SimEvents

For Use with Simulink®

- Modeling
- Simulation
- Implementation

Getting Started

Version 1

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

Getting Started with SimEvents

Revision History

November 2005	Online only	New for Version 1.0 (Release 14SP3+)
March 2006	First printing	Revised for Version 1.1 (Release 2006a)

Contents

1 Introduction

2 Building Simple Models with SimEvents

3 Creating Entities Using Intergeneration Times

4 Basic Queues and Servers

5 Designing Paths for Entities

1

Introduction

What Is SimEvents?

SimEvents extends Simulink® with tools for modeling and simulating discrete-event systems using queues and servers. With SimEvents you can create a discrete-event simulation model in Simulink to simulate the passing of entities through a network of queues, servers, gates, and switches based on events. SimEvents and Simulink provide an integrated environment for modeling hybrid dynamic systems containing continuous-time, discrete-time, and discrete-event components.

What Is Discrete-Event Simulation?

Informally, a *discrete-event simulation*, or event-based simulation, permits the system's state transitions to depend on asynchronous discrete incidents that are called *events*. By contrast, a simulation based solely on differential equations in which time is an independent variable is a *time-based simulation* because state transitions depend on time. Simulink is designed for time-based simulation, while SimEvents is designed for discrete-event simulation. Your choice of a different simulation style can depend on the particular phenomenon you are studying and/or the way you choose to study it. Some examples illustrate these differences:

- Suppose you are interested in how long the average airplane waits in a queue for its turn to use an airport runway, but not interested in the details of how an airplane moves once it is cleared for takeoff. You might use discrete-event simulation in which the relevant events include the approach of a new airplane to the runway and the clearance for takeoff of an airplane in the queue.
- Suppose you are interested in the trajectory of an airplane as it takes off. You would probably use time-based simulation because finding the trajectory involves solving differential equations.
- Suppose you are interested in how long the airplanes wait in the queue but you want to model the takeoff in some detail instead of using a statistical distribution to model the length of time that each plane uses the runway. You might use a combination of time-based simulation and discrete-event simulation, where the time-based aspect controls details of a plane's takeoff and the discrete-event aspect controls the queuing behavior.

A detailed description and precise definition of discrete-event simulation are beyond the scope of this documentation set; for details, see [3] or [7].

Resources for Learning

To help you learn about SimEvents more effectively and efficiently, this section highlights some learning resources that might be appropriate for you depending on your background. Some resources are within this documentation set and others are outside it.

New Discrete-Event Simulation Modelers

If you are new to discrete-event simulation, then one or more of the works listed in Chapter 6, "Selected Bibliography" can help you learn about the subject. A detailed treatment of discrete-event systems is beyond the scope of this documentation set, which aims to explain how to use SimEvents along with Simulink.

When you are learning how to use SimEvents along with Simulink, you might be especially interested in the discussions of key concepts and timing issues, such as:

- "Key SimEvents Concepts" on page 2-34
- "Working with Entities" and "Working with Events" online
- "Working with Signals" online
- "How SimEvents Works" online

New Simulink Users

If you are new to Simulink, then this Getting Started guide and selected portions of the Simulink documentation can help you learn how to use the Simulink modeling environment. In addition, see the set of Simulink and SimEvents demonstrations, which you can access using the **Demos** tab of the MATLAB® Help browser.

Experienced Simulink Users

If you are accustomed to Simulink features and Simulink timing semantics, then you should understand how SimEvents and Simulink work together and how they differ from each other. In particular, see

- "Key SimEvents Concepts" on page 2-34
- "Working with Signals" online
- "Controlling Timing Using Subsystems" online
- "How SimEvents Works" online

Notes on Engineering Subject Matter

This guide expects that you are familar with the engineering subject matter that you want to address using SimEvents. While this guide might present examples from subject areas other than your own, you can still use the examples to learn about SimEvents features.

Installing SimEvents

To use SimEvents, you must first install all of these products:

- MATLAB
- Simulink
- SimEvents

For instructions, see the MATLAB installation documentation for your platform.

If you have installed MATLAB and want to check which other MathWorks products are installed, enter `ver` in the MATLAB Command Window.

What Is an Entity?

Discrete-event simulations typically involve discrete items of interest. By definition, these items are called *entities* in SimEvents. Entities can pass through a network of queues, servers, gates, and switches during a simulation. Entities can carry data, known in SimEvents as *attributes*.

Note Entities are not the same as events. Events are instantaneous discrete incidents that change a state variable, an output, and/or the occurrence of other events. See "What Is an Event?" on page 1-7 for details.

Examples of entities in some sample applications are listed in the table below.

Context of Sample Application	Entities
Airport with a queue for runway access	Airplanes waiting for access to runway
Communication network	Packets or messages to be transmitted
Bank of elevators	People traveling in elevators
Conveyor belt for assembling parts	Parts being assembled
Computer with one or more CPUs	Computational tasks or jobs

A SimEvents model uses blocks to represent components that process entities, but entities themselves do not have a graphical representation. When you design and analyze your discrete-event simulation, you might choose to focus on the entities themselves or on the processes they undergo. For example, you might pose questions about the average waiting time for a series of entities entering a queue, or questions about which step in a multistep process (that entities undergo) is most susceptible to failure.

What Is an Event?

In a discrete-event simulation, an event is an instantaneous discrete incident that changes a state variable, an output, and/or the occurrence of other events. Examples of supported events in SimEvents are

- The advancement of an entity from one block to another.
- The completion of service on an entity in a server.
- A zero crossing of a signal connected to a block that is configured to react to zero crossings. These events are also called *trigger edges*.
- A function call, which is a discrete invocation request carried from block to block by a special signal called a function-call signal. Function calls are the recommended way to make Stateflow® blocks and blocks in the Simulink libraries respond to asynchronous state changes.

For a full list of supported events and more details on them, see “Working with Events” online.

Relationships Among Events

Events in a simulation can depend on each other:

- One event might be the sole cause of another event. For example, the arrival of the first entity in a queue causes the queue length to change from 0 to 1.
- One event might enable another event to occur, but only under certain conditions. For example, the completion of service on an entity enables the entity’s departure from the server, but only if the subsequent block is able to accept the arrival of that entity. In this case, one event makes another event possible, but does not solely cause it.

Events that occur at the same value of the simulation clock are called simultaneous events, even if they are processed sequentially. When simultaneous events are not causally related to each other, the processing sequence can significantly affect the simulation behavior. For an example, see the Event Priorities demo or “Example: Race Conditions at a Switch”. For more details, see “Processing Sequence for Simultaneous Events” online.

Viewing Events

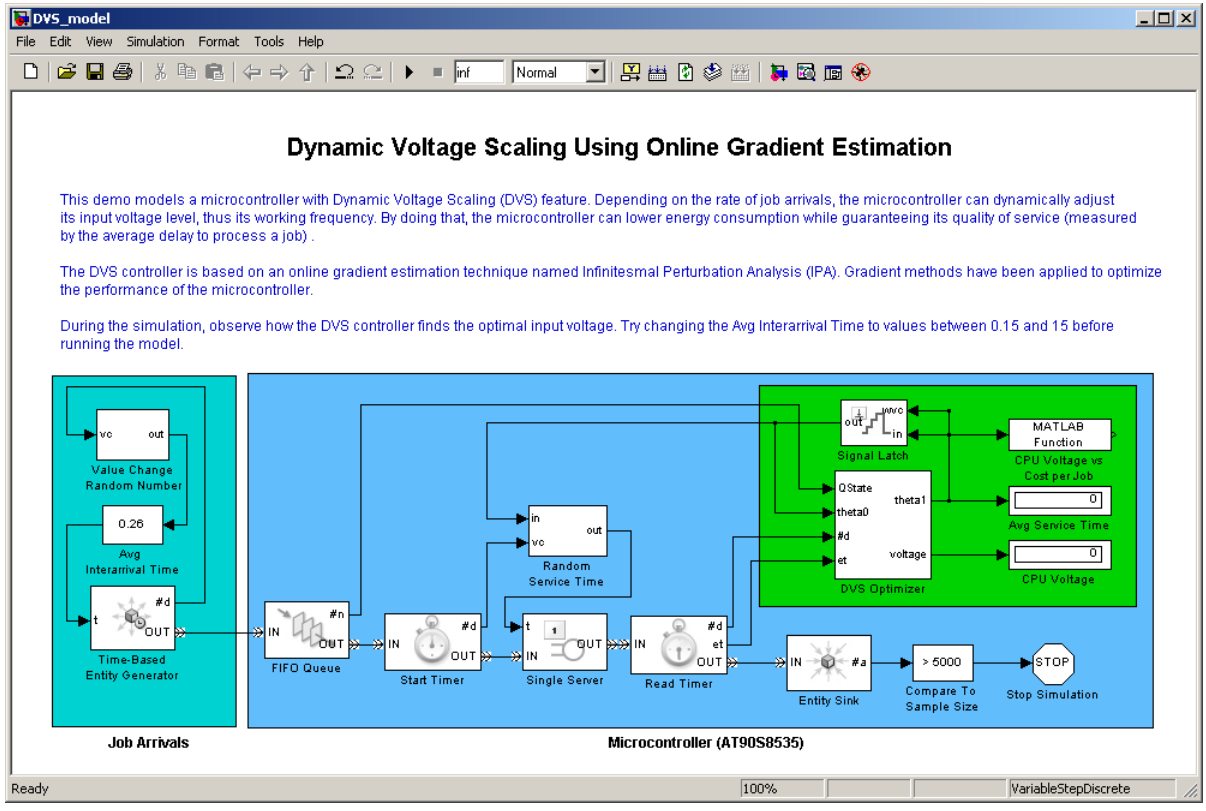
Events do not have a graphical representation. You can infer their occurrence by observing their consequences, by using the Instantaneous Event Counting Scope block, or by using the event logging feature. For details, see “Observing Events” online.

