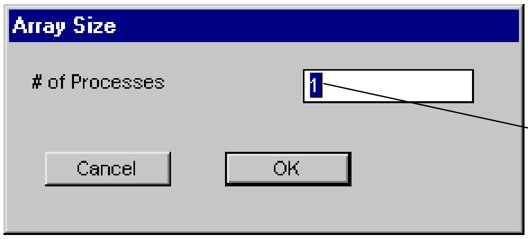
Step 3 ***Create*** a new source file named “logic.m”.

Step 4 ***Edit*** the source file and ***type*** the following model logic:

```
begin P_grinder arriving                /*P_grinder is arrayed by 3*/
   move into Q_grinder_wait(procindex)
      /*Move into waiting queue 1, 2, or 3*/
   move into Q_grinder(procindex)
      /*Move into processing queue 1, 2, or 3 */
   get R_grinder(procindex)             /*Claim grinder 1, 2, or 3*/
   use R_operator for n 20, 3 sec       /*Loading*/
   wait for 4 min                       /*Grinding*/
   use R_operator for e 30 sec          /*Unloading*/
   free R_grinder(procindex)            /*Free the grinder*/
   if procindex = 3 then send to die    /*If 3, processing is finished*/
   else send to P_grinder(procindex+1)  /*If 1 send to 2, if 2 send to 3*/
end
```

Step 5 ***Save*** and ***quit*** the source file. The Error Correction window opens, indicating that P_grinder is undefined.

Step 6 ***Select*** Define, then ***click*** Define As to define P_grinder as a process. The Array Size window opens.



Change the number of processes to 3 to create an array

Defining an arrayed process in BEdit

Step 7 In the # of Processes text box, ***type*** “3” and ***click*** OK. The Error Correction window opens, indicating that Q_grinder_wait is undefined.

Step 8 ***Select*** Queue in the “Define As” drop-down list, then ***click*** Define As. The Define A Queue window opens.

Step 9 In the Number of Queues text box, ***type*** “3” to create an array of three queues. In the Default Capacity text box, ***type*** “i” for infinite. ***Click*** OK to close the Define A Queue window. The Error Correction window opens, indicating that Q_grinder is undefined.

Step 10 ***Repeat*** steps 8 and 9 to define Q_Grinder as an array of 3 queues; however, ***define*** the capacity as 1 (instead of infinite). ***Click*** OK to close the Define a Queue window. The Error Correction window opens, indicating that R_grinder is undefined.

Step 11 ***Select*** Resource in the Define As drop-down list, then ***click*** Define As. The Define A Resource window opens.

Step 12 In the Number of Resources text box, ***type*** “3” to create an array of three resources. ***Click*** OK to close the Define A Resource window. The Error Correction window opens, indicating R_operator is undefined.

Step 13 ***Click*** Define As to define R_operator as a resource. The Define A Resource window opens. ***Click*** OK to define R_operator as a single resource with a capacity of one.

Tip

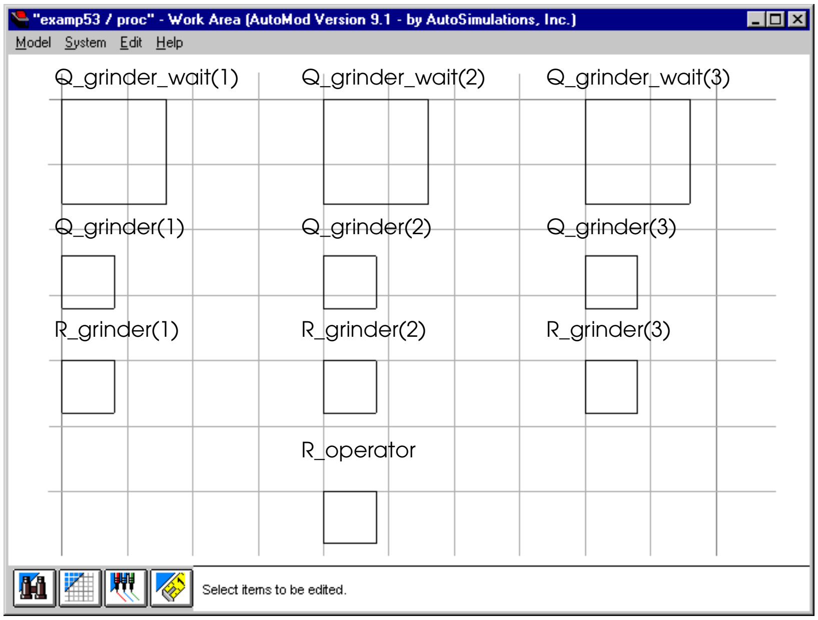
If you accidentally define an entity incorrectly, comment the code, close the source file, and edit the entity from the Process System palette. Then edit the source file and remove the comments.

Step 14 ***Export*** the model.

Now that you have defined all the entities necessary for the model, you are ready to place the entity graphics and define the load creation specification.

Placing graphics for arrayed entities

The graphics for example 5.3 are shown below:



Example 5.3 layout

Placing graphics for arrayed resources and queues requires one additional step to the procedure you have already learned.

To place graphics for arrayed queues:

Step 1 On the Process System palette, ***click*** Queues.

Step 2 In the Queues window, ***select*** Q_grinder_wait, then ***click*** Edit Graphic. The Edit Queue Graphics window opens.

Step 3 ***Select*** Q_grinder_wait(1) in the list and ***click*** Place.

Step 4 In the Work Area window, ***drag*** the queue until it is positioned as shown in the layout above.

Step 5 ***Place*** the graphics for the remaining queues in the array, then ***click*** Done.

Step 6 ***Place*** the remaining queue and resource graphics as shown above.

Defining the load creation specification

Now you need to create loads to send through the grinding process.

To define the load creation specification, do the following:

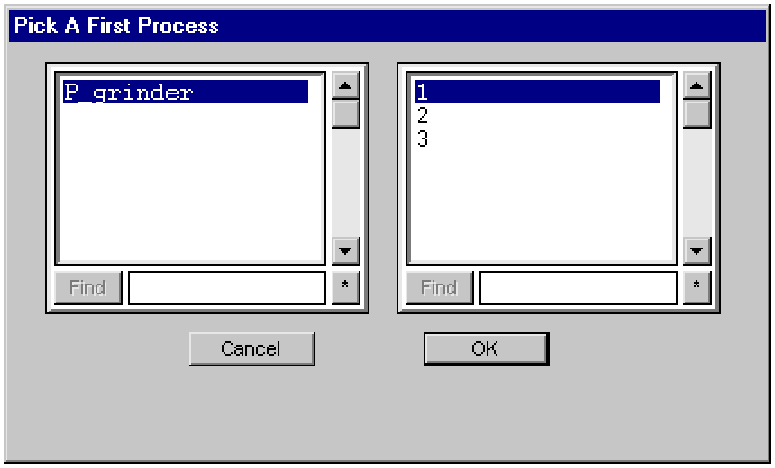
Step 1 ***Create*** a new load type called “L_parts.”

Step 2 ***Define*** a creation specification to generate loads with an interarrival time that is exponentially distributed with a mean of 6 minutes.

Step 3 ***Click*** First Process in the Define a LoadType window. The Pick a First Process window opens.

Step 4 ***Select*** P_grinder from the list on the left. The numbers for each arrayed process appear in the list on the right.

Step 5 ***Select*** 1 in the list on the right to choose P_grinder(1) as the first process for loads, as shown below:



Picking an arrayed process as the first process

Step 6 ***Click*** OK.

Step 7 ***Click*** OK to close the Define a Creation Specification window and ***click*** OK to close the Define a LoadType window.

Defining the run control

You want to simulate the system for seven days.

To define the run control:

Step 1 ***Click*** Run Control on the Process System palette.

Step 2 ***Define*** a snap of 7 days.

Running the example model

You are now ready to run the model and look at the results.

Step 1 ***Export*** the model and ***select*** Run model. The simulation environment opens.

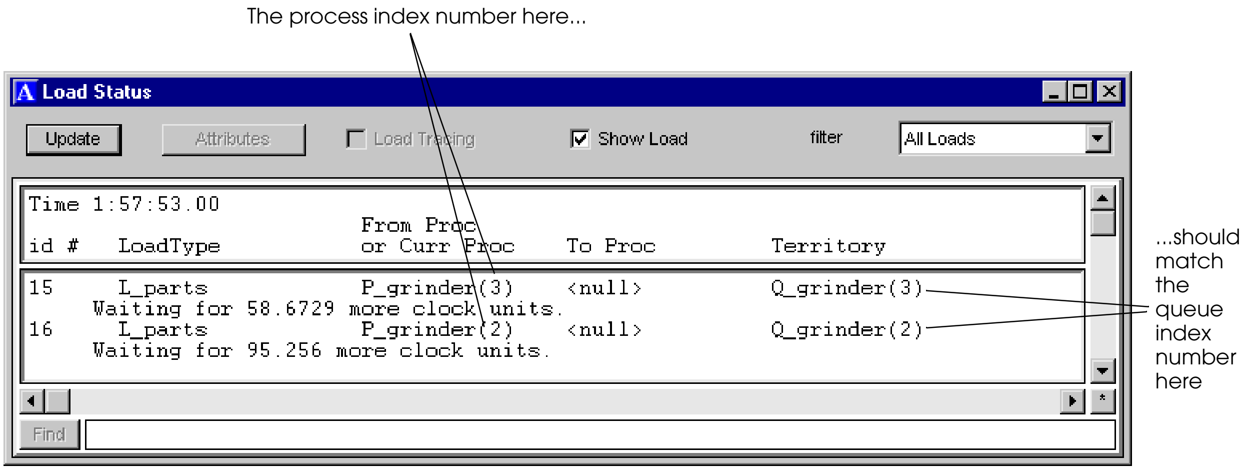
Step 2 ***Press*** “w” to turn graphics solid.

Step 3 ***Press*** “p” to continue the simulation.

In this grinding example, each load is sent to P_grinder(1). In that process, the load moves into Q_grinder_wait(1) then Q_grinder(1), based on a `procindex` of 1. The load uses R_grinder(1). The load is then sent to the next process, P_grinder(2), and the `procindex` value is now 2. The load uses the second of each arrayed queue and resource. The load is sent to the third process and uses the third of each arrayed queue and resource. The load is then sent to die.

To verify that the model is working correctly, use the Load Status window:

Step 4 ***Select*** Load Status from the Loads window. The Load Status window opens.



```
Time 1:57:53.00
                         From Proc
id #   LoadType          or Curr Proc    To Proc         Territory
15     L_parts           P_grinder(3)    <null>          Q_grinder(3)
     Waiting for 58.6729 more clock units.
16     L_parts           P_grinder(2)    <null>          Q_grinder(2)
     Waiting for 95.256 more clock units.
```

The process index number here...

...should match the queue index number here

Load Status window

Step 5 ***Verify*** that for each load, the index of the process name matches the index of the queue listed as the territory.

Step 6 As your model runs, ***click*** Update in the Load Status window periodically to get the current state of the simulation in the Load Status window.

Determining when to use arrays versus multiple-capacity resources

In the beginning of this chapter, you learned how to define a single resource that has the ability to process more than one item at a time (see "Defining a resource" on page 5.8). Now you have learned how to model similar resources using arrays. For situations in which you have multiple resources doing similar work, when would you use a multiple-capacity resource, and when would you use an arrayed resource? Use the following guidelines to help you decide when to use each approach.

Use **multiple-capacity resources** when you are modeling a single person or machine that can do more than one thing at a time.

You would also use multiple-capacity resources when you want to select the first available resource from a group of resources. For example, when modeling service centers where people are servicing other people, the people being helped often go to the next available attendant. In these cases, you must use a multiple-capacity resource to model the service attendants rather than an arrayed resource.

Use **arrayed resources** to select work from a group of resources based on criteria other than availability, such as evenly distributing work to various lines. (In AutoMod, distributing work evenly is modeled using the `nextof` distribution; see "Using the nextof distribution with arrayed entities" on page 5.47.)

Arrayed resources are also more realistic graphically; you can see all of the resources separately, such as six attendants, four machines, and so on. If you used a multiple-capacity resource, you would only see one graphic, no matter how many items the resource could process concurrently.

Selecting entities alternately using the nextof distribution

The **nextof** distribution is available in AutoMod to simulate alternating, or **round-robin**, selection (1, 2, 3, 1, 2, 3...).

For example, to send one-half of loads to P_1 and the other half to P_2, alternating between the two, use the `nextof` distribution:

```
   send to nextof(P_1, P_2)
```

The `nextof` distribution can also be used with resources, queues, and other entities. To move a load into one of several queues alternately:

```
   move into nextof(Q_1, Q_2, Q_3)
```

The first load to execute this action moves into Q_1, the second load moves into Q_2, the third Q_3, the fourth moves into Q_1, and so on. The alternating cycle continues throughout the simulation.

Using the nextof distribution with arrayed entities

The `nextof` distribution does *not* guarantee that multiple arrayed processes, queues, and resources are aligned, like `procindex` does (see “Using procindex to align arrayed entities” on page 5.40). Loads are delayed for varying amounts of time at queues or resources because of unavailable capacity or different processing times. When using `nextof`, the order in which loads select queues or resources depends on which loads execute the actions in a procedure first. Because loads are delayed for varying amounts of time, loads may execute actions “out of order,” resulting in cases where loads “jump lines” and use a non-corresponding entity.

To use the `nextof` distribution while also aligning arrayed entities, use arrayed processes in addition to arrayed queues and resources. Send loads alternately to the arrayed process using the `nextof` distribution, then use `procindex` to select other arrayed entities.

For example, suppose your system has three lines, which you are modeling as three resources that each have a waiting queue and a processing queue. You want to alternately send loads to one of the three lines. You could use the approach shown below:

```
begin P_send arriving
   send to nextof(P_machine(1), P_machine(2), P_machine(3))
end
begin P_machine arriving
   move into Q_mach_wait(procindex)
   move into Q_mach(procindex)
   use R_operator(procindex) for e 5 min
   send to die
end
```

The first load is sent to P_machine(1), so its `procindex` value is 1. The load moves into Q_mach_wait(1), moves into Q_mach(1), uses R_operator(1), then is sent to die. The second load is sent to P_machine(2), so its `procindex` value is 2. It uses the second of the queues in the arrays and the second resource in the resource array. The third load is sent to P_machine(3), so it uses the third set of entities. The fourth load is sent to P_machine(1), beginning the cycle again.

Summary

This chapter contains some core concepts that you will use regularly, including:

- Defining resources and queues
- Interpreting statistics
- Defining down times
- Using arrayed entities
- Using the nextof distribution

As you start to develop your own models from scratch, you will find it helpful to use a model-building methodology, such as:

Step 1 ***Define*** all resources and queues and ***place*** their graphics.*

Step 2 ***Define*** processes.

Step 3 ***Create*** loads and ***define*** creation specifications.

Step 4 ***Create*** a source file and ***write*** the model logic.*

Step 5 ***Define*** the run control.

Step 6 ***Export*** and ***run*** the model.

* Another approach is to define all of the logic first (step 4), then complete the steps as shown here.

Exercises

Exercise 5.1

Copy the *final* version of example model 5.1 to a new directory and use the copied model to answer the following questions with respect to the **average number of loads in P_lathe and P_drill:**

a) What is the difference in the above statistics when the capacity of queue Q_drill_wait is six (its current value), five, or seven widgets?

b) What is the impact on the statistics when the capacity of queue Q_drill_wait is changed to eight widgets, and the mean processing time of resource R_drill increases to 4.5 minutes (from 4 minutes)?

c) Using the same configuration that you set in model (b) above, what is the impact on the statistics if the operator takes a 45-minute lunch break, instead of 30 minutes, and skips the afternoon break (the morning break remains the same)?

Exercise 5.2

Create a new model containing three independent processes, each with one operator. In each process, loads first move into an infinite-capacity waiting queue, then into a single-capacity processing queue. Loads are processed by each operator for the times shown below:

Worker_A does the job in a normally distributed time with a mean of 60 minutes and a standard deviation of 10 minutes.

Worker_B does the job according to a triangular distribution with a minimum of 30 minutes, most likely value of 60 minutes, and a maximum of 90 minutes.

Worker_C does the job in a uniformly distributed time between 30 and 90 minutes.

Send loads to each process at the same rate (an exponentially distributed interarrival time with a mean of 68 minutes) to understand the difference in the processing times of the three workers.

Answer the following:

a) After running the model for 10,000 days, what was the average number of loads in each process?

b) Determine the variance and the mean of each processing time distribution. Explain what effect the variance of each distribution has on the total number of loads sent to each process.

Help

Formulas for calculating the variance (the square of the standard deviation) and mean for each distribution can be found in the AutoMod Syntax Help by searching the index for “variance” or “mean.”

Exercise 5.3

Create a new model to simulate the following vehicle licensing facility:

Customers arrive at the local Division of Motor Vehicles office according to an exponential distribution with a mean of 3.5 minutes. The customers wait in a line to be helped by one of two checkers (whoever is available first). The checkers each take an exponentially distributed 5.25 minutes to check a customer’s paperwork. The customers move to the checker’s desk when being helped.

After their paperwork is verified, customers wait in their current location for the next available clerk. There are six clerks. When a clerk becomes available, the customer moves to the clerk’s counter. The clerks’ processing times are exponentially distributed with a mean of 15 minutes.

The office is open 8 hours per day. Run the model for one day.

a) Make a timeline graph of the arrival line that shows the average and maximum number of people waiting to be helped by a checker.

b) What were the average and maximum number of people waiting to be helped by a checker? How many people were in the system at the end of the day?

c) What is the effect on the measures in (b) if the entire office takes a 15 minute lunch break 3.5 hours into the day?

Exercise 5.4

Apex Corporation produces widgets. The company has decided to use a simulation model to improve its system. The system consists of the following:

- A widget component waiting area with infinite capacity.
- Three single-capacity widget assembly machines (modeled as arrayed resources); each machine has its own queue for processing.
- An inspection waiting area with a capacity of three.
- One inspector and an inspection queue with a capacity of one.

Boxes of widget components arrive at the production facility with an exponential interarrival time with a mean of 5 minutes. Upon arrival, widget components wait for one of the three assembly machines. The assembly machines are assigned in round-robin order. The assembly machines process the components for a time that is normally distributed with a mean of 12 minutes and a standard deviation of 2 minutes.

Once assembled, widgets attempt to enter the waiting queue for inspection. Only three widgets can wait in the queue. If the queue is full, widgets must wait at their current assembly machine until space is available. From the waiting queue, loads move into the inspection queue. The inspector inspects each widget for a time that is uniformly distributed between one and five minutes; the inspector can inspect only one load at a time.

The inspector takes 5 minute breaks after working for a time that is uniformly distributed between 40 and 60 minutes. Print a message to the Message window each time that the inspector starts a break and another message when that break is finished.

Apex wants the system to be simulated for 24 hours.

Create a model of this system and determine the following information for Apex:

a) What is the average time that loads spend waiting for each assembly machine's processing queue? What is the average time each machine spends assembling a widget?

b) What is the average time that widgets wait in the inspection waiting queue?

c) How many widgets does the inspector complete in the 24-hour period?

d) What is the average number of widget components in the waiting queue for the assembly machines?

e) What is the utilization of the inspector?

Exercise 5.5

Create a new model in which jobs are created every 10 minutes. Jobs first move into an infinite-capacity queue where they await processing. Jobs are processed by a single-capacity drill for a time that is normally distributed with a mean of 7.5 minutes and a standard deviation of 20 seconds. The drill has its own queue where jobs are processed.

Using resource cycles, model the following failure and maintenance events:

The drill breaks down according to an exponential distribution with a MTBF of 2 hours. Repair time is also exponentially distributed with a MTTR of 15 minutes.

The drill is inspected every hour, when it is idle, for an exponentially distributed time with a mean of two minutes.

Chips build up in the drill until it requires cleaning. The drill requires cleaning after a processing time that is normally distributed with a mean of 2 hours and a standard deviation of 10 minutes. Cleanings are delayed (if necessary) until the drill is idle. Cleanings are completed in five minutes.

Simulate the system for 28 days.

a) What was the average and maximum number of jobs waiting to be drilled?
b) If the MTBF for the break downs could be lengthened to three hours, what is the impact on the statistics in (a) above?

Exercise 5.6

Create a new model in which jobs are created with an interarrival time that is exponentially distributed with a mean of 10 minutes. Jobs first move into an infinite-capacity queue where they await processing by a single-capacity machine. The machine has its own processing queue, and processes jobs for a time that is exponentially distributed with a mean of 8 minutes.

Run the model for 1000 days.

You are going to determine how modifying the failure and repair rates of the machine affects the number of jobs that are awaiting processing. Run each scenario below and record the maximum and average number of jobs in the waiting queue.

The failure and repair times of the machine are exponentially distributed, as shown in the table below:

MTBF	MTTR	Average loads in waiting queue	Maximum loads in waiting queue
2 hours	12 minutes		
1 hour	6 minutes		
30 minutes	3 minutes		
15 minutes	1.5 minutes		
7.5 minutes	0.75 minutes		

Exercise 5.7

Create a new model to simulate the following system:

Loads are created with an interarrival time that is uniformly distributed from 10 to 30 minutes. Loads wait in an infinite-capacity queue to be processed by one of three single-capacity, arrayed machines. Each machine has its own queue where loads are processed. Waiting loads move into one of the three queues in round-robin order. Each machine has a normally distributed processing time with a mean of 48 minutes and a standard deviation of 5 minutes. Each machine can only process one load at a time.

The machines were purchased at the same time. The mean time between failures is exponentially distributed with a mean of 170 minutes. The repair time is also exponentially distributed with a mean of 10 minutes.

The machines must also be cleaned. The time between cleanings is a constant 90 minutes and the cleaning time is a constant value of 10 minutes.

Define all failure and cleaning times using resource cycles.

Run the simulation for 100 days.

What was the average and maximum number of loads in the waiting queue?

Exercise 5.8

Create a new model in which loads are generated with an interarrival time that is exponentially distributed with a mean of 7 minutes. Loads first move into an infinite-capacity queue where they wait to be processed by one of two intake workers. As soon as there is room in a worker's queue, the next waiting load moves into a single-capacity processing queue. Each intake worker processes a load for a time that is exponentially distributed with a mean of 10 minutes. Each worker can only process one load at a time.

After intake processing, the loads move into an infinite-capacity queue to await their next process. The loads are then sent alternately to one of two single-capacity arrayed machines. Each arrayed machine has its own queue in which loads are processed. The machines require a processing time that is exponentially distributed with a mean of 7 minutes per load.

Before the load leaves the machine, it must be cut by a single-capacity tool for a time that is exponentially distributed with a mean of three minutes. There is only one tool that is shared by the two machines. Once cut, the load leaves the system. Every two hours, the tool has to be sharpened for 10 minutes.

Place all graphics and then complete the following:

a) Prepare two time lines that graph the current contents of each waiting queue. Graph the data for the first 5 days of simulated time.

Run the model for 100 days and then answer the following questions:

b) Did the system operate smoothly (were there any excessive lines)?
c) How many loads, on average, were in the initial queue and the holding queue throughout the simulation?

Exercise 5.9

Create a new model to simulate the following system:

Note

The solution for this assignment is required to complete exercise 7.1 (see "Exercise 7.1" on page 7.34 of the "Advanced Process System Features" chapter); be sure to save a copy of your model.

Loads are created with an interarrival time that is exponentially distributed with a mean of 20 minutes. Loads wait in an infinite-capacity queue to be processed by one of three single-capacity, arrayed machines. Each machine has its own single-capacity queue where loads are processed. Waiting loads move into one of the three queues in round-robin order. Each machine has a normally distributed processing time with a mean of 48 minutes and a standard deviation of 5 minutes.

The three machines were purchased at different times and have different failure rates. The failure and repair times are exponentially distributed with means as shown in the table below:

Machine	Mean time between failures	Mean time to repair
A	110 minutes	5 minutes
B	170 minutes	10 minutes
C	230 minutes	10 minutes

The machines also must be cleaned according to the following schedule. All times are constant:

Machine	Time between cleanings	Time to clean
A	90 minutes	5 minutes
B	90 minutes	5 minutes
C	90 minutes	10 minutes

Place the graphics for the queues and the resources.

Run the simulation for 100 days.

Define all failure and cleaning times using logic (rather than resource cycles). Answer the following questions:

a) What was the average number of loads in the waiting queue?

b) What were the current and average number of loads in Space? How do you explain these values?

Exercise 5.10

Create a new model to simulate the following system:

Widgets are created with a constant interarrival time of 2.5 minutes. The widgets first move into an infinite-capacity waiting queue. Making widgets requires three main processes: cutting, welding, and burnishing. Each of these processes has one machine that can process one load at a time. Each machine has its own queue where loads are processed.

The cutting process has three steps, all of which have times that are triangularly distributed, as defined in the table below:

Step	Minimum	Most-likely	Maximum
1	20 seconds	40 seconds	65 seconds
2	30 seconds	45 seconds	70 seconds
3	15 seconds	45 seconds	60 seconds

The welding process has a waiting queue before it that is limited to six widgets. Loads cannot leave the cutting process until there is room in the welding queue. The welding process requires two steps. The processing time of each step is exponentially distributed, as shown in the table below:

Step	Mean
1	90 seconds
2	40 seconds

The burnishing machine requires a processing time that is uniformly distributed between 20 seconds and 2 minutes 20 seconds. The waiting queue before burnishing is limited to 4 widgets. Loads cannot leave the welding process until there is room in the burnishing queue.

The burnishing machine requires an operator to load and unload the widget. The loading and unloading times are each a constant 20 seconds.

There are breaks and down times for the operator during each eight-hour shift. (Use one operator to work all shifts, seven days a week.) At the end of every eight-hour shift, the operator needs to sweep the floor, and is therefore unavailable for processing for the last 10 minutes of the shift. Every hour during the shift, the operator has two 5 minute breaks: one at the beginning of each hour to do paperwork and one at 30 minutes into the hour.

Simulate this system for 100 days.

a) There are currently six queueing positions before welding and four before burnishing. Record the time in system by adding up the average time that loads spent in each of the three processes.

Now vary the 10 queuing positions between welding and burnishing, trying all the configurations from 8 before welding and 2 before burnishing to 2 before welding and 8 before burnishing (use all 10 queues in each configuration). For each scenario, record the cycle time (average time in system) and determine which configuration of queuing positions produces the shortest cycle time.

Tip ☞ Use a spreadsheet to track the data. Later in this textbook, you will learn how to use the AutoStat software, which can automatically test each queue configuration and report changes in cycle time.

b) Using the base configuration (six queuing positions before welding and four before burnishing), what is the impact on the average number of loads in the initial waiting queue if the operator takes a single 10-minute break at the beginning of each hour instead of two separate 5-minute breaks?

Chapter 6

Introduction to Conveyors

Chapter 6

Introduction to Conveyors

Until now, loads have “beamed” between queues in models. Beginning in this chapter, you will learn how to model different types of movement systems to transport loads from location to location. This chapter discusses how to create a basic conveyor system.

You will learn that conveyors, like queues, are territories in which loads can reside physically in a model. See “Territories and space” on page 3.10 of the “AutoMod Concepts” chapter for a discussion of territories.

The concepts and tools needed to model conveyors are transferable to many of the other types of material handling systems available in the AutoMod software, such as path mover systems (discussed in chapter 11, “Introduction to Path Mover Systems,”) power & free systems, and forklift trucks.

Conveyor systems

In the AutoMod software, models can contain one or more systems. You must define a process system in every model, and you can define one or more movement systems as needed. A **conveyor** system has three required components:

sections Sections are the individual segments of a conveyor on which loads travel. Sections are one-directional and can differ in length, width, and speed.

transfers Transfers are connections between two conveyor sections. For loads to move from one section to another, the two sections must be connected by a transfer. Transfers are automatically created as you draw sections in the conveyor system. Transfers determine the speed at which loads move from one section to another, as well as the orientation of loads on a section. (You will learn more about load orientation in chapter 9, "Modeling Complex Conveyor Systems.")

stations Stations are locations at which loads get on or off a section, and where loads can stop to process on a section. Stations can be located anywhere on a section.

Measuring distances in the Work Area window

Before you begin drawing a conveyor, you must first learn how to measure distances in the Work Area window. Measuring distances is necessary to draw a model's movement systems to scale; otherwise, the model will be inaccurate. There are two tools in the AutoMod software to help you draw movement systems to scale:

- The drawing grid
- The Measurement window

Using the drawing grid

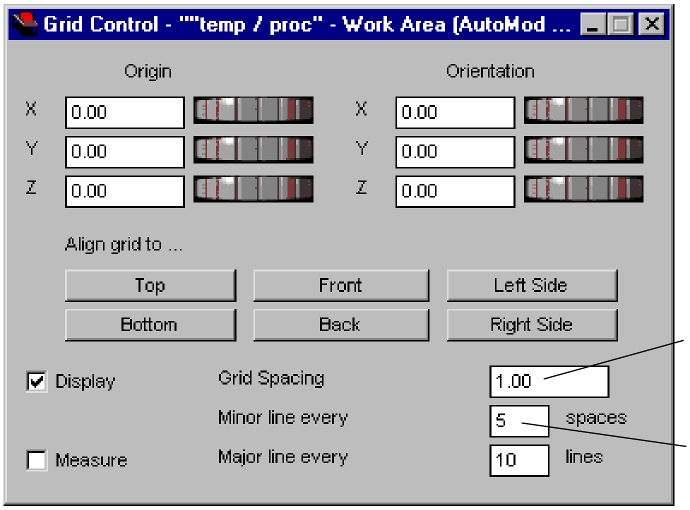
The drawing grid is similar to a piece of graph paper that allows you to easily draw and align graphics in a model. All graphics are drawn or placed on the grid. Each time you open a model, the size of the drawing grid is automatically resized to encompass the graphics in the model.

By default, lines in the drawing grid are spaced five feet apart. You can increase or decrease the spacing of the drawing grid.

To change the grid spacing, do the following:

Step 1 ***Open*** the AutoMod software and ***create*** a temporary model. The Work Area window opens.

Step 2 ***Click*** Grid Control. The Grid Control window opens, as shown below:



Grid Control - ""temp / proc" - Work Area (AutoMod ...

Origin

X 0.00

Y 0.00

Z 0.00

Orientation

X 0.00

Y 0.00

Z 0.00

Align grid to ...

☑ Display

□ Measure

Grid Spacing 1.00

Minor line every 5 spaces

Major line every 10 lines

The base distance of measurement (in model units)

A multiplier that determines the distance between minor lines

Grid Control window

The controls for changing the grid's origin, orientation, and alignment are not discussed in this textbook.

Options for changing the grid spacing are defined as follows:

Display Select and clear the Display check box to toggle the grid display on and off.

Measure The Measure option opens the Measurement window; this window is discussed in the next section.

Grid Spacing The value on which you want grid spacing to be based. This value, along with the "Minor line every" multiplier, determines the spacing of the grid. The Grid Spacing value is in model distance units, which by default are feet. (The default distance unit can be changed, but doing so is not discussed in this textbook.)

Minor line every A multiplier that, in conjunction with the Grid Spacing value, determines the distance between each grid line. The default Grid Spacing is 1.00 (foot) with a minor line every 5 (feet). Therefore, the distance between minor lines is 1.00 x 5 = 5 feet. You can change the distance that each pair of minor lines represents by editing either the Grid Spacing value or the "Minor line every" value (or both).

Major line every A value that determines the frequency of the thicker black grid lines. By default, every tenth line in the grid is a major line, while the nine intervening lines are minor lines.

Step 3 To increase the space between grid lines, ***type*** "2.00" in the Grid Spacing text box and ***press*** Enter. The distance between minor lines changes to 10 feet (2.00 x 5).

Step 4 By default, there is a major line every 10 lines. In the Major line text box, ***type*** "15" and ***press*** Enter. There are fewer heavier lines in the drawing grid.

Step 5 ***Reset*** the drawing grid to the default values (Grid Spacing is 1.00 with a major line every 10 lines).

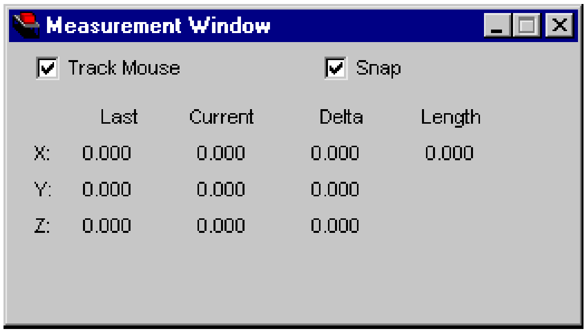
You are now ready to learn how to use the Measurement window to measure distances in the Work Area window.

Using the Measurement window

The Measurement window can be used to measure distances by tracking the movement of the mouse when drawing or placing graphics in the Work Area window. To open the Measurement window:

Step 1 ***Click*** Measurement. The Measurement window opens.

Step 2 ***Move*** the Measurement window below the Work Area window, then ***select*** Track Mouse.



Measurement Window

☑ Track Mouse ☑ Snap

	Last	Current	Delta	Length
X:	0.000	0.000	0.000	0.000
Y:	0.000	0.000	0.000	
Z:	0.000	0.000	0.000	

Measurement window

Options in the Measurement window are defined as follows:

Track Mouse The Track Mouse option allows you to measure distances by clicking (to set the starting point of a measurement) and moving the mouse to measure a distance in any direction.

Snap The Snap option causes the graphics you are placing to move in increments determined by the grid spacing (by default, graphics move in 1-foot increments). Clear Snap before drawing or placing objects that require positioning between grid spacing intervals.

When tracking the mouse, the measurement values in the window can be interpreted as follows:

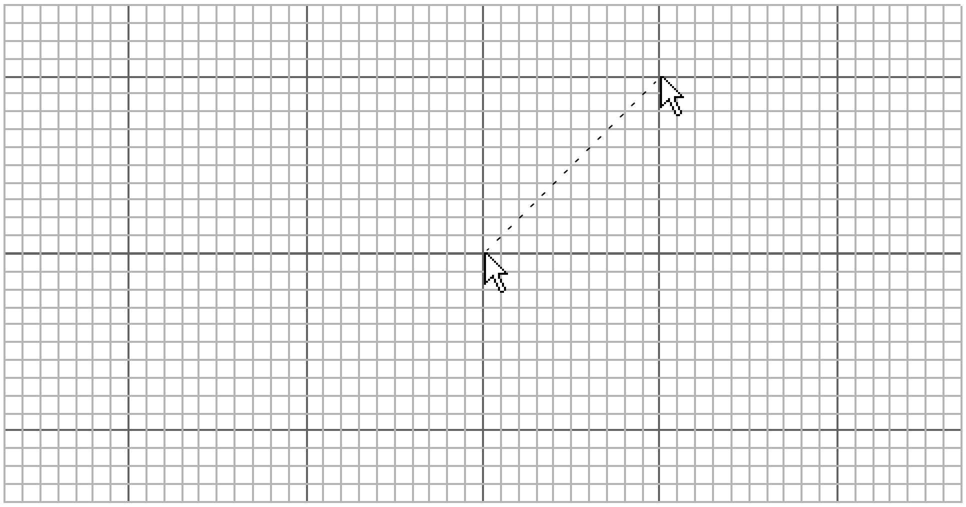
Last The Last column displays the X, Y, and Z coordinates of the most recent mouse click.

Current The Current column displays the X, Y, and Z coordinates of the arrow's current location.

Delta The Delta column displays the difference between the last coordinates and the current coordinates on the X, Y, and Z axes.

Length Displays the distance between the last coordinates and the current coordinates.

Suppose you wanted to know the diagonal distance when you move from the center of the drawing grid to the intersection of the next major lines in the positive X and Y directions, as shown below:



Measuring distances in the Work Area window

To determine the distance between two points:

Step 3 ***Watch*** the Current X, Y, and Z values in the Measurement window as you ***move*** the arrow to the point (0, 0, 0) in the center of the drawing grid. ***Click*** the left mouse button. Every value in the Measurement Window reads 0.000.

Step 4 ***Move*** the arrow to the intersection of the two major lines in the positive X and Y directions (the Current values are 50, 50, and 0). Notice the Length value is 70.711 feet.

You can verify that this distance is correct using the Pythagorean theorem:

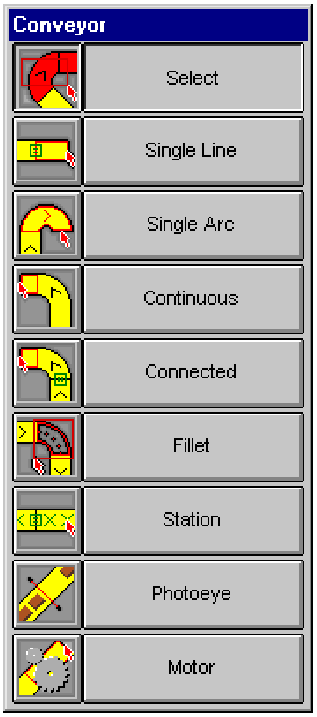
$$A^2 + B^2 = C^2$$

With A = 50 and B = 50, we find $\sqrt{5000} = 70.711$.

Now you are ready to learn about the tools available to draw conveyor systems.

Conveyor drawing tools

The tools for drawing and placing conveyor entities are located on a palette in the conveyor system, as shown below:



Conveyor palette

The tools are defined as follows:

Select The Select tool selects one or more entities in the conveyor system.

Single Line The Single Line tool draws a straight conveyor section.

Single Arc The Single Arc tool draws a curved conveyor section.

Continuous The Continuous tool draws a single section that consists of multiple straight or curved segments. All segments are part of one section; that is, *no transfers are created*. By default, the Continuous tool alternates between straight and curved segments.

Connected The Connected tool draws multiple sections connected by transfers. Unlike the Continuous tool, the Connected tool draws only straight sections by default.

Fillet The Fillet tool draws a curved section to automatically connect two non-parallel sections.

Station The Station tool places a station on a section.

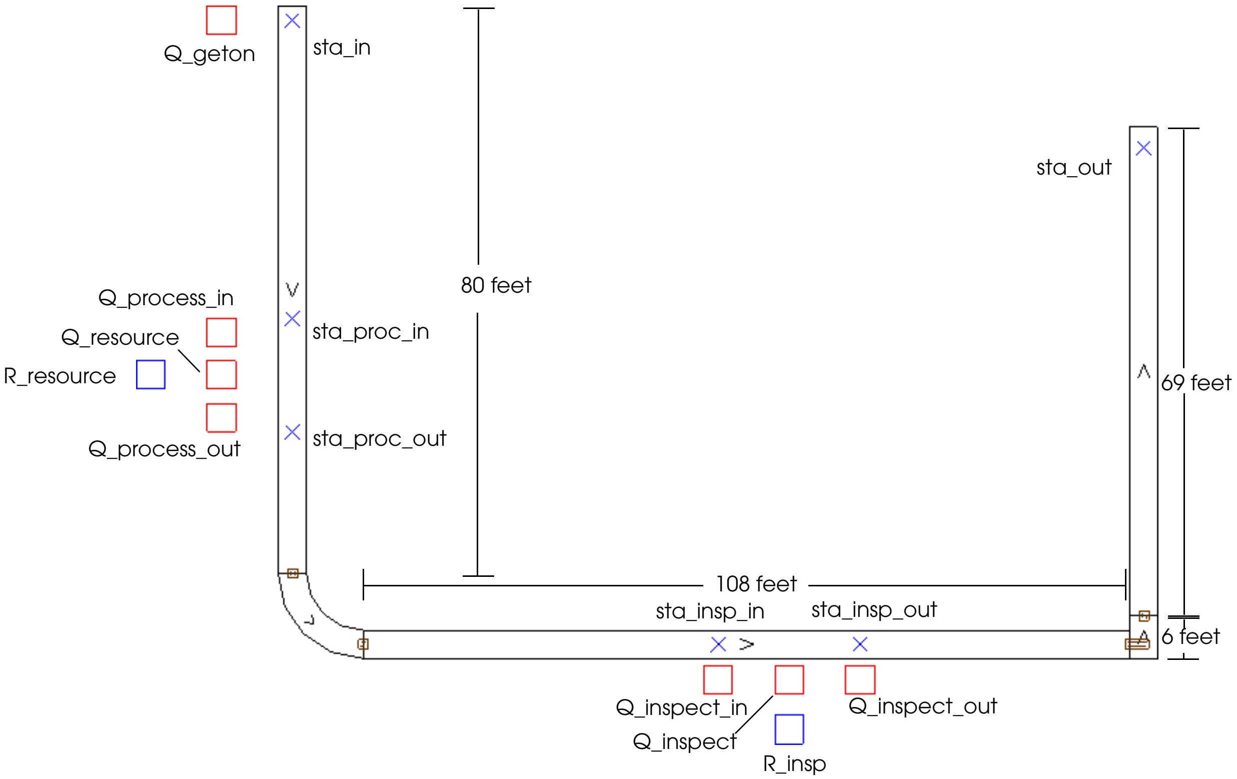
Photoeye The Photoeye tool draws an infrared beam across a conveyor to regulate load movement (photoeyes are not discussed in this textbook).

Motor The Motor tool defines a device that drives both conveyor sections and transfers; motors can be taken down to stop conveyor movement.

This chapter discusses the Select, Single Line, Fillet, and Station tools. Motors are discussed in chapter 9, “Modeling Complex Conveyor Systems.”

Example 6.1: Drawing a conveyor system

Consider the layout of the conveyor system shown below:



Layout of example model 6.1

Loads are created with an interarrival time that is exponentially distributed with a mean of 11 minutes. Loads move into the infinite-capacity queue “Q_geton” before getting on the conveyor at station “sta_in.”

Loads first travel to station “sta_proc_in,” where they get off the conveyor and move into the infinite-capacity waiting queue “Q_process_in.” Loads wait to be processed by a resource that can process two loads at the same time; the resource has its own queue for processing. Each load is processed for a time that is exponentially distributed with a mean of 20 minutes. When processing is complete, loads move into the infinite-capacity queue “Q_process_out” before getting back on the conveyor at station “sta_proc_out.”

Loads then travel on the conveyor to station “sta_insp_in,” where they again get off the conveyor by moving into the infinite-capacity waiting queue “Q_inspect_in.” Loads wait to be inspected by an inspector, who inspects each load for a time that is exponentially distributed with a mean of 20 minutes. The inspector can inspect only one load at a time. After inspection, loads move into the infinite-capacity queue “Q_inspect_out” before getting back on the conveyor at station “sta_insp_out.”

Loads then travel to station “sta_out,” which is located on a ramped conveyor that descends 20 feet in 69 feet. After arriving at station “sta_out,” loads are removed from the system.

Loads are blue and their dimensions are two feet by three feet by two feet.

You will simulate the system for one day.

Creating example model 6.1

To create the example model:

Step 1 ***Create*** a new model named “examp61.”

Step 2 ***Open*** the View Control and ***select*** Child Windows on Top. This keeps the Measurement and other windows from moving behind the Work Area window.

Step 3 ***Close*** the View Control window.

You are now ready to create the conveyor system.

Creating the conveyor system

To create the conveyor system:

Step 1 From the System menu in the Work Area window, ***select*** New. The Create A New System window opens.

Step 2 In the System Name text box, ***type*** “conv” and in the System Type drop-down list, ***select*** Conveyor.

Step 3 ***Click*** Create to create the conveyor system. The Conveyor palette appears.

Drawing conveyor sections

You will begin drawing the system starting with the 80-foot vertical conveyor section on the left (see “Layout of example model 6.1” on page 6.9).

To draw the first section:

Step 1 On the Conveyor palette**,** ***click*** Single Line. The Single Line window opens. The default section name is “sec1.”

Note

The section number increments every time you add another section (that is, sec2, sec3, and so on). If you want to edit the name of the section, type a new name in the Name text box and press Enter *before* you place the section. For this textbook, you can use the default section names.

Step 2 ***Select*** the Orthogonal check box to restrict the section you are drawing to either a horizontal or vertical section. The orthogonal option forces sections to be drawn parallel to grid lines.

Drawing to scale

It is important to draw the conveyor section using the correct scale. You can use both the grid and the Measurement window to help you draw an 80-foot section. By default, the grid lines are 5 feet apart, with major lines every 50 feet.

Step 1 ***Click*** Measurement to open the Measurement window.

Step 2 ***Move*** the Measurement window below the Work Area window and ***select*** Track Mouse.

Step 3 In the Work Area window, ***click*** once in the upper-left corner of the grid to place the starting point of the conveyor where two major lines intersect (refer to the illustration below).

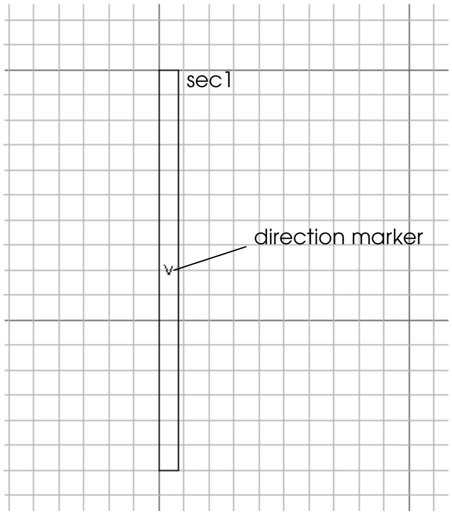
Tip: If the first point of the section is not correctly aligned with the drawing grid, you can press Esc to quit drawing the section and place the first point again.

Step 4 ***Move*** the arrow down the screen. ***Watch*** the Measurement window's Length value to determine how long the conveyor section is. When the Length value is close to 80, use the grid to position the end of the section at the 80-foot mark and ***click*** the mouse again.

Note: The Measurement window tracks the mouse's movement, not the section length; so, unless you drag the mouse in exactly a straight line, the Length is not exactly 80. That is why you should also use the grid lines as a guide for determining section length.

Tip: If you want to delete a section that you have drawn, click Select on the Conveyor palette, then click the conveyor section to select it (the section color changes to green). Select Delete from the Edit menu in the Work Area window.

You have now drawn the first section of conveyor in the model, as shown below:

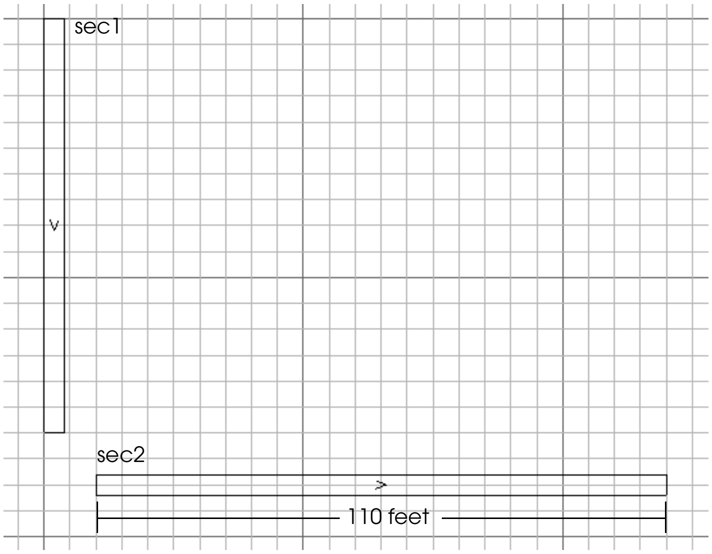


Drawing the vertical conveyor section

The direction marker in the center of the section indicates the direction that loads travel on the section. The direction of travel is determined when you draw the section (that is, from the first mouse click to the second mouse click). In this case, loads travel from the top of the section to the bottom.

Step 5 ***Draw*** the horizontal section of conveyor, as shown in the illustration below.

Tip ☞ Use the drawing grid and Measurement window to draw the section to scale. Place the section's left point first, so that the section's direction is from left to right.



Drawing the horizontal conveyor section

Filleting two paths

The Fillet tool is used to automatically connect two sections of path with an arc of a given radius.

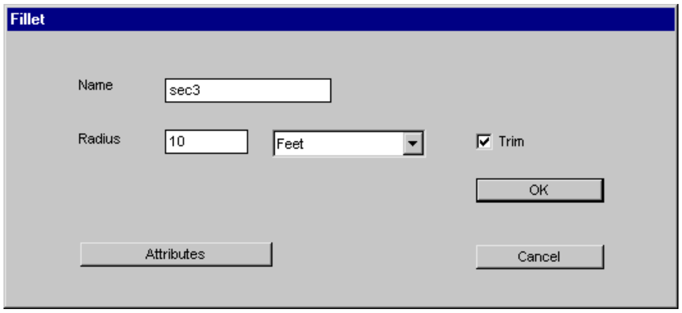
To create an arc between the vertical and horizontal paths, do the following:

Step 1 On the Conveyor palette, ***click*** Fillet.

Step 2 ***Select*** the two sections you want to connect in the Work Area window.

Tip: You can select multiple sections at the same time by dragging a box in the Work Area window over parts of each section. You do not need to select the entire section.

In the Work Area window, a red outline of an arc appears that connects the two sections. The Fillet window also opens, as shown below:


Fillet

Name: sec3

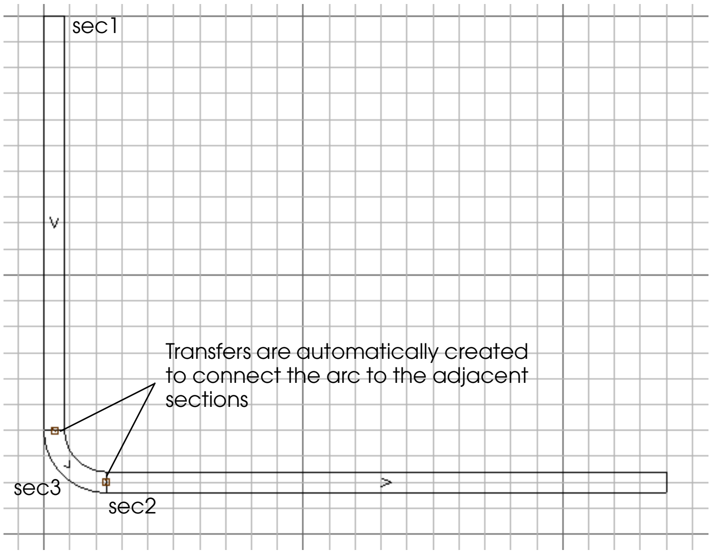
Radius: 10 Feet

☑ Trim

Fillet window

The Trim option in the Fillet window automatically adjusts the lengths of the conveyor sections so that they connect to the arc. In this case, the 110-foot horizontal section is trimmed to 108 feet. It is possible to prevent sections from being resized by clearing the Trim check box, but in the examples and exercises in this textbook, always trim conveyor sections to connect to arcs.

Step 3 To use the default arc (with a radius of 10 feet), ***click*** OK to fillet the paths. The arc section is drawn and transfers are automatically created to connect the three sections of conveyor, as shown below:


sec1
Transfers are automatically created to connect the arc to the adjacent sections
sec3
sec2

Filleting sections creates an arc section and transfers

Connecting sections using Snap to End

The next section of conveyor is a small six-foot vertical section that connects to the end of the horizontal section. To draw the small section to scale, adjust the grid.

To adjust the grid to draw the section:

Step 1 ***Click*** Grid Control to open the Grid Control window.

Step 2 ***Change*** the Minor Line value to 2 spaces (a minor line every two feet will make it easy to draw a section that is exactly six feet) and ***press*** Enter.

Step 3 ***Close*** the Grid Control window and ***zoom in*** on the end of the horizontal section in the Work Area window.

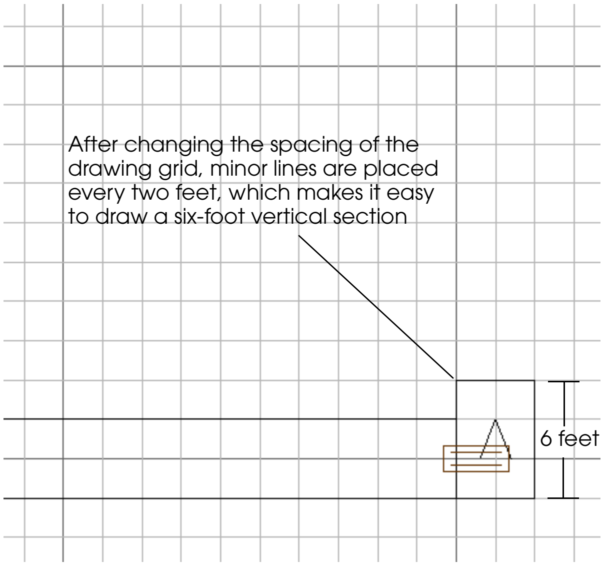
Step 4 On the Conveyor palette, ***click*** Single Line. The Single Line window opens.

The Snap to End option makes it easy to draw two sections that are connected end-to-end. You can also use the Snap to Section option to draw a new conveyor section that connects to the side of an existing conveyor section.

Step 5 You want the new section to start at the end of the horizontal section, so ***click*** Snap to End.

Step 6 In the Work Area window, ***click*** the end of the horizontal section and ***drag*** to draw a six-foot vertical section, as shown below. Use the grid lines to ensure that you draw a section that is six feet long.

Step 7 ***Click*** where you want the section to end.


After changing the spacing of the drawing grid, minor lines are placed every two feet, which makes it easy to draw a six-foot vertical section
6 feet

Drawing a section at the end of an existing section

The next section of conveyor is the 69-foot vertical section. To create the section, you will copy the vertical section that is on the left side of the model.

Copying conveyor sections

Copying a section is a quick way to create sections. To copy the 80-foot vertical section, do the following:

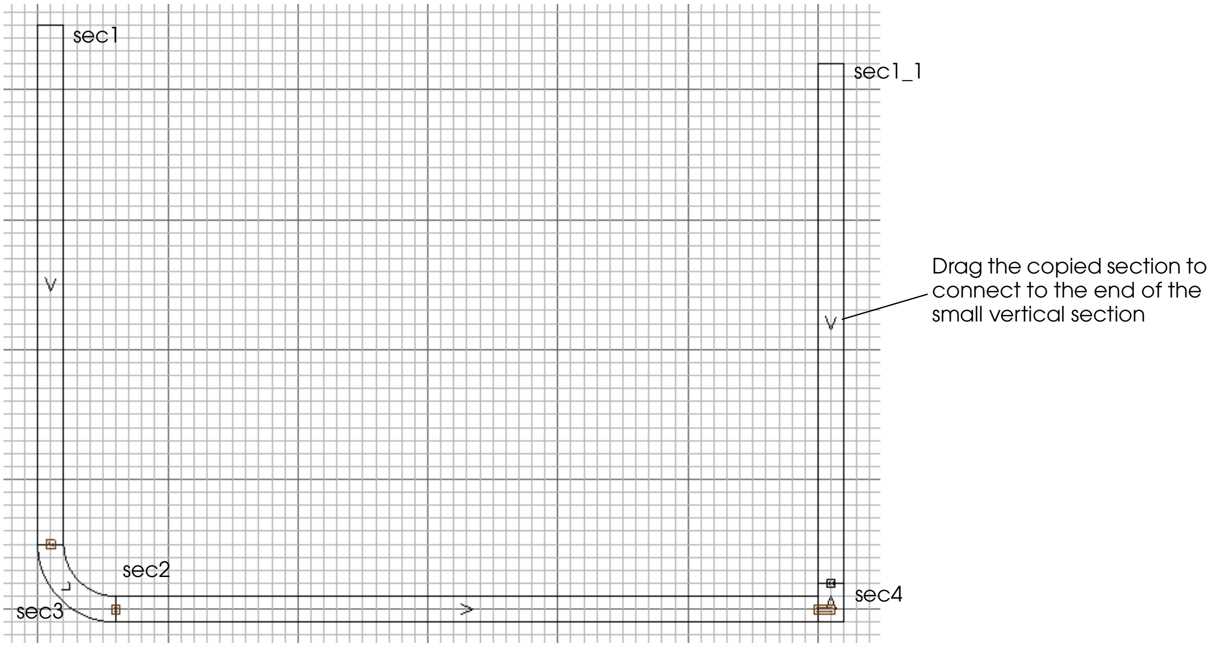
Step 1 On the Conveyor palette, ***click*** Select.

Step 2 ***Select*** the vertical conveyor section. The section turns green to indicate that it is selected.

Step 3 From the Edit menu, ***select*** Copy. A copy of the section is created on top of the original section, and the Copy window opens.

Note The default name of the copied section is “sec1_1.” The last number in the name increments every time you make another copy of the original section (that is, sec1_2, sec1_3, etc.).

Step 4 To move the copied section, ***drag*** it to the correct location, as shown below:



Drag the copied section to connect to the end of the small vertical section

Moving the copied section

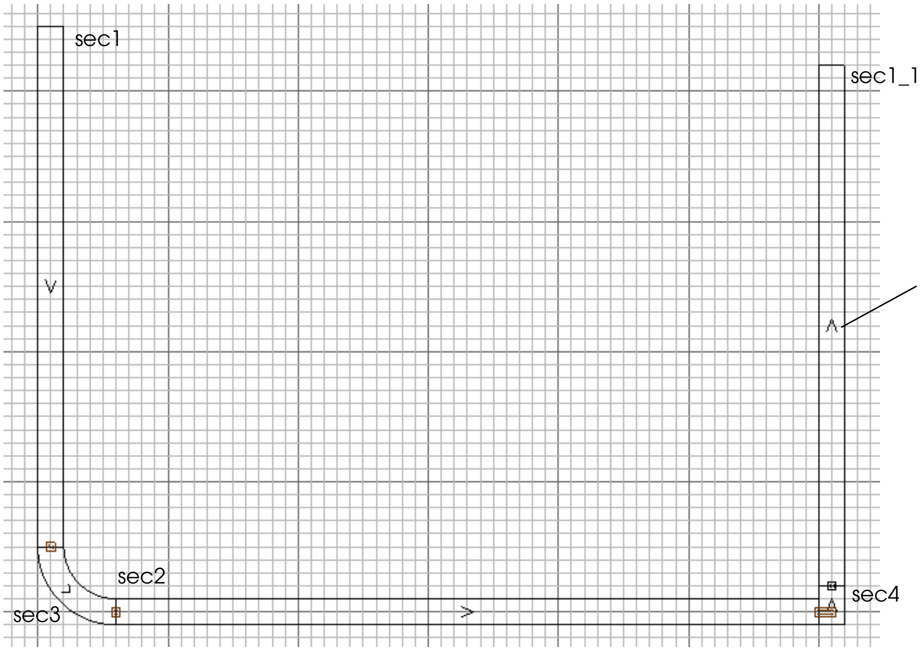
Step 5 ***Click*** OK to close the Copy window.

The copied section is now in the correct location. However, the direction of the section is incorrect. (We want loads to travel from the bottom of the section to the top, not from the top to the bottom, as is currently indicated.)

Changing section direction

To change the direction of the section, do the following:

Step 1 The section is already selected, so from the Edit menu, ***select*** Change Direction. The direction marker reverses, as shown below:



The direction marker now indicates that loads travel from the bottom of the conveyor to the top

Changing section direction

The next step is to edit the copied section's length to be 69 feet and make it ramped as described in "Example 6.1: Drawing a conveyor system" on page 6.9.

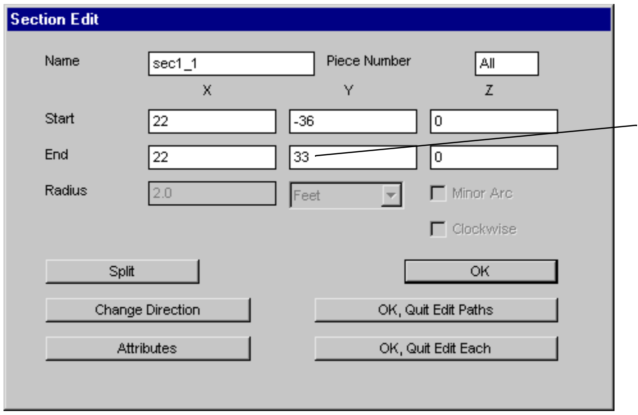
Editing section length

When drawing, you can use the grid spacing and Measurement window to determine a section's length. You can also edit a section's length once it is drawn by adjusting the starting or ending values of the section. For the copied section, you must edit its length from 80 to 69 feet.

Step 1 The section is already selected, so from the Edit menu, ***select*** Edit. The Section Edit window opens.

The section currently starts at 36 feet in the negative Y direction and ends at 44 feet in the positive Y direction for a total length of 80 feet. The section needs to be 69 feet long, which is 11 feet shorter than its current length. The start of the section must remain connected to the other sections, so adjust the end of the section.

Step 2 ***Change*** the Y End value from “44” to “33” and ***press*** Enter, as shown below:



Section Edit

Name: sec1_1 Piece Number: All

	X	Y	Z
Start	22	-36	0
End	22	33	0
Radius	2.0	Feet	

□ Minor Arc
□ Clockwise

Change the Y End value to 33 (36 + 33 = 69)

Changing a section's length

When you press Enter, the length of the section is updated in the Work Area window.

Moving sections

If you need to move a section that you have already drawn, use the following procedure:

Step 1 On the Conveyor palette, ***click*** Select.

Step 2 In the Work Area window, ***select*** the section you want to move (the color of the section changes to green).

Step 3 From the Edit menu, ***select*** Move.

Step 4 ***Drag*** the section to the desired location.

Step 5 ***Click*** OK.

Now that the section is the correct length, you need to adjust its slope to create a ramped section.

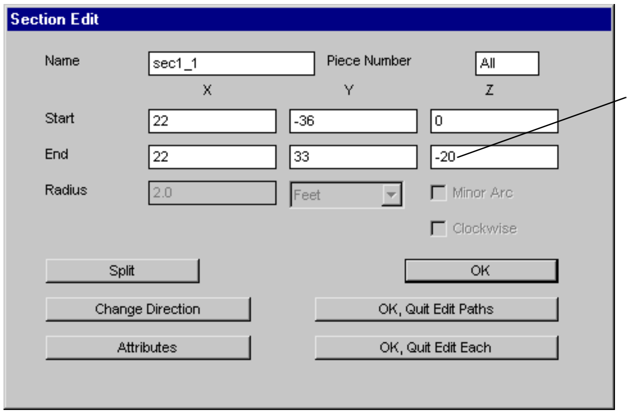
Creating ramped sections

Ramped sections are often used to model gravity feed conveyors that start on a mezzanine and go down to the main floor.

To make a ramped section, do the following:

Note The Section Edit window should still be open. If you closed the window, you can reopen it by selecting the copied section and selecting Edit from the Edit menu.

Step 1 The section’s Z axis values determine its slope. Currently, the conveyor is flat (the Z Start and Z End values are both zero). To make the conveyor descend 20 feet, ***change*** the Z End value to –20 and ***press*** Enter, as shown below:



Section Edit

Name: sec1_1 Piece Number: All

	X	Y	Z
Start	22	-36	0
End	22	33	-20
Radius	2.0	Feet	

☐ Minor Arc

☐ Clockwise

Changing the Z End value to –20 causes the section to descend 20 feet

Changing a section’s slope

When you press Enter, the slope of the section is updated in the Work Area window.

Step 2 ***Click*** OK to close the Section Edit window.

Step 3 When you select a section, any transfers for that section are also highlighted, so the Transfer Edit window opens. We do not want to edit the transfer, so ***select*** “OK, Quit Edit Each” to stop editing.

Tip To verify that the section is now ramped, use the View Control to rotate the view in the Work Area window. When you are finished, click Top to return to the top view.

Placing stations

Stations are locations at which a load can get on or off a section, or can stop to process while on a section. Stations can be located anywhere on a section. Your conveyor system must contain at least two stations in order for loads to travel on the conveyor: one station at which loads get on the conveyor, and one at which loads get off the conveyor. The system can have as many stations as the system design requires.

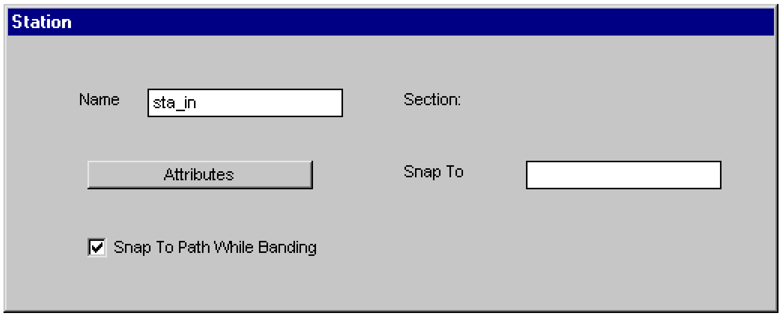
In this model, you need to place stations where loads get on and off the conveyor, as well as where work is performed (see “Example 6.1: Drawing a conveyor system” on page 6.9).

To place stations, do the following:

Step 1 On the Conveyor palette, ***click*** Station. The Station window opens.

Note The default station name is “sta1.” The station number increments every time you place another station (that is, sta2, sta3, etc.).

Step 2 ***Change*** the station name to “sta_in,” as shown below:



Station

Name: sta_in

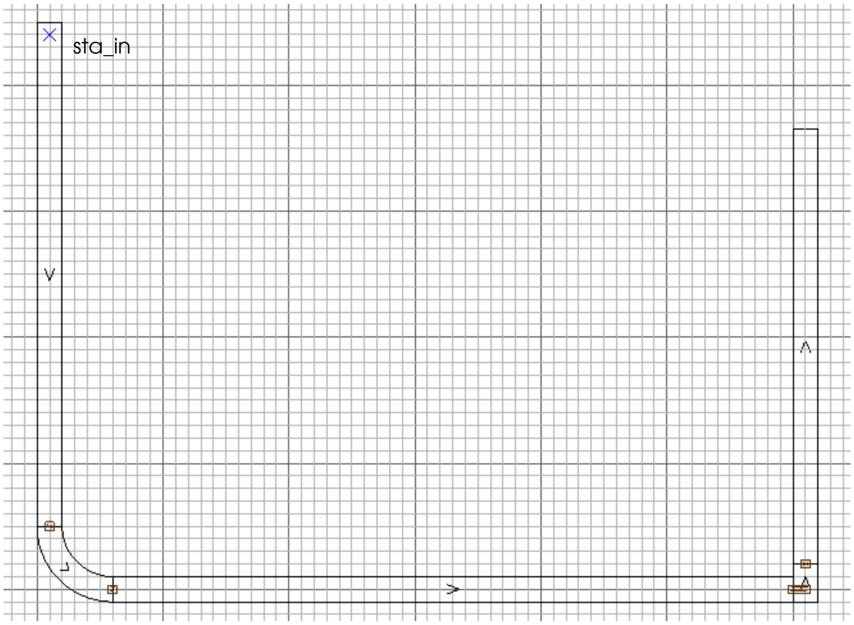
Section:

Snap To:

☑ Snap To Path While Banding

Station window

Step 3 ***Drag*** the station to the correct location at the beginning of the 80-foot vertical section, as shown below. (The graphic for the station appears when you click the mouse button and is placed when you release the mouse button).

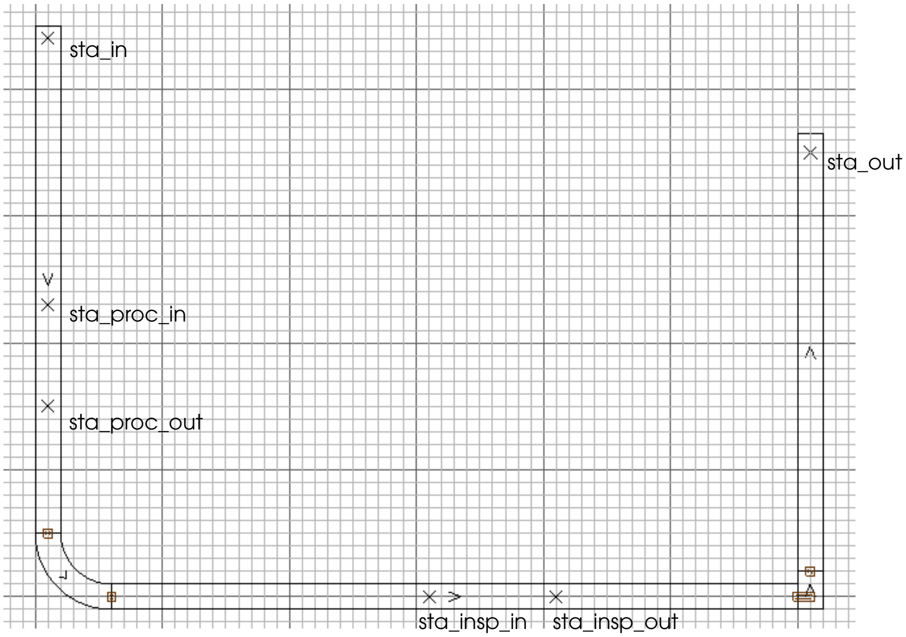


Adding station “sta_in”

You have now placed station “sta_in,” which is where loads will enter the system.

Step 4 ***Name*** and ***place*** the remaining stations as shown in the illustration below.

Tip ☞ To edit, delete, or move a station, select the station (using the Select tool) and then select the desired option from the Edit menu.



Placing the remaining stations

The conveyor system is drawn according to the initial layout. Now you need to write logic to move loads through the system.

Moving loads through the conveyor system

Loads get on and off the conveyor and travel through the system by executing the **move** and **travel** actions in an arriving procedure.

The `move` action causes loads to get on a conveyor by moving into a station. Loads can get on the conveyor from any other location in a simulation, such as a queue (including directly from Space). Similarly, to get off the conveyor, loads can move into any other location in a simulation, such as into a queue or another movement system, or the loads can be sent to die.

The `travel` action causes loads to travel between stations on the conveyor.

For example, the logic for moving loads through the conveyor system in example model 6.1 is shown below:

```
begin P_geton arriving
   move into Q_geton /*infinite capacity*/
   move into conv:sta_in
   travel to conv:sta_proc_in
   send to P_process
end
```

The first action that loads execute is the `move` action to move into the infinite-capacity waiting queue “Q_geton.” Loads then execute another `move` action to get on the conveyor at station “sta_in.”

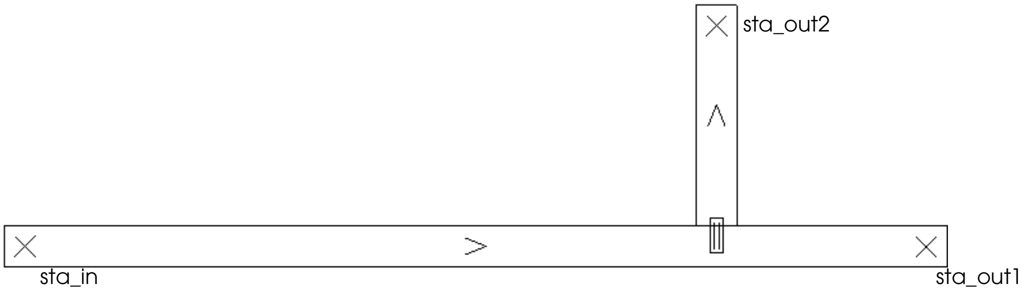
Note

When referring to movement system entities in logic, you must type the movement system name and entity name separated by a colon, such as `conv:sta_in,` where the system name is “conv” and the station name is “sta_in”.

Loads then execute the `travel` action to travel from station “sta_in” to station “sta_proc_in.” After arriving at station “sta_proc_in,” loads are sent to another process, P_process, and begin executing that process’ arriving procedure.

Alternately selecting stations

You can use the `nextof` distribution with the `move` or `travel` actions to alternately select the stations where loads get on or off the conveyor (for more information, see “Selecting entities alternately using the nextof distribution” on page 5.47). For example, consider the layout of the conveyor system shown below:



Selecting sequential load destinations

Loads get on the conveyor station at station “sta_in” and travel alternately to one of two exit stations (“sta_out1” or “sta_out2”). The logic for selecting a destination alternately is shown below:

```
begin P_start arriving procedure
   move into conv:sta_in
   travel to nextof(conv:sta_out1, conv:sta_out2)
   send to P_next
end
```

The `move` action causes loads to get on the conveyor at station “sta_in.” Loads then travel to one of the two exit stations, which are selected alternately using the `nextof` distribution.

Defining the example model logic

To move loads through the conveyor system in example model 6.1, do the following:

Step 1 ***Create*** a new process system named “proc.”

Tip

You can toggle between editing the process system and the conveyor system by selecting Open from the System menu in the Work Area window and opening the system you want to edit.

Step 2 ***Create*** a new source file named “logic.m”.

Step 3 ***Edit*** the source file and ***type*** the following logic:

```
begin P_geton arriving
   move into Q_geton                /*infinite capacity*/
   move into conv:sta_in
   travel to conv:sta_proc_in
   send to P_process
end

begin P_process arriving
   move into Q_process_in           /*infinite capacity*/
   move into Q_resource             /*capacity of 2*/
   use R_resource for e 20 min      /*capacity of 2*/
   move into Q_process_out          /*infinite capacity*/
   move into conv:sta_proc_out
   travel to conv:sta_insp_in
   send to P_inspect
end

begin P_inspect arriving
   move into Q_inspect_in           /*infinite capacity*/
   move into Q_inspect              /*capacity of 1*/
   use R_insp for e 8 min           /*capacity of 1*/
   move into Q_inspect_out          /*infinite capacity*/
   move into conv:sta_insp_out
   travel to conv:sta_out
   send to die
end
```

Take a moment to review the model logic. Refer to the example model description for an explanation of load activity in the simulation (see “Example 6.1: Drawing a conveyor system” on page 6.9).

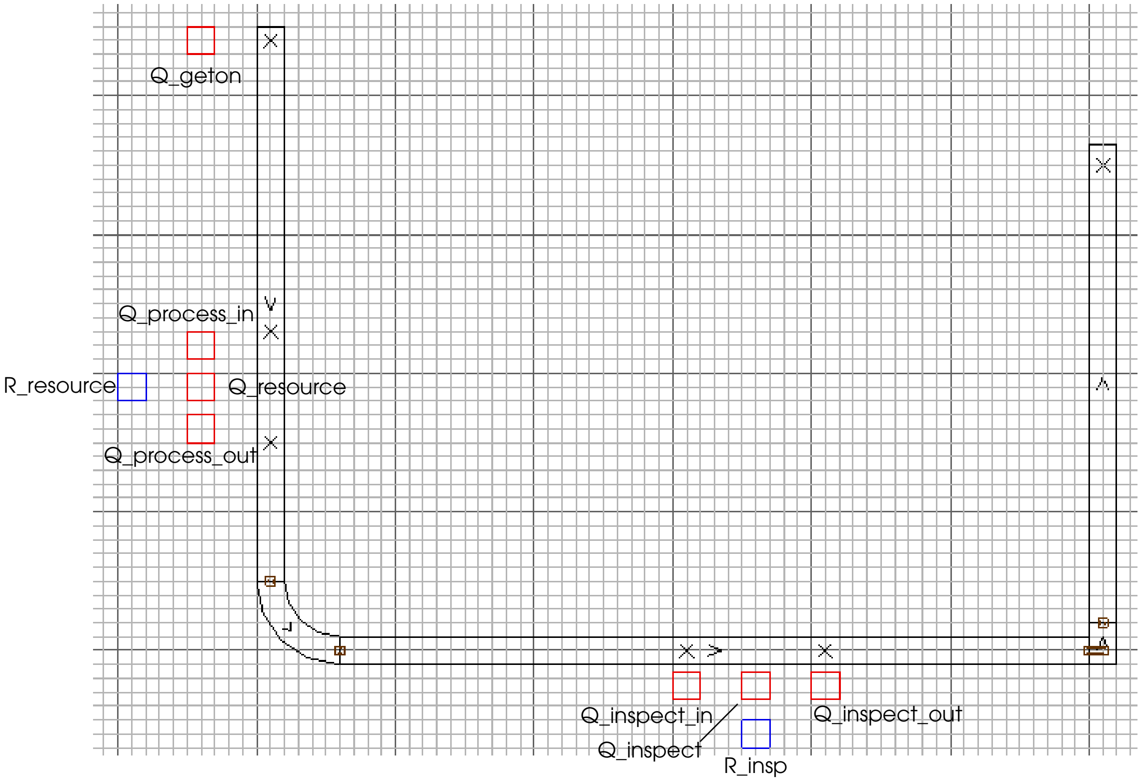
Step 4 ***Save*** and ***quit*** the source file. When prompted, define the following entities:

P_geton – A single process.
Q_geton – A single queue with infinite capacity.
P_process – A single process.
Q_process_in – A single queue with infinite capacity.
Q_resource – A single queue with a capacity of 2.
R_resource – A single resource with a capacity of 2.
Q_process_out – A single queue with infinite capacity.
P_inspect – A single process.
Q_inspect_in – A single queue with infinite capacity.
Q_inspect – A single queue with a capacity of 1.
R_insp – A single resource with a capacity of 1.
Q_inspect_out – A single queue with infinite capacity.

Placing queue and resource graphics

When modeling a system, place queue and resource graphics so that you can watch the animation and verify that the system is being simulated correctly.

Step 1 ***Place*** the queue and resource graphics, as shown below:



Placing queue and resource graphics

Placing load graphics

Until now, you have always used the default graphic for loads, because load size did not affect the accuracy of the model. Now that loads are traveling on a conveyor, load size is important for accurate simulation, because the size of a load affects the amount of space it requires on the conveyor.

In example model 6.1, you need to scale loads to 2 feet on the X axis, 3 feet on the Y axis, and 2 feet on the Z axis. The loads should also be blue.

To define and place load graphics, do the following:

Step 1 ***Click*** Loads on the Process System palette. The Loads window opens.

Step 2 ***Define*** a new load type named “L_job”.

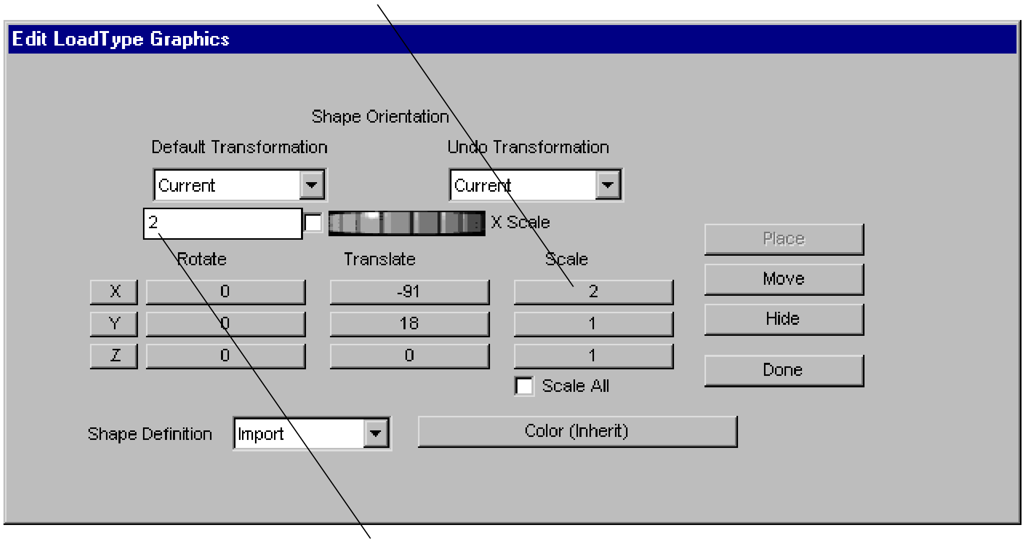
Step 3 ***Define*** a creation specification that sends loads to process “P_geton” with an interarrival time that is exponentially distributed with a mean of 11 minutes.

Step 4 In the Loads window, ***select*** L_job in the Load Types list and ***click*** Edit Graphic. The Edit Load Type Graphics window opens.

Step 5 ***Click*** Place. In the Work Area window, ***click*** above the vertical section where loads get on the conveyor; a small green box appears, representing a load of type L_job.

Step 6 To scale the load in the X direction, ***click*** the X Scale button then ***type*** “2” in the Scale text box and ***press*** Enter, as shown below:

Click the X Scale button



2 □ X Scale

	Rotate	Translate	Scale
X	0	-91	2
Y	0	18	1
Z	0	0	1

□ Scale All

Type “2” in the Scale text box and press Enter

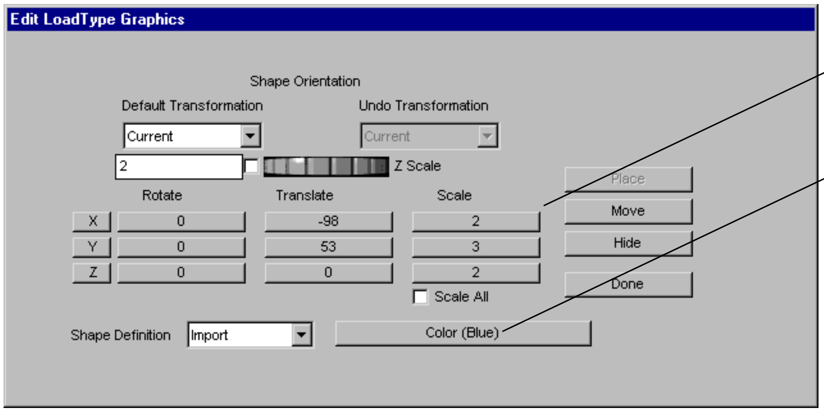
Scaling load graphics

Step 7 To scale the load in the Y direction, ***click*** the Y Scale button, ***type*** “3” in the Scale text box, and ***press*** Enter.

Step 8 To scale the load in the Z direction, ***click*** the Z Scale button, ***type*** “2” in the Scale text box, and ***press*** Enter.

Step 9 ***Click*** Color (Inherit). The Select a Color window opens.

Step 10 ***Click*** Blue and ***click*** OK. The Edit Load Type Graphics window opens, as shown below:



Default Transformation: Current

Undo Transformation: Current

□ Z Scale: 2

	Rotate	Translate	Scale
X	0	-98	2
Y	0	53	3
Z	0	0	2

□ Scale All

Shape Definition: Import

The load size is 2 x 3 x 2

The load color is set to blue

Defining load graphics for loads of type L_job

Step 11 ***Click*** Done to close the Edit Load Type Graphics window.

Running the model

To run the model, do the following:

Step 1 ***Define*** the run control to run the model for one day.

Step 2 ***Export*** and ***run*** the model.

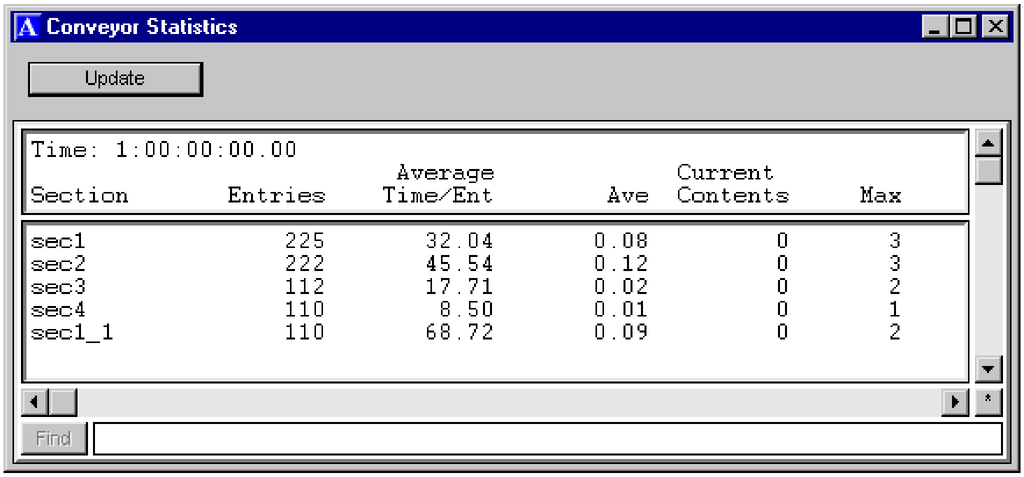
Watch the simulation and refer to the example description for an explanation of load activity in the simulation (see “Example 6.1: Drawing a conveyor system” on page 6.9). When you are familiar with how loads are processed in the system, press “g” to turn off graphics and run the simulation to completion so you can verify the conveyor system statistics, discussed next.

Displaying section statistics

Section statistics provide information about loads that traveled on a conveyor section. These statistics should be used in conjunction with process system statistics when analyzing a conveyor system simulation.

To display section statistics:

Step 1 From the Conveyor menu, ***select*** Section Statistics. The Section Statistics window opens, as shown below:



```
Time: 1:00:00:00.00
                           Average                  Current
Section         Entries    Time/Ent        Ave     Contents      Max

sec1                225       32.04       0.08            0        3
sec2                222       45.54       0.12            0        3
sec3                112       17.71       0.02            0        2
sec4                110        8.50       0.01            0        1
sec1_1              110       68.72       0.09            0        2
```

Conveyor section statistics

Section statistics are defined as follows:

Section The name of the section.

Entries The total number of loads that traveled on the section.

Average Time/Ent The average time a load spent on the section.

Ave The average number of loads on the section.

Current Contents The current number of loads on the section.

Max The maximum number of loads that were on the section at the same time.

Summary

In this chapter, you learned how to transport loads using a conveyor system. The basic steps for creating a model containing a conveyor system are:

Step 1 ***Create*** a new model.

Step 2 ***Create*** the conveyor system.

Step 3 ***Draw*** the conveyor sections, using the drawing grid and Measurement window to draw the sections to scale.

Step 4 ***Place*** stations where loads get on and off the conveyor and where processing is performed.

Step 5 ***Define*** the process system, including:

- The model logic
- Processes
- Resources and queues
- Loads

Step 6 ***Place*** the graphics for resources, queues, and loads.

Step 7 ***Define*** the run control.

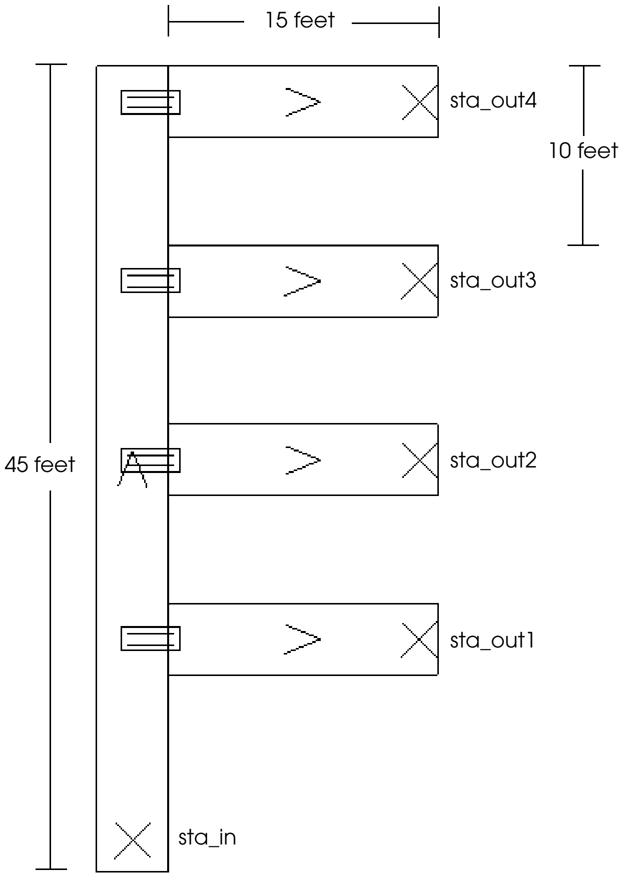
Step 8 ***Export*** and ***run*** the model.

Step 9 ***Analyze*** the simulation statistics.

Exercises

Exercise 6.1

Create a new model, then draw the conveyor shown below:



Loads are created with an interarrival time that is exponentially distributed with a mean of 15 seconds. Loads enter the conveyor system at station “sta_in.” Loads then travel sequentially to one of the four exit stations (“sta_out1” through “sta_out4”). The load size is defined as:

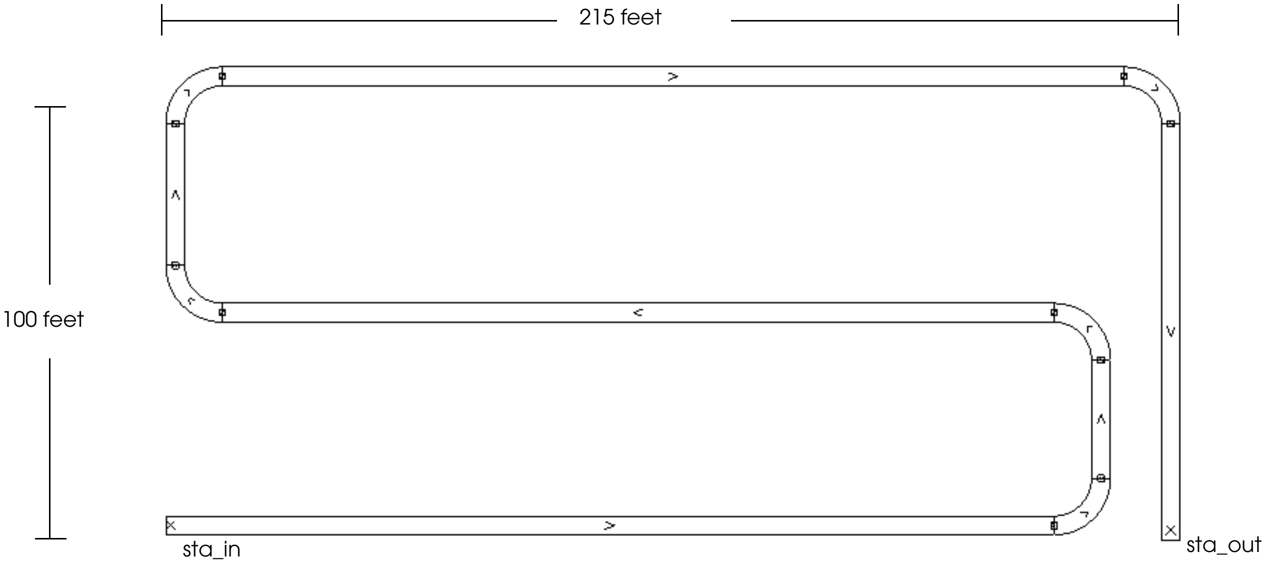
X = 2
Y = 3
Z = 1

Run the model for one day.

What are the average number of loads on each of the four exit sections of conveyor?

Exercise 6.2

Create a new model, then draw the conveyor shown below:



Loads are created with an interarrival time that is exponentially distributed with a mean of 10 seconds. Loads first move into an infinite-capacity queue. They then get on the conveyor at station “sta_in.” Loads travel to station “sta_out” at the end of the conveyor where they leave the system. The load size is defined as:

X = 3
Y = 3
Z = 1

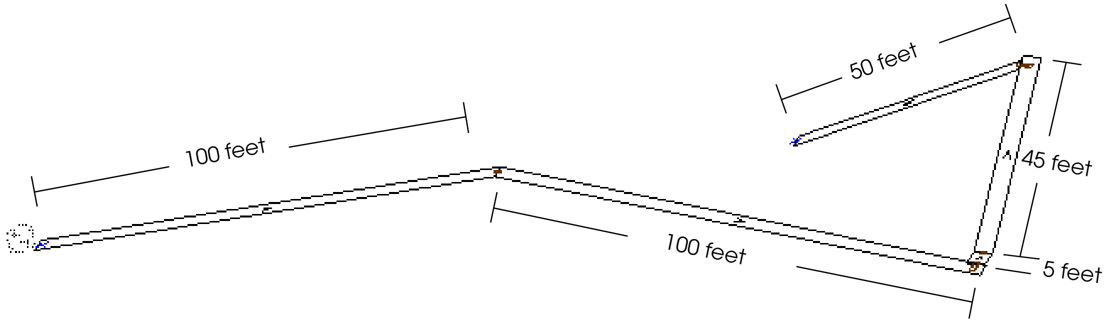
Run the model for five days, then answer the following:

a) What was the average time each load spent in the simulation?
b) What was the average number of loads in the simulation?
c) What was the maximum number of loads in the simulation?
d) What was the average number of loads in the queue?
e) What was the maximum number of loads in the queue?

Exercise 6.3

A conveyor system consists of a ramped conveyor. The conveyor starts at floor level and rises 20 feet in 100 feet. Then it drops back to the floor in 100 feet. A five-foot section stays at floor level, after which the conveyor rises 20 feet in 45 feet. Finally, the conveyor drops back to the floor.