Running a Demo Simulation

One way to become familiar with the basics of SimEvents models and the way they work is to examine and run a previously built model. This section describes a demo model that comes with SimEvents. The model simulates a technique for dynamically adjusting the energy consumption of a microcontroller based on the workload, without compromising quality of service. Changes in the workload can occur as discrete events.

Opening the Model

To open this demo, enter `DVS_model` in the MATLAB Command Window.



Alternatively, you can open the MATLAB Help browser and, in the **Demos** tab, click the **+** sign next to Simulink, SimEvents, and Application Demos. In the expanded list of application demos, double-click the listing for Dynamic Voltage Scaling Using Online Gradient Estimation.

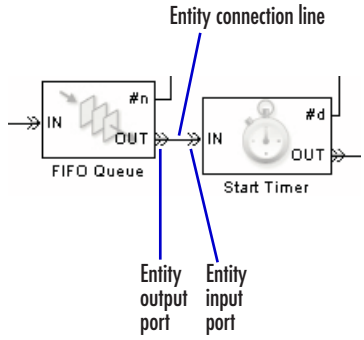
Examining Entities and Signals in the Model

This section describes the different kinds of ports and lines that appear in the `DVS_model` model. Compared to signal ports, entity ports look different and represent a different concept.

Entity Ports and Connections

Some blocks in this model process entities, which are mentioned in "What Is an Entity?" on page 1-6.

The FIFO Queue block and the Start Timer block, which are part of SimEvents, process entities in this model. Each of these blocks has an entity input port and an entity output port. The figure below shows the entity output port of the FIFO Queue block and the entity input port of the Start Timer block.



Entity connection lines represent relationships among two blocks (or among their entity ports) by indicating a path by which an entity can depart from one block to arrive at another block. The figure above shows the connection line from the FIFO Queue block's entity output port to the Start Timer block's entity input port. When you run the simulation, entities that depart from the

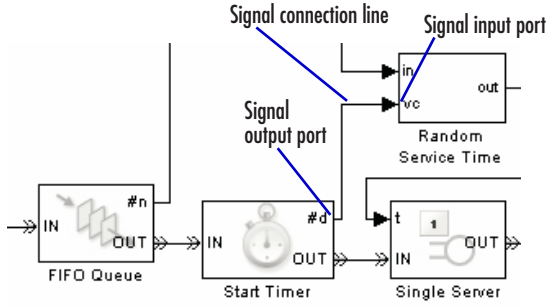
OUT port of the FIFO Queue block arrive simultaneously at the **IN** port of the Start Timer block.

By convention, entity ports use labels with words in uppercase letters, such as **IN** and **OUT**.

You cannot branch an entity connection line. If your application requires that an entity arrive at multiple blocks, use the Replicate block to create copies of the entity.

Signals and Signal Ports

Some blocks in this model process signals. Signals represent numerical quantities defined at all times during a simulation, not only at a discrete set of times. Signals are depicted as connection lines between signal ports of two blocks. The figure below shows that the Start Timer block has not only an entity output port but also a signal output port. The signal output port connects to the Random Service Time subsystem.



Key Components of the Model

The `DVS_model` model uses event-based blocks to simulate the workload of the microcontroller:

- At random times, the Time-Based Entity Generator block generates an entity that represents a job for the microcontroller.
- The FIFO Queue block stores jobs that the microcontroller cannot process immediately.

- The Single Server block models the microcontroller's processing of a job.

  This block can process at most one job at a time and thus limits the availability of the microcontroller to process new jobs. While a job is in this block, other jobs remain in the FIFO Queue block.

- The Start Timer and Read Timer blocks work together to compute the time that each job spends in the server. The result of the computation is the **et** output signal from the Read Timer block.

- The Entity Sink block absorbs jobs that have completed their processing.

Important discrete events in this model are the generation of a new job and the completion of a job's processing.

The model also includes blocks that simulate a dynamic voltage scaling (DVS) controller that adjusts the input voltage depending on the microcontroller's workload. The idea is to minimize the average cost per job, where the cost takes into account both energy consumption and quality of service. For more information about the cost and the optimization technique, see "Dynamic Voltage Scaling Using Online Gradient Estimation" online.

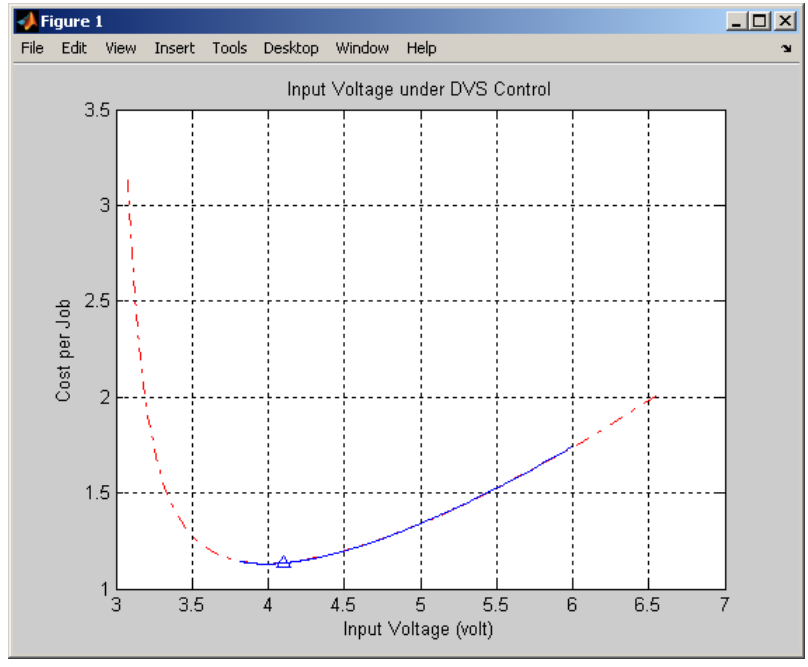
Appearance of Entities

Entities do not appear explicitly in the model window. However, you can gather information about entities using plots, signals, and the entity logging feature. See these sections for more information:

- "Example: Synchronizing Service Start Times with the Clock" online
- "Example: Selecting the First Available Server" on page 5-6
- "Plotting the Pending-Entity Signal" on page 2-15, which is part of the larger example "Building a Simple Discrete-Event Model" on page 2-2
- "Example: Entity Logging" online

Running the Simulation

To run the `DVS_model` simulation, choose **Simulation > Start** from the model window's menu. A Figure window opens with a dynamic plot showing how the DVS controller varies the voltage during the simulation to reduce the average cost per job. A triangle marker moves to indicate the current voltage and corresponding cost.



2

Building Simple Models with SimEvents

Building a Simple Discrete-Event Model

This section describes how to build a new model representing a discrete-event system. The system is a simple queuing system in which "customers" — entities — arrive at a fixed deterministic rate, wait in a queue, and advance to a server that operates at a fixed deterministic rate. This type of system is known as a D/D/1 queuing system in queuing notation. The notation indicates a deterministic arrival rate, a deterministic service rate, and a single server.

Using the example model, this section shows you how to

- Perform basic Simulink model-building tasks, such as adding blocks to models and configuring blocks using their parameter dialog boxes.
- Use plots to understand the behavior of a discrete-event simulation, including plots that show multiple values at a fixed time.
- Understand the notion of a blocked entity output port in a SimEvents block.

The topics in this section are as follows:

Opening a Model and Libraries

The first steps in building a model are to set up your environment, open a new model window, and open the libraries containing blocks.