Create a new model, then draw this conveyor, as shown below:



Loads are created with an interarrival time that is exponentially distributed with a mean of 6 seconds. Loads first move into an infinite-capacity queue. Loads then get on the conveyor and travel to the end of the conveyor, where they leave the system. The load size is defined as:

X = 2

Y = 2

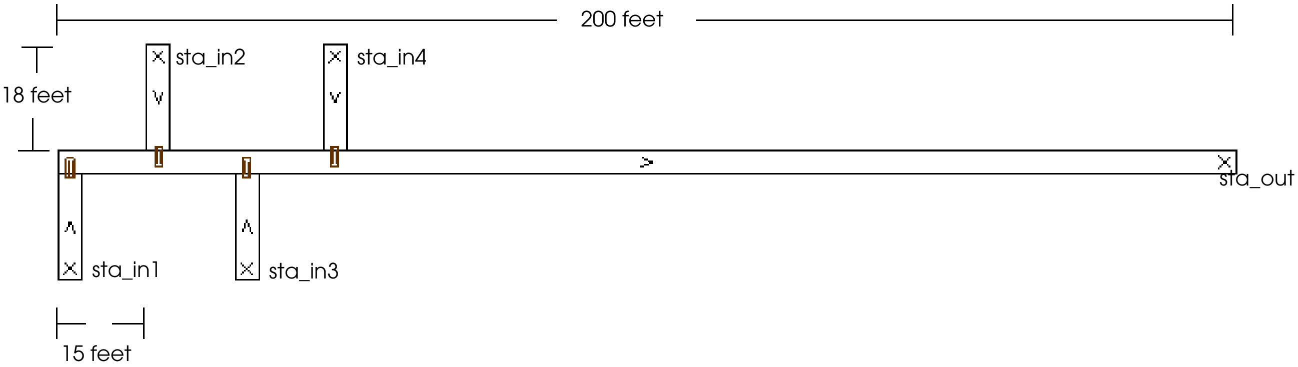
Z = 2

Run the model for five days then answer the following questions:

a) What was the maximum number of loads in the system?
b) What was the average number of loads in the system?
c) What was the maximum number of loads in the entry queue?

Exercise 6.4

Create a new model, then draw the conveyor shown below:



Loads are created with an interarrival time that is exponentially distributed with a mean of 3 seconds. Loads sequentially get on the conveyor at one of four entry stations (station “sta_in1” through “sta_in4”). Loads then travel to station “sta_out” where they leave the system. The load size is defined as:

X = 2
Y = 2
Z = 1

Run the model for five days, then answer the following:

a) What was the maximum number of loads on any of the entry conveyor sections?
b) What was the maximum number of loads on the horizontal section of conveyor?
c) What was the average number of loads on the horizontal section of conveyor?

Chapter 7

Advanced Process System Features

Chapter 7

Advanced Process System Features

This chapter introduces new syntax that is useful for modeling complex systems. First, you will learn how to use variables and load attributes to store and change data during a simulation. You will also use variables and load attributes to track custom statistics and write logic that repeats a specific number of times during a simulation.

This chapter discusses new actions that can give you greater control of a simulation. For example, until now, you have always created loads by defining a load creation specification. In this chapter you will learn how to create new loads and make copies of existing loads using actions in an arriving procedure. You will also learn how to end the simulation using an action in a procedure.

This chapter introduces a method for reading external data from a file and using it in a simulation. Reading data from a file allows easy use of data from other sources, such as a control system. It also allows you to run models using different data sets.

Finally, the chapter introduces two new ways to select from a series of entities.

Storing information in variables and load attributes

Variables and load attributes are used to store information during a simulation. This section discusses:

- Defining variables.
- Defining load attributes.
- Defining variable and load attribute types.
- Setting and changing variable and load attribute values.

Defining variables

A **variable** is a user-defined entity that stores information in a model. You can use a variable to store numbers, words, and other data, then reference it throughout the simulation.

For example, you could use a variable to store a processing time, as shown below:

```
use R_lathe for e V_proctime
```

A load executing this action uses the resource R_lathe for a time that is exponentially distributed with a mean that is stored in the variable V_proctime. The variable can represent any time value, such as 10 seconds or 20 minutes.

Any load in a model can reference the value of a variable. For example, if ten loads executed the above action, all ten would process for a time that is exponentially distributed with a mean of V_proctime seconds.

Variables are important because they allow you to create flexible models. Rather than entering data directly in the logic, you can use variables to represent the data. Then, when you run the model, the variables are replaced with specific values. Variables allow you to conduct experiments by running the same model with different data values. For example, you could vary the value of the variable V_proctime in the syntax above to see what effect longer or shorter processing times have on the system.

Tip ☞ In chapter 10, “Intermediate Statistical Analysis,” you will learn how to conduct experiments with the AutoStat software to vary the value of a variable.

Variable values can be changed during a simulation using the `set` action, which is discussed in “Setting variable and load attribute values” on page 7.7.

Like other entities, variables can be defined from the Process System palette or by using a variable name in the model logic and then defining the variable when saving and quitting the source file.

Defining load attributes

Like a variable, a **load attribute** is a user-defined entity that stores data. Attributes, however, store a different piece of data for each load in the model, whereas a variable stores a piece of data that is the same no matter which load is referencing it. For example, you might create an attribute in which each load stores its part type. This attribute will contain different data depending on which load references the attribute; a load of part type A could have "PartA" as its attribute value, while a load of part type B could have "PartB," and so on.

The previous section showed how to use a variable to store a processing time. The time was the same regardless of which load executed the `use` action. But what if the processing time varied for each part in the system? You could store the time in an attribute of the load and use the following syntax:

```
use R_lathe for e A_proctime
```

This action causes loads to use the resource R_lathe for a time that is exponentially distributed with a mean that is stored in the load attribute A_proctime. The value of A_proctime can vary depending on which load executes the action (that is, the value may be 45 seconds for some loads, 65 seconds for others, and so on).

Attribute values can be changed during a simulation using the `set` action, which is discussed in "Setting variable and load attribute values" on page 7.7.

Like other entities, load attributes can be defined from the Process system palette or by using a load attribute name in the model logic and then defining the attribute when saving and quitting the source file.

Determining when to use variables versus load attributes

Variables and attributes are both user-defined ways to store data during a simulation. Use the following guidelines to decide when to store data using a variable and when to use a load attribute:

Use variables to track information that applies to the entire model.

Because variable values can be referenced and changed by any load, they are useful for tracking values that describe the system being simulated. For example, you can use variables to track the number of loads that are currently in the simulation or the number of resources that are currently unavailable.

Use load attributes to track information that is specific to each load.

Load attributes are used to store or track information that describes a load. The type of information that you might want to store in load attributes will vary based on what the loads represent in the system. For example, if loads represent vehicles arriving at a warehouse, load attributes can be used to assign a parking location to each vehicle. If loads represent parts in the warehouse, then load attributes can be used to assign each part to a specific machine or track the amount of time that each part spends in the system.

Defining variable and load attribute types

When you define a variable or a load attribute, you must specify what type of data the variable or load attribute stores. Variables and load attributes can store numeric types, such as those listed in the table below:

Type	Description
Integer	A whole number. For example 0, 1, –50, and 13256 are all integer values.
Real	A number that can contain a fractional part. For example 1.0, 359.69 and –50.56 are all real values.
Time	A number that represents a time in seconds. For example the value 5 represents 5 seconds, the value 3.468 represents 3.468 seconds, and so on.

Variables and load attributes can also represent **string** values, which can be any combination of alphanumeric characters enclosed in quotes. For example “PartA,” “the 5 loads,” “135hd98” and “15” are all string values.

Variables and load attributes can also represent entities in the simulation. Variables and load attributes that represent entities are called **pointers**. Some commonly used pointers are shown in the table below:

Type	Description
QueuePtr	A pointer to a queue in the model.
ResourcePtr	A pointer to a resource in the model.
LoadTypePtr	A pointer to a load type in the model.

Pointer variables and attributes store the name of an entity in the simulation. For example, a variable that points to the queue Q_enter has the value “Q_enter”. Pointers are often used to refer to a randomly selected entity in the model logic. For example, the procedure below sets the value of the load attribute A_resource to a sequentially selected machine. The load attribute A_resource is of type ResourcePtr.

```
begin P_select arriving
   set A_resource to nextof(R_mach(1), R_mach(2), R_mach(3))
      /* Set the value of A_resource to point to the next machine */
   print “This load is using resource ” A_resource to message
      /* Print the selected machine’s name to the Message window */
   use A_resource for e 5 min
      /* Use the selected machine */
   send to P_next
end
```

After storing a pointer to the selected machine, the load is able to refer to the selected resource later in the procedure using the load attribute A_resource. For example, the first time a load executes the procedure, the value of A_resource is set to “R_mach(1)”. The load prints the message “This load is using resource R_mach(1)” to the Message window and then uses resource R_mach(1) for a time that is exponentially distributed with a mean of 5 minutes. When the next load executes the procedure, the value of its A_resource attribute is set to “R_mach(2),” and so on.

Later in this chapter, you will learn how to select from a group of queues based on which queue contains the fewest number of loads; the selected queue is saved in a variable or load attribute of type QueuePtr (see “Determining which queue contains the fewest loads” on page 7.29).

The syntax that is used for setting and changing the value of variables and load attributes is discussed in more detail in the next section.

Setting variable and load attribute values

The value of a variable or load attribute at the beginning of a simulation is referred to as the **initial value**. The initial value of numeric variables (type Integer, Real, or Time) is set when you define the variables in the software. (For information about defining variables, see “Defining variables and load attributes in example model 7.1” on page 7.10.) The initial value of numeric load attributes is automatically set to zero. The initial value of non-numeric variables and load attributes (for example, pointers) is automatically set to null. A **null** value indicates that there is no value yet assigned to the variable or load attribute; that is, the variable or attribute is not currently referencing any data in the simulation.

During a simulation, you can set or change the value of any variable or load attribute using the `set` action. For example, the procedure below sets the values of several variables and load attributes:

```
begin P_init arriving procedure
   set V_intval to 5                /*V_intval is of type Integer*/
   set V_intval to 6                /*The value changes from 5 to 6 */
   set A_realval to 5.3             /*A_realval is of type Real*/
   set V_timeval to 5               /*V_timeval is of type Time*/
   set A_stringval to “Simulation”  /*A_stringval is of type String*/
   set V_queueptr to “Q_enter”      /*V_queueptr is of type QueuePtr*/
   set A_loadtypeptr to “L_part”    /*A_loadtypeptr is of type LoadPtr */
end
```

Tip ☞

You can use the equal sign (=) in place of the `to` syntax. For example,

```
set V_intval = 5
```

is the same as

```
set V_intval to 5
```

You can also set the value of a variable or load attribute to the value of another variable or load attribute, as shown below:

```
set A_intval to 5
set V_intval to A_intval /* V_intval is now 5 */
set V_realval to 3.32
set A_realval to V_realval /* A_realval is now 3.32 */
```

Important ⚠

Some variable and load attribute types are inconsistent and their values cannot be set to one another. For example, you cannot set a variable of type Real to the value of a variable of type QueuePtr. If you try to mix inconsistent types in the model logic, an error appears when you save and quit the source file.

Incrementing or decrementing the value of a variable or load attribute

You can also increase or decrease the value of any numeric variable or load attribute using the `increment` or `decrement` actions, respectively, as shown below:

```
begin P_calc arriving procedure
   set V_firstval to 5                  /* V_firstval is now 5 */
   increment V_firstval by 1            /* V_firstval is now 6 */
   set V_secondval to 3                 /* V_secondval is now 3 */
   increment V_secondval by 2           /* V_secondval is now 5 */
   decrement V_firstval by V_secondval  /* V_firstval is now 1 */
   decrement V_secondval by V_firstval  /* V_secondval is now 4 */
end
```

Tip ☞

You can use the abbreviated syntax `inc` in place of `increment,` and `dec` in place of `decrement`. For example,

```
inc V_numval by 2
dec V_numval by 3
```

is the same as

```
increment V_numval by 2
decrement V_numval by 3
```

Example model 7.1 will demonstrate how to define variables and load attributes.

Example 7.1: Processing widgets by part type

Four types of widgets are processed in a plant; the widgets are color-coded by part type (red, blue, green, or yellow). Widgets arrive in sets, with each set containing four widgets. The interarrival time between each set is exponentially distributed with a mean of 7 minutes. The number of each type of widget in a set is randomly determined. Data collected from the system indicates that a widget has a 30 percent chance of being a red widget, a 20 percent chance of being a blue widget, a 25 percent chance of being a green widget, and a 25 percent chance of being a yellow widget.

Upon arrival, widgets are automatically sorted and placed into waiting queues to be processed by one of four machines. Each machine has its own infinite-capacity waiting queue and single-capacity processing queue. Each machine can process only one type of widget (the first machine processes only red widgets, the second machine processes only blue widgets, the third machine processes only green widgets, and the fourth machine processes only yellow widgets). Each machine can only process one widget at a time.

The time required to process each widget has been collected in a text file by the control system; this file can be used to determine the processing time of each load at a machine. The processing times that have been collected in the file are recorded in minutes.

After processing, the widgets move into an infinite-capacity queue to await inspection. Widgets are assigned to one of two inspectors in sequential (round-robin) order. Each inspector has a separate, single-capacity queue where loads are inspected. Inspection takes a time that is exponentially distributed with a mean of 2 minutes 30 seconds. Each widget has an 8 percent chance of being rejected after inspection. Rejected widgets are sent back to the correct waiting queue to be re-processed by a machine. (Reworked widgets have the same chance of being rejected, and can be reworked as many times as necessary until they pass.) Widgets that pass inspection are removed from the system.

The time between failures for each of the processing machines is exponentially distributed with a mean of 220 minutes. The time required to repair each machine is exponentially distributed with a mean of 10 minutes.

You need to simulate the system until the data file is out of data for the widget processing times. During simulation, track the number of loads that are currently in the system. Also, when a widget completes processing, print the time that the widget spent in the system to the Message window before it leaves the simulation.

Defining variables and load attributes in example model 7.1

The logic that is used to simulate example 7.1 has already been defined in the base version of the example model. However, the logic is currently commented because the variables and load attributes in the logic have not yet been defined. Therefore, the first step is to open the model and define the name and type of each variable and load attribute.

Note

The initial value of a numeric variable is set when you define the variable. The value of load attributes and non-numeric variables must be set using the `set` action in the model logic (see “Setting variable and load attribute values” on page 7.7 for more information).

To define the variables and load attributes in the model:

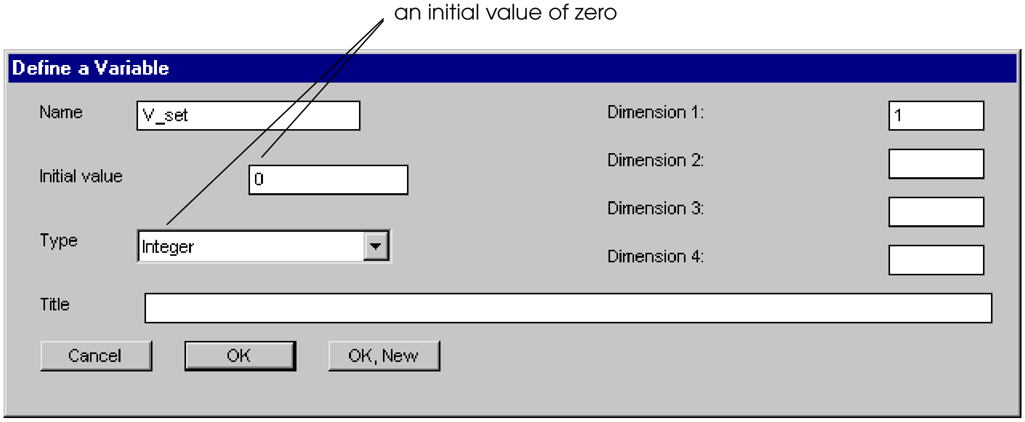
Step 1 ***Import*** a copy of the *base* version of example model 7.1.

Step 2 ***Edit*** the model logic and ***delete*** the comment markers `/*` and `*/` at the beginning and end of the source file.

Step 3 From the File menu, ***select*** Save and Quit. The Error Correction window opens, indicating that V_set is undefined. The V_set variable is used to limit the number of widgets in each set to 4 (an integer value).

Step 4 To define the variable, ***select*** Define. In the Define as drop-down list, ***select*** Variable.

Step 5 ***Click*** Define as. The Define a Variable window opens, as shown below:



The variable is set to store values of type Integer, with an initial value of zero

Name: V_set
Initial value: 0
Type: Integer
Title:
Dimension 1: 1
Dimension 2:
Dimension 3:
Dimension 4:

Defining the V_set variable

Options in the Define a Variable window are defined as follows:

Name The name of the variable.

Initial value The value of a numeric variable at the beginning of the simulation.

Type The type of variable you are defining (see “Defining variable and load attribute types” on page 7.6).

Title A comment that describes the variable (optional).

Dimension The number of variables that you are creating. Changing the value of Dimension 1 to a number greater than one creates an array of variables. For example, if you set the value of Dimension 1 to “4,” then four different V_set variables would be created, each of type Integer. As with other arrayed entities, you can refer to the arrayed variables in logic by using a parenthetical number after the variable name, for example:

```
set V_set(1) to 3
set V_set(2) to 2
```

and so on. Changing the value of a variable’s 2nd, 3rd, and 4th dimensions is not discussed in this textbook.

Step 6 Because Integer is already selected as the variable type and the initial value is already set to zero, ***click*** OK to define the variable.

Step 7 The Error Correction window opens, indicating that A_type is undefined. The A_type attribute is used to assign an integer value to each load to represent its part type (1, 2, 3, or 4).

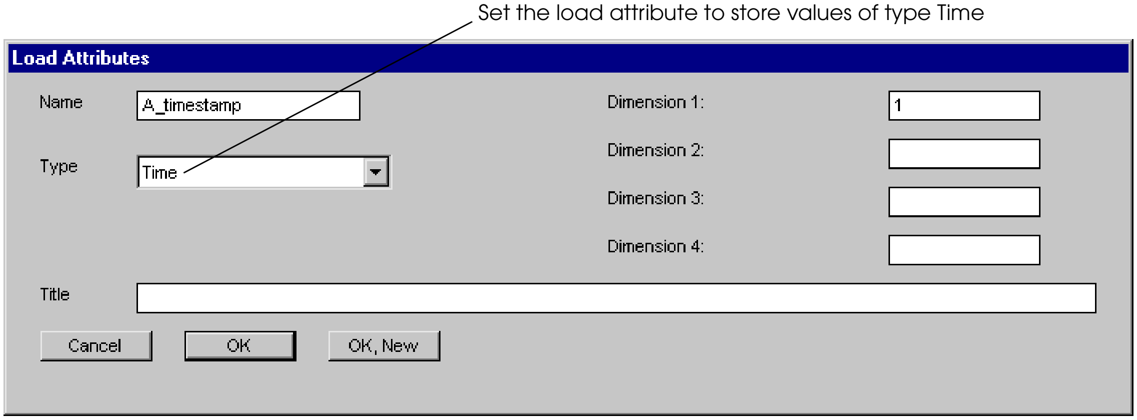
Step 8 In the Define as drop-down list, ***select*** Load Attribute and ***click*** Define as. The Load Attributes window opens.

Step 9 Because Integer is already selected as the attribute type, ***click*** OK to define the load attribute.

Step 10 The Error Correction window opens, indicating that A_timestamp is undefined. The A_timestamp attribute is used to track the time that each widget spent in the system.

Step 11 ***Click*** Define as. The Load Attributes window opens. To define the attribute type, ***select*** Time in the Type drop-down list, as shown below:

Set the load attribute to store values of type Time



Load Attributes

Name: A_timestamp

Type: Time

Dimension 1: 1

Dimension 2:

Dimension 3:

Dimension 4:

Title:

Defining the A_timestamp attribute

Step 12 ***Click*** OK to define the load attribute. The Error Correction window opens, indicating that V_insystem is undefined. The V_insystem variable is used to track the number of widgets that are currently in the simulation (an integer value). Because there are no widgets in the plant at the beginning of the simulation, the initial value of the variable should be zero.

Step 13 In the Define as drop-down list, ***select*** Variable and ***click*** Define as. The Define a Variable window opens.

Step 14 By default, the variable is already set to store values of type Integer and the initial value is set to zero, so ***click*** OK.

Step 15 ***Define*** the remaining load attributes, as shown in the table below:

Load attribute name	Type
A_time	Time
A_insp	Integer
A_index	Integer

Tip

If you incorrectly define the type of a variable or load attribute, you can edit it by clicking Variables or Loads on the Process System palette after closing the source file. You may need to comment the code before you can close the source file.

You have now defined the variables and load attributes that are used in the model logic.

Step 16 ***Export*** and ***run*** the model.

Step 17 Once you are familiar with the processing of loads in the system, ***edit*** the model.

You are now ready to look at the logic used to model this system.

Defining the model initialization function

There are five load types in example model 7.1, but there are no load creation specifications; all of the loads in the simulation are created in the model logic. The first action that creates loads appears in a new piece of logic known as a function. Like arriving procedures, functions are defined using the `begin` and `end` syntax. Functions are discussed in detail later in this textbook (see “Performing calculations with functions” on page 14.19 of the “Additional Features” chapter). In this chapter, however, we introduce one predefined function, called the model initialization function.

The **model initialization function** is a section of model logic that is automatically executed at the beginning of each simulation. The logic is executed as soon as the Simulation window opens (even before you continue a paused simulation). The model initialization function is useful for setting some of the initial conditions in a model. Example model 7.1 uses the model initialization function to create loads to start the simulation, as shown below:

```
begin model initialization function
   create 4 loads of type L_dummy to P_machdown /* Model down times */
   create 1 load of type L_dummy to P_start /* Start production */
   return true
end
```

Before learning how to create loads in the model initialization function, you need to know two important characteristics of functions:

- Some actions are illegal in functions.
- Functions must return a value.

Actions that are illegal in functions

You cannot cause a time delay in a function. Therefore, several AutoMod actions cannot be used in a function. Of the actions that have been discussed in this textbook, the following actions cannot be used in a function:

- `free`
- `get`
- `move`
- `send`
- `travel`
- `use`
- `wait`

Help

For a complete list of the actions that are illegal in a function, see the AutoMod Syntax Help.

Returning a value from a function

Functions must return a value using the `return` action. Often, functions are defined to perform one or more calculations and then return a value that is used in the simulation. For example, a function might compare a sequence of numbers and then return the largest number in the sequence. The model initialization function is a special type of function that returns a value that is not used. However, because all functions must return a value, the following syntax is used at the end of the model initialization function:

```
return true
```

The `return` action must be the last action in a function. Writing functions that use a return value is discussed in greater detail later in this textbook (see “Return value” on page 14.20 of the “Additional Features” chapter).

Creating new loads in the model logic

The model initialization function in example model 7.1 creates new loads to model machine down times and to model the arrival of widget sets in the plant. The loads are created using the `create` action, as shown below:

```
   create 4 loads of type L_dummy to P_machdown /* Model down times */
   create 1 load of type L_dummy to P_start /* Start production */
```

The first `create` action creates four dummy loads of type L_dummy that are sent to the process P_machdown, which defines the machine failures, as shown below:

```
begin P_machdown arriving
   /* Model down time for each of the four machines */
   set A_index to nextof(1,2,3,4)
   while 1=1 do begin
      wait for e 220 min
      take down R_mach(A_index)
      wait for e 10 min
      bring up R_mach(A_index)
   end
end
```

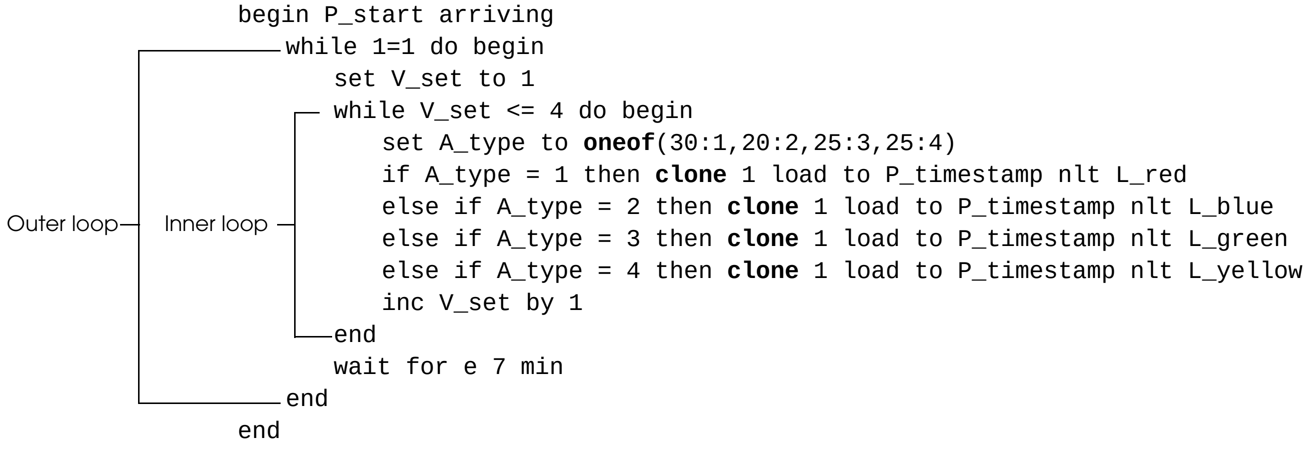
The `nextof` distribution assigns a sequential integer value to the load attribute A_index for each of the four loads that execute the procedure (the value of the first load’s attribute is set to 1, the value of the second load’s attribute is set to 2, and so on).

The four processing machines are modeled as an array. After a load sets the value of its A_index attribute, it executes the `while...do` loop that causes it to take down and bring up one of the arrayed machines throughout the simulation. The value of the A_index attribute is used to align each load with an arrayed machine (the first load takes down and brings up the first machine in the array, the second load takes down and brings up the second machine, and so on).

The second `create` action creates a single dummy load of type L_dummy that is sent to the process P_start. The arriving procedure for P_start contains two `while...do` loops that make copies of the L_dummy load to create sets of widgets in the plant, as discussed in the next section.

Writing repeating logic

At the beginning of the simulation, the model initialization function creates one load of type L_dummy and sends it to the P_start arriving procedure, shown below:



```
begin P_start arriving
   while 1=1 do begin
      set V_set to 1
      while V_set <= 4 do begin
         set A_type to oneof(30:1,20:2,25:3,25:4)
         if A_type = 1 then clone 1 load to P_timestamp nlt L_red
         else if A_type = 2 then clone 1 load to P_timestamp nlt L_blue
         else if A_type = 3 then clone 1 load to P_timestamp nlt L_green
         else if A_type = 4 then clone 1 load to P_timestamp nlt L_yellow
         inc V_set by 1
      end
      wait for e 7 min
   end
end
```

The procedure makes clones, or copies, of the L_dummy load to simulate the arrival of the widget sets in the simulation. The procedure contains new syntax: the `oneof` distribution and the `clone` action. The procedure also contains two loops of logic controlled using `while...do` actions. The outer loop repeats continuously throughout the simulation, and the inner loop (nested within the outer loop) uses a variable to repeat a limited number of times.

Writing logic that repeats indefinitely

The outer loop in the P_start arriving procedure is a `while...do` loop that repeats continuously throughout the simulation. The loop is used to model the delay between arrivals of widget sets in the simulation. The loop is defined using the `while...do` syntax that you have used to model down time procedures in previous chapters.

Look at the outer loop, as shown below:

```
begin P_start arriving
   while 1=1 do begin
      set V_set to 1
      ...
      (Inner loop is defined here)
      ...
      wait for e 7 min
   end
end
```

The first action in the loop sets the value of the variable V_set to one. This means that each time the outer loop is executed, the value of the V_set variable is reset to one so that the inner loop can count from one to four. After setting the variable, the inner loop is executed (discussed in the next section).

When the inner loop is completed, a time delay, which is exponentially distributed with a mean of seven minutes, is taken. The delay represents the interarrival time between sets of widgets in the simulation. After the time delay, the outer loop repeats.

Important ⚠

Whenever you write logic that repeats continuously throughout a simulation, the logic must contain a time delay to avoid an infinite loop (see “Avoiding infinite loops” on page 7.17).

Writing logic that repeats a limited number of times

The nested (inner) loop in the P_start procedure demonstrates how to limit the number of times a loop is executed. The inner loop models the creation of widget sets in the simulation. Each time the inner loop is executed, a copy of the L_dummy load is cloned to represent a widget (cloning loads is discussed in “Cloning loads in the model logic” on page 7.19). Because a widget set contains four widgets, the loop must be executed four times, as shown below:

```
while V_set <= 4 do begin
    ...
    (A copy of the L_dummy load is cloned here)
    ...
    inc V_set by 1
end
```

The `while...do` loop uses the variable V_set to define a condition that determines the number of times the loop is executed. In the loops that you have defined previously, you have always used the condition `while 1=1` to make the loop repeat indefinitely. This example uses the condition `while V_set <= 4` to cause the loop to repeat as long as the value of the variable V_set is less than or equal to four. This condition generates the required number of widgets in a set. You can use any of the following relational operators to define a condition in a `while...do` loop:

Relational operator	Description
=	equal to
is	equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
<>	not equal to

The first time that the L_dummy load executes the inner loop, the value of the variable V_set is one (recall that the variable’s value is set in the outer loop). Because one is less than four, the inner loop is executed and a copy of the L_dummy load is made to represent a widget. Before ending the inner loop, the value of the V_set variable is incremented by one.

The L_dummy load then attempts to execute the loop again. V_set is now equal to two. Because two is less than four, the inner loop is executed again and another copy of the L_dummy load is made to represent another widget in the set. Before ending the inner loop, the value of the V_set variable is incremented again (the variable is now equal to three).

The loop repeats until all four loads in the set have been created and the value of the V_set variable is equal to five. Because the value of the variable no longer meets the condition required to execute the inner loop, the load stops executing the inner loop and moves on to the next action in the outer loop (the time delay). When the load repeats the outer loop, the value of the variable V_set is reset to one and the inner loop is executed four more times to create the next set of widgets. This process repeats throughout the simulation.

Important ⚠ Variable and load attribute values must be initialized to the correct value before executing a loop.

Because the inner loop does not include a time delay, the loop executes four times instantaneously (the set of four widgets is created at the same instant in the simulation).

You can also write logic that decrements a variable to repeat a limited number of times. For example, consider modeling a system in which parts are removed from boxes by a machine. You would write a loop that tracks of the number of parts in each box (a variable number), as shown below:

```
begin P_box arriving procedure
   set A_parts to uniform 10, 2
      /* A_parts is a load attribute of type Integer */
   while A_parts > 0 do begin
      use R_machine for uniform 15, 5 sec
      dec A_parts by 1
   end
   send to P_next
end
```

The procedure is executed by loads that represent boxes in the system. A box first sets the value of its load attribute A_parts to randomly determine the number of parts in the box. Then, while the number of parts is greater than zero, the box uses R_machine to remove a part and decrements the number of remaining parts in the box. When the box is empty (A_boxes = 0), the loop stops repeating and the box is sent to the process P_next.

Loops that repeat a limited number of times are used frequently. However, special care must be taken to avoid a common modeling error: accidentally creating an infinite loop, discussed next.

Avoiding infinite loops

You have learned that a loop is a series of actions that repeats until a condition is met or as long as a condition remains true. An **infinite loop** is a loop in which the condition is always true *such that the loop repeats indefinitely without any delays between cycles of the loop*. There are several ways to accidentally create an infinite loop:

- Write a continuously repeating loop (`while 1=1`) that does not contain a time delay.
- Write a finite loop that does not contain a time delay and for which the condition is always true.

An infinite loop does not contain a time delay between actions. Because there is not a time delay, the actions in the loop all occur at the same instant during a simulation and repeat indefinitely, which causes the simulation clock to stop advancing. As a result, the simulation environment “freezes up” and becomes unresponsive.

Infinite loops in continuously repeating loops

There are situations in which you want actions to repeat continuously throughout a simulation, such as modeling resource down times. To model continuous behavior, you can use the condition `while 1=1` as long as you include a time delay in the loop, such as a `wait` or `use` action, so that other actions can be executed concurrently.

For example, if a down time loop did *not* contain a delay, such as a `wait` or `use` action, the resource would go down and up endlessly at the same instant in the simulation, as shown below:

```
begin P_opershift arriving
   while 1=1 do begin
      take down R_oper
      bring up R_oper
   end
end
```

Because there are no delays, the clock cannot advance, so nothing else happens in the model. To avoid an infinite loop, you must always include an action that causes a delay, such as:

```
begin P_opershift arriving
   while 1=1 do begin
      wait for 55 min /* other events can occur during this time delay */
      take down R_oper
      wait for 5 min /* other events can occur during this time delay */
      bring up R_oper
   end
end
```

Infinite loops in finite loops

Often you use loops to repeat actions a fixed number of times, such as repeating a loop to count the number of loads on a truck or for the number of parts in a bin. If you forget to adjust the value of the variable or load attribute that you are using for the condition, and there is no delay action, you will create an infinite loop.

For example, consider what would happen if you left out the `increment` action in the inner loop of the P_start arriving procedure, as shown below:

```
while V_set <= 4 do begin
   set A_type to oneof(30:1,20:2,25:3,25:4)
   if A_type = 1 then clone 1 load to P_timestamp nlt L_red
   else if A_type = 2 then clone 1 load to P_timestamp nlt L_blue
   else if A_type = 3 then clone 1 load to P_timestamp nlt L_green
   else if A_type = 4 then clone 1 load to P_timestamp nlt L_yellow
   /* The increment action has been omitted */
end
```

V_set is initialized to 1 before the loop. Notice that there are no actions that cause delays in the loop (no `wait` or `use` actions), and the `increment` action has also been omitted. Therefore, the loop would repeat indefinitely, because 1 is always less than 4. The procedure would continuously clone loads at a fixed instant in the simulation and no other events in the simulation would occur. Your computer would rapidly run out of memory due to the ever-increasing number of loads in the model.

Note

Adding a delay to this loop would prevent an infinite loop, but the loop would still repeat continuously because the condition is always true. The delay would simply allow other simulation events to occur, preventing the model from locking up. You need to add the `increment` action to get the expected results.

If you accidentally create an infinite loop during a simulation, you can end the simulation using the Windows Task Manager, as shown below:

Ending a simulation using the Windows Task Manager

Note

These steps were written for a computer that is running Windows NT 4.0. Refer to the help for your operating system for information about ending a task using a different version of the Windows operating system.

To end a simulation:

Step 1 ***Press*** Ctrl+Alt+Delete. The Windows NT Security window is displayed.

Step 2 ***Click*** Windows Task Manager. The Windows NT Task Manager opens.

Step 3 On the Applications tab, ***select*** the Task with the name of the model that you are currently running.

Important

Make sure that the task you select is the model executable (the name includes “Simulation”), rather than the model editor (the name includes “Work Area”).

Step 4 ***Click*** End Task. The Simulation environment closes.

You can now edit your model to resolve the infinite loop.

Cloning loads in the model logic

Now that you understand how loops are defined, you are ready to learn how to create copies of the L_dummy load to represent widgets in the simulation. Each time the inner `while...do` loop is executed in the P_start arriving procedure, a copy of the L_dummy load is created, set to a new load type, and sent to the process P_timestamp.

Recall from the description of example model 7.1 that each widget arriving in the system has a chance of being a red, blue, green, or yellow widget. This means that each time a copy of the L_dummy load is created, it needs to be assigned a load type (either L_red, L_blue, L_green, or L_yellow). The logic for making four copies of the L_dummy load and setting each copy to a randomly selected load type is shown below:

```
while V_set <= 4 do begin
   set A_type to oneof(30:1,20:2,25:3,25:4)
   if A_type = 1 then clone 1 load to P_timestamp nlt L_red
   else if A_type = 2 then clone 1 load to P_timestamp nlt L_blue
   else if A_type = 3 then clone 1 load to P_timestamp nlt L_green
   else if A_type = 4 then clone 1 load to P_timestamp nlt L_yellow
   inc V_set by 1
end
```

The first `set` action in the loop uses the `oneof` distribution to set the value of the load attribute A_type to 1, 2, 3, or 4, based on the probability defined in the `oneof` distribution (the `oneof` distribution is discussed in detail later in this chapter; see “Randomly selecting from a series of entities” on page 7.28 for more information).

The `clone` action allows you to specify the number of copies of the load you want to create (in this case, one load), the process to which you want to send the cloned load (in this case, P_timestamp), and a new load type for the cloned load (in this case, L_red, L_blue, L_green, or L_yellow).

Tip ☞

The syntax for assigning a new load type in the `clone` action is `new load type`, which can be abbreviated as `nlt`. For example,

```
if A_type = 1 then clone 1 load to P_timestamp new load type L_red
```

is the same as

```
if A_type = 1 then clone 1 load to P_timestamp nlt L_red
```

In this procedure, there are four `clone` actions defined, but because of the `if...then...else` condition, only one `clone` action is executed each time the L_dummy load executes the loop. Which `clone` action is executed depends on the value of the load attribute A_type. If the value of A_type is equal to one, a copy of the load is created and set to the load type L_red, if the value of A_type is equal to two, a copy of the load is created and set to the load type L_blue, and so on.

Because the loop is executed four times, four different copies of the L_dummy load are created and sent to the process P_timestamp, each with a randomly determined load type. These four loads represent the four widgets that comprise a widget set.

Important ⚠

When you clone a load, the copied loads retain the same load attribute values as the original load. For example, when the value of L_dummy’s A_type attribute is one, the copied load (of type L_red) retains the value of one for its A_type attribute. Similarly, copied loads of type L_blue retain an A_type value of two, copied loads of type L_green retain an A_type value of three, and so on. This concept of retaining attribute values is useful because you can use the attribute values later in the model (see “Aligning entities using load attributes” on page 7.22).

Assigning a new load type to cloned loads

When using the `clone` action, cloned loads can either keep the same part type as the original load or be assigned a new part type using the syntax `new load type` (or `nlt`). In example model 7.1, cloned loads are assigned one of four load types L_red, L_blue, L_green, or L_yellow. When you assign a new load type to a cloned load, the graphic of the cloned load inherits the load size and color that have been defined for that load type. In example model 7.1, graphics have been placed for each of the four load types and they have been set to the correct colors (see "Placing load graphics" on page 6.25 of the "Introduction to Conveyors" chapter for more information). Assigning a new load type to the loads allows you to verify that loads are using the correct queues and resources for their type by watching the animation during a simulation.

Determining which method to use when generating new loads

To create new loads in a simulation, you can use any of the following methods:

- Define a creation specification for the load type
- Use the `create` action to create new loads in the model logic
- Use the `clone` action to make copies of an existing load in model logic

Regardless of which method you use, the load type must be defined from the Process System palette. For a given load type, you generally use one load generation approach or the other, that is, either use a creation specification or generate the loads in logic, but not both.

Note

Regardless of which method you use, newly created loads are initially located in Space until they execute an action to move into a different territory.

To determine which method to use when creating loads, use the following general guidelines:

When new loads must be created continuously throughout the simulation, but the initial values of load attributes are not important, generate the loads using a creation specification.

When you generate loads using a creation specification, the attribute values of the newly created loads are initialized to either zero or null (see "Setting variable and load attribute values" on page 7.7). When the initial value of load attributes is unimportant, defining a load creation specification is the easiest way to generate loads throughout a simulation.

When a limited number of loads must be generated at a specific time during simulation, but the initial values of load attributes are not important, generate the loads using the `create` action.

As when defining a creation specification, the attribute values of loads that are created using the `create` action are initialized to either zero or null. The `create` action is often useful when creating a limited number of loads at a specific time during a simulation. For example, you can use the `create` action in the model initialization function to send dummy loads to processes that model resource down times.

When new loads must be generated with specific initial load attribute values, generate the loads using the `clone` action.

Use the `clone` action when you want newly generated loads to have specific load attribute values. In example model 7.1, the load attribute A_type is defined to assign an integer value to each load in the simulation. The initial value of this attribute must be set to 1, 2, 3, or 4 for newly created loads in the system; these values are used later in the simulation to ensure that loads use the correct queues and resources in the system (see "Aligning entities using load attributes" on page 7.22). To create loads with the correct attribute value, the value is first set for a dummy load, which then clones a load with the correct load attribute value. This process, of setting the dummy load's attribute value and then cloning a new load to represent a widget, repeats throughout the simulation.

Tracking custom statistics using variables and load attributes

According to the description of example 7.1, you need to track the number of loads that are currently in the system. Also, when a widget completes processing, you must print the time that the widget spent in the system before removing it from the simulation.

Because the number of loads in the system is a value that applies to the simulation as a whole, you can track the value using a variable. A load's time in the system, however, is a value that is unique to each load in the simulation and must be tracked using a load attribute.

In example model 7.1, cloned loads are sent to the P_timestamp process, shown below:

```
begin P_timestamp arriving
   set A_timestamp to ac /* Set start time */
   inc V_insystem by 1
   send to P_process
end
```

Each cloned load sets the value of its A_timestamp attribute to the current simulation time (ac); this value will be used in the process P_finish to calculate each load's total time in system. Each load then increments the value of the variable V_insystem, which represents the number of widgets in the system.

After leaving P_timestamp, loads are sent first to P_process, then to an inspection process (both are discussed later in this chapter). From inspection, loads are sent to the P_finish process. The P_finish arriving procedure updates the variable and timestamp, then removes loads from the system, as shown below:

```
begin P_finish arriving
   dec V_insystem by 1
   set A_timestamp to (ac - A_timestamp) /* Calculate time in system */
   print “Load of type ” A_type “ had time in system of ”,
      (A_timestamp / 60) ,“ min” to message
   send to die
end
```

The load first decrements the variable V_insystem (to indicate that the load is leaving the system). The load then calculates its total time in system by subtracting the time at which it entered the system (stored in the load attribute A_timestamp) from the current simulation time. The difference is stored in the attribute A_timestamp (the new value replaces the old value).

After calculating its time in system, the load prints its load type and time in system (in minutes) to the Message window and is sent to die.

Aligning entities using load attributes

You have already learned how to align arrayed entities using `procindex` (see “Using procindex to align arrayed entities” on page 5.40 of the “Process System Basics” chapter). As an alternative to using `procindex`, you can also align arrayed entities using load attributes.

In example model 7.1, there are four work areas that each consist of a waiting queue, a processing queue, and a machine. Widgets are sent to the correct area based on part type. The procedure that ensures that each widget moves into the correct queues and uses the correct machine for its type is shown below:

```
begin P_process arriving
    move into Q_mach_wait(A_type)
    move into Q_mach(A_type)
    read A_time from “arc/data.dat”
        at end
            begin
                print “Ran out of data in file data.dat” to message
                terminate
            end
    use R_mach(A_type) for (A_time * 60)
        /* Convert time values from minutes to seconds*/
    send to P_inspect
end
```

Note

The procedure includes new syntax that is used to read processing time data from a text file; the syntax is discussed later in this chapter (see “Reading data from files” on page 7.24).

The waiting queues, the processing queues, and the machines are all modeled as arrays. The attribute A_type is used to guarantee that each load moves into the correct waiting and processing queues and uses the correct machine. For example, a load of type L_red has an A_type value of 1. Consequently the load moves into the waiting queue Q_mach_wait(1), then moves into the processing queue Q_mach(1), and then uses machine R_mach(1). Similarly, a load of type L_blue has an A_type value of 2. Consequently the load moves into the waiting queue Q_mach_wait(2), followed by the processing queue Q_mach(2), and then uses machine R_mach(2), and so on.

In this model, aligning entities using the load attribute A_type is possible because the cloned loads retain the value of A_type that was set for the original load of type L_dummy.

Understanding concurrent processing in a simulation

When you use syntax to create new loads in a model, it is important to understand which loads are executing the procedures in the model. In example model 7.1, a single load of type L_dummy is sent to the process P_start. The L_dummy load executes the P_start arriving procedure and generates all the loads that represent widgets in the system. Notice, however, that the L_dummy load stays in the process P_start throughout the simulation, because the P_start arriving procedure is a continuous loop that causes the L_dummy load to clone four loads, wait for a time delay, clone four more loads, wait for another time delay, and so on. The cloned loads, however, are sent to the process P_timestamp at the same instant during the simulation. This means that all four loads execute the P_timestamp arriving procedure at the same moment during the simulation. Because the P_timestamp arriving procedure does not include any time delays, loads are then sent to the process P_process, again at the same instant during simulation.

Until they are sent to P_process, the four cloned loads have been synchronously executing the same procedures in the simulation. In the P_process arriving procedure, each load moves into the correct waiting queue for its type and waits to be processed by a machine. Because the amount of time that each load processes at a machine is random, the loads become staggered (some loads process at a machine longer than others) and the four loads execute the remaining procedures in the model at different times throughout the simulation.

Reading data from files

You are now ready to learn how to read data from a file during a simulation. In example model 7.1, the time required to process each widget at a machine has been collected in a data file by the control system. The data file is a text file that includes 2000 lines of data; the file is saved in the model archive directory and is named “data.dat.” The first few lines of the data file are shown below:

```
4.25
4.99
8.57
0.04
6.52
3.74
6.10
...
```

The time values in the file are recorded in minutes. The simulation needs to read the time values from the file to determine how long each load should be processed by a machine.

You read values from a file using the `read` action in the model logic. Files that are read during a simulation must be delimited ASCII text. When reading from a file, you can specify any delimiter to separate values based on how the file is formatted; by default, the `read` action uses the whitespace delimiter which treats consecutive spaces, tabs, and/or new lines as a single delimiter. You will learn more about reading files with different delimiters in “Specifying a delimiter when reading from a file” on page 7.32.

In example model 7.1, the P_process arriving procedure includes the syntax for reading and using the time values from the data.dat file, as shown below:

```
read A_time from “arc/data.dat”
   at end
      begin
         print “Ran out of data in file data.dat” to message
         terminate
      end
use R_mach(A_type) for (A_time * 60)
   /* Convert time values from minutes to seconds*/
```

This syntax is described in the following sections.

Defining a file’s location

In example model 7.1, each load reads a time value from the data file and stores it in the load attribute A_time. As with printing to a file (see “The print action” on page 4.12 of the “Introduction to AutoMod Syntax” chapter), when reading from a file, you can specify the path to the data file’s location as either an absolute path (for example, “C:/files/data.txt”) or a relative path from the model directory. In example model 7.1, the “arc/” syntax is used to read from the model’s archive directory.

Tip ☞ Reading from the archive directory is advantageous because it makes your model more portable (you do not need to change the file location if you move the model).

Important ⚠ Use forward slashes “/” when specifying a path to the file you want to read. The file path and name must be enclosed in quotes.

Determining the reading position in a file

When a load reads from a file, the simulation keeps track of the current reading position within the file. After one action stops reading values, the next action that reads from the same file starts at the point where the last `read` action stopped.

In example model 7.1, each load reads only one value from the file. The action reads a time value, then stops reading when a new line is reached (when using the whitespace delimiter, new lines are treated as delimiters). The load stores the value in the load attribute A_time. When the next load executes the arriving procedure, it reads the next time value and stops at the delimiter, and so on.

Reading to the end of a file

By default, when all of the values have been read from a data file, the simulation continues by reading values from the beginning of the file. In example 7.1, you want the simulation to stop when the end of the file is reached.

You can use the `at end` syntax after the `read` action to define one or more actions that occur when the end of the file is reached. In example model 7.1, two actions are defined within the `begin` and `end` syntax, as shown below:

```
read A_time from “arc/data.dat”
   at end
      begin
         print “Ran out of data in file data.dat” to message
         terminate
      end
```

When a load tries to read a value from the data file, but encounters the end of the file instead, the load executes the actions defined after the `at end` syntax. First, the load prints a message to the Message window to indicate that there are no more values in the data file. Second, the load executes the `terminate` action. The `terminate` action immediately stops the simulation (see “Terminating a simulation” on page 7.26 for more information).

Converting data values

The description of example 7.1 states that the control system records processing times in the data file in minutes. However, in example model 7.1, the time values that are read from the file are stored in the attribute A_time, which is defined as a load attribute of type Time. Variables and load attributes of type Time always represent a time value *in seconds*. The syntax:

```
use R_mach(A_type) for A_time
```

causes loads to use an arrayed resource for the number of seconds defined by the value of the attribute A_time, not minutes. Consequently it is necessary to convert the time stored in the load attribute A_time from minutes to seconds. Example model 7.1 converts the values by multiplying them by 60 in the `use` action, as shown below:

```
use R_mach(A_type) for (A_time * 60)
```

Thus, the values are converted for use in the simulation without editing the original data.

Note

It is also possible to change the default units of measurement for both time and distance in the model; however, changing the default model units is not discussed in this textbook.

Terminating a simulation

In example model 7.1, the `terminate` action is used to stop the simulation when all the values in the data.dat file have been read, as shown below:

```
read A_time from “arc/data.dat”
   at end
      begin
         print “Ran out of data in file data.dat” to message
         terminate
      end
```

You can use the `terminate` action in model logic to immediately stop a simulation. The action forces a simulation to quit before the time specified in the run control. When the terminate action is executed, the message “Simulation terminating” is printed to the Message window, and the simulation ends. The model reports are generated and variables and statistics can be viewed from the menus in the Simulation window.

Displaying variable values during a simulation

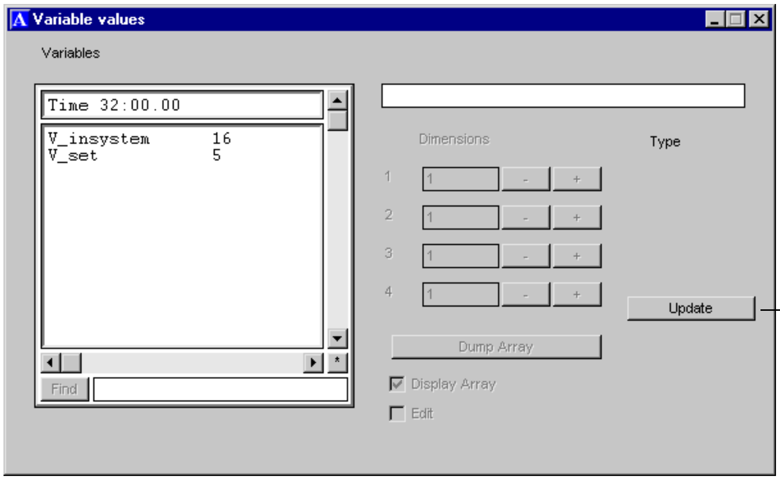
During a simulation, you can display the current value of all variables that are defined in the model. By displaying the value of the variable V_insystem in example model 7.1, you can display the number of loads that are in the system at any point during the simulation.

To display a variable’s value during a simulation:

Step 1 ***Run*** the example model.

Tip ☞ If you have already run the model to completion (and the Simulation environment is still open), you can restart the simulation by selecting Restart Model from the Control menu.

Step 2 While the model is running, from the Loads/Variables menu, ***select*** All Variable Values. The Variable values window opens, as shown below:



```
Time 32:00.00
V_insystem       16
V_set            5
```

Click Update to update the display to the current simulation time

Displaying variable values in example model 7.1

The variable values window displays the current value of all variables at a specific instant during the simulation. In the window above, you can see that 32 minutes into the simulation, there are 16 loads in the system. The value of V_set is 5, indicating that a set of loads has been cloned and the L_dummy load is currently executing the time delay before resetting the value of the variable (see “Writing repeating logic” on page 7.14 for more information).

Tip ☞ To refresh the value of the variables while the simulation is running, click Update in the Variable values window.

Selecting randomly using the oneof distribution

To simulate example 7.1, you must be able to model random selection. The `oneof` distribution allows you to randomly select from a series of values or entities based on the frequency of each selection.

Randomly selecting from a series of values

Recall from the description of example model 7.1 that each widget that arrives in the system has a 30 percent chance of being a red widget, a 20 percent chance of being a blue widget, a 25 percent chance of being a green widget, and a 25 percent chance of being a yellow widget. To model the random arrival of widgets, an integer value (1, 2, 3 or 4) is assigned to the load attribute A_type in the P_start arriving procedure; the L_dummy load uses the load attribute when copying loads to determine the type of the load that is created next (a red widget is created when the value is 1, a blue widget is created when the value is 2, and so on). The load attribute A_type is set to a new random value each time the nested loop is executed.

The syntax that is used to randomly select the integer value is shown below:

```
set A_type to oneof(30:1,20:2,25:3,25:4)
```

Each entry in the series is separated by a comma. The format for each entry is `frequency:value,` where the frequency determines how often each value is selected, relative to the other values. In this case, the load attribute A_type has a 30 percent chance of being set to 1 (representing a red widget), a 20 percent chance of being set to 2 (representing a blue widget), and so on.

Because the frequencies in the `oneof` distribution are normalized, they do not need to add up to 100. For example, consider the following syntax:

```
set V_numbox to oneof(1:3, 1:4, 2:6)
```

The value of the variable V_numbox is set to 3 one-quarter of the time, to 4 one-quarter of the time, and to 6 one-half of the time.

Randomly selecting from a series of entities

In addition to selecting random values, you can also use the oneof distribution to randomly select from a series of entities, such as processes or queues. In example model 7.1, widgets that have completed processing must travel through an inspection process. Each widget that is inspected has an 8 percent chance of being rejected and sent back to the processing machines to be reworked. The logic that is used to simulate the inspection process is shown below:

```
begin P_inspect arriving
    move into Q_insp_wait
    set A_insp to nextof(1,2)
    move into Q_insp(A_insp)
    use R_insp(A_insp) for e 2.5 min
    send to oneof(92:P_finish,8:P_process)
        /* Each load has an 8 percent chance of repeating processing */
end
```

Each load moves into a queue, where it awaits inspection. The nextof distribution is used to move loads alternately into one of two inspection queues, where the loads are inspected by an inspector. After the inspection is complete, the oneof distribution simulates each load’s chance of passing inspection. Each load has a 92 percent chance of passing and an 8 percent chance of being rejected. Loads that pass inspection are sent to the process P_finish. Rejected loads are sent to the process P_process; the rejected loads execute the P_process arriving procedure and move back into a waiting queue to be reworked by a machine.

Example 7.2: Choosing a queue based on the fewest loads

Like the `oneof` distribution, the `choose` action allows you to select from a series of entities. Example model 7.2 demonstrates how to choose from a series of queues based on the number of loads currently in each queue.

Loads arrive at an infinite-capacity queue in a factory. The interarrival time between loads is exponentially distributed with a mean of 20 seconds. The factory consists of three production lines, each with an infinite-capacity processing queue and a single-capacity machine. Loads are assigned to a production line by moving into whichever of the three infinite-capacity processing queues contains the fewest number of loads. Loads are then processed by a machine for a time that is uniformly distributed between 50 and 60 seconds, after which the loads are removed from the system.

The system must be simulated for 100 days.

Determining which queue contains the fewest loads

You can select the queue that contains the fewest loads from a series of queues by using the `choose` action. Loads in example model 7.2 use the `choose` action to select a processing queue and store the name of the queue in an attribute of type QueuePtr, as shown below:

```
begin P_choose arriving
   move into Q_arrive
   choose a queue from among Q_mach(1), Q_mach(2), Q_mach(3)
      whose current loads is minimum
   save choice as A_qpointer
   move into A_qpointer
   if A_qpointer = Q_mach(1)
      then use R_mach(1) for u 55, 5 sec
   else if A_qpointer = Q_mach(2)
      then use R_mach(2) for u 55, 5 sec
   else if A_qpointer=Q_mach(3)
      then use R_mach(3) for u 55, 5 sec
   send to die
end
```

After moving into the infinite-capacity arrival queue, loads use the `choose` action to select from among three processing queues (Q_mach(1), Q_mach(2), and Q_mach(3)). The selection is based on the queue that contains the fewest loads at the instant that the load makes the selection. This selection criteria is represented by the syntax `whose current loads is minimum`.

Note If more than one queue contains the fewest number of loads (for example, if two queues are empty), the `choose` action randomly selects from the queues that satisfy the selection criteria; each qualifying queue has an equal probability of being selected.

After choosing a queue, the load saves a pointer to the queue in the load attribute A_qpointer. The load then uses this pointer in the `if...then...else` syntax to ensure that the load uses the correct machine in its production line. After using the machine, the load is sent to die.

You can see the results of the queue selection by running example model 7.2 to completion and displaying the queue and resource statistics. The average number of loads in each processing queue is 2.73 and the average time that loads spent in each queue is approximately 163 seconds. The queuing statistics are similar for the three machines because the `choose` action maintains an even distribution of work.

Aligning arrayed entities using the “index” attribute

Notice that although the logic in example model 7.2 evenly distributes loads in each of the three production lines, the logic does not take advantage of the fact that the queues and resources are arrayed. Resource selection is based on a lengthy `if...then...else` condition that checks the value of the load attribute A_qpointer and then assigns the load to the correct machine, as shown below:

```
if A_qpointer = Q_mach(1)
   then use R_mach(1) for u 55, 5 sec
if A_qpointer = Q_mach(2)
   then use R_mach(2) for u 55, 5 sec
else if A_qpointer = Q_mach(3)
   then use R_mach(3) for u 55, 5 sec
```

If 20 machines were added to the simulation, this logic would become even more redundant.

You have already learned how to align arrayed entities using either `procindex` or a load attribute. In this section you will learn how to align the arrayed entities using a pre-defined attribute of arrayed entities called `index`.

Using entity attributes

Loads are not the only entities in a model that have attributes. Other entities, such as queues and resources, have pre-defined attributes that provide information about the entity. For example, resources have an attribute named `capacity` that you can use to determine and change a resource’s capacity. Arrayed queues and resources have an attribute named `index` containing the entity’s integer number in the array. For example, the value of the `index` attribute for Q_mach(1) is 1, for Q_mach(2) `index` is 2, and so on.

Help

This chapter only discusses the `index` attribute for queues and resources; for a complete list of attributes available for each entity, see the AutoMod Syntax Help.

To use an entity attribute in the model logic, use an entity pointer followed by the name of the attribute. Example model 7.2 uses the following syntax to align entities:

```
begin P_choose arriving
   move into Q_arrive
   choose a queue from among Q_mach(1), Q_mach(2), Q_mach(3)
      whose current loads is minimum
   save choice as A_qpointer
   move into A_qpointer
   set A_index = A_qpointer index
      /* Set the load’s attribute to the value of the queue’s “index”
         attribute */
   use R_mach(A_index) for u 55, 5 sec
   send to die
end
```

When a load chooses a queue, it saves a pointer to the chosen queue in the load attribute A_qpointer. This pointer is then used with the index attribute in the following syntax:

```
set A_index = A_qpointer index
```

The syntax sets the attribute A_index of the load executing the procedure to the value of the pre-defined index attribute for the chosen queue. For example, when A_qpointer is “Q_mach(1),” A_index is set to 1; when A_qpointer is “Q_mach(2),” A_index is set to 2; and so on.

The load attribute is then used to align the arrayed resource and queue, as shown below:

```
use R_mach(A_index) for u 55, 5 sec
```

For more information, see “Aligning entities using load attributes” on page 7.22.

Example 7.3: Generating loads from a data file

Example model 7.3 reads a data file to generate 30 loads of varying part types and processing times. The loads are generated with an interarrival time that is exponentially distributed with a mean of one minute. When the last load has been processed, the simulation ends.

Reading multiple-column data files

In example model 7.1, you learned how to read data from a file (see “Reading data from files” on page 7.24). The data file contained processing times for equipment, but the data was all in one column and did not contain any heading information. More commonly, files will contain multiple columns of data, often with headings above each column that you will need to ignore. Example model 7.3 shows how to read a tab-delimited, multiple-column file and how to handle headings.

In example model 7.3, a tab-delimited data file is used to define a load’s part type and its processing time. The file contains 30 lines of data like those shown below, with the four part types listed randomly:

```
Part Type   Processing Time (in seconds)
L_one       45
L_two       35
L_three     60
L_four      50
...
```

To store this data for each load as it is read from the file, you need two load attributes: one for the part type and one for the processing time. You also need a **dummy variable** to store the values from the file that you are not going to use during the simulation; the dummy variable will store the first line of the file (the headings). The model uses the following variables and load attributes to store data from the file:

Name	Type	Description
V_headers	string	A dummy variable to store the headings
A_parttype	string	An attribute to store a load’s part type
A_proctime	time	An attribute to store a load’s processing time

There are 30 loads listed in the file. The model must stop after the last load has been processed, so the model uses a variable to count the loads that have been processed.

A dummy load executes the logic in the P_read arriving procedure to read the values from the data file and clone loads to P_process based on the data in the file:

```
begin P_read arriving
   read V_headers from “arc/proctime.txt” with delimiter “\n”
   while 1=1 do begin
      read A_parttype, A_proctime from “arc/proctime.txt” with delimiter “\t”
      set load type to A_parttype
         /* sets the current load type to the value read from the file */
      clone 1 load to P_process /* cloned load has correct part type */
      wait for e 1 min /* interarrival of parts */
   end
end
```

```
begin P_process arriving
   move into Q_machine_wait /* infinite-capacity waiting queue */
   move into Q_machine /* processing queue */
   use R_machine for A_proctime /* processing time varies by load type */
   inc V_total by 1
   if V_total = 30 then terminate
   send to die
end
```

The syntax is discussed in more detail in the next section.

Specifying a delimiter when reading from a file

By default, the `read` action uses the white space delimiter (see “Reading data from files” on page 7.24). If you want to read from a file using a delimiter other than white space, you must specify the delimiter for the `read` action using the syntax `with delimiter`. Use the backslash “\” character to define special characters (such as tab or new line) as delimiters:

“\t” Tab
“\n” New line

The file in example model 7.3 is tab-delimited, so most of the data is read using the tab delimiter. However, all of the headings in the first line of the file are read into one dummy variable using the new line delimiter (“\n”):

```
read V_headers from “arc/proctime.txt” with delimiter “\n”
```

Next, an indefinite loop is created to read the file line by line and store each value in a row in one of two load attributes; the first value is stored in A_parttype, and the second value is stored in A_proctime. The values are read using a tab delimiter (“\t”). To read multiple values in the same `read` action, separate the variables or attributes in which you are storing the data with a comma, as shown below:

```
read A_parttype, A_proctime from “arc/proctime.txt” with delimiter “\t”
```

Note

You can read any number of values in the same action by separating each load attribute or variable with a comma. However, to read a file one row at a time, the number of values read in each action must equal the number of columns in the data file. Otherwise, you will read less or more than one row in the same action and may use the wrong values during the simulation.

After each line is read, the dummy load’s part type is set to the type read from the file, then it clones a load to the process P_process. The dummy load delays for an exponentially distributed interarrival time with a mean of 1 minute, then reads from the file again.

The cloned load’s A_parttype attribute is set to the part type that was read from the file by the dummy load. The cloned load’s A_proctime attribute is set to the processing time read from the file. In P_process, the load is processed for A_proctime seconds. Once processed, the load increments the variable V_total. When the total reaches 30 loads (the number of loads in the data file), the simulation is terminated.

Important

You cannot have a delay between when the V_total variable is incremented and when its value is checked, as shown below:

```
inc V_total by 1
if V_total = 30 then terminate
```

If there is any delay between the two statements, the variable could be incremented past 30 by another load before the `if...then` syntax is executed and the equality would never be true, causing the simulation to continue indefinitely.

Summary

This chapter introduces new syntax that you can use to create flexible models. You will regularly use the techniques that are discussed in this chapter, including:

- Representing values in a simulation using variables and load attributes.
- Writing repeating loops.
- Creating new loads in a simulation using the `create` and `clone` actions.
- Reading data from external data files.
- Randomly selecting from a series of entities using the `oneof` distribution or the `choose` action.

These concepts are necessary to build accurate models that simulate real-world systems.

Exercises

Exercise 7.1

Copy your solution model for exercise 5.9 to a new directory.

Run the simulation for 100 days with each of the following configurations:

a) The three machine queues are selected in a round-robin manner (as they were in exercise 5.9).

b) The three machine queues are selected in a random manner, each with an equal chance of being selected.

c) Waiting loads select the machine queue containing the fewest number of loads.

Record the average number of loads and the average time in the waiting queue for each configuration (a, b, and c). Is choosing the processing queue with the fewest number of loads the most efficient option in this model? Why or why not?

Exercise 7.2

Create a new model in which jobs are generated at a time that is exponentially distributed with a mean of 60 seconds.

Tip ☞ Use the `clone` action to create these jobs.

Jobs are processed in an area containing three single-capacity lathes and three infinite-capacity queues (one for each lathe). Newly created jobs move into the queue that currently contains the fewest number of loads. After moving into a queue, a job waits to use the lathe associated with that queue. Each lathe processes a job for a time that is normally distributed with a mean of 150 seconds and a standard deviation of 20 seconds.

After using the lathe, the jobs move into an infinite-capacity queue to wait for inspection. There is one inspector who can inspect one job at a time; inspection takes a time that is uniformly distributed between 20 and 40 seconds. Of the inspected jobs, 95 percent are “good” jobs that pass quality inspection, the remaining 5 percent are “bad” jobs that are sent back to the lathe queues to be reworked; the bad jobs repeat the same process until they pass quality inspection.

Stop the simulation after 1000 good jobs are completed. Determine the average time that each good job (*not* all jobs) spent in the system.

Exercise 7.3

Create a new model in which jobs are generated at a time that is exponentially distributed with a mean of 60 seconds.

Newly created jobs move into an infinite-capacity queue where they wait to use a single-capacity machine. The machine has its own queue for processing. The machine processes jobs for a time that is exponentially distributed with a mean of 55 seconds. Jobs then leave the system.

When there are 10 or more jobs in the system and a new job is generated, print a message that indicates the current clock time *in minutes* and the number of jobs in the system. Round the time value to two decimal places.

When there are 20 or more jobs in the system and a new job is generated, print a message that indicates the current clock time *in hours* and then terminate the simulation. Round the time value to two decimal places.

Exercise 7.4

Apex Corporation wants to simulate its new widget-making facility. The raw materials needed to construct the widgets arrive on trucks that form a waiting line before parking at one of three loading docks. The trucks arrive at a time that is exponentially distributed with a mean of 20 minutes. Trucks are sequentially assigned to a dock based on the order of their arrival (the first truck parks at dock1, the second parks at dock2, etc.).

One unloader works at the docks and unloads one truck at a time. Each truck contains 20 boxes of widget parts. It takes the unloader a time that is exponentially distributed with a mean of 50 seconds to unload each box. There are three types of boxes on each truck; the boxes are color coded (red, blue, and green) according to the type of parts they contain. The number of boxes of each color on a truck is randomly determined. Data collected from the system indicates that an unloaded box has a 30 percent chance of being a red box, a 30 percent chance of being a blue box, and a 40 percent chance of being a green box.

Tip ☞ Assign a numeric attribute to each load to determine its type.

Each box that is unloaded is placed in an infinite-capacity queue to wait for one of three assembly machines. Each machine has its own assembly queue and can assemble only one type of parts, one box at a time (the first machine assembles only red boxes, the second machine assembles only blue boxes, and the third machine assembles only green boxes).

The time required to assemble each box is defined as follows:

- Red boxes require a time that is exponentially distributed with mean of 1.5 minutes.
- Blue boxes require a time that is exponentially distributed with mean of 1.5 minutes.
- Green boxes require a time that is exponentially distributed with mean of 2 minutes.

After assembly, the widgets move into an infinite-capacity queue to wait for inspection. There is one inspector, who inspects loads in a separate queue. The inspector can inspect only one widget at a time. Inspection takes a uniformly distributed time between 0.25 and 1.25 minutes per widget. After inspection, completed widgets leave the system.

The inspector always takes a 5-minute break after working for a uniformly distributed time between 40 and 60 minutes.

Apex would like to know the following from a seven-day (24 hours-per-day) simulation:

a) How many boxes of each type were assembled?

b) What was the average time a truck spent at each dock (including unloading time)?

c) What was the maximum and average number of trucks waiting in line to park at a dock?

d) What was the maximum and average number of widgets waiting to be inspected?

e) Record the maximum and average number of widgets waiting to be inspected. What happens to these values if the inspector takes 20-minute breaks instead of 5-minute breaks?

Exercise 7.5

Trucks arrive at a dock for unloading. Each truck can contain red, blue, yellow, and/or green pallets. The time between truck arrivals is exponentially distributed with a mean of one hour. Trucks that are waiting to be unloaded form a waiting line before the dock (there is only room for one truck to park at the dock at a time). A total of 100 trucks arrive. The number of pallets on each truck is defined in a data file. Create the file in Excel using the following spreadsheet functions:

$$\text{Number of red pallets} = 1 + \mathrm{INT}(2 \times \mathrm{RAND}(\,))$$

$$\text{Number of blue pallets} = \mathrm{INT}(3 \times \mathrm{RAND}(\,))$$

$$\text{Number of yellow pallets} = \mathrm{INT}(\mathrm{SQRT}(1 + (9 \times \mathrm{RAND}(\,))))$$

$$\text{Number of green pallets} = \mathrm{INT}(\mathrm{RAND}(\,) + \mathrm{RAND}(\,) + \mathrm{RAND}(\,))$$

The functions cause the number of red pallets to range between 1 and 2, the number of blue pallets to range between 0 and 2, the number of yellow pallets to range between 1 and 3, and the number of green pallets to range between 0 and 2. The file should contain 100 rows of data, with each row defining the number of pallets of each color for one truck.

One worker is employed to unload trucks at the dock. The worker takes an exponentially distributed time with a mean of 10 minutes to unload each pallet, one truck at a time. After unloading, the trucks wait 2 minutes to complete paperwork before they leave the dock.

Create a model to simulate this system. The simulation should end after the paperwork is completed for the last truck.

Tip ☞

Do *not* create loads to represents palettes in the system; you only need to track the total number of pallets on each truck.

What was the average number of trucks waiting to park at the dock?

Chapter 8

Basic Statistical Analysis Using AutoStat

Chapter 8

Basic Statistical Analysis Using AutoStat

When random samples are used as input for a model, a single simulation run may not be representative of the true behavior of the real-world system. Therefore, you could make erroneous inferences about the system if you only make one run. The AutoStat software helps you apply the proper statistical sampling techniques to your model in order to accurately estimate performance under random conditions.

This chapter discusses how to perform basic statistical analysis on a model using the AutoStat software, teaching you how to calculate confidence intervals for several performance metrics.The AutoStat software is covered in two other chapters of this textbook: chapter 10, “Intermediate Statistical Analysis,” and chapter 15, “Warmup Analysis Using AutoStat.” These chapters discuss other types of analyses, such as comparing multiple scenarios and warmup determination.

Why use AutoStat?

The key to a successful simulation study is proper analysis. However, in too many simulation studies, analysts spend most of their time developing the model and not enough time analyzing the simulation results. Sometimes decisions are incorrectly based on just one run of the model.

The AutoStat software helps you to determine which results and alternatives are statistically significant, which means that with high probability the observed results are not caused by random fluctuations but are due to the change you are experimenting with. This chapter discusses using AutoStat to calculate confidence intervals.

Calculating confidence intervals

In chapter 1, you learned how to calculate confidence intervals by hand (see “Statistical confidence” on page 1.25 of the “Principles of Simulation” chapter). This chapter shows you how to use AutoStat to calculate confidence intervals automatically.

Example 8.1: Why confidence intervals are important

Example model 8.1 illustrates why confidence intervals are important when analyzing a random model. In this system, products have an interarrival time that is uniformly distributed between zero and 10 minutes. The products go through two processes: checking and processing.

The checking process is modeled using a resource with a capacity of two. Checking each load takes a time that is uniformly distributed between six and 10 minutes.

The processing step uses a resource with a capacity of eight. The resource can process each load in a time that is uniformly distributed between 25 and 35 minutes.

The source file for example 8.1 is shown below:

```
begin P_init arriving
   while 1=1 do
   begin
      clone 1 load to P_checkers nlt L_job
      wait for u 5,5 min
   end
end

begin P_checkers arriving
   move into Q_checkers /* capacity is infinite */
   use R_checkers for u 8,2 min /* capacity of 2*/
   send to P_processors
end

begin P_processors arriving
   move into Q_processors /* capacity infinite */
   use R_processors for u 30,5 min / *capacity of 8 */
   send to die
end
```

Suppose you simulated this system for 12 hours and then looked at the average time spent in the checking process. Because there is so much randomness in the model (including the arrival times and the service time of each process), the average time reported for the checking process for that run may or may not be an accurate estimate of how the system behaves over a longer period of time.

Now suppose you ran the model 10 times, each time using different random numbers (such runs are called **replications**). The following table shows the results of 10 such replications (this table is also shown in “Statistical confidence” on page 1.25 of the “Principles of Simulation” chapter):

Replication Number (i)	Average Time in the Checking Process
1	752.23
2	785.49
3	645.13
4	639.96
5	610.13
6	661.42
7	645.28
8	606.32
9	677.74
10	584.53

Average time in the checking process for 10 replications

Each of these times is a valid average time for the checking process. But the range between the lowest and highest value is 200.96 seconds. So what is the “correct” value, and how do you make decisions based on these numbers?

Using the average value from a single replication can be misleading. It’s called a point estimate, and it can be very different than the true mean of the process. The appropriate action in this case is to report a confidence interval, not just a single number.

How AutoStat calculates confidence intervals

The AutoStat software uses the **replication/deletion** technique for computing confidence intervals. The replication/deletion method strives to use steady-state data in the formation of point estimates and confidence intervals for the various responses, which is accomplished by obtaining the average level of the response for each replication *after* a warmup period (the deletion of warmup periods for non-terminating systems is also discussed in “Deletion” on page 1.29 of the “Principles of Simulation” chapter). The averages obtained after each warmup period are independent and are approximately normally distributed random numbers. Thus, they can be used to construct a confidence interval for the steady-state mean value of the response.

To generate the average times shown in the table above, the model was run using an 8-hour warmup followed by a 24-hour snap length. Thus, the simulation was run for 8 hours, at which time statistics were *reset* (all statistics, except for Cur(rent) values, were set to zero), to remove the warmup bias. The simulation was run for 24 hours, during which statistics were gathered. This procedure was followed 10 times, each using different random numbers.

This chapter focuses on setting up AutoStat to determine the confidence interval for the average time loads spent in the checking process (P_checkers).

Performing statistical analysis with AutoStat

The following steps outline the process for conducting an analysis with AutoStat:

Step 1 ***Open*** a model in AutoStat.

Step 2 ***Define*** an analysis (for example, a single scenario analysis to determine confidence intervals).

Step 3 ***Make*** the runs.

Step 4 ***Define*** the responses (statistics that you want to measure).

Step 5 ***Display*** the results.

The rest of the chapter walks you through these steps to determine the confidence intervals for example model 8.1.

Opening a model in AutoStat

To use the AutoStat software, you must open a model that has been compiled in AutoMod.

To open example model 8.1 for use in the AutoStat software:

Step 1 ***Copy*** the *base* version of example model 8.1 to a new directory.

Step 2 ***Import*** examp81.

Step 3 From the Model menu, ***select*** Build. The model compiles.

Step 4 From the Model menu, ***select*** Run AutoStat. AutoStat opens and the AutoStat Setup wizard opens.

Tip ☞ If you have already compiled a model, you can open AutoStat from the Start menu and select Open from the File menu, then navigate to the <modelname>.mod file in the <modelname>.dir directory to open the model.

The first time you open a model in AutoStat, you must use the AutoStat Setup wizard to set several parameters for your analysis.

Using the AutoStat Setup wizard

Whenever you open a model for the first time in the AutoStat software, the AutoStat Setup wizard asks you to:

- Indicate whether the model contains randomness.
- Determine a time limit to stop models that are potentially in an infinite loop.
- Estimate the model’s warmup length.
- Define how long to run the model to collect statistics.

The information in the wizard is used to define several model properties. The following sections explain the questions that the wizard is asking.

Help While using the AutoStat software, click Help if you want more information about a particular screen.

Step 1 In the AutoStat Setup Wizard window, ***click*** Next to advance to the first question in the wizard.

Is the model random or deterministic?

As discussed earlier in this chapter, when a simulation model contains random input (such as the random arrival and service times in example model 8.1), the statistics from one run may not be representative of long-term average system output. Therefore, you need to perform multiple replications (runs using different random numbers) to get an accurate understanding of the system's behavior, including meaningful confidence intervals.

Note

A **deterministic** model contains no random input and therefore no variability in output from run to run, so only a single run of each scenario is necessary. *Very few simulation models are deterministic.*

Step 1 ***Select*** Model is random.

Step 2 ***Click*** Next.

Do you want to stop runs that may be in an infinite loop?

An infinite loop causes a model to repeat the same section of logic so that the simulation never ends (for more information about infinite loops, see "Avoiding infinite loops" on page 7.17 of the "Advanced Process System Features" chapter). If you are using AutoStat to make runs, the software can automatically stop a run that seems to be taking longer than you expected.

Example model 8.1 runs very quickly (in about 1 or 2 seconds) if it is working correctly. If a run takes substantially longer than that (for example, 30 seconds), it might indicate that there is an infinite loop in the model and that the run should be cancelled.

To set an infinite loop time of 30 seconds:

Step 1 ***Select*** Yes to indicate that you want to set up an infinite loop time limit.

Step 2 ***Click*** Next.

Step 3 ***Type*** "30" in the Maximum run time text box and ***select*** seconds from the drop-down list.

Step 4 ***Click*** Next.

Does the model require time to warm up?

If a model does not reflect the state of the system being modeled at the beginning of the simulation, it needs to warm up, or "reach steady state," before you gather statistics.

Some systems start "empty," such as service centers. In these systems, there are no customers left over from the day before, and new customers cannot enter the system until the center opens in the morning.

Systems such as the factory being modeled in example 8.1, however, are usually full of products that are spread throughout the factory and that are in various stages of being manufactured. In the example model, loads do not get created until the simulation starts, so it will take some time before the system is primed with jobs. Therefore, you want to set up a warmup time and discard statistics gathered during that time.

Step 1 ***Select*** Yes to indicate that the model requires a warmup time.

Step 2 ***Click*** Next.

What is the estimated warmup time?

AutoStat can perform a warmup determination analysis, which you will learn about in chapter 15, “Warmup Analysis Using AutoStat.” For now, make a rough estimate without the help of AutoStat and say that the simulation takes 8 hours to warm up.

To define the warmup time:

Step 1 ***Type*** “8” in the warmup estimate text box and ***select*** hours from the drop-down list.

Step 2 ***Click*** Next.

Do you want to create the warmup analysis?

Warmup analyses are discussed in chapter 15, “Warmup Analysis Using AutoStat.” For this chapter, assume that you have estimated the warmup correctly and do not need the analysis.

Step 1 ***Select*** Do not create analysis.

Step 2 ***Click*** Next.

What is the snap length for collecting statistics?

After the warmup time, the statistics are going to be reset and then collected for some amount of time. The statistics-gathering time is called the **snap length**. The length of the snap varies depending on whether the system being modeled is a terminating or a non-terminating system (discussed in“Terminating versus non-terminating systems” on page 1.27 of the “Principles of Simulation” chapter). In a terminating system, the snap length is equal to the length of the system’s operation. For example, if you were simulating a bank that is open from 9:00 A.M. to 5:00 P.M., the snap length would be 8 hours.

In a non-terminating system, the snap needs to be long enough to collect meaningful, representative data from the system. The time required varies from system to system and is best determined by someone familiar with the system’s operation.

Example model 8.1 currently completes about 300 loads in 24 hours, which is meaningful enough to generate confidence intervals for this small model.

To define the snap length:

Step 1 ***Type*** “24” in the Snap length text box and ***select*** hours from the drop-down list.

Note The run control in AutoStat always overrides the run control defined in AutoMod.

Step 2 ***Click*** Next. The last screen of the wizard opens.

Step 3 ***Click*** Finish.

The information in the wizard is used to set up the model properties. You can change any of these settings, such as the snap length, warmup length, and so on, at any time by editing the model properties (discussed next).

Step 4 From the File menu, ***select*** Save to save the properties.

Editing model properties

If you need to change a model property after you have run the Model Setup wizard, edit the model properties.

To edit the model properties:

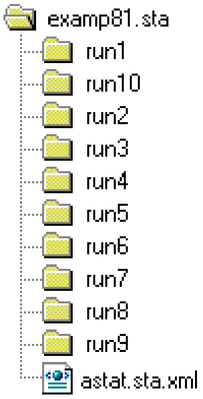
Step 1 From the Properties menu, ***select*** Edit Model Properties. The Model Properties window opens.

When conducting your own analyses, you may want to change some of these values. For this example model, however, do not change any of the properties.

Step 2 ***Click*** Cancel to close the Model Properties window.

The AutoStat file system

When you open a model and set it up in AutoStat, several directories and files are created.



```
examp81.sta
  run1
  run10
  run2
  run3
  run4
  run5
  run6
  run7
  run8
  run9
  astat.sta.xml
```

AutoStat file system

The main AutoStat directory is the .sta directory. The **.sta** directory contains all the AutoStat information for a model. The .sta directory contains a file called **astat.sta.xml**, which contains all the information about the model properties you set using the wizard, as well as information for analyses (which you will learn how to define in this chapter).

Once you have made runs for your analyses, **numbered run** directories are created, which contain message files, reports, and information about each run. For a complete description of the AutoMod file system, see “An AutoMod model’s file hierarchy” on page 3.4 of the “AutoMod Concepts” chapter.

Defining a single scenario analysis

As mentioned earlier, AutoStat can conduct several types of analyses. In this chapter, you will learn how to conduct a **single scenario** analysis, in which you can run your model “as is” (without any changes) in order to compute confidence intervals and other statistics. Other analyses will be discussed in chapter 10, “Intermediate Statistical Analysis,” and chapter 15, “Warmup Analysis Using AutoStat.”

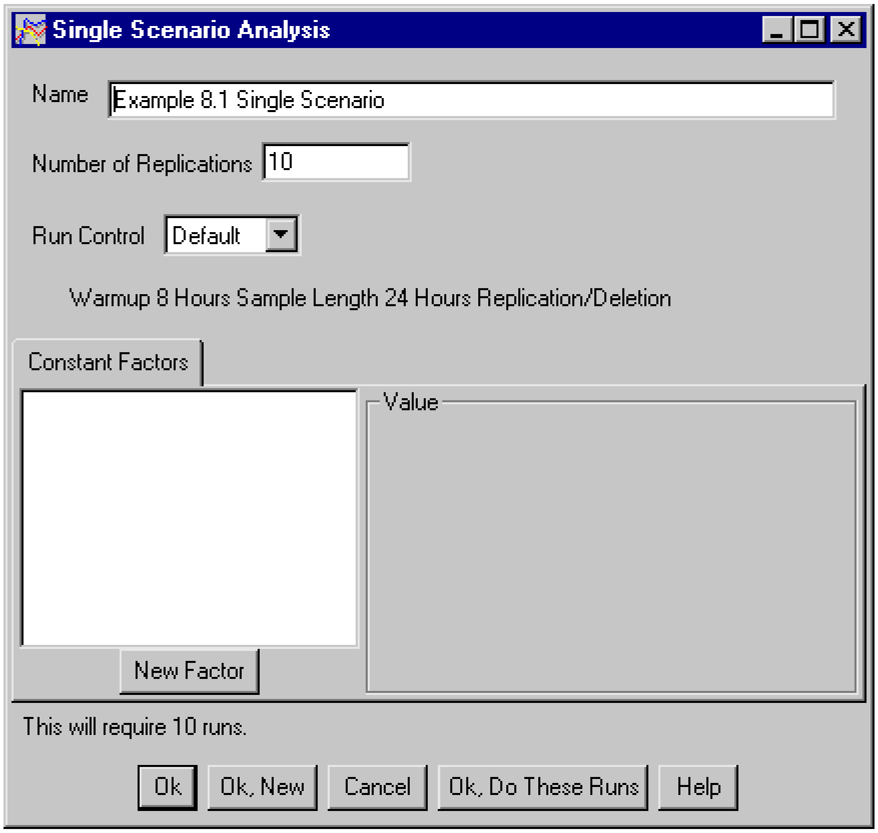
To define a single scenario analysis:

Step 1 From the Create New Analysis of Type drop-down list, ***select*** Single Scenario.

Step 2 ***Click*** New. The Single Scenario Analysis window opens.

Step 3 ***Name*** the analysis “Example 8.1 Single Scenario.”

Step 4 ***Type*** “10” in the Number of Replications text box and ***press*** Tab. AutoStat will make 10 runs, with each run using different random numbers for each random event in the model.



Single Scenario Analysis

Name: Example 8.1 Single Scenario

Number of Replications: 10

Run Control: Default

Warmup 8 Hours Sample Length 24 Hours Replication/Deletion

This will require 10 runs.

Single Scenario Analysis

For confidence intervals, you should use *at least* three to five replications. The greater the number of replications or the longer the snaps, the narrower the confidence intervals.

For example model 8.1, you want to use the default run control, which you set up using the wizard, so do not make any other changes in this window.

You have defined the single scenario analysis. Now you are ready to make the runs.

Making runs

There are several ways to make runs for an analysis in AutoStat. In example 8.1, you can make the runs directly from the Single Scenario Analysis window.

Step 1 ***Click*** OK, Do These Runs. AutoStat makes 10 runs, recording the statistics.

You can also make runs using either the Execution menu or the Runs tab using the following options:

Do All Runs Makes all runs that have been defined for all analyses.

Do Some Runs You can select the analysis for which you want to make runs.

Do Runs Until Defines a time to stop making runs. For example, make runs until tomorrow at 8:00 A.M.

Parallel Execution Makes runs on more than one computer (not discussed in this textbook).

While the runs are being made, or after they are finished, you can define responses, which are the statistics you are interested in analyzing.

Defining responses

A **response** is a statistic in your model that you want to study in an analysis. For example, you might be interested in the utilization of a resource, the average time spent waiting for a tool, the time in system, and so on.

There are three types of responses:

AutoMod A standard statistic from the report file.

User A statistic from a user-defined report (not discussed in this textbook).

Combination The combination of two or more responses, such as the time spent in several processes, or the average utilization of a group of equipment. An example is shown in “Defining a combination response” on page 8.18.

Tip You can define responses at any time (either before *or* after making runs). You can also define responses from most output windows.

Defining an AutoMod response

In the example model, you want to determine confidence intervals for the average time that loads spend in the checker process. Therefore, you need to define a response to track loads’ average time in that process.

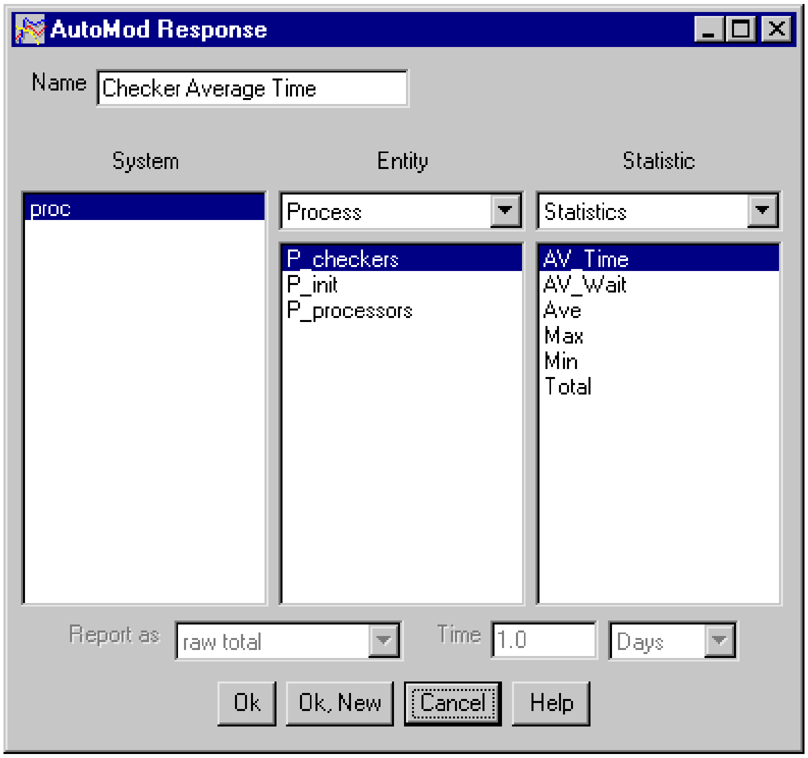
To define an AutoMod response:

Step 1 ***Click*** the Responses tab.

Step 2 ***Click*** New to create a new response of type AutoMod Response. The AutoMod Response window opens.

Step 3 ***Name*** the response “Checker Average Time”.

Step 4 ***Select*** the system, entity, and statistic you want to define as a response. By default, the process P_checkers and the statistic AV_Time are already selected.



AutoMod Response

Name: Checker Average Time

System: proc

Entity: Process — P_checkers

Statistic: Statistics — AV_Time

Report as: raw total Time: 1.0 Days

AutoMod Response window

Step 5 ***Click*** OK. The response name appears in the Defined responses list.

Displaying the results

Each analysis has several ways of displaying output. The output is calculated for all defined responses.

For single scenarios, the types of output are:

Summary Statistics A table of statistics for each response, including the average, standard deviation, minimum and maximum values, and other information.

Bar Graph A graph of a response's values.

Confidence Intervals The confidence interval for each response by confidence level (90 percent, 95 percent, and so on).

Runs Used A list of the runs that are being used for this analysis (useful if you have more than one analysis defined).

Run Results A table of each response by run.

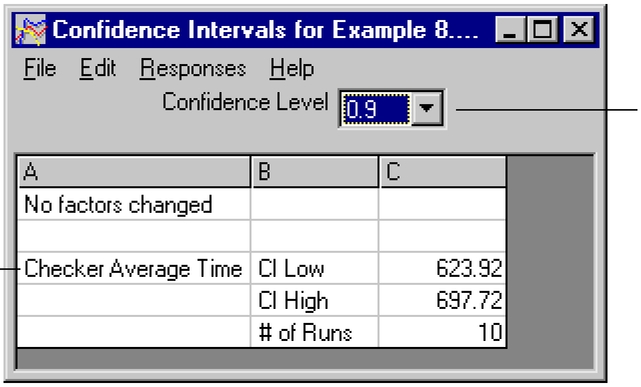
For example 8.1, you will look at confidence intervals and summary statistics.

Viewing confidence intervals

To view confidence intervals:

Step 1 From the Analysis tab, ***click*** the plus (+) sign to expand the list of output options.

Step 2 ***Double-click*** Confidence Intervals. The Confidence Intervals window opens.



Confidence Level 0.9

A	B	C
No factors changed		
Checker Average Time	CI Low	623.92
	CI High	697.72
	# of Runs	10

Change the confidence level using the Confidence Level drop-down list

Response name

90 percent confidence level for Checker Average Time

By default, the confidence intervals displayed are for a 90 percent confidence level. The window displays the response name, the low and high values in the interval, and how many runs are being used to generate the interval.

You can adjust the confidence level using the drop-down list. The following table shows the 95 percent and 99 percent intervals for the Checker Average Time response:

Measure	95%	99%
CI Low (seconds)	615.284	595.403
CI High (seconds)	706.356	726.237
# of Runs	10	10

Table of 95 and 99 percent confidence levels for Checker Average Time

Step 3 ***Close*** the Confidence Intervals window.

Narrowing the confidence interval

The narrower the range of a confidence interval, the more accurate the estimate. To decrease the width of a confidence interval by approximately half, you must increase the sample size by four. You can increase the sample size one of two ways:

- Making more runs (4 times the number of runs)
- Making longer runs (increasing the run length by a factor of 4)

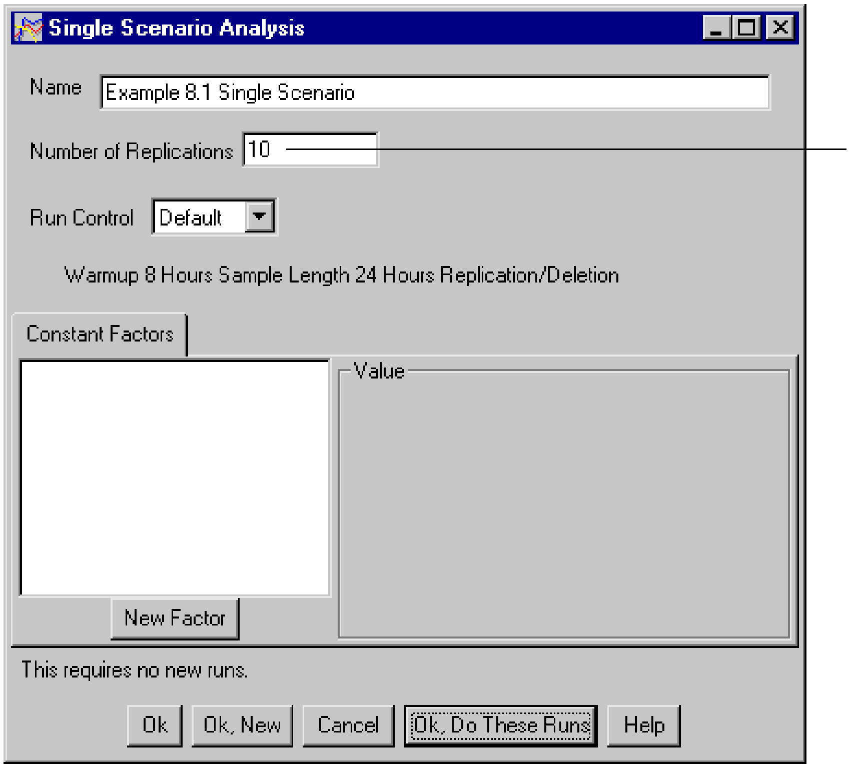
Either of these methods provide more information to AutoStat, allowing it to narrow the range of the interval.