Setting Default Parameters for Discrete-Event Simulation

To change the default Simulink model settings to values that are appropriate for discrete-event simulation modeling, enter this in the MATLAB Command Window:

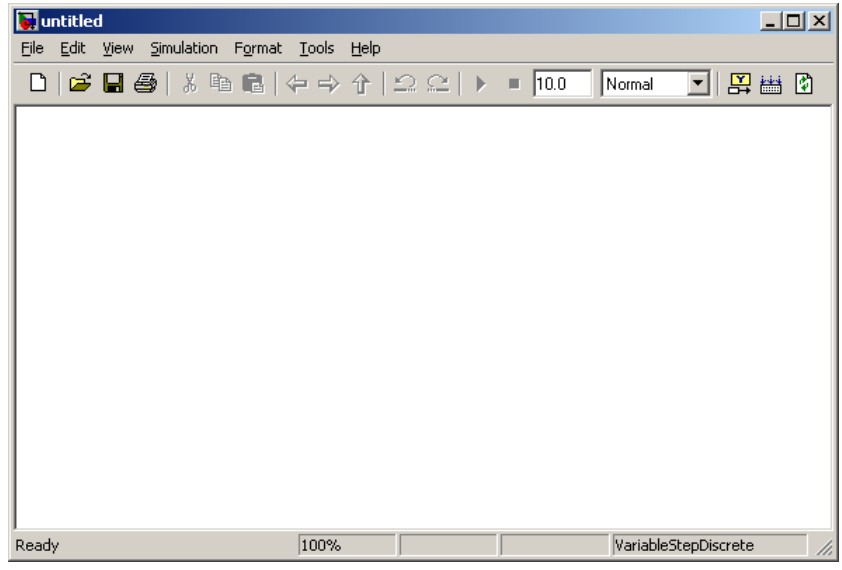
```
simeventsstartup('des');
```

MATLAB displays a message indicating that it changed default Simulink settings. The changed settings apply to new models that you create later in this MATLAB session, but not to previously created models.

Note To install these model settings each time you start MATLAB, invoke `simeventsstartup` from your `startup.m` file.

Opening a New Model Window

Select **File > New > Model** from the MATLAB window's menu. This opens an empty model window, shown below.



To name the model and save it as a file, select **File > Save** from the model window's menu. Save the model in your working directory under the filename `dd1.mdl`.

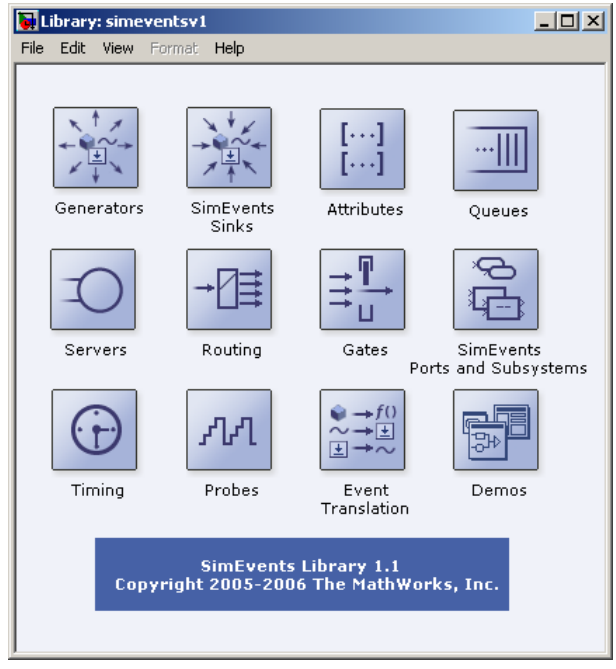
Opening SimEvents Libraries

In the MATLAB Command Window, enter

```
simeventslib
```

Alternatively, click the **Start** button in the lower-left corner of the MATLAB desktop. In the menu that appears, select **Simulink > SimEvents > Block Library**.

The main SimEvents window appears, as shown below. This window contains an icon for each SimEvents library. To open a library and view the blocks it contains, double-click the icon that represents that library.



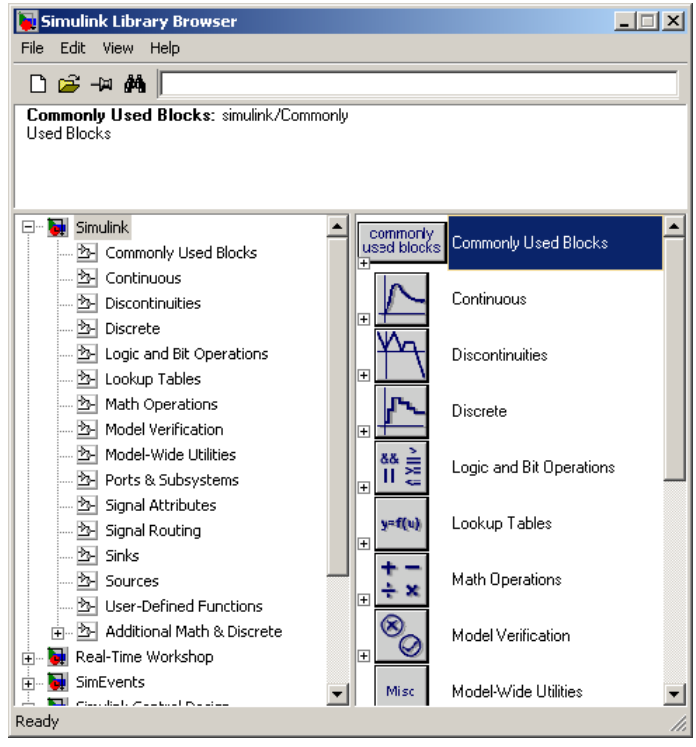
Opening Simulink Libraries

In the MATLAB Command Window, enter

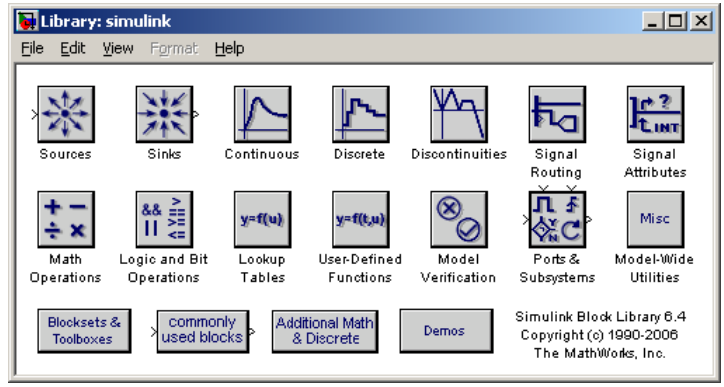
```
simulink
```

On Windows platforms, this opens the Simulink Library Browser, which uses a tree structure to display the available libraries and blocks. To view the blocks in a library listed in the left pane, select the library name and see the list of blocks in the right pane. The Simulink Library Browser provides access not only to Simulink but also to SimEvents.

On Unix platforms, the `simulink` command opens the Simulink library window, which contains an icon for each Simulink library. To open a library and view the blocks it contains, double-click the icon that represents that library.



Simulink Library Browser (Windows Platforms)

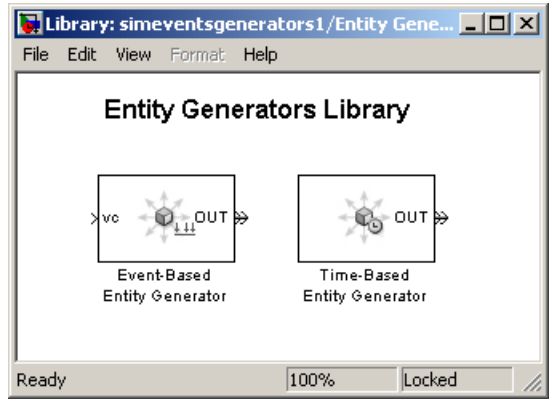


Simulink Library Window (UNIX Platforms)

Moving Blocks into the Model Window

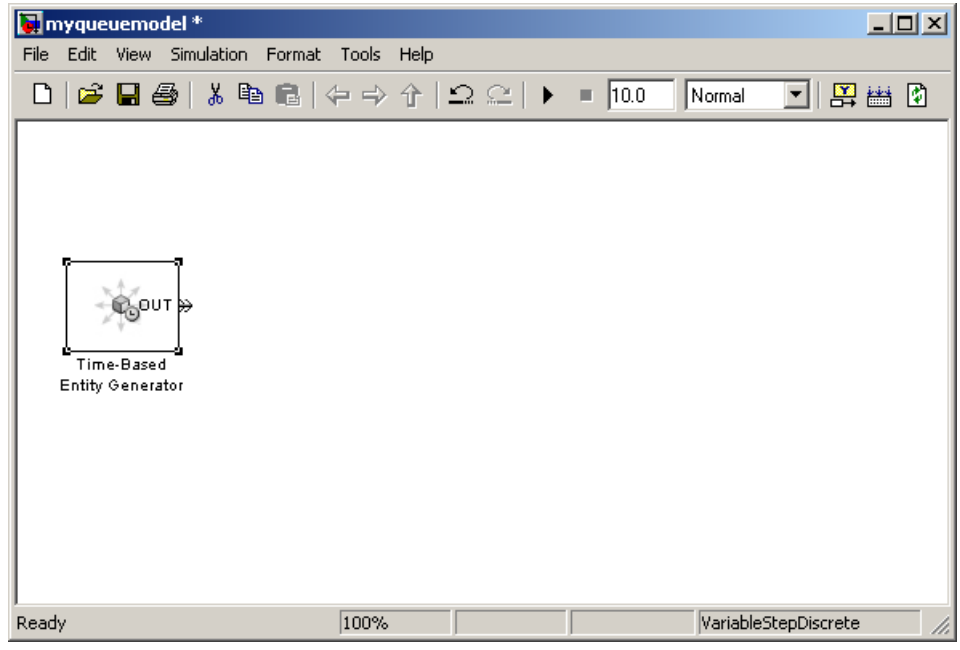
To move blocks from libraries into the model window, follow these steps:

1. In the main SimEvents window, double-click the Generators icon to open the Generators library. Then double-click the Entity Generators icon to open the Entity Generators sublibrary.