Making more runs

It is possible to make additional runs for an analysis at any time.

To make additional runs for an analysis:

1. Edit the analysis.
2. Edit the number of replications.
3. Make the additional runs.



Name Example 8.1 Single Scenario

Number of Replications 10

Run Control Default

Warmup 8 Hours Sample Length 24 Hours Replication/Deletion

This requires no new runs.

To make additional runs, increase the number of replications here

Editing the number of replications in the Single Scenario Analysis window

Making longer runs

You can increase the length of runs used for an analysis in two ways:

- Edit the model properties and change the default sample time.
- Edit the analysis and create a custom run control.

Both approaches require you to redo existing runs so that your analysis is based on the new information.

Changing the default sample time

Editing a model's properties changes the sample time for all analyses that use the default run control (for information about editing model properties, see "Editing model properties" on page 8.9).

For example model 8.1, you are going to define a custom run control, not change the default sample time, so do not edit the model's properties.

Defining a custom run control

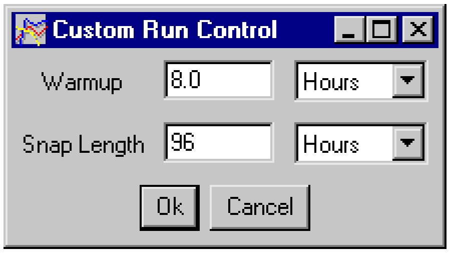
For example model 8.1, you want to make the runs for the analysis four times as long by defining a custom run control.

To define a custom run control for an analysis:

Step 1 From the Analyses tab, ***select*** Example 8.1 Single Scenario Analysis and ***click*** Edit. The Single Scenario Analysis window opens.

Step 2 From the Run Control drop-down list, ***select*** Custom... The Custom Run Control window opens.

Step 3 You want to lengthen the snap length by a factor of 4, so ***change*** the Snap Length from 24 to 96 hours.



Warmup: 8.0 Hours

Snap Length: 96 Hours

Custom run control

Step 4 ***Click*** OK to close the Custom Run Control window.

Step 5 ***Click*** OK, Do These Runs to make the new runs. Ten new runs are made using the new run control (the 10 runs using a 24 hour snap length are still saved, as well).

Step 6 ***Click*** OK when the runs have finished.

Step 7 ***Double-click*** Confidence Intervals to display the confidence intervals.

Step 8 ***Change*** the confidence level to 99 percent.

The following table shows the 99 percent intervals for the Checker Average Time response for 24 hour and 96 hour snap lengths:

Measure	99%	99%
CI Low (seconds)	595.403	600.169
CI High (seconds)	726.237	641.451
Snap Length	24 hours	96 hours

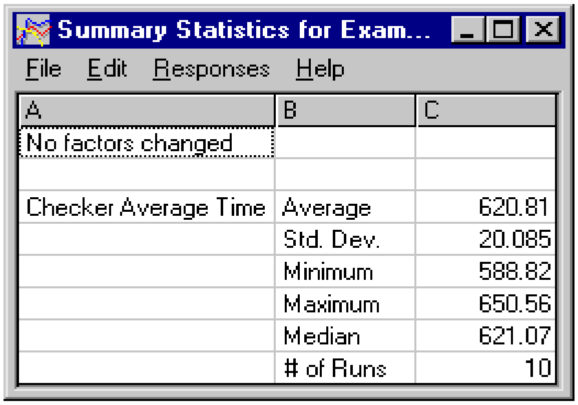
Table of 99 percent confidence levels for 24-hour and 96-hour snap lengths

The original interval has a range of 130.834 seconds, half of which is 65.417. The new snap length of 96 hours has an interval of only 41.282, so the “four times as long” guideline has been successful for this analysis.

Viewing summary statistics

To view summary statistics for the single scenario analysis:

Step 1 From the Analysis tab, ***expand*** the Example 8.1 Single Scenario analysis and ***double-click*** Summary Statistics.



A	B	C
No factors changed		
Checker Average Time	Average	620.81
	Std. Dev.	20.085
	Minimum	588.82
	Maximum	650.56
	Median	621.07
	# of Runs	10

Summary statistics

This table shows you the average, standard deviation, minimum, maximum, and median values of the average time response for the 10 runs.

Defining a combination response

You can combine two or more responses together into a single response using a combination response. Combination responses are useful when you want to combine statistics, such as averaging several machines' utilizations into one statistic for an equipment group, or summing several different WIP levels.

In example model 8.1, you have already defined a response for the average time loads spend in the checking process. If you create a response for the average time in the processor process and then add the two together in a combination response, the combination response is the average time that loads spend in the whole system.

To define a combination response for the time in system:

Step 1 From the Responses tab, ***define*** an AutoMod response, named "Processor Average Time," for the average time loads spend in P_processors. ***Click*** OK.

Step 2 From the New Response of Type drop-down list, ***select*** Combination Response and ***click*** New. The Combination Response window opens.

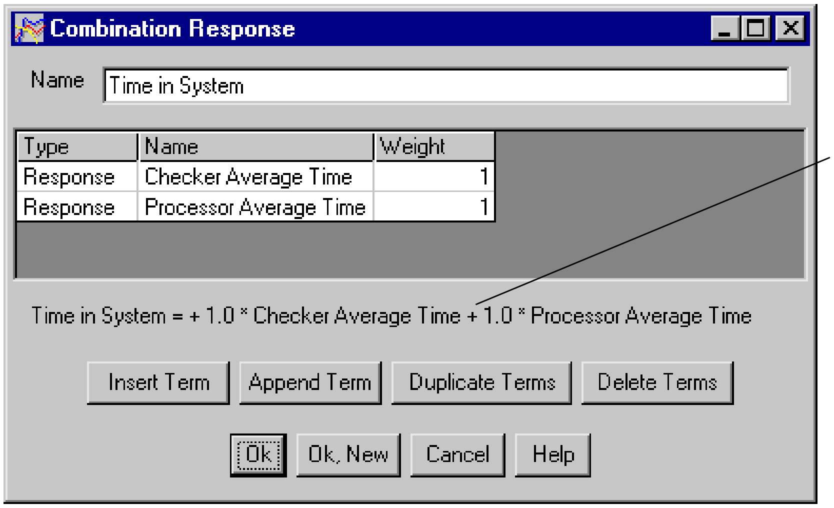
Step 3 ***Name*** the combined response "Time in System."

Step 4 ***Click*** Append Term to add a duplicate row below the Checker Average Time response.

Step 5 ***Double-click*** the Name in the new row.

Step 6 ***Select*** Processor Average Time.

The combination response is now a sum of the two AutoMod factors, as shown by the formula in the window.



Name: Time in System

Type	Name	Weight
Response	Checker Average Time	1
Response	Processor Average Time	1

Time in System = + 1.0 * Checker Average Time + 1.0 * Processor Average Time

The relationship of the terms is shown in the formula

Viewing the relationship between terms in a combination response

Step 7 ***Click*** OK to close the Combination Response window.

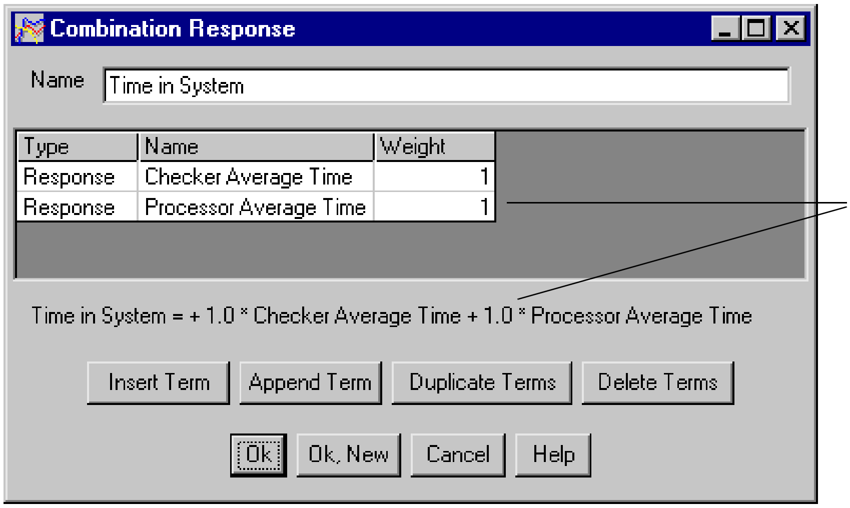
Step 8 ***Display*** the confidence intervals and summary statistics for the analysis. Notice that the two new responses, Processor Average Time and Time in System, are now being shown in addition to the original response, Checker Average Time.

Weighting terms in a combination response

When using a combination response, you can define a **weight** (or relative importance) for each of the terms being used to calculate the combined response. This is often useful when analyzing cost versus payback situations, such as whether the cost of a new piece of equipment is paid for by an increase in throughput. (Using weights in this context is discussed in detail in chapter 10, “Intermediate Statistical Analysis.”)

You can also use the weight value to convert the statistics being measured into different units. For example, you can convert time units from seconds to minutes, or convert one monetary unit to another.

In example model 8.1, the terms in the Time in System response are currently calculated in seconds (all standard AutoMod statistics are reported in seconds). By default, the terms are each given a weight of one, which means the terms have equal value and are being used in their default units.



Name: Time in System

Type	Name	Weight
Response	Checker Average Time	1
Response	Processor Average Time	1

Time in System = + 1.0 * Checker Average Time + 1.0 * Processor Average Time

The weight of a term determines its relative importance in the formula

Weighting terms in a combination response

Suppose you wanted to report the Time in System in minutes instead of seconds.

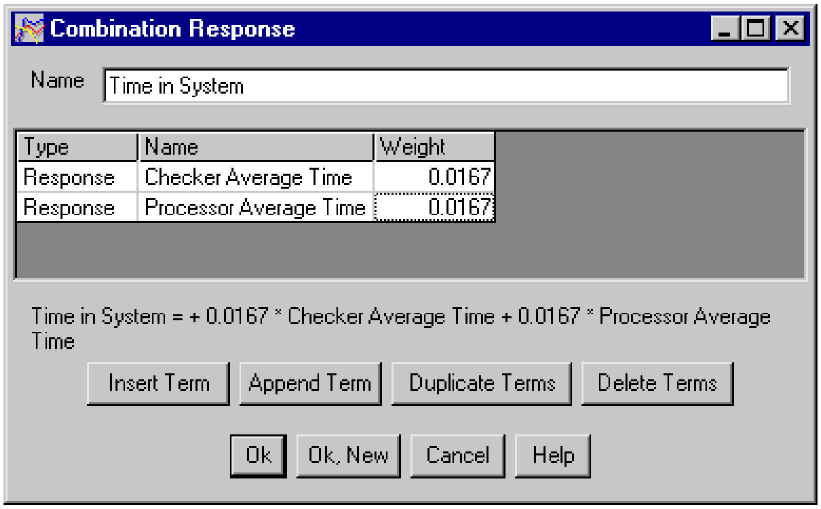
To report the Time in System in minutes:

Step 1 From the Responses tab, ***select*** Time in System in the Defined Responses list and ***click*** Edit. The Combination Response window opens.

Step 2 ***Double-click*** the Weight for Checker Average Time.

Step 3 ***Enter*** a Weight of 0.0167 (1 minute/60 seconds) as a multiplier to convert seconds to minutes.

Step 4 ***Enter*** 0.0167 as the weight for Processor Average Time, as shown below:



Combination Response

Name: Time in System

Type	Name	Weight
Response	Checker Average Time	0.0167
Response	Processor Average Time	0.0167

Time in System = + 0.0167 * Checker Average Time + 0.0167 * Processor Average Time

Converting seconds to minutes using the Weight column

Step 5 ***Click*** OK to close the Combination Response window.

Step 6 ***View*** the confidence intervals and summary statistics for the analysis. Notice that the values for the Time in System are now in minutes instead of seconds.

Tip ☞ You could rename the analysis to show the units being displayed. For example, you could call this analysis “Time in System (minutes).” Then you would know when viewing output that this response’s units are different from the other responses being displayed.

Summary

This chapter introduces how to conduct statistical analysis using the AutoStat software. AutoStat is a powerful statistical analysis tool, and should be used on every project to ensure you draw sound conclusions from your simulation efforts.

In this chapter, you learned how to use a single scenario analysis to determine confidence intervals for several responses. You learned how to view various types of output, including summary statistics and confidence intervals. You learned how to make a confidence interval range narrower by making more runs or making longer runs.

Finally, you learned how to combine several responses together into a combination response, using the example Time in System. You learned how to convert the units of responses using the Weight value in the Combination Response window.

Exercises

Note In the following exercises, round your answers to the nearest hundredth.

Exercise 8.1

Before beginning work on the problems in this exercise, complete the following steps:

Step 1 ***Copy*** the *base* version of example model 8.1 to a new directory.

Step 2 ***Open*** the copied model in AutoMod and ***change*** the amount of time that loads use resource R_checkers to “e 8 min”.

Step 3 ***Export*** and ***build*** the model.

Step 4 ***Open*** the model in AutoStat and ***use*** the Setup wizard to define the following properties:

- Model is random
- Check for infinite loops and stop runs that are longer than 2 minutes
- Define an 8 hour warmup
- Define a snap length of 40 hours

Use the model to complete the following problems:

a) Create a single-scenario analysis and run the model for 20 replications. Find the 90 percent, 95 percent, and 99 percent confidence intervals for the average time in seconds that loads spend in process P_checkers.

b) Use a combination response to find the 95 percent confidence interval for the average time *in minutes* that loads spend in the processes P_checkers and P_processors.

c) Find the 95 percent confidence interval for the average time in seconds that loads spend in process P_checkers when the number of replications is 20, 40, and 80. How do you explain the difference?

d) Find the difference between the minimum and maximum time in seconds that loads spend in process P_checkers when the number of replications is 100.

e) Find the median time in seconds that loads spend in process P_checkers when the number of replications in 100.

Exercise 8.2

Copy the *final* version of example model 7.1 to a new directory. Open the copied model in AutoStat and use the Setup wizard to define the following model properties:

- Model is random
- Check for infinite loops and stop runs that are longer than 2 minutes
- Define a 4 hour warmup
- Define a snap length of 24 hours

Record the summary statistics and find the 95 percent confidence interval for the average time in seconds that loads spend processing, *not* including the time that loads spend in the inspection and finishing processes. Perform the analysis with 10 replications.

Exercise 8.3

Copy example model 6.1 to a new directory. Open the copied model in AutoStat and use the Setup wizard to define the following model properties:

- Model is random
- Check for infinite loops and stop runs that are longer than 2 minutes
- Define a 4 hour warmup
- Define a snap length of 1 day

Record the summary statistics and find the 95 percent confidence interval for the average number of loads in P_process. Perform the analyses with 10 replications and the following snap lengths:

a) Each replication is based on a 1-day simulation.
b) Each replication is based on a 10-day simulation.

Exercise 8.4

Copy example model 6.1 to a new directory. Open the copied model in AutoStat and use the Setup wizard to define the following model properties:

- Model is random
- Check for infinite loops and stop runs that are longer than 2 minutes
- Define a 4 hour warmup
- Define a snap length of 10 days

Find the confidence interval for the average number of loads in P_process for the following levels of confidence:

a) 90 percent
b) 95 percent
c) 99 percent

Perform the analysis with 5 replications.

Exercise 8.5

Create a new model in which loads are generated with an interarrival time that is exponentially distributed with a mean of 12 seconds. Loads first move into an infinite-capacity queue where they wait to use a single-capacity resource. The resource has its own queue for processing. The resource's processing time is exponentially distributed with a mean of 8 seconds. After processing, loads leave the system.

Find the mean (average), standard deviation, and 95 percent confidence interval for the average time in seconds that loads spend in the system. Perform the analysis with no warmup, no infinite loop checks, 5 replications, and the following snap lengths:

a) 1 hour
b) 4 hours
c) 16 hours

How do you explain the difference in the confidence intervals when the snap length changes?

Chapter 9

Modeling Complex Conveyor Systems

Chapter 9

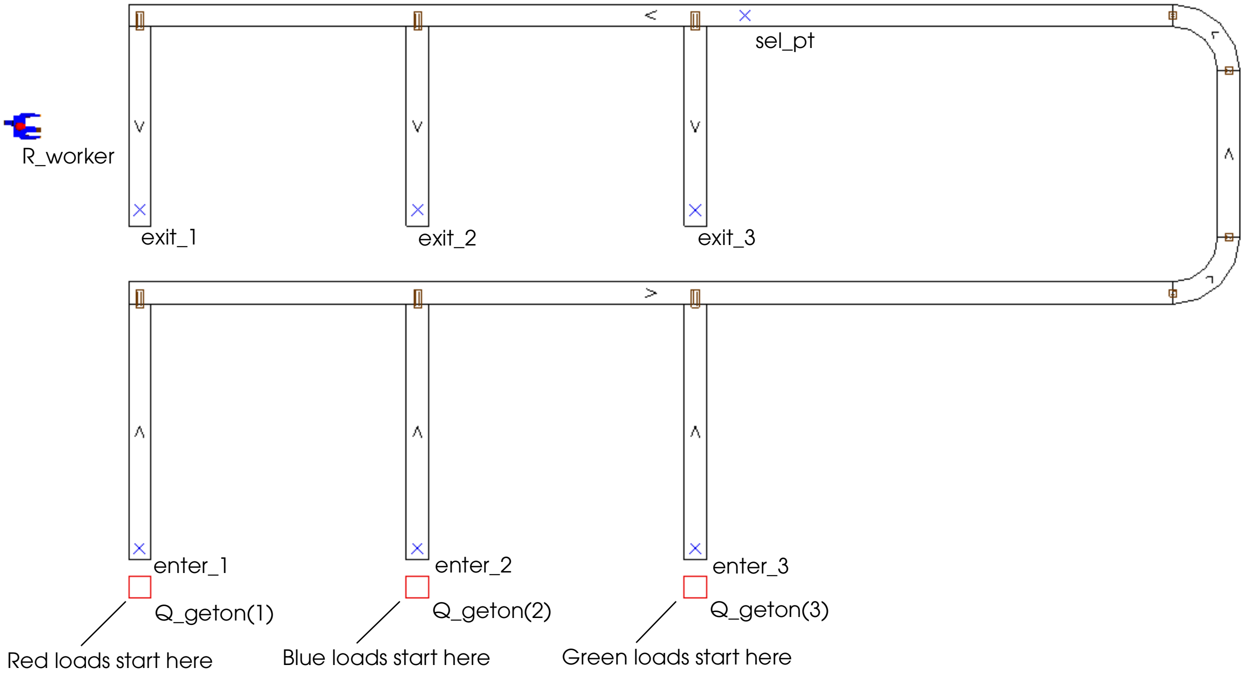
Modeling Complex Conveyor Systems

In chapter 6, “Introduction to Conveyors,” you learned how to draw basic conveyor systems. In this chapter, you will learn how to transport and sort multiple load types in a conveyor system. You will also learn how to edit conveyor system entities to model many different types of conveyors, such as belt, chain, and roller conveyors.

This chapter discusses transfers and the effect they have on load orientation. The chapter also introduces motors, which drive conveyor movement. You will learn how to model motor failures, as well as use motors to simulate slugging and indexing conveyor sections.

Example 9.1: Transporting multiple load types on a conveyor

Example model 9.1 is an example system that transports multiple load types. Consider the conveyor layout shown below:



Layout of example model 9.1

Red, blue, and green loads are transported in the conveyor system. Loads arrive in sets consisting of between one and three loads that are all of the same type. The quantity of loads in each set is randomly determined as shown in the table below:

Load type	Chance of one load in set	Chance of two loads in set	Chance of three loads in set
L_red	35 percent	40 percent	25 percent
L_blue	45 percent	40 percent	15 percent
L_green	40 percent	40 percent	20 percent

The interarrival times for sets of loads of a given type are exponentially distributed with a mean that is shown in the table below:

Load type	Mean
L_red	10 minutes
L_blue	8 minutes
L_green	10 minutes

Upon arrival, loads in a set are immediately sorted by type into one of three infinite-capacity entrance queues (Q_geton(1), Q_geton(2), or Q_geton(3)). The loads get on the conveyor at the entrance lane in front of their assigned queue. The loads then travel on the conveyor to one of the three ramped exit lanes, which are alternately selected based on the order in which loads arrive at station “sel_pt” (the first load to arrive exits at station “exit_3,” the second load to arrive exits at station “exit_2,” and so on.).

After arriving at an exit station, each load is processed by a worker for an amount of time that is exponentially distributed with a mean of 1.6 minutes. The completed loads then leave the system.

To become familiar with the movement of loads in the system:

Step 1 ***Import*** and ***run*** a copy of example model 9.1.

Step 2 When you are ready to continue, ***select*** Edit Model from the Control menu to return to the build environment.

You are now ready to look at the example model logic to see how the system is simulated.

Assigning load creation frequency using load attributes

In example model 9.1, the model initialization function creates one load of type L_dummy and sends the load to the process P_init. The P_init arriving procedure creates three loads that generate sets of red, blue, and green loads throughout the simulation.

The P_init arriving procedure is shown below:

```
begin P_init arriving
   set V_index to 1
   while V_index <= 3 do begin
      set A_type to V_index
      set A_freq1 to nextof(35,45,40)
      set A_freq2 to nextof(40,40,40)
      set A_freq3 to nextof(25,15,20)
      set A_time to nextof(10,8,10) /*time between arrivals*/
      clone 1 load to P_create nlt nextof(L_red, L_blue, L_green)
      inc V_index by 1
   end
end
```

The P_init arriving procedure includes a `while...do` loop that executes three times at the beginning of the simulation. Each time the loop executes, load attribute values are initialized and a load is cloned to the P_create process. The load attribute values define a number to represent each load type, three frequency values that determine how often multiple loads are created in a set, and a time value that represents the time between set arrivals in minutes.

The first time the loop is executed, the value of the attribute A_type is set to 1. The values of the three frequency attributes (A_freq1, A_freq2, and A_freq3) are set to 35, 40, and 25, respectively. These values correspond to the frequency with which multiple loads of type L_red are created in a set (see “Example 9.1: Transporting multiple load types on a conveyor” on page 9.4). The value of the attribute A_time is set to 10, which corresponds to the time between arrivals of each set of L_red loads. A load of type L_red is then cloned to the process P_create. The second and third times the loop is executed, values are initialized and loads of type L_blue and L_green are cloned, respectively.

Recall that cloned loads retain the attribute values of the original load; the values that were initialized in the P_init arriving procedure are used by the three cloned loads to create load sets in the P_create process, as shown below:

```
begin P_create arriving
   while 1=1 do begin
      clone oneof(A_freq1:1,A_freq2:2,A_freq3:3) loads to P_process
      wait for e (A_time * 60) /*time is converted to seconds*/
   end
end
```

The three cloned loads simultaneously execute the `while...do` loop in the P_create arriving procedure to clone sets of loads throughout the simulation. Each time the loop is executed, a new set of loads is cloned and sent to the process P_process.

The number of loads that are cloned in each set is determined using the `oneof` distribution and the frequency values that were set in the P_init arriving procedure. The time delay is defined using the value of the attribute A_time (also set in the P_Init arriving procedure). Because the time value represents minutes, but is stored in an attribute of type Time (which expresses values in seconds) each value is multiplied by 60 in the `wait` action to convert the value from minutes to seconds.

Aligning conveyor and process system entities using load attributes

According to the description of example 9.1, loads are sorted by type into one of three queues. Loads then get on the conveyor at the entrance lane in front of each queue and travel to one of three alternately selected exit stations. In example model 9.1, the logic that sorts and moves loads through the conveyor system is defined in the P_process arriving procedure. The procedure aligns the process system queues with the conveyor entrance stations using the value of the load attribute A_type, which is 1 for loads of type L_red, 2 for loads of type L_blue, and 3 for loads of type L_green. The value of the attribute A_type is then used to align queues and conveyor stations in the simulation, as shown below:

```
begin P_process arriving
   move into Q_geton(A_type)
   move into conv:enter_(A_type)
   travel to conv:sel_pt
   set A_index to nextof(3,2,1)
   travel to conv:exit_(A_index)
   use R_worker for e 1.6 min
   send to die
end
```

The entrance queues at the beginning of each conveyor section are modeled as an array. Each load that executes the P_process arriving procedure uses the value of the attribute A_type to determine which queue to move into, as shown below:

```
   move into Q_geton(A_type)
```

Loads of type L_red move into Q_geton(1), loads of type L_blue move into Q_geton(2), and loads of type L_green move into Q_geton(3).

When referring to movement system locations in the model logic, you can replace the numeric portion of the location name with a numeric variable or load attribute, similar to the way that you can point to arrayed queues. For example, notice that each of the entrance conveyor stations are defined using the same alphabetic name but with a different number (enter_1, enter_2, and enter_3). In the model logic, the numbers can be replaced with the value of the load attribute A_type, as shown below:

```
   move into conv:enter_(A_type)
```

Consequently, loads of type L_red move into station enter_1, loads of type L_blue move into station enter_2, and loads of type L_green move into station enter_3.

Now, look at the logic that sends loads to their destinations:

```
travel to conv:sel_pt
set A_index to nextof(3,2,1)
travel to conv:exit_(A_index)
```

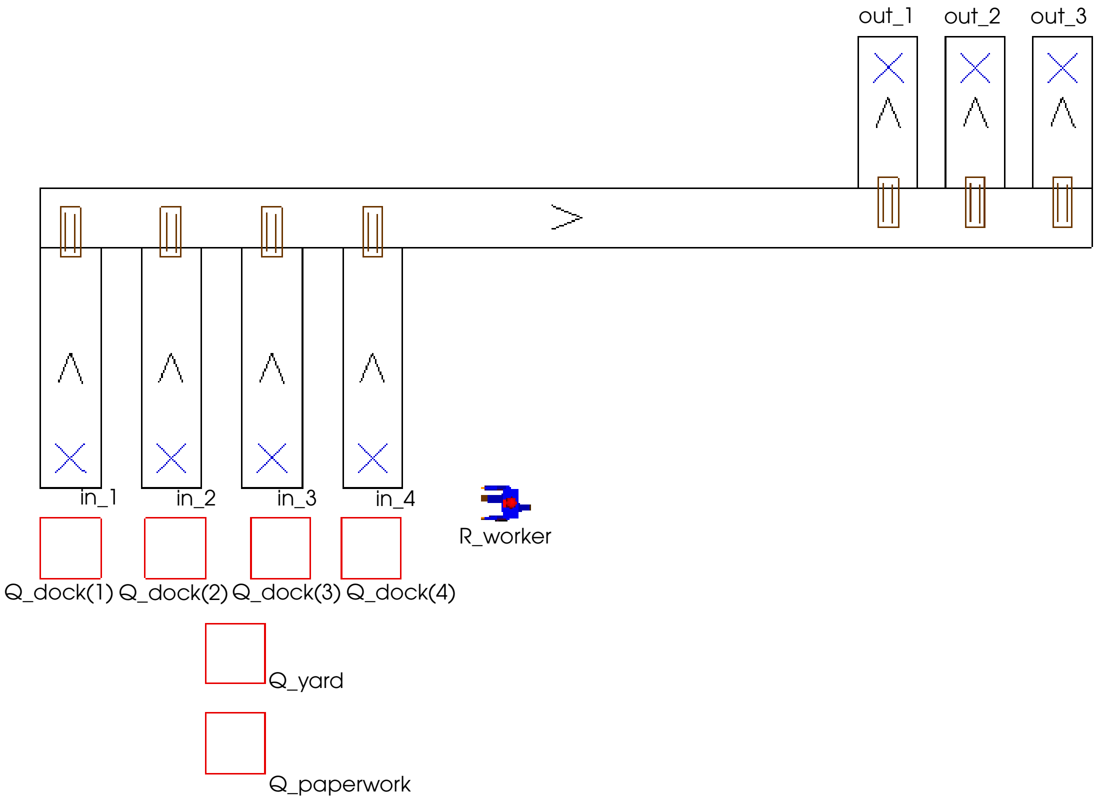
All loads travel to a selection station before determining their final destination. Can you see why? (Refer to “Example 9.1: Transporting multiple load types on a conveyor” on page 9.4 for an illustration of the conveyor layout.) According to the example description, loads need to travel to each of the exit stations in alternating order. However, the sequence of loads on the conveyor is not determined until all loads have passed the entrance lane for loads of type L_green (green loads can get on before, or in the middle of loads of other types). After all loads have passed the entrance lane for loads of type L_green, their sequence is fixed until they reach the exit lanes. Consequently, the selection station (sel_pt) can be located anywhere after the entrance lane for loads of type L_green but before the first exit lane. In example model 9.1, the station is located immediately before the first exit lane.

After traveling to the selection point, the value of the load attribute A_index is alternately set to 3, 2, or 1, which is used in the `travel` action to send loads to the correct destination. Once loads arrive at their destination, they use the resource R_worker (while on the conveyor) for an amount of time that is exponentially distributed with a mean of 1.6 minutes and are removed from the simulation.

In example model 9.1, loads are sorted in queues and then sent on a conveyor to one of three alternately-selected destinations. Now, close the model and take a look at another example model that determines the destination of loads based on the load’s type.

Example 9.2: Sorting load types in a conveyor system

In this example of a warehouse sortation system, loads on a conveyor are routed to one of three destinations, depending on load type. Consider the conveyor layout shown below:



Layout of example model 9.2

Trucks arrive at the warehouse every 15 minutes. Each truck carries a random number of light, medium, and heavy loads; the number and type of loads on each truck are defined in a data file.

Upon arrival, each truck’s contents are verified at an entrance location. Verification and completion of the required paperwork takes a time that is uniformly distributed between 7 and 13 minutes. The trucks then move into the yard (modeled as an infinite-capacity queue) where they wait to park at one of four alternately-selected unloading docks.

A worker at the docks unloads each truck, one truck at a time. The unloading time for each load varies by type, as shown in the table below:

Load type	Unloading time per load
Light	Uniformly distributed time between 23 and 27 seconds.
Medium	Normally distributed time with a mean of 30 seconds and a standard deviation of 3 seconds.
Heavy	Triangularly distributed time with a minimum value of 32 seconds, a most-likely value of 35 seconds, and a maximum value of 42 seconds.

The worker places loads on a lane of conveyor in front of the current dock. The loads travel to one of three destinations depending on type, as shown in the table below:

Load type	Destination location
Light	out_1
Medium	out_2
Heavy	out_3

After arriving at their destination, loads are removed from the system.

To become familiar with the movement of loads in the system:

Step 1 ***Import*** and ***run*** a copy of example model 9.2.

Step 2 When you are ready to continue, ***select*** Edit Model from the Control menu to return to the build environment.

You are now ready to look at the example model logic to see how the system is simulated.

Modeling the arrival and unloading of trucks

In example model 9.2, trucks are modeled as a load type named “L_truck.” A load creation specification creates a new load of type L_truck every 15 minutes and sends the load to the process P_arrive. The P_arrive arriving procedure models the unloading of trucks, as shown below:

```
begin P_arrive arriving
   move into Q_paperwork
   wait for u 10,3 min
   set A_dock to nextof(1,2,3,4)
   read A_light,A_medium,A_heavy from “arc/data2.txt”
   at end
      begin
         print “Ran out of data in file data2” to message
         terminate
      end
   move into Q_yard
   move into Q_dock(A_dock)
   get R_worker
   set A_type to 1
   while A_light > 0 do begin /* Unload light loads first */
      wait for uniform 25,2 sec
      clone 1 loads to P_conveyor new load type L_light
      dec A_light by 1
   end
   set A_type to 2
   while A_medium > 0 do begin/* Unload medium loads second */
      wait for normal 30,3 sec
      clone 1 loads to P_conveyor new load type L_medium
      dec A_medium by 1
   end
   set A_type to 3
   while A_heavy > 0 do begin /* Unload heavy loads last */
      wait for t 32,35,42 sec
      clone 1 loads to P_conveyor new load type L_heavy
      dec A_heavy by 1
   end
   free R_worker
   send to die
end
```

Trucks that execute the procedure first move into the infinite-capacity queue Q_paperwork and then delay for the time required for verification and paperwork. The value of the load attribute A_dock is then alternately set for each truck. The attribute is used later, not only to move trucks into the correct dock but also to place loads on the correct entrance station on the conveyor.

The number of loads in each truck is read from a data file, as explained in the following section.

Reading load quantities from a data file

The data file that defines the number of loads on each truck is named “data2.txt” and is saved in the model’s archive directory. The first few lines of the file are shown below:

```
Light       Medium      Heavy
5           3           4
2           3           7
4           7           1
...
```

The file headers are read in the model initialization function, as shown below:

```
begin model initialization function
   read V_headers from “arc/data2.txt” with delimiter “\n”
   return true
end
```

Three load attributes (one for each type) are used to keep track of the number of loads onboard each truck.The P_arrive arriving procedure reads the number of loads from the data file into the attributes, as shown below:

```
read A_light,A_medium,A_heavy from “arc/data2.txt”
at end
   begin
      print “Ran out of data in file data2” to message
      terminate
   end
```

At this point, a truck executing the arriving procedure has four attribute values defined. The first, A_dock, defines the truck’s parking location at the docks. The remaining three attributes define the number of light, medium, and heavy loads that are on the truck.

After reading from the data file, the truck moves into the yard and then moves into its assigned dock, as shown below:

```
move into Q_yard
move into Q_dock(A_dock)
```

The logic for unloading each truck is described in the next section.

Creating loads for each truck

The P_arrive arriving procedure defines three loops to unload each truck (one loop for each load type). The worker unloads the light loads first, then the medium loads, then the heavy loads, as shown below:

```
get R_worker
set A_type to 1
while A_light > 0 do begin
   wait for uniform 25,2 sec
   clone 1 loads to P_conveyor new load type L_light
   dec A_light by 1
end
set A_type to 2
while A_medium > 0 do begin
   wait for normal 30,3 sec
   clone 1 loads to P_conveyor new load type L_medium
   dec A_medium by 1
end
set A_type to 3
while A_heavy > 0 do begin
   wait for t 32,35,42 sec
   clone 1 loads to P_conveyor new load type L_heavy
   dec A_heavy by 1
end
free R_worker
send to die
```

The truck first claims the resource R_worker. The worker remains claimed throughout the unloading process, until the truck is empty.

Each of the three loops in the procedure is preceded by an action that sets the value of the load attribute A_type. The value of the attribute indicates the type of load that will be unloaded next (a value of one indicates light loads, a value of two indicates medium loads, and a value of three indicates heavy loads). The value of this attribute is used later to route loads to the correct destination.

Each loop begins by delaying for the required amount of unloading time. After each delay, the `clone` action is used to create a new load of the correct type and send it to the P_conveyor process (described in the next section). The load attributes that track the number of onboard loads are decremented after each new load is cloned. When all of the loads of a specific type have been unloaded (the attribute value is equal to zero), the next unloading loop is executed.

When all of the loads have been unloaded, the resource R_worker is freed and the load that represents the truck is sent to die.

Sorting loads by type

Loads that are cloned by a truck are sent to the P_conveyor process. This process places loads on the conveyor (at the correct station) and causes them travel to their destination, which is determined by load type.

The P_conveyor arriving procedure is shown below:

```
begin P_conveyor arriving
    move into conv:in_(A_dock)
    travel to conv:out_(A_type)
    send to die
end
```

Both actions in the procedure use load attribute values that were defined in the P_arrive arriving procedure. The `move` action that places loads on the conveyor uses the value of the load attribute A_dock to align the conveyor entrance stations with the dock queue of the parked truck. The `travel` action uses the value of the load attribute (A_Type) to send loads to one of the three destination stations (out_1, out_2, or out_3), depending on the load's type.

After arriving at the destination station, loads are sent to die.

Modeling different types of conveyors

Until now, loads in the conveyor systems that you have modeled have either been removed from the conveyor for processing (see "Example 6.1: Drawing a conveyor system" on page 6.9 of the "Introduction to Conveyors" chapter), or loads have traveled through the system from an entrance station to an exit station, without stopping. Often, however, a real-world conveyor system requires loads to temporarily stop on the conveyor for processing. How a conveyor behaves when a load stops varies depending on the type of conveyor. One of the primary distinctions between conveyor types is whether their sections are accumulating or non-accumulating.

An **accumulating section** is a section on which loads travel independently. Think of an accumulating section as a series of rollers; when a preceding load stops, trailing loads continue to move until they reach the stopped load. Accumulating sections are used to model roller and queueing conveyors.

A **non-accumulating section** is a section on which loads stop and start at the same time. Think of an accumulating section as a moving belt that carries loads; when the belt stops, the loads traveling on the belt also stop. Non-accumulating sections are used to model belt and chain conveyors.

Assume that your company manufactures glass picture tubes for televisions and transfers them from one station to another on a conveyor. What kind of conveyor sections would you use to transfer the tubes? You would use non-accumulating sections so that the picture tubes do not bump into each other when one tube stops.

In contrast, suppose that you are the manager of a distribution facility that moves boxes of T-shirts. It probably does not matter if those boxes bump, so you can use accumulating sections.

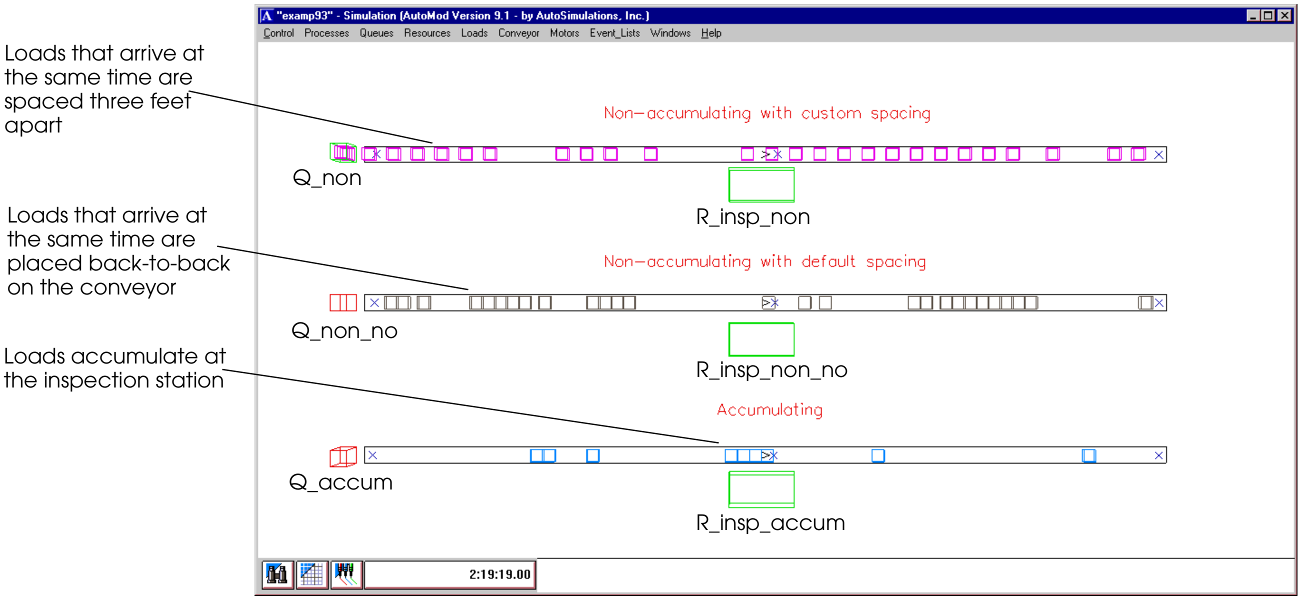
Note

When modeling non-accumulating sections, it is important to correctly simulate the amount of space between loads when they are moving and when they are stopped on the conveyor. These measures are usually not important when modeling an accumulating conveyor.

To understand different types of conveyor sections, close example model 9.2 and look at the following example.

Example 9.3: Accumulating and non-accumulating sections

Example model 9.3 contains three conveyor sections, as shown below:


Loads that arrive at the same time are spaced three feet apart
Loads that arrive at the same time are placed back-to-back on the conveyor
Loads accumulate at the inspection station
"examp93" - Simulation (AutoMod Version 9.1 - by AutoSimulations, Inc.)
Control Processes Queues Resources Loads Conveyor Motors Event_Lists Windows Help
Non–accumulating with custom spacing
Q_non
R_insp_non
Non–accumulating with default spacing
Q_non_no
R_insp_non_no
Accumulating
Q_accum
R_insp_accum
2:19:19.00

Layout of example model 9.3

Loads are created at the same time for each section. Loads first move into a queue at the beginning of the section and then travel across the section from left to right. Loads stop at a station in the middle of the section, where they are inspected by an inspector for a constant amount of time. When loads reach the end of a conveyor section, they are sent to die.

The topmost conveyor section is a non-accumulating section; when one load stops at the inspection station, all loads on the conveyor stop. The section is defined to maintain a minimum of three feet of space between loads that are moving or that are stopped on the conveyor. As a result, loads that arrive at the same time are placed three feet apart on the section.

The middle conveyor section is also non-accumulating. However, unlike the topmost section, this section is defined without any extra space requirements for loads that are traveling on the conveyor. As a result, loads that arrive at the same time are placed back-to-back on the conveyor.

The lowest conveyor section is an accumulating section; when one load stops at the inspection station, the remaining loads keep moving until they reach the loads in front of them. As a result, loads back up at the inspection station.

To become familiar with the differences between the three sections:

Step 1 ***Import*** and ***run*** a copy of example model 9.3.

Step 2 When you are ready to continue, ***close*** the model.

Changing conveyor attributes

The following entities in a conveyor system have attributes to define their characteristics:

- Sections
- Transfers
- Motors
- Stations
- Photoeyes (not discussed in this textbook)

For example, conveyor sections have attributes that define their width, accumulation, velocity, and so on. Until now, you have been drawing conveyor systems using the default attribute values for entities in the system. In this chapter, you will learn how to change attribute values for two conveyor entities: sections and transfers.

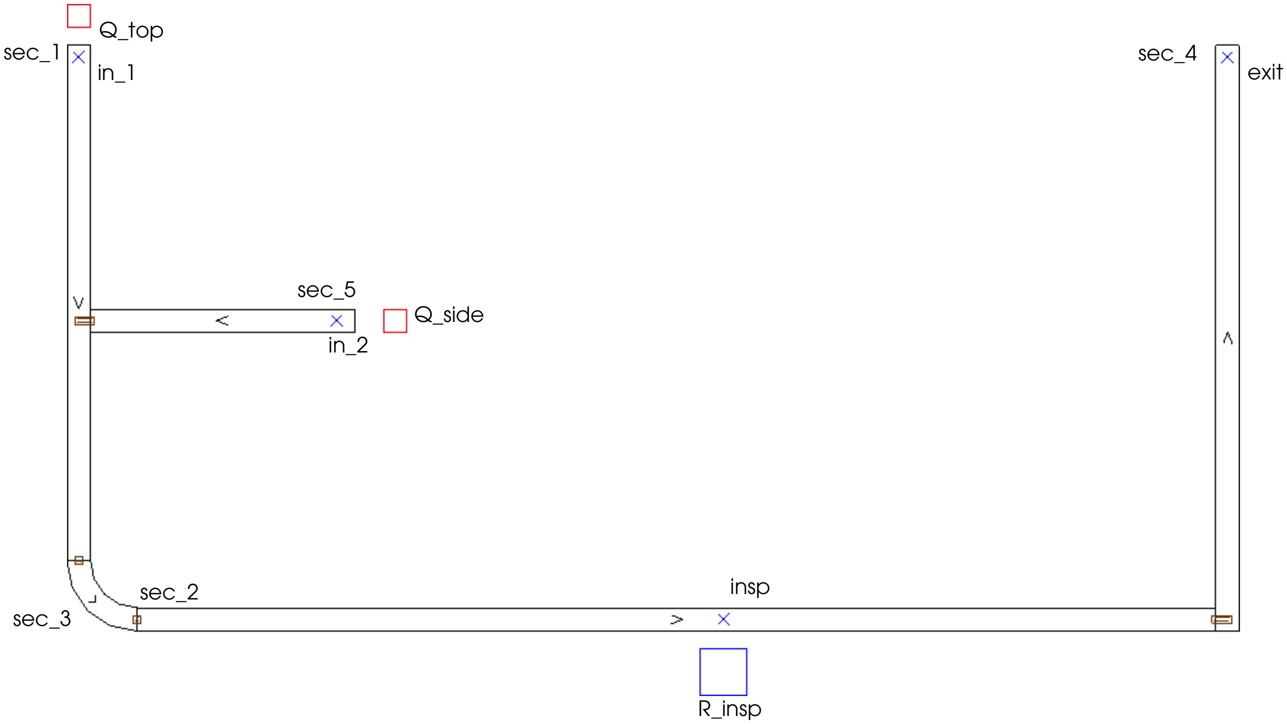
There are two ways to change conveyor attribute values. You can change the values for an individual conveyor entity (for example, a single section), or you can change the value for a group of conveyor entities that are all of the same type (for example, a group of sections that are all of type “Roller”). Think of conveyor types as templates that define attributes for more than one conveyor entity.

Each conveyor entity in a model belongs to a default type. For example, new sections that you draw in a conveyor system are all of type “DefaultSection,” and new transfers are all of type “DefaultTransfer.” These default templates are pre-defined using attribute values that are commonly used in conveyor systems. For example, the template “DefaultSection” defines section width as 4 feet, sections as accumulating, and section velocity as 1 foot per second. When you change attribute values in the template, all of the entities of that type are automatically updated to use the new values. For example, you can edit the template “DefaultSection” and change the section velocity to 3 feet per second; all sections that are of type “DefaultSection” are automatically updated to a speed of 3 feet per second.

To gain some experience changing the attributes of entities in a conveyor system, use the following example model.

Example 9.4: Customizing a conveyor system

Consider the conveyor layout for example model 9.4, shown below:



Layout of example model 9.4

Two types of loads are processed in this system: loads of type “L_top” and type “L_side.” Both load types have an interarrival time that is exponentially distributed with a mean of 10 minutes and 30 seconds. Loads of type “L_top” move into the infinite-capacity queue “Q_top” and get on the conveyor at station “in_1.” Loads of type “L_side” move into the infinite-capacity queue “Q_side” and get on the conveyor at station “in_2.”

Both load types travel to the inspection station “insp,” where they are inspected by an inspector for a time that is exponentially distributed with a mean of 5 minutes. The loads then travel to the station “exit,” where they are removed from the system.

You will use example model 9.4 to experiment with different conveyor entity attributes and see how to model virtually any conveyor system. Currently, all the conveyor entities in the model are using the default attribute values. Before changing any values, do the following:

Step 1 ***Import*** and ***run*** a copy of example model 9.4 to become familiar with load movement in the system.

Step 2 When you are ready to continue, ***edit*** the model.

Editing attributes in section templates (types)

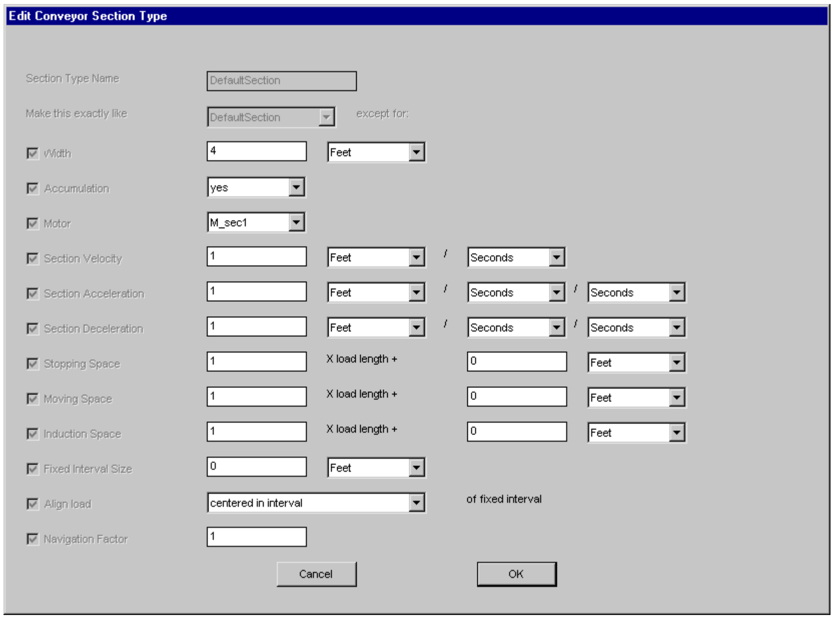
Currently, all the sections in example model 9.4 are of type “DefaultSection.” To edit the attribute values for the “DefaultSection” template, do the following:

Step 1 ***Open*** the conveyor system (the system is named “conv”).

Step 2 From the System menu, ***select*** Show Defaults. The Conveyor Defaults window opens.

Step 3 ***Click*** Section Types. The Section Type window opens.

Step 4 The “DefaultSection” type is already selected, so ***click*** Edit to edit this template. The Edit Conveyor Section Type window opens, as shown below:



Edit Conveyor Section Type

Section Type Name	DefaultSection				
Make this exactly like	DefaultSection	except for:			
☑ Width	4	Feet			
☑ Accumulation	yes				
☑ Motor	M_sec1				
☑ Section Velocity	1	Feet	/ Seconds		
☑ Section Acceleration	1	Feet	/ Seconds	/ Seconds	
☑ Section Deceleration	1	Feet	/ Seconds	/ Seconds	
☑ Stopping Space	1	X load length +	0	Feet	
☑ Moving Space	1	X load length +	0	Feet	
☑ Induction Space	1	X load length +	0	Feet	
☑ Fixed Interval Size	0	Feet			
☑ Align load	centered in interval	of fixed interval			
☑ Navigation Factor	1				

The Edit Conveyor Section Type window

The window provides a list of all the section attributes and their values.

Note Several options in the Edit Conveyor Section Type window are disabled, because these options are used only when editing individual sections (discussed on the next page.)

All the sections in example model 9.4 are of type “DefaultSection,” so each individual section inherits the attribute values that are displayed in this window; when you change attribute values in the template, all sections in the model are automatically updated. Before changing any values, you will learn how to edit attribute values for individual sections.

Step 5 ***Click*** OK to close the Edit Conveyor Section Type window.

Step 6 To close the Conveyor Defaults window, ***click*** OK.

Editing individual section attributes

You can edit the attribute values of an individual section to apply a new section type, or to override attribute values that the section has inherited from its current section type. To edit an individual section's attributes:

Step 1 If the Select tool is not already selected on the Conveyor palette, ***click*** Select.

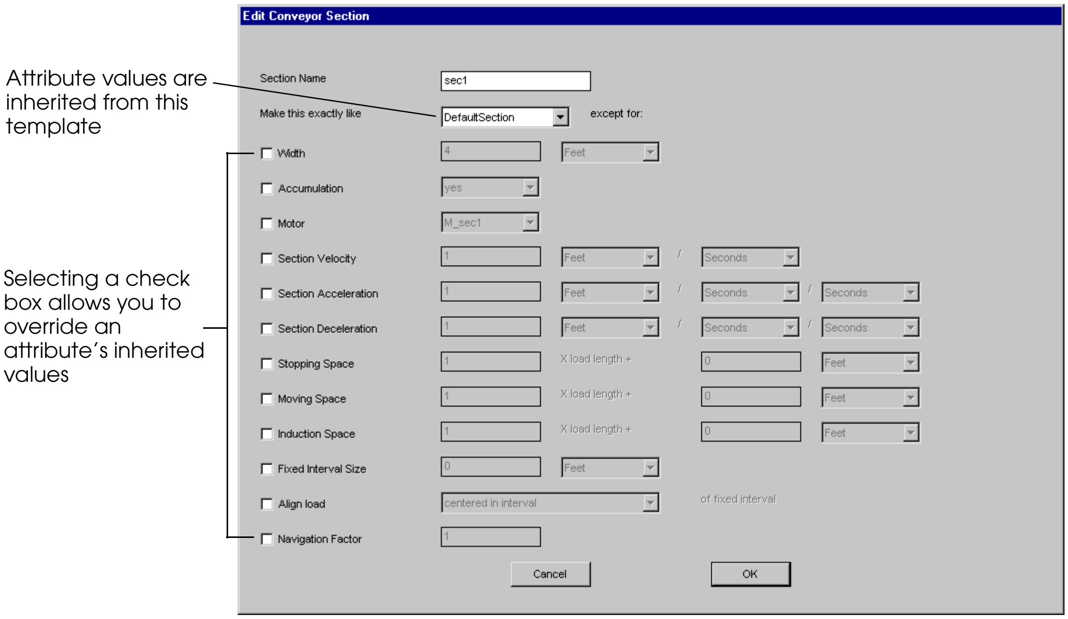
Step 2 ***Select*** a section to edit. (For this example, you can select any section in the model.)

Tip You can select a section by dragging a selection box that includes part of the section or by clicking the section in the Work Area window. The section color turns green to indicate that it is selected.

Step 3 From the Edit menu, ***select*** Edit. The Section Edit window opens.

Step 4 ***Click*** Attributes. The Edit Conveyor Section window opens.

The Edit Conveyor Section window allows you to change the section's name or type. You can also select one or more attribute check boxes to override values that are defined by the section's type.



Attribute values are inherited from this template

Selecting a check box allows you to override an attribute's inherited values

Edit Conveyor Section

Section Name: sec1

Make this exactly like: DefaultSection except for:

☐ Width: 4 Feet

☐ Accumulation: yes

☐ Motor: M_sec1

☐ Section Velocity: 1 Feet / Seconds

☐ Section Acceleration: 1 Feet / Seconds / Seconds

☐ Section Deceleration: 1 Feet / Seconds / Seconds

☐ Stopping Space: 1 X load length + 0 Feet

☐ Moving Space: 1 X load length + 0 Feet

☐ Induction Space: 1 X load length + 0 Feet

☐ Fixed Interval Size: 0 Feet

☐ Align load: centered in interval of fixed interval

☐ Navigation Factor: 1

Edit Conveyor Section window

Tip To restore the inherited value of an attribute that is currently overriding the template, clear the check box for that attribute.

Take a few moments to become familiar with how to change attribute values in the window.

Step 5 To accept your changes, you would click OK. In this case, however, ***click*** Cancel to cancel your changes and close the Edit Conveyor Section window.

Step 6 When you select a section, the section's stations and transfers are also selected. To stop editing all selected entities, ***click*** OK, Quit Edit Each in the Section Edit window.

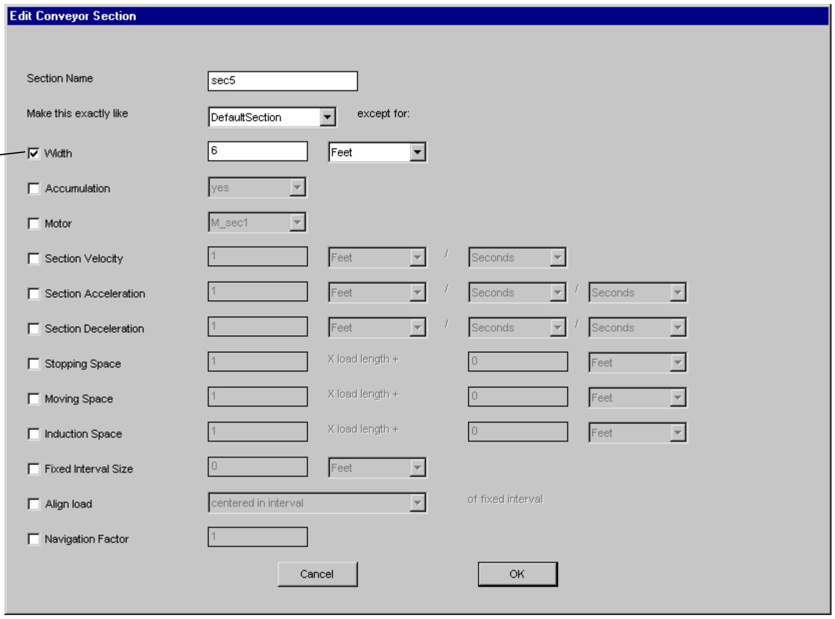
Now you are ready to customize the conveyor system in example model 9.4 by changing entity attribute values.

Defining section width

The **Width** attribute defines how wide a conveyor section is. In example model 9.4, you want to increase the width of the horizontal entrance lane by 2 feet.

Step 1 ***Select*** section “sec5.”

Step 2 ***Edit*** the attributes of section “sec5” and ***change*** the section width from 4 to 6 feet, as shown below:



The section’s width is set to 6 feet

Edit Conveyor Section

Section Name	sec5					
Make this exactly like	DefaultSection	except for:				
☑ Width	6	Feet				
☐ Accumulation	yes					
☐ Motor	M_sec1					
☐ Section Velocity	1	Feet	/	Seconds		
☐ Section Acceleration	1	Feet	/	Seconds	/	Seconds
☐ Section Deceleration	1	Feet	/	Seconds	/	Seconds
☐ Stopping Space	1	X load length +		0		Feet
☐ Moving Space	1	X load length +		0		Feet
☐ Induction Space	1	X load length +		0		Feet
☐ Fixed Interval Size	0	Feet				
☐ Align load	centered in interval			of fixed interval		
☐ Navigation Factor	1					

Step 3 ***Click*** OK to close the Edit Conveyor Section window. The increased width of the conveyor in the Work Area window is increased.

Step 4 ***Click*** OK, Quit Edit Each in the Section Edit window.

Tip In general, you should change the width of a section *before* you draw it. To do this, edit the section’s attributes by clicking Attributes in the window that opens when you select a section drawing tool.

Defining section accumulation

Currently, all sections in example model 9.4 are accumulating sections. When you run the model, each load stops at the inspection station, and trailing loads accumulate behind the stopped load. You want to see the effect on traveling loads if the inspection lane is modeled as a belt conveyor (a non-accumulating section).

Step 1 ***Select*** section “sec_2.”

Step 2 ***Edit*** the attributes of section “sec_2” and ***change*** the Accumulation value to “no.”

Step 3 ***Click*** OK to close the Edit Conveyor Section window, then ***click*** OK, Quit Edit Each to close the Section Edit window.

Step 4 ***Export*** and ***run*** the model to observe the difference. Loads no longer accumulate on section “sec_2.” When a load needs to be inspected, the belt stops and all loads on the inspection lane stop moving.

Notice that section “sec_2” is less efficient as a non-accumulating section than it was as an accumulating section. There is a greater amount of empty space on the conveyor before the inspection station, because while the inspection lane is stopped, loads from the entrance lanes accumulate at the transfer to the inspection lane. In addition, loads that have completed the inspection process are not able to travel as quickly to the exit lane, because they must stop and wait each time another load on the section is inspected. Because the section is less efficient, you should now set the inspection lane back to an accumulating section.

To restore the original Accumulation attribute value, do the following:

Step 5 ***Edit*** the attributes of section “sec_2” and ***clear*** the Accumulation check box. The section is now an accumulating section.

Leave the Edit Conveyor Section window open to edit the section’s velocity, as described below.

Defining section velocity

The **Velocity** attribute defines the constant speed of a conveyor section when it is not accelerating or decelerating. You want to increase the velocity of the inspection lane.

Step 1 ***Edit*** the attributes of section “sec_2” and ***change*** the velocity of the section from 1 foot per second to 3 feet per second.

Step 2 ***Click*** OK to close the Edit Conveyor Section window, then ***click*** OK, Quit Edit Each to close the Section Edit window.

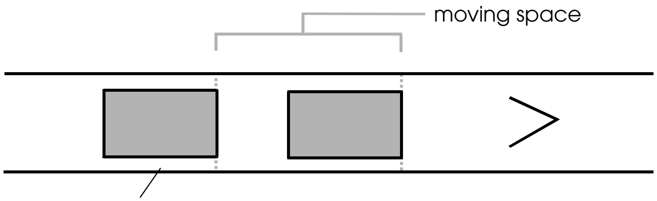
Step 3 ***Export*** and ***run*** the model, but do *not* change the display step.

Notice the difference in section velocity when the first load in the simulation transfers from the arc section to the inspection lane; the load travels much faster on the inspection lane.

Step 4 From the Control menu, ***select*** Edit Model to return to the build environment.

Defining section moving space

A load’s **leading edge** is the front edge in the load’s current direction of travel. The **Moving Space** attribute defines the minimum amount of space between the leading edges of two loads that are moving on a conveyor section, as shown below:



Because of the section’s moving space, the trailing load cannot be any closer to the preceding load while moving on the section

Moving space on a conveyor section

A section’s moving space is defined by the equation:

Moving Space = ____ × load length + ____ feet

The default moving space defined for the “DefaultSection” type is:

Moving Space = 1 × load length + 0 feet

That is, a moving load requires a space equal to its length on the conveyor.

If you increase a section’s moving space to a value greater than the load’s length, the extra space is modeled as a “buffer” behind the load (as shown in the illustration above). To demonstrate the effect of increasing a section’s moving space, change the moving space required by loads on the vertical entrance lane in example model 9.4.

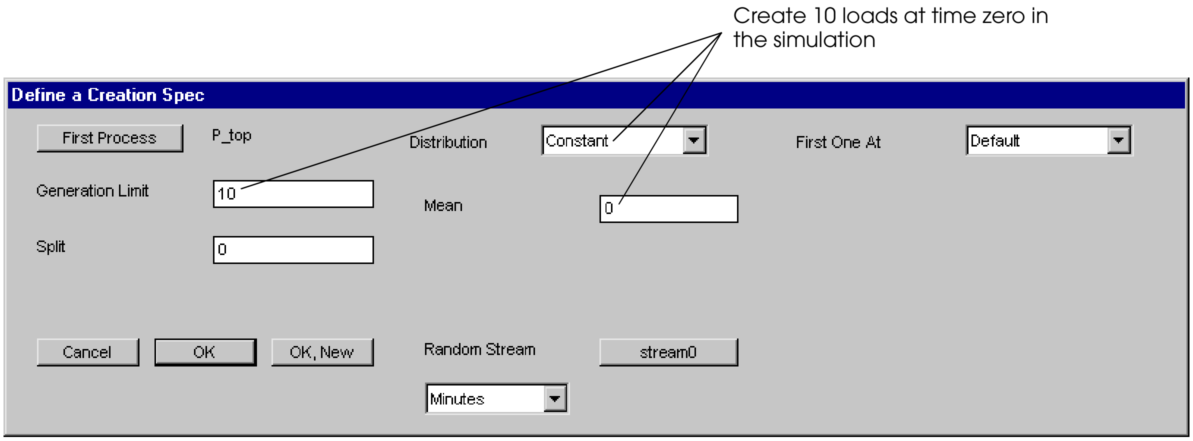
Step 1 ***Select*** section “sec_1.”

Step 2 ***Edit*** the section’s attributes and ***set*** the moving space to:

Moving Space = 1 × load length + 10 feet

Step 3 ***Click*** OK to close the Edit Conveyor Section window, then ***click*** OK, Quit Edit Each to close the Section Edit window.

Step 4 To see the buffer between loads, you need to create several loads at the same time during the simulation. ***Open*** the Process system, then ***edit*** the load creation specification for loads of type L_top to create 10 loads at time zero in the simulation, as shown below:



Create 10 loads at time zero in the simulation

Define a Creation Spec

First Process: P_top
Distribution: Constant
First One At: Default
Generation Limit: 10
Mean: 0
Split: 0
Random Stream: stream0
Minutes

Editing the load creation specification for loads of type L_top

Step 5 ***Export*** and ***run*** the model.

Notice the space between the loads of type L_top as they travel to the inspection lane. Although all loads attempt to get on the conveyor at the same time, they are placed 10 feet apart due to the section's moving space.

Step 6 From the Control menu, ***select*** Edit Model to return to the build environment.

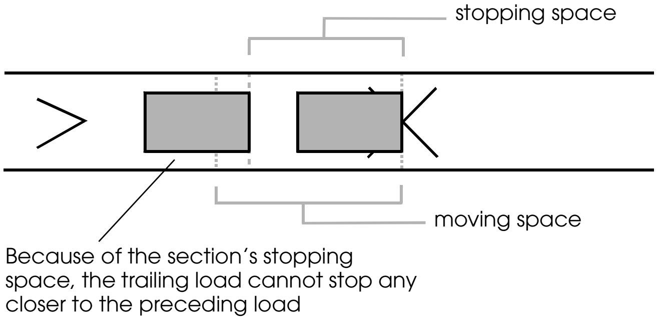
Because a load's size is factored into a section's moving space, different sized loads can have different moving space requirements on the section. You can define a constant moving space for all loads as follows:

$$\text{Moving Space} = 0 \times \text{load length} + 12 \text{ feet}$$

In this case, the moving space for all loads on the section is 12 feet. Because moving space is measured from the load's leading edge, the actual amount of "buffer" space behind each load varies depending on the length of loads in the system.

Defining section stopping space

The **Stopping Space** attribute defines the minimum amount of space between the leading edges of two loads that are stopped on a conveyor section, as shown below:


stopping space
moving space
Because of the section's stopping space, the trailing load cannot stop any closer to the preceding load

Stopping space on a conveyor section

Important

A section's stopping space can be less than its moving space, allowing loads to stop closer than they were able to travel on the section (as shown above). However, *a section's moving space must be greater than or equal to its stopping space*. If two moving loads on a section are already closer together than the section's defined stopping space, the loads cannot be repositioned when they stop to create a greater distance between them.

A section's stopping space is defined by the equation:

Stopping Space = ____ × load length + ____ feet

The default stopping space defined by the "DefaultSection" type is:

Stopping Space = 1 × load length + 0 feet

That is, a stopped load requires a space equal to its length on the conveyor.

As with moving space, if you increase a section's stopping space to a value greater than the load's length, the extra space is modeled as a "buffer" behind the load (as shown in the illustration above). To demonstrate the effect of increasing a section's stopping space, change the stopping space required by loads on the inspection lane.

Step 1 ***Open*** the conveyor system and ***select*** section "sec_2."

Step 2 Because the moving space must be greater than or equal to the stopping space, ***edit*** the section's attributes and ***set*** the moving space to:

Moving Space = 1 × load length + 5 feet

and ***set*** the stopping space to:

Stopping Space = 1 × load length + 5 feet

Step 3 ***Click*** OK to close the Edit Conveyor Section window and ***click*** OK, Quit Edit Each to close the Section Edit window.

Step 4 ***Export*** and ***run*** the model. Notice the effect that the increased stopping space has on loads that accumulate at the inspection station. The loads stop 5 feet apart when accumulated on the section, instead of right next to each other as they did before.

Step 5 From the Control menu, ***select*** Edit Model to return to the build environment.

When modeling non-accumulating conveyors, it is important to accurately simulate the space between stopped loads on the conveyor.

Stopping space can also be important when modeling accumulating conveyors. For example, live (powered) roller conveyors are an accumulating conveyor that can stop loads before they collide. Another use for setting stopping space on an accumulating section is for simulating the transportation of bulk material on a conveyor (for example, letters in a postal facility). In this case, the loads stack on top of each other when they stop on an accumulating conveyor. You can model load overlap by defining a section’s stopping space using a fraction of the load length. For example,

$$\text{Stopping Space} = 0.5 \times \text{load length} + 0 \text{ feet}$$

This stopping space causes trailing loads to overlap half of the preceding load on the conveyor.

Now that you have some experience editing section attributes, you are ready to learn more about transfers and their attributes in a model.

Modeling transfers

As previously mentioned, transfers are created automatically as you draw sections in a conveyor system. Transfers allow loads to move from one section to another. There are three types of transfers in a simulation:

- Ahead transfers
- Side transfers
- Reverse transfers

The angle between two connected sections determines a transfer's type. The transfer types differ based on how they affect load orientation in a simulation. These concepts are discussed in the following sections.

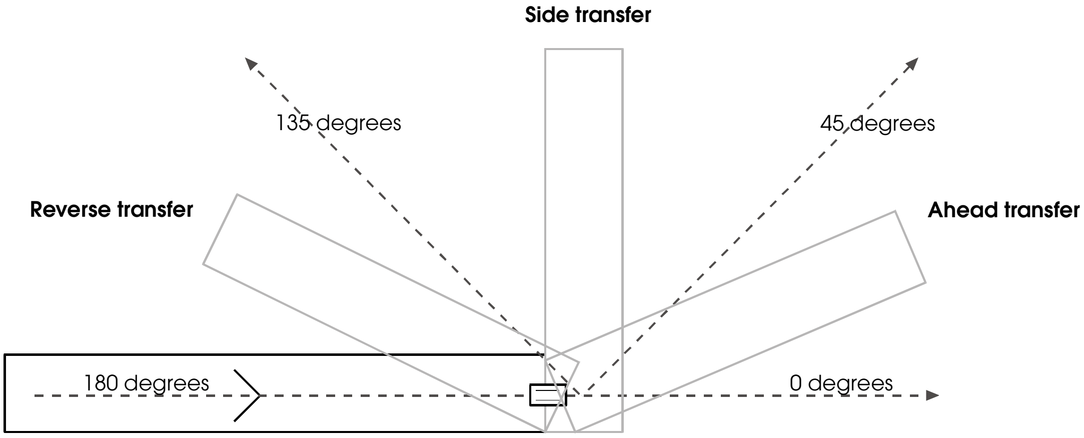
How a transfer's angle determines its type

A transfer's angle is determined when you draw two connected sections in a conveyor system. The **transfer angle** is the difference between the direction traveled prior to and after a transfer, measured in degrees. Therefore, the transfer angle cannot exceed 180 degrees. Depending on the transfer angle, a transfer's type is either an ahead, side, or reverse transfer, as shown in the table below:

If the transfer angle is...	Then the transfer type is...
Less than 45 degrees	Ahead transfer
Greater than or equal to 45 degrees, and less than or equal to 135 degrees	Side transfer
Greater than 135 degrees	Reverse transfer

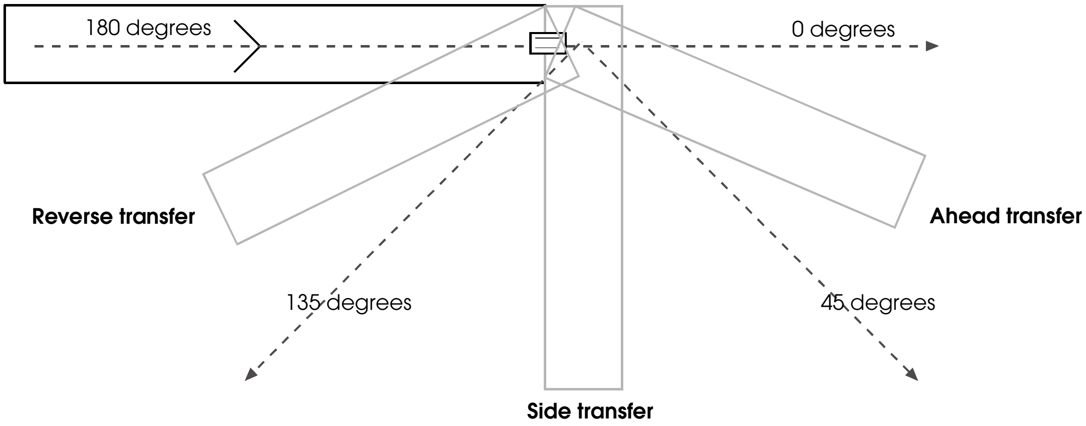
Note It is possible to define different ranges (in degrees) for each of the transfer types, but that concept is not discussed in this textbook.

The illustration below shows the relation between a transfer's angle and the transfer's type.



Relation between a left transfer angle and transfer type

Note that the relation also holds true for transfers in the opposite direction, as shown below:



Relation between a right transfer angle and transfer type

A transfer's type affects how transferring loads are oriented on a section after a transfer.

Determining load orientation on a conveyor

When a load gets on a conveyor for the first time, the load is positioned so that the direction of travel is along the load's positive X axis (in other words, the load's length on the conveyor is determined by the size of the load on the X axis and its width is determined by the size of the load on the Y axis).

When a load transfers from one section to another, the load's orientation may change depending on whether the transfer is an ahead, side, or reverse transfer (as explained in the following sections).

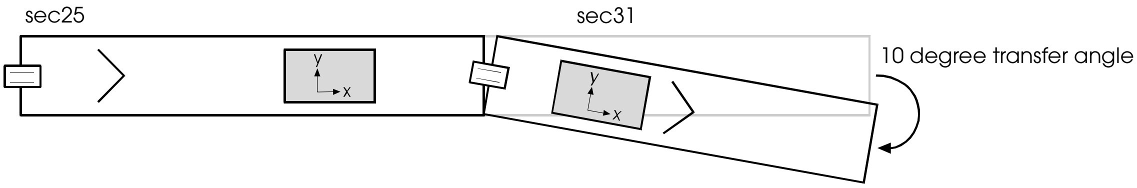
Note

Once changed, a load's orientation is maintained throughout the simulation. For example, if a load's orientation changes as a result of one or more transfers, and the load gets off the conveyor (for example, to move into a queue), and then back on the conveyor, the load maintains its last orientation. Therefore, the load's X axis may no longer be aligned with the direction of travel.

Load orientation after an ahead transfer

A load's direction of travel remains the same during an ahead transfer (a transfer with an angle that is less than 45 degrees).

An illustration of an ahead transfer with an angle of 10 degrees is shown below:



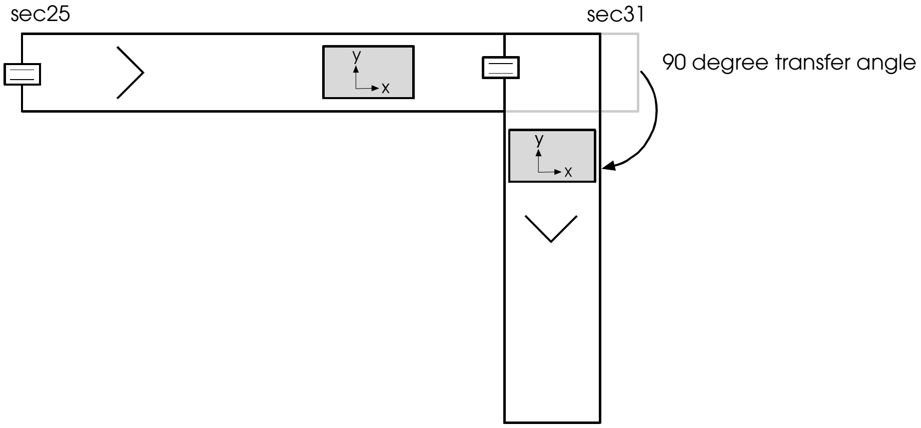
A load's direction of travel remains the same after an ahead transfer

The load, which is traveling in the positive X direction, continues traveling in the positive X direction on the destination section after the transfer (the **destination section** refers to the section to which the load is transferring).

Load orientation after a side transfer

A load's direction of travel changes during a side transfer (a transfer with an angle that is greater than or equal to 45 degrees and less than or equal to 135 degrees).

An illustration of a side transfer with an angle of 90 degrees is shown below:



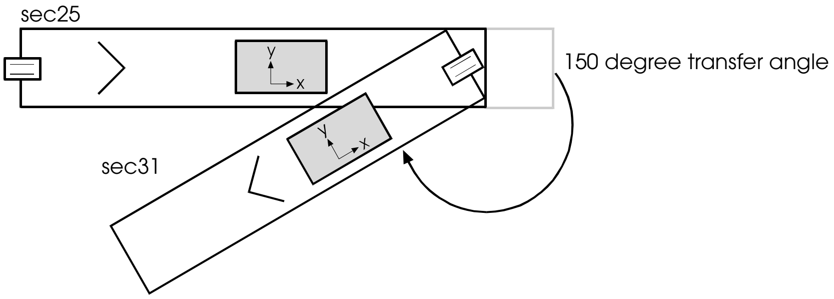
A load's direction of travel changes during a side transfer

The load's direction of travel changes from the positive X axis to the negative Y axis on the destination section.

Load orientation after a reverse transfer

A load's direction of travel reverses during a reverse transfer (a transfer with an angle that is greater than 135 degrees and less than or equal to 180 degrees).

An illustration of a reverse transfer with an angle of 150 degrees is shown below:



A load's direction of travel reverses during a reverse transfer

The load's direction of travel changes from the positive X axis to the negative X axis on the destination section.

Preparing example model 9.4

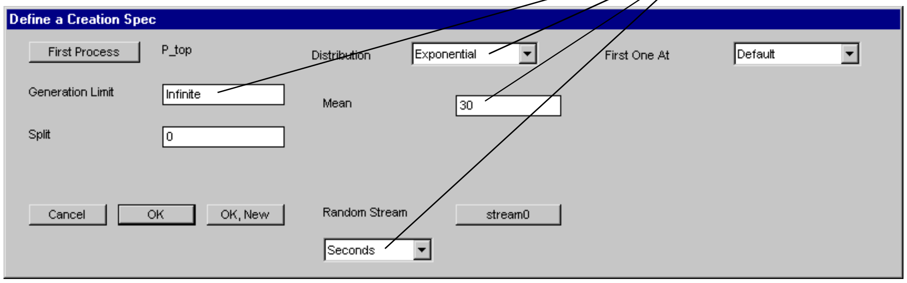
Now that you are more familiar with how transfers affect load movement during a simulation, you are ready to edit some of the transfer attribute values in the example model. But first, you need to speed up the load creation rate and prevent loads from stopping at the inspection station; making these modifications to the model will make changes to transfer attributes easily visible in the simulation.

To modify the example model (before editing any transfer attribute values), do the following:

Step 1 ***Open*** the Process system and ***edit*** the load creation specification for loads of type L_top.

Step 2 ***Change*** the specification to create an infinite number of loads with an interarrival time that is exponentially distributed with a mean of 30 seconds, as shown below:

Create an infinite number of loads with an interarrival time that is exponentially distributed with a mean of 30 seconds



Define a Creation Spec

First Process: P_top

Generation Limit: Infinite

Split: 0

Distribution: Exponential

Mean: 30

First One At: Default

Random Stream: stream0

Seconds

Editing the load creation specification for loads of type L_top

Step 3 ***Edit*** the load creation specification for loads of type L_side so that it is the same as the creation specification for loads of type L_top (exponentially distributed with a mean of 30 seconds).

Step 4 ***Edit*** the model logic and ***comment*** the two use actions that cause loads to stop at the inspection station, as shown below:

```
begin P_top arriving
   move into Q_top
   move into conv:in_1
   travel to conv:insp
/* use R_insp for e 5 min */
   travel to conv:exit
   send to die
end

begin P_side arriving
   move into Q_side
   move into conv:in_2
   travel to conv:insp
/* use R_insp for e 5 min */
   travel to conv:exit
   send to die
end
```

Step 5 ***Open*** the conveyor system and ***edit*** the attributes of section “sec_1.” ***Clear*** the Moving Space check box to restore the values defined by the section’s type.

Step 6 ***Click*** OK, Quit Edit Each to close the Edit Conveyor Section window.

You are now ready to edit transfer attributes in the example model.

Editing transfer attributes

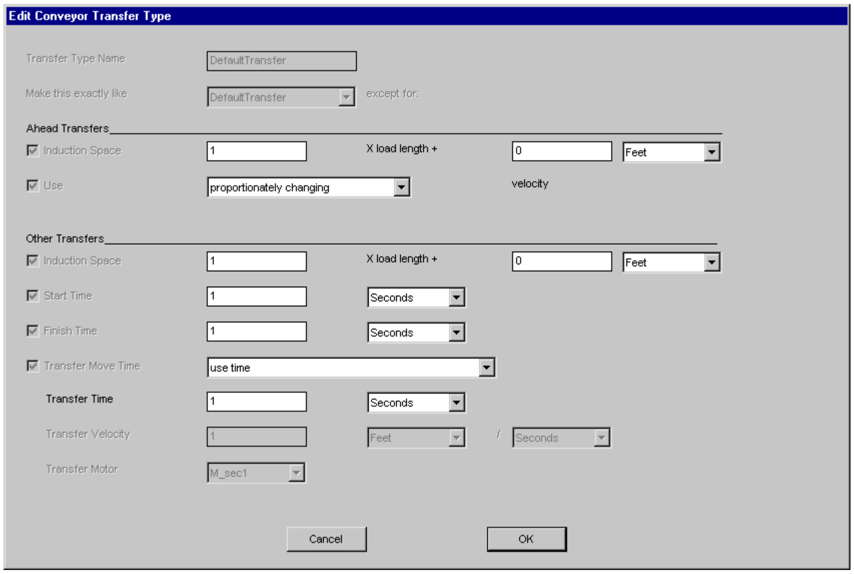
Like sections, transfer attributes can be edited either for an individual transfer or for a group of transfers of the same type. In example model 9.4, all transfers belong to the type “DefaultTransfer.” You will edit this template to change the values of all transfers in the system.

To edit the attribute values for the “DefaultSection” template, do the following:

Step 1 From the System menu, ***select*** Show Defaults. The Conveyor Defaults window opens.

Step 2 ***Click*** Transfer Types. The Transfer Type window opens.

Step 3 The “DefaultTransfer” type is already selected, so ***click*** Edit to edit the template. The Edit Conveyor Transfer Type window opens, as shown below:



Transfer Type Name: DefaultTransfer
Make this exactly like: DefaultTransfer except for:

Ahead Transfers
☑ Induction Space 1 X load length + 0 Feet
☑ Use proportionately changing velocity

Other Transfers
☑ Induction Space 1 X load length + 0 Feet
☑ Start Time 1 Seconds
☑ Finish Time 1 Seconds
☑ Transfer Move Time use time
Transfer Time 1 Seconds
Transfer Velocity 1 Feet / Seconds
Transfer Motor M_sec1

The Edit Conveyor Transfer Type window

The window provides a list of all the transfer attributes and their values.

Note

Several options in the Edit Conveyor Transfer Type window are disabled, because these options are used only when editing individual transfers.

Notice that attributes for “ahead” transfers are defined separately, whereas “side” and “reverse” transfers share the attribute values under the heading “Other Transfers.”

Defining transfer induction space

The **Induction Space** attribute defines the amount of space that must be present for a load to transfer from one conveyor section to another. To understand induction space, think of traveling on the entrance ramp of a busy highway; you look at what vehicle traffic is approaching to determine whether there is enough space for you to accelerate and merge into traffic. In a simulation, transfers determine whether or not loads can get on a new section by requiring the destination section to have a specific amount of induction space available for the transferring load.

Like moving and stopping space, a transfer's induction space is defined by the equation:

Induction Space = ____ × load length + ____ feet

The default induction space defined for the "DefaultTransfer" type is:

Induction Space = 1 × load length + 0 feet

That is, a destination section must have empty space equal to the length of a transferring load in order for the load to transfer. If the destination section does not have the required amount of empty space, the load waits at the end of its current section until the destination section has sufficient induction space available.

Induction space is measured starting at the leading edge of each load. Consequently, if you increase a transfer's induction space to a value greater than the load's length, the load requires extra space behind it in order to get on the destination section.

Important

A transfer's induction space must be greater than or equal to the destination section's moving space. Otherwise, the transfer would allow loads to violate the moving space of another load on the destination section.

To demonstrate the effect of changing a transfer’s induction space, increase the amount of space required for loads to make side transfers in example model 9.4.

There are two side transfers in the example model: one from section “sec_5” to “sec_1,” and the other from section “sec_2” to “sec_4.”

To increase the induction space for the side transfers, use an exaggerated value as follows:

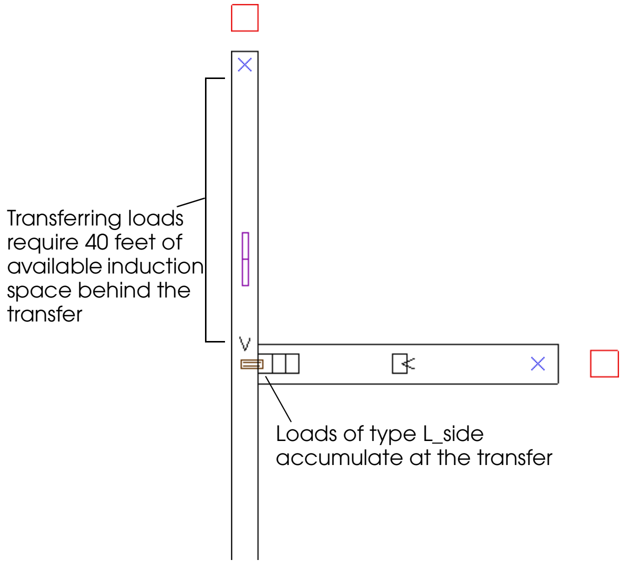
Step 1 In the Edit Conveyor Transfer Type window, ***change*** the induction space for Other Transfers to:

Induction Space = 1 × load length + 40 feet

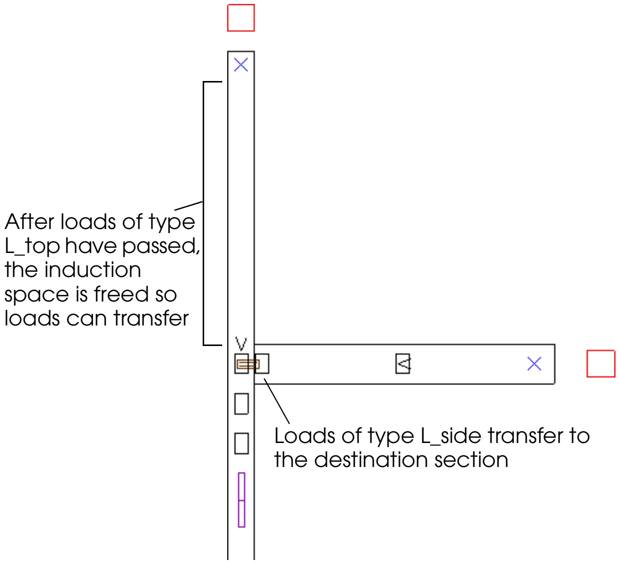
Step 2 ***Click*** OK to close the Edit Conveyor Transfer Type window, then ***click*** OK to close the Conveyor Defaults window.

Step 3 ***Export*** and ***run*** the model.

Notice the effect that the increased induction space has on loads of type L_side that are merging on the conveyor.


Transferring loads require 40 feet of available induction space behind the transfer
Loads of type L_side accumulate at the transfer

Loads on the destination section prevent the transfer


After loads of type L_top have passed, the induction space is freed so loads can transfer
Loads of type L_side transfer to the destination section

Loads transfer when 40 feet of induction space is available

Step 4 From the Control menu, ***select*** Edit Model to return to the build environment.

Defining transfer times

The Start Time, Finish Time, and Transfer Move Time attributes allow you to define how long a load takes to transfer from one section to another. The attributes are defined as follows:

Start Time The amount of time that a load takes to start the transfer.

Finish Time The amount of time that a load takes to complete the transfer.

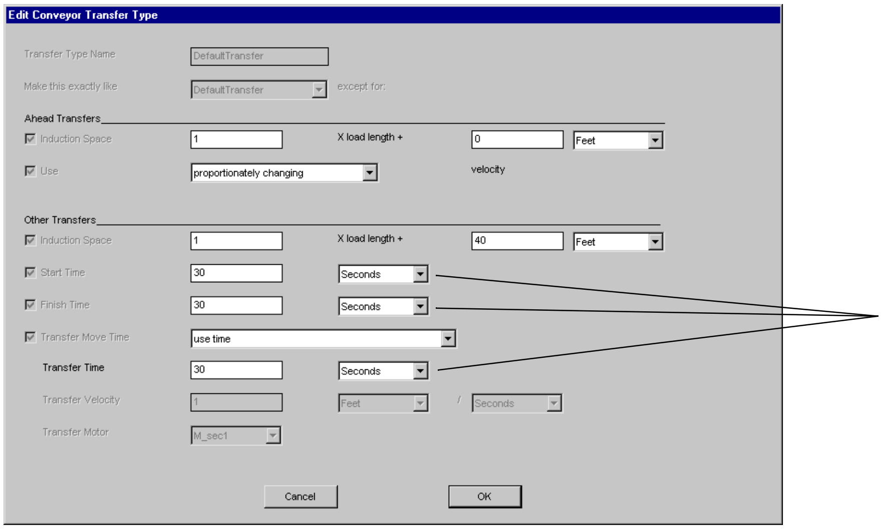
Transfer Move Time The amount of time that a load takes to move from one conveyor section to the next, exclusive of the transfer start and finish times. The Transfer Move Time can be defined as a time value or it can be calculated automatically by defining a transfer velocity and motor to drive the transfer.

Note The sum of the start, finish, and move times of the transfer defines the total amount of time that the load takes to transfer from one section to the next.

To understand why three separate times are modeled for the transfer, consider modeling a pop-up side transfer. A load arrives, the mechanism moves up (start time), the load is pushed to the next section (transfer move time), then the mechanism moves down (finish time).

To demonstrate the effect of changing transfer times, use exaggerated transfer times as follows:

Step 1 ***Edit*** the attributes of transfers of type “DefaultTransfer” and ***set*** the times for Other Transfers to 30 seconds each, as shown below:



Edit Conveyor Transfer Type

Transfer Type Name	DefaultTransfer			
Make this exactly like	DefaultTransfer	except for:		
Ahead Transfers				
☑ Induction Space	1	X load length +	0	Feet
☑ Use	proportionately changing		velocity	
Other Transfers				
☑ Induction Space	1	X load length +	40	Feet
☑ Start Time	30	Seconds		
☑ Finish Time	30	Seconds		
☑ Transfer Move Time	use time			
Transfer Time	30	Seconds		
Transfer Velocity	1	Feet	/	Seconds
Transfer Motor	M_sec1			

Set each of the transfer times to 30 seconds

Changing the transfer times

Tip To define the Transfer Move Time, select “use time” in the Transfer Move Time drop-down list and define a Transfer Time of 30 seconds, as shown above.

Step 2 ***Click*** OK to close the Edit Conveyor Transfer Type window, then ***click*** OK to close the Conveyor Defaults window.

Step 3 ***Export*** and ***run*** the model.

Notice the effect that the new times have on transferring loads. Each load at a side transfer delays for 30 seconds, takes 30 seconds to move to the new section, and then delays for another 30 seconds to complete the transfer.

Step 4 ***Close*** the model.

Now you are ready to learn how to use motors for greater control of load movement in a conveyor system.

Modeling motors

Motors drive conveyor sections and transfers. Motors, unlike other conveyor system entities, are not represented graphically in the simulation. When you draw the first conveyor section in a model, a new motor is automatically created; by default, this motor is used by all sections and transfers in the system. You can create additional motors and assign them to any section or transfer.

This chapter demonstrates how to use motors to:

- Model motor failures
- Model slugging conveyors
- Model indexing conveyors

Motor failures are similar to resource failures. When a motor goes down, loads on the sections and/or transfers that are powered by the motor also stop. After a delay (to simulate the time to repair), motors can be brought up, at which time load movement resumes.

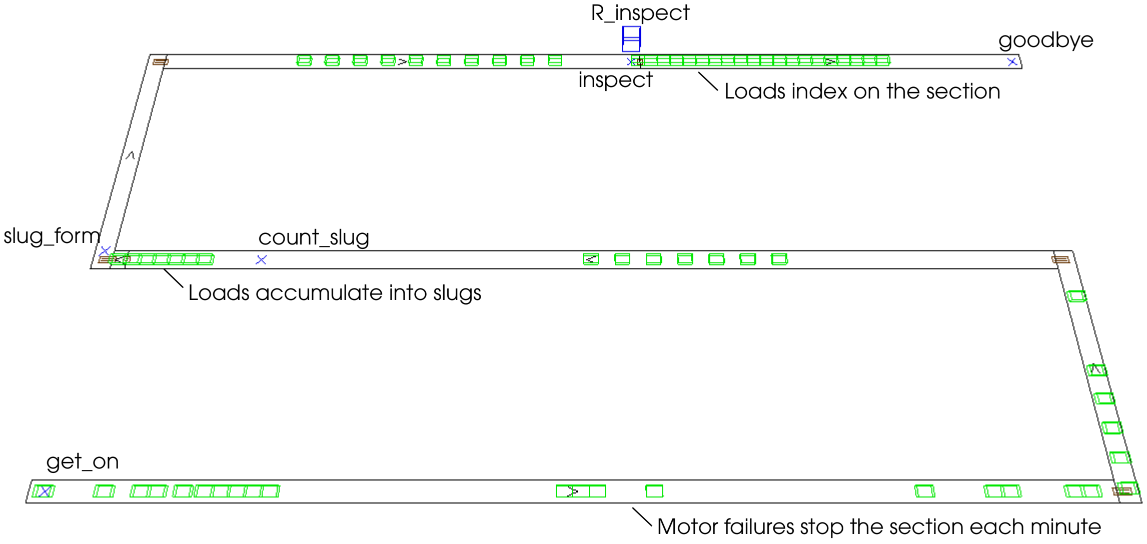
Slugging conveyors are sections where loads stop and accumulate into slugs. A **slug** is a group of two or more loads that travel together on the conveyor. When the required number of loads stop and accumulate (forming a complete slug), the loads are released as a group and continue traveling to their next destination.

Indexing conveyors are sections that advance only when a new load is transferred to the section. Indexing conveyors are useful when one conveyor section is less-used than other sections in a system; instead of running the conveyor section continuously, it runs only when required. Indexing conveyors save electricity, conveyor wear and tear, motor deterioration, and so on. An obvious disadvantage of indexing conveyors is that they increase the amount of WIP in a system.

To illustrate how motors are used in a simulation, take a look at the following example model.

Example 9.5: Modeling slugging and indexing conveyors

Consider the conveyor system illustrated below:


R_inspect
goodbye
inspect
Loads index on the section
slug_form
count_slug
Loads accumulate into slugs
get_on
Motor failures stop the section each minute

Layout of example model 9.5

Loads get on the conveyor at station “get_on” with an interarrival time that is exponentially distributed with a mean of 20 seconds. The horizontal entrance section demonstrates motor failures by operating for one minute, then failing for one minute, throughout the simulation.

Loads travel from station “get_on” to station “count_slug,” where they stop and accumulate into slugs consisting of 10 loads. As soon as 10 loads have accumulated, the slug continues to station “inspect,” where the loads are inspected by an inspector for a time that is exponentially distributed with a mean of 15 seconds.

Loads then get on an indexing conveyor section and travel to station “goodbye,” where they leave the system.

To become familiar with the movement of loads in the system:

Step 1 ***Import*** and ***run*** a copy of the *final* version of example model 9.5.

Step 2 When you are ready to continue, ***close*** the model.

You are now ready to learn how to create motors for use in a conveyor system.

Defining motors

When you draw the first section in a conveyor system, a motor is automatically created to drive the section. The motor is named using the section's name with an "M_" prefix. For example, if the first section you draw is named "sec1," a motor is created with the name "M_sec1." This motor drives all sections that you draw in the system.

If you want to stop a section independently during a simulation, the section must use its own motor. To learn how to create motors and assign them to sections in a conveyor system, you will create three motors for example model 9.5.

To define the motors, do the following:

Step 1 ***Import*** a copy of the *base* version of example model 9.5.

Note

When you import the model, an Attention window opens indicating that there were possible errors during the import. The warning appears because the motors that are referenced in the model logic are undefined. You will define the required motors in the following steps, so click OK to close the window.

Step 2 ***Open*** the conveyor system (named "conv"), then on the Conveyor palette, ***click*** Motor. The Motor window opens.

Step 3 To create the motor that simulates failures on the entrance section***, click*** New to create a new motor. The New Conveyor Motor window opens.

Step 4 ***Name*** the motor "M_fail" and ***click*** OK to define the motor.

Step 5 To create the motor that forms slugs on the conveyor, ***click*** New to create another motor. The New Conveyor Motor window opens.

Step 6 ***Name*** the motor "M_slug" and ***click*** OK to define the motor.

Step 7 To create the motor that indexes loads on the exit section, ***click*** New to create another motor. The New Conveyor Motor window opens.

Step 8 ***Name*** the motor "M_index" and ***click*** OK to define the motor.

You have now created three motors for use in the model. Currently, all sections in the model are using the default motor "M_sec1." To use the new motors that you have created, you must assign each motor to at least one section in the conveyor system.

Assigning motors to conveyor sections

When you define a new motor, you must assign it to a conveyor section. The motors that you created in example model 9.5 need to be assigned to the sections listed in the table below:

Motor name	Assign to section...
M_fail	sec1 (the horizontal entrance section)
M_slug	sec4 (the small section before the side transfer leading to station “form_slug”)
M_index	sec7 (the indexing section on which station “goodbye” is located)

To assign the motors that you created in example model 9.5, do the following:

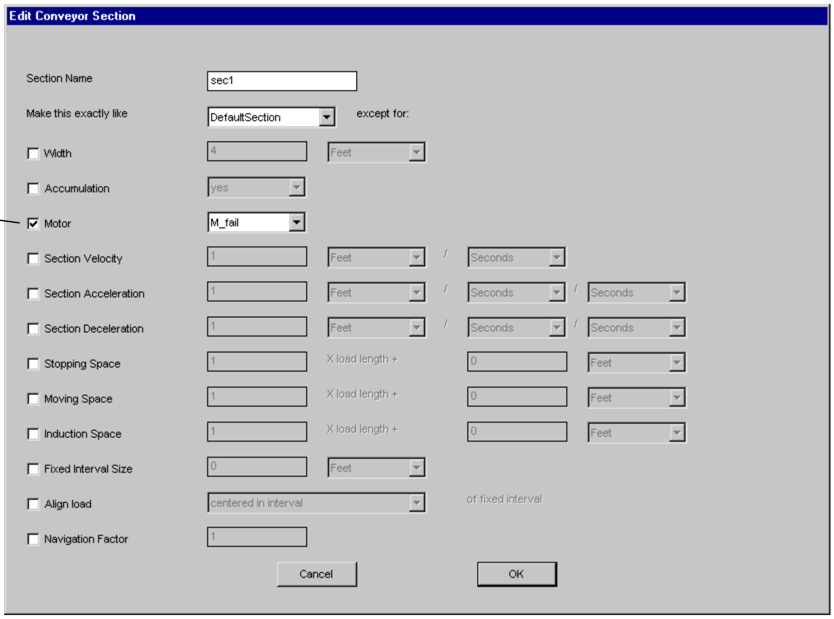
Step 1 On the Conveyor palette, ***click*** Select.

Step 2 ***Select*** section “sec1” and ***select*** Edit from the Edit menu. The Section Edit window opens.

Step 3 ***Click*** Attributes. The Edit Conveyor Section window opens.

Step 4 ***Select*** Motor, then ***select*** M_fail in the drop-down list, as shown below:

The section is set to use the motor M_fail



Edit Conveyor Section

Section Name: sec1

Make this exactly like: DefaultSection except for:

☐ Width: 4 Feet

☐ Accumulation: yes

☑ Motor: M_fail

☐ Section Velocity: 1 Feet / Seconds

☐ Section Acceleration: 1 Feet / Seconds / Seconds

☐ Section Deceleration: 1 Feet / Seconds / Seconds

☐ Stopping Space: 1 X load length + 0 Feet

☐ Moving Space: 1 X load length + 0 Feet

☐ Induction Space: 1 X load length + 0 Feet

☐ Fixed Interval Size: 0 Feet

☐ Align load: centered in interval of fixed interval

☐ Navigation Factor: 1

Assigning motor M_fail to section sec1

Step 5 ***Click*** OK to close the Edit Conveyor Section window. The motor “M_fail” is now assigned to section “sec1.”

Step 6 ***Repeat*** steps 2–5 to assign motor “M_slug” to section “sec4” and motor “M_index” to section “sec7.”

Tip ☞ When assigning the motor “M_slug,” zoom in on section “sec4” to be sure that you select only the small section.

Now that you know how to create and assign motors, you are ready to learn how to control motors in the model logic.

Modeling motor failures

Like resources, motors can be taken down and brought up using the `take down` and `bring up` actions in the model logic. In example model 9.5, the model initialization function takes down two motors at the beginning of the simulation. The function also creates a dummy load and sends it to the process P_fail to simulate motor failures on the entrance lane.

The model initialization function is shown below:

```
begin model initialization function
   take down conv:M_slug
   take down conv:M_index
   create 1 load of type L_dummy to P_fail
   return true
end
```

Note When referring to movement system entities in logic, you must type the movement system name and entity name separated by a colon, for example, conv:M_slug where the system name is “conv” and the motor name is “M_slug”.

The motors M_slug and M_index are taken down at the beginning of the simulation to start slugging and indexing loads on the conveyor; modeling slugging and indexing conveyors is discussed in the next two sections in this chapter.

The P_fail process simulates motor failures by taking down and bringing up the horizontal entrance section’s motor (M_fail) in one-minute intervals throughout the simulation. Although not realistic, the one-minute cycles of up and down time allow you to easily verify the motor’s failures during the simulation. The P_fail arriving procedure is shown below:

```
begin P_fail arriving
   while 1=1 do begin
      wait for 1 min
      take down conv:M_fail
      wait for 1 min
      bring up conv:M_fail
   end
end
```

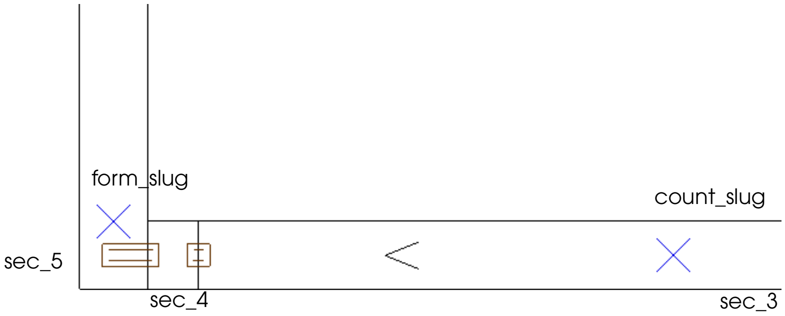
The P_fail arriving procedure includes a continuously repeating loop that delays for one minute (while the motor is up) and then takes down the motor “M_fail.” The procedure then delays for another minute (while the motor is down) to simulate the time to repair before bringing the motor back up; this loop repeats throughout the simulation.

Modeling slugging conveyors

Note

This approach for modeling slugging conveyors can only be used when forming a slug immediately before a side transfer. When forming slugs elsewhere in a conveyor system (for example, in the middle of a section or before an ahead transfer), you should use photoeyes or order lists to create the slugs. You will learn how to create slugs using order lists in chapter 13, “Indefinite Delays.” Photoeyes are not discussed in this textbook; for information on how to use photoeyes, refer to the “Conveyors” chapter in volume 2 of the *AutoMod User’s Manual*, online.

In example model 9.5, slugs consisting of 10 loads each are formed by drawing a small section of conveyor at the point where loads must accumulate, as shown below:



Creating slugs in example model 9.5

Slugs are formed by taking down the motor for section “sec_4,” which causes loads to accumulate before the side transfer (the motor is initially taken down at the beginning of the simulation in the model initialization function). As soon as 10 loads accumulate (forming a complete slug), the section motor is brought up and the slug transfers to section “sec_5” and travels to the inspection station. When the slug has completed transferring to the next conveyor section, the motor is taken down again to begin forming the next slug.

Important

Two stations are placed to control the formation of slugs. One station, “form_slug,” is placed immediately after the transfer to section “sec_5;” this station is placed as close to the transfer as possible to prevent extra loads from transferring with the slug. The other station, “count_slug,” is placed so that it allows only 10 loads (a complete slug) to accumulate between the station’s location and the transfer to section “sec_4.”

The logic that creates the slugs is defined in two arriving procedures. The first procedure is shown below:

```
begin P_move arriving
   move into conv:get_on
   travel to conv:count_slug
   send to P_slug
end
```

The P_move arriving procedure causes loads to get on the conveyor at station “get_on” and travel to station “count_slug.” As soon as loads arrive at the station “count_slug,” they are sent to the P_slug process and begin executing that process’ arriving procedure.

The P_slug arriving procedure uses the integer variable V_accum to count the number of loads that have accumulated in the current slug, as shown below:

```
begin P_slug arriving
   inc V_accum by 1
   if V_accum = 10 then
      begin
         bring up conv:M_slug
         set V_accum to 0
         set A_last = true
      end
   travel to conv:form_slug
   if A_last = true then take down conv:M_slug
   travel to conv:inspect
   send to P_inspect
end
```

If fewer than 10 loads have accumulated (V_accum is not equal to 10), then the procedure causes the current load to attempt to travel to the station “form_slug,” which is on the next section. Because the intervening section’s motor is down, the load accumulates and becomes part of the currently forming slug. If the current load is the tenth load in the slug (V_accum is equal to 10) then the following actions are performed:

- The load brings up the motor M_slug, which allows the slug to begin moving (loads in the slug start transferring to the vertical conveyor section).
- The variable V_accum is reset to zero (to begin counting the number of loads in the next slug).
- The load attribute A_last is set to true to indicate that the current load is the last load in the slug.

Note

The load attribute A_last is of type Integer. In the AutoMod software, integer values are often used as a flag to indicate whether something is true or false (similar to Boolean values in other programming languages). The syntax “true” and “false” can be used in place of the values “1” and “0,” respectively. For example,

```
set A_last = true
```

is the same as

```
set A_last = 1
```

In example model 9.5, the load attribute A_last is used as a flag to indicate whether or not the current load is the last load in a slug. By default, the value is initialized to false (0) for all loads in the model. The attribute value is set to true (1) only for the last load in each slug.

As soon as the section motor is brought up, each of the accumulated loads travel to station “form_slug” and then execute the `travel` action to travel to the inspection station.

Note

Because there is no delay between the two `travel` actions in the arriving procedure, loads do not stop at the station “form_slug” on their way to the inspection station.

The last load (identified by the value of the load attribute A_last) takes down the section motor when it arrives at the station “form_slug” (which is on the next section). Taking down the motor causes the trailing loads to stop before transferring, so the next slug begins accumulating.

Modeling indexing conveyors

In example 9.5, completed slugs travel on the conveyor to the station “inspect,” where they are inspected. After each inspection, the inspector advances the indexing conveyor to make just enough room for the inspected load. Loads on the indexing conveyor advance (one load at a time) until they reach the end of the conveyor and are removed from the system.

The logic used to model the inspection and indexing processes is shown below:

```
begin P_inspect arriving
   use R_inspect for e 10 sec
   clone 1 load to P_index nlt L_dummy
   travel to conv:goodbye
   send to die
end

begin P_index arriving
   bring up conv:M_index
   wait for 3 sec
   take down conv:M_index
   send to die
end
```

Loads that arrive at the station “inspect” are sent to the P_Inspect arriving procedure. After using the resource R_inspect for the required inspection time, each load clones a dummy load to the P_index arriving procedure, then executes an action to travel to the station “goodbye.”

Each cloned load that executes the P_index arriving procedure brings up the motor M_index for 3 seconds, which is long enough to let the original load move up 3 feet (the length of a single load) on the indexing section of conveyor. The cloned load then takes down the motor M_index and is sent to die.

Summary

This chapter introduced concepts that are extremely important to the simulation of real-world material handling systems.

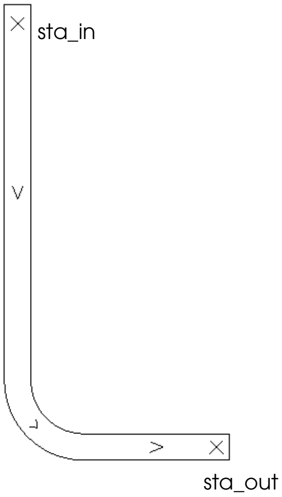
In this chapter, you learned how to sort multiple load types in a conveyor system. Sorting load types is necessary for modeling virtually any type of distribution center. You also learned how to customize section and transfer attributes to simulate many different types of conveyors. By changing a section’s width, velocity, accumulation, and so on, you can simulate almost any real-world conveyor system.

Finally, you learned how to use motors to control load movement on a conveyor. By taking down and bringing up motors, you can model failures, as well as slugging and indexing conveyor sections.

Exercises

Exercise 9.1

Consider the conveyor layout, shown below:



Assume the moving space of the continuous section is defined as:

0 × load length + 10 feet

Loads get on the conveyor at station “sta_in” and travel to station “sta_out,” where they leave the system. There are two types of loads in the system, L_long and L_short. The size of L_long loads is defined as:

X = 4
Y = 1
Z = 1

The size of L_short loads is defined as:

X = 2
Y = 3
Z = 1

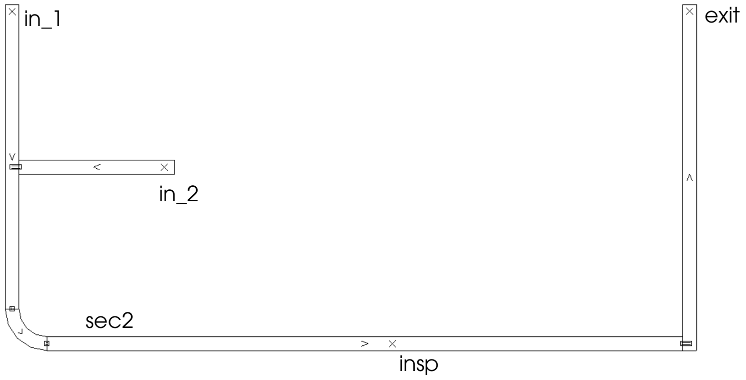
Determine the minimum possible space between two moving loads in each of the following instances:

Note: When answering these questions, measure the “space between loads” from the trailing edge of the preceding load to the leading edge of the following load.

a) Two L_long loads
b) Two L_short loads
c) An L_long load followed by an L_short load
d) An L_short load followed by an L_long load

Exercise 9.2

Consider the layout of example model 9.4, shown below:



Layout of example model 9.4

Assume that the horizontal section, “sec2,” is an accumulating section with stopping space defined as:

1.5 × load length + 0 feet

Loads of type “L_top” enter at station “in_1” and travel to station “exit.” The size of L_top loads is defined as:

X = 4
Y = 1
Z = 1

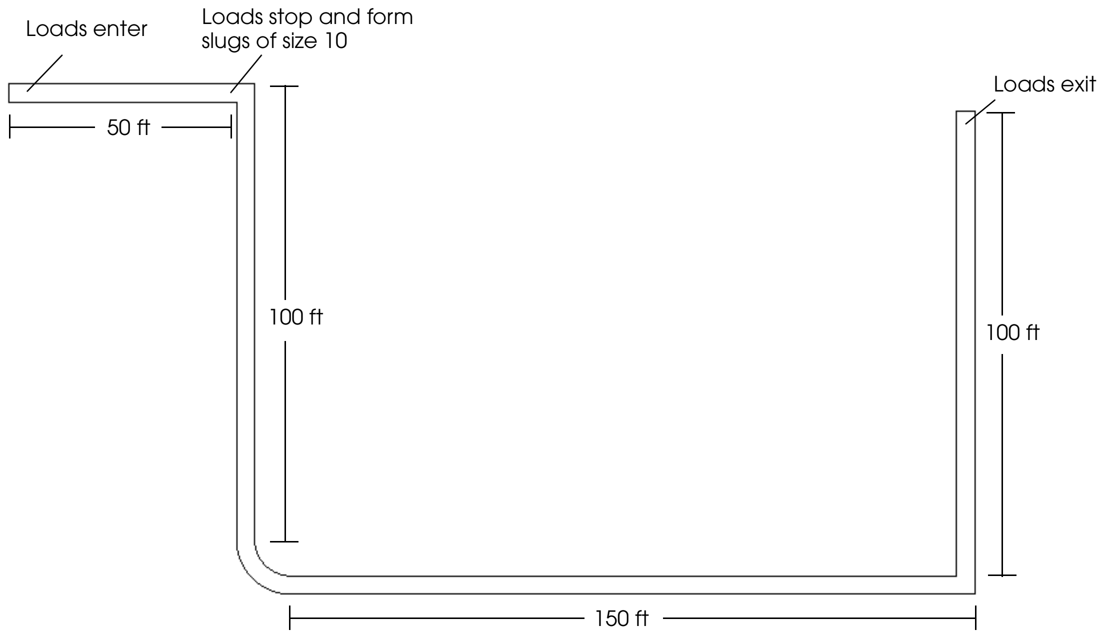
Loads of type “L_side” enter at station “in_2” and travel to station “exit.” The size of L_side loads is defined as:

X = 2
Y = 3
Z = 1

There are three loads of type L_top and two loads of type L_side accumulated at station “insp.” Assuming that no loads leave the station, determine the distance from the station to the leading edge of the next load that accumulates at the station.

Exercise 9.3

Given the conveyor layout shown below:



Loads enter at the beginning of the horizontal lane on the left-hand side and travel to the end of the lane where they form slugs of size 10. When a complete slug has formed, the slug travels on the conveyor to the exit at the end of the vertical lane on the right-hand side.

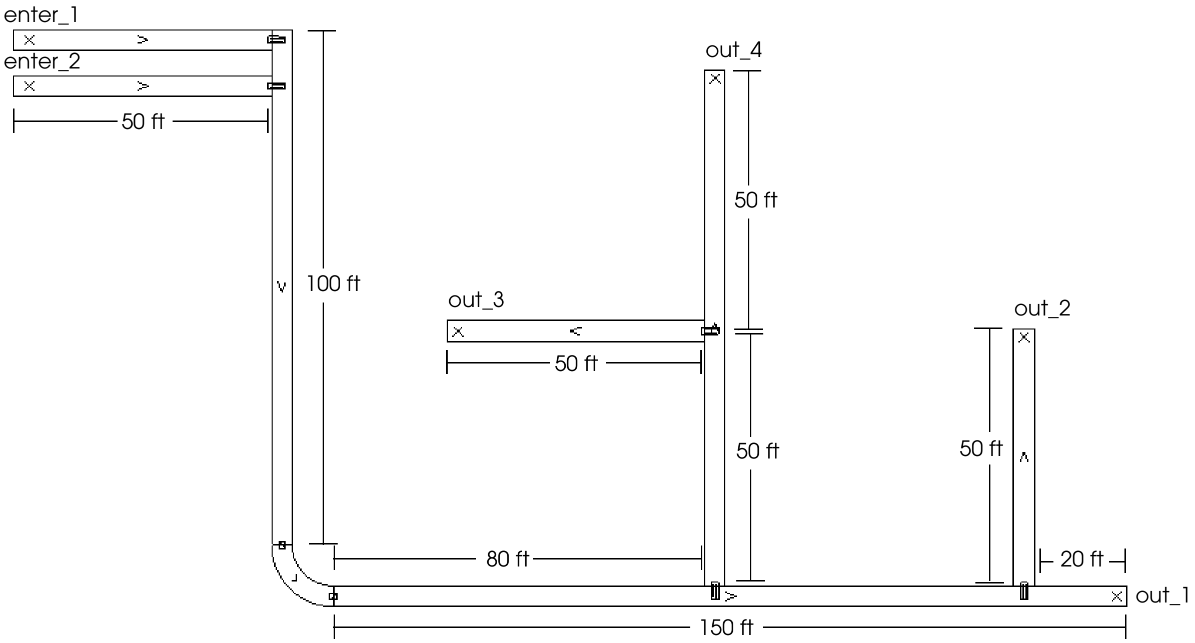
Loads are created every constant 20 seconds. The load size is defined as:

X = 2
Y = 3
Z = 1

Model this system. You need to determine the number and placement of conveyor entities in the model. Simulate the system for seven days.

Exercise 9.4

Given the conveyor system shown below:



Loads arrive at each of the two entrance stations on the left-hand side according to an exponential distribution with a mean of 20 seconds (loads are generated independently at each entrance). There are four different types of loads in the model, each with a unique color. The number of each type of load is randomly determined as follows: 30 percent are red, 30 percent are blue, 25 percent are yellow, and 15 percent are green.

Each load's size and destination station depends on its color, as shown in the table below:

Load color	Load size	Destination station

red	X = 4
	Y = 2
	Z = 1
		out_1

blue	X = 2
	Y = 4
	Z = 1
		out_2

yellow	X = 2
	Y = 3
	Z = 1
		out_3

green	X = 3
	Y = 4
	Z = 1
		out_4

Model this system and simulate for seven days.

Chapter 10

Intermediate Statistical Analysis

Chapter 10

Intermediate Statistical Analysis

In chapter 8, “Basic Statistical Analysis Using AutoStat,” you learned how to use a single scenario analysis in AutoStat to determine the confidence intervals of several responses, or statistics. In this chapter, you will learn about two new types of analyses that allow you to experiment with model input parameters, or factors, to perform “What if...” analyses.

Most simulation studies require the examination of multiple scenarios to determine which scenario is the “best,” based on measures such as cost versus payback in monetary terms, the time and resources required, and so on. Therefore, it is important to remember that building a model and validating it are only part of a simulation analyst’s job; performing experiments is another key part of the job. Make sure you allow enough time to conduct the analysis thoroughly to make sound decisions.

Experimenting with model scenarios

An experiment involves changing one or more parameters in your model, such as a processing time, the number of operators, the type of conveyor, and so on, to see its effect on the system. To change any model parameter in AutoStat, you must define the parameter as a factor. A **factor** is a model parameter that you want to change in an experiment. Once you have defined a model's factors, you define an analysis and view the responses, or statistics, in which you are interested.

The two types of analyses discussed in this chapter are:

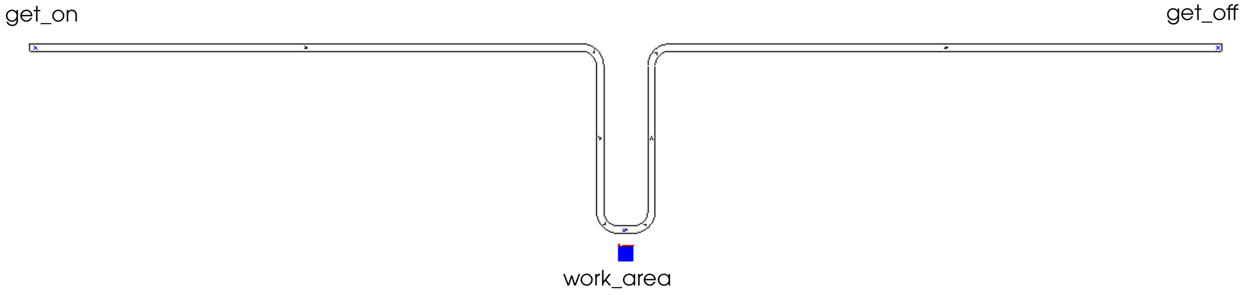
Vary one factor Analyze how the model behaves when you change the value of one factor while keeping other factors constant. For example, you can determine the effect of changing the speed of a conveyor system to determine whether the improved throughput outweighs the cost of the faster conveyor system.

Vary multiple factors Analyze how the model behaves when you change the value of several factors at the same time. For example, you can determine the effect of varying both the speed of a conveyor and the number of operators in the simulation.

This chapter explains how to define both "vary one factor" and "vary multiple factor" analyses (including how to define factors) and discusses how to interpret the results.

Example 10.1: Performing a financial analysis

A company is currently using the following conveyor system:



Example model 10.1: Layout

Products arrive in the system at a rate that is exponentially distributed with a mean of 2.4 minutes. Loads travel down a continuous conveyor whose speed is 1 foot per second. There is a piece of processing equipment, called "work_area," at the bottom of the "U" that processes loads on the conveyor in a constant 132 seconds.

The loads that are being processed have a cost of $1,000,000. The anticipated rate of return for the product is 14.4 percent. Therefore, each work in process (WIP) load in the system, on the average, translates into an annual carrying cost of $144,000. If the average number of loads in the system during the year is 8, then the annual cost of carrying WIP is $1,152,000.

The company wants to upgrade the system to reduce WIP and increase throughput. The company is evaluating whether to continue leasing the same type of processing equipment or whether to lease a faster model. The company could also upgrade the conveyor system to a faster one. Your job is to analyze whether a faster machine and/or a quicker conveyor are cost-effective ways to reduce WIP costs and increase throughput.

The equipment vendor has supplied specifications and operating costs for a one-year lease on several pieces of equipment:

Processing Time (Per Load)	Annual Cost
72 seconds	$2,000,000
84 seconds	$1,700,000
96 seconds	$1,400,000
108 seconds	$1,100,000
120 seconds	$ 800,000
132 seconds	$ 500,000 (current speed)

Equipment specifications

The slower the machine, the less expensive it is. Currently, the company is leasing the slowest model.

The conveyor vendor has provided the following annual quotes for three possible speeds of the conveyor:

Conveyor Speed	Annual Cost
1 foot/second	$800,000 (current speed)
1.5 feet/second	$1,200,000
2.0 feet/second	$1,600,000

Conveyor specifications

The slower the conveyor, the less expensive it is. Currently, the company is leasing the slowest conveyor.

In this example, you will use the AutoStat software to recommend a piece of equipment to lease and a speed of conveyor to lease, if any. You will base the analysis on five replications of the model, each for 10 simulated days of operation.

Compiling and setting up example model 10.1

In order to set up analyses for a model in AutoStat, you must first compile the model in AutoMod and use the AutoStat Setup wizard to define the warmup and snap information.

To compile example model 10.1:

Step 1 ***Import*** the *base* version of example model 10.1.

Step 2 From the Model menu, ***select*** Run AutoStat and ***click*** Yes to build the model. The AutoStat Model Setup wizard opens.

Step 3 ***Set up*** the model as follows:

- Model is random.
- Check for infinite loops, and stop any runs that take longer than one minute.
- No warmup required.
- Define a snap length of 10 days.

You are now ready to define the model’s factors, as well as the analyses to vary those factors.

Note The problem statement requires you to vary both the equipment processing time and the conveyor speed. At first, focus on varying the processing time. Once you have learned how to vary one factor, you will conduct another analysis to vary both factors.

Defining factors in AutoStat

Factors are model parameters that you want to vary in an analysis. Defining factors is similar to defining responses (see “Defining responses” on page 8.12 of the “Basic Statistical Analysis Using AutoStat” chapter).

There are four types of factors in AutoStat:

AutoMod A parameter of an entity that you want to vary in an analysis, such as the number of vehicles in a movement system, a resource’s processing time, or a conveyor section’s speed.

Combination The combination of two or more factors that you want to change simultaneously (not discussed in this textbook).

Data file cell A cell in an external data file (not discussed in this textbook).

Entire data file An entire data file being read by your model (not discussed in this textbook).

In this model, you are interested in looking at the processing time of the equipment and the speed of the conveyor, both of which you will define as AutoMod factors.

Defining processing time as a factor

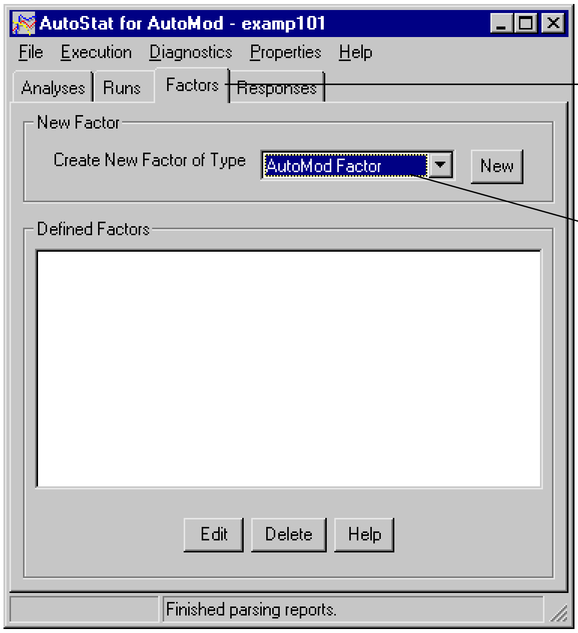
In this model, the equipment’s processing time is defined in a source file using a variable so that it can be defined as a factor in AutoStat (for more information about using variables instead of numbers for processing times, refer to “Defining variables” on page 7.4 of the “Advanced Process System Features” chapter). The source file is shown below:

```
begin P_process arriving
    move into Q_geton
    move into conv:get_on
    travel to conv:work_area
    use R_processor for V_proctime/* In seconds using the initial value */
    travel to conv:get_off
    send to die
end
```

The variable V_proctime, which is defined as type Time, has an initial value of 132 seconds, the slowest configuration possible. In order to test the other possible configurations listed in the “Equipment specifications” on page 10.5, you must define V_proctime as a factor in AutoStat.

To define V_proctime as an AutoMod factor:

Step 1 From the Factors tab, ***click*** New to define a new factor of type AutoMod Factor, as shown below:


AutoStat for AutoMod - examp101
File Execution Diagnostics Properties Help
Analyses Runs Factors Responses
New Factor
Create New Factor of Type
AutoMod Factor
New
Defined Factors
Edit Delete Help
Finished parsing reports.

Select the Factors tab

Define an AutoMod factor using the drop-down list

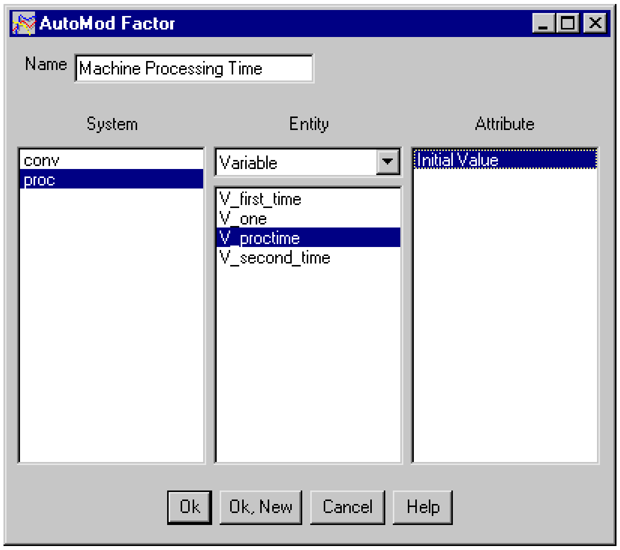
Defining an AutoMod factor

The AutoMod Factors window opens.

Step 2 ***Name*** the factor “Machine Processing Time”.

Step 3 From the System list, ***select*** proc.

Step 4 ***Select*** Variable from the Entity drop-down list and ***select*** V_proctime, as shown below:



Defining V_proctime as a factor

Step 5 The attribute Initial Value is already selected, so ***click*** OK.

The next step is to define an analysis to vary the equipment’s processing time to see the effect on the system.

Varying one factor in an analysis

The longer that equipment takes to process loads, the longer that loads are in the system. The more loads that are in the system and the longer that loads spend in the system, the greater the inventory costs. The question you need to answer is, “Which piece of equipment returns enough of an inventory savings to be cost-effective?”

In order to vary the equipment’s processing speeds over the possible values, you need to define a “vary one factor analysis,” then define responses to determine the effect that changing the processing time has on WIP levels and inventory cost.

To define a vary one factor analysis:

Step 1 From the Analysis Tab, ***select*** Vary One Factor from the Create New Analysis of Type drop-down list and ***click*** New. The Vary One Factor Analysis window opens.

Step 2 ***Name*** the analysis “Vary Processing Time”.

Leave the number of replications as 5, and use the default run control (no warmup, snap length of 10 days).

Step 3 In the Factors to Vary tab, ***select*** Machine Processing Time.

The values in the **Begin, End,** and **Increment** fields are the model’s default value of 132.0 seconds. These fields allow you to specify the **range of values** to use for the processing time during the analysis.

Tip ☞ You can define specific values rather than incremental values by selecting Individual Values from the “Set values using” drop-down list (not discussed in this textbook).

The time values you need to test are:

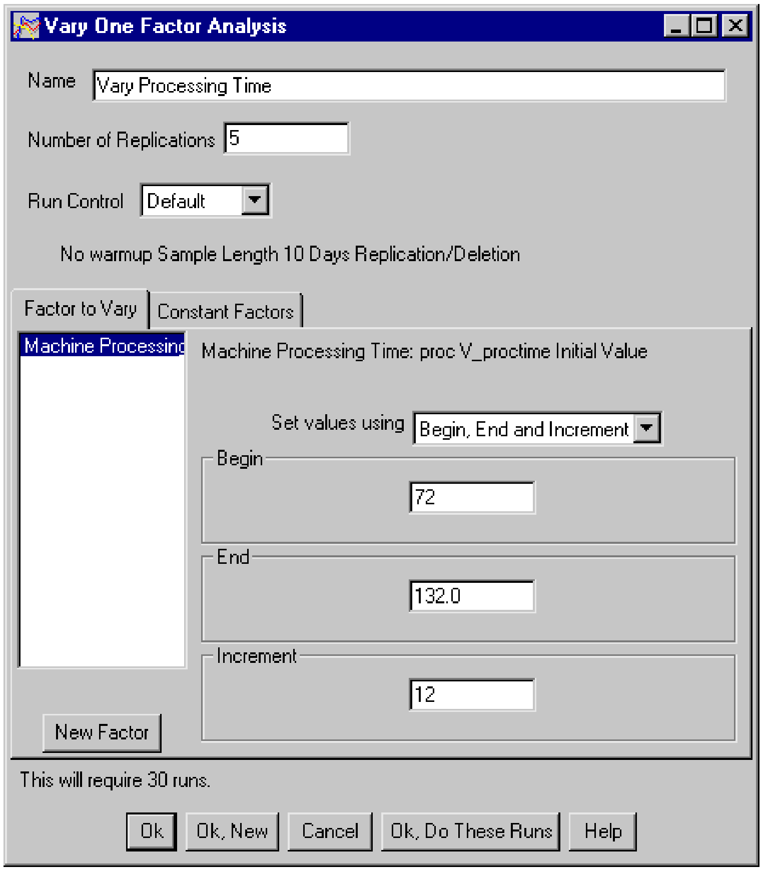
Processing Time (Per Load)	Annual Cost
72 seconds	$2,000,000
84 seconds	$1,700,000
96 seconds	$1,400,000
108 seconds	$1,100,000
120 seconds	$ 800,000
132 seconds	$ 500,000 (current speed)

Equipment specifications

The model is currently using the largest value of 132. Therefore, that will be the End value for the analysis. The smallest value in the specification is 72, which will be the Begin value. The values are 12 seconds apart, so the increment will be 12.

Step 4 ***Type*** “72” as the Begin value.

Step 5 ***Type*** “12” as the Increment value. Your analysis should look like the following:



Vary One Factor Analysis

Name: Vary Processing Time

Number of Replications: 5

Run Control: Default

No warmup Sample Length 10 Days Replication/Deletion

Machine Processing Time: proc V_proctime Initial Value

Set values using: Begin, End and Increment

Begin: 72

End: 132.0

Increment: 12

This will require 30 runs.

Vary one factor analysis

Because the number of replications is five, and there are six time values to test for the factor, this analysis is going to require 30 runs, as shown at the bottom of the window.

Step 6 ***Click*** OK, Do These Runs. The runs begin.

While the model is running, you are going to define average WIP as a response so you can compare the different scenarios.

Defining average WIP as a response

To determine the effect of the various machine processing times, define the WIP level as an AutoMod response. Later, you will learn how to calculate the cost and return of each scenario using a combination response.

To define average WIP as a response:

Step 1 From the Responses tab, ***click*** New to define an AutoMod response. The AutoMod Response window opens.

Step 2 ***Name*** the response “Average WIP”.

Step 3 ***Select*** the System proc, the Entity P_process, and the Statistic Ave (Average) to report the average number of loads in the process. In this model, this statistic also represents the average number of loads in the system, because there is only one process in the model.

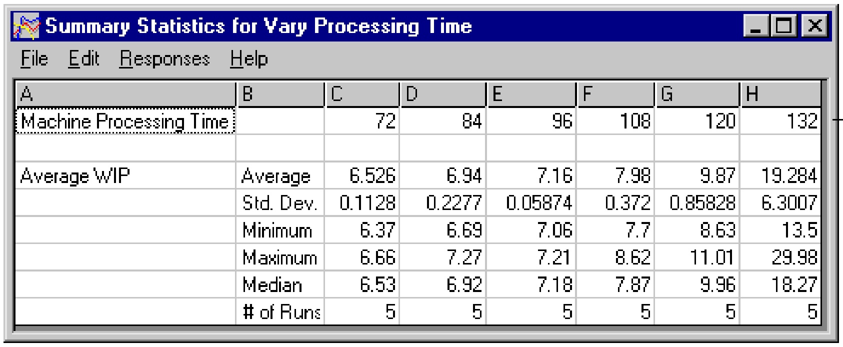
Step 4 ***Click*** OK.

Viewing statistics for WIP levels

Once your response is defined and your runs have finished, you can view various types of output. All output is calculated from the report files of each run made.

To see the effect that processing time has on average WIP, look at summary statistics and confidence intervals.

Step 1 From the Analysis tab, ***expand*** the Vary Processing Time analysis and ***double-click*** Summary Statistics.



A	B	C	D	E	F	G	H
Machine Processing Time		72	84	96	108	120	132
Average WIP	Average	6.526	6.94	7.16	7.98	9.87	19.284
	Std. Dev.	0.1128	0.2277	0.05874	0.372	0.85828	6.3007
	Minimum	6.37	6.69	7.06	7.7	8.63	13.5
	Maximum	6.66	7.27	7.21	8.62	11.01	29.98
	Median	6.53	6.92	7.18	7.87	9.96	18.27
	# of Runs	5	5	5	5	5	5

Processing times are listed across the top

Summary Statistics for Machine Processing Time

The average amount of WIP in the system increases as the processing time increases, which is logical. At the fastest value of 72 seconds, WIP averages 6.526 loads. For the slowest time (132 seconds), WIP averages 19.284 loads.

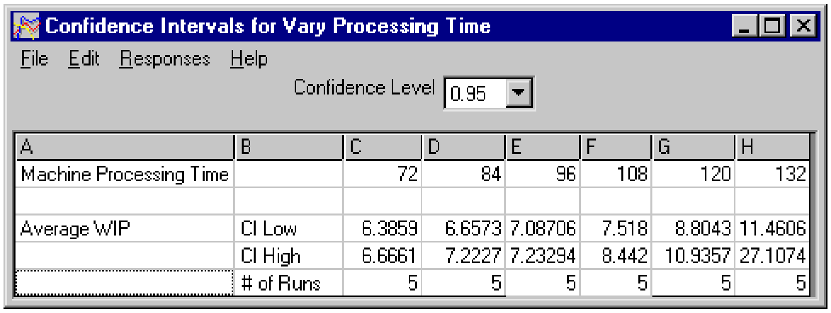
The difference between the minimum and maximum loads in the system varies widely depending on the processing time. For the fastest processing time of 72 seconds, the range is 6.37 to 6.66 loads, a very narrow range. But at 132 seconds, the range is 13.5 to 29.98 loads, with a standard deviation of 6.3007 loads.

Step 2 ***Close*** the Summary Statistics window.

Now view the confidence intervals for the Average WIP response:

Step 1 From the Analysis tab, ***double-click*** Confidence Intervals.

Step 2 ***Select*** 0.95 from the Confidence Level drop-down list to view the 95 percent intervals, as shown below:



Confidence Level 0.95

A	B	C	D	E	F	G	H
Machine Processing Time		72	84	96	108	120	132
Average WIP	CI Low	6.3859	6.6573	7.08706	7.518	8.8043	11.4606
	CI High	6.6661	7.2227	7.23294	8.442	10.9357	27.1074
	# of Runs	5	5	5	5	5	5

Confidence intervals for Machine Processing Time

The longer the processing times, the larger (wider) the confidence intervals, because the standard deviation is greater (as shown in the summary statistics).

Step 3 ***Close*** the Confidence Intervals window.

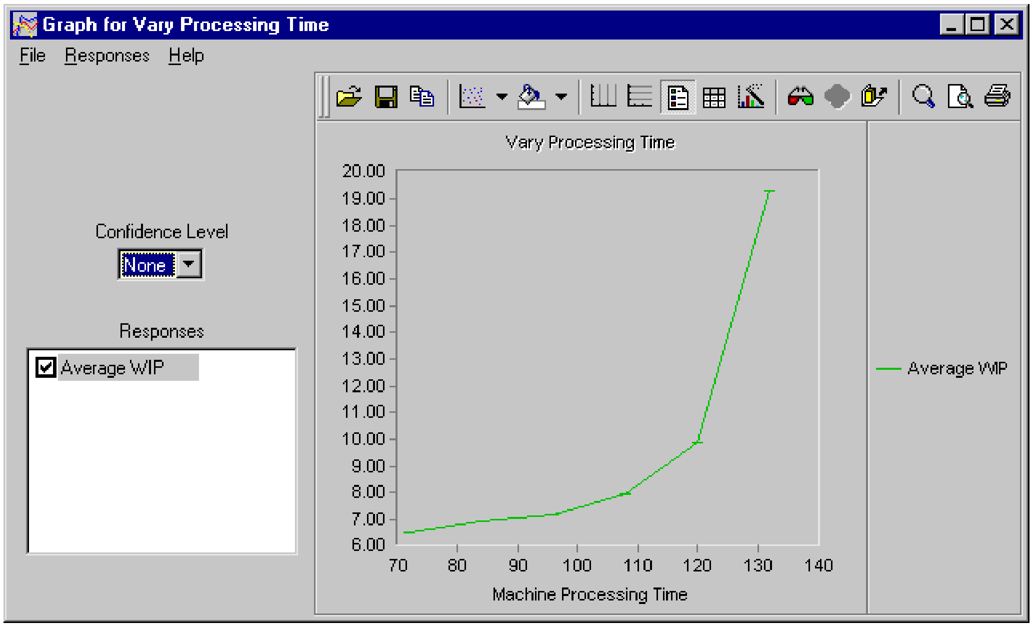
Viewing a line graph

In addition to the summary statistics and other types of tabular results, AutoStat provides graphs for your responses.

To view a line graph for the Average WIP response:

Step 1 In the Analysis tab, ***double-click*** Line Graph. A blank graph opens.

Step 2 ***Select*** Average WIP in the Responses list.



Average WIP line graph

Notice that the graph is automatically scaled along the X axis to the processing time values used in the analysis. The Y axis is scaled to the average number of WIP loads in the system.

Tip

To change many attributes of the graph, including its scale, color, and so on, right-click on the graph and select Properties. Select Help from the graph’s Help menu to learn more about the properties of the graph and how to edit them.

Step 3 ***Close*** the line graph.

Comparing all scenarios to one scenario

In this analysis, you are testing six different values for the machine's processing time. How do you compare the different scenarios to tell whether one is better than another? The vary one factor analysis, as well as the vary multiple factor analysis (discussed later in this chapter), provide output that compares each scenario to the others.

The scenario comparison uses confidence intervals to determine whether the average difference in response values for the two scenarios is statistically significant. If the confidence interval spans zero (for example, the low value is –6 and the high value is 4), there is no significant difference between the two response values. **If the interval does not span zero, there is a statistically significant difference between the two values** (either positive or negative) at the level of confidence selected.

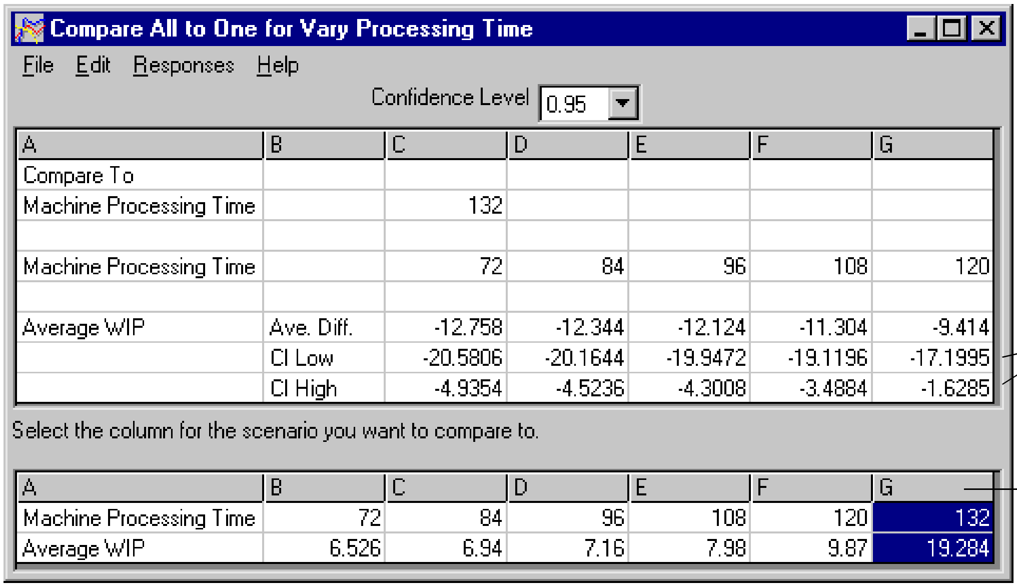
In this example model, the base scenario uses equipment that processes in 132 seconds. Therefore, it would be helpful to compare all the other scenarios to that one.

To compare scenarios:

Step 1 In the Analysis tab, ***double-click*** Compare All to One. The Compare All to One window opens.

Step 2 From the Confidence Interval drop-down list, ***select*** a confidence level of 0.95 (95 percent).

Step 3 In the lower half of the window, ***select*** the right-most scenario (column G), which is the 132-second processing time scenario. The comparison information appears in the upper part of the window, as shown below:



Confidence Level 0.95

A	B	C	D	E	F	G
Compare To						
Machine Processing Time		132				
Machine Processing Time		72	84	96	108	120
Average WIP	Ave. Diff.	-12.758	-12.344	-12.124	-11.304	-9.414
	CI Low	-20.5806	-20.1644	-19.9472	-19.1196	-17.1995
	CI High	-4.9354	-4.5236	-4.3008	-3.4884	-1.6285

Select the column for the scenario you want to compare to.

A	B	C	D	E	F	G
Machine Processing Time	72	84	96	108	120	132
Average WIP	6.526	6.94	7.16	7.98	9.87	19.284

None of the intervals spans zero

Select the base scenario here to display the comparison data for the response

Compare All to One

Notice that none of the intervals (CI Low and CI High) spans zero, indicating that compared to the 132-second scenario, all the other scenarios have a statistically significant effect on average WIP levels. The average difference of the response for each scenario is negative, indicating that all other scenarios result in lower WIP levels than the base scenario.

Note To see a comparison with a confidence interval that spans zero, select 96 as the base scenario and look at the interval for 84 seconds.

Step 4 ***Close*** the Compare All to One window.

The next section explains how to determine which of the scenarios is the most feasible financially.

Analyzing financial payback

As stated in “Example 10.1: Performing a financial analysis” on page 10.4, each WIP load, on the average, translates into a carrying cost of $144,000 per year. In addition, each piece of equipment has a cost, as shown in “Equipment specifications” on page 10.5. We can use this information to calculate the cost of new equipment.

To calculate the cost manually, the formula is:

$$\text{Total Cost} = \$144{,}000 \times \text{Avg. WIP} + \text{Equipment cost}$$

Using this formula, you can calculate the cost of each scenario, as shown in the following table:

Processing Time (Seconds)	Average WIP Level	Average WIP Cost (WIP x $144,000)	Equipment Cost (From Specification)	Total Cost (Average WIP Cost + Equipment Cost)
72	6.526	$939,744	$2,000,000	$2,939,744
84	6.94	$999,360	$1,700,000	$2,699,360
96	7.16	$1,031,040	$1,400,000	$2,431,040
108	7.98	$1,149,120	$1,100,000	$2,249,120
120	9.87	$1,421,280	$800,000	$2,221,280
132 (base)	19.284	$2,776,896	$500,000	$3,276,896

Total cost of equipment

The machine with the lowest total cost ($2,221,280) is the machine with the processing time of 120 seconds.

Now use AutoStat to perform this calculation for you. Because several responses and calculations are involved, you will need to use a combination response.

Defining a combination response to show total cost

A combination response adds together several factors and/or responses, which can each be manipulated using **weights**, or multipliers (see “Weighting terms in a combination response” on page 8.19 of the “Basic Statistical Analysis Using AutoStat” chapter for more information).

As stated previously, the formula to determine the incremental cost of new equipment is:

$$\text{Total Cost} = \$144{,}000 \times \text{Avg. WIP} + \text{Equipment cost}$$

The first part of this equation is easy to represent in AutoStat, because the average WIP level is already defined as a response for the model. Therefore, you can use a multiplier of $144,000 in the response to calculate that part of the equation.

However, the equipment cost is not currently part of the model. Also, the response must determine the difference between any two of the scenarios. Therefore, you must determine how to include the equipment cost and calculate the incremental difference in the scenarios.

Calculating equipment costs and scenario differences with an equation

In order to determine which scenario is better, you need to compare each of the scenarios to one of them (a base scenario) and determine what difference the change in processing time has on the total cost. To make the calculations easy, use the 72-second processing time as the base scenario (see “Total cost of equipment” on page 10.14 for a table of the scenarios).

To describe the incremental cost between the base scenario and any other scenario, you can use the following formula:

$$\text{Equipment cost} = \text{Cost of base equipment} - (\text{T} - 72) \times \frac{\text{Change in equipment cost per scenario}}{\text{Change in processing time per scenario}}$$

Where T equals the processing time of the scenario that you are comparing to the base scenario.

The last term of the equation, which involves the changes in cost and processing time per scenario, can be represented by a constant value, because the change in equipment cost between each scenario is $300,000, and the change in the processing time between each scenario is 12 seconds:

$$\frac{\$300{,}000}{12} = \$25{,}000$$

We also know that the cost of the base equipment is $2,000,000. So:

$$\text{Equipment cost} = \$2{,}000{,}000 - (\text{T} - 72) \times \$25,\ 000$$

To simplify $(\text{T} - 72) \times \$25,\ 000$, multiply both T and 72 by $25,000, resulting in:

$$\text{Equipment cost} = \$2{,}000{,}000 - (\$25{,}000\text{T} - \$1{,}800{,}000)$$

Further simplification results in:

$$\mathbf{Equipment\ cost = \$3{,}800{,}000 - \$25{,}000T}$$

To verify this equation, solve for $\text{T} = 84$ seconds:

$$\$3{,}800{,}00 - \$25{,}000(84) = \$1{,}700{,}000$$

The resulting cost matches the vendor specification. So if:

$$\text{Total Cost} = \$144{,}000 \times \text{Avg. WIP} + \text{Equipment cost}$$

Then:

$$\mathbf{Total\ Cost = \$144{,}000 \times Avg.\ WIP + (3{,}800{,}000 - 25{,}000T)}$$

Defining the cost equation in a combination response

Now you must define a response to represent the formula:

Total Cost = ($144,000 × Average WIP) + $3,800,000 – $25,000T

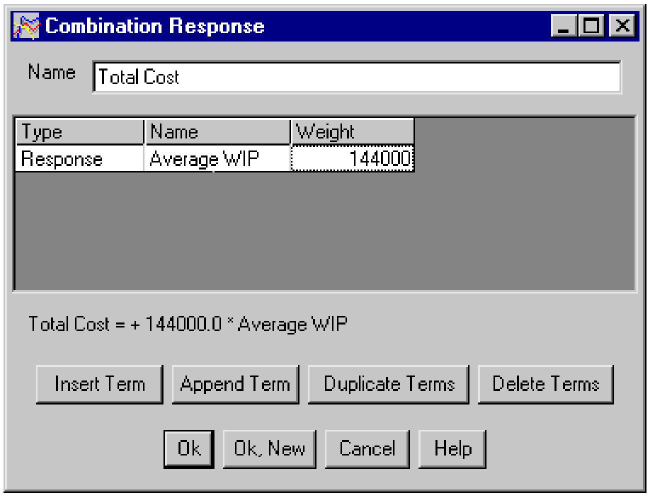
To define the combination response:

Step 1 From the Responses tab, ***select*** Combination Response from the Create New Response of Type drop-down list and ***click*** New. The Combination Response window opens.

Step 2 ***Name*** the response “Total Cost”.

Define the first half of the equation (the cost of average WIP) first. Average WIP is already defined as a response, so simply adjust the Weight value to represent the dollar cost of each WIP load.

Step 3 ***Double-click*** the Weight column and ***type*** “144000,” as shown below:



Name: Total Cost

Type	Name	Weight
Response	Average WIP	144000

Total Cost = + 144000.0 * Average WIP

Defining the cost of WIP in the combination response

The second part of the equation must contain the formula for the relative difference in value for each scenario. The processing time of the scenario (T) is defined as a factor, so we can use a weight of –25,000 to represent that term. But the $3.8 million cost is not part of the model.

To use values that are not included in the model, you must define the values as a Weight for a factor or response that *is* included in the model. Therefore, the model uses a “dummy” variable, V_one, that you define as a factor and initialize to 1. Setting the weight to 3,800,000 and multiplying it by 1 results in 3,800,000 being added to the equation.

To define V_one’s initial value as a factor:

Step 1 ***Move*** the Combination Response window aside.

Step 2 From the Factor tab, ***select*** AutoMod Factor from the Create New Factor of Type drop-down list and ***click*** New.

Step 3 ***Name*** the factor “V_one initial value”.

Step 4 ***Select*** proc as the system.

Step 5 ***Select*** Variable from the Entity drop-down list and ***select*** V_one as the entity.

Step 6 ***Select*** Initial Value as the attribute.

Step 7 ***Click*** OK.

Now finish editing the combination response:

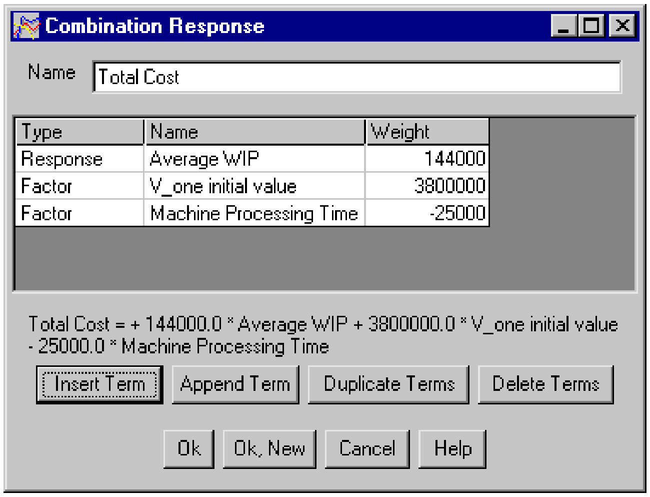
Step 1 In the Combination Response window, ***click*** Append Term to add a new line to the response.

Step 2 ***Double-click*** the new line's Type cell and ***select*** Factor from the drop-down list.

Step 3 ***Double-click*** the Name cell and ***select*** "V_one initial value" from the drop-down list.

Step 4 ***Double-click*** the Weight column and ***type*** "3800000." ***Press*** Tab.

Step 5 ***Append*** another term and ***define*** it as the factor Machine Processing Time with a weight of "–25000," as shown below:



Name: Total Cost

Type	Name	Weight
Response	Average WIP	144000
Factor	V_one initial value	3800000
Factor	Machine Processing Time	-25000

Total Cost = + 144000.0 * Average WIP + 3800000.0 * V_one initial value - 25000.0 * Machine Processing Time

Defining the equipment cost in a combination response

The formula at the bottom of the window verifies that the response is defined correctly.

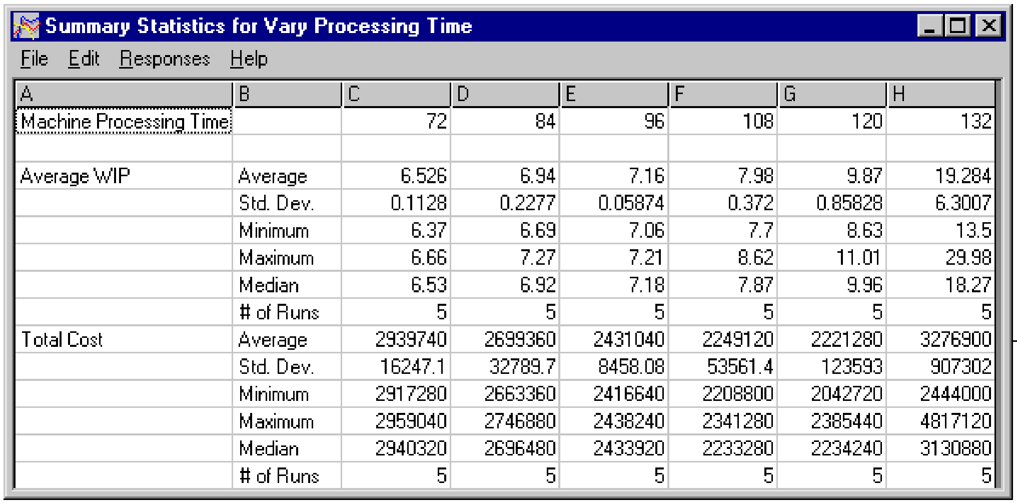
Step 6 ***Click*** OK.

Viewing the summary statistics for the Total Cost response

Now that the response is defined, you can view the Total Cost response using any of the types of output, including summary statistics, confidence intervals, line graphs, and so on.

To view summary statistics for Total Cost:

Step 1 From the Analysis tab, ***expand*** the Vary Processing Time analysis and ***double-click*** Summary Statistics. The Summary Statistics window opens.



A	B	C	D	E	F	G	H
Machine Processing Time		72	84	96	108	120	132
Average WIP	Average	6.526	6.94	7.16	7.98	9.87	19.284
	Std. Dev.	0.1128	0.2277	0.05874	0.372	0.85828	6.3007
	Minimum	6.37	6.69	7.06	7.7	8.63	13.5
	Maximum	6.66	7.27	7.21	8.62	11.01	29.98
	Median	6.53	6.92	7.18	7.87	9.96	18.27
	# of Runs	5	5	5	5	5	5
Total Cost	Average	2939740	2699360	2431040	2249120	2221280	3276900
	Std. Dev.	16247.1	32789.7	8458.08	53561.4	123593	907302
	Minimum	2917280	2663360	2416640	2208800	2042720	2444000
	Maximum	2959040	2746880	2438240	2341280	2385440	4817120
	Median	2940320	2696480	2433920	2233280	2234240	3130880
	# of Runs	5	5	5	5	5	5

The average Total Cost that AutoStat calculates for each scenario matches the manual calculations

Summary statistics for Total Cost

The average Total Cost for each scenario in the summary statistics matches the calculations performed manually (see "Total cost of equipment" on page 10.14).

Note Results vary slightly due to rounding differences.

The summary statistics confirm that the lowest cost configuration is 120 seconds, with a cost of $2,221,280.

Step 2 ***Close*** the Summary Statistics window.

Now that you have analyzed the effect of varying the equipment processing times, it is time to evaluate the effect of varying both the equipment processing time and the conveyor speed using a vary multiple factor analysis.

Varying multiple factors in an analysis

In the last analysis, you varied one factor: the processing time. However, the problem statement requires you to consider the effect of changing the conveyor speed, as well. To analyze the interaction of both changes, you must define a new factor for the conveyor speed (you can reuse the factor you have already defined for the machine's processing time). You must also define a new analysis of type vary multiple factors and create a new response to measure how the two factors influence WIP costs.

Defining conveyor speed as a factor

To define conveyor speed as a factor:

Step 1 From the Factors tab, ***select*** AutoMod Factor from the Create New Factor of Type drop-down list and ***click*** New.

Step 2 ***Name*** the factor "Conveyor Velocity".

Step 3 ***Select*** "conv" as the system.

Step 4 From the Entity drop-down list, ***select*** Section Type and ***select*** Default Section in the Entity list.

Step 5 ***Select*** Velocity as the attribute.

Step 6 ***Click*** OK.

Now you are ready to define a "vary multiple factor" analysis.

Defining a vary multiple factors analysis

The conveyor vendor has supplied specifications for systems that have speeds of 1 foot per second, 1.5 feet per second, and 2 feet per second (see "Conveyor specifications" on page 10.5 for more information). You want to vary the velocity and the processing time.

To define a multiple factor analysis:

Step 1 From the Analysis tab, ***select*** Vary Multiple Factors from the Create New Analysis of Type drop-down list and ***click*** New. The Vary Multiple Factors window opens.

Step 2 ***Name*** the analysis "Vary Speed and Processing Time".

Leave the number of replications as 5, and use the default run control (the same one you used for the single-factor analysis).

You need to set the possible values for both the processing time and the conveyor speed.

Step 3 In the Factors to Vary tab, ***select*** Machine Processing Time and ***define*** the processing times as you did in the last analysis, with a beginning value of 72 seconds, an ending value of 132 seconds, and an increment of 12 seconds.

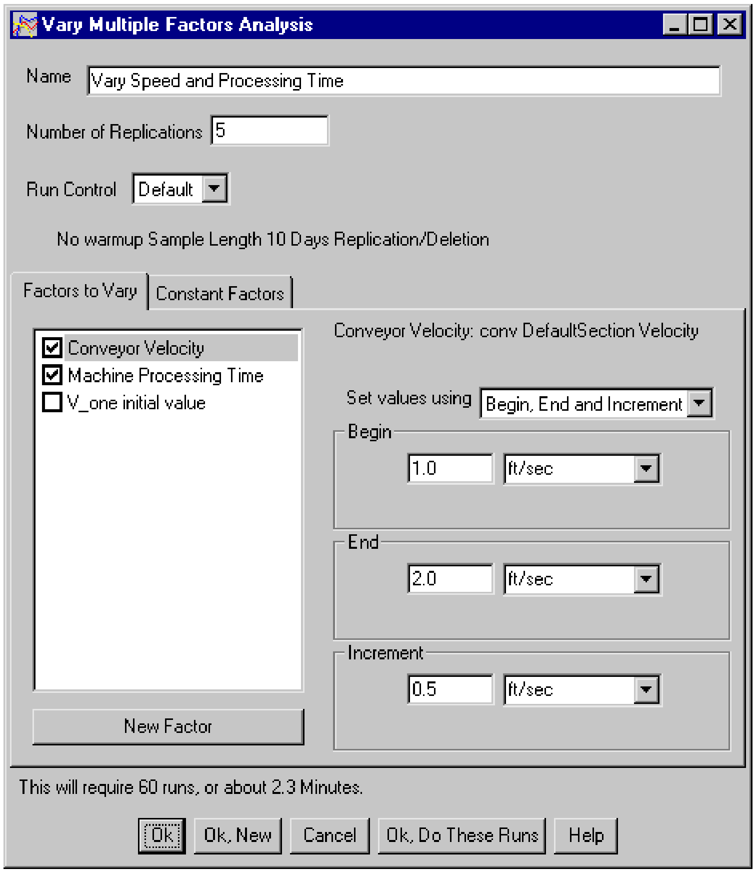
Step 4 In the Factors to Vary tab, ***select*** Conveyer Velocity. The default velocity of 1.0 foot per second appears as the beginning, ending, and increment value.

You need to vary the speed to 1.5 feet per second and 2.0 feet per second.

Step 5 ***Type*** "2.0" in the End field.

Step 6 ***Type*** ".5" in the Increment field.

Your analysis should look like the following:



Name: Vary Speed and Processing Time

Number of Replications: 5

Run Control: Default

No warmup Sample Length 10 Days Replication/Deletion

☑ Conveyor Velocity
☑ Machine Processing Time
□ V_one initial value

Conveyor Velocity: conv DefaultSection Velocity

Set values using: Begin, End and Increment

Begin: 1.0 ft/sec

End: 2.0 ft/sec

Increment: 0.5 ft/sec

This will require 60 runs, or about 2.3 Minutes.

Vary multiple factors analysis

Notice that AutoStat is reporting that this analysis requires 60 runs. There are six processing times and three conveyor speeds, which equal 18 possible scenarios. In addition, you want five replications of each. Eighteen multiplied by 5 is 90. Why then does AutoStat need to make only 60 runs? AutoStat can use the runs made for your previous analysis, because those runs already use one full set of processing times with the base speed of one foot per second. Therefore, only the scenarios that involve different conveyor speeds need to be run, reducing the number of runs required to 60.

Note AutoStat estimates the time required to make runs based on the time it took to make previous runs. The time required varies based on processor speed.

Step 7 ***Click*** OK, Do These Runs.

Defining a combination response to show the revised total cost

You need to define a new response for the combined incremental cost of the equipment and the conveyor. The conveyor costs are shown below:

Conveyor Speed	Cost
1 foot/second	$800,000 (current speed)
1.5 feet/second	$1,200,000
2.0 feet/second	$1,600,000

Conveyor specifications

The increased cost for each foot per second is $800,000. To calculate the cost of *both* equipment and conveyors as compared to the base scenario, you need to add the cost of the conveyors to the response you defined for total cost of the equipment and adjust the formula to calculate the difference from the base conveyor cost, as shown below:

$$\text{Revised Total Cost} = \text{Total Cost} - \text{Base Conveyor Cost} + (\text{Change in Conveyor Cost} \times \text{Conveyor Speed})$$

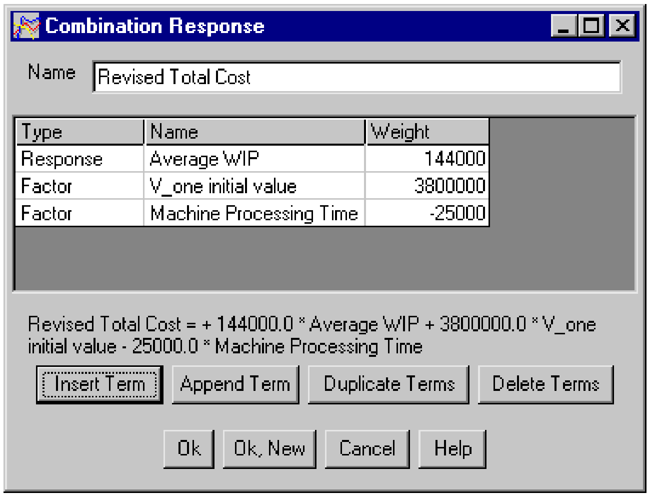
The base conveyor cost is $800,000.

To define this formula as a combination response:

Step 1 From the Responses tab, ***select*** Combination Response from the Create New Response of Type drop-down list and ***click*** New. The Combination Response window opens.

Step 2 ***Name*** the response “Revised Total Cost.”

Step 3 ***Define*** the equipment cost the same way that you did in the Total Cost response, as shown below:



Name: Revised Total Cost

Type	Name	Weight
Response	Average WIP	144000
Factor	V_one initial value	3800000
Factor	Machine Processing Time	-25000

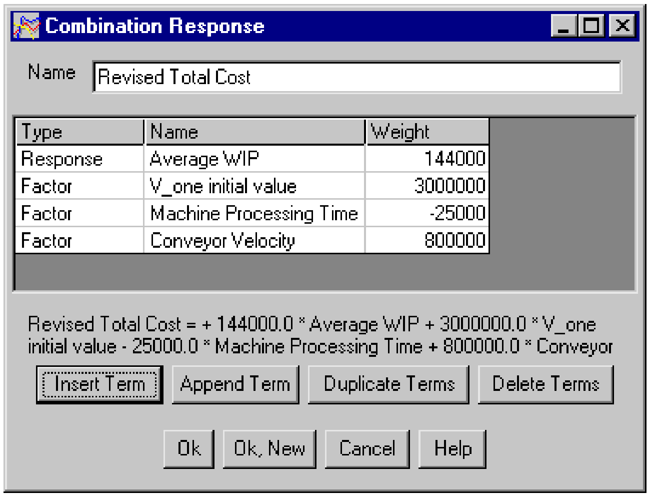
Revised Total Cost = + 144000.0 * Average WIP + 3800000.0 * V_one initial value - 25000.0 * Machine Processing Time

Creating a revised total cost response

This gives us the first part of the equation (Total Cost). Now we need to subtract a constant 800,000 for the base cost of the conveyor. Because the variable V_one initial value is already acting as a constant in this formula (see “Defining the cost of WIP in the combination response” on page 10.16), we can edit its value and subtract 800,000 (rather than defining another dummy variable for the second constant value).

Step 4 ***Double-click*** the Weight of V_one initial value and ***type*** “3000000.”

Step 5 ***Append*** a term and ***define*** the factor Conveyor Velocity with a Weight of 800000 (the change in conveyor cost). ***Press*** Tab.



Name: Revised Total Cost

Type	Name	Weight
Response	Average WIP	144000
Factor	V_one initial value	3000000
Factor	Machine Processing Time	-25000
Factor	Conveyor Velocity	800000

Revised Total Cost = + 144000.0 * Average WIP + 3000000.0 * V_one initial value - 25000.0 * Machine Processing Time + 800000.0 * Conveyor

The Revised Total Cost response

Step 6 ***Click*** OK.

Viewing a bar graph for the Revised Total Cost response

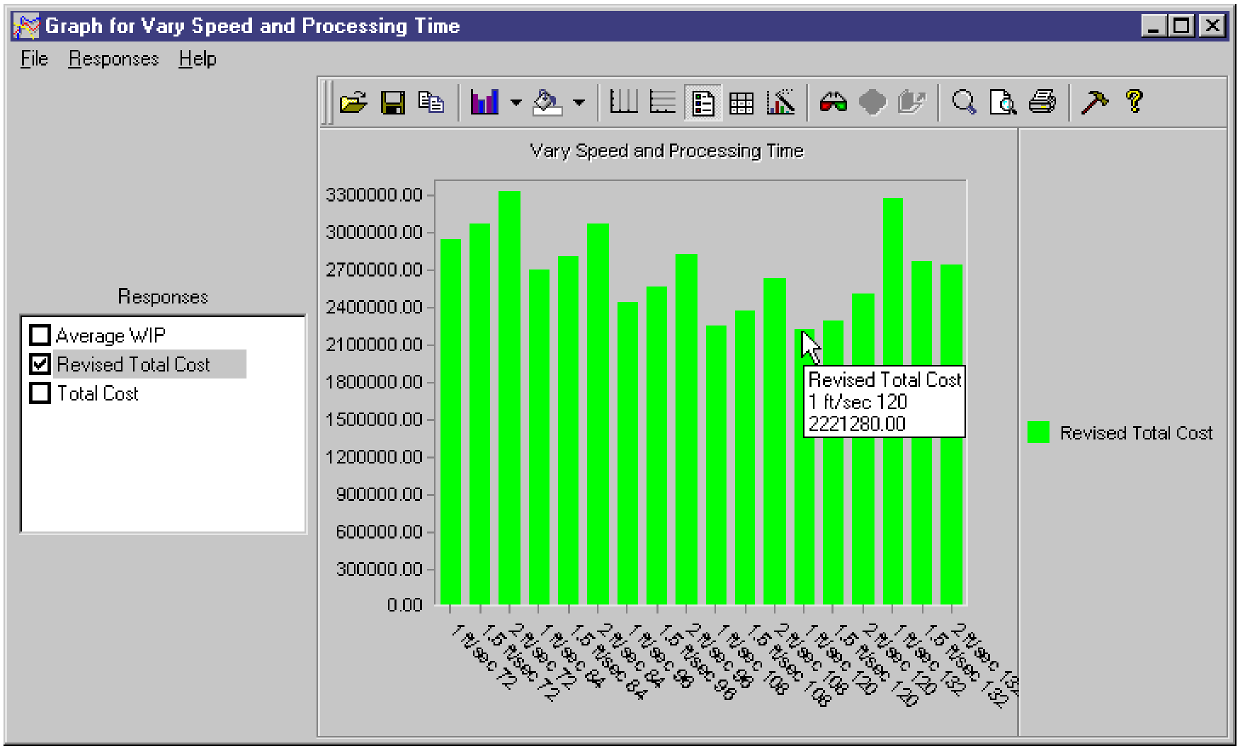
Once runs have finished, you can view the output for the newly created response using a new type of output: a bar graph.

To view a bar graph:

Step 1 From the Analysis tab, ***expand*** the Vary Speed and Processing Time analysis and ***double-click*** Bar Graph. A blank graph opens.

Step 2 ***Select*** Revised Total Cost in the Responses list. The response values are graphed.

Tip Hold the mouse pointer over any bar to view its values.



Responses

☐ Average WIP
☑ Revised Total Cost
☐ Total Cost

Revised Total Cost
1 ft/sec 120
2221280.00

Bar graph for Revised Total Cost

In this graph, the scenario values (equipment processing time and conveyor speed) are graphed along the X axis, and the response (cost) is graphed along the Y axis. The shortest bar, which indicates the lowest cost, is for the scenario of 120 seconds and 1.0 foot per second, so the choice for best processing time has not changed.

Step 3 ***Close*** the graph.

When looking at multiple factors together, it can be useful to compare different combinations of factor values. The multiple factor graph is useful for comparing responses against different sets of factor values.

Viewing the multiple factor graph

The multiple factor graph illustrates the effect that varying one factor’s value while holding another factor’s value constant has on a response. For example, look at how the total cost changes when the conveyor speed is held constant at the base level of 1 foot per second and the processing time changes.

To view the Multiple Factor graph:

Step 1 In the Analysis tab, ***expand*** the Vary Speed and Processing Time analysis and ***double-click*** Multiple Factor Graph. A blank graph opens.

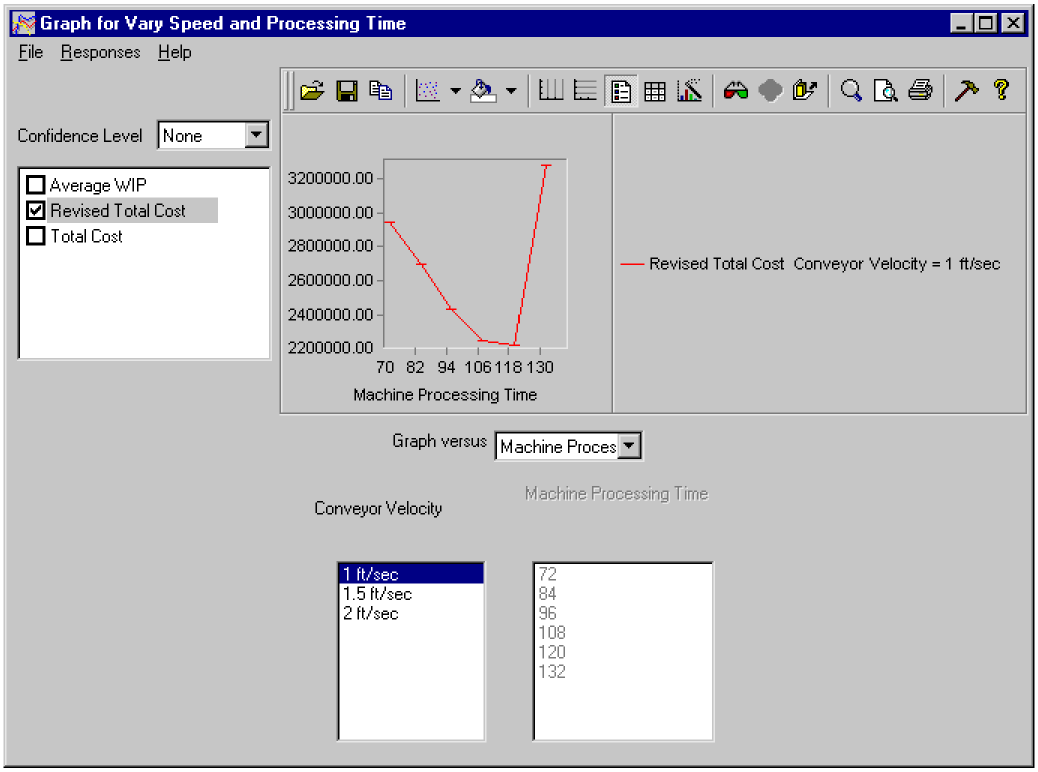
Step 2 ***Select*** Revised Total Cost in the Responses list.

The factor selected in the “Graph versus” list is the factor that is varied.

Step 3 From the “Graph versus” drop-down list, ***select*** Machine Processing Time.

Now select a value for the conveyor velocity against which to graph equipment processing time.

Step 4 ***Select*** 1 ft/sec in the Conveyor Velocity list. The response values are graphed.



Confidence Level: None

☐ Average WIP
☑ Revised Total Cost
☐ Total Cost

Graph versus: Machine Proces

Conveyor Velocity

1 ft/sec
1.5 ft/sec
2 ft/sec

Machine Processing Time

72
84
96
108
120
132

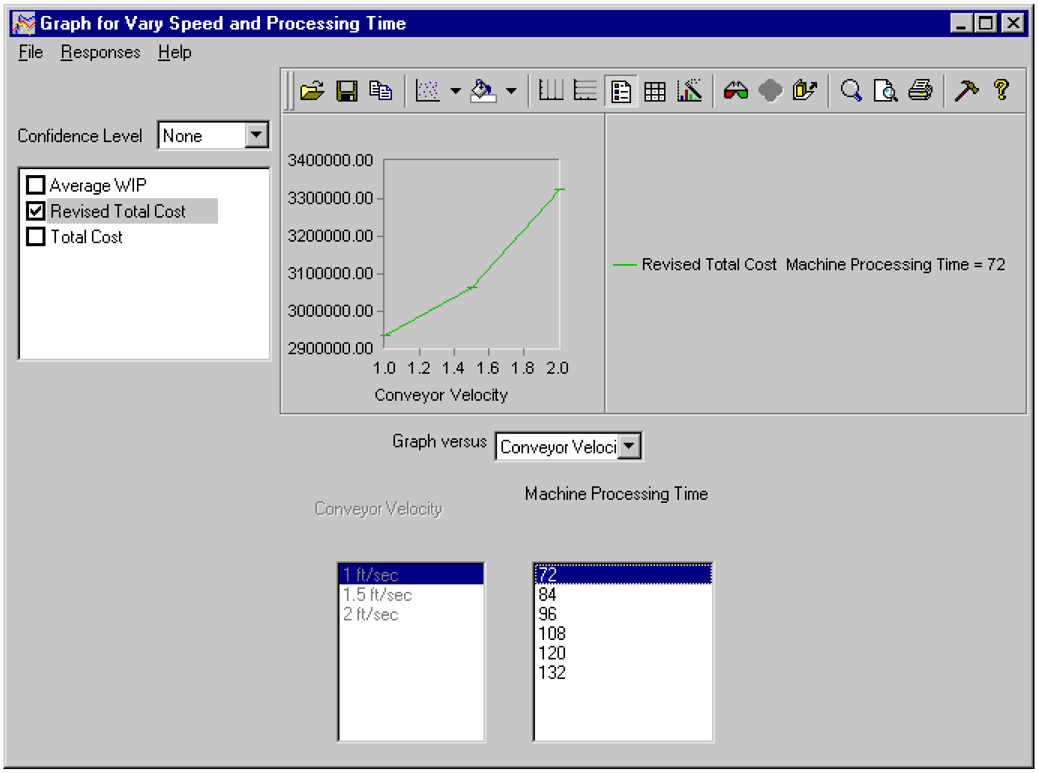
The Multiple Factors graph

Notice that for the base conveyor speed, the lowest cost is at the 120-second configuration. If you select any other conveyor velocity, the same result occurs, which indicates that when the conveyor speed is held constant, varying the processing time results in decreasing costs that eventually reach a lowest cost option (120 seconds) before costs increase again.

Now look at the graph when processing time is held constant and the conveyor speed varies:

Step 1 From the "Graph versus" drop-down list*, **select*** Conveyor Velocity.

Step 2 ***Select*** 72 (the base equipment speed) in the Machine Processing Time list. The response values are graphed.



Confidence Level: None

□ Average WIP
☑ Revised Total Cost
□ Total Cost

Graph versus: Conveyor Veloci

Conveyor Velocity: 1 ft/sec, 1.5 ft/sec, 2 ft/sec

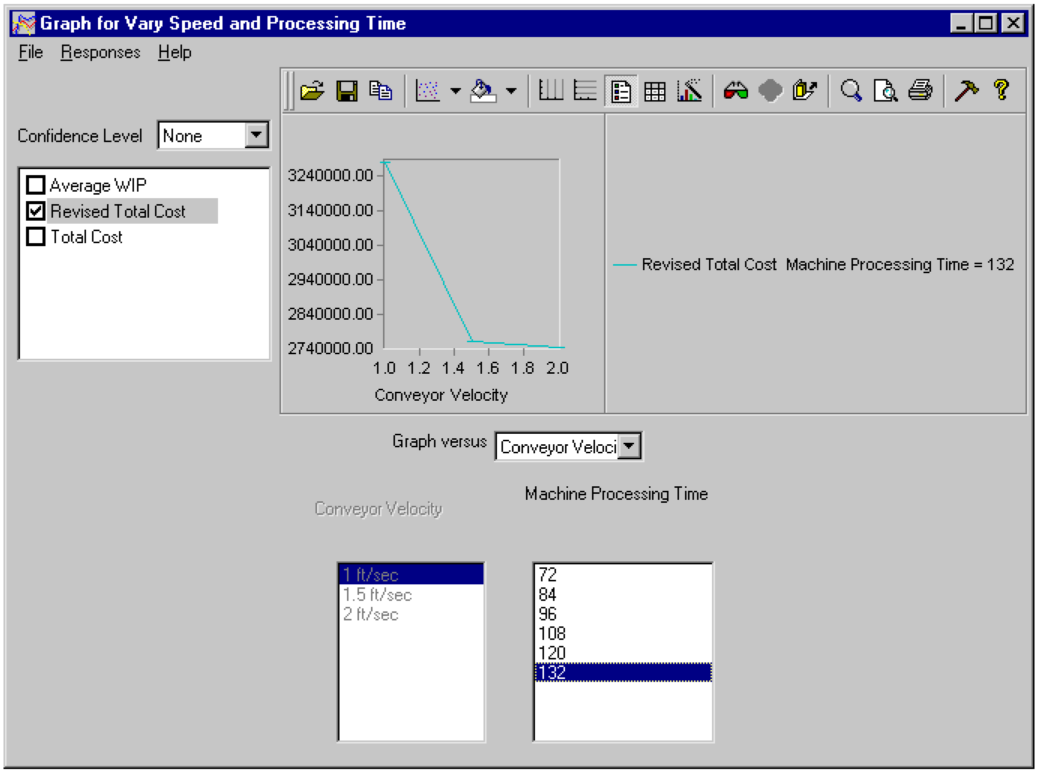
Machine Processing Time: 72, 84, 96, 108, 120, 132

Graphing processing time versus conveyor velocity

This graph illustrates that when the processing time is 72 seconds, increasing the conveyor speed results in increased costs.

Step 3 ***View*** the graph for each of the remaining processing times.

With the exception of a processing time of 132 seconds, each case of increased conveyor speed results in increased costs. When the processing time is 132 seconds, the costs decrease, as shown in the graph below:



Confidence Level: None

□ Average WIP
☑ Revised Total Cost
□ Total Cost

Graph versus: Conveyor Veloci

Conveyor Velocity
1 ft/sec
1.5 ft/sec
2 ft/sec

Machine Processing Time
72
84
96
108
120
132

The cost drops for a processing time of 132 seconds

Although the costs are decreasing at the fastest conveyor speed (2 ft/sec), the cost is still $2,740,000; this cost is greater than the lowest cost we have discovered in our previous analyses. The graph suggests that at a processing time of 132 seconds, increased conveyor speeds (greater than 2 ft/sec) may eventually result in lower costs. However, faster conveyors are not available to the company at this time, so based on the analyses you have conducted, you can conclude that the lowest cost option is a processing time of 120 seconds and a conveyor speed of 1.0 ft/sec, which costs $2,221,280.

Determining which runs your analysis is using

In this chapter, you conducted two different analyses for one model. In the first analysis, you analyzed the effect of varying the machine's processing time. In the second analysis, you varied both machine processing time and conveyor velocity. You made runs for both analyses.

To see how many runs have been made for all analyses:

Step 1 ***Select*** the Runs tab.

Step 2 ***Scroll*** the Current Runs list. There are 90 runs listed.

How do you know which runs are being used for the first analysis, and which ones are being used for the second analysis?

To check which runs an analysis is currently using:

Step 1 ***Select*** the Analyses tab.

Step 2 ***Expand*** the Vary Processing Time analysis and ***double-click*** Runs Used. The Runs Used window opens, listing each configuration of factor values used for the analysis.

Step 3 ***Expand*** the list of runs used for each configuration.

Notice that runs 1 through 30 are currently being used by the first analysis. All statistics and confidence intervals for this analysis are being generated from the output of these 30 runs. (As discussed in "Vary multiple factors analysis" on page 10.20, these 30 runs are also being used by the second analysis. To verify this, look at the runs used for the Vary Speed and Processing Time analysis.)

Tip ☞ If you edit the definition of an analysis, or if you delete it, you could have runs that are not being used anymore. Because runs take up space on your hard drive, you may want to delete unused runs using the Delete Unused Runs button on the Runs tab.

Summary

In this chapter, you learned how to conduct “What if...” experiments with your model by defining factors and using the vary one factor and vary multiple factors analyses. You also learned how to define combination responses to include cost information that is not in the model, and to perform calculations in the response using the Weight field.

You viewed several new types of output, including bar graphs, line graphs, and vary multiple factor graphs. You learned how to compare all scenarios to a base scenario to make decisions about whether changes have a positive or negative impact on the system you are modeling. Finally, you learned how to determine which runs are being used by a particular analysis and how to delete any unused runs.

AutoStat is a powerful tool to help you perform “What if...” scenarios. There are numerous types of output to help you analyze your scenarios and make a sound decision about how to improve a system.

Exercises

Note

Round your answers to the nearest hundredth.

Exercise 10.1

Copy the *base* version of example model 10.1 to a new directory. Open the copied model in AutoStat and use the Setup wizard to define the following model properties:

- Model is random
- Check for infinite loops and stop runs that are longer than 2 minutes
- Do not define a warmup
- Define a snap length of 10 days

Perform an analysis that varies the processing time of resource R_Processor from 84 to 108 seconds in 12 second increments. Make 5 replications, then complete the following:

a) What is the average time that loads spend in the system when the processor time is 96 seconds?

b) What is the 95 percent confidence interval for the average time that loads spend in the system when the processor time is 108 seconds?

c) Compare the 95 percent confidence interval for the average time that loads spend in the system when the processing time is 84 seconds to the same interval for average time that loads spend in the system when the processing time is 96 or 108 seconds. Based on the comparison, is there a statistically significant difference in the average time that loads spend in the system?

d) Display a bar graph of the average time that loads spend in the system and change its scheme to a black and white pattern. Print the graph.

e) Display a line graph of the average time that loads spend in the system and change its scale so that the lowest value on the Y axis is 800. Print the graph.

Exercise 10.2

Copy the *base* version of model 10.1 to a new directory. Open the copied model in AutoStat and use the Setup wizard to define the following model properties:

- Model is random
- Check for infinite loops and stop runs that are longer than 2 minutes
- Do not define a warmup
- Define a snap length of 10 days

Perform an analysis that varies the processing time of R_Processor from 72 to 132 seconds in 12 second increments. Make 5 replications.

Given a $1,000 per second cost based on the average time that loads spend in the system, develop a combination response to determine which equipment model provides the lowest cost. Use the equipment costs shown in “Equipment specifications” on page 10.5 for your analysis.

Tip

The cost for time in system replaces the AvgWIP cost in example 10.1.

Determine the processing time that results in the lowest total cost.

Exercise 10.3

Copy the *final* version of example model 10.1 to a new directory. Open the copied model in AutoStat. The factors, responses, and analyses used in this chapter are already defined.

Edit the second analysis, Vary Speed and Processing Time, that was conducted for example 10, changing the snap length from 10 days to 5 days.

Make the runs for that analysis and determine the processing time and conveyor speed that result in the lowest total cost. Are the results the same as were found for a snap length of 10 days?

Exercise 10.4

Copy the *base* version of example model 10.1 to a new directory. Open the copied model in AutoStat and use the Setup wizard to define the following model properties:

- Model is random
- Check for infinite loops and stop runs that are longer than 2 minutes
- Do not define a warmup
- Define a snap length of 10 days

Given the new conveyor speeds and costs provided below:

Speed	**Cost**
1 ft/sec	$150,000
2 ft/sec	$300,000
3 ft/sec	$450,000

Perform an analysis that varies the conveyor speed from 1 to 3 ft/sec and the processing time from 72 to 132 seconds in 12 second increments. Use the same WIP and equipment costs shown in “Example 10.1: Performing a financial analysis” on page 10.4.

Determine the conveyor speed and processing time that results in the lowest total cost.

Exercise 10.5

Copy the *base* version of example model 10.1 to a new directory. Open the copied model in AutoMod, and edit the P_process arriving procedure so that the delay time for the equipment is divided into two times, a setup time and a processing time, that can be varied independently. Edit the logic as shown below (changes are indicated in bold text):

```
begin P_process arriving
   move into Q_geton
   move into conv:get_on
   travel to conv:work_area
   get R_processor
   wait for V_first_time /* variable of type Time */
   wait for e V_second_time/* variable of type Time */
   free R_processor
   travel to conv:get_off
   send to die
end
```

Export the model.

Open the model in AutoStat and use the Setup wizard to define the following model properties:

- Model is random
- Check for infinite loops and stop runs that are longer than 2 minutes
- Do not define a warmup
- Define a snap length of 7 days

Perform an analysis to vary both delays for the piece of equipment. Vary V_first_time from 12 to 36 seconds in 12 second increments and V_second_time from 60 to 96 seconds in 6 second increments. Make 5 replications.

Define a Total Cost response exactly as shown in “Defining the cost equation in a combination response” on page 10.16. View the results and determine which values of V_first_time and V_second_time result in the lowest total cost with a 95 percent level of confidence.