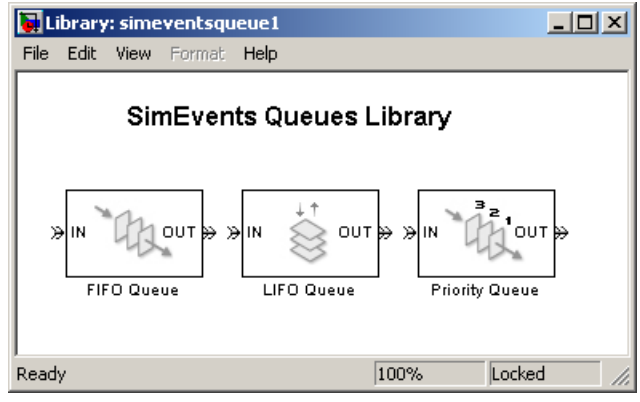


2. Drag the Time-Based Entity Generator block from the library into the model window.

This might cause an informational dialog box to open, with a brief description of the difference between entities and events.

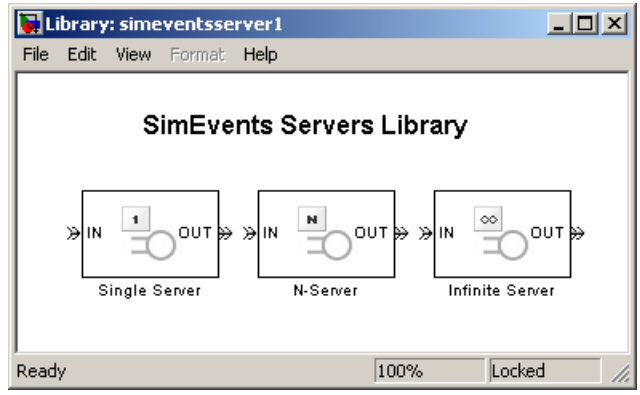


3 In the main SimEvents window, double-click the Queues icon to open the Queues library.



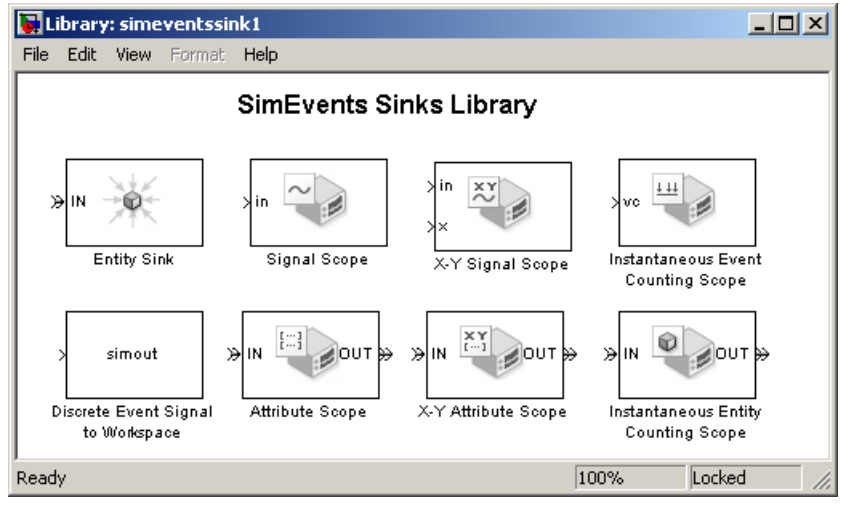
4 Drag the FIFO Queue block from the library into the model window.

5 In the main SimEvents window, double-click the Servers icon to open the Servers library.



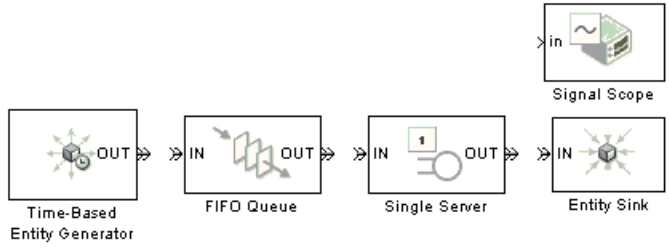
6 Drag the Single Server block from the library into the model window.

7 In the main SimEvents window, double-click the SimEvents Sinks icon to open the SimEvents Sinks library.



8 Drag the Signal Scope block and the Entity Sink block from the library into the model window.

As a result, the model window looks like the figure below. The model window contains blocks that represent the key processes in the demo: blocks that generate entities, store entities in a queue, serve entities, and create a plot showing relevant data.



Configuring Blocks

Configuring the blocks in `dd1` means setting their parameters appropriately to represent the system being modeled. Each block has a dialog box that enables you to specify parameters for the block. Default parameter values might or might not be appropriate, depending on what you are modeling.

Two important parameters in this D/D/1 queuing system are the arrival rate and service rate. The reciprocals of these rates are the duration between successive entities and the duration of service for each entity. To examine these durations, do the following:

1 Double-click the Time-Based Entity Generator block to open its dialog box. Observe that the **Distribution** parameter is set to `Constant` and that the **Period** parameter is set to `1`. This means that the block generates a new entity every second.

2 Double-click the Single Server block to open its dialog box. Observe that the **Service time** parameter is set to `1`. This means that the server spends one second processing each entity that arrives at the block.

3 Click **Cancel** in both dialog boxes to dismiss them without changing any parameters.

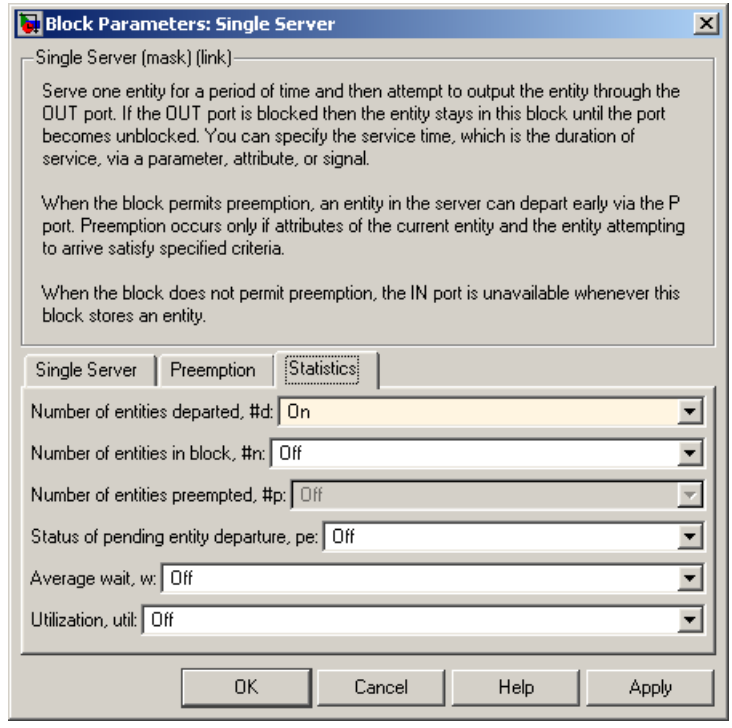
The **Period** and **Service time** parameters have the same value, which means that the server completes an entity's service at exactly the same time that a new entity is being created. The Event Priorities demo discusses this simultaneity in more detail.