Exercise 10.6

Copy example model 9.1 (from the previous chapter) to a new directory. Open the copied model in AutoStat and use the Setup wizard to define the following model properties:

- Model is random
- Check for infinite loops and stop runs that are longer than 2 minutes
- Do not define a warmup
- Define a snap length of 100 hours

Additional information about the system is provided below:

The facility runs 24 hours a day, 7 days a week. WIP has an annual cost of 10 percent of the total value of the loads in the system; loads in the system have a value of $200,000 each.

You have the option of changing the following system components:

a) You can schedule one or two workers in the system. Workers are each paid $40/hour.
b) You can increase the conveyor speed, with costs defined in the table below:

Speed	**Cost**
1 ft/sec	$300,000 (Current speed)
1.5 ft/sec	$400,000
2 ft/sec	$500,000

Use AutoStat to analyze the system and determine the number of workers and the conveyor speed that result in the lowest total cost. Make 5 replications.

Chapter 11

Introduction to Path Mover Systems

Chapter 11

Introduction to Path Mover Systems

A path mover system in the AutoMod software is a material handling system in which vehicles or people move along a guide path, carrying loads from pickup locations to delivery locations. In this chapter, you will learn how to draw a path mover system and place vehicle graphics to model an automated guided vehicle (AGV) system. Many of the concepts and tools that are used to draw path mover systems are similar to those you have already used to simulate conveyor systems.

In chapter 12, “Modeling Complex Material Handling Systems,” you will learn how to write model logic to control load movement and vehicle scheduling in path mover systems.

Path mover systems

Path mover systems can be used to simulate any type of vehicle system in which vehicles follow a specific route or path, for example, manually operated lift trucks or automated guided vehicles. Vehicles in path mover systems can also represent people who move along a predetermined route in the system.

Path mover systems consist of the following components:

Guide paths Guide paths are segments of path on which vehicles travel. Guide paths represent routes that are taken by people or vehicles in a system. Guide paths can be one- or two-directional.

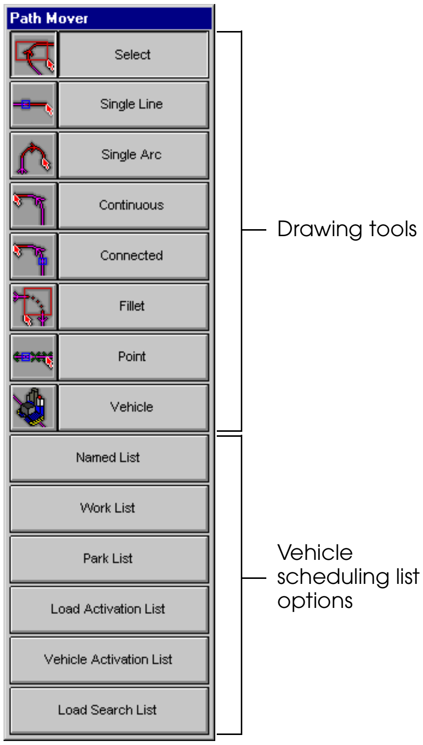
Transfers Transfers are connections that join two segments of guide path. For vehicles to move from one segment to another, the two segments must be connected by a transfer. Transfers are automatically created as you draw paths in the path mover system.

Control points Control points are locations at which vehicles can pick up or set down loads in the system. Control points can be located anywhere on a path.

Vehicles Vehicles transport loads from one location to another by following a path in the path mover system. Vehicles can be defined and grouped by type and can differ in velocity, capacity, and the time required to pick up and set down loads in the system. Vehicles can also be defined with different attributes based on the type of loads they are carrying; for example, empty vehicles or vehicles that are carrying heavier or lighter loads may have different rates of acceleration or velocity.

Path mover drawing tools

The tools for drawing and placing path mover entities are located on a palette in the path mover system, as shown below:



Note In addition to the drawing tools, the path mover palette contains vehicle scheduling list options that are used to define scheduling lists in the path mover system. These options are discussed in chapter 12, “Modeling Complex Material Handling Systems.”

Many of the drawing tools in path mover systems are the same as those used to draw sections in conveyor systems. The drawing tools are defined as follows:

Select The Select tool selects one or more entities in the path mover system.

Single Line The Single Line tool draws a straight path mover segment.

Single Arc The Single Arc tool draws a curved path mover segment.

Continuous The Continuous tool draws a single path mover segment that consists of multiple straight or curved pieces. All pieces are part of the same segment, that is, *no transfers are created*. By default, the Continuous tool alternates between drawing straight and curved pieces.

Connected The Connected tool draws multiple path mover segments connected by transfers. Unlike the Continuous tool, the Connected tool draws straight segments by default.

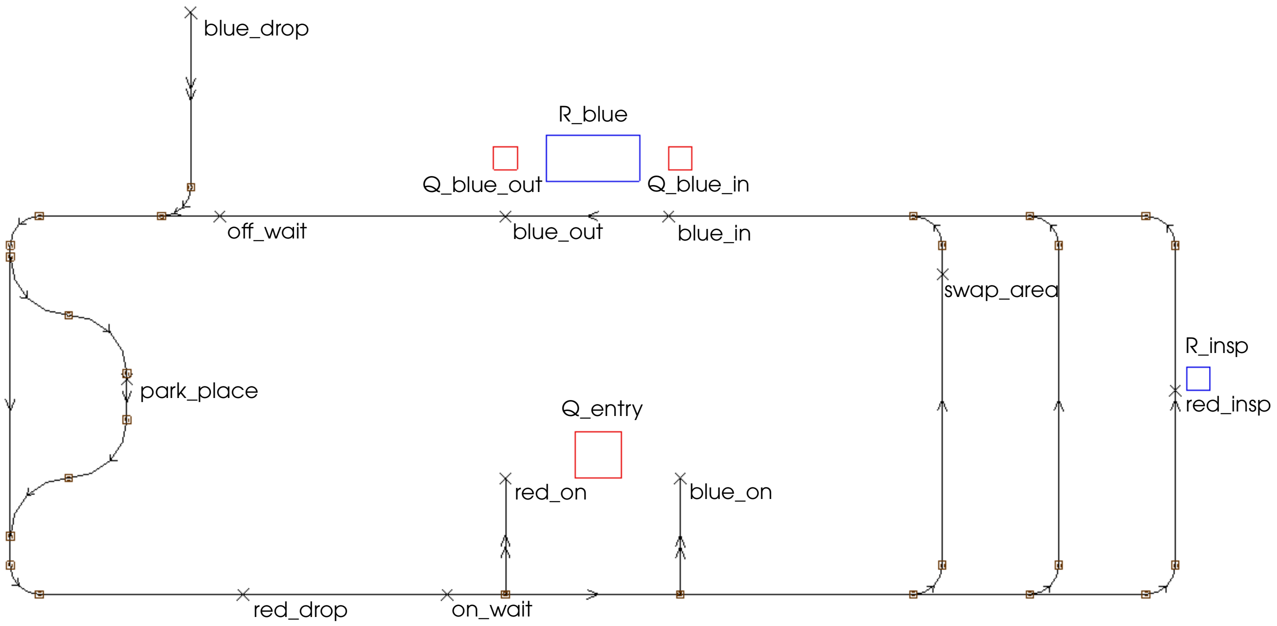
Fillet The Fillet tool draws a curved path mover segment to automatically connect two non-parallel segments.

Point The Point tool places a control point on a path.

Vehicles The Vehicles tool defines one or more vehicles to travel on the paths in the path mover system.

Example 11.1: Drawing a path mover system

Consider the layout of the path mover system shown below:



Layout of example model 12.1

Two types of loads are processed in this system: red loads and blue loads. Both types of loads have an interarrival time that is exponentially distributed with a mean of 5 minutes. Loads first move into an infinite-capacity queue (Q_entry). Loads are then picked up by a vehicle at one of two control points, depending on type: red loads are picked up at control point “red_on” and blue loads are picked up at control point “blue_on.”

Red loads are carried to the control point “red_insp” where they are inspected (while onboard the vehicle) by an inspector for a time that is exponentially distributed with a mean of 3 minutes. After being inspected, red loads are carried to the control point “red_drop” where they get off the vehicle and leave the system.

After getting onboard a vehicle, blue loads are carried to the control point “blue_in” where they get off the vehicle and are placed in an infinite-capacity processing queue, Q_blue_in. Loads are processed in the queue by a single-capacity resource, R_blue, for a time that is normally distributed with a mean of 4 minutes and a standard deviation of 30 seconds. After completing processing, the loads move into an infinite-capacity queue, Q_blue_out, where they wait to be picked up by a vehicle at control point “blue_out.” After being picked up, loads travel to the control point “blue_drop,” where they get off the vehicle and leave the system.

In this chapter, you will draw the path mover layout and define the vehicles to simulate example 11.1. In chapter 12, “Modeling Complex Material Handling Systems,” you will complete the example model by writing the model logic, placing process system entities, and defining the scheduling lists that control load and vehicle movement in the system.

To become familiar with load and vehicle movement in the system:

Step 1 ***Import*** and ***run*** a copy of the *final* version of example model 12.1

Note Example model 12.1 is a working version of the system; example model 11.1 only contains the drawing of the path mover system.

While the model is running, notice the following:

- There are three vehicles in the system. At the beginning of the simulation, only one vehicle is parked at the control point “park_place;” the remaining vehicles are not yet displayed in the simulation. When the first vehicle leaves, the second vehicle appears parked at the control point, and so on, until all of the vehicles are displayed in the simulation.
- Vehicles travel in two directions on the paths containing the control points “red_on” and “blue_on.” Vehicles enter the path to pick up a red or blue load and then leave on the same path to deliver the load. Notice that vehicles travel sideways while on the two paths.
- Vehicles also travel in two directions on the path containing the control point “blue_drop.” However, unlike the two pickup paths, vehicles do not travel sideways on this path.
- Because only one vehicle can travel on the two-directional paths at a time (the vehicle enters and leaves on the same path), the control points “off_wait” and “on_wait” provide a location where vehicles that need to enter an occupied path can stop and wait for the path to become available.
- Idle vehicles (vehicles that are neither picking up nor delivering a load) travel to the parking location “park_place” to wait for work. When a load requires transportation in the system, an idle vehicle leaves the parking location and travels to the load’s pickup point.
- When a vehicle is obstructed by a preceding vehicle on a path, the trailing vehicle accumulates behind the preceding vehicle.

Note The system layout contains a control point named “swap_area,” located on a vertical path. Currently, vehicles in the system do not travel on this path. The path and control point provide an area where vehicles can have their batteries replaced during a simulation. You will simulate battery replacement later (see “Example 12.2: Modeling battery replacement” on page 12.16 of the “Modeling Complex Material Handling Systems” chapter).

Step 2 When you are ready to continue, ***quit*** the model.

Creating example model 11.1

Example model 11.1 consists only of a path mover layout. To create the example model, do the following:

Step 1 ***Create*** a new model named “examp111.”

Step 2 ***Open*** the View Control, then ***select*** Child Windows on Top.

Step 3 ***Close*** the View Control window.

You are now ready to create the path mover system.

Creating the path mover system

To create the path mover system, do the following:

Step 1 From the System menu in the Work Area window, ***select*** New. The Create A New System window opens.

Step 2 In the System Name text box, ***type*** “pm”, and in the System Type drop-down list, ***select*** Path Mover.

Step 3 ***Click*** Create to create the path mover system. The Path Mover palette appears.

Drawing paths

To draw the path layout in example model 11.1, use the following methodology:

Step 1 ***Draw*** straight paths.

Step 2 ***Fillet*** the straight paths (to connect all of the paths in the system).

Step 3 ***Draw*** arc paths (to create the parking area).

Each of these steps is discussed in detail in the following sections.

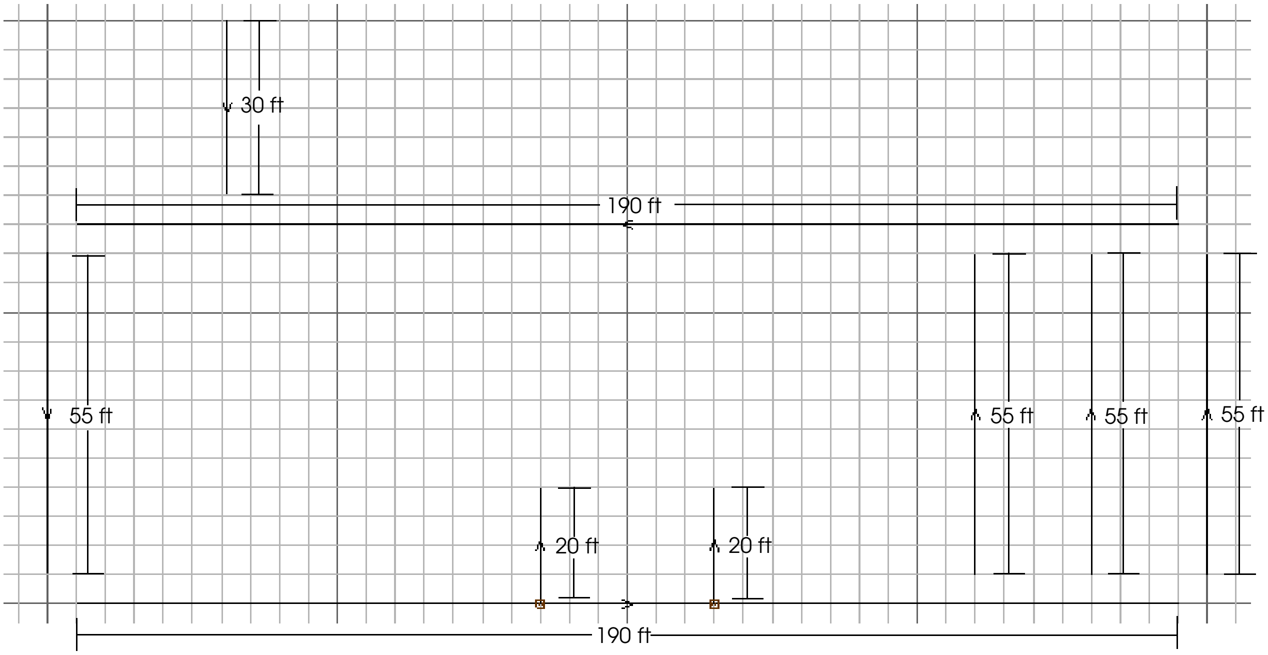
Drawing straight paths

To draw the straight paths in the system, use the Single Line tool as you did in “Drawing conveyor sections” on page 6.10 of the “Introduction to Conveyors” chapter.

Step 1 ***Draw*** the straight paths, using the measurements shown below:

Tip

While drawing the paths, use the Orthogonal option to draw straight lines and use the Snap to Path option to force the two 20-foot entrance sections to connect to the horizontal path.



Drawing straight paths in example model 11.1

Note

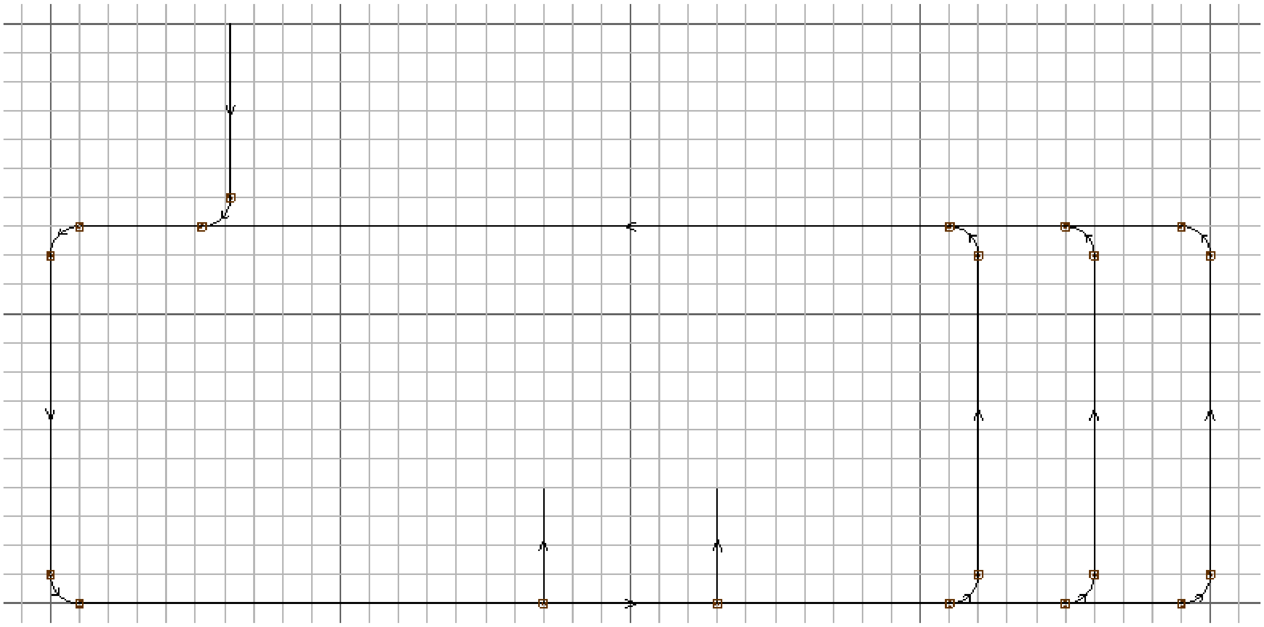
Be sure to draw the paths with the correct direction of travel (you will change the entrance and exit paths to be two-directional spur paths later in this chapter).

Now you are ready to connect the straight paths using the Fillet tool.

Filleting paths

To connect the straight paths in the model, use the Fillet tool as you did in “Filleting two paths” on page 6.13 of the “Introduction to Conveyors” chapter.

Step 1 ***Fillet*** the paths, as shown below:



Filleting paths in example model 11.1

Now you are ready to draw the parking area.

Drawing arcs

To draw the parking area in the model, use the Single Arc tool as described below:

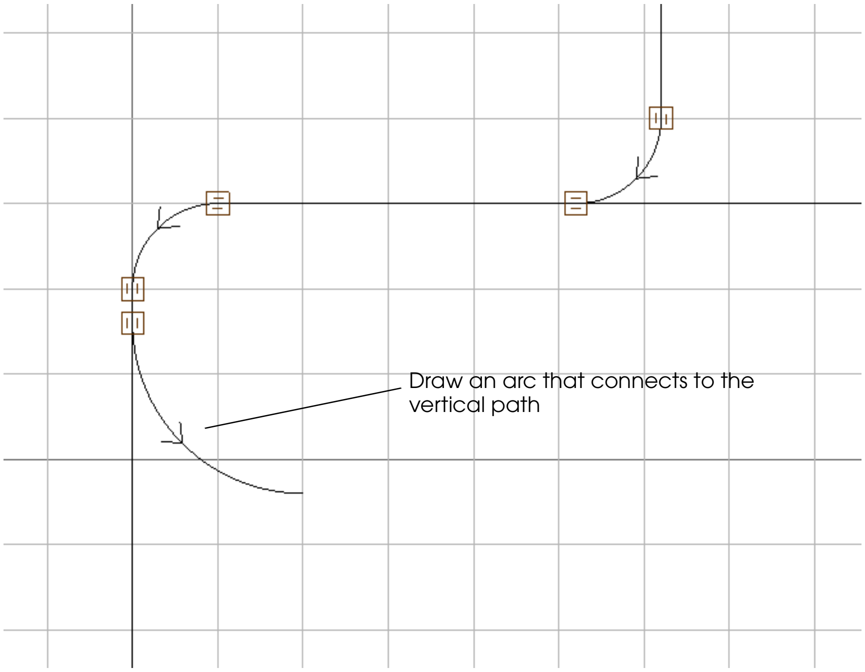
Step 1 On the Path Mover palette, ***click*** Single Arc. The Single Arc window opens.

Step 2 ***Change*** the radius of the segment to 10 feet.

Step 3 ***Select*** Minor Arc to force the arc to be equal to or less than 180 degrees.

Step 4 ***Click*** Snap to Path.

Step 5 ***Drag*** an arc that begins below the transfer of the top of the left vertical segment, as shown below:



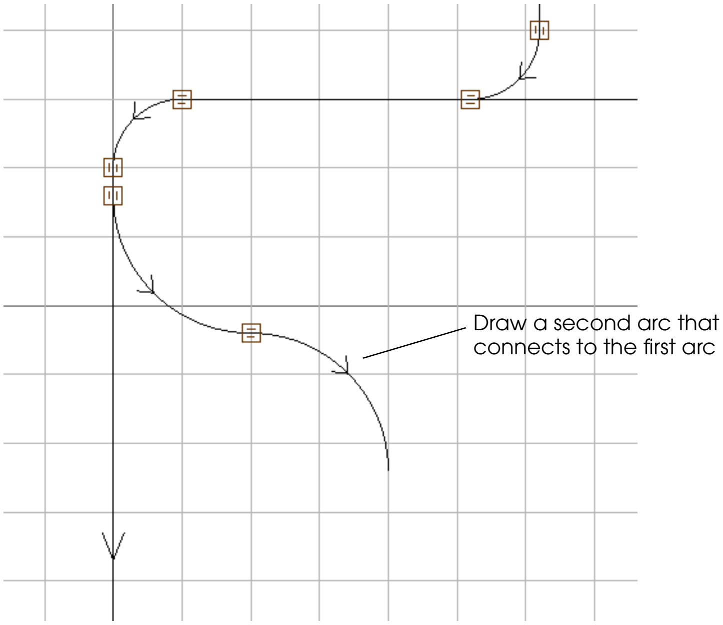
Drawing the first arc in the parking area in example model 11.1

You have now drawn the first arc in the parking area.

Step 6 In the Single Arc window, ***select*** Clockwise.

Step 7 ***Click*** Snap to End.

Step 8 ***Drag*** an arc that connects to the existing arc, as shown below:



Drawing the second arc in the parking area in example model 11.1

You have now drawn the first half of the parking area. To draw the second half of the parking area, copy the two arc paths you just drew, as discussed in the next section.

Copying arc paths

Step 1 On the Path Mover palette, ***click*** Select.

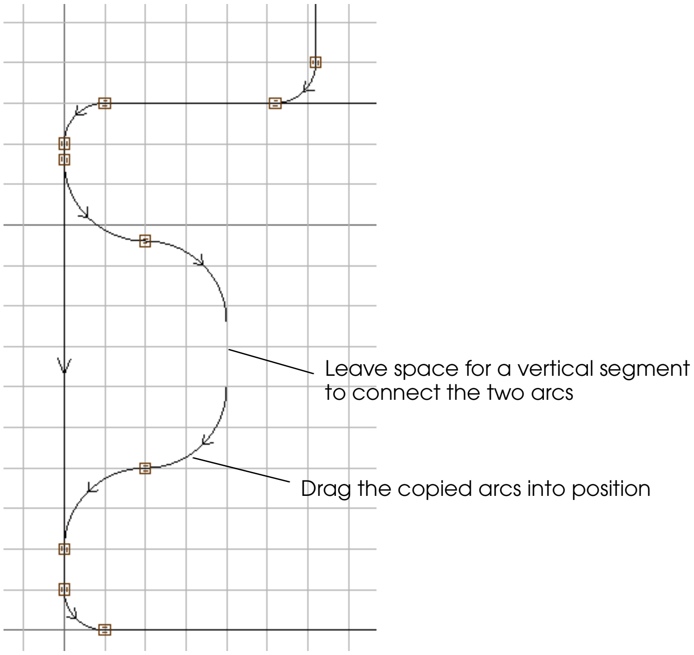
Step 2 ***Select*** the two arc paths.

Tip You can select multiple paths at the same time by dragging a box in the Work Area window over parts of each path. The paths turn green to indicate that they are selected.

Step 3 From the Edit menu, ***select*** Copy. The Copy window opens.

Step 4 In the Copy window, ***click*** Flip Vertical.

Step 5 ***Drag*** the copied arcs into position, as shown below:


Leave space for a vertical segment to connect the two arcs
Drag the copied arcs into position

Positioning the copied arcs in example model 11.1

Step 6 Using the Single Line tool, ***draw*** a vertical segment connecting the two arcs. Be sure to draw from top to bottom so that the direction of vehicle travel is correct.

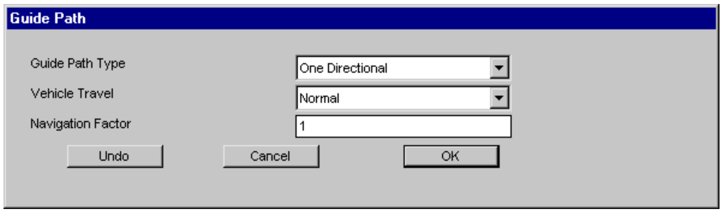
Tip When drawing the vertical segment, use the Snap to End option to ensure that the path connects to the arcs.

You have now drawn the parking area.

Modeling different types of paths

In chapter 9, “Modeling Complex Conveyor Systems,” you learned how to model different types of conveyor sections. Similarly, you can edit the attributes of individual paths in a path mover system to change path characteristics. For example, you can change the direction in which vehicles can travel or the orientation of vehicles that travel on a path.

Path attributes are edited in the Guide Path window, as shown below:



Guide Path

Guide Path Type: One Directional

Vehicle Travel: Normal

Navigation Factor: 1

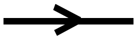
The Guide Path window

The Guide Path window lists three attributes that can be edited for paths:

Guide Path Type

The Guide Path Type attribute determines the direction in which vehicles can travel and the velocity of vehicles on the path. Options in the Guide Path Type drop-down list are:

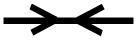
One Directional – Vehicles can travel in only one direction on the path; the direction in which vehicles can travel is indicated by the path’s direction marker, as shown below:



One-directional path

Vehicles that are traveling on the path use the Forward, Reverse, or Curve velocity defined in the vehicle’s specification by load type (see “Specifying vehicle attributes by load type” on page 11.23).

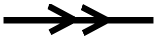
Two Directional – Vehicles can travel in two directions on the path. The path’s direction markers point in both directions to indicate a two-directional path, as shown below:



Two-directional path

Vehicles that are traveling on the path use the Forward, Reverse, or Curve velocity defined in the vehicle’s specification by load type (see “Specifying vehicle attributes by load type” on page 11.23).

Spur – Vehicles can travel in two directions on the path. The path’s direction markers point in the path’s forward direction of travel, as shown below:



Spur path

Although vehicles can travel in both directions on the path, the path has a forward direction of travel that is used to determine vehicle orientation on the path (for more information, see “Determining vehicle orientation on a path” on page 11.26). Vehicles that are traveling on the path use the Forward or Reverse spur velocity defined in the vehicle’s specification by load type (see “Specifying vehicle attributes by load type” on page 11.23).

Note

When modeling two directional or spur paths, it is important to prevent vehicle collisions by placing control points to limit the number of vehicles that can travel on the path at a time (for more information, see “Setting control point capacity” on page 11.20).

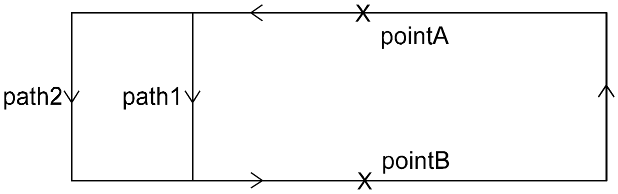
Vehicle Travel The Vehicle Travel attribute determines the alignment of vehicles that are traveling on the path. Options in the Vehicle Travel drop-down list are:

Normal – A vehicle's X axis is aligned with the path's direction of travel.

Crab – A vehicle's Y axis is aligned with the path's direction of travel (the vehicle travels sideways on the path).

Navigation Factor By default, vehicles take the shortest route to their destination. A path's Navigation Factor is a number that is multiplied by the length of the path; this factor is used when vehicles are searching for their shortest route. The default navigation factor is one. By changing a path's navigation factor, you can force vehicles to take a different path to their destination.

For example, in the system illustrated below, a vehicle traveling from pointA to pointB would take path1, because the default navigation factor of 1 makes path1 the shortest route.



Navigation factor

However, if path1 had a navigation factor that made it longer than path2, for example, a navigation factor of "3," then the vehicle would use path2 as the shortest path instead.

Each of the path attributes is discussed in the following sections.

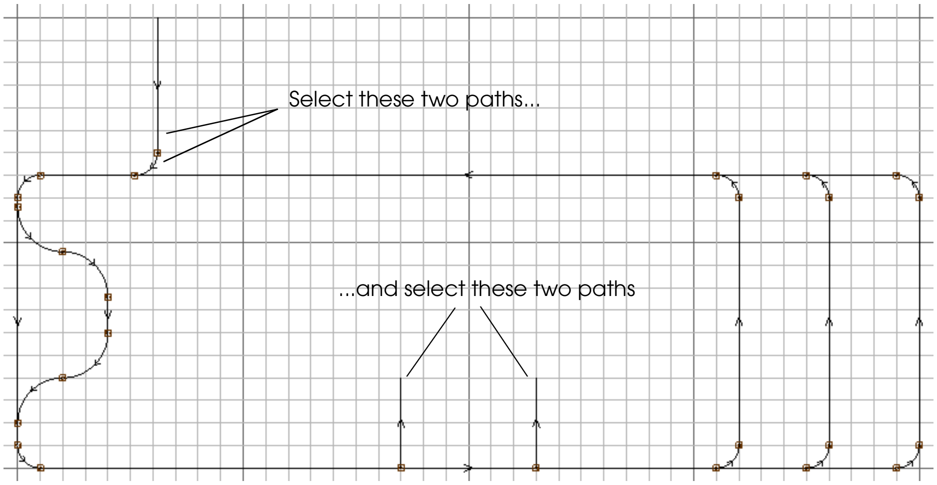
Setting the direction of travel on paths

By default, vehicles can travel in only one direction on a path; the direction in which vehicles can travel is indicated by the path's direction marker. In example model 11.1, there are four spur paths on which vehicles can travel in both directions.

To set the paths' direction, do the following:

Step 1 ***Select*** the four paths on which vehicles travel to pick up or deliver loads, as shown below:

Tip ☞ Hold the Shift key to select more than one path at a time.



Setting the direction of paths in example model 11.1

Step 2 From the Edit menu, ***select*** Edit. The Path Edit window opens.

Step 3 ***Click*** Attributes to edit the attributes of the first path. The Guide Path window opens.

Step 4 In the Guide Path Type drop-down list, ***select*** Spur, then ***click*** OK. You have now changed the path to a spur path.

Step 5 In the Path Edit window, ***click*** OK to begin editing the next selected path.

Step 6 ***Repeat*** steps 3 through 5 to set the value of the Guide Path Type attribute for the remaining selected paths. After editing the last path's attributes, ***click*** OK, Quit Edit Each in the Path Edit window to stop editing selected entities.

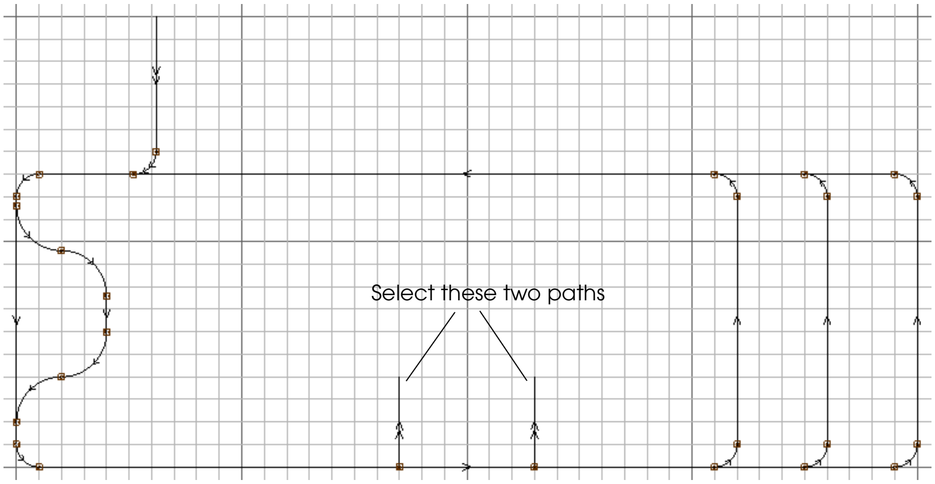
Notice that the direction markers on the paths are replaced with double arrows to indicate they are spur paths.

Setting crab movement on paths

On a normal path, a vehicle's X axis is aligned with the path's direction of travel (see "Determining vehicle orientation on a path" on page 11.26 for more information). On a crab path, a vehicle's Y axis is aligned with the path, causing the vehicle to travel sideways. In example model 11.1, vehicles travel sideways on the two paths where vehicles pick up red and blue loads.

To define crab movement on the paths, do the following:

Step 1 ***Select*** the two paths on which vehicles travel to pick up loads, as shown in the illustration below:



Setting crab movement on paths in example model 11.1

Step 2 ***Edit*** each path's attributes and ***set*** the value of the Vehicle Travel attribute to Crab.

Tip ☞ After setting a path's attribute value, click OK in the Path Edit window to begin editing the next selected path.

Step 3 When you have finished editing each path's attributes, ***click*** OK, Quit Edit Each in the Path Edit window to stop editing selected entities.

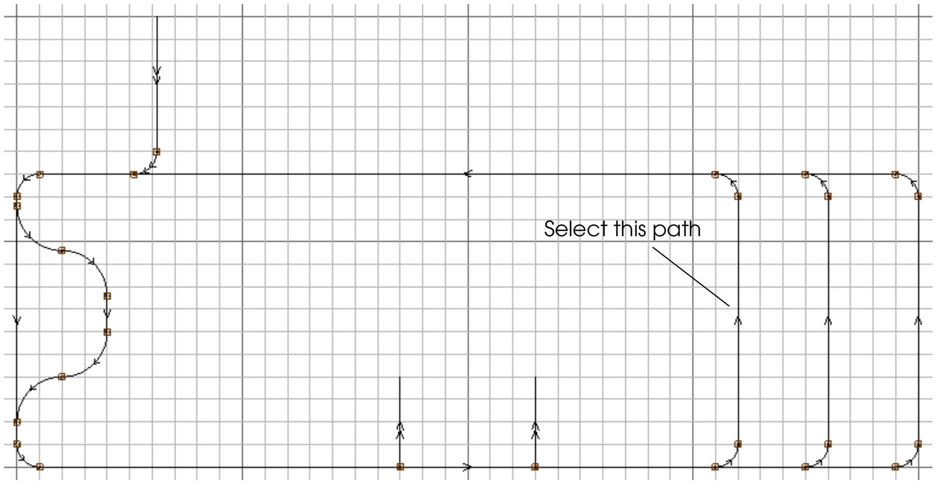
Vehicles are now able to travel sideways on the paths leading to the two pickup locations.

Setting the navigation factor of paths

In example model 11.1, a vertical lane is defined where vehicles can have their batteries replaced. This battery replacement area is not implemented until later (see "Example 12.2: Modeling battery replacement" on page 12.16 of the "Modeling Complex Material Handling Systems" chapter). When the replacement area is added, vehicles will travel down the vertical lane and stop for 15 minutes to change batteries. Because battery replacement is a lengthy process, we want this lane to be dedicated only to vehicles that are having their batteries replaced. To prevent other vehicles from traveling down the lane, we can change the path's navigation factor so that it is not selected as the shortest route when dropping off or picking up loads.

To set the navigation factor of the battery replacement lane, do the following:

Step 1 ***Select*** the vertical path where vehicles travel to change batteries, as shown below:



Setting the navigation factor of the battery replacement lane in example model 11.1

Step 2 ***Edit*** the path's attributes and ***set*** the value of the Navigation Factor attribute to "4."

Step 3 ***Click*** OK, Quit Edit Each in the Path Edit window to stop editing selected entities.

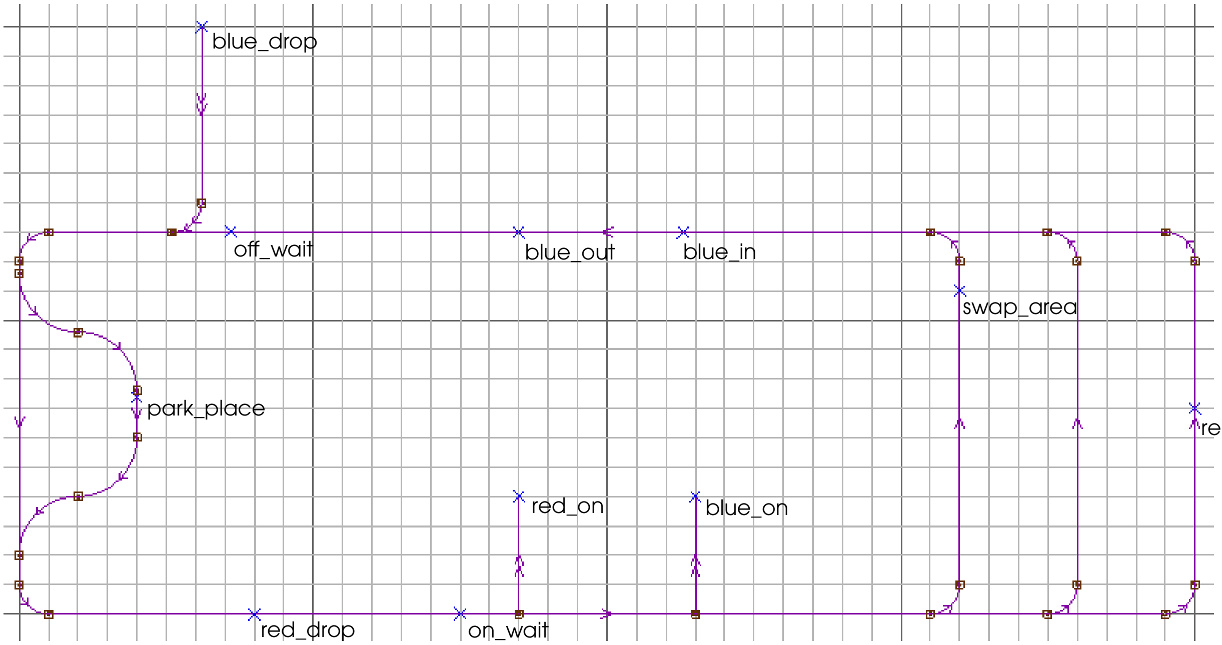
The battery replacement lane is no longer the shortest route for vehicles that are traveling to pick up or set down loads in the system.

Placing control points

Now that you have drawn the necessary paths and set their attributes in example model 11.1, you are ready to place control points in the path mover system. **Control points** represent locations where vehicles can stop, for example, to pick up or set down loads, or to park when idle.

Placing control points is like placing stations in a conveyor system. To place control points in example model 11.1, use the Point tool on the Path Mover palette.

Step 1 ***Place*** and ***name*** the control points, as shown below:



Placing control points in example model 11.1

Now that you have placed the control points in the model, you are ready to learn how to change control point attributes.

Changing control point attributes

Control point attributes allow you to change the characteristics of individual control points in a model. This textbook discusses the following control point attributes:

Control Point Capacity A control point's capacity determines the number of vehicles that can simultaneously travel to a control point during a simulation.

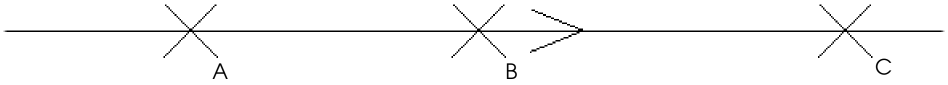
Control Point Release A control point's release value determines when a vehicle releases a claimed control point so that other vehicles can travel to the point. There are many ways to define a control point's release value. For example, a vehicle can release a control point immediately upon leaving the point, after delaying for a specified amount of time, after traveling a certain distance beyond the point, and so on.

To get a better understanding of how control point attributes affect vehicle behavior, you will edit the attributes of individual control points as described in the following sections.

Setting control point capacity

Each control point has a capacity that can be used to limit the number of vehicles that can simultaneously travel to the point. In order for a vehicle to travel to a control point, it must be able to claim one unit of the control point's capacity. After leaving the control point, the vehicle releases one unit of capacity, which can then be claimed by another vehicle. By default, the capacity of control points is "Infinite," which allows an unlimited number of vehicles to travel to the control point simultaneously. However, you can limit the number of vehicles that can travel to a control point by setting its capacity to a finite number.

Limiting control point capacity also has the effect of restricting the number of vehicles that can simultaneously travel on a path leading to a control point. For example, consider the path shown below:



Control point illustration

In order for a vehicle to leave its current control point, it must be able to claim the next control point on the path. A vehicle that is traveling from point "A" to point "C" must be able to claim point "B" *before* it can leave point "A." If the capacity of control point "B" is set to one, only one vehicle can travel on the path between point "A" and point "B" at the same time. Trailing vehicles will stop and accumulate at point "A" and wait for point "B" to be released by the preceding vehicle before they can continue. Similarly, after arriving at point "B," a vehicle must be able to claim point "C" before it can leave point "B."

In example model 11.1, the number of vehicles that can simultaneously travel on the spur paths where vehicles pick up loads or set down blue loads must be limited to one. Otherwise, multiple vehicles may enter a path, resulting in a deadlock. A **deadlock** is an impasse that occurs when two vehicles are trying to move but are blocking one another's path. For example, consider what would happen if two vehicles entered the spur path to control point "red_on" at the same time. The first vehicle would travel to the control point and pick up a red load. The second vehicle would accumulate behind the first vehicle. Once the load was on the first vehicle, the vehicle would attempt to leave the spur path, but would not be able to move, because the trailing vehicle, which is waiting to travel to the control point, is blocking its path.

To prevent deadlocks from occurring, limit the number of vehicles that can travel to a point by changing its capacity, as follows:

Step 1 ***Select*** the control points "blue_drop," "red_on," and "blue_on."

Tip ☞ To select multiple control points, hold down the Shift key and click each point (the control point graphics turn green to indicate that they are selected).

Step 2 From the Edit menu, ***select*** Edit. The Control Point Edit window opens for the first selected control point.

Step 3 ***Click*** Attributes. The Edit Control Point window opens.

Step 4 ***Change*** the control point capacity to "1" and ***click*** OK to close the Edit Control Point window.

Step 5 ***Click*** OK in the Control Point Edit window to begin editing the next control point.

Step 6 ***Repeat*** steps 3 through 5 to limit the capacity of the remaining control points to 1.

You are now ready to set the control points' release values.

Setting control point release values

A control point's release value determines when the point is released by a leaving vehicle. Releasing a control point allows other vehicles in the system to claim and travel to the point.

Note

A control point's release value has no effect on control points with infinite capacity (any number of vehicles can claim a point with infinite capacity, regardless of how many vehicles are currently claiming it).

You can define control point release values in terms of distance. For example, a control point release value of "50 feet" causes vehicles to release the point after leaving it and traveling 50 feet in the system. By default, the control point release value is "0 feet," which means that a vehicle releases a control point as soon as it leaves the point and begins traveling to its next point.

You can also define control point release values in terms of time. For example, a control point release value of "5 minutes" causes vehicles to release the point 5 minutes after leaving it, regardless of how far the vehicle travels in that amount of time.

You can also define control point release values using the setting "at end." This setting indicates that vehicles do not release a control point until they reach their destination in the system.

In example model 11.1, you need to change the control point release values for each of the control points at the end of a spur lane. You have already set the capacity of these control points to one, which limits the number of vehicles that can enter the lane simultaneously. However, the default control point release value is "0 feet," which could allow a vehicle to enter a spur lane before the preceding vehicle (which is on its way out) has left, resulting in a collision. To prevent collisions, you must change the release values for each control point so that a vehicle does not release the point until it has left the spur path.

To change the control point release values, do the following:

Step 1 ***Select*** the control point "blue_drop."

Step 2 From the Edit menu, ***select*** Edit. The Control Point Edit window opens.

Step 3 ***Click*** Attributes. The Edit Control Point window opens.

The path that this point is on is 30 feet long, and there is a short arc path on which the vehicle must travel, as well. To guarantee that the release value is long enough to cover both paths, you will set the release value to 38 feet.

Step 4 ***Change*** the control point release value to "38 feet" by selecting "feet" in the Control Point Release drop-down list and then typing "38" in the text box. ***Click*** OK to close the Edit Control Point window.

Step 5 ***Click*** OK in the Control Point Edit window to close the window. You have now changed the release value of the control point "blue_drop."

Step 6 ***Repeat*** this procedure to set the release value of the control points "red_on" and "blue_on" to 20 feet each. Because these paths are shorter, vehicles can release the control points after traveling a shorter distance.

You are now ready to define vehicles in the path mover system.

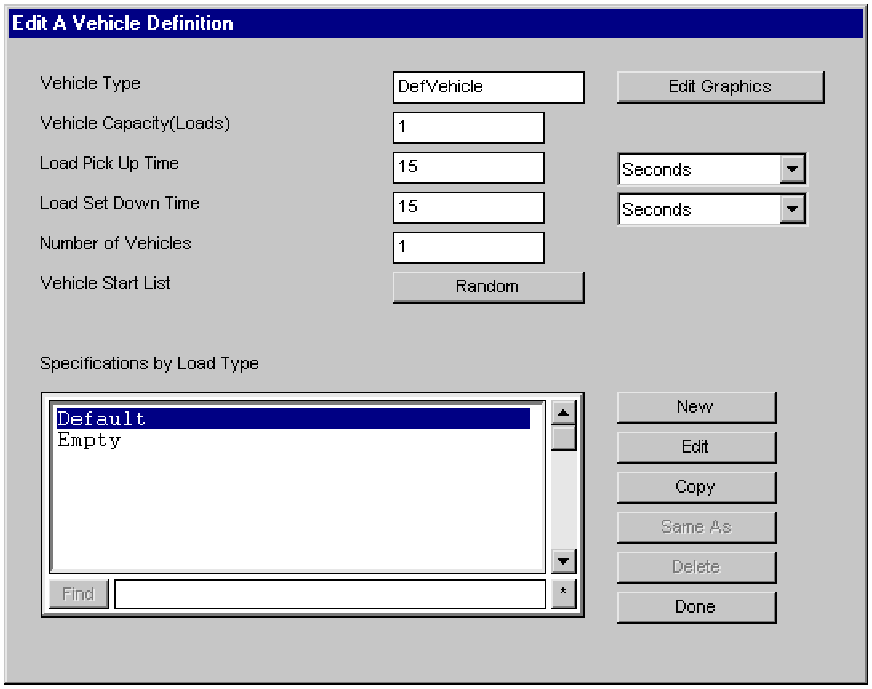
Defining vehicles

Vehicles can represent anything that carries loads in a system, such as manually operated fork trucks, automated guided vehicles, or people. As with load types, you can define different types of vehicles that vary according to characteristics such as vehicle size, the time required to pick up and set down loads, and so on. Defining multiple vehicle types allows you to simulate different types of vehicles in the same system. For example, fork trucks and people might travel on the same paths in the system. By default, path mover systems include one vehicle of type “DefVehicle.”

In example model 11.1, you need to create three vehicles that are all of the same type. To define the vehicles, do the following:

Step 1 On the Path Mover palette, ***click*** Vehicle. The Vehicles window opens.

Step 2 Because the vehicle type “DefVehicle” is already selected, ***click*** Edit to edit the default vehicle type. The Edit A Vehicle Definition window opens, as shown below:



Vehicle Type: DefVehicle
Vehicle Capacity(Loads): 1
Load Pick Up Time: 15 Seconds
Load Set Down Time: 15 Seconds
Number of Vehicles: 1
Vehicle Start List: Random

Specifications by Load Type:
Default
Empty

The Edit A Vehicle Definition window

Options in the Edit A Vehicle Definition window are defined as follows:

Vehicle Type The name of the vehicle type.

Edit Graphics The Edit Graphics option allows you to define the size, color, and shape of graphics for vehicles of this type.

Vehicle Capacity The number of loads that vehicles of this type can carry at the same time during a simulation.This textbook only discusses single-capacity vehicles. For information about multiple capacity vehicles, see the “Path Mover System” chapter in volume 2 of the *AutoMod User’s Manual*, online.

Load Pick Up Time The amount of time that vehicles of this type require to pick up a load.

Load Set Down Time The amount of time that vehicles of this type require to set down a load.

Number of Vehicles The number of vehicles of this type in the system.

Vehicle Start List A list of locations where vehicles can be parked at the beginning of the simulation. The value “Random” indicates that vehicles start at randomly selected points in the system. For more information, see “Defining vehicle starting locations” on page 12.12 of the “Modeling Complex Material Handling Systems” chapter.

Specifications by Load Type The Specifications by Load Type option allows you to define different vehicle attributes, such as acceleration and velocity, based on the type of load the vehicle is carrying. This option is discussed in the next section.

Step 3 Change the Number of Vehicles to “3.” You have now created three vehicles of type “DefVehicle.”

You are now ready to define specifications by load type for the vehicles in the system.

Specifying vehicle attributes by load type

Defining a specification by load type allows you to describe how a vehicle behaves when it carries different types of loads. For example, vehicles might be required to carry very long loads or very heavy loads, which can change characteristics such as the accumulation distance between vehicles and the vehicle’s velocity.

The Specifications by Load Type list in the Edit A Vehicle Definition window contains two predefined specifications: default and empty. The default is used when a specification has not been defined for the type of load a vehicle is transporting. The empty specification is used by vehicles that are not transporting loads.

In example model 11.1, two load types are processed in the system: L_red and L_blue. You could define a specification for each load type that would cause vehicles to behave differently depending on whether they were carrying a red or blue load. In this system, however, load type has little effect on vehicle behavior, so you will edit the default specification to describe how vehicles behave when carrying either type of load.

To edit the default specification, do the following:

Step 1 Because the “Default” specification is already selected in the Specifications by Load Type list, ***click*** Edit to edit the specification. The Vehicle Specification window opens.

The Vehicle Specification window allows you to define the acceleration and deceleration rates of the three vehicles, as well as the velocity of the vehicles when traveling in a forward or reverse direction. You can also define separate velocities that the vehicles use when traveling on spur, curve, or crab paths. In addition, you can define the velocity of vehicles when rotating in the system and the space between vehicles when accumulating on a path.

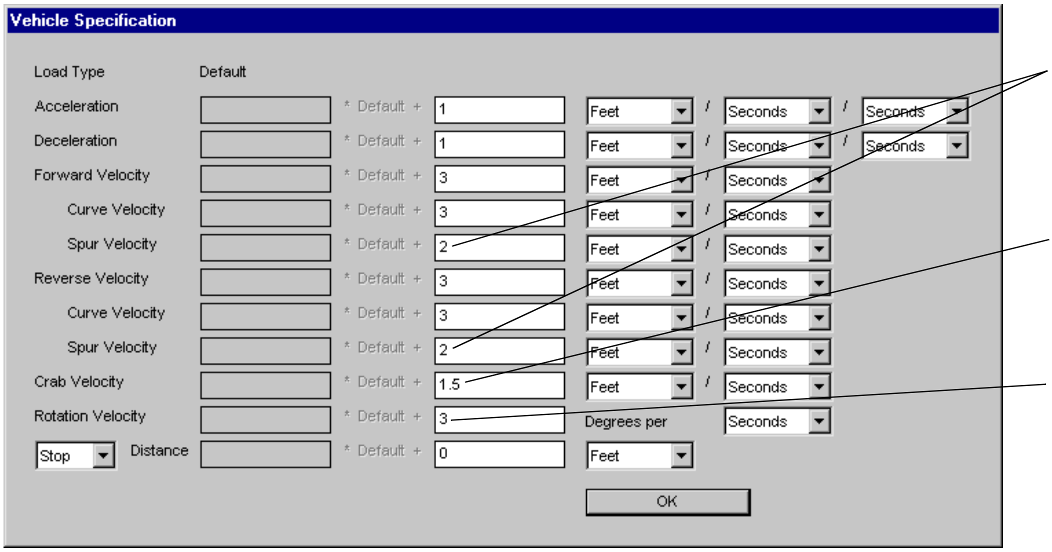
Note For more detailed information about defining vehicle attributes in a specification by load type, see the “Path Mover System” chapter in volume 2 of the *AutoMod User’s Manual*, online.

Step 2 ***Change*** the forward and reverse velocity of vehicles traveling on a spur path to be 2 feet per second.

Step 3 ***Change*** the velocity of vehicles traveling on a crab path to be 1.5 feet per second.

Note For crab paths, the crab velocity takes precedence over the other path velocities. For example, if a path is both a crab path and a spur path, vehicles use the crab velocity when traveling on the path.

Step 4 ***Change*** the velocity of vehicles when they are rotating to be 3 degrees per second. The Vehicle Specification window should appear as shown below:



Vehicle Specification

Load Type	Default							
Acceleration		* Default +	1	Feet	/	Seconds	/	Seconds
Deceleration		* Default +	1	Feet	/	Seconds	/	Seconds
Forward Velocity		* Default +	3	Feet	/	Seconds		
Curve Velocity		* Default +	3	Feet	/	Seconds		
Spur Velocity		* Default +	2	Feet	/	Seconds		
Reverse Velocity		* Default +	3	Feet	/	Seconds		
Curve Velocity		* Default +	3	Feet	/	Seconds		
Spur Velocity		* Default +	2	Feet	/	Seconds		
Crab Velocity		* Default +	1.5	Feet	/	Seconds		
Rotation Velocity		* Default +	3	Degrees per		Seconds		
Stop Distance		* Default +	0	Feet				

The forward and reverse velocities of vehicles that are traveling on spur paths is set to 2 feet per second

The velocity of vehicles that are traveling on crab paths is set to 1.5 feet per second

The velocity of rotating vehicles is set to 3 degrees per second

Defining the velocity of vehicles in example model 11.1

Step 5 ***Click*** OK to close the Vehicle Specification window. You have now edited the default velocities that vehicles use when carrying loads in the system.

Before closing the Edit A Vehicle Definition window, you need to place a graphic to represent the vehicles you have defined in the system (described next).

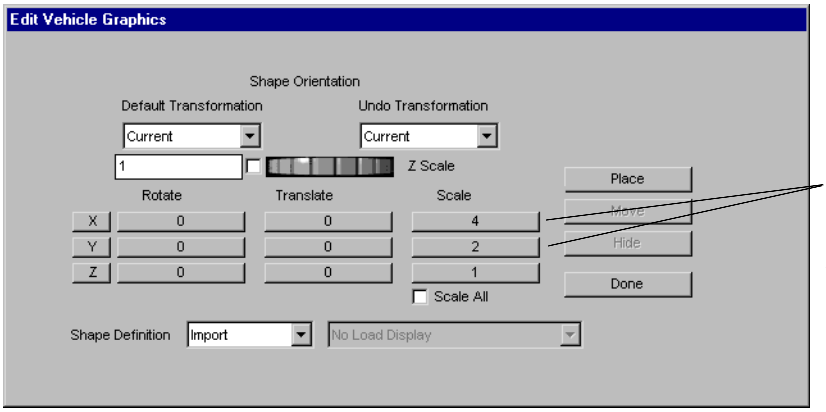
Placing vehicle graphics

You place vehicle graphics the same way that you place graphics for other entities.

To place vehicle graphics, do the following:

Step 1 In the Edit A Vehicle Definition window, ***click*** Edit Graphics. The Edit Vehicle Graphics window opens.

Step 2 ***Scale*** the vehicle graphic to 4 feet on the X axis and 2 feet on the Y axis, as shown below:



Default Transformation: Current

Undo Transformation: Current

1 □

	Rotate	Translate	Scale
X	0	0	4
Y	0	0	2
Z	0	0	1

Shape Definition: Import — No Load Display

Scale the vehicle graphic to 4 feet on the X axis and 2 feet on the Y axis

Scaling vehicle graphics in example model 11.1

Step 3 ***Click*** Place, then ***click*** to the left of the paths in the Work Area window to place the vehicle graphic in the system (as with load graphics, you can place vehicle graphics anywhere in the Work Area window). A rectangular box appears representing a vehicle of type "DefVehicle."

Step 4 In the Edit Vehicle Graphics window, ***click*** Done to close the window.

Step 5 In the Edit A Vehicle Definition window, ***click*** Done to close the window.

Step 6 ***Export*** the model.

Now that you have placed the vehicle graphic, you are ready to learn how to determine vehicle orientation during a simulation.

Determining vehicle orientation on a path

At the beginning of a simulation, vehicles are parked at a control point on a path and are oriented to begin traveling in a forward direction. The orientation of the vehicle depends on the path's type. If a vehicle starts at a control point on a *normal* path, the vehicle's X axis is aligned with the path. If a vehicle starts at a control point on a *crab* path, the vehicle's Y axis is aligned with the path.

The vehicle's graphic determines which edge of the vehicle is the "front" and which is the "back." In this textbook, you will use the default graphic for vehicles (either a square or rectangle). Using the default graphic, you cannot easily tell which end of the vehicle is the front or back; however, if the vehicle starts on a normal path, the **front** of the vehicle is the vehicle's leading edge at the beginning of a simulation.

Important

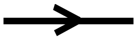
Throughout a simulation, vehicles maintain the alignment of their axes with the paths in the system: on a normal path, the X axis of a vehicle is always aligned with the path; on a crab path, the Y axis of a vehicle is always aligned with the path.

When a vehicle transfers from one path to another, the vehicle may need to rotate to maintain the alignment of its axes with the destination path. Whether or not the rotation takes any time, and whether the vehicle's direction of travel is forward or reverse after the transfer, depends on the amount of the vehicle's rotation and whether the destination path is a normal path or a crab path.

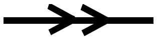
Determining vehicle orientation after a transfer to a normal path

When a vehicle transfers to a normal path, the vehicle will be traveling in either a forward or a reverse direction after the transfer. Whether the vehicle is traveling in a forward or reverse direction is important because the vehicle uses different velocities for each direction of travel (as defined in the vehicle's specification by load type).

A vehicle is traveling in a **forward direction** when the front of the vehicle is the leading edge in the path's forward direction of travel. A vehicle is traveling in a **reverse direction** when the back of the vehicle is the leading edge in the path's forward direction of travel. For normal one-directional and spur paths, the direction markers on the path indicate the path's **forward direction of travel**, as shown below:



One-directional path



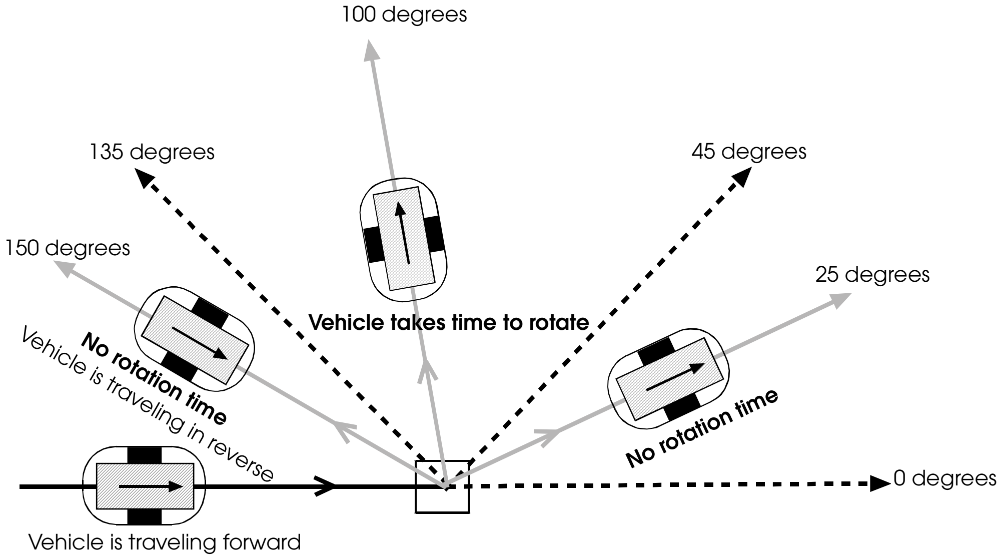
Spur path

In the illustrations above, both paths' forward direction of travel is to the right. For normal two-directional paths, the path's forward direction of travel is whichever direction the vehicle is currently traveling.

Whether a vehicle's rotation takes any time, and whether the vehicle's direction of travel changes after the transfer, depends on the amount of the vehicle's rotation, measured in degrees, as shown in the table below:

If the vehicle's rotation amount is...	The rotation takes time?	The vehicle's direction of travel...
Less than 45 degrees	No.	Remains the same (if the vehicle was traveling in a forward direction, it continues traveling in a forward direction; if the vehicle was traveling in reverse, it continues traveling in reverse).
Greater than or equal to 45 degrees, and less than or equal to 135 degrees	Yes; the vehicle rotates using the rotation velocity defined in its specification by load type.	Is forward (the front of the vehicle becomes the leading edge in the path's forward direction of travel).
Greater than 135 degrees	No.	Is reversed (if the vehicle was traveling in a forward direction, it travels in a reverse direction after the transfer; if the vehicle was traveling in a reverse direction, it travels in a forward direction after the transfer).

The illustration below shows the relation between a vehicle's rotation amount and the vehicle's direction of travel on normal paths. The paths in the illustration are all normal one-directional paths; however, the vehicle's behavior is the same for normal spur paths (with the same forward direction of travel) and normal two-directional paths.



Relation between a vehicle's rotation amount and direction of travel on a normal path

In the illustration above, the arrow inside the vehicle points to the front of the vehicle. Notice that the vehicle's direction of travel remains the same after transfers with a rotation amount less than or equal to 135 degrees, but reverses after transfers with a rotation amount greater than 135 degrees.

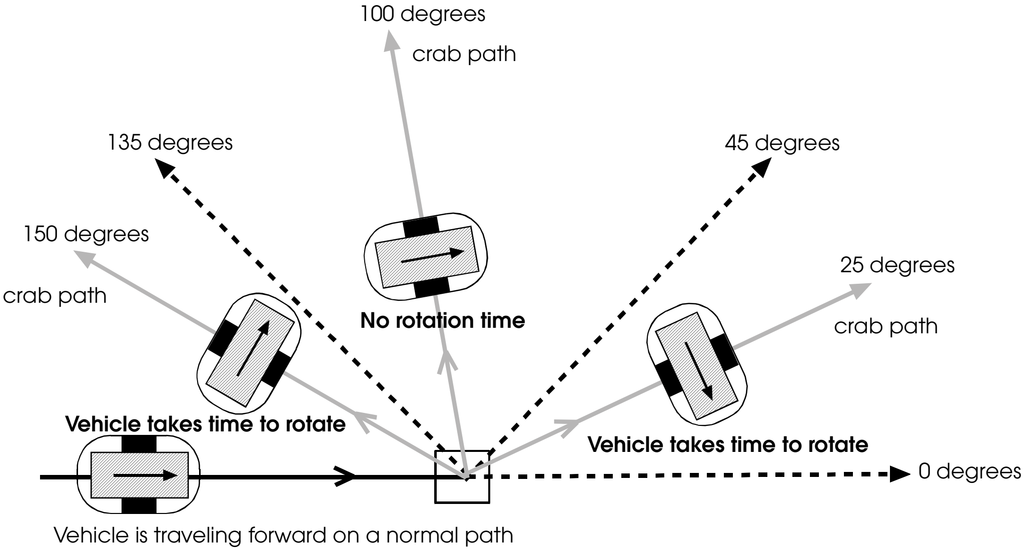
Determining vehicle orientation after a transfer to a crab path

When a vehicle transfers to a crab path, a forward or reverse direction of travel does not apply; the vehicle travels sideways using the velocity defined for crab paths in the vehicle's specification by load type. Either side of the vehicle can be the leading edge.

As with transfers to normal paths, whether or not the vehicle's rotation takes any time depends on the amount of the vehicle's rotation, measured in degrees, as shown in the table below:

If the vehicle's rotation amount is...	The rotation takes time?
Less than 45 degrees	No.
Greater than or equal to 45 degrees, and less than or equal to 135 degrees	Yes; the vehicle rotates using the rotation velocity defined in its specification by load type.
Greater than 135 degrees	No.

When a vehicle transfers to a crab path, the vehicle makes the shortest possible rotation to begin traveling sideways, as shown below:



Relation between a transfer's angle and a vehicle's orientation for crab paths

In the illustration above, the arrow inside the vehicle points to the front of the vehicle. Notice that when a vehicle transfers to a crab path, the amount of the vehicle's rotation is not the same as the path's transfer angle. Whether or not the vehicle's rotation takes any time is determined by the change in the vehicle's rotation, not the angle of the transfer. For example, when the vehicle transfers from a normal path to a crab path with a 25 degree angle, the amount of rotation required to orient the vehicle to begin traveling sideways is greater than 45 degrees, so the vehicle takes time to rotate.

Summary

You have successfully drawn a path mover system’s path and control points. You cannot run the model currently, because there are no vehicle scheduling lists defined to control vehicle movement in the system; you will add vehicle scheduling lists to the model in chapter 12, “Modeling Complex Material Handling Systems.”

In addition to drawing paths and placing control points, you have learned how to edit path attributes to create spur and crab paths, and you have learned how to edit control point attributes to limit the number of vehicles that can travel on a path. You also defined vehicles and their attributes and learned how to determine a vehicle’s orientation before and after a transfer.

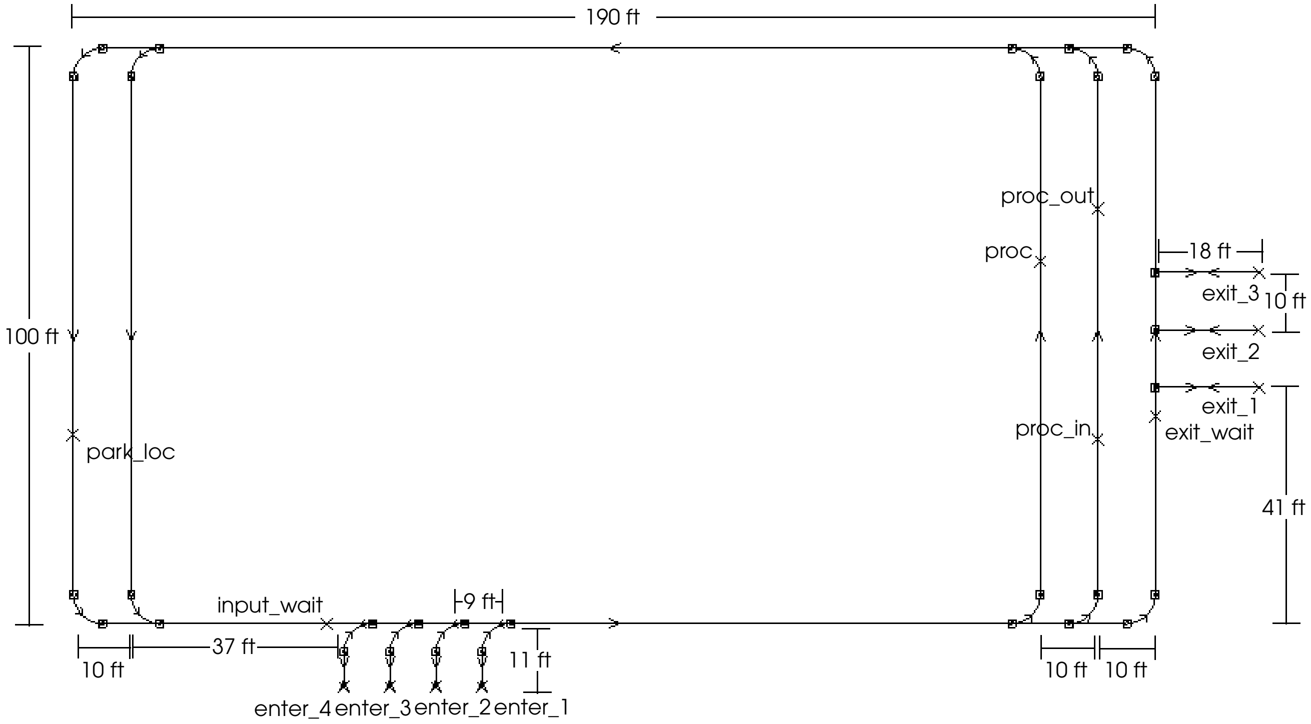
Exercises

Exercise 11.1

Note

The solution for this assignment is required to complete exercise 12.2 (see "Exercise 12.2" on page 12.30 of the "Modeling Complex Material Handling Systems" chapter); be sure to save a copy of your model.

Draw the path mover system shown below:



Place and name the control points as indicated (control point locations are approximate). The parking location, "park_loc," has capacity for three vehicles. The enter and exit paths are each two-directional paths and the enter and exit control points each have a capacity of one (to prevent vehicle collisions) and a control point release value of 25 feet (to prevent vehicle deadlocks). All other control points have infinite capacity and the default control point release value.

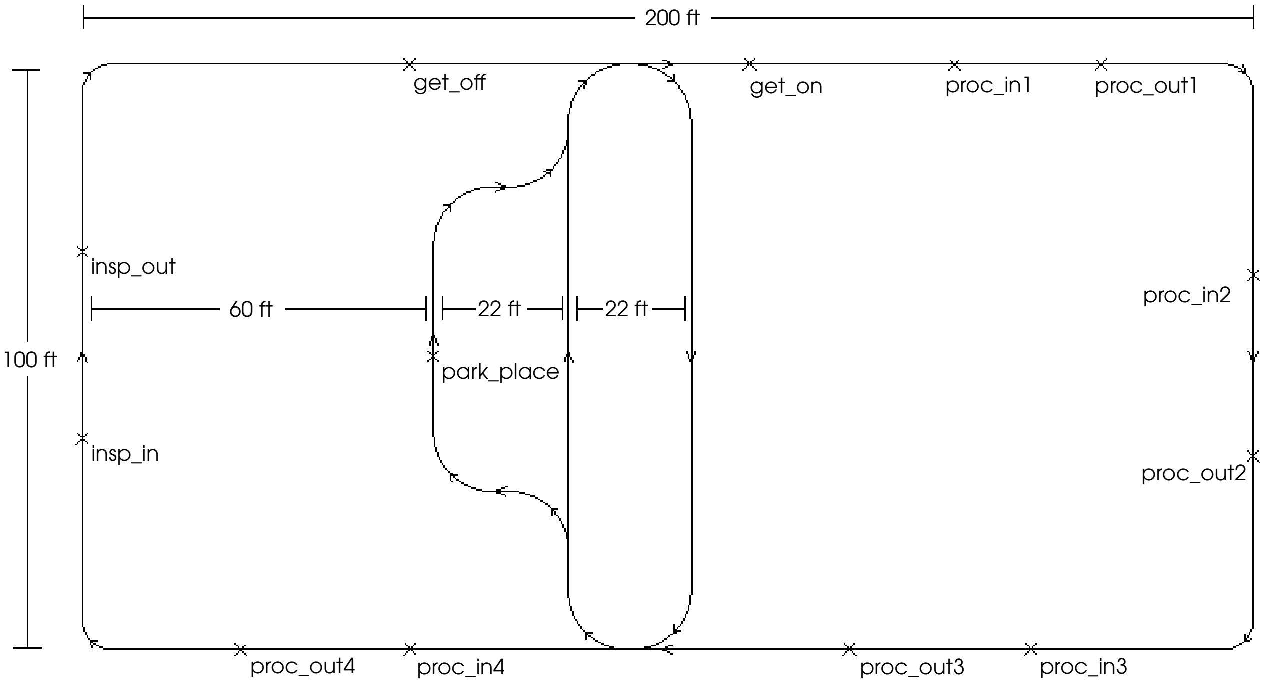
Place the graphics for vehicles; the vehicles' size is defined as:

X = 4
Y = 2
Z = 1

Exercise 11.2

Note The solution for this assignment is required to complete exercise 12.3 (see "Exercise 12.3" on page 12.32 of the "Modeling Complex Material Handling Systems" chapter); be sure to save a copy of your model.

Draw the path mover system shown below:



The curves on the outer loop of path each have a five foot radius. The curves on the inner loops of path each have a ten foot radius. Place and name the control points as indicated (control point locations are approximate). The parking location, "park_place," has capacity for three vehicles; all other control points have infinite capacity.

Place the graphics for vehicles; the vehicles' size is defined as:

X = 4
Y = 2
Z = 1

Chapter 12

Modeling Complex Material Handling Systems

Chapter 12

Modeling Complex Material Handling Systems

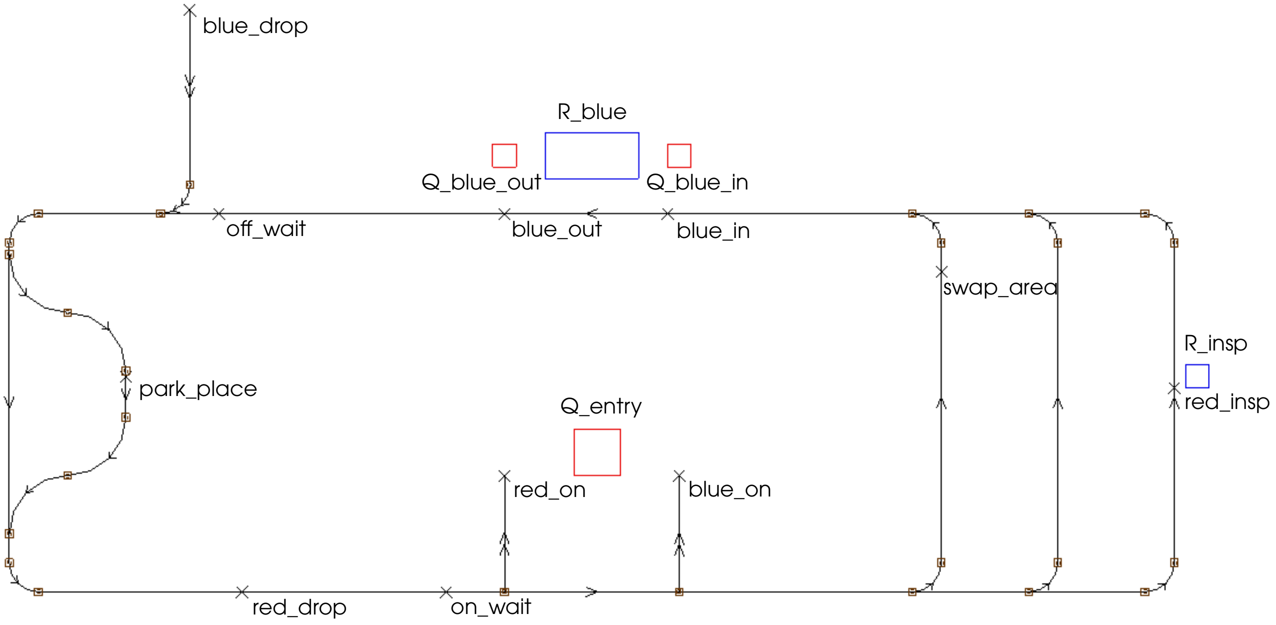
In chapter 11, “Introduction to Path Mover Systems,” you learned how to draw path mover systems. In this chapter, you will learn how to write model logic to move loads through a path mover system.

This chapter introduces scheduling lists, which is one method for controlling vehicle movement in a path mover system. By defining scheduling lists, you can control where vehicles travel to pickup loads or to park in the system. You will also learn how to define and place blocks, which can be used to prevent vehicle collisions and deadlocks.

Example 12.1: Drawing a path mover system

To learn how to control load and vehicle movement in a path mover system, you will complete the model of the path mover system that you created in chapter 11, "Introduction to Path Mover Systems." Currently, the model contains a drawing of a material handling system; vehicle graphics have been placed, and the necessary path and control point attributes have been defined. To complete the model, you will write the model logic, place the necessary queues and resources, and define scheduling lists to control vehicle movement in the path mover system.

An illustration and description of the system is provided below:



Layout of example model 12.1

Two types of loads are processed in this system: red loads and blue loads. Both types of loads have an interarrival time that is exponentially distributed with a mean of 5 minutes. Loads first move into an infinite-capacity queue (Q_entry). Loads are then picked up by a vehicle at one of two control points, depending on type; red loads are picked up at control point "red_on" and blue loads are picked up at control point "blue_on."

Red loads are carried to the control point "red_insp" where they are inspected (while onboard the vehicle) by an inspector for a time that is exponentially distributed with a mean of 3 minutes. After being inspected, red loads are carried to the control point "red_drop" where they get off the vehicle and leave the system.

After getting onboard a vehicle, blue loads are carried to the control point "blue_in" where they get off the vehicle and are placed in an infinite-capacity processing queue, Q_blue_in. Loads are processed in the queue by a single-capacity resource, R_blue, for a time that is normally distributed with a mean of 4 minutes and a standard deviation of 30 seconds. After completing processing, the loads move into an infinite-capacity queue, Q_blue_out, where they wait to be picked up by a vehicle at control point "blue_out." After being picked up, loads travel to the control point "blue_drop," where they get off the vehicle and leave the system.

Note

The system also contains a lane for replacing vehicle batteries; you will implement the battery swapping area later in this chapter (see "Example 12.2: Modeling battery replacement" on page 12.16 for more information).

Moving loads through a path mover system

Loads get on a vehicle and travel through a path mover system by executing the `move` and `travel` actions in an arriving procedure.

The `move` action causes loads to get on a vehicle at a specific control point. For example, the action:

```
move into pm:red_on
```

causes the load executing the action to move into the control point “red_on” in the path mover system “pm.” After moving into the control point, the load waits to be picked up by a vehicle. Once the load is on a vehicle, the `move` action is complete.

The `travel` action causes loads to travel between control points onboard a vehicle. For example, the action:

```
travel to pm:red_insp
```

causes the load executing the action to travel to the control point “red_insp” in the path mover system “pm.” A load must already have moved into a control point and been picked up by a vehicle before executing the `travel` action. Once the load arrives at the destination control point, the `travel` action is complete.

Note The `travel` action does *not* cause a load to get off a vehicle. To get off a vehicle, a load must move into another location (for example, a queue or a station in a conveyor system).

Defining the model logic in example model 12.1

To define the logic that controls load movement in example model 12.1, do the following:

Step 1 ***Import*** a copy of the model you created in chapter 11, “Introduction to Path Mover Systems.”

Note If you did not create the model, you can import the *base* version of example model 12.1, which is a completed drawing of the path mover system.

Step 2 ***Create*** a new process system named “proc.”

Step 3 ***Create*** a new source file named “logic.m.”

Step 4 ***Edit*** the source file and ***type*** the following logic.

```
begin P_agvsys arriving
   move into Q_entry
   if load type = L_red then
      begin
         move into pm:red_on    /*Get onboard a vehicle*/
         travel to pm:red_insp  /*Travel to “red_insp”*/
         use R_insp for e 3 min /*Load stays on vehicle*/
         travel to pm:red_drop  /*Travel to “red_drop”*/
         send to die            /*Get off vehicle (sent to die)*/
      end
   else if load type = L_blue then
      begin
         move into pm:blue_on          /*Get onboard a vehicle*/
         travel to pm:blue_in          /*Travel to “blue_in”*/
         move into Q_blue_in           /*Get off vehicle*/
         use R_blue for n 4, .5 min
         move into Q_blue_out
         move into pm:blue_out         /*Get on another vehicle*/
         travel to pm:blue_drop        /*Travel to “blue_drop”*/
         send to die                   /*Get off vehicle (sent to die)*/
      end
end
```

Take a moment to review the model logic. Refer to the example model description for an explanation of load activity in the simulation (see “Example 12.1: Drawing a path mover system” on page 12.4).

Notice that red loads remain onboard a vehicle during inspection (vehicles transporting red loads are delayed throughout the inspection process), while blue loads get off a vehicle for processing (after setting down a blue load for processing, vehicles are free to park or pick up another load).

Step 5 ***Save*** and ***quit*** the source file. When prompted, ***define*** the following entities:

P_agvsys – A single process.
Q_entry – A single queue with infinite capacity.
L_red – A load type with a creation rate that is exponentially distributed with a mean of 5 minutes; the red loads’ first process is P_agvsys.
R_insp – A single resource with a capacity of 1.
L_blue – A load type with a creation rate that is exponentially distributed with a mean of 5 minutes; the blue loads’ first process is P_agvsys.
Q_blue_in – A single queue with infinite capacity.
R_blue – A single resource with a capacity of 1.
Q_blue_out – A single queue with infinite capacity.

Step 6 ***Place*** the graphics for queues and resources, as shown in the illustration on page 12.4.

Step 7 ***Place*** the graphics for both load types (you can place the graphics anywhere in the Work Area window); ***set*** the loads’ color to be either red or blue, respectively, and ***scale*** the graphics of both L_red and L_blue loads to the following size:

X = 2
Y = 2
Z = 1

Controlling vehicles in a path mover system

Each vehicle in a path mover system is located at a control point at the beginning of a simulation. The first action the vehicle performs is to check its current control point to see whether a load is waiting to be picked up at that location.

Note Loads move into control points and wait to get on a vehicle by executing the `move` action in an arriving procedure (see “Moving loads through a path mover system” on page 12.5 for more information).

If a load is waiting at the vehicle’s current control point, the vehicle picks up the load. Once onboard a vehicle, the load determines the vehicle’s destination.

Note Loads determine a vehicle’s destination control point by executing a `travel` action in their current procedure (see “Moving loads through a path mover system” on page 12.5 for more information).

The vehicle automatically calculates the shortest route to the load’s destination and takes that route to deliver the load. After arriving at the delivery location, the vehicle’s next action is determined by the load. The load can perform processing while onboard the vehicle (which delays the vehicle), it can travel to another destination (which causes the vehicle to calculate the shortest route and travel to the new destination), or it can get off the vehicle by moving into another territory (for example, a queue or a conveyor station). If a load gets off the vehicle, the empty vehicle again checks its current location to see if a load is waiting to be picked up.

If an empty vehicle checks its current control point for a waiting load, but there is no load requiring pick up, the vehicle attempts to find work or a parking location using the scheduling lists defined at its current control point. A **scheduling list** is a list of one or more control points in the path mover system. Vehicles use scheduling lists to determine their next destination. There are several types of scheduling lists; in this textbook, we discuss the following three types:

Work lists A work list is a set of locations at which a vehicle can search for loads that are waiting to be picked up. A vehicle searches the locations on the work list without leaving its current control point. When a load is found, the vehicle calculates the shortest route to the pickup location and travels to that point to retrieve the load.

Park lists A park list is a set of locations at which a vehicle can search for available parking. A vehicle searches the locations on the park list without leaving its current control point. When a control point with available capacity is found, the vehicle travels to the parking location.

Named lists A named list is a set of locations that can be referenced in a work or park list. Named lists allow you to easily create and sort work or park lists for several different control point locations. Named lists also allow you to define the starting locations of vehicles in a system.

When a vehicle is empty at a control point, the vehicle completes the following steps:

1. The vehicle looks for a load that is waiting to be picked up at its current location; if a load is found, the vehicle picks up the load.
2. If no load is found at the vehicle's current location, the vehicle searches the locations on the work list defined for its current location. If a waiting load is found, the vehicle travels to the pickup location and picks up the load.
3. If no load is found on the work list, the vehicle searches the locations on the park list defined for its current location. If a parking location with available capacity is found, the vehicle travels to the parking location and looks for work at that location.
4. If no parking locations are found, the vehicle parks at its current location and becomes idle until it is wakened, either by a load at its current location or a load at another location in the system (see the next section for more information).

Important

It is important to define scheduling lists for control points where vehicles park or set down loads in a system. If a vehicle is empty at a control point that does not have any scheduling lists defined, the vehicle parks and becomes idle at its current location. If vehicles park at incorrect locations in the system, they may block other vehicles that are delivering or picking up loads.

How loads waken idle (parked) vehicles in a system

A vehicle becomes idle when it searches the work and park lists defined for its current location, but does not find a load that is waiting to be picked up or a parking location with available capacity. Idle vehicles park and wait to be wakened by loads in the system.

When a load moves into a control point, the load completes the following steps:

1. The load looks for an idle vehicle at its current location. If there is an idle vehicle, the load wakens and gets on that vehicle.
2. If there is no idle vehicle at the load's current location, the load wakens all idle vehicles in the system and then waits to be picked up. When idle vehicles awake, they search their current location for loads that are waiting to be picked up, and then search the work and park lists defined for their current location to find loads that are waiting at other control points in the system.

Defining locations where vehicles can search for work

When a vehicle cannot find work at its current control point, the vehicle checks to see whether a work list has been defined for the point. Vehicles search the locations on a work list in order, from top to bottom, until a load is found that is waiting to get on a vehicle. In general, you should define a work list for every point where a vehicle can park or set down a load in the system.

In example model 12.1, vehicles set down loads at the locations “blue_in,” “red_drop,” and “blue_drop,” and park at the location “park_place.” You need to define a work list at each of these locations.

If a vehicle sets down a load at the control point “blue_in,” where should the vehicle look for work? Loads wait to get on a vehicle at three pickup locations in the system: “red_on,” “blue_on,” and “blue_out.”

Tip ☞ Vehicles will be more efficient if they search for work at pickup locations in the order of the locations’ distance to the vehicles’ current control point. Consequently, you should add control points to work lists in order of distance, with the closest control point listed first.

When a vehicle sets down a load at control point “blue_in,” the closest location where work is available is “blue_out,” followed by “red_on,” then “blue_on” (see “Example 12.1: Drawing a path mover system” on page 12.4 for an illustration of the system).

To define work lists for the control points in example model 12.1, do the following:

Step 1 ***Open*** the path mover system.

Step 2 On the Path Mover palette, ***click*** Work List. The Work Lists window opens.

Step 3 ***Click*** New to define a new work list. The New Work List window opens.

Step 4 ***Select*** “blue_in,” then ***click*** New to define a new work list for that control point. The Edit Work Lists window opens.

Step 5 To add a location to the work list, ***click*** Add After. The Add Work List Locations window opens.

Step 6 To add the first pickup location for blue loads, ***select*** “blue_out,” then ***click*** Add. The control point is added to the work list in the Edit Work Lists window.

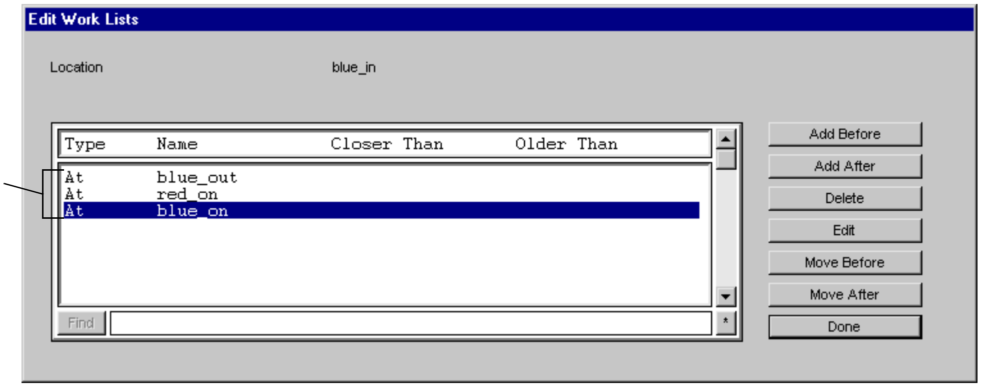
Step 7 ***Click*** Add After to add another location. The Add Work List Locations window opens.

Step 8 To add the pickup location for red loads, ***select*** “red_on,” then ***click*** Add. The control point is added to the work list in the Edit Work Lists window.

Step 9 ***Click*** Add After to add another location. The Add Work List Locations window opens.

Step 10 To add the second pickup location for blue loads, ***select*** “blue_on,” then ***click*** Add. The control point is added to the work list in the Edit Work Lists window.

The Edit Work Lists window appears as shown below:



Location blue_in

Type	Name	Closer Than	Older Than
At	blue_out		
At	red_on		
At	blue_on		

Pickup locations are added in order of their distance to the current drop-off location (“blue_in”)

Defining the work list for control point “blue_in”

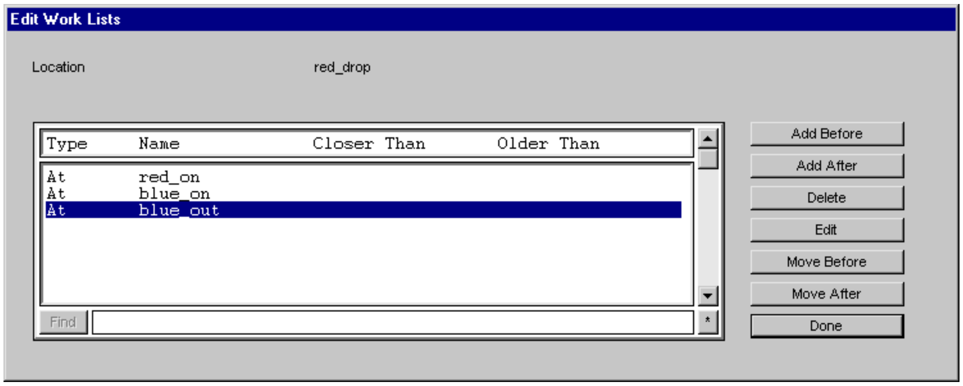
Step 11 In the Edit Work Lists window, ***click*** Done.

You have now defined a work list for the control point “blue_in.” When a vehicle sets down a blue load, the vehicle will look for work at the closest control point where loads wait to get on a vehicle, “blue_out, ” followed by the point “red_on,” then the point “blue_on.”

Now consider a vehicle that sets down a load at control point “red_drop.” Where should the vehicle look for work? The closest pickup location is “red_on,” followed by “blue_on,” and lastly “blue_out.”

Step 12 ***Create*** the work list for the control point “red_drop.”

After adding the last point, the Edit Work Lists window appears as shown below:



Location red_drop

Type	Name	Closer Than	Older Than
At	red_on		
At	blue_on		
At	blue_out		

Defining the work list for control point “red_drop”

Step 13 In the Edit Work Lists window, ***click*** Done.

You have now defined a work list for the control point “red_drop.” When a vehicle sets down a red load, the vehicle will look for work at the closest control point where loads wait to get on a vehicle, “red_on,” followed by the point “blue_on,” then the point “blue_out.”

You still need to define a work list for the second point where vehicles set down blue loads (“blue_drop”) and for the parking location (“park_place”). Notice that, at the remaining points, the order in which vehicles should search the pickup locations for work is the same as the order for the location “red_drop.” You can save time when defining the work lists for the remaining two points by copying the work list you just created for the point “red_drop.”

Copying scheduling lists

To copy the work list defined for the control point “red_drop,” do the following:

Step 1 In the Work Lists window, ***select*** “red_drop,” then ***click*** Copy. The Copy Work List window opens.

Step 2 ***Select*** “blue_drop,” then ***click*** Copy. A copy of the work list is created for the control point “blue_drop.”

Step 3 In the Work Lists window, ***click*** Copy to create another copy of the list. The Copy Work List window opens.

Step 4 ***Select*** “park_place,” then ***click*** Copy. A copy of the work list is created for the control point “park_place.”

You have now created a work list for each of the points where vehicles park or set down loads in the example model.

Defining locations where vehicles can search for parking

When a vehicle cannot find work on a work list, the vehicle searches for a parking location on the park list defined for its current control point. Vehicles search the locations on a park list in order, from top to bottom, until a parking location with available capacity is found. The vehicle then travels to the parking location and resumes looking for work. You should define a park list at every point where a vehicle sets down loads in the system.

Note If a system contains multiple parking locations, you can define a park list at each of the parking locations to cause empty vehicles to continuously travel between parking control points until work is found.

In example model 12.1, vehicles set down loads at the locations “blue_in,” “red_drop,” and “blue_drop.”

To create a park list for the control points where vehicles set down loads, do the following:

Step 1 On the Path Mover palette, ***click*** Park List. The Park Lists window opens.

Step 2 ***Click*** New to define a new park list. The New Park List window opens.

Step 3 ***Select*** “blue_drop,” then ***click*** New to define a new park list for that control point. The Edit Park Lists window opens.

Step 4 To add a location to the park list, ***click*** Add After. The Add Park List Locations window opens.

Step 5 To add the parking location to the park list, ***select*** “park_place,” then ***click*** Add. The control point is added to the park list in the Edit Park Lists window.

Step 6 In the Edit Park Lists window, ***click*** Done.

Step 7 ***Copy*** the park list for “blue_drop” for the control points “red_drop” and “blue_in.”

You have now defined a park list for each of the control points where vehicles set down loads. When a vehicle sets down a load, the vehicle will look for work; if no work is found, the vehicle will travel to the control point “park_place.” Vehicles remain idle at the control point “park_place” until they are wakened by loads in the system (see “How loads waken idle (parked) vehicles in a system” on page 12.8 for more information).

Defining vehicle starting locations

By default, each vehicle in the path mover system starts at a random control point. Random placement of vehicles is usually not a good method of starting vehicles in a system. For example, the software could randomly place a vehicle at a control point without a work or park list; as a result, the vehicle would park and remain idle throughout the simulation, waiting for a load to require pickup at the vehicle's current point.

You can control where vehicles start in the system by defining a named list and assigning it to a vehicle type. If the named list contains only one control point, all of the vehicles start at that point.

Note A control point's capacity is not a consideration when selecting the starting location for vehicles, because vehicles are added to the control point one at a time (a new vehicle is not added until the previous vehicle has left).

If the named list contains multiple control points, vehicles are sequentially assigned to each point in the list, in order from top to bottom.

In example model 12.1, all vehicles must start at the control point "park_place." To create a named list that causes vehicles to start at the parking location, do the following:

Step 1 On the Path Mover palette, ***click*** Named List. The Named Lists window opens.

Step 2 ***Click*** New to define a new named list. The New Named List window opens.

Step 3 In the List Name text box, ***type*** "start_here", then ***click*** Create. The Edit Named List window opens.

Step 4 ***Select*** "park_place" in the Location Selection List, then ***click*** Add After. The parking location is added to the named list.

Step 5 ***Click*** Done to close the Edit Named List window.

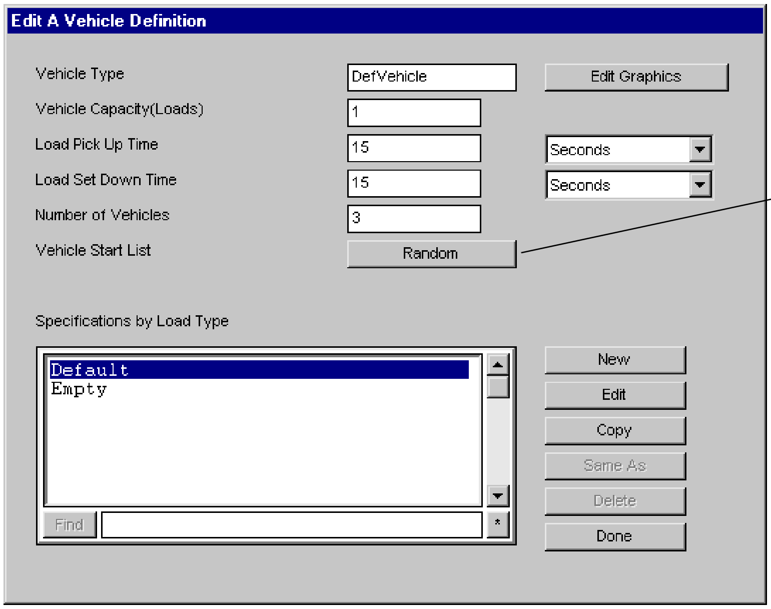
You have now created a named list that contains only the parking location "park_place." However, you still need to define the list as the starting location for vehicles of type "DefVehicle."

Step 6 On the Path Mover palette, ***click*** Vehicle. The Vehicles window opens.

Step 7 The vehicle type "DefVehicle" is already selected, so ***click*** Edit to edit the vehicle definition. The Edit A Vehicle definition window opens.

Note Named lists can also be referenced in other scheduling lists. Referencing a named list allows you to easily add the same locations to multiple work or park lists. Named lists also allow vehicles to automatically sort locations by distance or by loads that have been waiting the longest for pickup. Referencing and sorting named lists is not discussed in this textbook.

Step 8 To define the vehicle start list, ***click*** Random, as shown below:



Edit A Vehicle Definition

Vehicle Type: DefVehicle

Vehicle Capacity(Loads): 1

Load Pick Up Time: 15 Seconds

Load Set Down Time: 15 Seconds

Number of Vehicles: 3

Vehicle Start List: Random

Specifications by Load Type:

```
Default
Empty
```

Click Random to define the Vehicle Start List

Changing the Vehicle Start List in example model 12.1

The Vehicle Start List window opens.

Step 9 ***Select*** “start_here,” then ***click*** OK.

Step 10 ***Click*** Done to close the Edit A Vehicle Definition window.

Vehicles in the system now start at the control point “park_place.”

Interpreting statistics in example model 12.1

Now that you have defined the process system and the vehicle scheduling lists necessary to simulate the example model, you are ready to run the model and analyze the simulation:

Step 1 ***Open*** the process system, then ***click*** Run Control on the Process System palette.

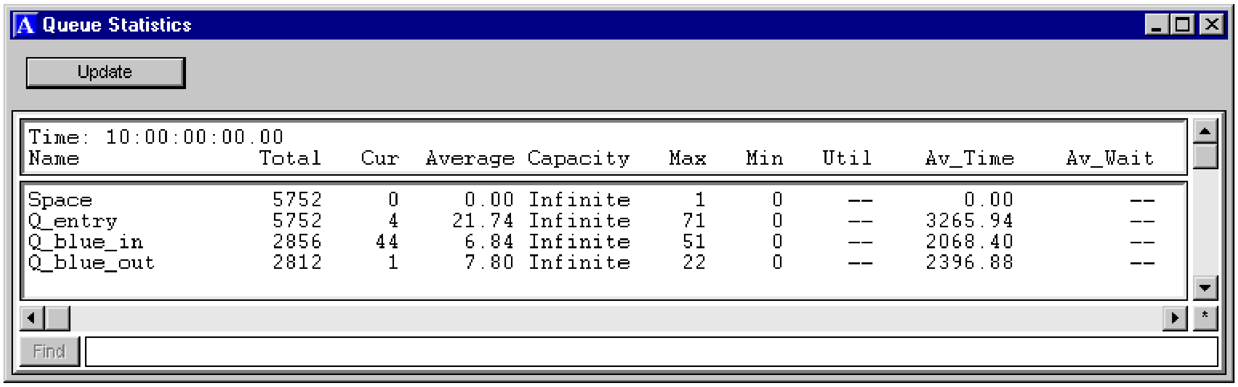
Step 2 ***Define*** a snap of 10 days.

Step 3 ***Export*** and ***run*** the model to completion.

Note During simulation, warnings appear in the Message window indicating that vehicle collisions have occurred. You will learn how to block vehicle collisions later in this chapter; for now, ignore the warning messages.

When the simulation is complete, do the following:

Step 4 From the Queues menu, ***select*** Statistics Summary. The Queue Statistics window opens, as shown below:

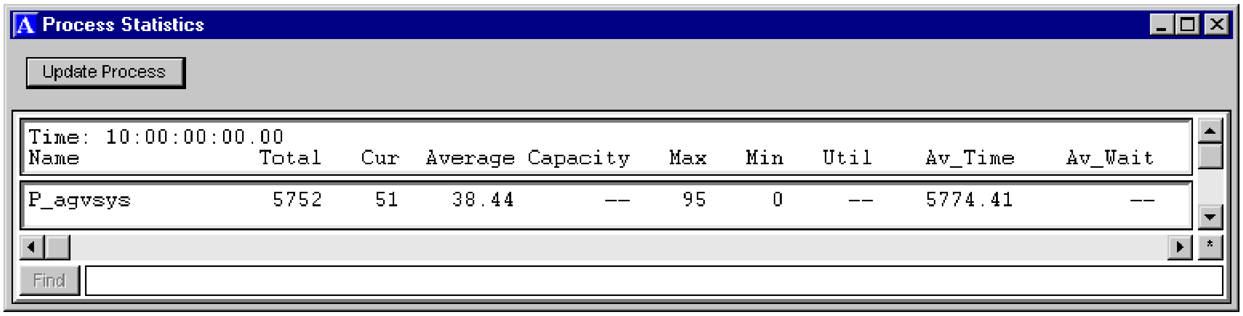


```
Time: 10:00:00:00.00
Name                  Total    Cur  Average Capacity   Max   Min   Util     Av_Time     Av_Wait

Space                  5752      0     0.00 Infinite     1     0    --         0.00          --
Q_entry                5752      4    21.74 Infinite    71     0    --      3265.94          --
Q_blue_in              2856     44     6.84 Infinite    51     0    --      2068.40          --
Q_blue_out             2812      1     7.80 Infinite    22     0    --      2396.88          --
```

Queue summary statistics in example model 12.1

From the statistics for the queue Q_entry, we see that there is an average of 21.74 loads that wait to enter the system; these loads wait an average time of 54.43 minutes (3265.94 seconds) to get on a vehicle.

Step 5 From the Processes menu, ***select*** Statistics Summary. The Process Statistics window opens, as shown below:



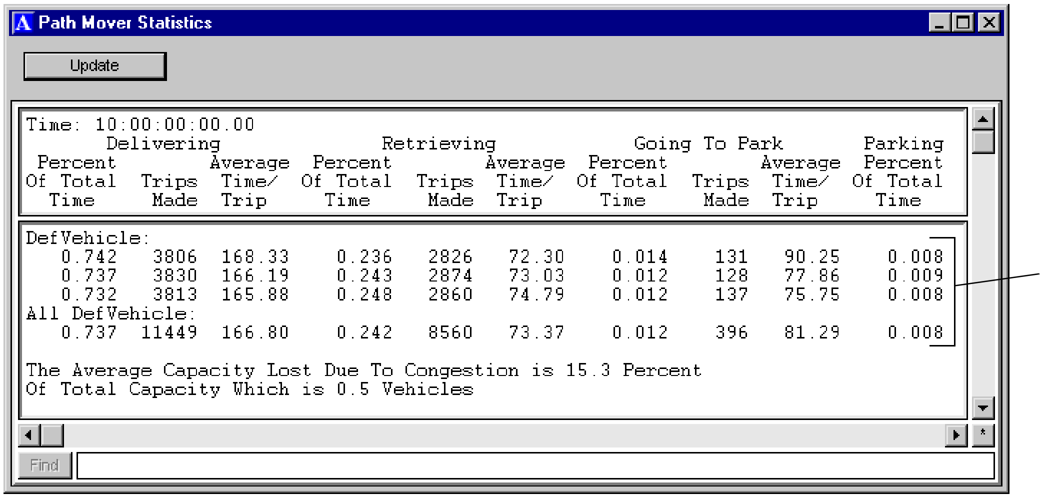
```
Time: 10:00:00:00.00
Name                  Total    Cur  Average Capacity   Max   Min   Util     Av_Time     Av_Wait

P_agvsys               5752     51    38.44       --    95     0    --      5774.41          --
```

Process summary statistics in example model 12.1

The average time that loads spend in the system is 96.24 minutes (5774.41 seconds).

You may be able to reduce the number of loads that wait, the waiting time of each load, and the time that loads spend processing, by increasing the number of vehicles in the system. To check the utilization of vehicles, look at the vehicle statistics.

Step 6 From the Path Mover menu, ***select*** Vehicles. The Path Mover Statistics window opens, as shown below:



```
Time: 10:00:00:00.00
       Delivering                Retrieving                Going To Park         Parking
 Percent        Average  Percent        Average  Percent        Average  Percent
Of Total  Trips  Time/  Of Total  Trips  Time/  Of Total  Trips  Time/  Of Total
  Time     Made  Trip     Time     Made  Trip     Time     Made  Trip     Time

DefVehicle:
   0.742   3806  168.33    0.236   2826   72.30    0.014    131   90.25    0.008
   0.737   3830  166.19    0.243   2874   73.03    0.012    128   77.86    0.009
   0.732   3813  165.88    0.248   2860   74.79    0.012    137   75.75    0.008
All DefVehicle:
   0.737  11449  166.80    0.242   8560   73.37    0.012    396   81.29    0.008

The Average Capacity Lost Due To Congestion is 15.3 Percent
Of Total Capacity Which is 0.5 Vehicles
```

Vehicles spend very little time parking, indicating high utilization of vehicles in this system

Vehicle statistics in example model 12.1

These statistics suggest the need for additional vehicles in the system. The percentage of time that each vehicle spends idle (parked) is less than one-hundredth of a percent.

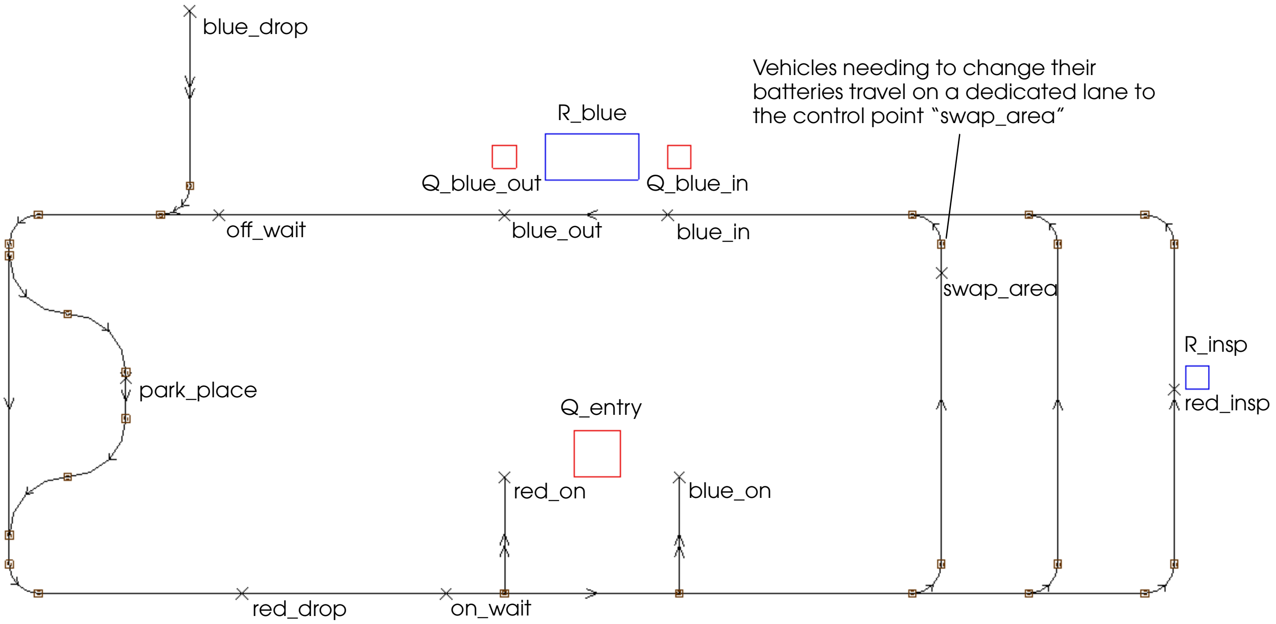
The average capacity lost due to vehicle congestion is 15.3 percent. In this model, vehicle congestion is largely due to delays at the spur paths where vehicles travel one at a time.

From this preliminary analysis, you can see the need to conduct further cost analysis using the AutoStat software; by comparing the cost of WIP with the cost of adding additional vehicles in the system, you could determine the optimal number of vehicles to operate in the system. This analysis is the focus of exercise 12.1 (see “Exercise 12.1” on page 12.30).

Now, you are ready to implement the battery replacement area in the example model.

Example 12.2: Modeling battery replacement

AGVs are often powered by batteries that must be replaced when their charge is running low. You will expand the material handling system in example model 12.1, so that vehicles use a dedicated lane for battery replacement, as shown below:



Layout of example model 12.2

The time between battery replacements is determined by a normal distribution with a mean of 8 hours and a standard deviation of 1 hour. A technician replaces each vehicle's battery; the technician can only replace the battery for one vehicle at a time. The time required to replace a battery is 15 minutes.

Using process attributes and system attributes

In order to implement the battery replacement area, you will use two standard entity attributes:

total The `total` attribute is a standard attribute of processes that indicates the total number of loads that have been sent to a process. Think of the `total` attribute as a unique integer value belonging to each process; when a load is sent to a process, the process' `total` is incremented by one. You can use the `total` attribute to refer to the number of loads that have been sent to a process at any given moment during a simulation.

When using the `total` attribute in logic, it must be preceded by the name of the process for which you want the total. For example, the syntax `P_Start total` represents the number of loads that have been sent to the process P_start.

vehicles size The `vehicles size` attribute is a standard attribute of path mover systems that indicates the number of vehicles that are in a path mover system. The value of the `vehicles size` attribute is equal to the total number of vehicles (of any vehicle type) that you have defined in the system.

When using the `vehicle size` attribute in logic, it must be preceded by the name of the system for which you want the number of vehicles. For example, the syntax `pm vehicle size` represents the number of vehicles in the path mover system named "pm."

To see how these attributes are used in the model logic, define the logic for the battery replacement area, as described in the next section.

Defining the model logic in example model 12.2

To define the logic that simulates battery replacement, do the following:

Note

If you have created the example model in this chapter, you can continue using your current model. If you have not created the model but want to start at this point, you can import a copy of the *base* version of example model 12.2.

Step 1 ***Edit*** the source file named “logic.m.”

Step 2 ***Insert*** the following function and procedure at the beginning of the source file:

```
begin model initialization function
   create (pm vehicles size) loads of type L_dummy to P_swap
   return true
end

begin P_swap arriving
   set A_index to P_swap total
   wait for 480*(A_index - 1)/(pm vehicles size) min /* stagger swaps */
   while 1=1 do
      begin
         wait for n 8, 1 hr         /* time between swaps */
         move into pm:swap_area     /* wait to get on a vehicle */
         use R_swap for 15 min      /* swap time */
         move into Q_dummy          /* get off vehicle */
      end
end
```

This logic is written so that any number of vehicles can be added to the simulation without needing to edit the battery replacement logic.

First, the model initialization function creates the same number of dummy loads as there are vehicles in the path mover system. The function sends each load to the P_swap arriving procedure (one load for each vehicle).

When the first load executes the P_swap arriving procedure, the value of the `total` attribute is one; the load’s attribute A_index is set to this value. When the second load executes the P_swap arriving procedure, the value of the `total` attribute is two; the load’s attribute A_index is set to that value, and so on.

The A_index value is then used in a `wait` action that delays each load for a different amount of time. The time is calculated using a fraction of the mean time between battery replacements (8 hours, or 480 minutes). The first load is delayed for 0 minutes (480*0/3), the second load is delayed for 160 minutes (480*1/3), and the third load is delayed for 320 minutes (480*2/3). These delays are used to stagger the battery swap times so that they do not all occur around the same time during the simulation.

The logic that causes vehicles to travel to the battery replacement area is defined in a `while...do` loop that repeats throughout the simulation. First, each load delays for a time that is normally distributed with a mean of 8 hours and a standard deviation of 1 hour (this delay represents the time between battery replacements). The loads then move into the control point “swap_area,” where they wait to get on a vehicle. (Later, you will edit the work lists defined in the model to cause vehicles to look for work at that location.) A vehicle that is searching for work will find the dummy load and travel to the “swap_area” point to pick up the load. Once onboard the vehicle, the load uses the resource R_swap for 15 minutes (the vehicle is delayed at the control point while the load is using the resource). After 15 minutes, the load gets off the vehicle by moving into a dummy queue, which releases the vehicle to travel to another location.

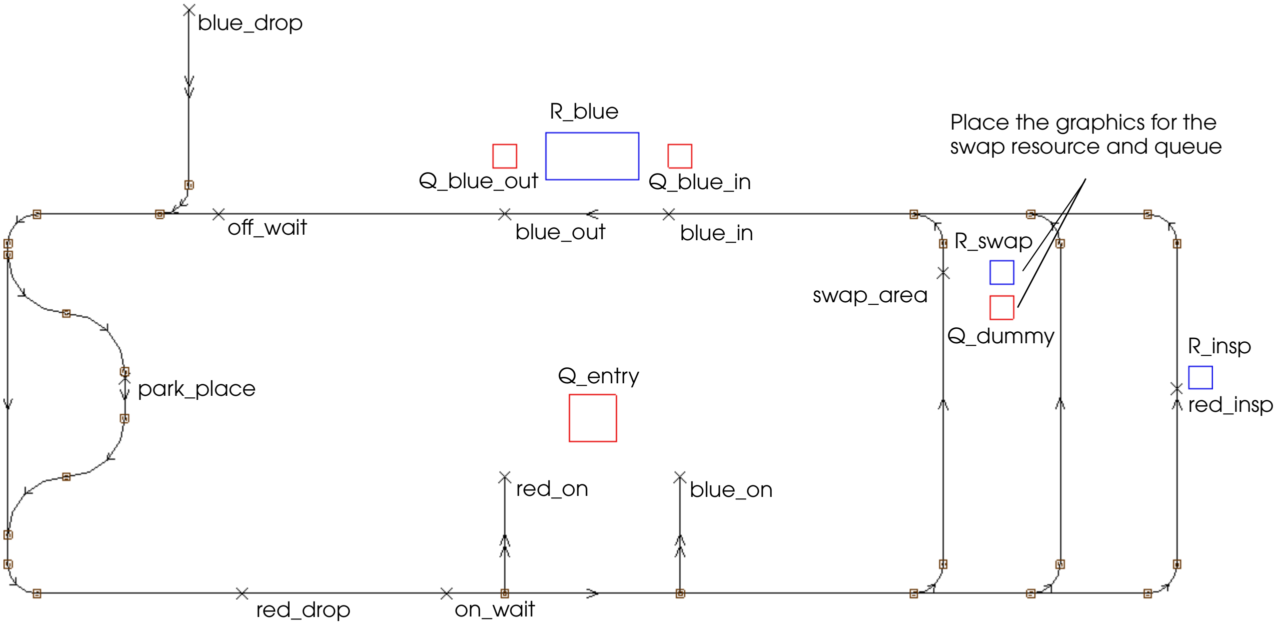
Note

In this simulation, it is not important which vehicle travels to the control point and is delayed for the battery swapping time (the same vehicle could by delayed at the swap area multiple times in sequence); it is only important that a vehicle be prevented from picking up and delivering loads during this time period. If more accurate simulation is required, you could assign specific vehicles to travel to the swap area by changing the vehicle's currently scheduled job (not discussed in this textbook). For information about scheduled jobs, see the "Vehicle Scheduling" chapter in volume 2 of the *AutoMod User's Manual*, online.

Step 3 ***Save*** and ***quit*** the source file. When prompted, ***define*** the following entities:

L_dummy – A load type with no creation rate.
P_swap – A single process.
A_index – A load attribute of type integer.
R_swap – A single resource with a capacity of 1.
Q_dummy – A single queue with infinite capacity.

Step 4 ***Place*** the graphics for the resource and queue, as shown below:



Place the graphics for the swap resource and queue

Placing the resource and queue graphics in example model 12.2

Before running the simulation, you need to edit the work lists in the example model to cause vehicles to look for work at the "swap_area" control point.

Editing work lists

To cause vehicles to look for work at the “swap_area” control point, you need to edit the work lists at the control points where vehicles park or set down loads in the system. When an idle vehicle’s battery is running low, battery replacement should take a higher priority than picking up waiting red or blue loads. To ensure that vehicles check the “swap_area” control point first, you will insert that control point at the top of each existing work list.

To edit the scheduling lists:

Step 1 On the Path Mover palette, ***click*** Work List. The Work Lists window opens.

Step 2 The point “blue_drop” is already selected, so ***click*** Edit to edit the list. The Edit Work Lists window opens.

Step 3 ***Select*** the first point on the list, in this case point “red_on,” then ***click*** Add Before to insert a point at the top of the list. The Add Work List Locations window opens.

Step 4 ***Select*** “swap_area,” then ***click*** Add. The point is inserted at the beginning of the list.

Step 5 ***Click*** Done to close the Edit Work Lists window.

Step 6 ***Repeat*** these steps to insert the “swap_area” control point at the beginning of the work lists for the points “blue_in,” “park_place,” and “red_drop.”

You have now edited all work lists in the model to cause vehicles to look for work at the “swap_area” control point first.

One other change is needed. Because vehicles set down a load and become idle at the “swap_area” control point, you need to define a work and park list for this location.

Defining a work and park list for the swap area

After the dummy load at the point “swap_area” completes delaying for the swap time, the load moves into the queue Q_dummy and releases the current vehicle. To cause the vehicle to leave the point “swap_area,” you need to define a work and park list for the point.

To define the lists:

Step 1 ***Create*** a work list for the “swap_area” control point that causes vehicles to look for work at the following points (in order, from top to bottom):

- blue_out
- red_on
- blue_on

Step 2 ***Create*** a park list for the “swap_area” control point that causes vehicles to look for parking at the point “park_place.”

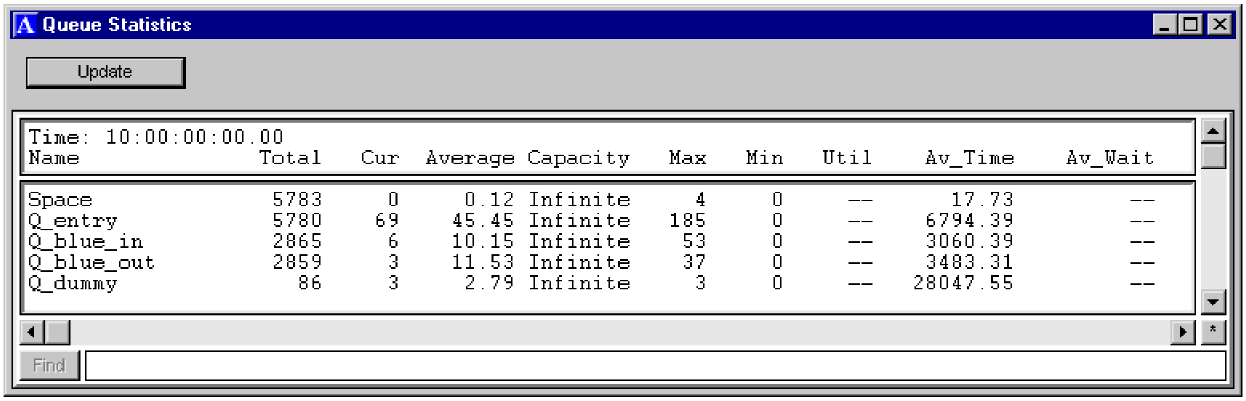
You have now defined the necessary scheduling lists to run example model 12.2.

Interpreting statistics in example model 12.2

Now that you have implemented the battery replacement area, you are ready to run the model and analyze the simulation:

Step 1 ***Export*** and ***run*** the model to completion.

Step 2 From the Queues menu, ***select*** Statistics Summary. The Queue Statistics window opens, as shown below:

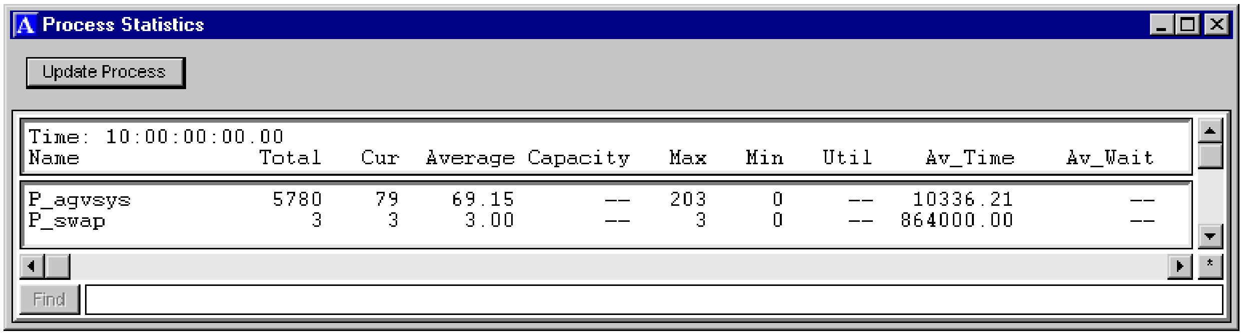


```
Time: 10:00:00:00.00
Name                  Total    Cur  Average Capacity   Max   Min   Util    Av_Time     Av_Wait
Space                  5783      0     0.12 Infinite     4     0     --      17.73          --
Q_entry                5780     69    45.45 Infinite   185     0     --    6794.39          --
Q_blue_in              2865      6    10.15 Infinite    53     0     --    3060.39          --
Q_blue_out             2859      3    11.53 Infinite    37     0     --    3483.31          --
Q_dummy                  86      3     2.79 Infinite     3     0     --   28047.55          --
```

Queue summary statistics in example model 12.2

The statistics for the queue Q_entry shows that there is an average of 45.45 loads that wait to enter the system, as compared with 21.74 loads before the battery replacement area was implemented. Loads wait an average time of 113.24 minutes (6794.39 seconds) to get on a vehicle, as compared with 54.43 minutes before the battery replacement area was implemented.

Step 3 From the Processes menu, ***select*** Statistics Summary. The Process Statistics window opens, as shown below:

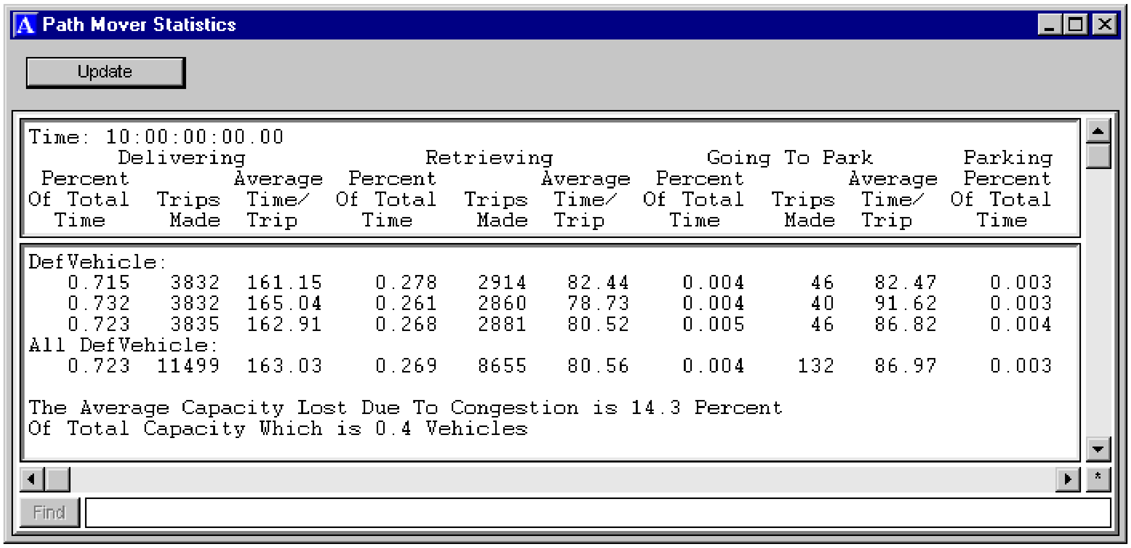


```
Time: 10:00:00:00.00
Name                  Total    Cur  Average Capacity   Max   Min   Util    Av_Time     Av_Wait
P_agvsys               5780     79    69.15       --   203     0     --   10336.21          --
P_swap                    3      3     3.00       --     3     0     --  864000.00          --
```

Process summary statistics in example model 12.2

Loads spend an average time of 172.27 minutes (10336.21 seconds) in the system, as compared with 96.24 minutes before the battery replacement area was implemented.

Step 4 From the Path Mover menu, ***select*** Vehicles. The Path Mover Statistics window opens, as shown below:



```
Time: 10:00:00:00.00
       Delivering               Retrieving              Going To Park          Parking
 Percent          Average  Percent         Average  Percent         Average  Percent
Of Total   Trips  Time/   Of Total   Trips  Time/  Of Total   Trips  Time/  Of Total
  Time      Made  Trip      Time      Made  Trip     Time      Made  Trip     Time

DefVehicle:
   0.715    3832  161.15    0.278    2914   82.44    0.004      46   82.47    0.003
   0.732    3832  165.04    0.261    2860   78.73    0.004      40   91.62    0.003
   0.723    3835  162.91    0.268    2881   80.52    0.005      46   86.82    0.004
All DefVehicle:
   0.723   11499  163.03    0.269    8655   80.56    0.004     132   86.97    0.003

The Average Capacity Lost Due To Congestion is 14.3 Percent
Of Total Capacity Which is 0.4 Vehicles
```

Vehicle statistics in example model 12.2

Vehicles spend even less idle time in this system than before, reconfirming the need for additional vehicles in the system.

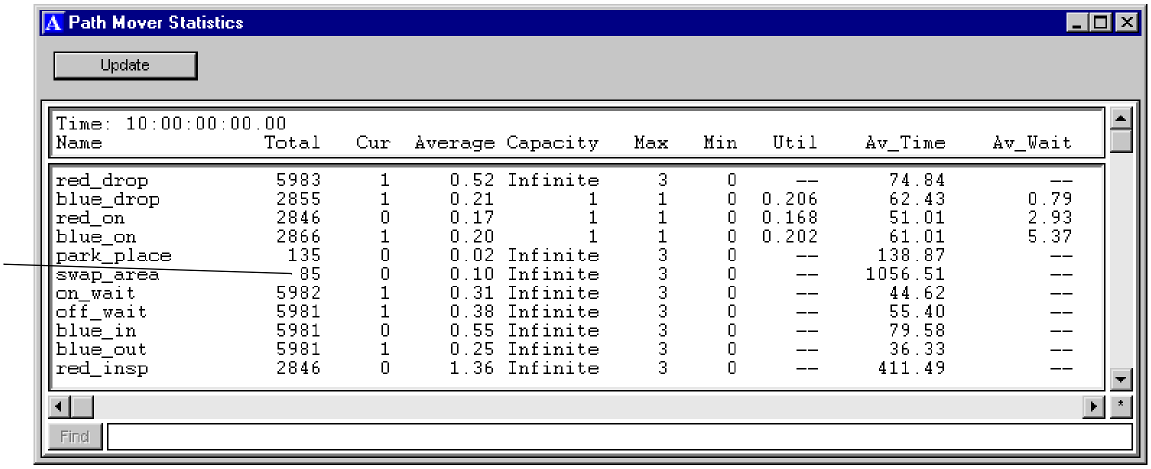
Notice that because vehicles spent time in the battery replacement lane during the simulation, the vehicle congestion in the system decreased (the average capacity lost due to congestion decreased from 15.3 percent to 14.3 percent).

Displaying control point statistics

Another way to verify that vehicles are traveling to the battery swapping area is by checking control point statistics.

To display control point statistics:

Step 1 From the Path Mover menu, ***select*** Control Points. The Path Mover Statistics window opens, as shown below:



```
Time:  10:00:00:00.00
Name                Total    Cur  Average Capacity   Max    Min    Util     Av_Time      Av_Wait

red_drop             5983     1     0.52 Infinite     3      0      --       74.84          --
blue_drop            2855     1     0.21        1     1      0   0.206       62.43        0.79
red_on               2846     0     0.17        1     1      0   0.168       51.01        2.93
blue_on              2866     1     0.20        1     1      0   0.202       61.01        5.37
park_place            135     0     0.02 Infinite     3      0      --      138.87          --
swap_area              85     0     0.10 Infinite     3      0      --     1056.51          --
on_wait              5982     1     0.31 Infinite     3      0      --       44.62          --
off_wait             5981     1     0.38 Infinite     3      0      --       55.40          --
blue_in              5981     0     0.55 Infinite     3      0      --       79.58          --
blue_out             5981     1     0.25 Infinite     3      0      --       36.33          --
red_insp             2846     0     1.36 Infinite     3      0      --      411.49          --
```

Vehicles changed batteries a total of 85 times

Control point statistics in example model 12.2

Control point statistics are defined as follows:

Name The name of the control point.

Total The total number of vehicles that claimed the control point.

Cur The number of vehicles that are currently claiming the control point.

Average The average number of vehicles that claimed the control point at the same time.

Capacity The total number of vehicles allowed to claim the control point at the same time.

Max The maximum number of vehicles that claimed the control point at the same time.

Min The minimum number of vehicles that claimed the control point at the same time.

Util The fraction of the control point's capacity that vehicles utilized.

Av_Time The average time a vehicle claimed the control point.

Av_Wait The average time a vehicle waited to claim the control point. This is the average of all vehicles, including vehicles that claimed the control point without waiting.

From the statistics for the point "swap_area," you can see that vehicles travel to the battery replacement area a total of 85 times during the simulation. The average time that vehicles spend at the control point is 17.61 minutes (1056.51 seconds). This number is slightly higher than the constant 15 minutes defined for the swapping time, because a vehicle sometimes needs to wait for a preceding vehicle to leave the point before it can have its battery replaced.

Tip You can tell that multiple vehicles were waiting for battery replacement at the same time by checking the value of the Max statistic for the control point "swap_area;" at least once during the simulation, all three vehicles were waiting at the control point.

Blocking vehicle movement

Often, it is necessary to limit vehicle movement in a system to prevent vehicle collisions or deadlocks. You can limit the movement of vehicles in a path mover system by placing blocks in the Work Area window. A **block** is graphical region that has a limited capacity; usually the capacity is set to one to prevent multiple vehicles from traveling through the block at the same time, however, the capacity can be set to any number. When a vehicle enters a block, it automatically claims one unit of the block's capacity; when the vehicle leaves a block, it automatically releases one unit of capacity. If a vehicle attempts to enter a block that has no available capacity, the vehicle is delayed at the edge of the block until capacity becomes available.

When using blocks as a **collision control** feature, the block's graphic is placed at intersections to prevent collisions when vehicles are merging from one path to another.

When using blocks as a **deadlock avoidance** feature, the block's graphic can be placed to encompass a two-directional guide path and prevent vehicles that are traveling in opposite directions from blocking each other on the same path.

Note

Control point capacity can also be used to prevent deadlocks on a two-directional path. For more information, see "Placing control points" on page 11.19 of the "Introduction to Path Mover Systems" chapter.

Blocks can also be used to prevent vehicle deadlocks on loops of path with a limited number of control points (this concept is discussed later in the chapter).

To demonstrate how blocks can be used to prevent vehicle collisions, you will place blocks in the example material handling system that you have created.

Example 12.3: Blocking vehicle collisions

When you run example model 12.2, warning messages appear indicating that there are vehicle collisions during the simulation. To prevent these collisions, you will place blocks at the intersections where vehicles merge onto a new path.

Placing blocks in example model 12.3

There are six intersections in the example model where vehicles merge from one path to another. To prevent vehicle collisions at these intersections, do the following:

Note

If you have created the example model in this chapter, you can continue using your current model. If you have not created the model but want to start at this point, you can import a copy of the *base* version of example model 12.3.

Step 1 ***Open*** the process system, then ***click*** Blocks on the Process System palette. The Blocks window opens.

Step 2 ***Click*** New to define a new block. The Define A Block window opens.

Step 3 ***Name*** the block "B_intersect."

Step 4 ***Change*** the Number of Blocks to "6" (changing the number of blocks creates an array of blocks).

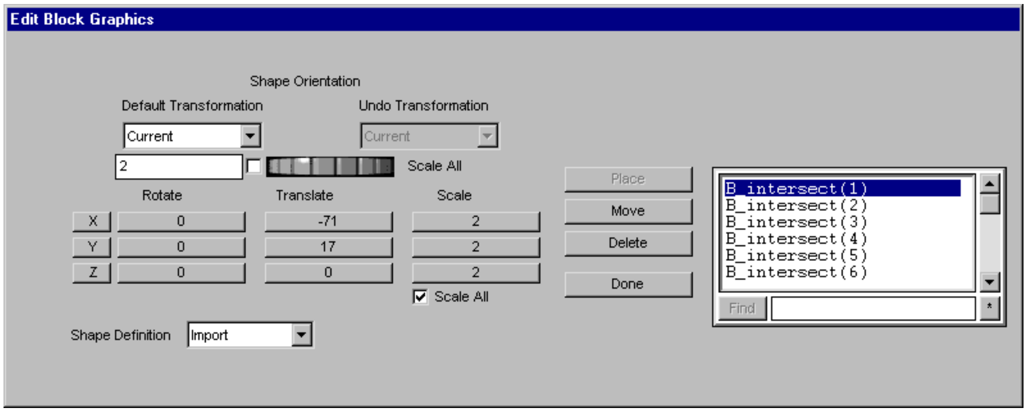
Step 5 Because the Default Capacity is already set to one, ***click*** OK to close the Define A Block window.

Step 6 Because B_intersect is already selected in the Blocks window, ***click*** Edit Graphic. The Edit Block Graphics window opens.

Step 7 ***Select*** B_intersect(1), then ***click*** Place to place the block's graphic.

Step 8 ***Click*** anywhere in the Work Area window to place the default block.

Step 9 ***Scale*** the block to two feet in every direction, as shown below:



Edit Block Graphics

Shape Orientation

Default Transformation: Current

Undo Transformation: Current

2 □ Scale All

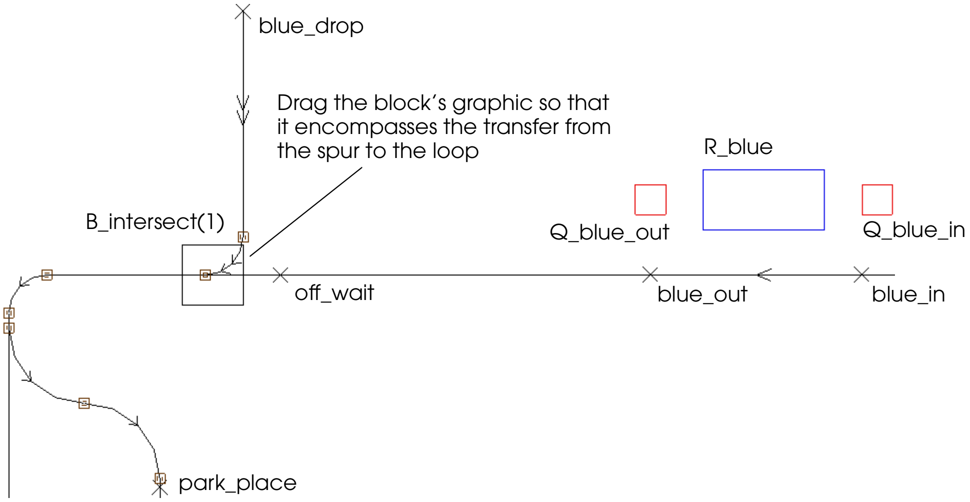
	Rotate	Translate	Scale
X	0	-71	2
Y	0	17	2
Z	0	0	2

☑ Scale All

Shape Definition: Import

Scaling the size of block B_intersect(1)

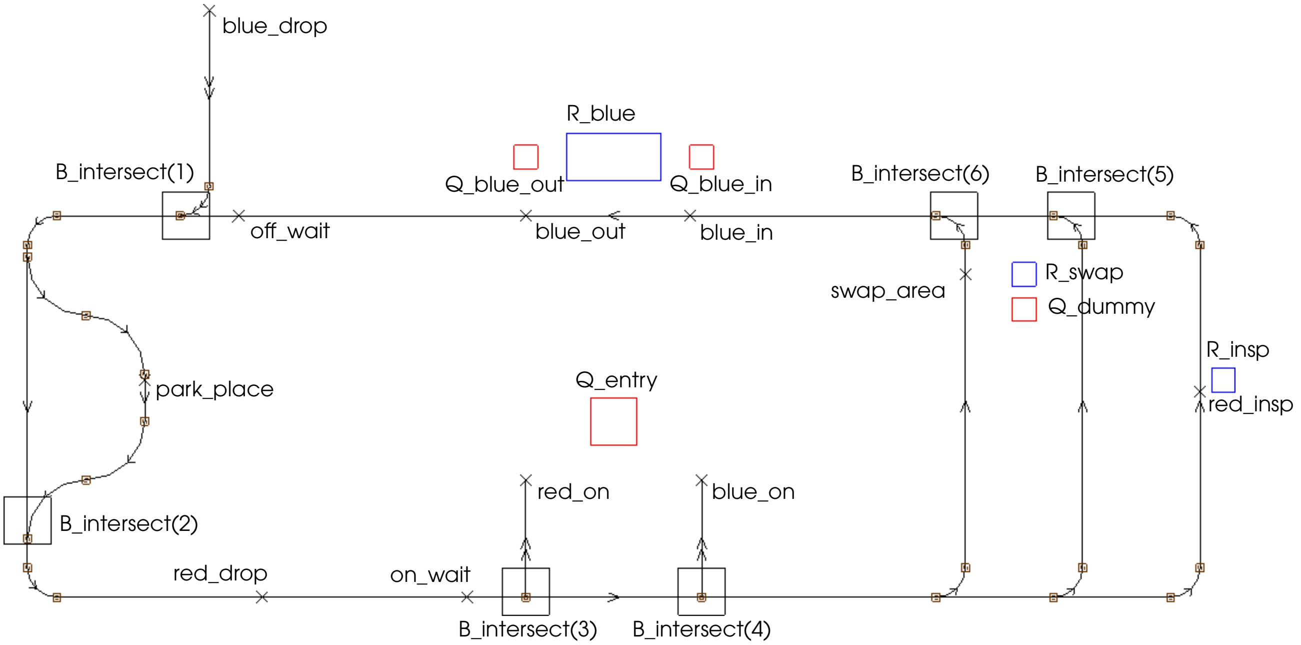
Step 10 ***Click*** Move and ***drag*** the block's graphic so that it encompasses the intersection where the spur path to point "blue_drop" merges into the loop, as shown below:


blue_drop
Drag the block's graphic so that it encompasses the transfer from the spur to the loop
R_blue
B_intersect(1)
Q_blue_out
Q_blue_in
off_wait
blue_out
blue_in
park_place

Placing a block in example model 12.3

Note You only need to place blocks at intersections where vehicles merge onto a path on which other vehicles may be traveling, as shown above. You do *not* need to place blocks at intersections where vehicles diverge (for example, the diverging path that leads to the point "park_place") because vehicle collisions are not possible at those intersections.

Step 11 ***Place*** the remaining blocks at intersections where vehicles merge, as shown in the illustration below (all block graphics are scaled to two feet in every direction):



Placing blocks in example model 12.3

You have now placed blocks at all of the intersections where vehicles merge during the simulation.

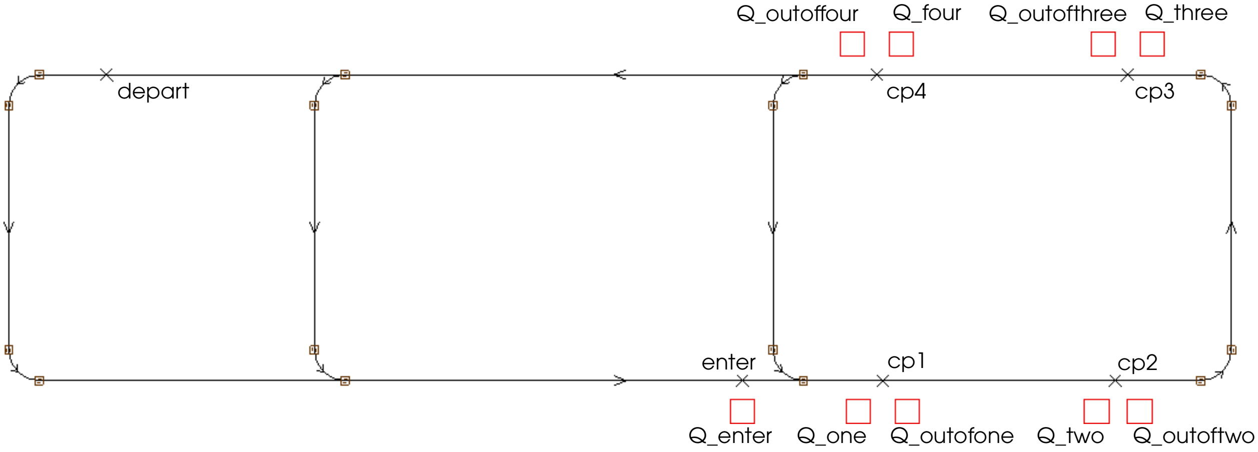
Step 12 ***Click*** Done to close the Edit Block graphics window.

Step 13 ***Export*** and ***run*** the model. Notice that there are no collision warnings during the simulation. The example model is now complete.

The next example model is a different path mover system that demonstrates the use of blocks as a deadlock avoidance feature.

Example 12.4: Blocking vehicle deadlocks

Consider the path mover layout shown below:



Layout of example model 12.4

Loads arrive and move into the control point “enter” with an interarrival time that is exponentially distributed with a mean of 5 minutes. After getting on a vehicle, loads move to point “cp1,” where they get off the vehicle and move into the infinite-capacity queue, “Q_one.” Loads delay in the queue for a time that is exponentially distributed with a mean of 5 minutes and then move into the infinite-capacity queue “Q_outofone,” where they wait to get back on a vehicle at the point “cp1.”

After getting on a vehicle, loads travel to point “cp2,” where they again get off the vehicle and move into the infinite-capacity queue “Q_two.” After delaying for a time that is exponentially distributed with a mean of 5 minutes, they move into the infinite capacity queue “Q_outoftwo” and wait to get back on a vehicle at the point “cp2.”

Loads continue this procedure, traveling to points “cp3” (where they get off the vehicle) and “cp4” (where they again get off the vehicle). After being picked up at point “cp4,” loads travel to the point “depart,” where they leave the system.

The capacity of each of the numbered control points is set to one. The limited capacity allows only one vehicle to claim and travel to each of these points at a time.

A named list is defined to start vehicles at the control point “depart.” A park list at point “enter” sends empty vehicles to point “cp1.” The remaining park lists are defined to keep empty vehicles moving in the system. For example, a park list is defined at point “cp1” to send empty vehicles to “cp2.” A park list at point “cp2” sends empty vehicles to “cp3,” and so on. The park list at “cp4” sends loads back to “cp1.”

The work lists in the system are defined as shown in the table below:

If empty at point...	Then look for work at point(s)...
cp1	cp2 cp3 cp4 enter
cp2	cp3 cp4 enter
cp3	cp4 enter
cp4	enter
depart	enter
enter	cp1

Shortly after 20 minutes of simulation, the vehicles in this system become deadlocked.

To view the deadlock:

Step 1 ***Import*** a copy of the *base* version of example model 12.4.

Step 2 ***Run*** the model.

Tip ☞ To speed up the simulation, increase the display step to update the display every 2 seconds.

Shortly after 20 minutes of simulation time, the fourth vehicle enters the loop and the vehicles become deadlocked. The deadlock occurs because each vehicle is trying to travel to a control point that is currently occupied by another vehicle.

To prevent the deadlock, you need to place a block to ensure that only three vehicles enter the loop containing the numbered control points at the same time.

Placing blocks in example model 12.4

To place a block that allows only three vehicles in the loop in example model 12.4, do the following:

Step 1 ***Edit*** the model.

Step 2 On the Process System palette, ***click*** Blocks. The Blocks window opens.

Step 3 Click New to define a new block. The Define A Block window opens.

Step 4 ***Name*** the block “B_deadlock.”

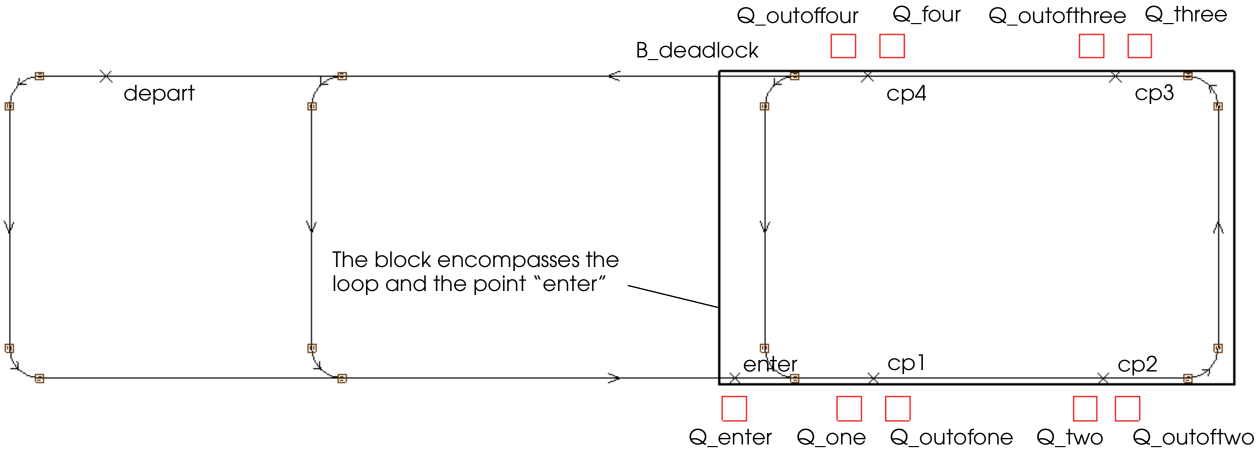
Step 5 ***Change*** the Default Capacity to “3” and ***click*** OK to close the Define A Block window.

Step 6 Because B_deadlock is already selected in the Blocks window, ***click*** Edit Graphic. The Edit Block Graphics window opens.

Step 7 ***Click*** Place to place the block’s graphic, then ***click*** anywhere in the Work Area window to place the default block.

Step 8 ***Scale*** the block to 21.25 feet in the X direction and 13 feet in the Y direction.

Step 9 ***Click*** Move, then ***drag*** the block’s graphic so that it encompasses both the loop containing the numbered control points and the point “enter,” as shown below:



Placing the block in example model 12.4

Important

The block must encompass the control point “enter.” Otherwise, a vehicle outside the loop (at point “enter”) may claim the control point “cp1” before a vehicle inside the loop at point “cp4.” As a result, the vehicle outside the loop would remain at the block’s boundary, unable to enter the block, and the vehicle at point “cp4” would be unable to travel to the point “cp1” because it was already claimed — another deadlock situation.

Step 10 ***Click*** Done to close the Edit Block Graphics window.

Step 11 ***Export*** and ***run*** the model.

Tip

Set an alarm to pause the simulation after 20 minutes; you can then run the simulation without graphics to quickly advance to the time when the deadlock occurred. Remember to turn graphics back on before continuing the simulation.

The block prevents the deadlock from occurring.

Summary

In this chapter, you learned how to write the model logic that controls load movement in a path mover system. You also learned how to control vehicles by creating scheduling lists at control point locations. The chapter also discussed preventing vehicle collisions and deadlocks by placing blocks in the system.

Exercises

Exercise 12.1

Copy the *final* version of example model 12.3 to a new directory. Additional information about this system is provided below.

The facility's operations department wants to lease additional vehicles to operate in the system while management wants to operate fewer vehicles. An AGV has a lease cost of $300 per day and daily maintenance is 50 percent of the lease cost. Each work in process (WIP) load in the system, on the average, translates into a daily carrying cost of $10 (the cost is the same for both red and blue loads).

Conduct an analysis that varies the number of vehicles in the system; determine the number of vehicles that results in the lowest total cost.

Tip ☞

Use AutoStat to conduct the analysis with five replications and use the Setup wizard to define the following model properties:

- Model is random
- Check for infinite loops and stop runs that are longer than two minutes
- Do not define a warmup
- Define a snap length of three days

Exercise 12.2

Note ✎

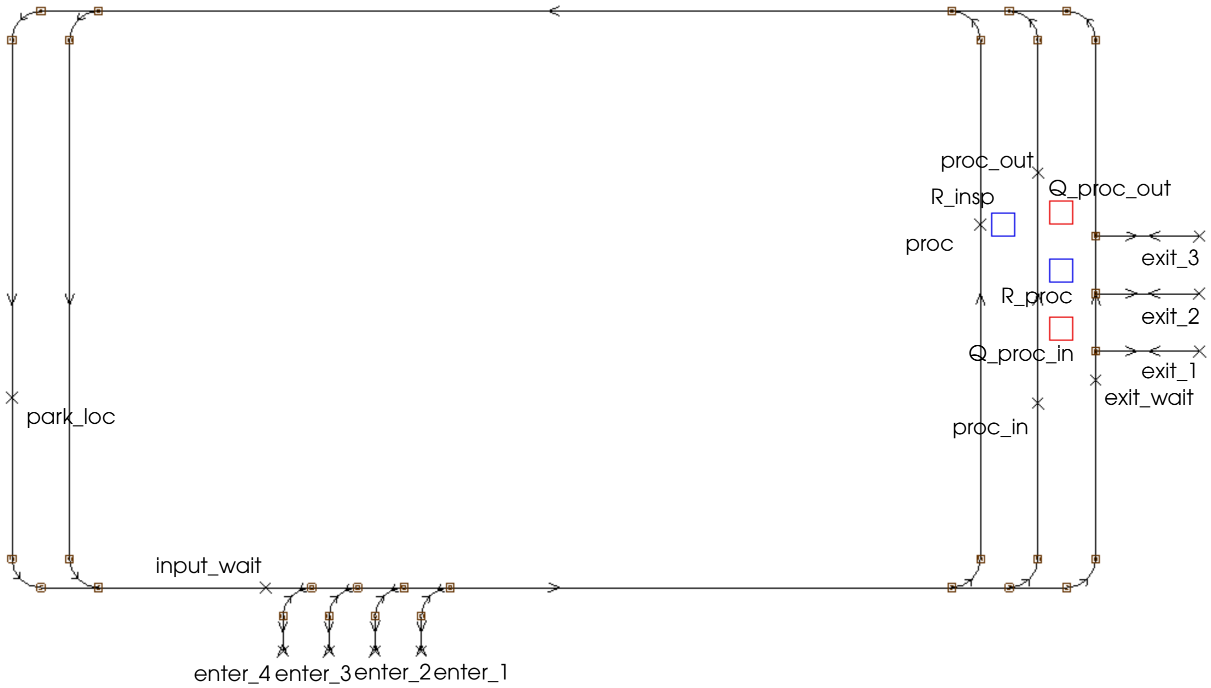
The solution for this assignment is required to complete exercise 14.1 (see "Exercise 14.1" on page 14.27 of the "Additional Features" chapter); be sure to save a copy of your model.

Copy your solution model for exercise 11.1 to a new directory. Additional information about this system is provided below and on the following page.

Four different types of loads are processed in this system; each load type is assigned a unique color. The interarrival rate, pickup location, and exit location for each type of load is defined in the following table:

Load color	Interarrival rate	Pickup location	Exit location
red	exponential 9 minutes	enter_1	exit_1
blue	exponential 9 minutes	enter_2	exit_2
yellow	exponential 12 minutes	enter_3	exit_3
green	exponential 12 minutes	enter_4	exit_3

After being picked up by a vehicle, each load travels to control point "proc_in" and is unloaded into an infinite-capacity queue, "Q_proc_in" for processing (refer to the illustration below).



Exercise 12.2 path mover layout

There is a single-capacity resource, "R_proc," operating between "proc_in" and "proc_out" that processes each load for a time that is exponentially distributed with a mean of 100 seconds. After processing, loads move into another infinite-capacity queue, "Q_proc_out," and wait to be picked up by a vehicle at control point "proc_out."

After being picked up by a vehicle, loads travel to control point "proc," where they are inspected (while onboard the vehicle) by an inspector. The inspector inspects each load for a constant 100 seconds. After inspection, loads travel to their assigned exit location, where they are removed from the system.

Vehicles start the simulation at the point "park_loc."

Model this system and place blocks at each of the intersections where vehicle collisions are possible. Conduct an analysis that varies the number of vehicles in the system; determine the number of vehicles that results in the lowest average time in system for loads.

Tip

Use AutoStat to conduct the analysis with five replications and use the Setup wizard to define the following model properties:

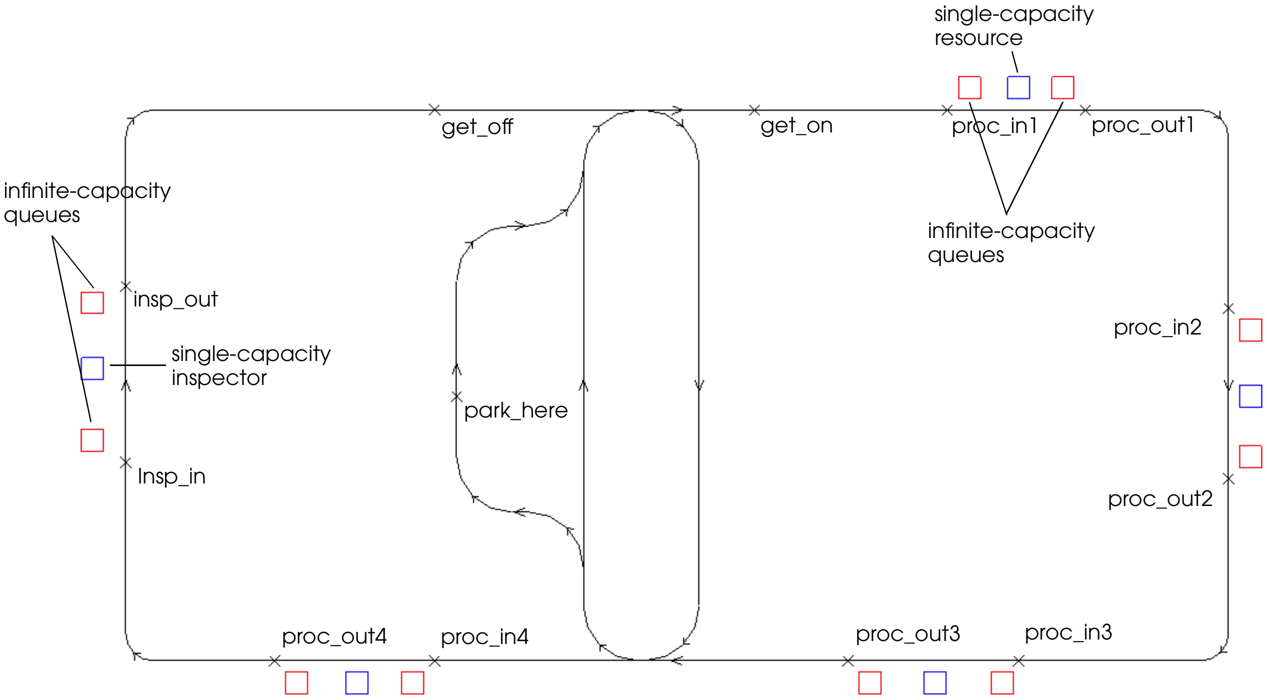
- Model is random
- Check for infinite loops and stop runs that are longer than two minutes
- Do not define a warmup
- Define a snap length of three days

Exercise 12.3

Note The solution for this assignment is required to complete exercise 14.2 (see "Exercise 14.2" on page 14.27 of the "Additional Features" chapter); be sure to save a copy of your model.

Copy your solution model for exercise 11.2 to a new directory. Additional information about this system is provided below.

Loads are generated in the system at a time that is exponentially distributed with a mean of 4 minutes. A single-capacity resource for processing loads is located between each "proc_in" and "proc_out" control point (refer to the illustration below).



There is an infinite-capacity queue before and after each resource. Similarly, there is a single-capacity inspector between "insp_in" and "insp_out" with an infinite-capacity queue before and after the inspector.

Loads are picked up by a vehicle at control point "get_on." After being picked up, each load travels to the first processing location. Loads are unloaded into the first queue at "proc_in1" where they wait to use the processing resource. The resource processes each load for a time that is exponentially distributed with a mean of 3 minutes. When processing is complete, loads move into the second queue and wait to be picked up by a vehicle at control point "proc_out1." Loads then move to the next processing location where the process is repeated (all resources process loads for the same amount of time).

After completing processing at the fourth processing location, loads travel to control point "insp_in" where they are unloaded into the inspector's first queue. Loads are inspected for a time that is exponentially distributed with a mean of 3 minutes. When inspection is complete, loads move into the inspector's second queue and wait to be picked up by a vehicle at control point "insp_out." Loads then travel to control point "get_off" where they are removed from the system.

Vehicles start the simulation at point "park_here."

Model this system, and place blocks at each of the intersections where vehicle collisions are possible.

Complete the following:

a) Conduct an analysis to vary the number of vehicles in the system from 1 through 6 and determine the number of vehicles that results in the lowest average time in system for loads.
b) Is there a statistically significant difference between the vehicle configuration with the lowest average time in system and all other vehicle configurations?
c) Based on your findings in (a) and (b), how many vehicles would you recommend leasing for operation in the system?

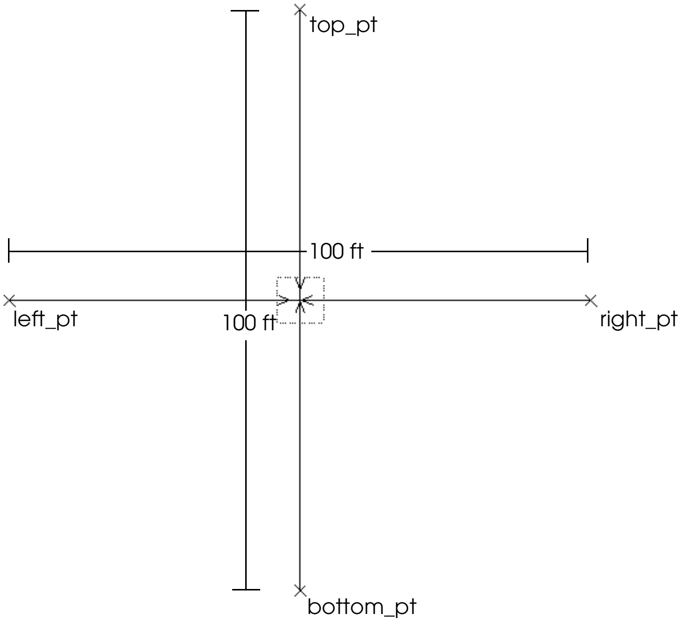
Tip ☞

Use AutoStat to conduct the analysis with five replications, and use the Setup wizard to define the following model properties:

- Model is random
- Check for infinite loops and stop runs that are longer than two minutes
- Do not define a warmup
- Define a snap length of three days

Exercise 12.4

Consider the path mover system shown below:



The path mover system consists of two intersecting two-directional paths. Loads independently arrive at control point “top_pt” and control point “left_pt” at a time that is exponentially distributed with a mean of three minutes.

The system operates two vehicles. The first vehicle starts at control point “top_pt” and the second vehicle starts at control point “left_pt.” The vehicles’ size is defined as:

X = 4
Y = 2
Z = 1

A block is placed at the center of the intersection of the paths. The block’s size is defined as:

X = 2
Y = 2
Z = 2

Create this model and run the simulation for 24 hours. Determine the number of collisions that were prevented during the simulation.

Note

You can discover the number of prevented collisions by finding out how many times vehicles were required to wait to claim the block at the intersection. Block wait statistics can be found by selecting Single Block from the Blocks menu in the Simulation window.

Chapter 13

Indefinite Delays

Chapter 13

Indefinite Delays

You have already learned several ways in which loads can be delayed during a simulation, for example, a load delays when executing the `wait` action in an arriving procedure or when waiting to be picked up by a vehicle. This chapter focuses on delaying loads for an indefinite amount of time by using an order list.

Causing loads to wait on an order list is a flexible way to delay loads during a simulation; the duration of the delay depends on the processing of other loads in the simulation. Indefinite delays are useful when individual loads must be grouped at some point during their processing. For example, consider a system in which loads arrive at a queue to be packaged in boxes. Each load must wait in the queue until the required number of loads have arrived, at which point the waiting loads can be packaged in a box. In the AutoMod software, you can delay the loads, for as long as necessary, by using an order list.

Delay types

This textbook discusses three different types of load delays in the AutoMod software:

Definite During a definite delay, a load waits for a known amount of time. Definite delays are created by using the `wait` action to delay a load in an arriving procedure; the duration of the delay can be either constant or random. If the delay is random, a distribution that determines the duration of the delay is defined in the action. The following actions are examples of definite delays:

```
wait for 10 min
wait for e 2 hr
wait for V_time
```

State-based During a state-based delay, a load waits for an entity with limited capacity to become available. For example, the load may need to wait for a path mover vehicle to arrive or a resource or queue to become available. The duration of the delay is not known when the action is executed, but is determined during the simulation. The following actions are examples of state-based delays:

```
get R_worker
use R_worker for 10 min
move into pm:get_off
move into Q_mach(1)
```

Indefinite During an indefinite delay, a load waits until it is ordered to continue by another load in the simulation. Like state-based delays, the duration of the delay is not known when the action is executed, but is determined by the processing of other loads during the simulation. Because loads execute the model logic, almost any condition that a load might wait for can be changed by another load's processing during a simulation. This type of delay is discussed in greater detail throughout this chapter.

Creating indefinite delays

Creating an indefinite delay requires two loads.

The first load executes a `wait to be ordered` action to get on an order list. An **order list** is a user-defined list of loads that are delayed in the simulation. An order list does not have a capacity; any number of loads can wait on a single order list. Once on the list, a load is delayed indefinitely.

Later in the simulation, a second load executes an `order` action to order the waiting load off the list. The second load can order any number of waiting loads off an order list at the same time. The `order` action determines the next action that is performed by the waiting loads.

These concepts are discussed in greater detail in the following sections.

Causing loads to wait on an order list

To get on an order list, a load executes a `wait to be ordered` action in an arriving procedure. In the action, the load specifies the order list on which it wants to wait. By default, loads are appended to the list in the order in which they execute the `wait to be ordered` action (the load that has been waiting the longest is at the top of the list).

For example, if a load has arrived at a queue to be packaged in a box, but additional loads are required to fill the box, the load can wait for the additional loads by executing the following action to get on the order list OL_delay:

```
wait to be ordered on OL_delay
```

After executing the `wait to be ordered` action, the load is delayed in its current process and waits to execute the next action in its current arriving procedure. The territory of the load does not change; loads on an order list remain in their current queue or movement system location.

Ordering loads off an order list

A load can order any number of waiting loads off an order list by executing the `order` action in an arriving procedure. In the action, the ordering load specifies the number of waiting loads to order. Loads get off the list in order from top to bottom (the top load in the list is the first to get off). This textbook discusses ordering the following quantities of loads off a list:

- The first load
- Some number of loads
- All loads

When loads are ordered off a list, they can be ordered to do one of the following:

- Continue (loads execute the next action in their current arriving procedure)
- Go to another process (loads execute the first action in the new process' arriving procedure)
- Wait on another order list (loads continue to delay on another order list)
- Die (loads are removed from the simulation)

For example, the following syntax orders the first load on the order list OL_delay to continue executing in its current process:

```
order 1 load from OL_delay to continue
```

The load at the top of the order list OL_delay gets off the list and resumes executing the next action in its current arriving procedure.

To order five loads from OL_delay to the process P_inspect, use the following syntax:

```
order 5 loads from OL_delay to P_inspect
```

The first five loads get off the order list and leave their current process to enter the process P_inspect. The ordered loads immediately begin executing the arriving procedure defined for the P_inspect process.

To order all loads to move from one order list to another, use the following syntax:

```
order all loads from OL_delay to OL_wait
```

All the loads that are currently on the order list OL_delay are taken off the list and added to the list OL_wait. By default, loads are sorted on the new list in the same order; the load that has been waiting the longest is at the top of the list. The loads' current process and territory do not change.

To order three loads from OL_delay to die, use the following syntax:

```
order 3 loads from OL_delay to die
```

The first three loads on the list are immediately removed from the simulation; the loads do not complete the execution of their current arriving procedure.

Note

When ordering loads off an order list, you can also order only those loads that satisfy a specific condition. For example, you can order only loads of type L_blue off the list. Using conditions when ordering loads is not discussed in this textbook. For more information, see the `order` action in the AutoMod Syntax Help.

Backordering loads

An order is **filled** when an order list contains at least as many loads as are ordered off the list. For example, if you order five loads off an order list that contains six loads, the order is filled: five loads get off the list, leaving one load still waiting on the list.

Note

When you use the `all` syntax in the `order` action, the order is always filled. Regardless of the number of loads on the list (whether there are 20 loads or 0 loads), all loads get off the list.

An order list may not always contain a sufficient number of loads to completely fill an order; in such cases, all of the loads are ordered off the list to partially fill the order. For example, if you order five loads off a list that contains only two loads, the two waiting loads get off the list.

You can use the `in case order not filled` syntax to create a backorder when an order is not completely filled. When loads are back ordered, any loads that are ordered to wait on the list later in the simulation are automatically ordered off the list until the backorder is filled. For example, consider the syntax below:

```
order 5 loads from OL_delay to continue
in case order not filled backorder on OL_delay
```

If the order list OL_delay only contains three loads when the `order` action is executed, all three loads are ordered to continue and a backorder of two loads is created. The next two loads that are ordered to wait on the order list OL_delay are immediately ordered to continue (to fill the backorder) and do not delay in the simulation.

Note

You can execute any actions after the `in case order not filled` syntax. For example, the syntax below prints warnings to the Message window when an order is not filled, but does not create a backorder.

```
order 5 loads from OL_delay to continue
in case order not filled
   begin
      print “Warning! Order for 5 loads from OL_delay not filled at time ”
         ac to message
      print “No loads back ordered.” to message
   end
```

Using the attribute “current loads”

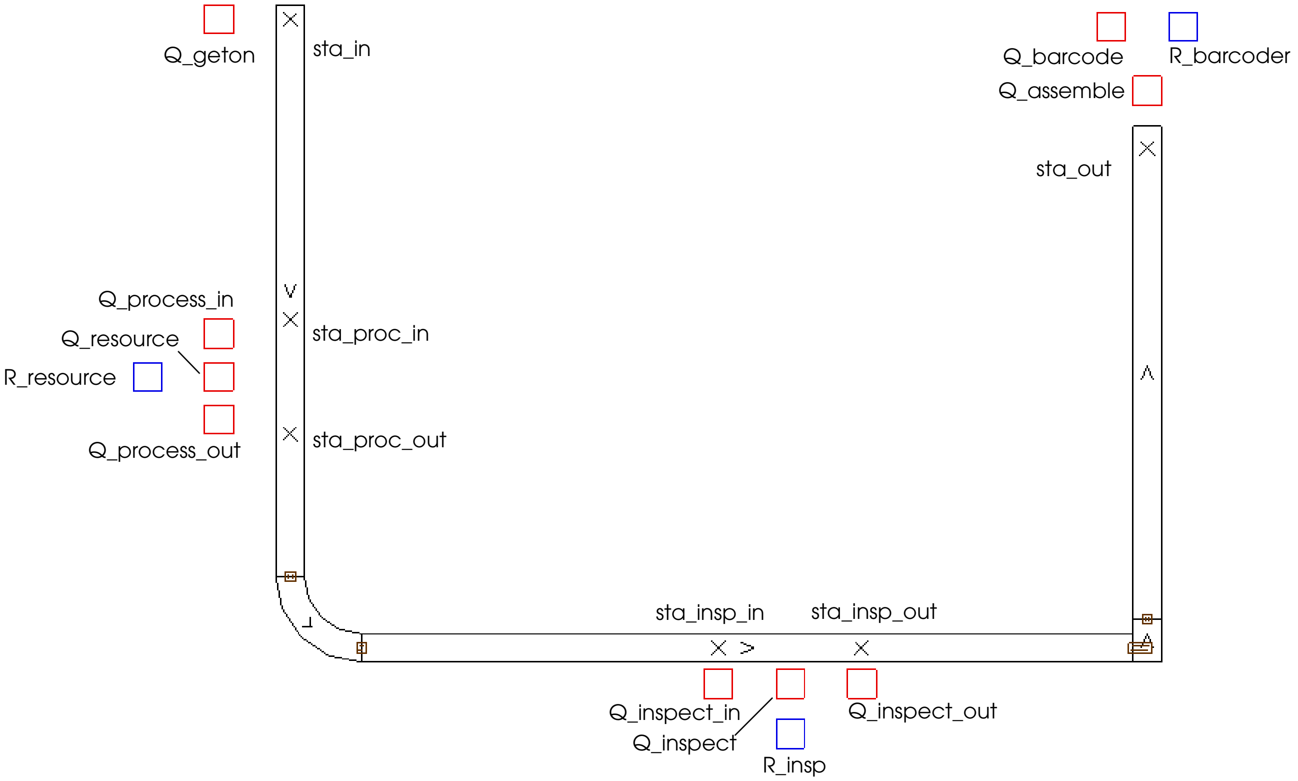
Like other AutoMod entities, order lists have standard attributes with values that are automatically set by the software. The **current loads** attribute is an attribute of several different AutoMod entities, including resources, processes, and order lists. The `current loads` attribute indicates the number of loads that are currently claiming an entity. For example, the attribute can refer to the number of loads that are currently using a resource, executing a process, or waiting on an order list.

The `current loads` attribute represents a unique integer value for each entity; for example, when a load executes a `wait to be ordered` action to wait on a list, the order list’s `current loads` attribute increments by one. When a load is ordered off a list, the order list’s `current loads` attribute decrements by one. You can use the `current loads` attribute to refer to the number of loads on an order list at any given moment during a simulation.

When using the `current loads` attribute in logic, it must be preceded by the name of the order list, resource, or process for which you want the number of current loads. For example, the syntax `OL_delay current loads` represents the number of loads that are currently waiting on the order list OL_delay.

Example 13.1: Modeling an assembly and packaging operation

To better understand how order lists are used in a simulation, consider the conveyor layout shown below:



Layout of example model 13.1

The model is the same as the example conveyor system you created in chapter 6, but with an added assembly and packaging operation at the end of the conveyor. The processing and inspection operations, as well as the load arrival rates, remain the same as were defined in the original model (see “Example 6.1: Drawing a conveyor system” on page 6.9 of the “Introduction to Conveyors” chapter for more information).

In this expanded model, loads leave the conveyor at station “sta_out” and move into the infinite-capacity queue Q_assemble. As soon as the queue contains four loads, the loads are assembled and packaged in a box. An order list is used to group the four loads for packaging. The time required to package the loads is normally distributed with a mean of 8 minutes and a standard deviation of 30 seconds.

After packaging, each box moves into the infinite-capacity queue Q_barcode where the box uses an automatic barcoder for 2 seconds. The completed box then leaves the system.

Defining the order list in example model 13.1

The logic that is used to simulate example 13.1 has already been defined in the base version of example model 13.1. However, the logic is currently commented because the order list that groups four loads in the queue Q_assemble has not yet been defined. You need to import the model and define the order list.

To define the order list:

Step 1 ***Import*** a copy of the *base* version of example model 13.1.

Step 2 ***Edit*** the model logic and ***delete*** the comment markers `/*` and `*/` at the beginning and end of the source file.

Take a moment to review the new assembly and barcode procedures, shown below:

```
begin P_assemble arriving
   move into Q_assemble                        /*assemble and box here*/
   if OL_assemble current loads = 3 then
      begin
         order 3 loads from OL_assemble to die
            /*The fourth loads kills the first three loads*/
         set load type to L_box
            /*Fourth load becomes a box*/
         wait for n 8, .5 min                  /*assemble and box time*/
         send to P_barcode
      end
   else wait to be ordered on OL_assemble
end

begin P_barcode arriving
   move into Q_barcode                         /*barcode here*/
   use R_barcoder for 2 sec                    /*use the barcode machine*/
   send to die
end
```

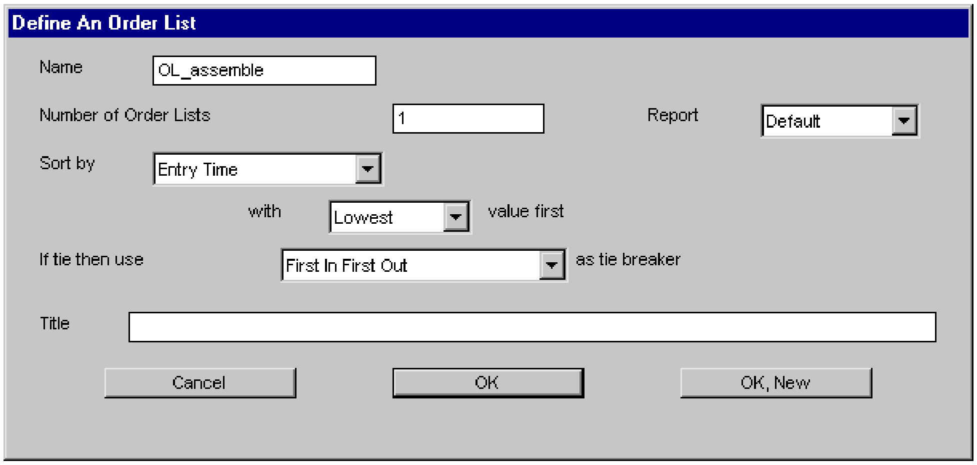
After getting off the conveyor, each load moves into the infinite-capacity queue Q_assemble. Each load then checks to see how many loads are currently waiting on the order list OL_assemble. If there are not yet three loads on the list, the current load is ordered to wait on the list. If there are already three loads waiting on the list, the current (fourth) load orders the three loads to die and then changes its load type to L_box, representing a box in the system. The load then delays for the assembly and packaging time and is then sent to the P_barcode arriving procedure.

In the P_barcode arriving procedure, the box moves into the infinite-capacity queue Q_barcode, then uses an automatic barcoder for 2 seconds before leaving the system.

Step 3 ***Save*** and ***quit*** the source file. The Error Correction window appears, indicating that OL_assemble is undefined.

Step 4 ***Select*** Define. In the Define as drop-down list, ***select*** Orderlist.

Step 5 *Click* Define as. The Define An Order List window appears, as shown below:



Name OL_assemble

Number of Order Lists 1

Report Default

Sort by Entry Time

with Lowest value first

If tie then use First In First Out as tie breaker

Title

The Define An Order List window

Options in the Define An Order List window are defined as follows:

Name The name of the order list.

Number of Order Lists The number of order lists you are defining; typing a number greater than one creates an array of order lists.

Sort by The criterion by which the loads on the order list are sorted.

Options in the Sort by list are:

Entry Time – Loads are sorted in the order in which the loads got on the order list. By default, the loads are sorted with the lowest value first, which causes the load that has been waiting the longest to be at the top of the order list. This is the default sortation criterion.

Priority – Loads are sorted by an integer priority value that is unique to each load. By default, the loads are sorted with the lowest value first, which causes loads with lower integer values to be the first loads ordered off the list. This sortation criterion is discussed in more detail later in this chapter.

Load attribute – Loads are sorted according to the value of a numeric load attribute. By default, the loads are sorted with the lowest value first, which causes loads with the lower load attribute values to be the first loads ordered off the list.

Tie breaker If two loads have the same value in the sortation criterion, the tie-breaker criterion decides which load is listed first. The tie-breaker uses the load's entry time to determine the sortation order; consequently, the tie-breaker criterion is useful when the sortation criterion is set to either priority or load attribute.

Options in the Tie breaker list are:

First In First Out – The first load that was added to the list is the first load to get off the list.

Last In First Out – The last load that was added to the list is the first load to get off the list.

Step 6 You want to use the default options for the order list, so ***click*** OK to close the Define An Order List window.

You have now defined the order list OL_assemble.

Step 7 ***Export*** and ***run*** the model.

Step 8 Once you are familiar with how loads are processed in the simulation, ***quit*** the model.

Now you are ready to learn another use for order lists: creating slugs on a conveyor.

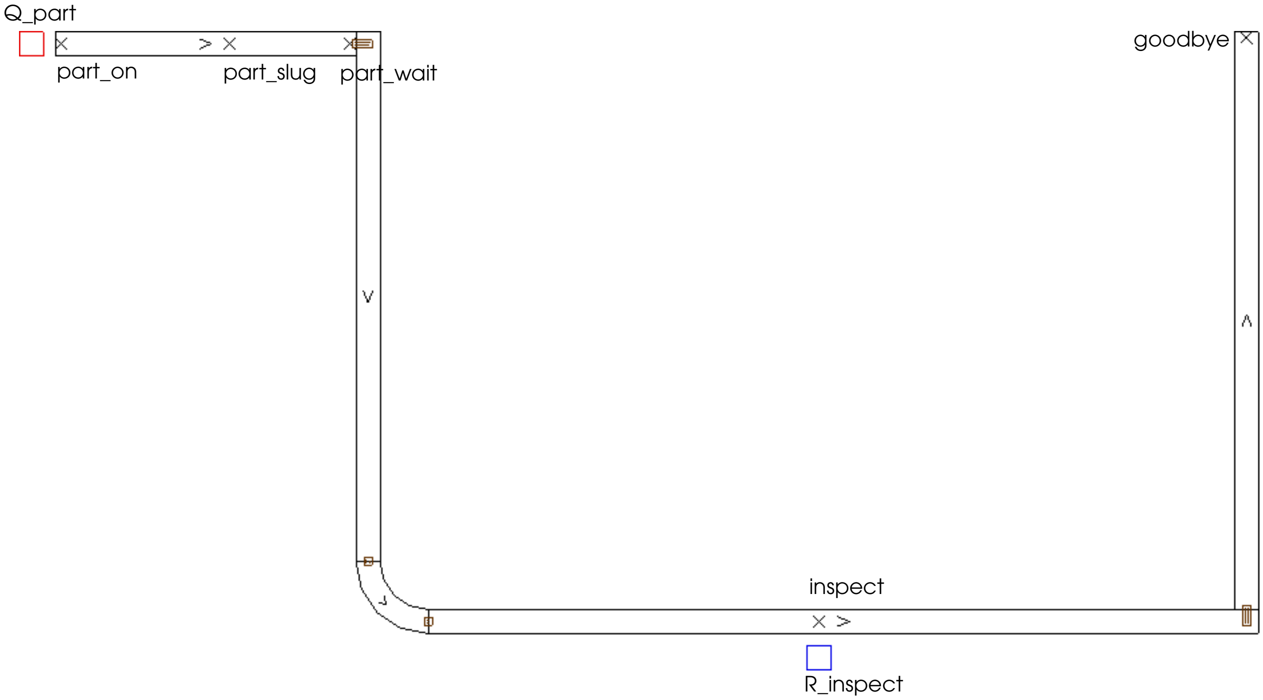
Modeling slugging conveyors using order lists

In chapter 9, “Modeling Complex Conveyor Systems,” you learned how to form slugs by taking down and bringing up the motor of a small section of conveyor before a side transfer. In this chapter, you will learn an easier method of forming slugs using order lists. When you use order lists, you can form slugs at any station in a conveyor system (the station does not need to be located before a side transfer and you do not need to draw any extra sections of conveyor, as was required in chapter 9).

This chapter uses two example models to demonstrate how to form slugs using order lists. The first model creates slugs on a single entrance lane in the conveyor system. The second model creates slugs on two separate entrance lanes and introduces a new method of keeping the slugs separate as they travel through the system.

Example 13.2: Creating slugs on one entrance lane

Consider the conveyor layout shown below:



Layout of example model 13.2

Loads arrive in the infinite-capacity queue Q_part at a rate that is exponentially distributed with a mean of 20 seconds. After arrival, loads get on the conveyor at station “part_on.” Loads travel down the entrance lane to station “part_wait,” where they accumulate into slugs consisting of 10 loads each.

As soon as 10 loads have accumulated, the slug continues to station “inspect,” where the loads are inspected by an inspector for a time that is exponentially distributed with a mean of 15 seconds.

After inspection, the loads travel to the end of the conveyor and leave the system.

To become familiar with the formation of slugs in the simulation, do the following:

Step 1 ***Import*** and ***run*** a copy of example model 13.2

Step 2 When you are ready to continue, ***close*** the model.

Modeling example 13.2 using order lists

The model logic for example model 13.2 is shown below:

```
begin P_part arriving
   move into Q_part
   move into conv:part_on
   travel to conv:part_slug
   inc V_slug by 1
   if V_slug = 10 then               /* release the slug */
      begin
         order V_slug loads from OL_part to continue
         in case order not filled backorder on OL_part
         set V_slug to 0
      end
   travel to conv:part_wait
   wait to be ordered on OL_part     /* form the slug */
   travel to conv:inspect
   use R_inspect for e 15 sec
   travel to conv:goodbye
end
```

To understand the logic, consider the actions that are executed by the first ten loads in the simulation.

The first load moves into the queue Q_part, gets on the conveyor at station “part_on,” and travels to the station “part_slug.” At the station, the load increments the integer variable V_slug, which counts the number of loads in the currently forming slug. The load then travels to the station “part_wait,” where the load is ordered to wait on the order list “OL_part.”

The next eight loads in the simulation perform the same actions. However, because the first load is already stopped at the station “part_wait” (while waiting on the order list), the trailing loads accumulate behind the first load and do not arrive at the station. Consequently, the loads do not complete the `travel to conv:part_wait` action and are not yet ordered to wait on the order list.

When the tenth load arrives at the station “part_slug,” it releases the current slug by ordering 10 loads off the order list. Because only the first load in the slug is currently on the list, the remaining 9 loads are back ordered. After releasing the slug, the tenth load resets the value of the V_slug variable to begin counting the next slug, then travels to the station “part_wait.”

After the first load in the slug leaves the station “part_wait” to travel to the station “inspect,” each of the trailing loads arrive at the station, are ordered to wait on the list, and are then immediately ordered to continue (to fill the backorder). Each load in the slug travels past the station without stopping.

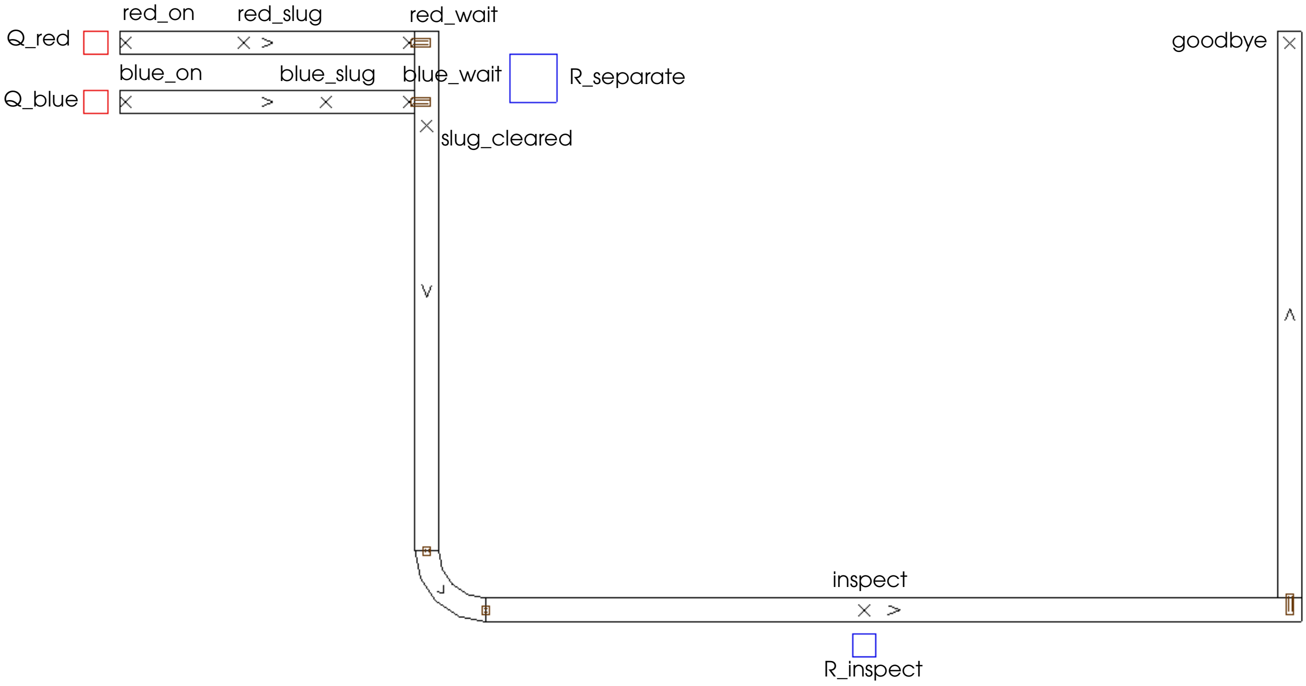
When the eleventh load arrives at the station, the backorder has been filled, so the load stops and waits on the order list, which causes the next slug to begin accumulating.

Important

When using order lists to form slugs on a conveyor, you must place stations correctly in order to prevent a gap between loads in the slug. The last load in the slug must be as close to the preceding load as possible when the slug is released. In example model 12.3, the station “part_slug” is placed 18 feet from station “part_wait,” to allow enough space on the conveyor for exactly 9 loads to accumulate (the loads are scaled to 2 feet on the X axis); when the tenth load arrives at the station to release the slug, it is already in the correct position to begin traveling immediately behind the ninth load.

Example 13.3: Creating slugs on two entrance lanes

Now consider the conveyor layout shown below:



Layout of example model 13.3

In this example, two different load types arrive at the infinite-capacity queues before the two entrance lanes in the system. Loads of type L_red arrive at the queue Q_red, and loads of type L_blue arrive at the queue Q_blue. The size of red loads is defined as:

X = 2
Y = 3
Z = 1

The size of blue loads is defined as:

X = 3
Y = 2
Z = 1

Both load types arrive at a rate that is exponentially distributed with a mean of 30 seconds.

After arriving in a queue, loads of each type get on the conveyor and travel to the end of their respective entrance lanes, where the loads accumulate into slugs. Red loads form slugs consisting of 10 loads each and blue loads form slugs consisting of 8 loads each.

The slugs are prevented from mixing when transferring from the entrance lanes to the vertical section of conveyor. If slugs on both entrance lanes are ready to release at the same time, one of the slugs is delayed until the other slug has traveled passed the entrance lanes, at which time the delayed slug is released.

The completed slugs travel to the station "inspect," where they are inspected by an inspector for a time that is exponentially distributed with a mean of 10 seconds. The inspected loads then travel to the station "goodbye," where they leave the system.

To become familiar with the formation of slugs in the simulation, do the following:

Step 1 ***Import*** and ***run*** a copy of example model 13.3.