For now, configure blocks to create a plot that shows when each entity departs from the server, and to make the queue have an infinite capacity. Do this as follows:

1 Double-click the Single Server block to open its dialog box.

2 Click the **Statistics** tab to view parameters related to the statistical reporting of the block.

3 Set the **Number of entities departed** parameter to `On`.



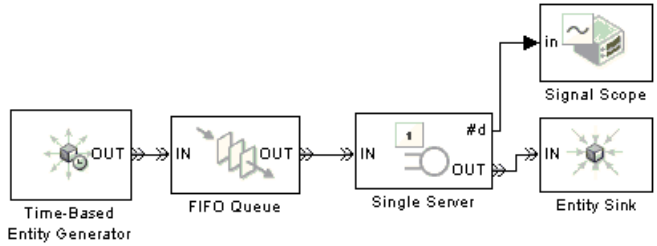
Then click **OK**. The Single Server block acquires a signal output port labeled **#d**. During the simulation, the block will produce an output signal at this **#d** port; the signal's value is the running count of entities that have completed their service and departed from the server.

4 Double-click the FIFO Queue block to open its dialog box.

5 Set the **Capacity** parameter to `Inf` and click **OK**.

Connecting Blocks

Now that the model window for `dd1` contains blocks that represent the key processes, connect the blocks to indicate relationships among them as shown in the figure below. To connect blocks with the mouse, drag from the output port of one block to the input port of another block.



Running the Simulation

Save your model. Then start the simulation by choosing **Simulation > Start** from the model window's menu.

Suppressing Solver Warnings

If you skipped "Setting Default Parameters for Discrete-Event Simulation" on page 2-3, then you might see warning messages in the MATLAB Command Window about continuous states and the maximum step size. These messages appear because certain default parameters for a Simulink model are inappropriate for this particular example model. Simulink overrides the inappropriate parameters and alerts you to that fact.

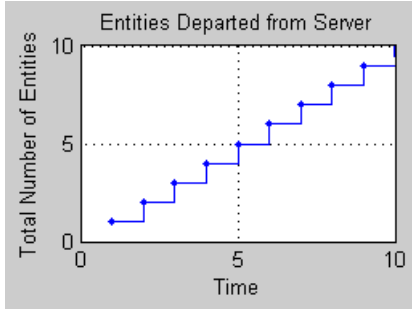
One way to suppress the warning messages when you run this simulation in the future is to enter this command in the MATLAB Command Window:

```
simeventsconfig('dd1','des');
```

MATLAB displays a message indicating that it changed the Simulink settings for this particular model.

Results of the Simulation

When the simulation runs, the Signal Scope block opens a window containing a plot. The horizontal axis represents the times at which entities depart from the server, while the vertical axis represents the total number of entities that have departed from the server.



After an entity departs from the Single Server block, the block updates its output signal at the **#d** port. The updated values are reflected in the plot and highlighted with plotting markers. From the plot, you can make these observations:

- Until T=1, no entities depart from the server. This is because it takes one second for the server to process the first entity.
- Starting at T=1, the plot is a stairstep plot. The stairs have height 1 because the server processes one entity at a time, so entities depart one at a time. The stairs have width equal to the constant service time, which is one second.

One fact not revealed by the plot is that when the simulation ends, one entity is still in the server. This entity is not included in the **#d** signal because the entity has not departed from the server. To see whether an entity remains in the server when the simulation ends, set the Single Server block's **Number of entities in block** parameter to `On` to obtain a **#n** signal output port, connect the port to a Signal Scope block, and set the scope block's **X value from** parameter to `Index`. In the resulting plot, the last value plotted is 1, not 0.

Creating Additional Plots

The `dd1` model that you created in the previous sections plots the number of entities that depart from the server. This section modifies the model to plot other quantities that can reveal aspects of the simulation. The topics are as follows:

Enabling the Pending-Entity Signal

When at least one entity is in the FIFO Queue block but is unable to depart and advance to the subsequent block, the FIFO Queue block's entity output port (labeled **OUT**) is said to be *blocked* and the entity at the head of the queue is said to be a *pending entity*. The FIFO Queue block can report whether it has a pending entity using a signal whose value is 1 if an entity is pending and 0 if no entity is pending or a previously pending entity has just departed. Note that a value of 0 could mean either that the queue is empty or that an entity trying to depart is successful. To configure the FIFO Queue block to report pending-entity status, do the following:

1. Double-click the FIFO Queue block to open its dialog box. Click the **Statistics** tab to view parameters related to the statistical reporting of the block.

2. Set the **Status of pending entity departure** parameter to `On` and click **OK**. This causes the block to have a signal output port for the pending-entity signal. The port label is **pe**.

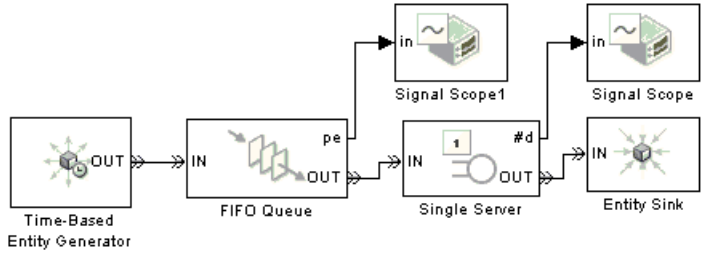
Plotting the Pending-Entity Signal

The model already contains a Signal Scope block for plotting the entity count signal. To add another Signal Scope block for plotting the pending-entity signal (enabled above), follow these steps:

1. In the main SimEvents window, double-click the SimEvents Sinks icon to open the SimEvents Sinks library.

2. Drag the Signal Scope block from the library into the model window. Simulink gives it a unique block name, Signal Scope1, to avoid a conflict with the existing Signal Scope block in the model.

3. Connect the **pe** signal output port of the FIFO Queue block to the **in** signal input port of the Signal Scope1 block by dragging the mouse pointer from one port to the other. The model now looks like the figure below.

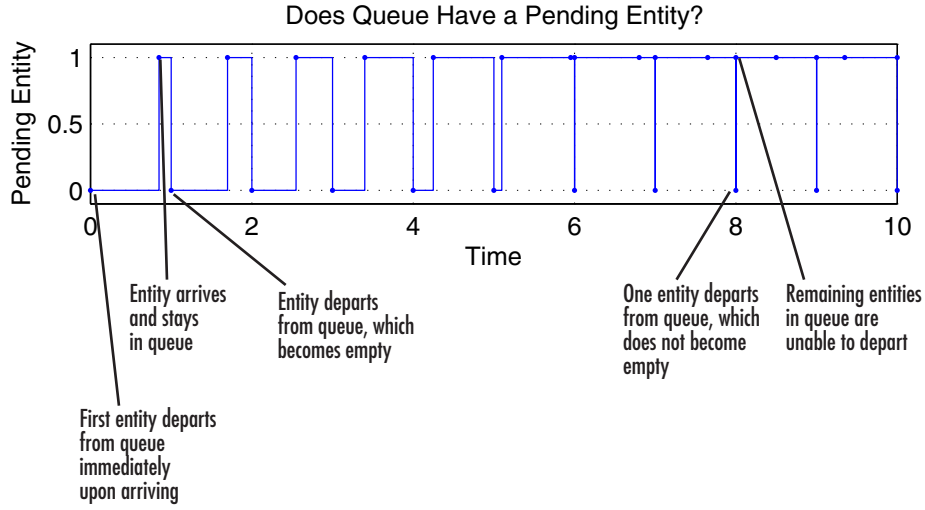


Simulating with Different Intergeneration Times

By changing the intergeneration time (that is, the reciprocal of the entity arrival rate) in the Time-Based Entity Generator block, you can see how the queue's entity output port becomes blocked and unblocked. Try this procedure:

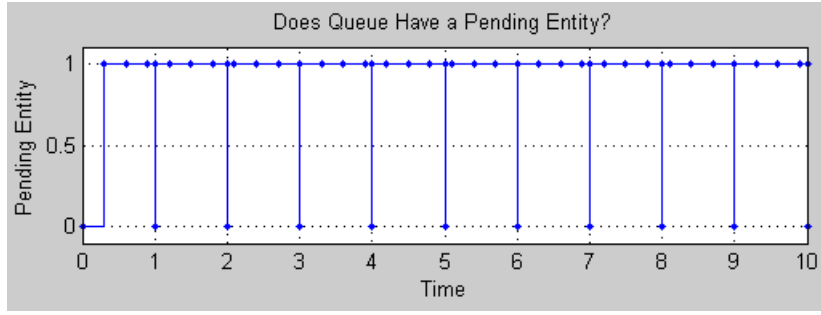
1. Double-click the Time-Based Entity Generator block to open its dialog box, set the **Period** parameter to `0.85`, and click **OK**. This causes entities to arrive somewhat faster than the Single Server block can process them. As a result, the queue is not always empty.

2 Save and run the simulation. The plot whose title bar is labeled Signal Scope1 represents the pending-entity signal, where a value of 1 means true and a value of 0 means false. The figure below explains some of the points on the plot. The vertical range on the plot has been modified to fit the data better.