Step 2 When you are ready to continue, ***close*** the model.

Modeling example 13.3 using order lists

The model logic for example model 13.3 is shown below:

```
begin P_red arriving
   move into Q_red
   move into conv:red_on
   travel to conv:red_slug
   inc V_red by 1
   if V_red = 10 then                    /* release the slug */
      begin
         get R_separate
         order 10 loads from OL_red to continue
         in case order not filled backorder on OL_red
         set V_red to 0
         set A_last to true
      end
   travel to conv:red_wait
   wait to be ordered on OL_red          /* form the slug */
   travel to conv:slug_cleared
   if A_last = true then free R_separate
   travel to conv:inspect
   use R_inspect for e 10 sec
   travel to conv:goodbye
   send to die
end

begin P_blue arriving
   move into Q_blue
   move into conv:blue_on
   travel to conv:blue_slug
   inc V_blue by 1
   if V_blue = 8 then                    /* release the slug */
      begin
         get R_separate
         order 8 loads from OL_blue to continue
         in case order not filled backorder on OL_blue
         set V_blue to 0
         set A_last to true
      end
   travel to conv:blue_wait
   wait to be ordered on OL_blue         /* form the slug */
   travel to conv:slug_cleared
   if A_last = true then free R_separate
   travel to conv:inspect
   use R_inspect for e 10 sec
   travel to conv:goodbye
   send to die
end
```

The logic to control the formation and release of slugs is defined in two separate procedures, one for each load type. The P_red and P_blue arriving procedures are similar to the P_part arriving procedure that is defined in example model 13.2 (see “Example 13.2: Creating slugs on one entrance lane” on page 13.11). The procedures form a slug by delaying the first load in each slug on an order list and then releasing the slug by backordering each of the remaining loads in the slug. Example model 13.3 demonstrates some important differences in the way that slugs are released, however.

Before releasing a slug, the last load in each slug first claims a single-capacity dummy resource, R_separate. The load then orders the correct number of loads off the order list and resets the value of the variable that is used to count the number of loads in the slug. The load also sets the value of the load attribute A_last to true. This load attribute is used as a flag to indicate the last load in the slug.

Notice that a new station, “slug_cleared,” has been placed on the conveyor. All loads travel to this station before continuing on to the station “inspect.” As each load passes the “slug_cleared” station, they check to see if the value of their A_last attribute is true. When the last load passes by the station, the value is true, so the load releases the dummy resource R_separate.

Because the dummy resource is claimed from the moment a slug releases until the last load has passed the station “slug_cleared,” the other slug cannot release during this time period. If the slug is ready for release, the last load is delayed until it can claim the resource R_separate. Using the dummy resource prevents the slugs from releasing at the same time and merging with one another as they travel down the vertical section of conveyor.

Note

You could claim any entity with limited capacity to simulate the separation of slugs. In chapter 14, “Additional Features,” you will learn how to use counters, which could be used in place of the dummy resource in this example.

Sorting loads by priority

You can sort loads on an order list using a criterion other than entry time. In the following example, you will sort loads using the load attribute `priority`.

Priority is a standard load attribute of type integer that is used to indicate the relative precedence of loads on an order list. By default, when an order list is sorted by priority, loads with lower integer values are sorted above loads with higher integer values (the load with the lowest integer value is the highest priority). The default priority value of all loads is 0. You can set a load’s priority to a higher value to decrease its precedence on the list or to a lower value (including negative numbers) to increase its precedence on the list. For example, to set the priority of the current load to 1, use the following action in an arriving procedure:

```
set priority to 1
```

To see how to use the `priority` attribute with order lists, look at the following example.

Example 13.4: Modeling load priority

In this example, loads are processed in a machine shop. Some loads are high priority (hot). Hot loads are processed in the machine center before the other loads in the system. The time between arrivals for hot loads is exponentially distributed with a mean of 10 minutes. The time between arrivals for other loads is exponentially distributed with a mean of 40 seconds. All loads complete the same steps in the system.

Loads first move into an infinite-capacity waiting queue, where they wait to use a single-capacity machine. The machine processes each load for a time that is normally distributed with a mean of 30 seconds and a standard deviation of 2 seconds. The machine has its own single-capacity queue for processing.

After processing at the machine, loads are removed from the system.

You will simulate the system for 10 days.

Defining the order list in example model 13.4

The logic that is used to simulate example 13.4 has already been defined in the base version of the example model. However, the logic is currently commented, because the order list in the logic has not yet been defined. Therefore, you need to import the model and define the order list.

To define the order list:

Step 1 ***Import*** a copy of the *base* version of example model 13.4.

Step 2 ***Edit*** the model logic and ***delete*** the comment markers `/*` and `*/` at the beginning and end of the source file.

Take a moment to review the model logic, shown below:

```
begin P_inshop arriving
   move into Q_mach_wait
   if R_mach current value = 0   /* If machine is idle */
      then
         send to P_machine
   if load type =  L_hot then
      set priority to 1
   else
      set priority to 2          /* 2 is a lower priority than 1 */
   wait to be ordered on OL_wait
      /* sorted by priority, lowest integer values first */
end

begin P_machine arriving
   move into Q_mach
   use R_mach for n 30,2 sec
   order 1 load from OL_wait to P_machine
   send to die
end
```

Two load types are defined in the system, L_hot and L_other. Both types of loads are sent to the P_inshop arriving procedure.

Each load first moves into the infinite-capacity queue Q_mach_wait. If the resource R_mach is currently idle (no loads are claiming the resource), the current load is immediately sent to the P_machine arriving procedure; the load moves into the processing queue and claims the resource. If, however, the resource is currently processing a load, a priority value is assigned to the load and the load is ordered to wait on an order list. Loads of type L_hot are assigned a priority value of one and loads of type L_other are assigned a priority value of two, which causes L_other loads to be sorted below L_hot loads on the order list.

Each time a load completes processing at the machine, it orders a load off the order list to the process P_machine, causing the next waiting load to execute the P_machine arriving procedure. If there are no loads currently waiting on the list, the order action is ignored (a backorder is *not* created).

Step 3 ***Save*** and ***quit*** the source file. The Error Correction window appears, indicating that OL_await is undefined.

Step 4 To define the order list, ***select*** Define. In the Define as drop-down list, ***select*** Orderlist.

Step 5 ***Click*** Define as. The Define An Order List window appears.

Step 6 In the Sort list, ***select*** Priority.

Step 7 The tie breaker is already set to First In First Out, so ***click*** OK to close the Define An Order List window.

You have now defined the order list OL_wait.

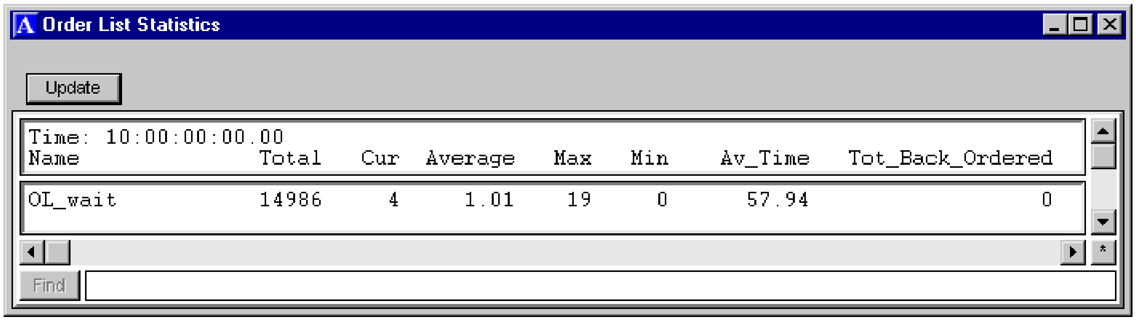
Displaying order list statistics

To verify that loads are being ordered to wait in example model 13.4, look at the order list statistics.

To display summary statistics for the order list OL_wait:

Step 1 ***Export*** and ***run*** the model to completion.

Step 2 From the Order Lists menu, ***select*** Statistics Summary. The Order Lists Statistics window appears, as shown below:



```
Time: 10:00:00:00.00
Name                  Total    Cur  Average   Max   Min     Av_Time    Tot_Back_Ordered
OL_wait               14986      4     1.01    19     0       57.94                   0
```

Order list summary statistics

Order list statistics are defined as follows:

Name The name of the order list.

Total The total number of loads that were ordered to wait on the order list.

Cur The number of loads that are currently on the order list.

Average The average number of loads that were on the order list at the same time.

Max The maximum number of loads that were on the order list at the same time.

Min The minimum number of loads that were on the order list at the same time.

Av_Time The average time that each load spent on the order list.

Tot_Back_Ordered The total number of loads on the order list that filled back orders.

The summary statistics for the order list OL_wait indicate that there were 14,986 loads that attempted to use the resource R_mach, but were ordered to wait on an order list until the machine became available. The average time that each load spent on the order list was about one minute (57.94 seconds).

Step 3 ***Quit*** the model.

Summary

In this chapter, you learned how to delay loads indefinitely using order lists. Practical applications for order lists include modeling a packaging operation, modeling conveyor slugging, and prioritizing loads in a system.

Exercises

Exercise 13.1

Copy the *final* version of example model 13.1 to a new directory. Edit the copied model using the information provided below.

After bar coding, boxes move into an infinite-capacity queue, where they are assembled into cases; four boxes form a case. Completed cases then move into another infinite-capacity queue where they are loaded onto pallets; four cases form a pallet. Completed pallets then leave the system. You do not need to model a time delay for the case and pallet assemblies—they occur instantaneously.

The size of cases is defined as:

X = 4
Y = 4
Z = 4

The size of pallets is defined as:

X = 5
Y = 5
Z = 5

Model these additional operations using order lists. Use variables to count the total number of completed cases and pallets. Run the model for 10 days. Verify the output data to make sure that the assemblies, cases, and pallets are being formed properly.

Exercise 13.2

Copy example model 13.2 to a new directory. Edit the copied model using the information provided below:

Immediately after the inspection station, loads on the conveyor form slugs consisting of four loads per slug. Completed slugs travel to the end of the conveyor, where they move into an infinite-capacity queue and are assembled into cases (each slug forms a case). The time required to package a case of loads is normally distributed with a mean of 3 minutes and a standard deviation of 30 seconds.

Model these additional operations using order lists and run the model for five days.

What was the maximum number of loads that were on section “sec5” at the same time?

Exercise 13.3

Copy example model 13.2 to a new directory.

Edit the model so that the size of loads is defined as:

X = 3
Y = 4
Z = 1

Edit the model so that loads form slugs consisting of six loads each (instead of ten loads each).

Run the model for five days.

What was the maximum number of loads that could not get on the conveyor due to congestion?

Exercise 13.4

Copy the *final* version of example model 13.4 to a new directory.

First, run the copied example model for 200 days and find the average time that loads spent in process “P_inshop.”

Now edit the model using the information provided below:

In addition to the current load creations, a third load type is created and sent to the process “P_inshop” with an interarrival time that is exponentially distributed with a mean of 20 minutes. The new load type is given higher priority than the other two load types on the waiting list for resource R_machine.

After editing the model, run the simulation for 200 days and determine the *percentage increase* in the average time that loads spent in process “P_inshop” due to the processing of the new load type.

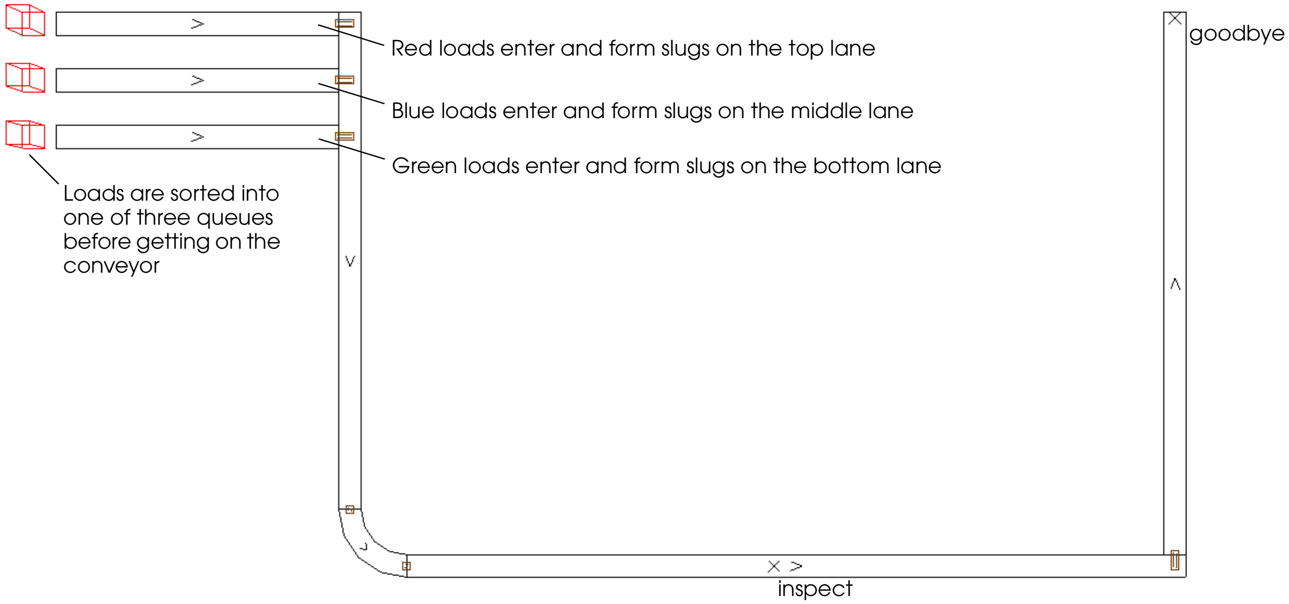
Exercise 13.5

Copy example model 13.2 to a new directory. Edit the copied model to simulate the system described below.

Three different types of loads are processed in this system; each load type is assigned a unique color. The interarrival rate, load size, and slug size for each type of load is defined in the following table:

Load color	Interarrival time	Load size	Slug size
red	exponential 30 seconds	2×3×1	10
blue	exponential 45 seconds	3×2×1	12
green	exponential 60 seconds	3×4×1	8

Loads are sorted by color into one of three infinite-capacity queues. Loads then get on one of three conveyor sections, as shown below:


Red loads enter and form slugs on the top lane
Blue loads enter and form slugs on the middle lane
Green loads enter and form slugs on the bottom lane
Loads are sorted into one of three queues before getting on the conveyor
goodbye
inspect

At the end of each entrance section, the loads form slugs. The number of loads in each slug is determined by the load's color (see the table above). When a complete slug is formed, it travels to the inspection station, where each load is inspected for a time that is exponentially distributed with a mean of 10 seconds.

The slugs are prevented from mixing when transferring from the entrance lanes to the vertical section of conveyor. If slugs on multiple entrance lanes are ready to release at the same time, only one slug is released while the others delay.

After inspection, loads travel to station "goodbye," where they leave the system.

Run the model for 10 days.

What was the maximum number of loads of each type that could not get on the conveyor due to congestion?

Chapter 14

Additional Features

Chapter 14

Additional Features

This chapter discusses several features that can be useful when building a model in the AutoMod software:

- **Counters** – Allow you to track custom statistics (like a variable) and limit loads using capacity (like a queue or resource)
- **Labels** – Allow you to place text in your model and update it throughout the simulation
- **Tables** – Allow you to track and tabulate custom statistics
- **Subroutines** – Allow you to reuse code for more efficient programming
- **Functions** – Allow you to perform calculations and reuse code for more efficient programming

Collecting custom statistics and controlling capacity with counters

Counters are generic entities used to:

- Track integer values (like an integer variable)
- Control capacity (like a queue or resource)

A **counter** is an integer entity with a value that can be set, incremented, and decremented like a variable using any positive integer (See “Setting variable and load attribute values” on page 7.7 of the “Advanced Process System Features” chapter for more information). Counters are initially set to zero, which is their minimum value (counter values cannot be negative).

You can use counters for the same types of things for which you have used integer variables, such as controlling loops or tracking the number of loads in the system. Use a counter instead of a variable when you are interested in statistics and attributes such as average, current, minimum, maximum, total and average time, which are not available for a variable. If you are not interested in the statistics, use a variable, because it requires less CPU time than a counter, which allows your model to run more quickly.

Counters have capacity, like a resource or a queue, so you can use them to delay a load. For example, you could limit the number of loads in the process P_2 to one load at a time using a counter, C_P2cap, of capacity one, as shown below:

```
begin P_1 arriving
   use R_resource for 5 min
   inc C_P2cap by 1 /* capacity one */
   send to P_2
end
begin P_2 arriving
   use R_resource for 10 min
   dec C_P2cap by 1
   send to die
end
```

The first load increments C_P2cap by one and enters P_2. The counter’s capacity is one, meaning it cannot be incremented any higher. Therefore, as long as the first load is in P_2, all other loads in P_1 are delayed at the `increment` action until the counter is decremented.

Example 14.1: Tracking the number of red and blue loads in the system

Example model 14.1 is based on example 12.3. The model uses vehicles to carry red and blue loads through the system. Example model 14.1 uses counters to answer the following questions for a five-day simulation:

- What is the maximum number of loads (both red and blue) in the system at any time?
- What is the average and maximum number of red loads in the system at any time?
- What is the average and maximum number of blue loads in the system at any time?
- How many battery replacements occur?

Defining counters

You will need four counters to track the required information:

Counter name	Purpose
C_insystem	Tracks the total number of loads in the system
C_red	Tracks the number of red loads in the system
C_blue	Tracks the number of blue loads in the system
C_swap	Tracks the number of battery replacements that occur

To define the necessary counters:

Step 1 ***Import*** a copy of the *base* version of example model 14.1.

Step 2 ***Edit*** the model logic and ***delete*** the comment markers `/*` and `*/` at the beginning and end of the source file. The model's logic uses the necessary counters, as shown in bold below:

```
begin P_agvsys arriving
   inc C_insystem by 1 /* a load enters the system */
   move into Q_entry
   if load type = L_red then
      begin
         inc C_red by 1 /* add one red */
         move into pm:red_on
         travel to pm:red_insp
         use R_insp for e 3 min
         travel to pm:red_drop
         dec C_red by 1 /*subtract one red*/
         dec C_insystem by 1 /* a load leaves the system */
         send to die
      end
   else
      begin
         inc C_blue by 1 /* add one blue */
         move into pm:blue_on
         travel to pm:blue_in
         move into Q_blue_in
         use R_blue for n 4, .5 min
         move into Q_blue_out
         move into pm:blue_out
         travel to pm:blue_drop
         dec C_blue by 1 /*subtract one blue*/
         dec C_insystem by 1 /* a load leaves the system */
         send to die
      end
end

begin P_init arriving
   clone pm vehicles size to P_swap
   send to die
end

begin P_swap arriving
   set A_index = P_swap total
   wait for 480*(A_index-1)/(pm vehicles size) min
   while 1=1 do
      begin
         wait for n 480, 60 min
         move into pm:swap_area
         use R_swap for 15 min
         inc C_swap by 1 /* count number of battery swaps */
         move into Q_hide
      end
end
```

When a load enters the arriving procedure, it increments the counter C_insystem. Depending on its load type (red or blue), the load increments C_red or C_blue, then moves into the correct location to be processed. When finished processing, the load decrements the counter for its load type and the system counter C_insystem. Vehicle battery swaps are tracked using the counter C_swap. None of these counters are used to control capacity, only to count events. Because you do not know how many loads are in the model, you will define the counters as infinite capacity.

Step 3 From the File menu, ***select*** Save & Quit. The Error Correction window opens, indicating that C_insystem is undefined.

Step 4 To define the counter, ***select*** Define. In the Define as drop-down list, ***select*** Counter.

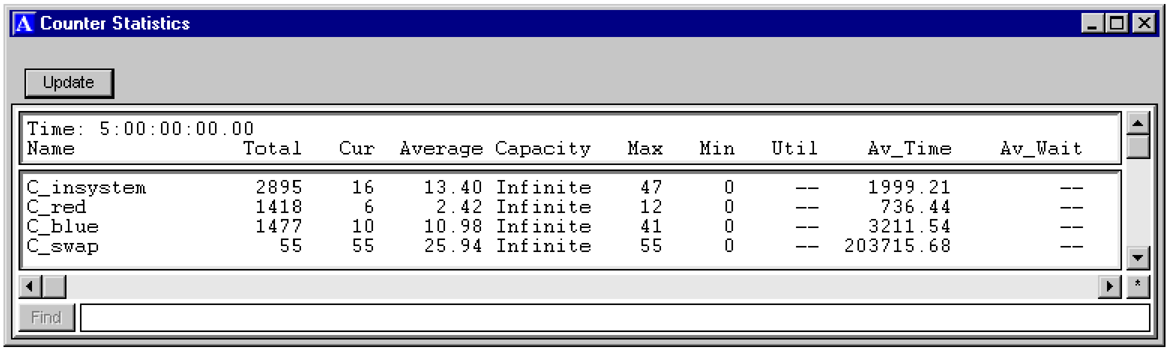
Step 5 ***Click*** Define as. The Define a Counter window opens.

Step 6 Because the counter is set to infinite capacity by default, ***click*** OK to define the counter.

Step 7 ***Define*** C_red, C_blue, and C_swap as infinite-capacity counters.

Step 8 ***Export*** and ***run*** the model to the end of the simulation.

Step 9 From the Counters menu, ***select*** Statistics Summary. The Counter Statistics window opens.



```
Time: 5:00:00:00.00
Name                 Total   Cur  Average Capacity   Max   Min   Util    Av_Time      Av_Wait

C_insystem            2895    16    13.40 Infinite    47     0     --    1999.21           --
C_red                 1418     6     2.42 Infinite    12     0     --     736.44           --
C_blue                1477    10    10.98 Infinite    41     0     --    3211.54           --
C_swap                  55    55    25.94 Infinite    55     0     --  203715.68           --
```

Counter statistics

As you can see, counter statistics are similar to statistics for other entities. Use these statistics to answer the following questions:

1. What is the maximum number of loads (both red and blue) in the system at any time?
   The maximum value of C_insystem is 47.
2. What is the average and maximum number of red loads in the system at any time?
   The average value of C_red is 2.42 and the maximum value is 12.
3. What is the average and maximum number of blue loads in the system at any time?
   The average value of C_blue is 10.98 and the maximum value is 41.
4. How many battery replacements occur?
   The total value of C_swap is 55.

This model uses counters to perform their most basic function, that is, counting things. You can also use a counter’s capacity. For example, you could limit how many red or blue loads could be in the system at one time. For more information about counters, see the “Process System” chapter in volume 1 of the *AutoMod User’s Manual*, online.

Displaying text in the Simulation window with labels

A **label** is text that you add to your model's graphics. Labels can contain static text (text that does not change during the simulation), such as the name of a work area. Labels can also contain dynamic text that is updated based on events in the simulation, such as the value of a variable or counter that is continually updated. You can update the text in a label using the `print` action.

For example, you could print the total number of loads in the system to the screen whenever the counter C_insystem gets incremented or decremented. Then, as you run the model, you will always be able to see the current number of loads in the system. To print the current value of C_insystem to a label, use the following syntax:

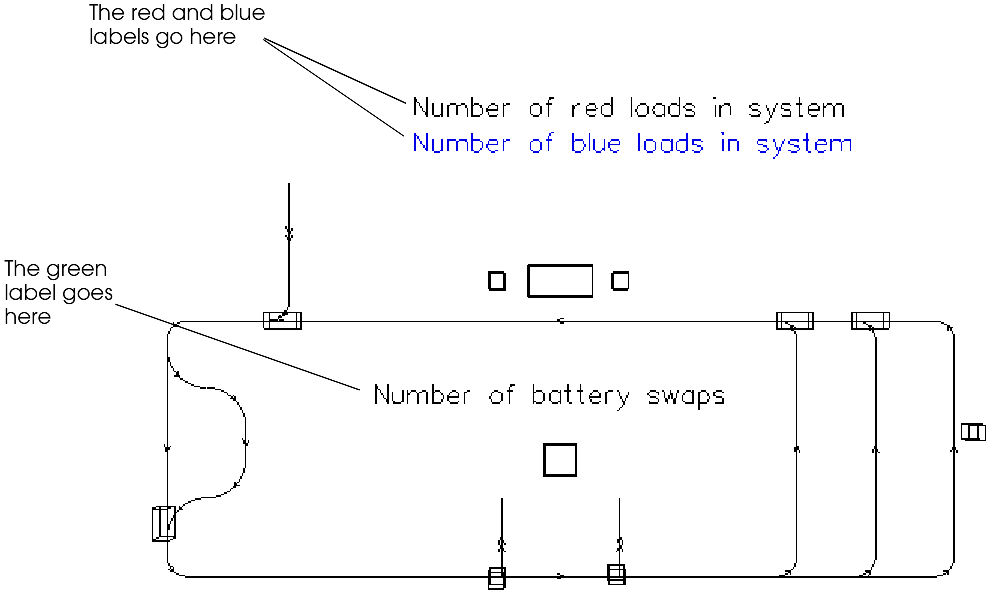
```
print “Total in system = ” C_insystem current value to LBL_insystem
```

This logic prints the words “Total in system = ”, followed by the current value of the counter C_insystem, to the label LBL_insystem.

Like other process system entities, labels are defined from the Process System palette. You place them graphically in your model just as you would place a graphic for a resource or queue.

Defining labels

In example model 14.1, you need to add labels to print the current values for the number of red and blue loads in the system. You also need to print the total number of battery swaps that have been completed. The labels should be red for red loads, blue for blue loads, and green for the battery swaps. The labels should be placed as shown below:


The red and blue labels go here
Number of red loads in system
Number of blue loads in system
The green label goes here
Number of battery swaps

Labels for example model 14.1

To define the labels for example model 14.1:

Step 1 ***Edit*** the model.

Step 2 From the Process System palette, ***select*** Labels. The Labels window opens.

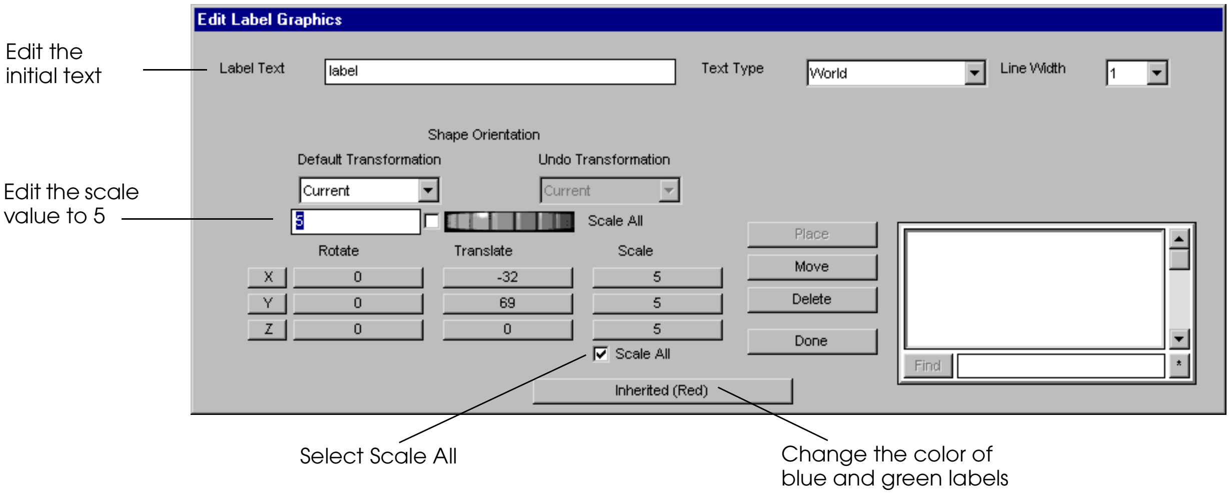
Step 3 ***Click*** New to define a label.

Step 4 To define the label for red loads, ***name*** the label “LBL_red” and ***click*** OK. The new label appears in the Labels window.

Step 5 To place the label graphically, ***click*** Edit Graphic. The Edit Label Graphics window opens.

Step 6 ***Click*** Place and ***drag*** to place the label in the Work Area window in the center of the top of the picture, as shown in “Labels for example model 14.1” on page 14.8.

Step 7 ***Select*** Scale All. ***Edit*** the scale value to 5, as shown below:



Edit Label Graphics

Label Text: label
Text Type: World
Line Width: 1

Shape Orientation
Default Transformation: Current
Undo Transformation: Current

5 □ Scale All

	Rotate	Translate	Scale
X	0	-32	5
Y	0	69	5
Z	0	0	5

☑ Scale All

Edit the initial text

Edit the scale value to 5

Select Scale All

Change the color of blue and green labels

Step 8 ***Press*** Enter. The text size is increased.

Step 9 ***Change*** the Label Text to “Number of red loads in system” and ***press*** Enter. This changes the label’s initial text (the text that is displayed until you print to the label for the first time).

Step 10 By default, the label’s text is red, so ***click*** Done.

Step 11 ***Define*** the following labels, ***place*** their graphics, ***change*** their color (using the Inherited (Red) button shown above), ***change*** their scale, and ***edit*** their initial text:

Label name	Color	Initial text	Scale
LBL_blue	Blue	Number of blue loads in system	5
LBL_swap	Green	Number of battery swaps	5

Step 12 ***Export*** the model.

Now that you have defined and placed the labels, you must edit the source file to print the current value of the counters to the labels.

Printing to labels

Whenever a counter’s value changes, you need to print the new value to the correct label. The counters that track the number of red and blue loads in the system, C_red and C_blue, are each incremented and decremented once in the model logic. Therefore, you need to update the appropriate label after the `increment` and `decrement` actions in the model logic. Similarly, when C_swap is incremented, the label must be updated.

To print the current value of the counters to the correct label:

Step 1 ***Edit*** the source file.

Step 2 ***Add*** the `print` actions for each label to the P_agvsys and P_swap procedures, as shown in bold below:

```
begin P_agvsys arriving
    inc C_insystem by 1 /* a load enters the system */
    move into Q_entry
    if load type = L_red then
        begin
            inc C_red by 1 /* add one red */
            print “Reds in system = ” C_red current value to LBL_red
            move into pm:red_on
            travel to pm:red_insp
            use R_insp for e 3 min
            travel to pm:red_drop
            dec C_red by 1 /*subtract one red*/
            print “Reds in system = ” C_red current value to LBL_red
            dec C_insystem by 1 /*a load leaves the system */
            send to die
        end
    else
        begin
            inc C_blue by 1 /* add one blue */
            print “Blues in system = ” C_blue current value to LBL_blue
            move into pm:blue_on
            travel to pm:blue_in
            move into Q_blue_in
            use R_blue for n 4, .5 min
            move into Q_blue_out
            move into pm:blue_out
            travel to pm:blue_drop
            dec C_blue by 1 /*subtract one blue*/
            print “Blues in system = ” C_blue current value to LBL_blue
            dec C_insystem by 1 /* a load leaves the system */
            send to die
        end
end
```

```
begin P_swap arriving
   set A_index = P_swap total
   wait for 480*(A_index-1)/(pm vehicles size) min
   while 1=1 do
      begin
         wait for n 480, 60 min
         move into pm:swap_area
         use R_swap for 15 min
         inc C_swap by 1 /* count number of battery swaps */
         print “Swaps = ” C_swap current value to LBL_swap
         move into Q_hide
      end
end
```

Step 3 ***Select*** Save & Quit.

Step 4 ***Export*** and ***run*** the model. The label values update whenever a load enters or leaves the system.

Tip ☞

Printing text to a label slows down the simulation. In our model, we are printing the string “Number of <color> loads =” every time we update the label. Printing the entire label is inefficient, because the text part of the label does not change, only the number does. Therefore, you could use two labels for each value: a label for the static part of the message, and another label that is updated dynamically with the integer value of the counter. Updating only the integer value would make the model run faster.

Collecting custom statistics with tables

The AutoMod software provides numerous statistics for every model. However, each model and each simulation project are different, and depending on the objectives of your study, you may need information that is not gathered automatically. **Tables** allow you to gather your own statistics and sort them into frequency classes. Tables provide data such as the mean and standard deviation for your tabulated values. You can use tables to track things such as:

- Product cycle times
- Process cycle times
- Interarrival rates
- Station idle time (or time in any state)
- Throughput over time (number of loads processed per day)

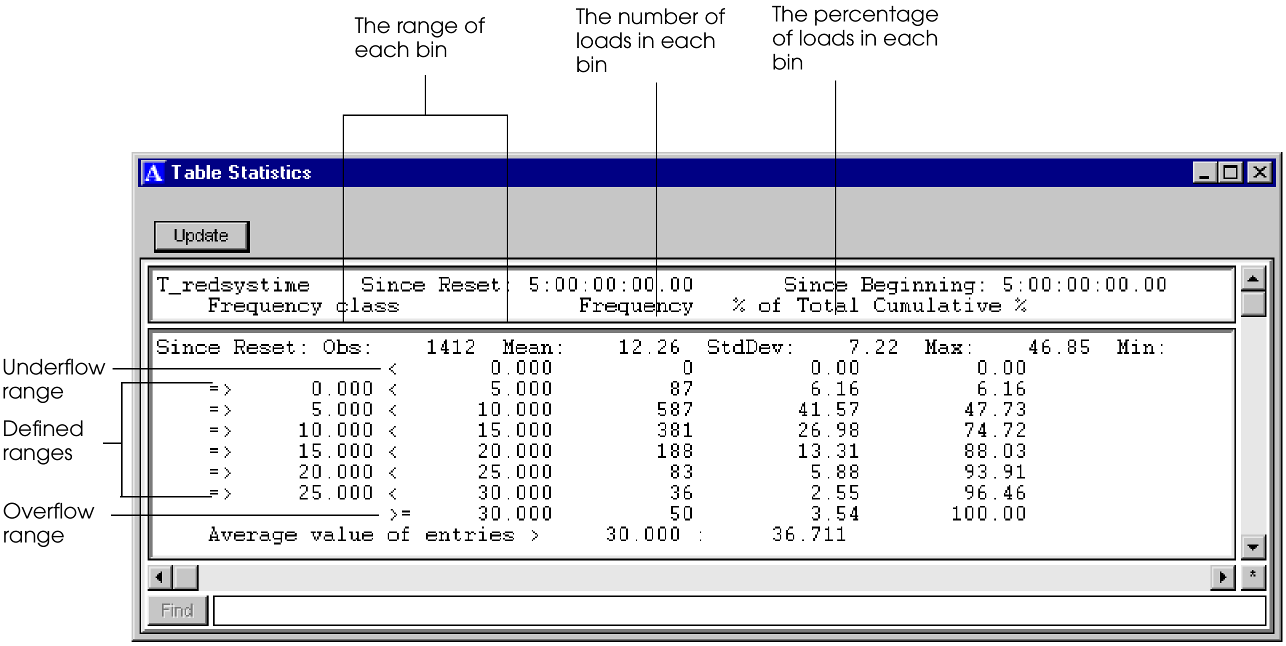
You can collect data for tables using the `tabulate` action. For example, to track the time in system for red loads in a table called “T_redsystime,” use the following logic:

```
tabulate (ac – A_timestamp)/60 in T_redsystime
```

Assuming the load’s A_timestamp attribute is set to the absolute clock when the load enters the system, this logic calculates the time in system (in minutes) and tabulates it in the table.

Tables are similar in format to a histogram. In order to define a table, you need to define the number of bins, or categories, of the data. You also need to describe the width, or range of values, for each bin.

For T_redsystime, suppose there are six bins with five-minute widths, creating six ranges of time: 0-5 minutes, 5-10 minutes, and so on up to 25-30 minutes. Whenever a load leaves the system, its time in system is added to the appropriate bin, as shown below:



```
T_redsystime    Since Reset: 5:00:00:00.00      Since Beginning: 5:00:00:00.00
    Frequency class          Frequency   % of Total Cumulative %

Since Reset: Obs:    1412  Mean:    12.26  StdDev:    7.22  Max:    46.85  Min:
                      <       0.000          0         0.00        0.00
    =>     0.000 <       5.000         87         6.16        6.16
    =>     5.000 <      10.000        587        41.57       47.73
    =>    10.000 <      15.000        381        26.98       74.72
    =>    15.000 <      20.000        188        13.31       88.03
    =>    20.000 <      25.000         83         5.88       93.91
    =>    25.000 <      30.000         36         2.55       96.46
                      >=      30.000         50         3.54      100.00
    Average value of entries >     30.000 :      36.711
```

Example table for T_redsystime

The table statistics tell you the frequency for each bin range, that is, how many loads had a time in system within a given five-minute range. When a value falls outside of any of the ranges, it is placed in an underflow or overflow category. In this table, the first range starts at zero, so there cannot be any underflow values. There were 50 loads with times in the system greater than or equal to 30 minutes, as shown by the Frequency value for the overflow range.

Categories of table statistics

Tables include two categories of statistics:

- Table statistics
- Frequency statistics

Table statistics

Table statistics refer to tabulated values. Table statistics are defined as follows:

Name The name of the table.

Entries The number of tabulated values.

Mean The mean tabulated value.

Std Dev The standard deviation of the tabulated values.

Max The largest tabulated value.

Min The lowest tabulated value.

Tip ☞ You can display tabulated data in a histogram by defining a bar chart business graph. For information about defining business graphs, see “Displaying a business graph” on page 5.27 of the “Process System Basics” chapter.

Frequency statistics

Frequency statistics provide a frequency distribution of all entries in the table. Frequency statistics are defined as follows:

Frequency class The range of tabulated values being measured.

Underflow (<) The number of entries with values smaller than the lowest bin value.

Overflow (>) The number of entries with values larger than the highest bin value.

Frequency The number of tabulated values in this class.

% of Total The percent of total tabulated values in the frequency class.

Cumulative % The cumulative percent of total tabulated values since the initial frequency class. When the simulation has finished, the last value in this column is 100 percent. (If there are no entries in the table, the value is 0 percent.)

Defining tables

For example model 14.1, you need to calculate the time in system, in minutes, for each load type. You also need to know the maximum and minimum times, in minutes, for loads of each type. The best way to determine this kind of information is to use tables.

Define two tables: one that tracks the time in system, in minutes, for red loads, and another to track time in system, in minutes, for blue loads. The tables should have 6 bins (ranges) each, with few or no overflows (values that do not fit into the defined bins).

To define the tables in example model 14.1:

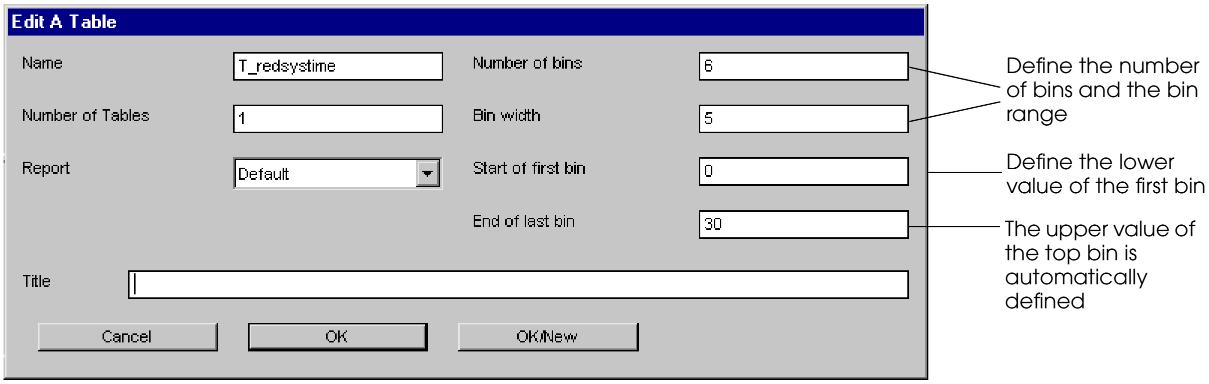
Step 1 ***Edit*** the model.

Step 2 From the Process System palette, ***select*** Tables. The Tables window opens.

Step 3 ***Click*** New to define the table to track the time in system of red loads.

Step 4 ***Name*** the table "T_redsystime."

Step 5 ***Define*** the number of bins as "6" and the width as "5," as shown below:



Edit A Table

Name: T_redsystime
Number of Tables: 1
Report: Default
Number of bins: 6
Bin width: 5
Start of first bin: 0
End of last bin: 30
Title:

Cancel
OK
OK/New
Define the number of bins and the bin range
Define the lower value of the first bin
The upper value of the top bin is automatically defined

Defining the T_redsystime table

Note Typing a number greater than one in the Number of Tables text box creates an array of tables. You will not use arrayed tables in this example model.

Step 6 ***Define*** the lower, or starting, value of the first bin range as "0" and ***press*** Tab. The end of the last bin range is automatically calculated as 30.

Step 7 ***Click*** OK.

Step 8 ***Define*** another table, called "T_bluesystime," with the following values:

Number of bins:10

Bin width: 15

Start of first bin: 0

These values create 10 ranges: 0-15, 15-30, and so on up to 135-150. The time in system for blue loads is much longer than for red loads, so the table for blue loads must span more values.

Now that you have defined the tables, you must edit the source file to update the correct table whenever a load finishes.

Updating tables

Whenever a red or blue load leaves the system, it decrements a counter and prints the updated value to the screen. Now the load must also track its time in system using a load attribute and update the table with that value before the load is sent to die.

To add each load’s time in system to a table:

Step 1 ***Edit*** the source file.

Step 2 ***Add*** the `set` and `tabulate` actions for each load type to the P_agvsys arriving procedure, as shown in bold below:

```
begin P_agvsys arriving
   inc C_insystem by 1 /* a load enters the system */
   move into Q_entry
   if load type = L_red then
      begin
         inc C_red by 1 /* add one red */
         print “Reds in system = ” C_red current value to LBL_red
         set A_timestamp to ac /* set the load’s entry time */
         move into pm:red_on
         travel to pm:red_insp
         use R_insp for e 3 min
         travel to pm:red_drop
         dec C_red by 1 /*subtract one red*/
         print “Reds in system = ” C_red current value to LBL_red
         dec C_insystem by 1 /* a load leaves the system */
         tabulate (ac - A_timestamp)/60 in T_redsystime
         send to die
      end
   else
      begin
         inc C_blue by 1 /* add one blue */
         print “Blues in system = ” C_blue current value to LBL_blue
         set A_timestamp to ac /* set the load’s entry time */
         move into pm:blue_on
         travel to pm:blue_in
         move into Q_blue_in
         use R_blue for n 4, .5 min
         move into Q_blue_out
         move into pm:blue_out
         travel to pm:blue_drop
         dec C_blue by 1 /*subtract one blue*/
         print “Blues in system = ” C_blue current value to LBL_blue
         dec C_insystem by 1 /* a load leaves the system */
         tabulate (ac - A_timestamp)/60 in T_bluesystime
         send to die
      end
end
```

Step 3 From the File menu, ***select*** Save & Quit. The Error Correction window opens, indicating that A_timestamp is undefined.

Step 4 ***Define*** A_timestamp as a load attribute of type time.

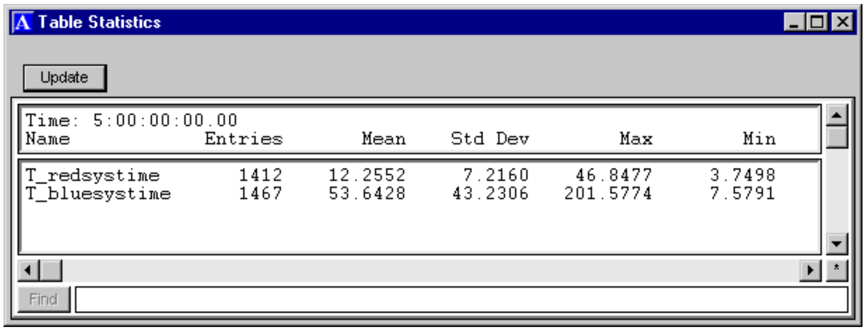
Step 5 ***Export*** and ***run*** the model to the end of the simulation.

Viewing table statistics

After running the model, view the table statistics.

To view table statistics:

Step 1 From the Tables menu, ***select*** Statistics Summary.



```
Time: 5:00:00:00.00
Name                Entries        Mean     Std Dev         Max         Min
T_redsystime           1412     12.2552      7.2160     46.8477      3.7498
T_bluesystime          1467     53.6428     43.2306    201.5774      7.5791
```

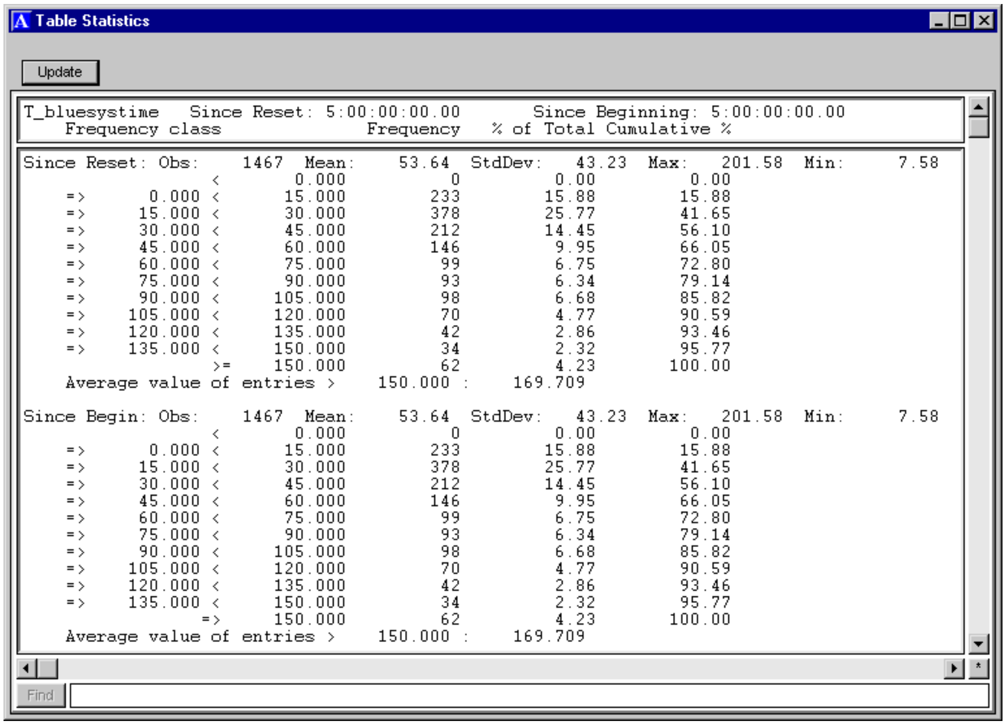
Summary statistics for tables

You created these tables to determine the time in system, in minutes, for each load type; this value can be determined from the mean value for each table. The maximum and minimum times (in minutes) for loads of each type is also displayed in the table summary.

For information about the number of loads in each bin, you can view the frequency statistics for an individual table:

Step 1 From the Tables menu, ***select*** Single Table. The Pick a Table window opens.

Step 2 ***Select*** T_bluesystime and ***click*** OK. The Table Statistics for the blue loads opens.



```
T_bluesystime   Since Reset: 5:00:00:00.00       Since Beginning: 5:00:00:00.00
    Frequency class                  Frequency   % of Total Cumulative %
Since Reset: Obs:    1467  Mean:    53.64  StdDev:    43.23  Max:    201.58  Min:      7.58
                  <      0.000           0        0.00        0.00
    =>      0.000 <     15.000         233       15.88       15.88
    =>     15.000 <     30.000         378       25.77       41.65
    =>     30.000 <     45.000         212       14.45       56.10
    =>     45.000 <     60.000         146        9.95       66.05
    =>     60.000 <     75.000          99        6.75       72.80
    =>     75.000 <     90.000          93        6.34       79.14
    =>     90.000 <    105.000          98        6.68       85.82
    =>    105.000 <    120.000          70        4.77       90.59
    =>    120.000 <    135.000          42        2.86       93.46
    =>    135.000 <    150.000          34        2.32       95.77
                  >=   150.000          62        4.23      100.00
    Average value of entries >    150.000 :    169.709

Since Begin: Obs:    1467  Mean:    53.64  StdDev:    43.23  Max:    201.58  Min:      7.58
                  <      0.000           0        0.00        0.00
    =>      0.000 <     15.000         233       15.88       15.88
    =>     15.000 <     30.000         378       25.77       41.65
    =>     30.000 <     45.000         212       14.45       56.10
    =>     45.000 <     60.000         146        9.95       66.05
    =>     60.000 <     75.000          99        6.75       72.80
    =>     75.000 <     90.000          93        6.34       79.14
    =>     90.000 <    105.000          98        6.68       85.82
    =>    105.000 <    120.000          70        4.77       90.59
    =>    120.000 <    135.000          42        2.86       93.46
    =>    135.000 <    150.000          34        2.32       95.77
                 =>    150.000          62        4.23      100.00
    Average value of entries >    150.000 :    169.709
```

Frequency statistics for T_bluesystime

You can determine the number of loads that had times that were not within the defined ranges by looking at the overflow category. The number of overflow values (as shown in the Frequency column) is 62.

Reusing logic with subroutines

When writing model logic, you sometimes need to have loads do similar things in more than one place. It can be tedious to write and edit logic that is duplicated in several places, and it can lead to errors if not all occurrences of the logic are updated correctly. To avoid duplication and errors, you can define the logic once and reuse it in several places by defining it as a subroutine. A **subroutine** is a global procedure that you can call from any process procedure. Subroutines make your model smaller and easier to update.

For example, suppose you wanted to write information to a custom report, or update tables whenever a load finishes processing. Rather than write the logic to update statistics in each arriving procedure, you can define a subroutine to update statistics and call the subroutine from any arriving procedure in which a load leaves the process.

A subroutine is written much like an arriving procedure:

```
begin S_subroutine
   <actions>
end
```

However, you do not send loads to a subroutine. Instead, loads call the subroutine from a procedure using the `call` action, as shown below:

```
begin P_process arriving
   use R_resource for 5 min
   call S_subroutine /* load does actions in subroutine */
   send to die
end
```

When a load calls a subroutine, the load stops executing the current arriving procedure and completes the actions in the subroutine. Then the load returns to the procedure from which it called the subroutine and continues with the next line. In this example, the load is sent to die.

Defining subroutines

In example model 14.1, each load type updates a counter for the total loads in system before it dies. The loads update the tables that track time in system by load type. Suppose you want to print the current number of loads in the system to the Message window whenever the C_insystem counter is updated, and also add a table to track the time in system for all loads. Because you need to add the logic to do this in two places (where red loads finish the model and where blue loads finish the model), using a subroutine is a good approach.

To define a subroutine to print the current loads in the system and update a new table with the total time that all loads spend in the system:

Step 1 ***Edit*** the model.

Step 2 ***Edit*** the source file.

Step 3 To define the new subroutine, ***type*** the following logic in the source file after the P_agvsys arriving procedure:

```
begin S_updatesystem
   dec C_insystem by 1
   print “Total loads in the system = ” C_insystem current value to message
   tabulate (ac - A_timestamp)/60 in T_totalsystime
end
```

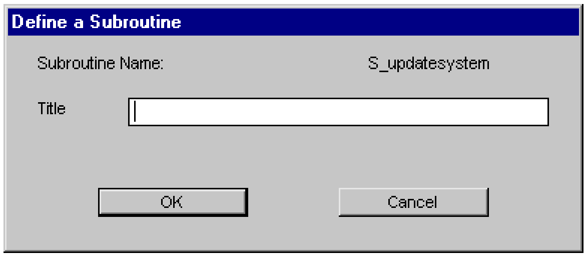
This subroutine updates the counter C_insystem for all load types, so you need to replace the `decrement` action for each load type with the call to the subroutine. This logic also prints the current value of the counter to the Message window and tabulates the time in system for all loads in a new table, T_totalsystime.

Step 4 ***Replace*** the `dec C_insystem by 1` action for each load type in P_agavsys with a call to the subroutine S_updatesystem, as shown in bold below:

```
begin P_agvsys arriving
          ...
          print “Reds in the system = ” C_red current value to LBL_red
          call S_updatesystem
          tabulate (ac - A_timestamp)/60 in T_redsystime
          send to die
       end
   else
          ...
          print “Blues in the system = ” C_blue current value to LBL_blue
          call S_updatesystem
          tabulate (ac - A_timestamp)/60 in T_bluesystime
          send to die
       end
end
```

Step 5 From the File menu, ***select*** Save & Quit. The Error Correction window opens, indicating that S_timestamp is undefined.

Step 6 ***Select*** Define as and ***select*** Subroutine from the Define as drop-down list. ***Click*** Define as. The Define a Subroutine window opens.



Define a Subroutine

Subroutine Name: S_updatesystem

Title

Defining a subroutine

Step 7 The only parameter you can change is the Title (an optional description of the subroutine). You do not need to specify a title, so ***click*** OK. The Error Correction window opens, indicating that T_totalsystime is undefined.

Step 8 ***Define*** T_totalsystime as a table with the following bin values:

Number of bins: 10

Bin width: 15

Start of first bin: 0

Step 9 ***Export*** and ***run*** the model to completion.

Step 10 ***Display*** Table Statistics, as shown below:

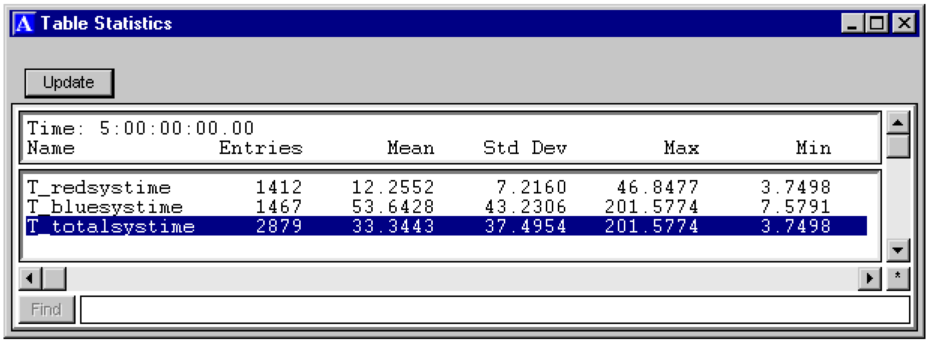


Table Statistics

```
Time:  5:00:00:00.00
Name                Entries       Mean    Std Dev         Max        Min

T_redsystime           1412    12.2552     7.2160     46.8477     3.7498
T_bluesystime          1467    53.6428    43.2306    201.5774     7.5791
T_totalsystime         2879    33.3443    37.4954    201.5774     3.7498
```

T_totalsystime

The statistics for T_totalsystime are displayed.

Performing calculations with functions

A **function** is a piece of code that is passed parameters, or input data, and calculates information. The function then sends back, or **returns**, the calculated information to the load executing the function. Any time that you need to repeatedly perform calculations or determine information, especially if it is based on the current events in the simulation, using a function is a good approach. For example, you can use a function to determine a load's next location, determine how long a vehicle delays at a station, or perform mathematical operations, such as finding the absolute value or square root of a number.

A function's format is:

```
begin F_functionname(parameter1,parameter2,…) function
    /* actions */
    return <value>

end
```

Functions can be used anywhere that you would use a value or an entity name in the model logic. For example, a function can replace an integer or real value, or a resource name. The function's return value is used in place of a constant value in the model logic. For example, rather than increment a variable by a fixed value:

```
inc V_count by 1
```

You could use a function to calculate a value and increment the variable by the function's return value, as shown below:

```
inc V_count by F_myfunction(parameter)
```

Because a function can be substituted anywhere that you use a value or a name, functions can be used with many actions. You can `set`, `increment`, or `decrement` a variable, attribute, or counter by a function's return value. You can `print` or `tabulate` a function's return value. You can also `call` a function, like you call a subroutine.

Characteristics of functions

When defining a function, you must define two characteristics:

- **Function type** (the type of data the function is returning)
- **Parameters** (data being used in the function)

Function type When you define a function, you must define the function's type, such as Integer, Real, or String (just as you do when you define a variable). The type of a function is determined by the type of data you want it to return. For example, if you write a function to return a real value, the function must be defined as type Real, and if you write a function to return a queue name, the function must be defined as type QueuePtr.

Parameters A parameter stores information that is passed into the function by the load that is executing the function call. Loads pass the information required for the function to perform a calculation. For example, if you wanted to find the square root of a number, the number must be passed as a parameter to the function. The function determines the square root of the number and returns the value of the square root. The square root value can then be used in the simulation. The parameters passed to a function can be of different data types than the function itself. When you define a function, you must declare the number and type of parameters that are passed to the function. For example, a function could accept two values of type Real and one value of type Integer. There is no limit on the number of parameters that you can pass to a function.

When writing a function, you must include two things:

- **Actions** that calculate the necessary information using the input data (parameters)
- **A return value** (the data the function is returning)

Actions When writing a function, you can use most, but not all, AutoMod actions. You cannot use any action that can cause a simulation delay, such as using a resource or waiting an amount of time. Of the actions that have been discussed in this textbook, the following actions cannot be used in a function:

- `free`
- `get`
- `move`
- `send`
- `travel`
- `use`
- `wait`
- `wait to be ordered`

In addition, you cannot `call` a subroutine from within a function and when using the `clone` action, you must include a pointer to the load you want to clone.

For a complete list of actions that are illegal or that have limitations when used in a function, refer to the AutoMod Syntax Help.

Return value When you write a function, it *must* return a value. To return a value from a function that you are defining in a source file, use the `return` action, such as:

```
return (timeinsys/60) /* return the time in system in minutes */
```

Types of functions

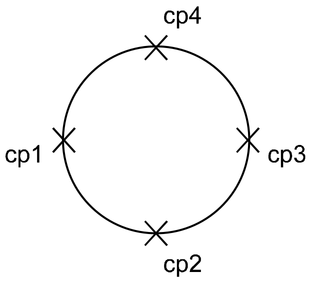
There are several types of functions that you can use in a model:

- User-defined functions
- Standard math library functions
- Time-specific functions
- Pre-defined AutoMod functions
- C functions (not discussed in this textbook)

User-defined functions

A user-defined function is an AutoMod function that you write to determine information that you need to use in the model. For example, you can determine which section of path or conveyor a load is on, what queue it is in, where to send a load, and so on. The function returns the desired information, which you can then use in the simulation.

For example, suppose you had a circular path with four stations on it, as shown below:



Layout for circular path

Loads use a resource at each location for processing, then continue to the next position. Loads must use a vehicle to travel to the points in order, starting with control point cp1. After cp4, the loads must return to cp1. You can use a function to determine the next location for each load by passing the load’s current location to the parameter “position”:

```
begin F_nextloc function
   inc position by 1 /* increment the load’s location to next point */
   if (position > 4) then begin
      set position = 1 /* after cp4, return to cp1 */
   end
   return position /* return the integer value of the next location */
end
```

This function is passed the integer value of the load’s current position via the load attribute A_position, and it returns an integer value, as shown below:

```
begin P_station arriving
   move into pmover:cp(A_position) /* cp1 */
   use R_resource for u 10,2 min
   set A_position to F_nextloc(A_position) /* now cp2 */
      /* set the load’s attribute to the return value of the function */
   travel to pmover:cp(A_position) /* cp2 */
   send to P_next
end
```

The comments in the procedure show the control point values that are used the first time the procedure is executed (assuming that the value of A_position is one).

The function increments the numeric value for the location name (stored in the parameter “position”) by one. If “position” exceeds four, it is reset to one. The next location value is then returned from the function, stored in the attribute A_position, and is used as the next location.

Each time a load completes processing at a point, the load is sent to the process P_station and travels to the next control point on the path.

Standard math library functions

A standard math library function is a pre-written function that resides in either the AutoMod or C libraries that come with the AutoMod software. Because the math functions are prewritten, you do not need to write code for them; you only need to define them in the model and use them.

Standard math library functions are defined as either type Real or Integer, and their parameters are either type Real or Integer. The following table lists some common math library functions, their types, and parameters (parameter types are shown in parentheses):

Function name	Function type	Parameters (type)	Definition
exp	Real	parameter 1 (Real)	returns the exponential function of parameter 1
pow	Real	parameter 1 (Real) parameter 2 (Real)	returns Parameter 1 raised to the parameter 2 power
abs	Integer	parameter 1 (Integer)	returns the absolute value of parameter 1
sqrt	Real	parameter 1 (Real)	returns the square root of parameter 1

To call one of these math library functions from an arriving procedure, you define the function as an AutoMod function, giving it the name, type, and parameter types shown in the table above (you can name the parameters anything you want).

For example, suppose each load in a model has an attribute called A_num that you want to raise to the third power. You would define a function, called “pow,” of type Real. The function requires two parameters of type Real: the first is the base number you want to raise to a power, and the second is the power to which you want to raise the base. In this example, the base number is the value in the load’s A_num attribute, and the power is 3.0.

You would define the following function, parameters, and variable in the model (you will learn how to define a function in “Defining functions” on page 14.23):

pow The function “pow” is a math function with two parameters of type Real: “base” and “power.”

base The first parameter is of type Real. The load’s attribute A_num value is passed to this parameter as the base value.

power The second parameter, which is also of type Real, is the power to which you want to raise A_num.

V_result A variable of type Real used to store the value that the “pow” function returns.

Once you have defined the function and its parameters, you can use the function to return a value. In this example, the return value is stored in the variable V_result using the `set` action:

```
set V_result to pow(A_num,3.0)  /* cube A_num */
print “A_num cubed =”,V_result to message
```

The parameters are passed to the function in parentheses and are separated by a comma. The parameters store the base and power values, the function performs the calculation, and the result is returned as a real number. The variable V_result is set to the return value and the value is printed to the Message window.

Time-specific functions

Time-specific functions are pre-defined AutoMod functions that allow custom processing at particular times before, during, and after the simulation. There are four time-specific functions:

- model initialization function (see “Defining the model initialization function” on page 7.12 of the “Advanced Process System Features” chapter for more information)
- model ready function
- model snap function
- model finished function

These functions are pre-defined, meaning the parameters, return values, and other information is already defined in the software. All you need to do is add the desired logic to the function in a source file.

Except for the model initialization function, using time-specific functions is not discussed in this textbook.

Help

For information about time-specific functions, refer to the AutoMod Syntax Help.

Pre-defined functions

The AutoMod software contains numerous pre-defined functions that you can use to determine information about the simulation (including the time-specific functions discussed above). AutoMod has pre-defined functions for movement systems, such as conveyor and path mover systems. There are also pre-defined functions to create custom interfaces for AutoMod using ActiveX controls. Using pre-defined functions is not discussed in this textbook.

Help

For information about pre-defined functions, refer to “Functions” in the AutoMod Syntax Help.

Defining functions

Currently, there are three tables in example model 14.1: one for each part type and one for all part types combined. All three tables are updated with the time (in minutes) of each load leaving the system:

```
tabulate (ac - A_timestamp)/60 in T_<tablename>
```

Because this calculation must be done in more than one place, a good approach would be to use a function to perform the calculation. Therefore, you need to define a function that converts each load’s A_timestamp attribute to minutes and returns the time so that it can be tabulated.

You also need to define a function to print the square of a red load’s time in system to the Message window before the load leaves the system.

To define the functions in the model:

Step 1 ***Edit*** the model.

Step 2 ***Edit*** the source file.

Converting time in system to minutes using a function

You will define a function to convert the time in system to minutes and replace the current `tabulate` actions with the function call.

Step 1 In the source file, ***type*** the following function after any process procedure:

```
begin F_time function
   return (ac - Timeinsys)/60 /*time in minutes*/
end
```

This function will be defined as type Time with one parameter, also of type Time, called “Timeinsys.” The function is passed a load’s time in system from the attribute A_timestamp. The function converts the time in system to minutes and returns the converted value.

Step 2 ***Find*** each `tabulate` action and ***replace*** it with a call to the function.

For example, for red loads:

```
tabulate (ac - A_timestamp)/60 in T_redsystime
```

becomes

```
tabulate F_time(A_timestamp) in T_redsystime
```

Replace the tabulate actions for the time in system of blue loads and the time in system of all loads, as well.

Step 3 From the File menu, ***select*** Save & Quit. The Error Correction window opens, indicating that F_time is undefined.

Step 4 ***Click*** Define and ***select*** Function from the Define as drop-down list. ***Click*** Define as.

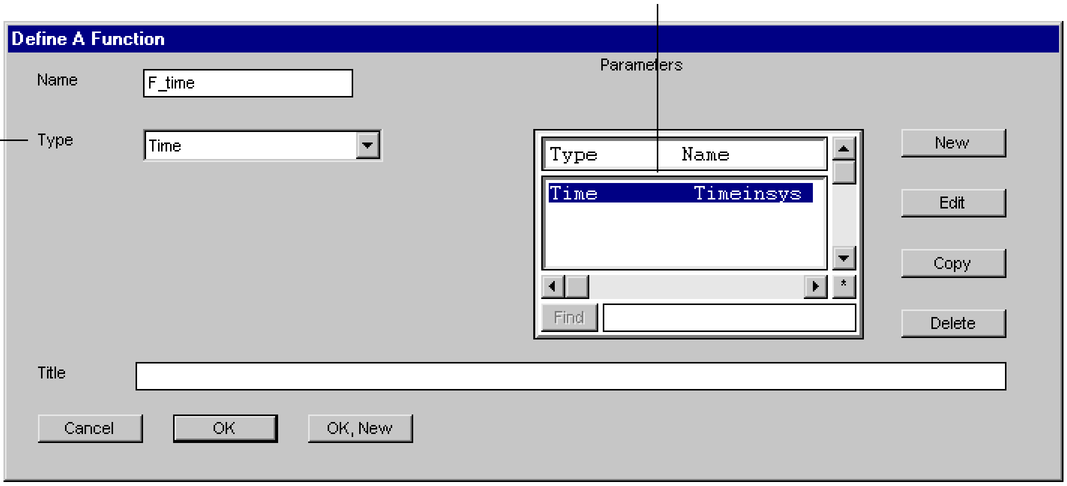
Step 5 ***Name*** the function “F_time” and ***select*** Time from the Type drop-down list.

Step 6 To define the parameter, ***click*** New. The Define a Parameter window opens.

Step 7 ***Name*** the parameter “Timeinsys” and ***select*** Time from the Type drop-down list. ***Click*** OK. The Define A Function window appears as shown below:

Define one parameter of type Time called Timeinsys

Select Time from the Type drop-down list



Define A Function

Name: F_time

Type: Time

Parameters

Type	Name
Time	Timeinsys

Title:

Defining the function F_time

Step 8 ***Export*** and ***run*** the model to completion.

Step 9 ***Compare*** these table statistics to those shown in “T_totalsystime” on page 14.18. Notice that you have not changed the values being tabulated, only how the values are put into the table.

Now you are ready to add a function to square the A_timestamp value and print it to the Message window before sending red loads to die.

Squaring a value using a math library function

The second part of the problem statement requires you to print the square of a red load’s time in system to the Message window before the load leaves the system. There is a standard math library function for raising a number to a power called “pow” (see “Standard math library functions” on page 14.22). The function “pow” requires two parameters of type Real: the base number and the power.

Because this function is a standard library function, you do not need to write the code for it; you only need to define its type and the correct parameters, then call it when necessary.

To use a math library function to print the square of a value:

Step 1 ***Edit*** the model.

Step 2 ***Edit*** the source file and ***type*** the following `print` action (shown in bold below) in the P_agvsys arriving procedure for red loads:

```
if load type = L_red then
       begin
           inc C_red by 1/* a red load enters */
           print “Reds in the system = ” C_red current value to LBL_red
           set A_timestamp to ac
           move into pm:red_on
           travel to pm:red_insp
           use R_insp for e 3 min
           travel to pm:red_drop
           dec C_red by 1/* subtract one red */
           print “Reds in the system = ” C_red current value to LBL_red
           call S_updatesystem
           tabulate F_time(A_timestamp) in T_redsystime
           print this load A_timestamp as .2, “squared =”, pow(A_timestamp,2.0)
               as .2 to message
           send to die
       end
```

This statement prints the red load’s current timestamp and the current timestamp squared, each rounded to two decimal places.

Step 4 From the File menu, ***select*** Save & Quit. The Error Correction window opens, telling you that “pow” is undefined.

Step 5 ***Click*** Define and ***select*** Function from the Define as drop-down list. ***Click*** Define.

Step 6 ***Name*** the function “pow” and ***select*** Real from the Type drop-down list.

Step 7 To define the first parameter, ***click*** New. The Define a Parameter window opens.

Step 8 ***Name*** the parameter “base” and define it as type Real. ***Click*** OK, New.

Step 9 ***Define*** the second parameter, named “power,” also of type Real. ***Click*** OK to close the Define a Parameter window.

Step 10 ***Click*** OK to define the function.

Step 11 ***Export*** and ***run*** the model. Watch the Message window. The time in system squared is printed to the Message window for red loads.

Summary

This chapter has given you a brief overview of several additional features that you can use as you develop more detailed models.

Counters can be used to track custom statistics. Counters are similar to variables, except counters have capacity.

Data, such as text and counter values, can be printed to the screen using labels. You can adjust the color, size, and position of labels the same way you can change the graphics for other entities in a model.

You can tabulate user-defined statistics in tables to gather mean, standard deviation, and frequency information.

Subroutines help you modularize and reuse code. Functions perform calculations and return a value that you can use in the simulation.

Exercises

Exercise 14.1

Copy your solution model to exercise 12.2 to a new directory. Edit the copied model and define seven vehicles in the path mover system, then complete the following:

a) Generate a table with tabulated values for each load's total time in system. Generate additional tables that categorize each load's total time in system by load type. All tables must have 8 bins and not more than 5 percent of the entries in overflow.
b) Use labels to print the tabulated average time that loads spend in the system to the Simulation window. Update the printed value each time ten loads leave the system.

Run the simulation for five days.

Exercise 14.2

Copy your solution model to exercise 12.3 to a new directory. Edit the copied model and define seven vehicles in the path mover system, then complete the following:

a) Define a counter to track the total number of loads currently in the system.
b) Use labels to display the counter's value in the Simulation window. Update the value each time a load enters or leaves the system.
c) Generate a table with tabulated values for each load's total time in system. The table must have 10 bins and not more than 5 percent of the entries in overflow.
d) Model the load's processing and inspection activities in two separate subroutines.

Run the simulation for five days.

Exercise 14.3

Create a new model and generate 10,000 random numbers for each of the following distributions:

a) Uniformly distributed values between 0 and 1.
b) Uniformly distributed values between 8 and 12.
c) Normally distributed values with a mean of 10 and a standard deviation of 0.667
d) Exponentially distributed values with a mean of 10.

Tabulate the values from each distribution in one of four tables that are distinguished by distribution type. Do not allow more than 5 percent of the entries in overflow.

Create a business graph for each distribution that displays a bar chart of the tabulated values.

Chapter 15

Warmup Analysis Using AutoStat

Chapter 15

Warmup Analysis Using AutoStat

As discussed in “Terminating versus non-terminating systems” on page 1.27 of the “Principles of Simulation” chapter, when simulating a non-terminating system, you need to remove the effects of a model’s initial conditions to accurately study its steady-state behavior. In AutoStat, the process of removing initial bias from a model is accomplished by defining a **warmup** snap for a model. Statistics gathered during the warmup snap are deleted, and analysis is based on the subsequent snap’s statistics.

Until now, whenever you have conducted an analysis in AutoStat, you were told the warmup time for the model. But it is important that you be able to perform warmup determination for your own projects as a simulation analyst.

This chapter discusses how to determine the warmup time for four different types of models:

- Systems with “classic” warmup behavior
- Explosive systems
- Cyclical systems
- Systems with extreme variation

For each type of system, the chapter gives guidelines for making a sound determination of the warmup period.

Understanding when a warmup determination is necessary

A **warmup analysis** is used to estimate how long it takes a system to reach steady state. Some models require a warmup time and some do not.

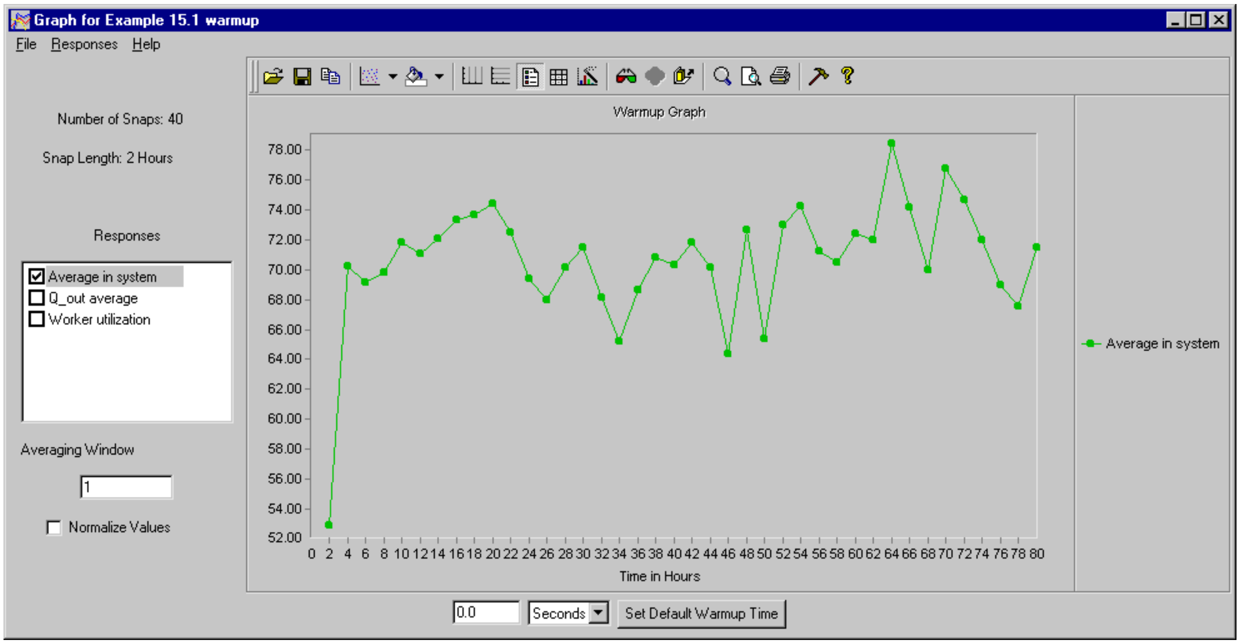
As described in “Terminating versus non-terminating systems” on page 1.27 of the “Principles of Simulation” chapter, terminating systems do not require a warmup. For example, if you are studying a bank, which starts empty of customers each day, or are trying to determine the throughput for one 8-hour cycle in a distribution center that begins with a new set of orders each day, a warmup time is not necessary.

Non-terminating systems, on the other hand, may or may not need a warmup time, depending on whether you are analyzing them during steady-state behavior. If a simulation starts with the system empty and idle, then you need to determine the system’s warmup time.

If you are not concerned with steady-state analysis, but with other conditions, your analysis may not require a warmup. For example, if you are studying a manufacturing system in which you are trying to understand how the system recovers from the insertion of rush orders, or a fiberglass-manufacturing firm when studying the transition to steady state after a forced shut down, you may not need a warmup period.

Using graphs to determine warmup time

After many years of research in simulation, and several attempts to base warmup analysis on a statistical test, most simulation analysts now agree that the best way to make a warmup determination is to use a graph of the responses for the model. Looking at the statistics of interest and determining when they represent steady state behavior is the easiest, most accurate way to determine when a system has warmed up. AutoStat uses graphs like the one shown below to help you determine warmup time:


Graph for Example 15.1 warmup

Number of Snaps: 40

Snap Length: 2 Hours

Responses

☑ Average in system
☐ Q_out average
☐ Worker utilization

Averaging Window: 1

☐ Normalize Values

0.0 Seconds

Warmup graph in AutoStat

Sometimes it is easy to determine when a model has warmed up and sometimes it is not. The first example model in this chapter, example model 15.1, illustrates the “ideal” case, in which the time to reach steady state is obvious. Models with an obvious warmup time are referred to in this chapter as systems with **classic** warmup behavior.

Many models do not demonstrate an obvious warmup, however. For example, in an **explosive system**, the utilization of one or more resources, $\rho,$ is greater than or equal to 1 (as discussed in “Queueing theory” on page 1.30 of the “Principles of Simulation” chapter). So one or more queues grow without bound, and the corresponding responses never level off. Explosive models by definition never reach steady state, and must be changed so that they are no longer explosive before a warmup or any other type of analysis can be conducted.

Some systems have **cyclical** behavior, for which it is more difficult to determine a warmup time than a system with classic warmup behavior. Suppose there is a vehicle that moves material in the facility and that every eight hours the vehicle goes to a battery swap area for 30 minutes. During the time the vehicle’s battery is being replaced, loads waiting to be delivered stack up. It may take hours to work off the backlog once the vehicle is operational again. This cyclical behavior *is* the steady state operation of the system, and can be understood with analysis.

Some models demonstrate **extreme variability**, in which behavior varies widely. It takes more analysis to perform warmup determination for models with extreme variability, but it can be done.

Preventing statistical inaccuracy when using random numbers

In the AutoMod software, whenever you use a random number for events such as load creation rates, machine failures, processing times, and so on, AutoMod gets the random number from a random stream of numbers. A **random stream** is a series of approximately 2.1 billion numbers that are statistically random and repeatable.

When conducting statistical analysis on a model with AutoStat, you must ensure that no random numbers are reused between replications; otherwise, the results from the runs may be correlated, causing statistical inaccuracy.

Note Random number reuse is also a concern if your model uses more than one random stream, but using multiple streams is not discussed in this textbook. All random events in this textbook are generated from a single default random stream, stream0.

For every replication of a model, AutoStat begins using random numbers at a specific location in the random stream called the **seed**. To ensure that each replication uses unique random numbers, you must define a **seed increment** value, which is the offset between starting locations in the stream for each replication.

To determine the starting location in the random number stream for a run, AutoStat starts at the beginning of the stream and increments by the seed increment value, which is 10,000 by default. Therefore, the first replication uses random numbers beginning at the 10,001st random number in the stream (the first 10,000 random numbers in the stream are not used). The second replication uses random numbers beginning at location 20,001 in the stream, and so on.

The seed increment must be large enough so that random numbers are not reused across replications. If each replication uses fewer than 10,000 random numbers, for example 6,000, the default seed increment value is adequate. For a model with 6,000 random events and a seed increment of 10,000, the first replication would use the 10,001st to 16,001st random numbers. The second replication would use the 20,001st to 26,001st random numbers, and so on, and none of the random numbers would be reused.

However, if there are more than 10,000 random events in a replication, some of the random numbers used for the first run would also be used for the second run, causing the results to be correlated. For example, if a replication uses 17,000 random numbers, the second run would use almost 7,000 of the same random numbers that were used in the first run.

If your model is using more than the default 10,000 random numbers, you must set the seed increment value to a larger number. To determine how many random numbers your model is using:

1. Run the model for the desired length of time.
2. Open the report file (see “Interpreting reports” on page 5.28 of the “Process System Basics” chapter), then find the Total for Random Number Streams.
3. Set the seed increment in AutoStat to a value several thousand numbers larger than the number in the report file to allow for variations in the random distributions. How to set the seed increment value is discussed in “Changing the seed increment” on page 15.11.

Understanding warmup parameters

A warmup analysis consists of running the model and taking short “snapshots” of what is happening over time. The snapshots are used to graph responses, such as WIP levels, to determine when the model reaches steady state. The process is as much an art as a science, because there are no fixed guidelines for how long to make a snap, how many runs to make, and so on; you must learn through experience. Every model is different, as well, so what works for one model may not work for another. However, this chapter provides general guidelines and examples to help you get started.

Later in this chapter you will learn how to define a warmup analysis in AutoStat. First you must understand the concepts and parameters involved in defining a warmup analysis:

Snap Length The length of time for a reporting period.

Number of Snaps The number of reporting periods for each replication.

Number of Replications The number of times that the model is run using different random numbers for each random event.

Averaging Window When viewing a warmup graph, you can choose to average response values from multiple snaps together to help smooth the graph so that you can detect a trend more easily. The averaging window value is the number of response values to average.

Guidelines for setting warmup parameters

When performing a warmup analysis, you are looking for an underlying trend in response values amid “noise.”A good analogy is the “signal to noise” ratio from the field of electronics. You must differentiate between the response variations that are normal, steady-state behavior and the response variations that represent a warmup trend. Warmup determination is an iterative process that involves setting parameters, making runs, viewing the results, and repeating the process until you are confident that you have determined the warmup time correctly. To set warmup parameters initially, use the following guidelines.

Setting the snap length

Your knowledge of the system being modeled is necessary to estimate the snap length. The snap length should be long enough so that at least one load completes during each snap. It is preferable to have many loads complete in each snap. Other events, such as failures, should occur in every snap, as well. For example, for a model with 10 minutes of maintenance occurring every 230 minutes, do not use a snap length of two hours, because only every other snap contains the maintenance event.

Setting the number of snaps

The more snaps you use, the more flexibility you will have when adjusting the warmup graph. In order to have a meaningful graph, use a minimum of 20 snaps; 30 to 50 snaps are preferable.

Setting the number of replications

The number of replications to use depends on the amount of variability in the output. The higher the variability, the more replications that are required. Start with three replications, and increase it if there is too much variability in response values to determine a warmup trend.

Note

The snap length, number of snaps, and number of replications are also affected by the speed of your computer and the time that you have available for analysis. For example, suppose you are conducting an analysis and are using 40 snaps of length 4 hours. If it takes your computer 2 hours to make one replication, and you only have 8 hours to do the warmup analysis, you are only going to be able to make 4 replications.

Setting a warmup graph's averaging window

All warmup graphs use a default averaging window value of 5. Increase and decrease the value of the averaging window in small increments to try to detect a trend. First, look at the graph with an averaging window of 1 and 2. Then use a larger value, such as 10 or 15. With the smaller window values, the graph will show high variability. With larger window values, the graph will smooth out so much that the first few points are in the same range as the rest of the graph, making it impossible to detect the transient phase. Choose an averaging window value between the high and low that results in both an obvious initial phase and a uniform steady state.

The appropriate value of the averaging window is related to the inherent variability in the output and the snap length. If changing the window value does not help you make a warmup determination, it may be necessary to adjust the warmup parameters, as discussed next.

Adjusting warmup parameters

Ideally, at least two-thirds of the data in the warmup graph should show steady-state behavior. If there is not enough data to make the steady state twice as long as the warmup time, or if you cannot detect an obvious warmup time, adjust the warmup parameters as follows:

1. If there is not enough data in steady state, increase the number of snaps.
2. If you think that the model has *not* reached steady state, increase the run length by increasing either the number of snaps, the snap length, or both.
3. If you suspect that the system *has* reached steady state but you are not sure, increase the number of replications to reduce the variability of the graph.

When adjusting warmup parameters, use the following guidelines:

Number of Snaps AutoStat becomes slower as the number of snaps increases, because it must write reports more often. Therefore, for faster analysis, use the fewest number of snaps you need. In many instances, you should not need more than 60 snaps.

Snap Length You can adjust the snap length in the graph using the averaging window value if you have enough snaps defined. For example, suppose that there are 50 snaps in each replication, and each snap is 2 hours long. When the averaging window value is 1, each data point represents 1 snap, or 2 hours. If you want the snap length to be 4 hours, set the averaging window value 2 to avoid additional simulation. Remember, though, that too few points can flatten the graph too much. In such cases, additional simulation is necessary.

Number of Replications As was true for confidence intervals, you can cut variability of warmup data in half by increasing the number of replications by a factor of 4 (see “Narrowing the confidence interval” on page 8.14 of the “Basic Statistical Analysis Using AutoStat” chapter).

The remainder of this chapter uses example models to demonstrate how to apply these guidelines when analyzing the warmup time for different types of systems.

Determining warmup times for systems with classic warmup behavior

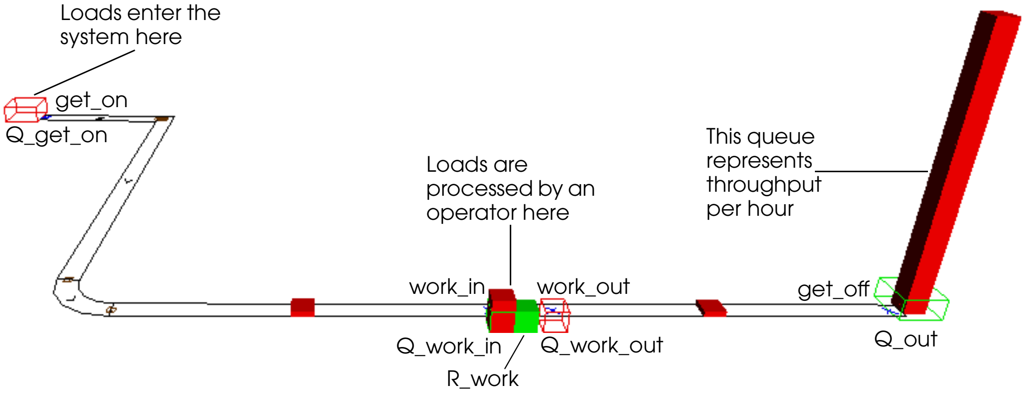
Simulation textbooks often show warmup graphs that have a warmup trend that is easy to detect. The first example model demonstrates such a “classic” warmup graph and demonstrates how to determine a warmup time for a system with classic warmup behavior.

Example 15.1: Classic warmup behavior

In example model 15.1, loads have an interarrival rate that is exponentially distributed with a mean of one minute. Loads travel to a processing center, where they are processed by one of eight workers for a time that is uniformly distributed between 0 and 12 minutes. The loads are then sent to an output queue, where they wait for one hour, then leave the system. The average number of loads in the output queue equals throughput per hour.

Step 1 ***Import*** a copy of the *base* version of example model 15.1.

The loads travel through the system using the conveyor shown below:


Loads enter the system here
get_on
Q_get_on
Loads are processed by an operator here
This queue represents throughput per hour
work_in
work_out
get_off
Q_work_in
Q_work_out
Q_out
R_work

Example model 15.1: layout

The logic for the model is shown below:

```
begin P_init arriving
   while 1=1 do
      begin
         clone 1 load to P_process
         wait for e 1 min /* interarrival time */
      end
end

begin P_process arriving
   move into Q_get_on
   move into conv:get_on
   travel to conv:work_in
   move into Q_work_in
   use R_work for u 6,6 min
   move into Q_work_out
   move into conv:work_out
   travel to conv:get_off
   move into Q_out
   wait for 1 hr
   send to die
end
```

Step 2 From the Model menu, ***select*** Run AutoStat, then ***click*** Yes to build the model. The AutoStat Setup wizard opens.

Step 3 ***Set up*** the model using the following parameters:

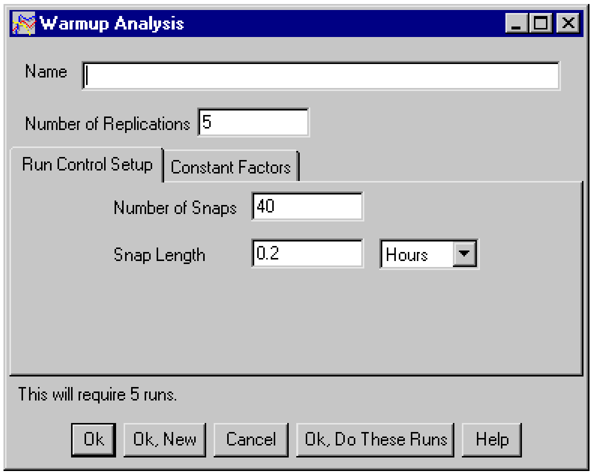
- Model is random
- Do not check for infinite loops
- Do not define a warmup (you will conduct an analysis to determine the warmup length)
- Snap length is 2 hours (a load finishes in less than 2 hours)

Defining a warmup analysis

To define a warmup analysis:

Step 1 From the Create New Analysis of Type drop-down list, ***select*** Warmup.

Step 2 ***Click*** New. The Warmup Analysis window opens.



Name

Number of Replications 5

Number of Snaps 40

Snap Length 0.2 Hours

This will require 5 runs.

Warmup analysis window

Step 3 ***Name*** the analysis “Example 15.1 warmup”.

By default, the number of replications is 5 and the number of snaps is 40, which is adequate for this analysis. You need to make the snap length longer, however.

Step 4 ***Change*** the Snap Length from 0.2 hours to 2 hours.

Step 5 ***Click*** OK to close the Warmup Analysis window.

Before making runs, you must edit the seed increment value to ensure that there are enough unique random numbers.

Changing the seed increment

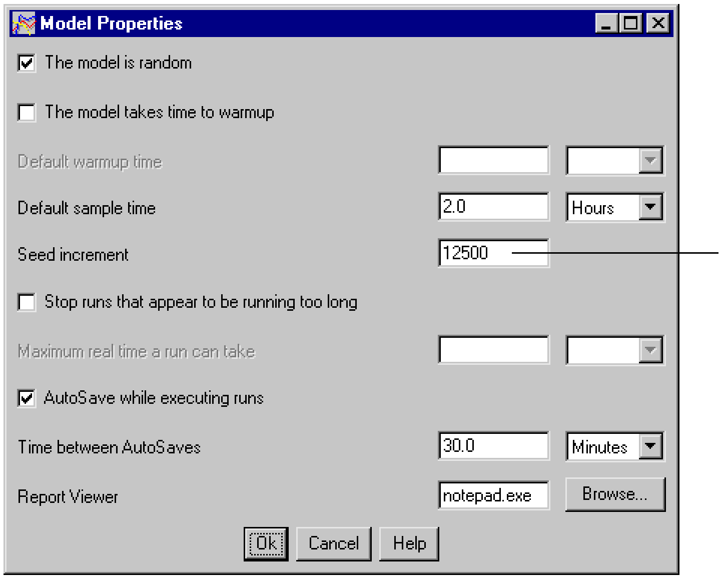
As discussed in “Preventing statistical inaccuracy when using random numbers” on page 15.6, when using AutoStat for any kind of analysis, you must determine how many random numbers your model is using and adjust the seed increment accordingly to avoid correlation between replications.

The report file for this model indicates that for an 80 hour run, 9,567 random numbers are used, which is too close to the default seed increment value of 10,000 to allow for variations as you experiment with the model. Therefore, you need to make the increment value larger.

To set the seed increment for the model to 12,500:

Step 1 From the Properties menu, ***select*** Edit Model Properties. The Model Properties window opens.

Step 2 ***Change*** the Seed Increment to “12500,” as shown below:



Model Properties

☑ The model is random

□ The model takes time to warmup

Default warmup time

Default sample time 2.0 Hours

Seed increment 12500

□ Stop runs that appear to be running too long

Maximum real time a run can take

☑ AutoSave while executing runs

Time between AutoSaves 30.0 Minutes

Report Viewer notepad.exe

Edit the seed increment value to 12500

Changing the seed increment value for AutoStat

Step 3 ***Click*** OK.

Step 4 From the Execution menu, ***select*** Do All Runs. While AutoStat is making runs, you can define the responses you will use to analyze the model’s warmup behavior.

Defining responses

The responses you need to determine the warmup time are the average number of loads in the system (average for the process P_Process), the average number of loads in the queue Q_out (throughput per hour), and the utilization of the workers.

To define the responses:

Step 1 *Click* the Responses tab.

Step 2 *Click* New to create a new response of type AutoMod Response. The AutoMod Response window opens.

Step 3 *Name* the response “Average in system”.

Step 4 *Select* the system “Proc,” the process “P_Process,” and the statistic “Ave.”

Step 5 *Click* OK, New to define the next response (the average number of loads in Q_out):

Step 6 *Name* the response “Q_out average”.

Step 7 *Select* the system “Proc,” the queue “Q_out,” and the statistic “Ave.”

Step 8 *Click* OK, New to define the next response (worker utilization):

Step 9 *Name* the response “Worker utilization”.

Step 10 *Select* the system “Proc,” the resource “R_work,” and the statistic “Util.”

Step 11 *Click* OK.