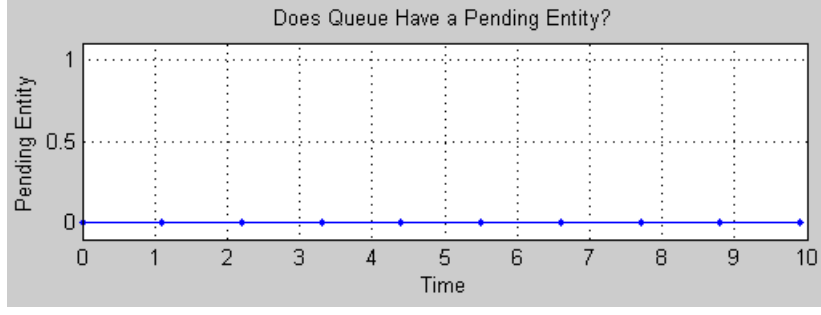
3 Reopen the Time-Based Entity Generator block's dialog box and set **Period** to `0.3`.

4 Run the simulation again. Now the entities arrive much faster than the server can process them. The pending-entity signal is true nearly all the time because the queue nearly always has at least one entity that is trying and failing to advance to the server. However, the pending-entity signal is false at the beginning of the simulation. The pending-entity signal is also false at time instants when an entity in the queue is able to advance to the server; these values of false change back to true when the queue determines that it still has an entity that cannot depart.



5 Reopen the Time-Based Entity Generator block's dialog box and set **Period** to `1.1`.

6 Run the simulation again. Now the entities arrive slower than the server's service rate, so entities never stay in the queue for a positive amount of time. The pending-entity signal is always false because every entity that arrives at the queue is able to depart immediately.



Viewing Waiting Times and Utilization

The pending-entity signal shown above is an example of a statistic that quantifies a state at a particular instant. Other statistics, such as average waiting time and server utilization, summarize behavior between T=0 and the current time. To modify the model so that you can view entities' average waiting time in the queue and server, as well as the proportion of time that the server spends storing an entity, use the following procedure:

1 Double-click the FIFO Queue block to open its dialog box. Click the **Statistics** tab, set the **Average wait** parameter to `On`, and click **OK**. This

causes the block to have a signal output port for the signal representing the average duration that entities wait in the queue. The port label is **w**.

2 Double-click the Single Server block to open its dialog box. Click the **Statistics** tab, set both the **Average wait** and **Utilization** parameters to `On`, and click **OK**. This causes the block to have a signal output port labeled **w** for the signal representing the average duration that entities wait in the server, and a signal output port labeled **util** for the signal representing the proportion of time that the server spends storing an entity.

3 Copy the Signal Scope1 block and paste it into the model window.

Note If you modified the plot corresponding to the Signal Scope1 block, then one or more parameters in its dialog box might be different from the default values. Copying a block also copies parameter values.

---

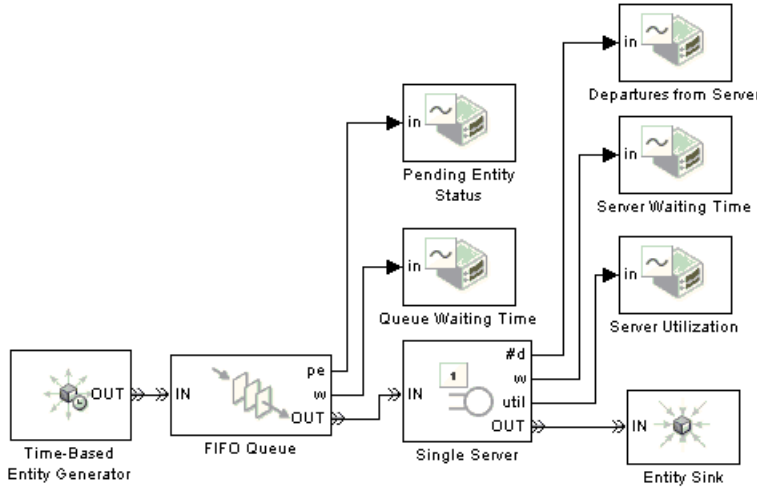
4 Double-click the new copy to open its dialog box.

5 Set **Plot type** to `Continuous` and click **OK**. For summary statistics like average waiting time and utilization, a continuous-style plot is more appropriate than a stairstep plot. Note that the `Continuous` option refers to the appearance of the plot and does *not* change the signal itself to make it continuous-time.

6 Copy the Signal Scope2 block that you just modified and paste it into the model window twice. You now have five scope blocks.

Simulink gives each copy a unique name. If you want to make the model and plots easier to read, you can click the names underneath each scope block and rename the block to use a descriptive name like Queue Waiting Time, for example.

7 Connect the **util** signal output port and the two **w** signal output ports to the **in** signal input ports of the unconnected scope blocks by dragging the mouse pointer from port to port. The model now looks like the figure below.

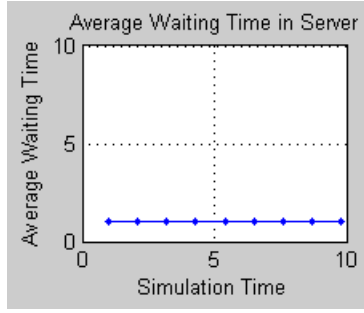


8 Save the simulation and run it with different values of the **Period** parameter in the Time-Based Entity Generator block, as described in "Simulating with Different Intergeneration Times" on page 2-15. Look at the plots to see how they change if you set the intergeneration time to `0.3` or `1.1`, for example.

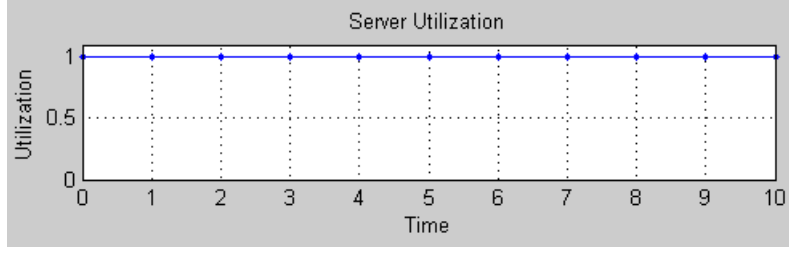
Observations from Plots

- The average waiting time in the server does not change after the first departure from the server because the service time is fixed for all departed

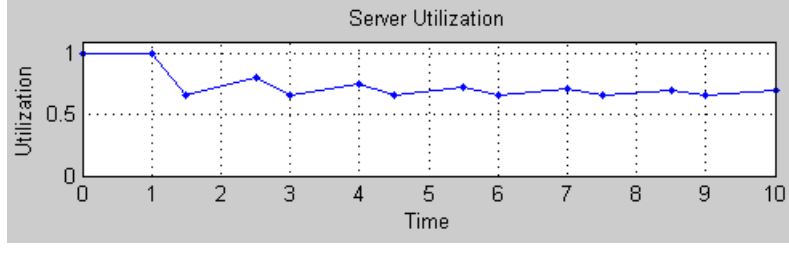
entities. The average waiting time statistic does not include partial waiting times for entities that are in the server but have not yet departed.



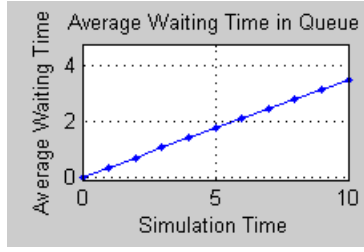
- The utilization of the server is nondecreasing if the intergeneration time is small (such as `0.3`) because the server is constantly busy once it receives the first entity.



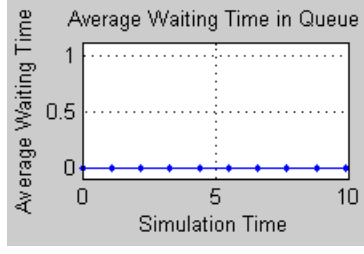
The utilization might decrease if the intergeneration time is larger than the service time (such as `1.5`) because the server has idle periods between entities.



- The average waiting time in the queue increases throughout the simulation if the intergeneration time is small (such as `0.3`) because the queue gets longer and longer.



The average waiting time in the queue is zero if the intergeneration time is larger than the service time (such as `1.1`) because every entity that arrives at the queue is able to depart immediately.



Information About Race Conditions and Random Times

Other examples modify this one by varying the processing sequence for simultaneous events or by making the intergeneration times and/or service times random. The modified examples are

- “Example: Using Random Intergeneration Times in a Queuing System” on page 3-5
- “Example: Using Random Service Times in a Queuing System” on page 4-7
- Event Priorities demo

Building a Simple Hybrid Model

This section describes how to modify a time-based model by adding some discrete-event behavior. The original demo is a model of a flight controller in an aircraft. The modifications are a first step toward simulating a remote flight controller for the same aircraft. The aircraft dynamics are unchanged, but the controller and the aircraft (plant) are separated. A simple way to model a separation is a time delay, which is what this example does. A variation on the example also complicates the interaction between controller and aircraft by modeling occasional transmission failures.

Using the example model, this section shows you how to

- Attach data from time-based dynamics to entities whose timing is independent of the time-based dynamics
- Use an entity's departure event to cause the update of a signal that influences time-based dynamics
- Create a simple model of a hybrid system and then vary it to explore other behaviors

The topics in this section are as follows:

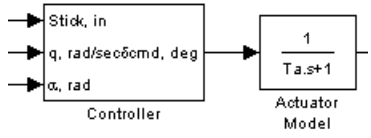
Note More realistic ways to model a remote-control system might involve communication over a shared network, where the time delays and transmission failures might depend on other network traffic. The `f14_control_over_network` demo shows a more complicated model of a remote flight controller.

Opening a Time-Based Simulink Demo

To open the Simulink F-14 demo, enter

```
sldemo_f14
```

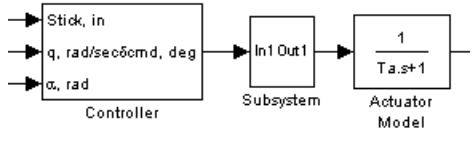
in the MATLAB Command Window. The model simulates the pilot's stick input with a square wave. The system outputs are the aircraft angle of attack and the G forces experienced by the pilot. A model scope displays the input and output signals. The Controller block connects to other components in the model, namely, the stick input, the q and σ signals from the aircraft dynamics model, and the actuator model.



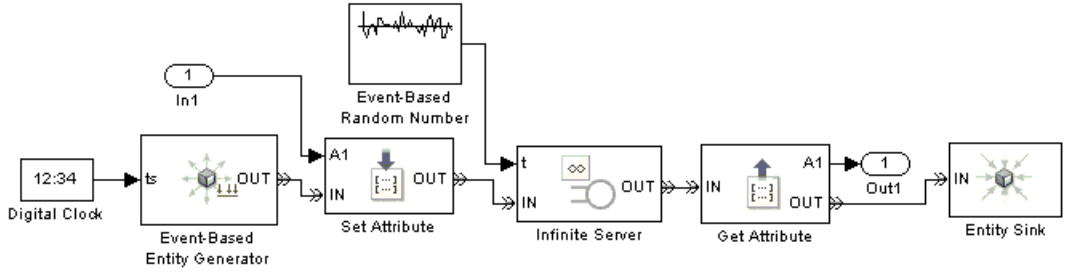
Run the simulation by choosing **Simulation > Start** from the model window's menu. You can view the results graphically in the model scope.

Adding Event-Based Behavior

This section modifies the `sldemo_f14` model by inserting several SimEvents blocks between the Controller and Actuator Model blocks. The result looks like the following figures, where the SimEvents blocks are contained in a subsystem for visual neatness.



Part of Top Level of Modified Model



Subsystem Contents

The following topics describe the subsystem and then provide instructions for building it yourself:

- "Behavior of the Subsystem" on page 2-24
- "How to Build the Subsystem" on page 2-25

Behavior of the Subsystem

The SimEvents blocks are an abstract representation of a simple communication link that samples the information from the remote controller and conveys that information to the aircraft:

- Data from the controller is related to the subsystem via the subsystem's In1 block.
- Periodically, the Event-Based Entity Generator block creates an entity, which serves as a vehicle for the data in this communication system between the controller and the aircraft.
- The Set Attribute block attaches the data to the entity.
- The Infinite Server block models the latency in the communication system by delaying each data-containing entity.
- The Get Attribute block models the reconstruction of data at the receiver. This block connects to the subsystem's Out1 block so that the actuator block at the top level of the model can access the data.
- The Entity Sink block absorbs entities after they are no longer needed.

Note This subsystem models communication from the controller to the actuator, but does not address the feedback path from the aircraft back to the controller. As stated earlier, this model is merely a first step toward modeling a remote controller. Next steps might involve modeling the communication in the feedback path and replacing the Infinite Server block with a more realistic representation of the communication link.

How to Build the Subsystem

To modify the `sldemo_f14` model to create this example, follow these steps:

1. Open Simulink and SimEvents, referring to instructions in "Opening a Model and Libraries" on page 2-2 if you are new to Simulink. Also, open the `sldemo_f14` model by entering its name in the MATLAB Command Window if you have not already done so.

2. Use **File > Save As** in the model window to save the model to your working directory as `sldemo_f14_des.mdl`.

3. Enter `simeventsconfig('sldemo_f14_des','hybrid')` in the MATLAB Command Window to make some model settings more appropriate for a simulation that includes discrete-event behavior.

4. From the Simulink Ports & Subsystems library, drag the Subsystem block into the model window and insert it between the Controller and Actuator Model blocks. The model window should look like Part of Top Level of Modified Model on page 2-23.

5. Double-click the newly inserted Subsystem block to open a subsystem window. The rest of this procedure builds the subsystem in this window.

6. From the Sources library of Simulink, drag the Digital Clock block into the subsystem window.

7. Double-click the Digital Clock block to open its dialog box, set **Sample time** to `0.1`, and click **OK**.

8 From the Entity Generators sublibrary of the Generators library of SimEvents, drag the Event-Based Entity Generator block into the subsystem window.

9 Double-click the Event-Based Entity Generator block to open its dialog box, set **Generate entities upon** to `Sample time hit from port ts`, and click **OK**.

10 From the Signal Generators sublibrary of the Generators library, drag the Event-Based Random Number block into the subsystem window.

11 Double-click the Event-Based Random Number block to open its dialog box. Set **Distribution** to `Uniform`, set **Minimum** to `0.01`, set **Maximum** to `0.06`, and click **OK**.

12 From the Attributes library, drag the Set Attribute and Get Attribute blocks into the subsystem window.

13 Double-click the Set Attribute block to open its dialog box. On the **A1** tab, set **Attribute assignment** to `From signal port A1`, set **Attribute name** to `Data`, and click **OK**. The block acquires a signal input port labeled **A1**.

14 Double-click the Get Attribute block to open its dialog box. On the **A1** tab, set **Send attribute value to signal port A1** to `On`, set **Attribute name** to `Data`, and click **OK**. The block acquires a signal output port labeled **A1**.

15 From the Servers library, drag the Infinite Server block into the subsystem window.

16 Double-click the Infinite Server block to open its dialog box. Set **Service time from** to `Signal port t` and click **OK**. The block acquires a signal input port labeled **t**.

17 From the SimEvents Sinks library, drag the Entity Sink block into the subsystem window.

18 Connect the blocks as shown in Subsystem Contents on page 2-24.

19 Save the model to preserve your modifications.

Running the Hybrid F-14 Simulation

Run the `sldemo_f14_des` simulation by choosing **Simulation > Start** from the model window's menu. By comparing the plots in the model scope with the plots in the original time-based `sldemo_f14` model, you can see how the discrete-event behavior affects the simulation. The latency in the control loop (that is, the delay between the controller and the actuator) degrades the behavior somewhat.

Changing the Latency

One way to experiment with the simulation is to change the latency in the control loop (that is, the delay between the controller and the actuator) and run the simulation again. Here are some ideas:

- In the Event-Based Random Number block, set **Maximum** to `0.1`.
- In the Event-Based Random Number block, set **Distribution** to `Beta`, set **Minimum** to `0.01`, and set **Maximum** to `0.06`.
- Replace the Event-Based Random Number block with a Step block from the Simulink Sources library. In the latter block's dialog box, set **Step time** to `30`, **Initial value** to `0.03`, **Final value** to `0.07`, and **Sample time** to `1`.

Visualizing the Sampling and Latency

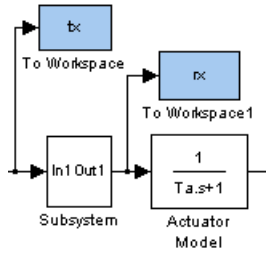
By sending relevant data from `sldemo_f14_des` to the MATLAB workspace and examining it after the simulation, you can determine when Simulink updates signals during the simulation. In particular, you can confirm that the data sent from the controller to the actuator is, in fact, delayed.

To send the controller's output and actuator's input to the MATLAB workspace and compare the signals after the simulation, follow these steps:

1. From the Simulink Sinks library, drag two copies of the To Workspace block into the top-level model window.
2. Double-click one To Workspace block to open its dialog box, set **Variable name** to `tx`, set **Limit data points to last** to `Inf`, set **Save format** to `Structure With Time`, and click **OK**.

3 Double-click the other To Workspace block to open its dialog box, set **Variable name** to `rx`, set **Limit data points to last** to `Inf`, set **Save format** to `Structure With Time`, and click **OK**.

4 Connect the To Workspace blocks to the input and output signals to the discrete event subsystem using branch lines, as shown below.