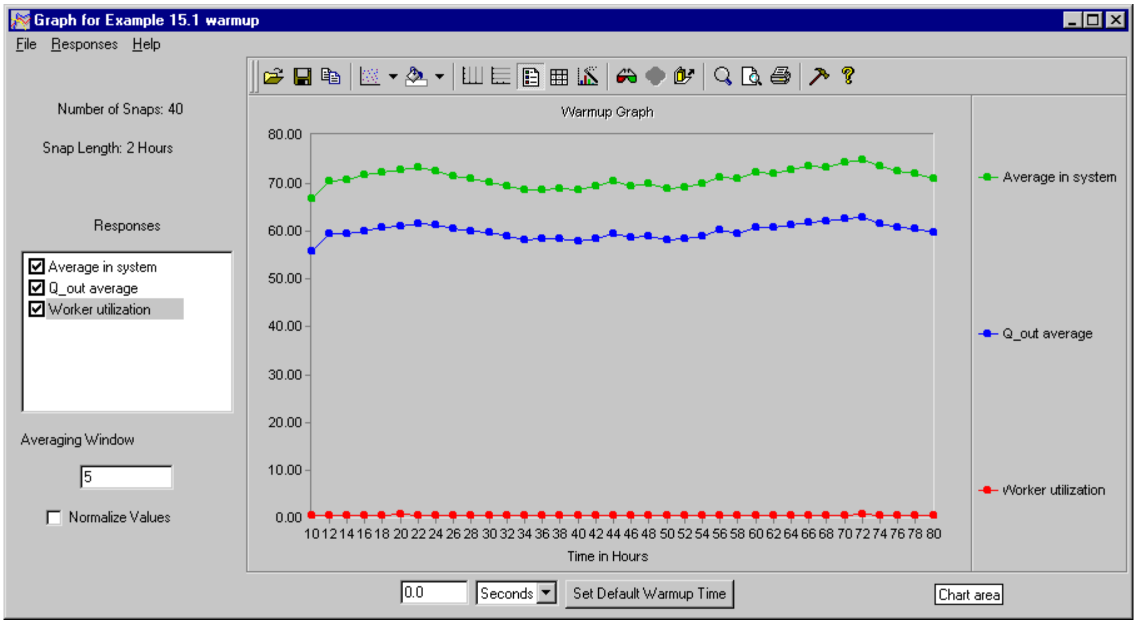
Now you can view the warmup graph (even if the runs have not finished).

Viewing the warmup graph

To view the warmup graph for this analysis:

Step 1 *Click* the Analysis tab.

Step 2 *Expand* the analysis for “Example 15.1 warmup” and ***double-click*** Warmup Graph. The warmup graph is displayed.



Graph for Example 15.1 warmup

Number of Snaps: 40

Snap Length: 2 Hours

Responses

☑ Average in system
☑ Q_out average
☑ Worker utilization

Averaging Window: 5

☐ Normalize Values

0.0 Seconds

Example model 15.1 initial warmup graph

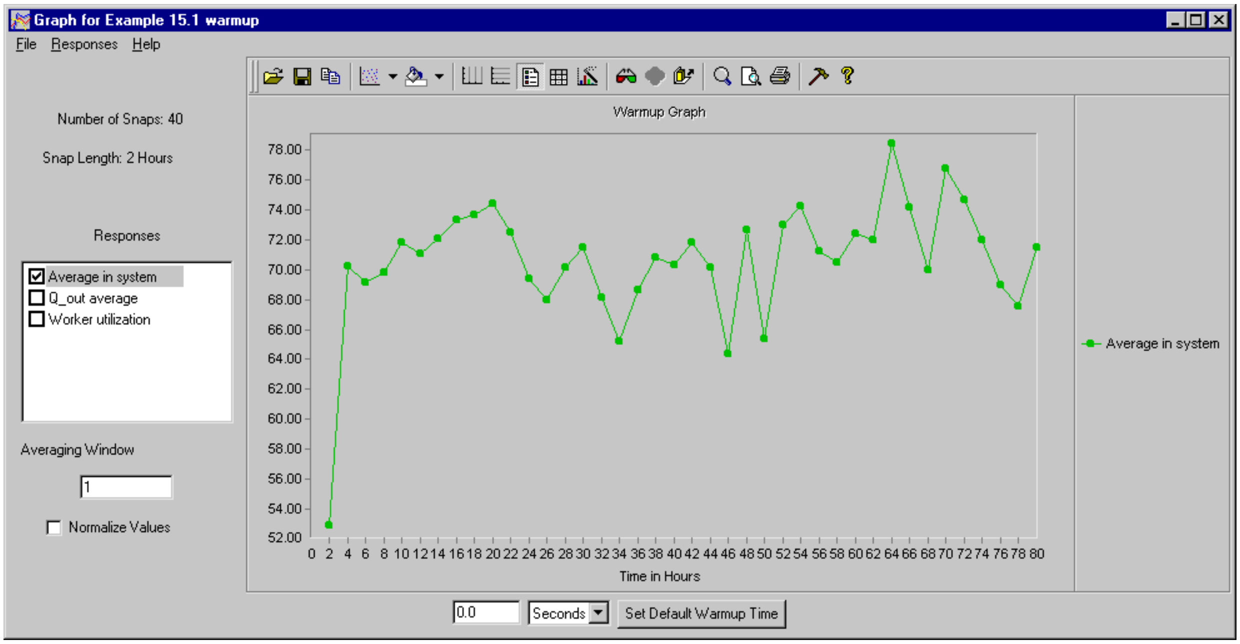
Determining when the response "Average in system" warms up

To begin analyzing the responses to determine steady state, display only the "Average in system" response:

Step 1 ***Clear*** the check boxes for Q_out average and Worker utilization.

Now, look at the "raw" data by setting the averaging window value to 1:

Step 2 ***Change*** the averaging window value from "5" to "1," then ***press*** Tab. The graph is updated as shown below:



Graph for Example 15.1 warmup

Number of Snaps: 40

Snap Length: 2 Hours

Responses

☑ Average in system
□ Q_out average
□ Worker utilization

Averaging Window: 1

□ Normalize Values

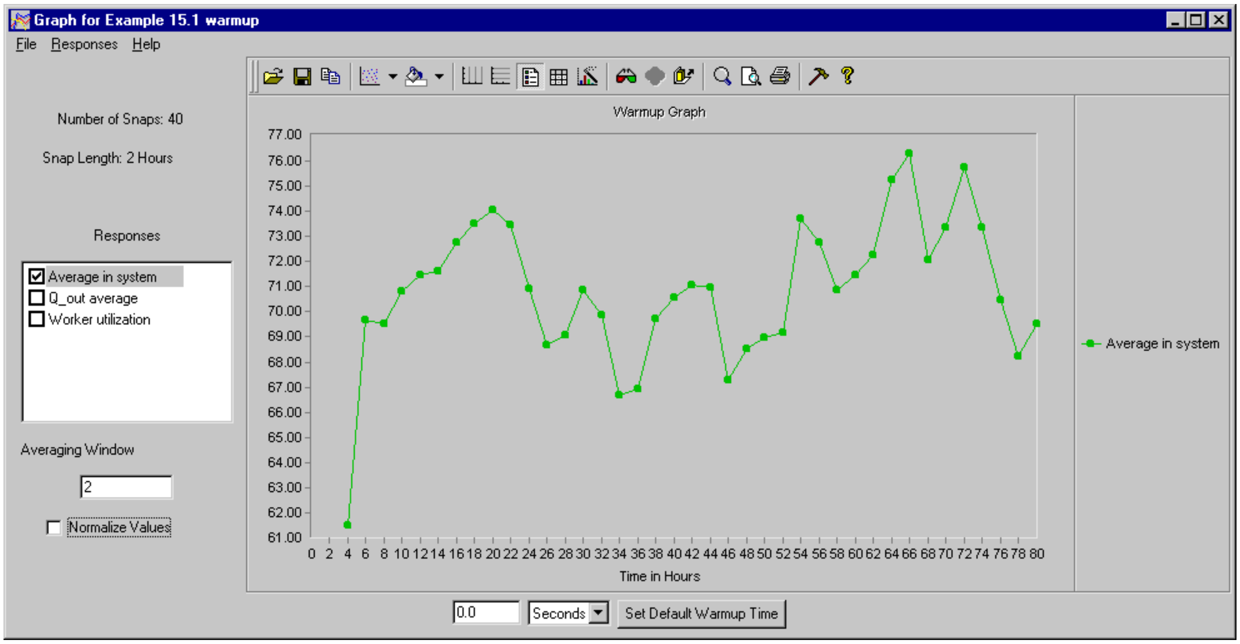
0.0 Seconds

Average in system with an averaging window of 1

When the averaging window is 1, each data point represents the average number of loads in the system for a two hour period of time (the snap length). The first data point is the average number of loads in system for the first 2 hours. The second data point is the average number of loads in the system between 2 hours and 4 hours, and so on.

The first response value is between 52 and 54. Starting with the second point, the values vary from approximately 64 to 79. Based on this graph, it seems that after the first two hours, the response value fluctuates within a stable range of values, indicating that the response warms up after two hours. However, there is quite a lot of variation, so further examination is necessary.

Step 3 ***Change*** the averaging window value from “1” to “2,” then ***press*** Tab. The graph is updated as shown below:



Graph for Example 15.1 warmup

Number of Snaps: 40

Snap Length: 2 Hours

Responses

☑ Average in system
□ Q_out average
□ Worker utilization

Averaging Window: 2

□ Normalize Values

0.0 Seconds

Average in system with an averaging window of 2 (Y axis scale has changed)

Each data point is now the average of two snaps. The first point is now drawn at time 4 hours, because it is the average of the first two 2-hour snaps (0-2 hours and 2-4 hours). The second data point is the average of the second and third snaps (2-4 hours and 4-6 hours).

Notice, however, the Y axis scale has changed from the previous graph’s scale of 52-79. The change in scale makes it difficult to compare the two graphs. Therefore, adjust this graph’s scale, as described next.

Changing the Y axis scale

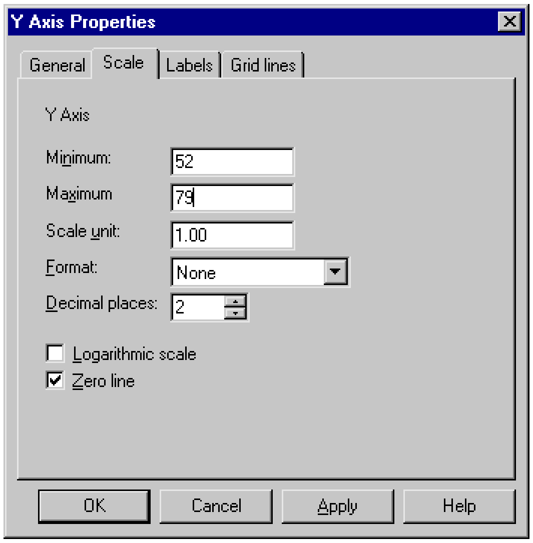
Whenever you change the averaging window, AutoStat automatically scales the graph based on the new high and low values. However, for warmup determination, in order to detect a trend, all graphs need to use the same scale.

To change the Y axis scale:

Step 1 ***Double-click*** a value on the left (Y) axis. The Y Axis Properties window opens.

Step 2 ***Click*** the Scale tab.

Step 3 ***Define*** the Minimum as 52 and Maximum as 79, as shown below:



Y Axis

Minimum: 52

Maximum 79

Scale unit: 1.00

Format: None

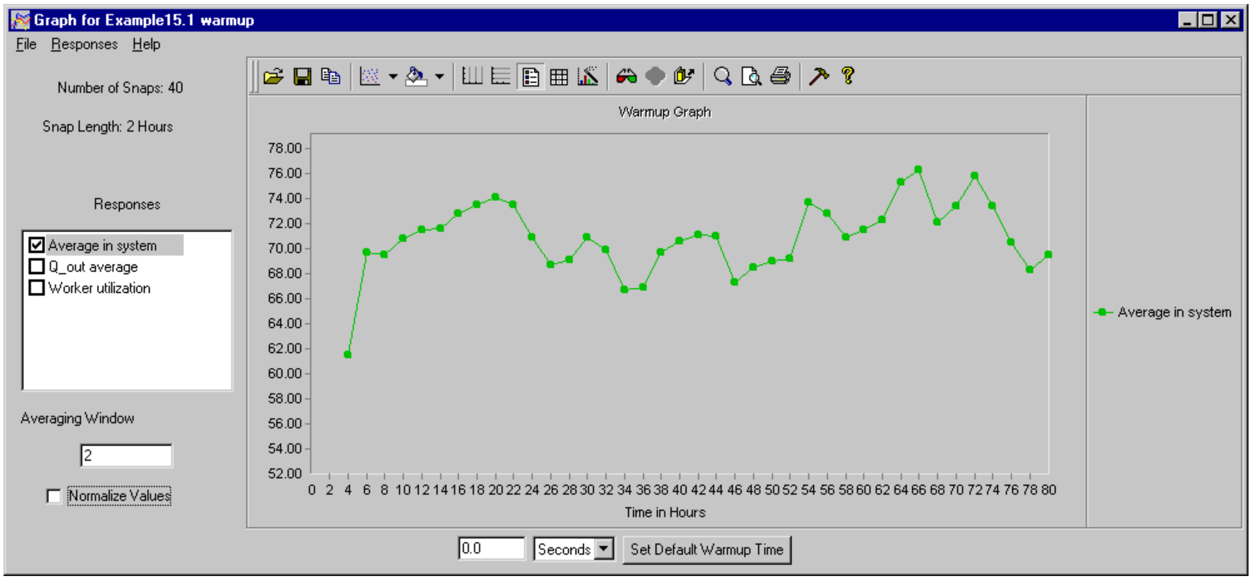
Decimal places: 2

☐ Logarithmic scale

☑ Zero line

Changing the Y axis scale values

Step 4 ***Click*** OK. The graph is updated as shown below:



Number of Snaps: 40

Snap Length: 2 Hours

Responses

☑ Average in system

☐ Q_out average

☐ Worker utilization

Averaging Window

2

☐ Normalize Values

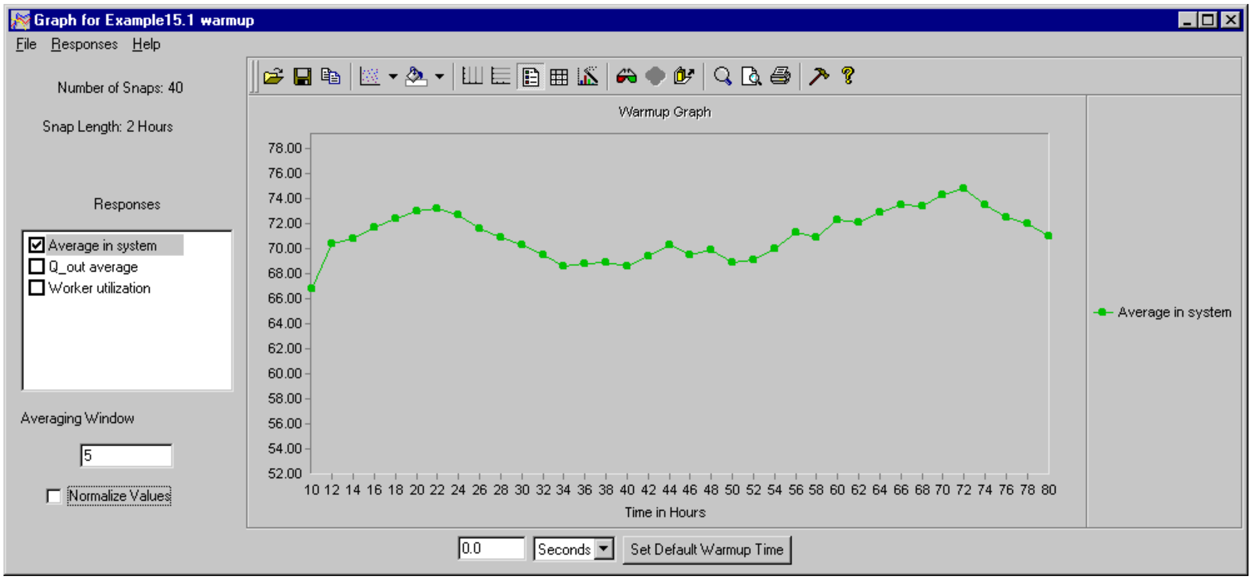
0.0 Seconds

Average in system with an averaging window of 2 (Y axis scale adjusted)

When this graph is compared to the original graph with an averaging window of 1, you can see that this graph is smoother. This graph makes it easier to see an initial rise in the response value and a steady variation in the response for the remainder of the graph.

Step 5 ***Change*** the averaging window value from “2” to “5,” then ***press*** Tab.

Step 6 ***Adjust*** the Y scale to range from 52 to 79. The graph is updated as shown below:



Number of Snaps: 40

Snap Length: 2 Hours

Responses

☑ Average in system
☐ Q_out average
☐ Worker utilization

Averaging Window: 5

☐ Normalize Values

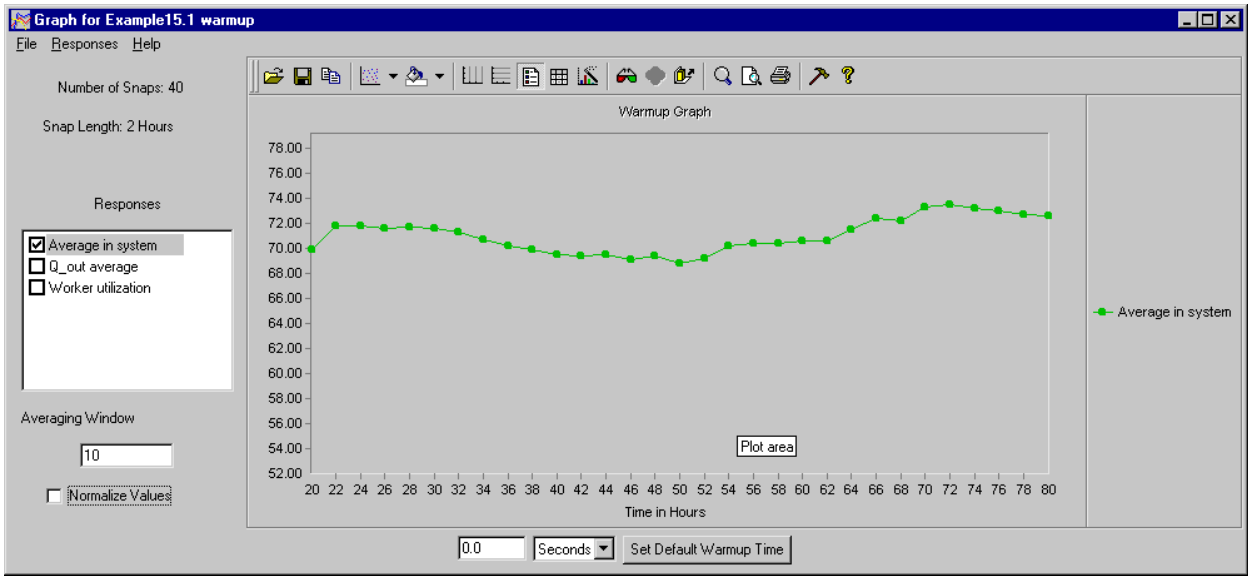
0.0

Average in system with an averaging window of 5 (Y axis scale adjusted)

The graph is getting smoother.

Step 7 ***Change*** the averaging window value from “5” to “10,” then ***press*** Tab.

Step 8 ***Adjust*** the Y scale to range from 52 to 79. The graph is updated as shown below:



Number of Snaps: 40

Snap Length: 2 Hours

Responses

☑ Average in system
☐ Q_out average
☐ Worker utilization

Averaging Window: 10

☐ Normalize Values

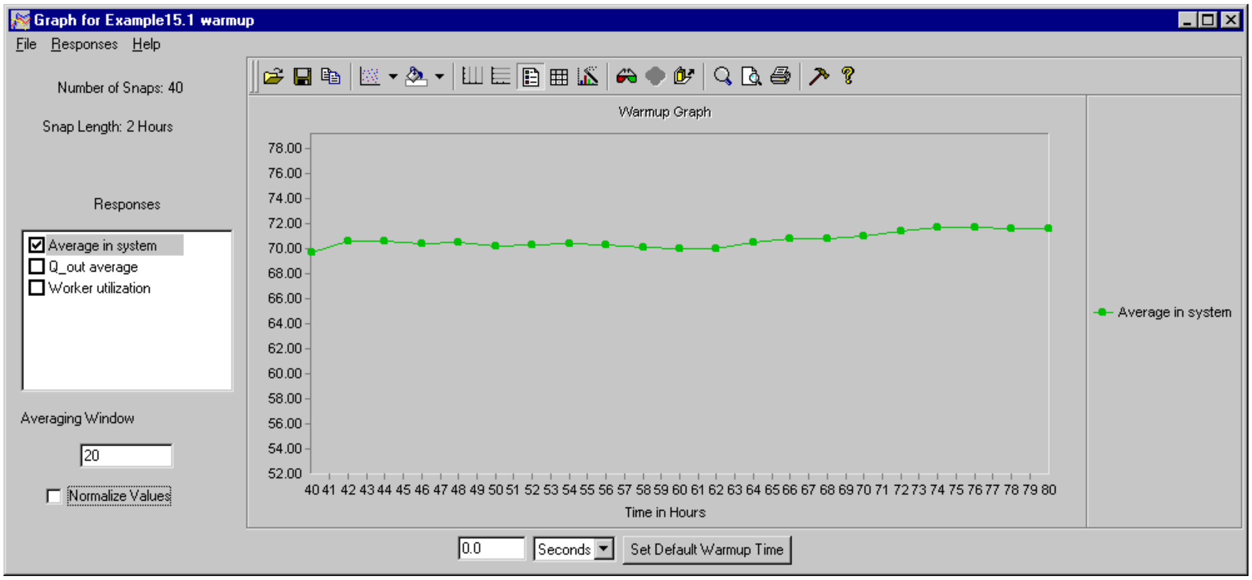
0.0

Average in system with an averaging window of 10 (Y axis scale adjusted)

The graph is getting very flat. It is hard to detect any variation.

Step 9 ***Change*** the averaging window value from “10” to “20,” then ***press*** Tab.

Step 10 ***Adjust*** the Y scale to range from 52 to 79. The graph is updated as shown below:



Average in system with an averaging window of 20 (Y axis scale adjusted)

When the averaging window is 20, the graph is almost a flat line. All detail has been lost. You can see that while small adjustments to the averaging window can help the analysis, adjusting the value too much (in this case, windows of 10 and 20) eliminates any trend at all.

Use the graph with an averaging window of 2 to determine the warmup time for this model (see the graph on page 15.15). As discussed earlier, the warmup time appears to be 2 hours. After the initial rise in the response value, the values fluctuate steadily between 66 and 76.

To determine the warmup time based on the windowing average and snap length currently being used in this graph, use the following formula:

$$\text{Warmup time} = \text{Steady state time} - w \times \text{Snap length}$$

where “Steady state time” is the time at which you believe steady state begins and “w” is the averaging window value currently in use in the graph. Using the window of 2, steady state occurs at time 6 hours (the time for the second data point). The averaging window is 2 and the snap length is also 2. So:

$$\text{Warmup time} = 6 - 2 \times 2$$

or

$$\text{Warmup time} = 2$$

Therefore, the warmup for this response is 2 hours. When conducting a warmup analysis, you must use all available responses, such as WIP levels at various points in the system, to ensure that you are getting the entire picture. One or even two responses are not usually enough in real-world models.

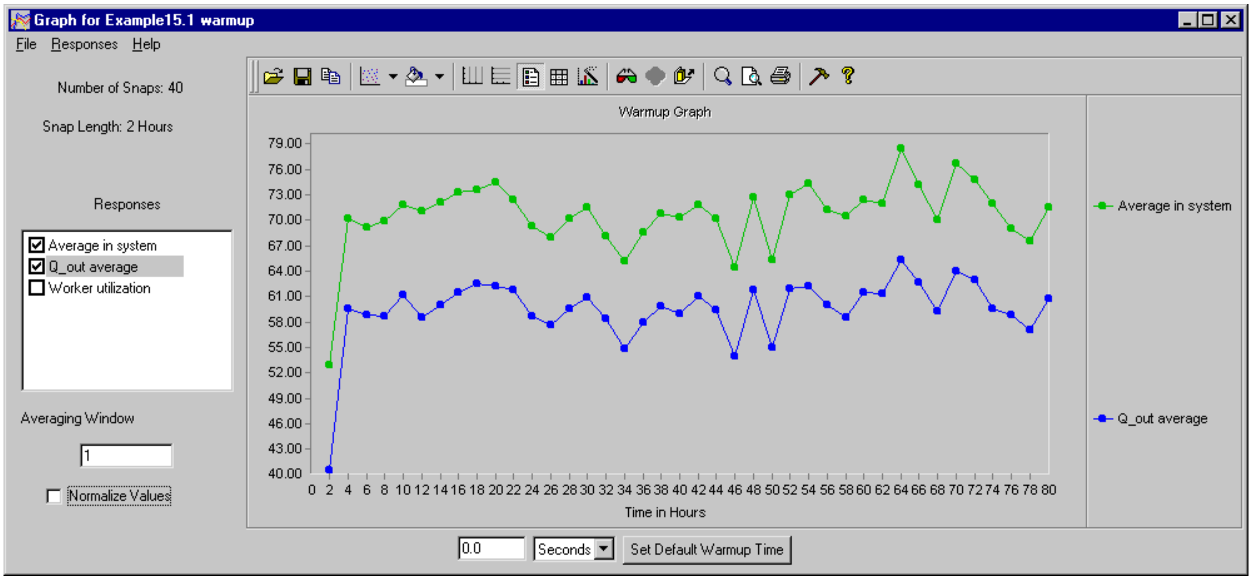
Therefore, look at the remaining two responses to confirm or refine the 2-hour warmup estimate for this model.

Determining when the remaining responses warm up

When performing warmup determination, whatever is the longest time for any of the responses to warm up is the warmup time you should use for the model. After analyzing the first response, Average in system, the model appears to warm up in 2 hours. Now look at the Q_out average response to see whether the 2 hour estimate changes.

Step 1 ***Select*** the “Q_out average” response (the “Average in system” response is already selected).

Step 2 ***Change*** the averaging window value from “20” to “1,” then ***press*** Tab. The graph is updated as shown below:



Number of Snaps: 40

Snap Length: 2 Hours

Responses

☑ Average in system
☑ Q_out average
☐ Worker utilization

Averaging Window: 1

☐ Normalize Values

0.0 Seconds

Q_out average with an averaging window of 1

The two responses are almost parallel. Therefore, you can conclude that the warmup time for the response “Q_out average” is also 2 hours.

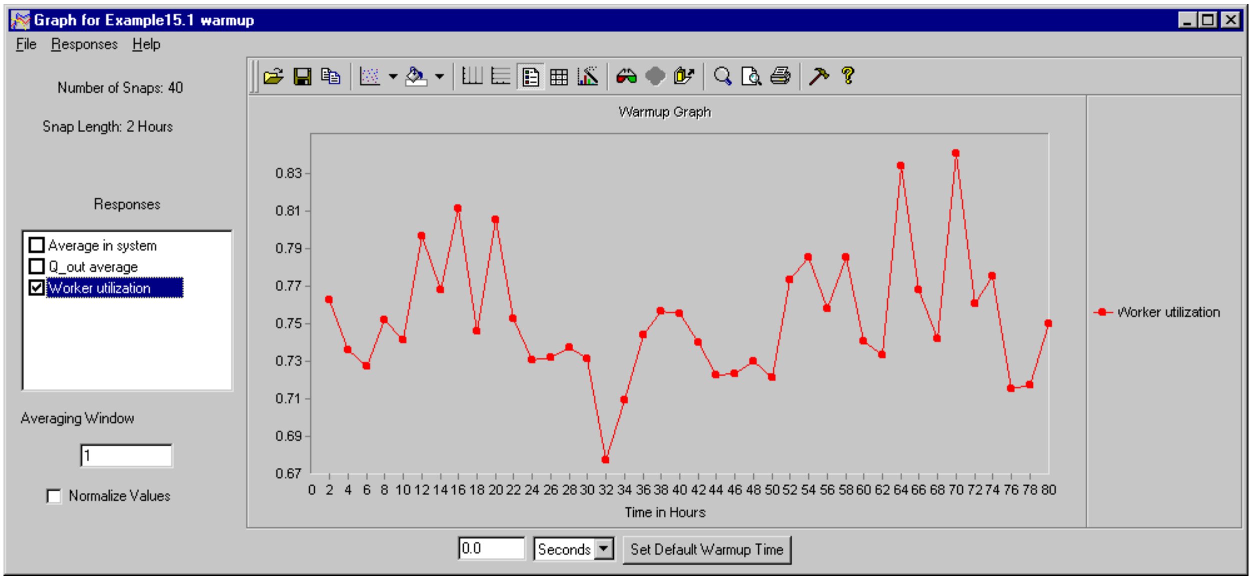
There is one more response to check for the warmup analysis: Worker utilization.

Note Utilization of people and equipment should not be used by itself to determine warmup time, as it almost always reaches steady state. Utilization often reaches steady-state more quickly than WIP levels, too, which may cause you to think the warmup time is shorter than it truly is. However, utilization is a useful response for confirming information gathered through other responses, such as WIP levels.

To view the “Worker utilization” response:

Step 3 ***Deselect*** the first two responses in the Responses list.

Step 4 ***Select*** the “Worker utilization” response. The graph is updated as shown below:



Number of Snaps: 40

Snap Length: 2 Hours

Responses

- ☐ Average in system
- ☐ Q_out average
- ☑ Worker utilization

Averaging Window: 1

☐ Normalize Values

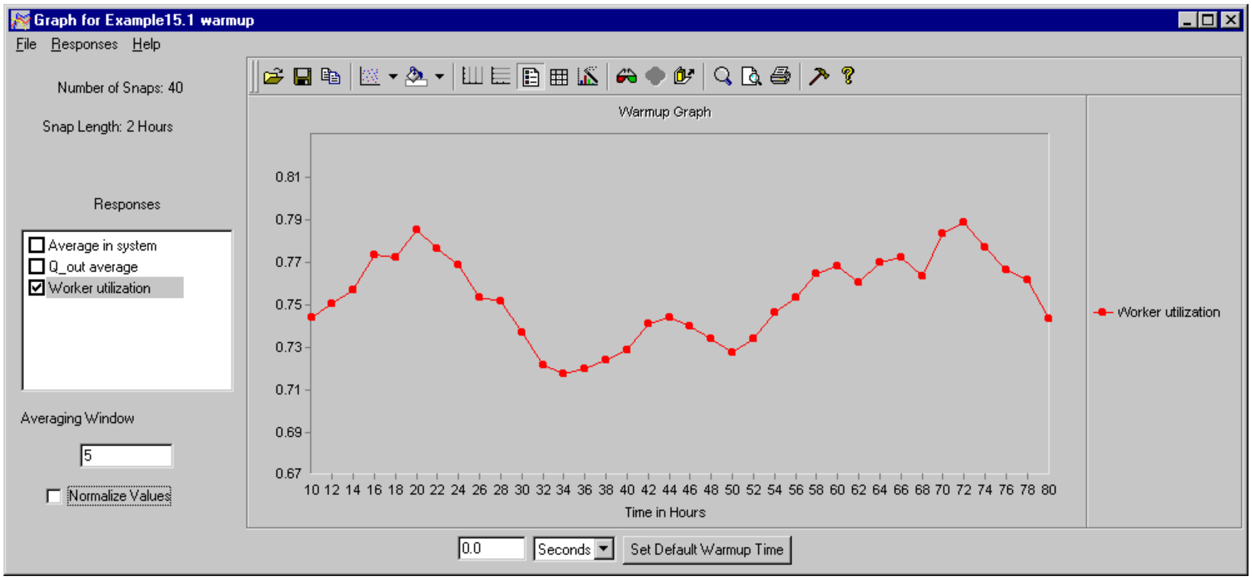
0.0 Seconds

Worker utilization with an averaging window of 1

The worker utilization varies between 0.67 and 0.83. There is so much variation that it is hard to tell whether the response ever reaches a steady state.

Step 5 ***Change*** the averaging window value from “1” to “5,” then ***press*** Tab.

Step 6 ***Adjust*** the Y scale to range from .67 to .83. The graph is updated as shown below:



Number of Snaps: 40

Snap Length: 2 Hours

Responses

- ☐ Average in system
- ☐ Q_out average
- ☑ Worker utilization

Averaging Window: 5

☐ Normalize Values

0.0 Seconds

Worker utilization with an averaging window of 5 (Y axis scale adjusted)

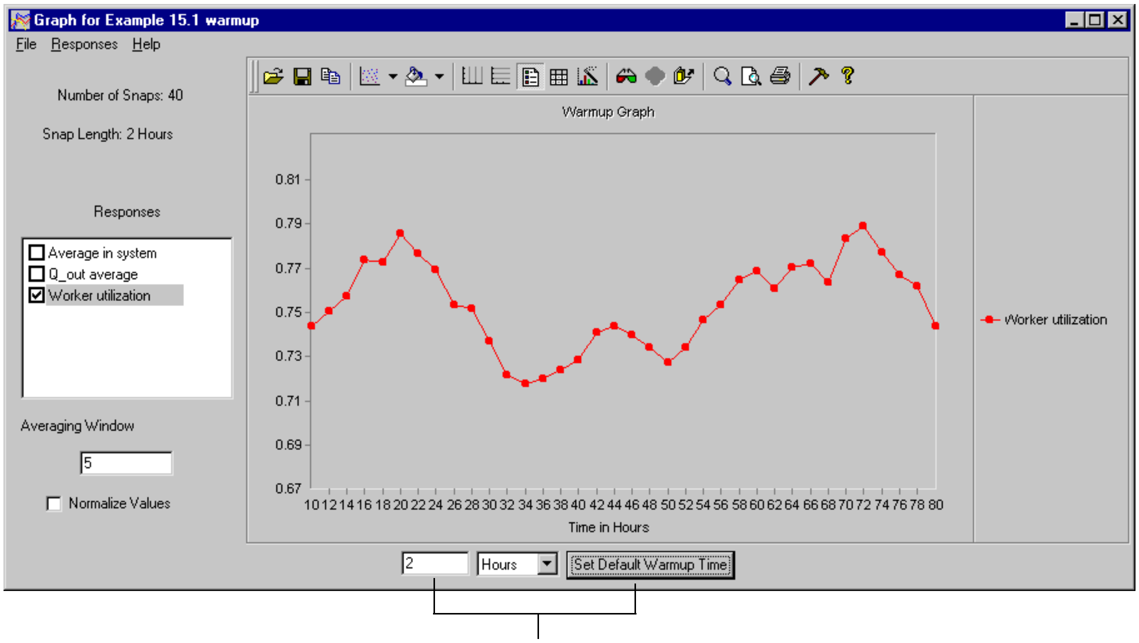
This graph shows that the variation is predictable, indicating that resource utilization starts in steady state and continues that way. Therefore, based on the analysis of the three responses, the warmup time for the model is 2 hours.

Setting the model's default warmup time

Once you have determined the warmup time for a model, set the default warmup time to that value, so that all of your analyses use the correct warmup time.

To set the warmup time for the model to 2 hours:

Step 1 ***Enter*** the warmup time as 2 hours in the bottom of the warmup graph, as shown below:



Graph for Example 15.1 warmup

Number of Snaps: 40

Snap Length: 2 Hours

Responses

☐ Average in system
☐ Q_out average
☑ Worker utilization

Averaging Window: 5

☐ Normalize Values

2 Hours Set Default Warmup Time

Set the default
warmup time here

Setting the default warmup time from the warmup graph

Step 2 ***Click*** Set Default Warmup Time. The default in the Model Properties window is updated. Once the time is set in the model properties, all subsequent analyses will use the specified warmup time. Statistics gathered during the initial transient warmup phase are discarded, and all of the summary statistics and other reports show only the statistics gathered during steady state.

You have successfully determined the warmup time for a system that exhibits "classic" warmup behavior. Now you will move on to systems for which warmup determination requires more analysis.

Determining that a system is explosive

Systems that are explosive never reach steady state. Loads continually back up in one or more locations. You may not realize that your model has a problem like this until you start analyzing it in AutoStat. Therefore, it is important to recognize whether response values are indicating that steady state has been reached or whether loads are backing up.

In order to conclude that a model has warmed up to a steady state, you must review all responses, including WIP levels at various queues throughout the system. Unless all responses level off, the model has not reached steady state.

Models that are explosive must be made stable before you conduct statistical analyses. No real-world system is explosive, and therefore an explosive model is not accurate. *You should not perform any analyses using an explosive model until you have made it stable.*

Example model 15.2: Explosive warmup behavior

Example model 15.2 is the same process as example model 15.1. However, the time that the workers take to process a load has been changed from:

```
use R_work for u 6,6 min
```

to:

```
use R_work for u 9,9 min
```

As illustrated in “Queueing theory” on page 1.30 of the “Principles of Simulation” chapter, resource utilization for multiple operators is calculated as:

$$\rho = \frac{\lambda}{c\mu}$$

where:

ρ = server utilization

λ = arrival rate

c = number of servers

μ = service rate

In this model, loads arrive for service at a rate of 1 per minute. There are 8 workers that can each service a load in a time that is uniformly distributed with a mean of 9 minutes, for a service rate of 1/9 per minute. Therefore:

ρ = 1/[8(1/9)] = 9/8 = 1.125

In this example model, operator utilization is greater than 1, which indicates that the system is explosive. The model’s explosive behavior will become apparent as you conduct the warmup analysis.

Step 1 ***Import*** a copy of the *base* version of example model 15.2 in AutoMod.

Step 2 From the Model menu, ***select*** Run AutoStat, then ***click*** Yes to build the model. The AutoStat Setup wizard opens.

Step 3 ***Set up*** the model using the following parameters:

- Model is random
- Do not check for infinite loops
- Do not define a warmup (you will conduct an analysis to determine the warmup length)
- Snap length is 2 hours

Defining the warmup analysis

Example model 15.2 uses the same warmup analysis information that you used for example model 15.1.

Step 1 ***Define*** a warmup analysis for example model 15.2 with a snap length of 2 hours (use the default of 5 replications and 40 snaps).

Step 2 ***Set*** the seed increment value to 12,500.

Step 3 ***Define*** the same responses as you did for example model 15.1 (see “Defining responses” on page 15.12):

- Average in system
- Q_out average
- Worker utilization

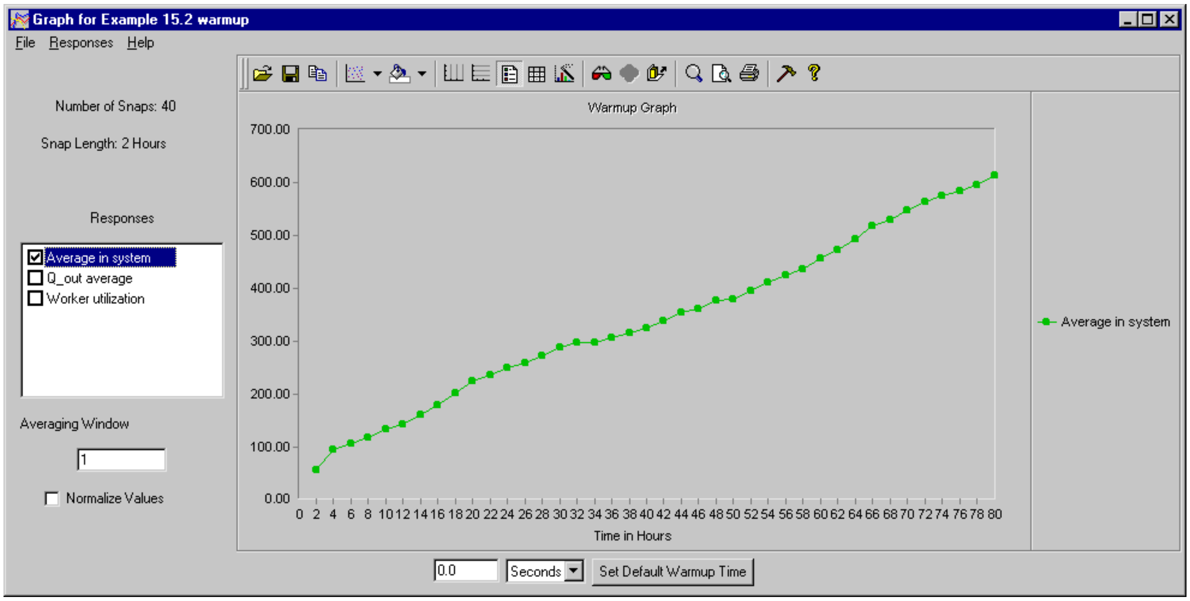
Step 4 ***Make*** all the runs.

Analyzing the warmup graph for example model 15.2

Once responses have been defined, you can begin analyzing the warmup for this model:

Step 1 ***Display*** the warmup graph for the response “Average in system.”

Step 2 ***Change*** the averaging window value from “5” to “1,” then ***press*** Tab. The graph is updated as shown below:



Number of Snaps: 40

Snap Length: 2 Hours

Responses

☑ Average in system
☐ Q_out average
☐ Worker utilization

Averaging Window: 1

☐ Normalize Values

0.0

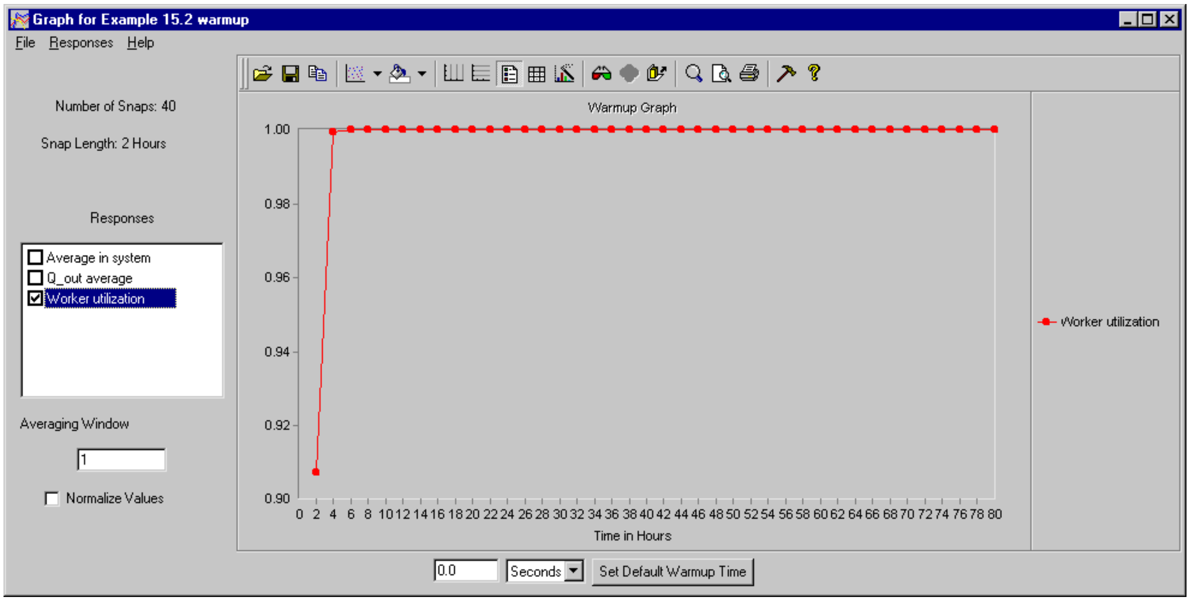
Average in system with an averaging window of 1

Immediately, you can see that the response “Average in system” is growing steadily. There is no leveling off after 80 hours of simulation.

Now check the utilization response:

Step 3 ***Deselect*** the “Average in system” response.

Step 4 ***Select*** “Worker utilization.” The graph is updated as shown below:



Number of Snaps: 40

Snap Length: 2 Hours

Responses

☐ Average in system
☐ Q_out average
☑ Worker utilization

Averaging Window: 1

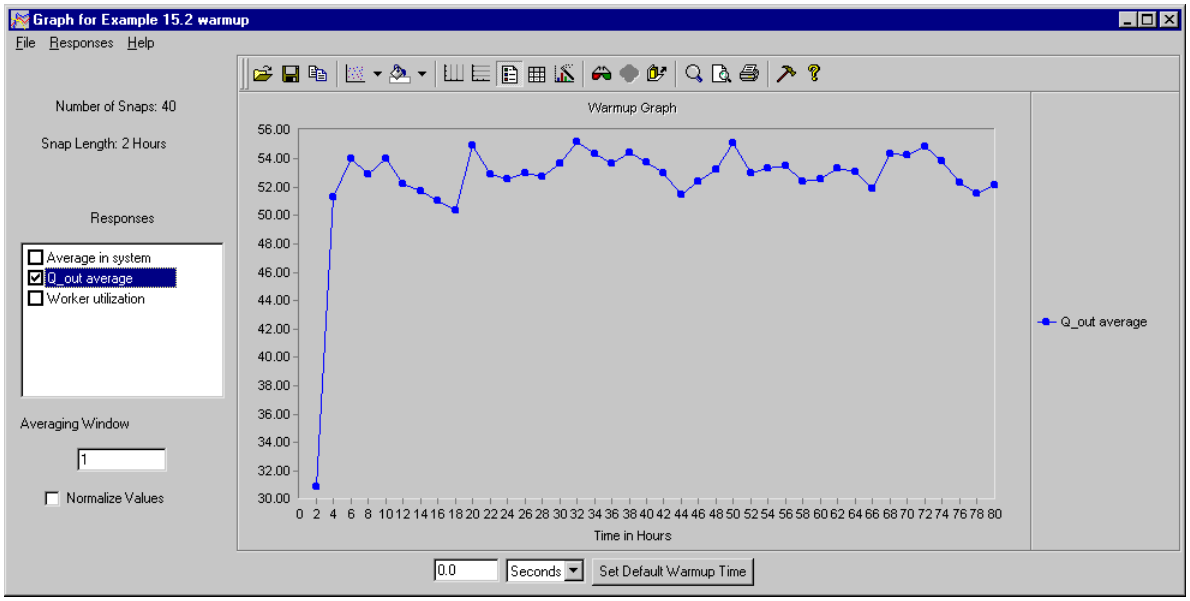
☐ Normalize Values

0.0 Seconds

Worker utilization with an averaging window of 1

The utilization rapidly grows to 1.0, the highest utilization possible, and remains there for the entire simulation. This response is confirming that the system is explosive. Now check the final response.

Step 5 ***Deselect*** “Worker utilization” and ***select*** “Q_out average.” The graph is updated as shown below:



Number of Snaps: 40

Snap Length: 2 Hours

Responses

☐ Average in system
☑ Q_out average
☐ Worker utilization

Averaging Window: 1

☐ Normalize Values

0.0 Seconds

Q_out average with an averaging window of 1

The Q_out average response value seems to be more steady. However, because not all of the responses level off, the system does not have a steady state. The model would need to be examined and corrected before further analysis could be conducted.

Determining warmup times for cyclical systems

Some systems have cyclical behavior, such as cyclical down times, grouped start times for lots, and so on. These systems have regular fluctuations that, with analysis, can be detected and understood in a warmup analysis.

Example model 15.3: Cyclical warmup behavior

Example model 15.3 is the same process as example model 15.1. However, a break has been defined for the workers, as shown below:

```
begin P_break arriving
   while 1=1 do
      begin
         wait for 200 min /* Time between breaks */
         take down R_work
         wait for 40 min /* Break time */
         bring up R_work
      end
end
```

The time between breaks and the length of the breaks are both constant.

To analyze this model with a cyclical break:

Step 1 ***Import*** a copy of the *base* version of example model 15.3 in AutoMod.

Step 2 From the Model menu, ***select*** Run AutoStat, then ***click*** Yes to build the model. The AutoStat Setup wizard opens.

Step 3 ***Set up*** the model using the following parameters:

- Model is random
- Do not check for infinite loops
- Do not define a warmup
- Snap length is 2 hours

Defining the warmup analysis

Example model 15.3 uses the same warmup analysis information that you used for example model 15.1.

Step 1 ***Define*** a warmup analysis for example model 15.3 with a snap length of 2 hours (use the default of 5 replications and 40 snaps).

Step 2 ***Set*** the seed increment value to 12,500.

Step 3 ***Define*** the same responses as you did for example model 15.1 (see “Defining responses” on page 15.12):

- Average in system
- Q_out average
- Worker utilization

Step 4 ***Make*** all the runs.

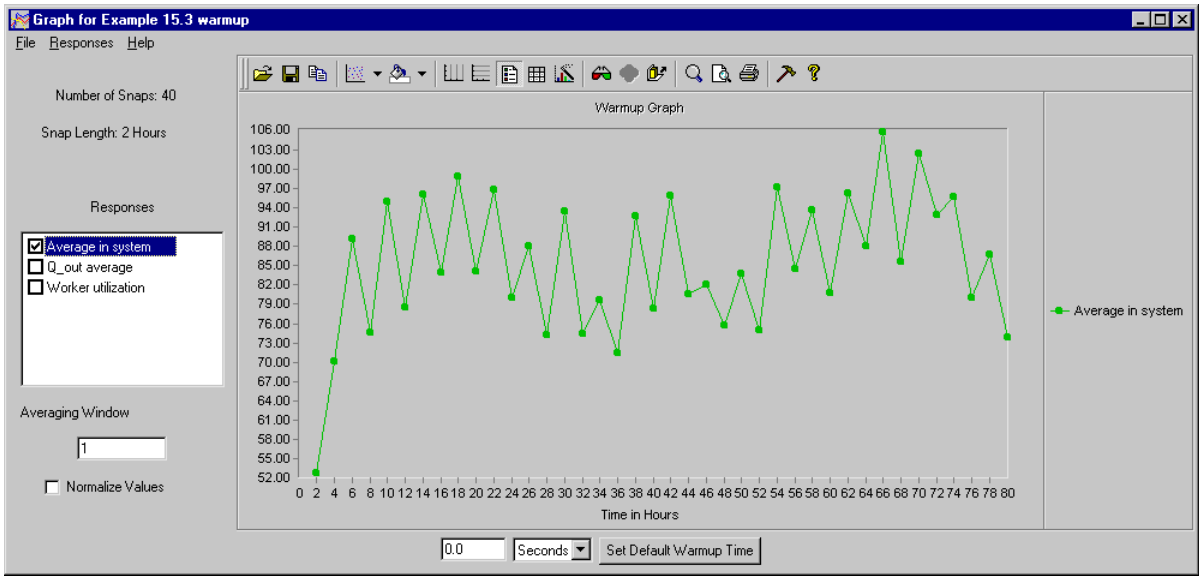
Analyzing the warmup graph for example model 15.3

Once the responses have been defined, you can begin analyzing the warmup for this model:

Step 1 ***Display*** the warmup graph for the response "Average in system."

Note The warmup analysis for this model and the next (example model 15.4) are based on the response "Average in system." Analyzing the "Q_out average" and "Worker utilization" responses for both examples are left as exercises for the chapter (See "Exercises" on page 15.36).

Step 2 ***Change*** the averaging window value from "5" to "1," then ***press*** Tab. The graph is updated as shown below:



Number of Snaps: 40

Snap Length: 2 Hours

Responses

☑ Average in system
☐ Q_out average
☐ Worker utilization

Averaging Window: 1

☐ Normalize Values

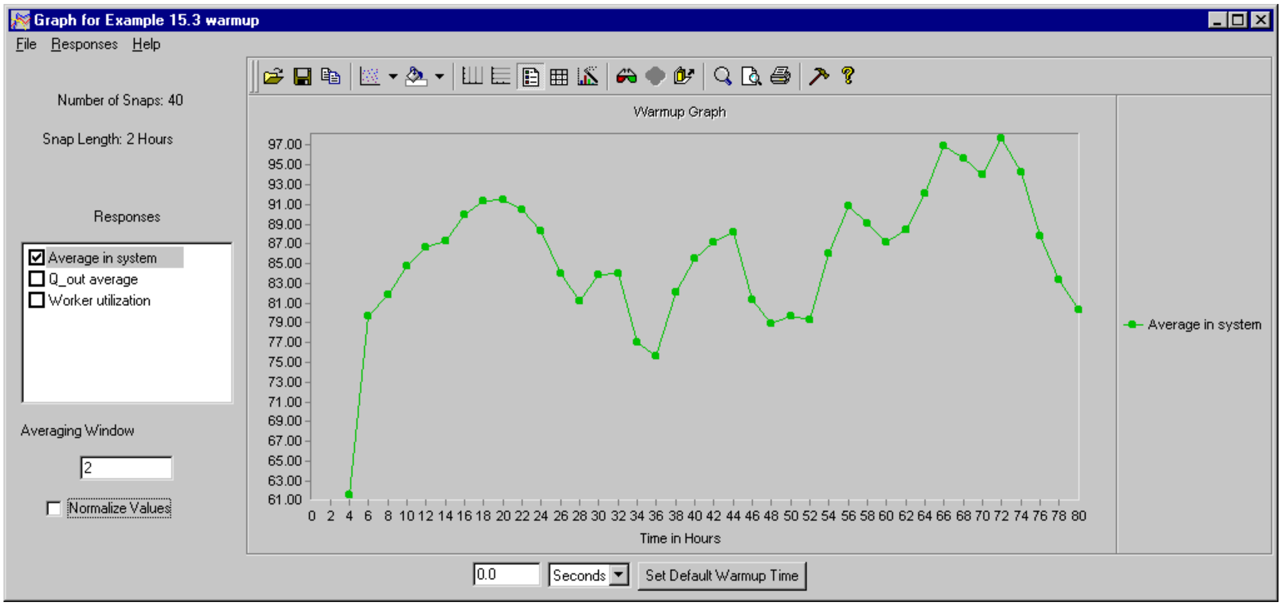
0.0 Seconds

Average in system with an averaging window of 1

The graph has pronounced peaks and valleys, which are caused by the time between the breaks and the break itself. The break interval spans a time of 4 hours, but the snap length is only 2 hours. Therefore, only every other snap is capturing the break. This violates one of the guidelines described in "Guidelines for setting warmup parameters" on page 15.7. Therefore, you need to adjust the snap length to be at least 4 hours so that each snap covers an entire break cycle.

Use the averaging window value to change the snap length to 4 hours:

Step 3 ***Change*** the averaging window value from “1” to “2,” then ***press*** Tab. The graph is updated as shown below:


Graph for Example 15.3 warmup

Number of Snaps: 40

Snap Length: 2 Hours

Responses

☑ Average in system
☐ Q_out average
☐ Worker utilization

Averaging Window: 2

☐ Normalize Values

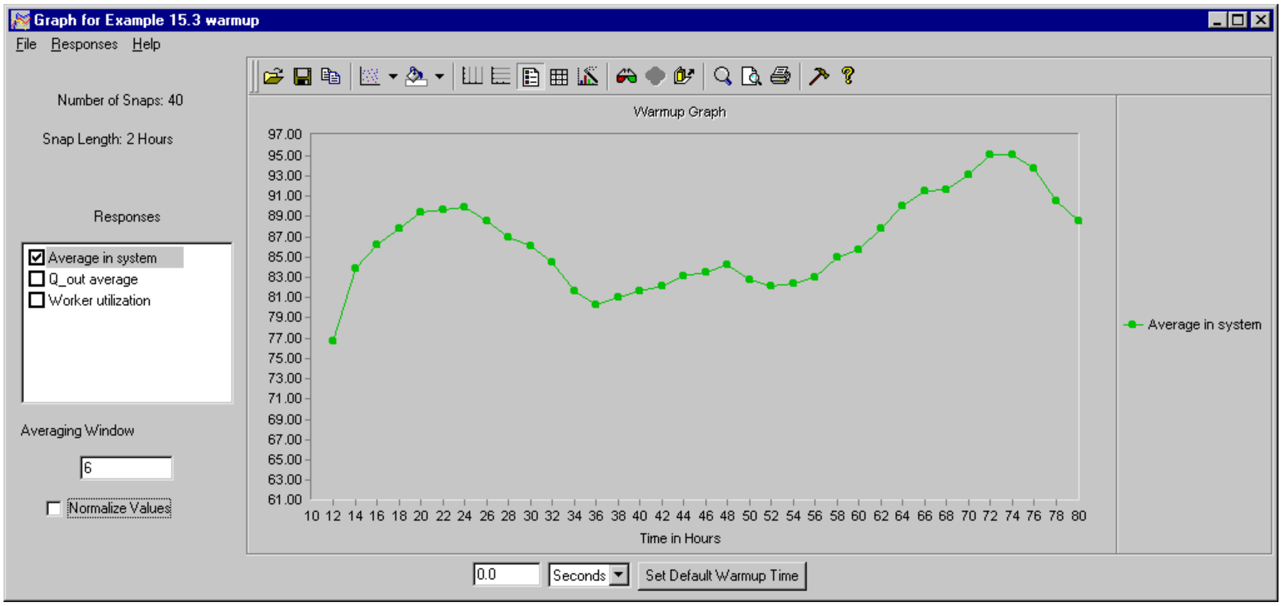
0.0 Seconds

Average in system with an averaging window of 2

The values vary from 61 to 97. The graph is a little smoother, but still has wide variation. The last peak is higher than the first, which indicates that the response has not leveled off.

Step 4 ***Change*** the averaging window value from “2” to “6,” then ***press*** Tab.

Step 5 ***Adjust*** the Y scale to range from 61 to 97. The graph is updated as shown below:


Graph for Example 15.3 warmup

Number of Snaps: 40

Snap Length: 2 Hours

Responses

☑ Average in system
☐ Q_out average
☐ Worker utilization

Averaging Window: 6

☐ Normalize Values

0.0 Seconds

Average in system with an averaging window of 6

There is a still an upward trend at the end of the graph. From the current information, you cannot make a determination.

Step 6 ***Close*** the graph.

Performing more replications

It is possible that the model has reached steady state, but you cannot be sure from the data available. As discussed in “Guidelines for setting warmup parameters” on page 15.7, when you are not confident that the model has reached steady state, you need to make more replications. Increasing the number of replications by a factor of 4 reduces the variability by approximately half. You have made 5 replications so far. Therefore, you need to make 15 additional replications.

Step 1 ***Edit*** the warmup analysis.

Step 2 ***Change*** the Number of Replications from 5 to 20.

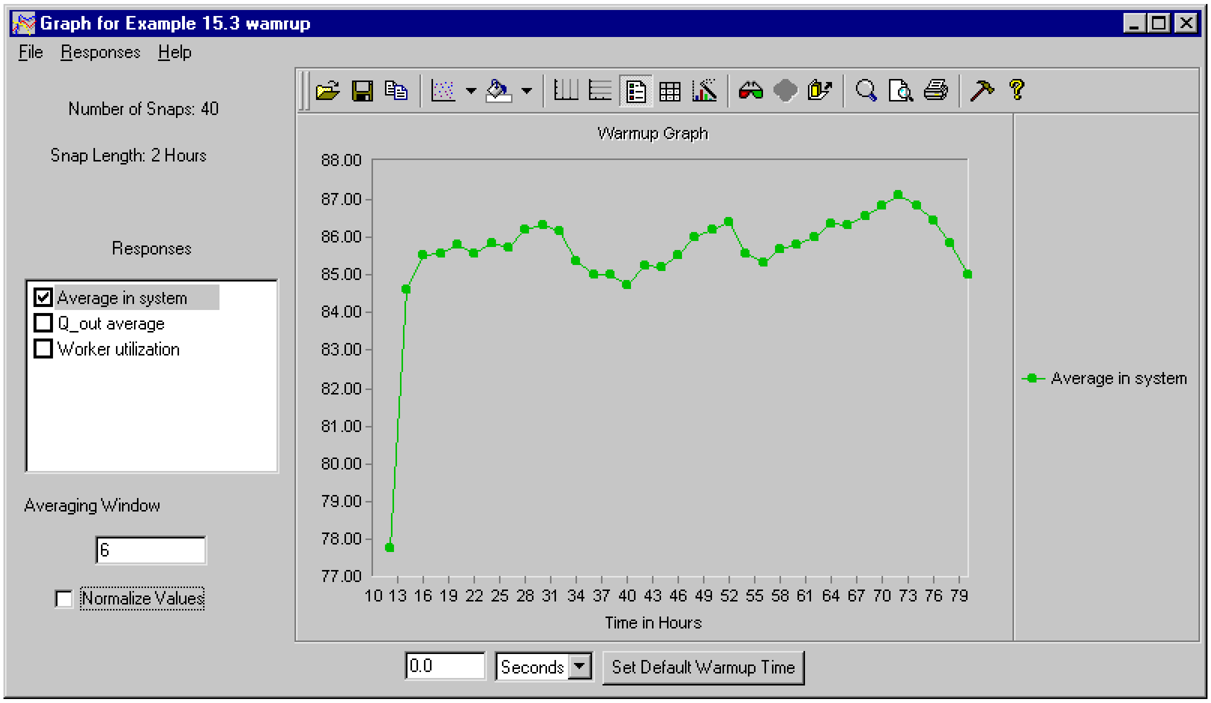
Step 3 ***Click*** OK, Do These Runs. AutoStat makes 15 additional runs.

When the runs have finished, display the warmup graph:

Step 1 ***Display*** the warmup graph for the response “Average in system.”

Each point is now the average of 20 snaps, instead of 5.

Step 2 ***Change*** the averaging window value to “6,” then ***press*** Tab. The graph is updated as shown below:



Graph for Example 15.3 warmup

Number of Snaps: 40

Snap Length: 2 Hours

Responses

☑ Average in system
☐ Q_out average
☐ Worker utilization

Averaging Window: 6

☐ Normalize Values

0.0 Seconds

Average in system with an averaging window of 6 for 20 replications

This graph has much less variation than before. Based on this response, the model reaches steady state after about the third data point (16 hours). To determine the warmup time from this graph, use the warmup formula:

$$\text{Warmup time} = \text{Steady state time} - w \times \text{Snap length}$$

For this model:

$$16 - (6 \times 2) = 4$$

Therefore, based on this response, the warmup time for the model is 4 hours.

Determining warmup times for systems with extreme variation

Sometimes response values for a model can vary even more widely than they did in the cyclical example model 15.3. It can be difficult to determine a warmup time for models with extreme variation. Example 15.4 will help illustrate how to analyze such models.

Example 15.4: Extreme variation warmup behavior

Example model 15.4 is the same process as example model 15.3. However, the break cycle for the workers has been changed to use times that are exponentially distributed, as shown below:

```
begin P_break arriving
   while 1=1 do
      begin
         wait for e 200 min /* Time between breaks */
         take down R_work
         wait for e 40 min /* Break time */
         bring up R_work
      end
end
```

The exponential times cause much wider variation than the constant breaks in example model 15.3.

To analyze example model 15.4:

Step 1 ***Import*** a copy of the *base* version of example model 15.4 in AutoMod.

Step 2 From the Model menu, ***select*** Run AutoStat, then ***click*** Yes to build the model. The AutoStat Setup wizard opens.

Step 3 ***Set up*** the model using the following parameters:

- Model is random
- Do not check for infinite loops
- Do not define a warmup
- Snap length is 4 hours

Defining the warmup analysis

Example model 15.4 will use new warmup analysis information, because the model has more variation and uses more random numbers than it did in the previous examples. Because of the greater variability in the breaks, the snaps need to be longer, as well.

Step 1 ***Define*** a warmup analysis for example model 15.4 with a snap length of 4 hours (use the default of 5 replications and 40 snaps).

Step 2 ***Set*** the seed increment value to 75,000.

Step 3 ***Define*** the same responses as you did for example model 15.1 (see “Defining responses” on page 15.12):

- Average in system
- Q_out average
- Worker utilization

Step 4 ***Make*** all the runs.

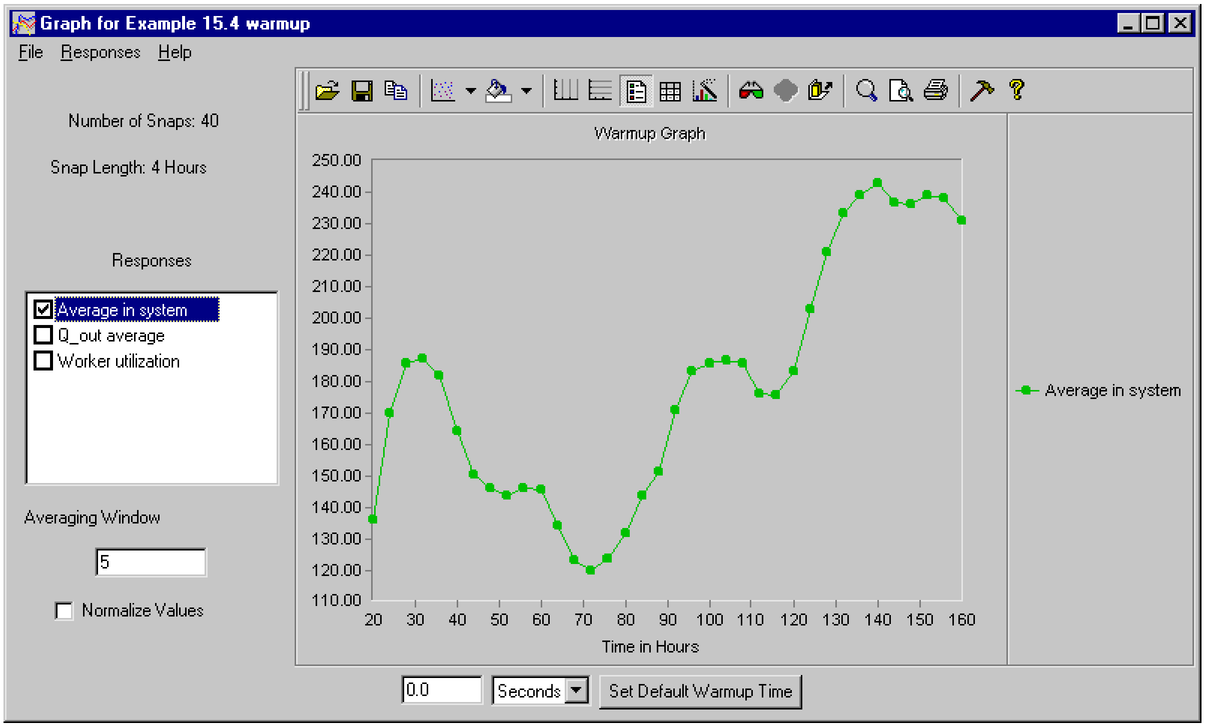
Analyzing the warmup graph for example model 15.4

Once the runs have finished, you can begin analyzing the warmup for this model:

Step 1 ***Display*** the warmup graph for the response “Average in system” with the default averaging window value of 5.

Note The warmup analysis for this model is based on the response “Average in system.” Analyzing the “Q_out average” and “Worker utilization” responses is left as an exercise for the chapter (See “Exercises” on page 15.36).

The graph is shown below:



Graph for Example 15.4 warmup

Number of Snaps: 40

Snap Length: 4 Hours

Responses

☑ Average in system
□ Q_out average
□ Worker utilization

Averaging Window: 5

□ Normalize Values

0.0 Seconds

Average in system with a snap length of 4 hours and an averaging window of 5

This response has large variation, and ends at a much higher level than it begins, which indicates that steady state has not been reached. According to “Adjusting warmup parameters” on page 15.8, when steady state has not been reached, you need to increase the run length by either increasing the number of snaps, increasing the snap length, or both.

Step 2 ***Close*** the warmup graph.

For this model, increase the snap length to 12 hours:

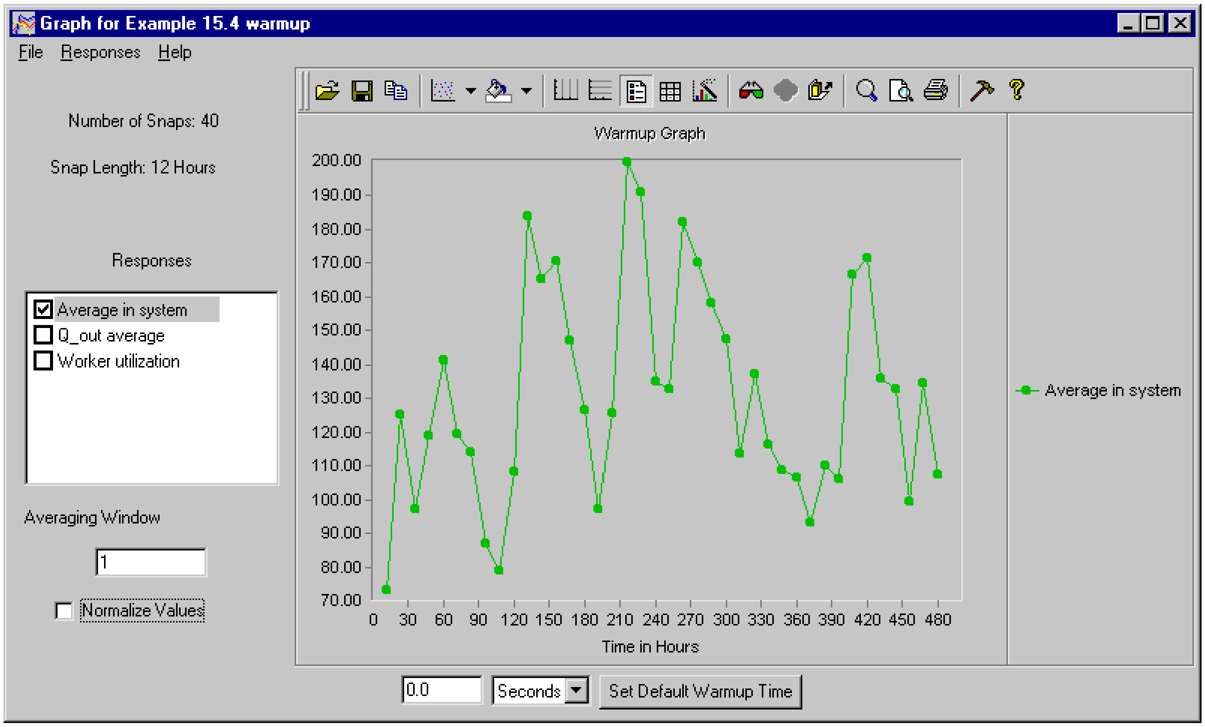
Step 1 ***Edit*** the warmup analysis.

Step 2 ***Change*** the snap length to 12 hours.

Step 3 ***Click*** OK, Do These Runs.

Step 4 ***Display*** the warmup graph for the response "Average in system."

Step 5 ***Change*** the averaging window value from "5" to "1," then ***press*** Tab. The graph is updated as shown below:



Number of Snaps: 40

Snap Length: 12 Hours

Responses

☑ Average in system
□ Q_out average
□ Worker utilization

Averaging Window: 1

□ Normalize Values

0.0 Seconds

Average in system with a snap length of 12 and an averaging window of 1

The graph is still not showing a steady state. The average number of loads in the system varies almost as much at the end of the graph as it does at the beginning. In order to decrease variability, you need to increase the run length more.

Step 6 ***Close*** the warmup graph.

Because the model has such wide variation, increase the number of replications to 20 to decrease the variability by half:

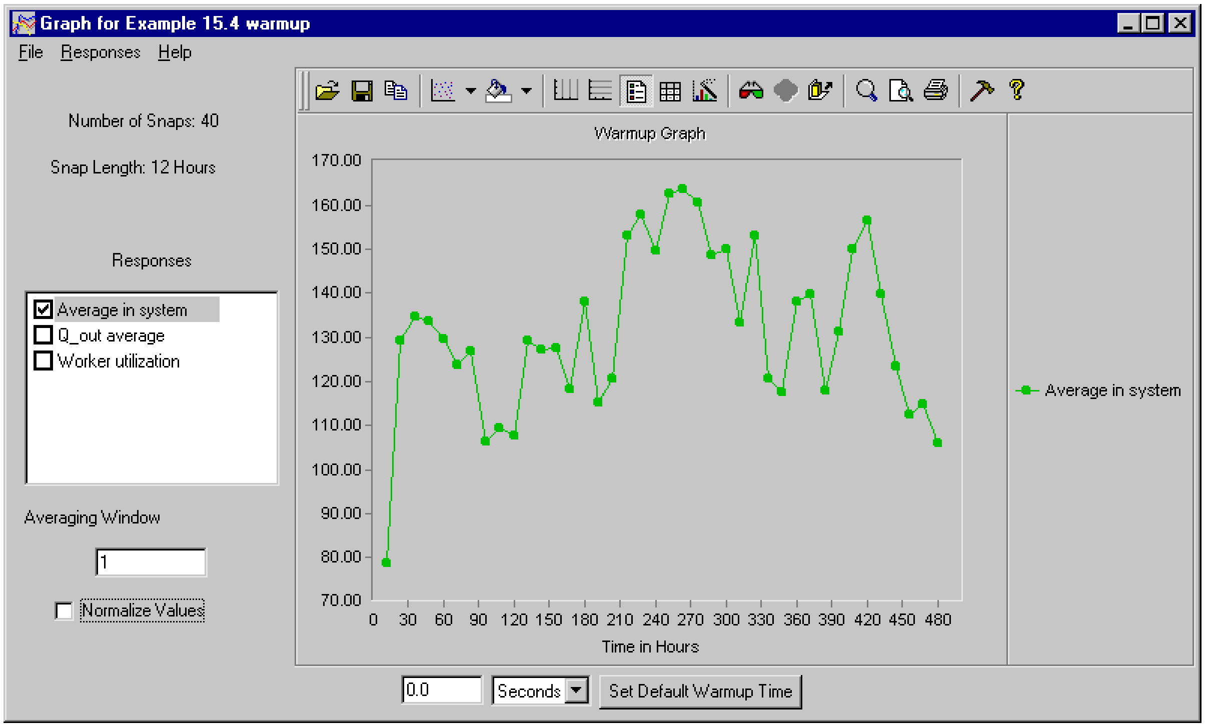
Step 1 ***Edit*** the warmup analysis.

Step 2 ***Change*** the number of replications to 20.

Step 3 ***Click*** OK, Do These Runs. AutoStat makes 15 additional runs.

Step 4 ***Display*** the warmup graph for the response “Average in system.”

Step 5 ***Change*** the averaging window value from “5” to “1,” then ***press*** Tab. The graph is updated as shown below:



Number of Snaps: 40

Snap Length: 12 Hours

Responses

☑ Average in system
☐ Q_out average
☐ Worker utilization

Averaging Window: 1

☐ Normalize Values

0.0

Average in system with 40 replications and an averaging window of 1

The graph shows a warmup trend at the beginning, but the rest of the graph still varies too widely to make a warmup determination. More replications are necessary to reduce the variability.

Step 6 ***Close*** the warmup graph.

Increase the number of replications to 80 to decrease the variability by half:

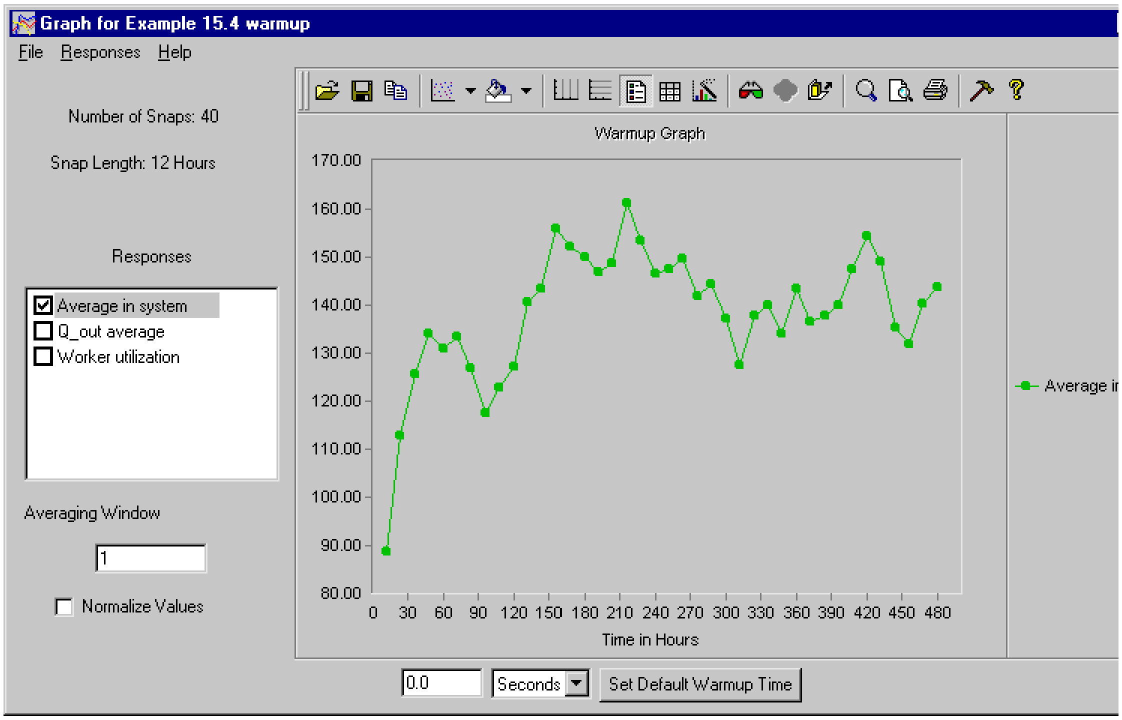
Step 1 ***Edit*** the warmup analysis.

Step 2 ***Change*** the number of replications to 80.

Step 3 ***Click*** OK, Do These Runs. AutoStat makes an additional 60 runs.

Step 4 ***Display*** the warmup graph for the response “Average in system.”

Step 5 ***Change*** the averaging window value from “5” to “1,” then ***press*** Tab. The graph is updated as shown below:



Graph for Example 15.4 warmup

Number of Snaps: 40

Snap Length: 12 Hours

Responses

☑ Average in system
☐ Q_out average
☐ Worker utilization

Averaging Window: 1

☐ Normalize Values

0.0 Seconds

Average in system with 80 replications and an averaging window of 1

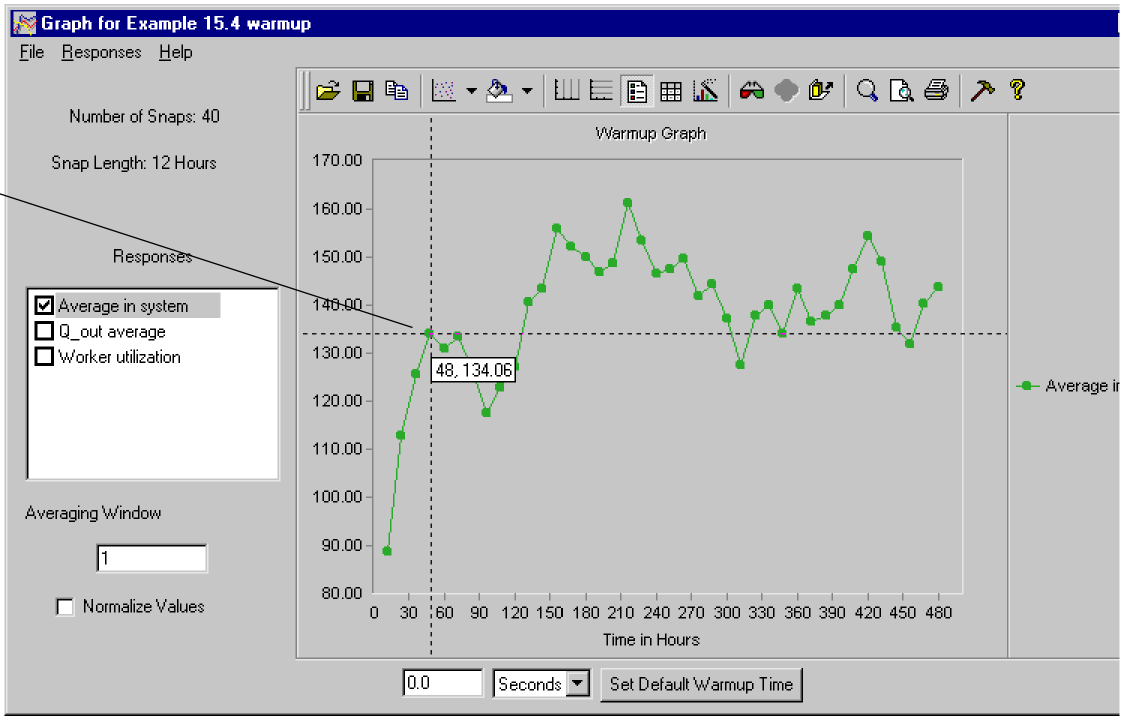
The additional replications have smoothed the end of the graph so that the response values appear to fluctuate within a steady range of values after the initial peak.

Additional replications could shorten the warmup time further. However, time constraints often require that you make a determination with data such as the data currently available in this graph.

The point at which analysts would say that steady state begins in this graph is subjective; some people choose the first point that falls within the range of values of steady state, while others choose a point that is more clearly in steady state and falls approximately in the middle of the steady state range.

For example, some analysts might designate the point at the top of the first peak in the graph the steady state point.

Step 1 ***Drag*** the mouse over the fourth data point, as shown below:



Graph for Example 15.4 warmup

Number of Snaps: 40

Snap Length: 12 Hours

Responses

☑ Average in system
☐ Q_out average
☐ Worker utilization

Averaging Window: 1

☐ Normalize Values

48, 134.06

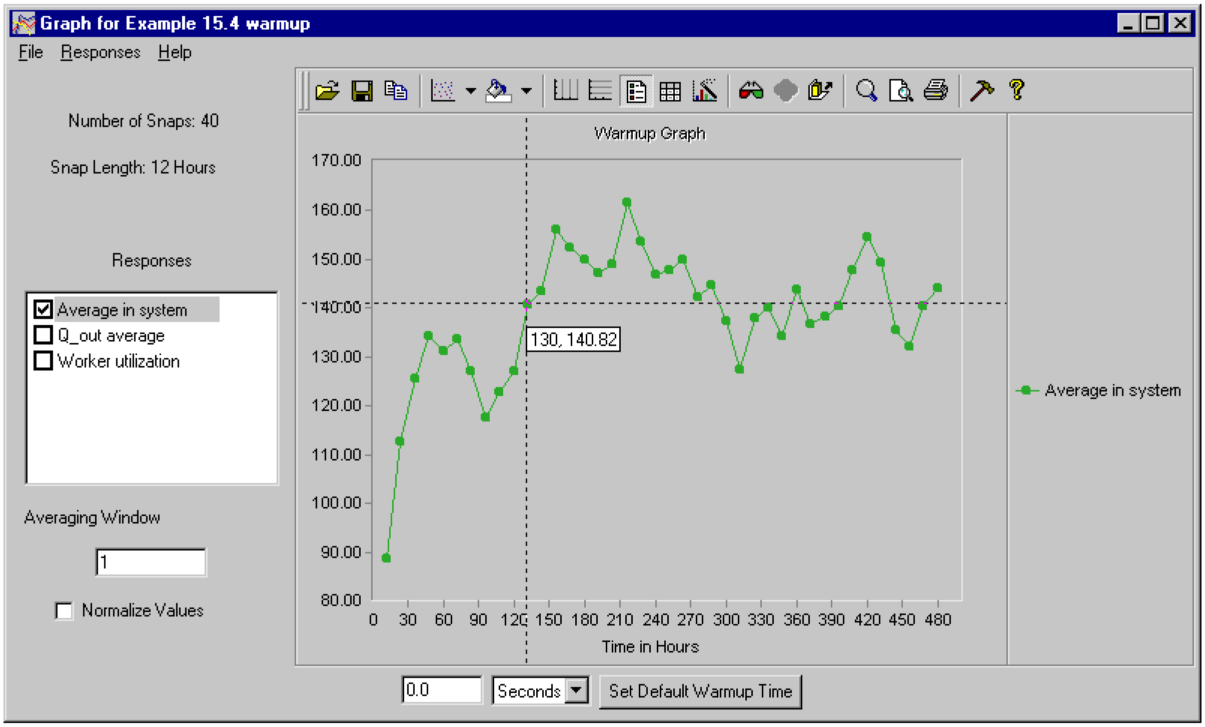
0.0 Seconds

The ToolTip contains the X and Y values of the data point

Using the mouse to determine the warmup time

Using this point, the warmup time is calculated as $48 - (1 \times 12) = 36$ hours. The 48-hour point eliminates the initial data points that range as low as 90. The fluctuations of the remaining portion of the graph indicate that the average loads in system range from 117 to 161.

Other analysts, however, might feel that 48 hours is not far enough into the graph to remove all of the initial bias and might choose a later point, such as 130 hours, as shown below:



Number of Snaps: 40

Snap Length: 12 Hours

☑ Average in system
☐ Q_out average
☐ Worker utilization

Averaging Window: 1

☐ Normalize Values

0.0 Seconds

Choosing a later warmup time

Given the point at 130 hours, the warmup time is $130 - (1 \times 12) = 118$ hours. The 130 hour point eliminates more of the initial fluctuations and narrows the range of the steady state values by 10 loads. However, the longer warmup time of 118 hours increases the time it will take to make all other analyses, because each replication must run for 118 hours before beginning to collect statistics.

As an analyst, you would need to decide whether to make more replications to reduce variability further, or use one of the points in this graph as your warmup time. The speed of your computer and the time you have available for completing all of your analyses may be deciding factors in which approach you choose.

Tip

For larger models, making large numbers of runs may not be possible, given the time available, unless you make runs on more than one CPU. Making runs on multiple CPUs is discussed in the AutoStat online help.

Summary

This chapter illustrates how to determine warmup times using the AutoStat software. Proper warmup determination is important for statistical analysis of systems that do not begin in steady state.

To conduct a warmup determination, you use a combination of several parameters: the number of replications, the snap length, the number of snaps, and the averaging window value.

This chapter discusses how to determine the warmup time for four different types of models:

- Systems with “classic” warmup behavior, for which the warmup time is easily determined.
- Explosive systems, for which the model never warms up. Explosive systems must be stabilized so that they are not explosive before any analyses, including warmup determination, are conducted or the results will be invalid.
- Cyclical systems, which illustrate a predictable level of variation that can be discerned with analysis.
- Systems with extreme variation, which require a large number of replications. If making a large number of replications is not possible due to time constraints of the project, making runs on multiple computers or CPUs can help.

Given enough analysis time, any system that reaches steady state could exhibit a “classic” warmup graph. In reality, time for analysis is finite; therefore, being able to determine warmup trends for different types of graphs is necessary for accurate modeling.

This chapter also discusses the importance of ensuring unique random number seeds for statistically sound analyses.

Exercises

Exercise15.1

Create a new model in which loads arrive in the system with an interarrival time that is uniformly distributed between 20 and 30 minutes. The loads enter an infinite-capacity queue, in which they are processed by a drill with a capacity of 2. It takes the drill a time that is uniformly distributed between 21 and 33 minutes to process each load.

After being drilled, the loads move into an infinite-capacity queue, in which they are processed by a grinder with a capacity of 4. It takes the grinder a time that is uniformly distributed between 36 and 60 minutes to process each load.

After building the model, perform a warmup analysis. Base the warmup analysis on the following responses:

- The average queue size for the drill
- The average queue size for the grinder
- The average number of loads in the system

Record the number of replications, number of snaps, snap length, averaging window value, and seed increment used to make your determination.

Exercise 15.2

Create a new model in which loads arrive in the system with an interarrival time that is uniformly distributed between 20 and 30 minutes. The loads enter an infinite-capacity queue, in which they are processed by a drill with a capacity of 2. It takes the drill a time uniformly distributed between 21 and 33 minutes to process each load.

After being drilled, the loads move into a queue with a capacity of 5, in which they are processed by a grinder with a capacity of 4. It takes the grinder a time that is uniformly distributed between 36 and 60 minutes to process each load.

After building the model, perform a warmup analysis. Base the warmup analysis on the following responses:

- The average queue size for the drill
- The average queue size for the grinder
- The average number of loads in the system

Record the number of replications, number of snaps, snap length, averaging window value, and seed increment used to make your determination.

Exercise 15.3

Create a new model in which loads arrive in the system with an interarrival time that is exponentially distributed with a mean of 15 minutes. Loads move into an infinite-capacity queue, in which they are serviced by a machine for a time that is exponentially distributed with a mean of 10 minutes.

After building the model, perform a warmup analysis. Determine the appropriate responses.

Record the number of replications, number of snaps, snap length, averaging window value, and seed increment used for your analysis, as well as the responses used to make your determination.

Exercise 15.4

Create a new model in which loads arrive in the system with an interarrival time that is uniformly distributed between 5 and 25 minutes. Loads move into an infinite-capacity queue, in which they are serviced by a machine with a capacity of 1 for a time that is uniformly distributed between 3 and 21 minutes.

The machine's time between failures is uniformly distributed between 40 and 70 minutes. The time to repair the machine is uniformly distributed between 2 and 8 minutes.

After building the model, perform a warmup analysis. Base the warmup analysis on the average number of jobs in the system.

Record the number of replications, number of snaps, snap length, averaging window value, and seed increment used to make your determination.

Exercise 15.5

Copy the *final* version of example model 15.3 to a new directory. Define the two responses that were not analyzed in the chapter:

- Q_out average
- Worker utilization

Make the runs for the defined warmup analysis. Determine the model's warmup based on these responses *and* the "Average in system" response (described in "Analyzing the warmup graph for example model 15.3" on page 15.25).

Record the averaging window value used to make your determination.

Exercise 15.6

Copy the *final* version of example model 15.4 to a new directory. Define the two responses that were not analyzed in the chapter:

- Q_out average
- Worker utilization

Make the runs for the defined warmup analysis. Determine the model's warmup based on these responses *and* the "Average in system" response (described in "Analyzing the warmup graph for example model 15.4" on page 15.29).

Record the averaging window value used to make your determination.

Exercise 15.7

Why might two simulation analysts determine different warmup times for the same model?

Exercise 15.8

There are two alternatives to conducting warmup determination:

1. Running an extremely long simulation.

2. Setting the system's initial values to approximate steady state.

Why is warmup analysis used more often than these methods?

Exercise 15.9

The warmup procedure used in this chapter is called Welch's Method and is attributed to P.D. Welch. Look at one or more of the following or other sources and summarize what they say about this technique:

Alexopoulos, C., and A.F. Seila [1998], "Output Data Analysis," in *Handbook of Simulation*, J. Banks, ed., John Wiley, New York, pp. 238-242.

Banks, J., J.S. Carson, II, B.L. Nelson, D.M. Nicol [2000], *Discrete-Event System Simulation*, Prentice-Hall, Upper Saddle River, NJ, pp. 451-458.

Law, A.M. and W.D. Kelton [2000], *Simulation Modeling and Analysis*, 3rd Ed., New York: McGraw-Hill.

Welch, P.D. [1983], "The Statistical Analysis of Simulation Results," in *The Computer Performance Modeling Handbook*, S. Lavenberg, ed., Academic Press, New York, pp. 268-328.

References

Balci, O. (1988) “The Implementation of Four Conceptual Frameworks for Simulation Modeling in High-level Languages,” in *Proceedings of the 1988 Winter Simulation Conference*, eds., M.A. Abrams, P.L. Haigh, J.C. Comfort, Institute of Electrical and Electronics Engineers, Piscataway, N.J., pp. 287-295.

Balci, O. (1998) “Verification, Validation, and Testing,” chapter 10 in *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*, ed., Jerry Banks, John Wiley & Sons, New York.

Banks, J., ed. (1998) *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*, John Wiley, New York.

Banks, J. and J. Dai (1997) “Simulation Studies of Multiclass Queueing Networks,” *IIE Transactions*, March.

Banks, J., J.S. Carson II, and D. Goldsman (1998) “Discrete-Event Computer Simulation,” in *Handbook of Statistical Methods for Engineers and Scientists*, 2nd Ed., ed. H.M. Wadsworth, McGraw-Hill, New York.

Banks, J., J.S. Carson II, B.L. Nelson, and D.M. Nicol, (2000), *Discrete-Event System Simulation*, 3rd Ed., Prentice-Hall, Upper Saddle River, NJ.

Banks, J. and V. Norman (1995) “Justifying Simulation in Today’s Manufacturing Environment,” *IIE Solutions*, November.

Carson, J.S. (1993) “Modeling and Simulation World Views,” in *Proceedings of the 1993 Winter Simulation Conference*, eds., G.W. Evans, M. Mollaghasemi, E.C. Russell, and W.E. Biles, Institute of Electrical and Electronics Engineers, Piscataway, N.J., pp. 18-23.

Knuth, D.W. (1969) *The Art of Computer Programming*, vol. 2: *Semi-Numerical Algorithms*, Addison-Wesley, Reading, Mass.

Law, A.M. and W.D. Kelton (2000) *Simulation Modeling and Analysis*, 3rd Ed., McGraw-Hill, New York.

Little, J.D.C. (1961) “A Proof for the Queueing Formula $L = \lambda w$,” *Operations Research*, Vol. 16, pp. 651-65.

Pidd, M. (1998) *Computer Modelling for Discrete Simulation*, 4th Ed., John Wiley & Sons, Chichester, England.

Sargent, R.G. (1992) “Validation and Verification of Simulation Models,” in *Proceedings of the 1992 Winter Simulation Conference*, eds., J.J. Swain, D. Goldsman, R.C. Crain, and J.R. Wilson, Institute of Electrical and Electronics Engineers, Piscataway, N.J., pp. 104-114.

Schriber, T.J. (1991) *An Introduction to Simulation Using GPSS/H*, John Wiley & Sons, New York.

Welch, P.D. (1983) “The Statistical Analysis of Simulation Results,” in *The Computer Performance Modeling Handbook*, ed. S. Lavenberg, Academic Press, Orlando, Fla.

Index