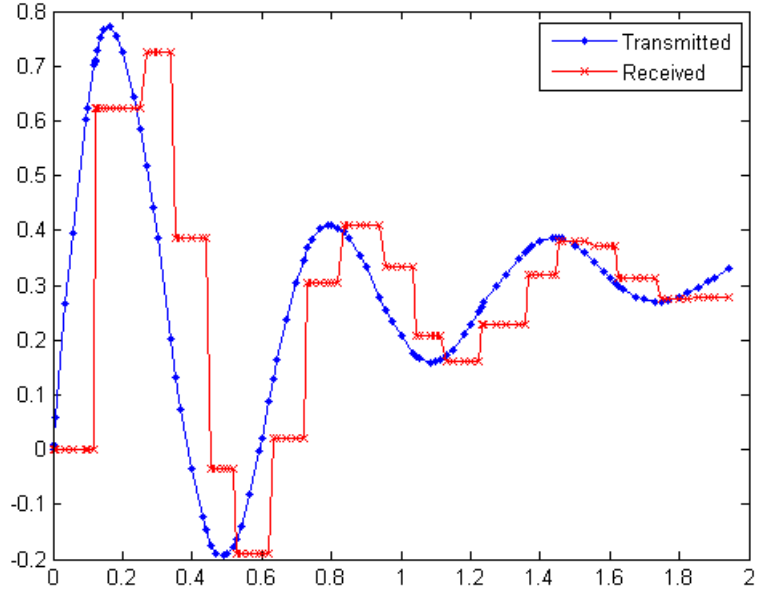


5 Run the simulation.

6 Enter the following in the MATLAB Command Window:

```
n = 100; % Plot first 100 values
plot(tx.time(1:n), tx.signals.values(1:n),'b.-',...
     rx.time(1:n), rx.signals.values(1:n),'rx-');
legend('Transmitted','Received')
```

The resulting plot, shown below, exhibits the data sampling and the delay in the discrete event subsystem. The data transmitted by the controller appears with blue dots, while the data received at the actuator appears with red x's. Notice that the data transmitted at T=0.1 is received slightly later and then held constant until the data transmitted at T=0.2 is received. The time points 0, 0.1, 0.2, 0.3, etc., are significant because the subsystem generates an entity at these times and it is the entities that carry data from the controller to the actuator.



Transmitted and Received Data

Event-Based and Time-Based Dynamics in the Simulation

In the `sldemo_f14_des` model, the time-based dynamics of the aircraft coexist with the event-based dynamics of the communication link. When you run this simulation, the ODE solver and an event calendar both play a role. The ODE solver simulates the time-based dynamics of the aircraft. Solving the event-based dynamics entails scheduling and processing events, such as service completion and entity generation, on the event calendar. The events representing service completion and entity generation are asynchronous and unrelated to the time-based simulation steps used in solving the ordinary differential equations of the aircraft.

In this model, time-based blocks interact with event-based blocks at the input and output of the Subsystem block. At each of its sample times, the Controller block updates the value at the input port of the Subsystem block. The Set Attribute block, which is event-based, uses the value upon the next entity arrival at the block. Such entity arrivals occur at times 0, 0.1, 0.2, etc., and explain why the value of each received data point in the plot, Transmitted and Received Data on page 2-29, is the value of the transmitted signal at one of the times 0, 0.1, 0.2, etc. The received data does not reflect values transmitted at other times.

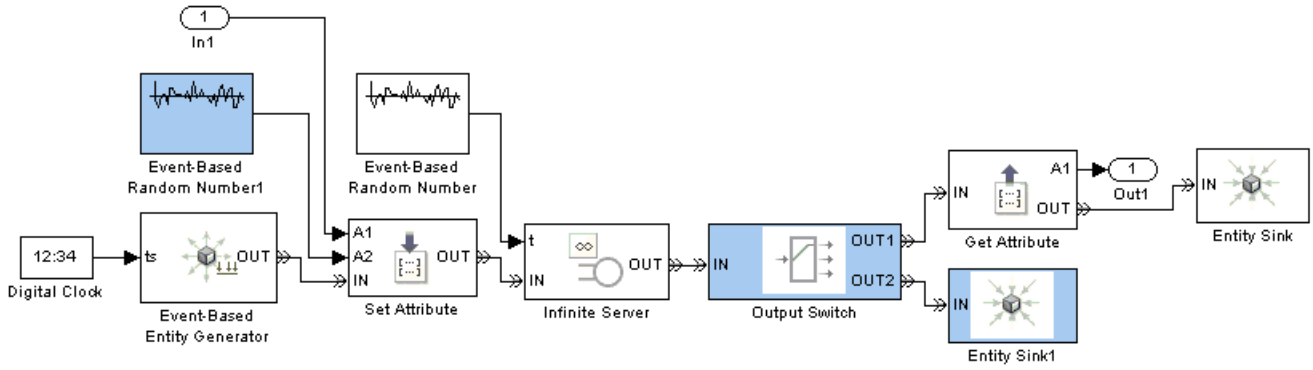
When an entity completes its service, the entity arrives at the Get Attribute block, which is event-based. This block updates the value at the output port of the subsystem. The Actuator Model block, which is time-based, uses the value upon the next time step determined by the ODE solver.

To learn more about the event calendar and the ODE solver, see “Working with Events” online, and “Simulating Dynamic Systems” in the Simulink documentation.

Modifying the Model to Drop Some Messages

You can vary the implementation of the `sldemo_f14_des` model’s remote communication from the controller to the actuator by having the communication link drop messages with small probability. When a message is dropped, the actuator continues to use the last received message, until the next time it gets an updated message.

The modified portion of the subsystem looks like the figure below.



Subsystem Modified to Drop Some Messages

The following topics describe the subsystem modifications and then provide instructions for building them yourself:

- "Behavior of the Modified Subsystem" on page 2-31
- "How to Modify the Subsystem" on page 2-32

Behavior of the Modified Subsystem

In the original subsystem, every entity (with data attached to it) reaches the Get Attribute block, which sends the data out of the subsystem and to the actuator. In the modified subsystem,

- The Set Attribute block assigns not only the `Data` attribute but also a new `DropMessage` attribute. The value of the `DropMessage` attribute is 1 with probability 0.95 and 2 with probability 0.05. The values 1 and 2 refer to the entity output ports on the Output Switch block.
- Entities advance to either the Get Attribute block or a new Entity Sink block. The Output Switch block uses the `DropMessage` attribute of each entity to determine which path that entity takes. Because of the probability distribution, 95% of entities advance to the Get Attribute block and the remaining 5% of entities are absorbed by the Entity Sink block.

- When an entity reaches the Get Attribute block, the attached data successfully reaches the actuator. When an entity uses the other path, the attached data is discarded and the actuator continues to see the data that it received from the last entity that reached the Get Attribute block.

The reason the actuator continues to see previous data is that Simulink holds a signal's value until instructed to update it. When an entity is absorbed without reaching the Get Attribute block, the block does not update the signal that goes to the subsystem's Out1 block. Therefore, the value of that signal is whatever value was attached to the last entity that reached the Get Attribute block during the simulation.

The same reasoning explains why the actuator sees a constant signal between successive entities, that is, between successive samples by the communication link. Although the controller issues a continuous-time signal, the communication link between the controller and actuator creates a new data-carrying entity according to a discrete-time schedule. In other words, the subsystem samples the data from the controller before transmitting it to the actuator.

How to Modify the Subsystem

To modify the subsystem in the `sldemo_f14_des` model to create this variation, follow these steps:

1. From the Routing library, drag the Output Switch block into the subsystem window.

2. Double-click the Output Switch block to open its dialog box. Set **Number of entity output ports** to `2`, set **Switching criterion** to `From attribute`, set **Attribute name** to `DropMessage`, and click **OK**. The block retains two entity output ports, labeled **OUT1** and **OUT2**.

3. Create copies of the Event-Based Random Number and Entity Sink blocks, which are already in the subsystem. You can create a copy by dragging the block with the right mouse button, or by using **Edit > Copy** followed by **Edit > Paste**.

4. Double-click the newly copied Event-Based Random Number block (labeled Event-Based Random Number1) to open its dialog box. Set **Distribution** to `Arbitrary discrete`, set **Value vector** to `[1 2]`, set **Probability**

vector to `[0.95 0.05]`, set **Initial seed** to an odd 5-digit number different from the one used in the other instance of this block, and click **OK**.

5. Double-click the Set Attribute block to open its dialog box. On the **A2** tab, set **Attribute assignment** to `From signal port A2`, set **Attribute name** to `DropMessage`, and click **OK**. The block acquires a signal input port labeled **A2**.

6. Delete the connection between the Infinite Server and Get Attribute blocks.

7. Connect the blocks as shown in Subsystem Modified to Drop Some Messages on page 2-31.

8. Use **File > Save As** in the model window to save the model to your working directory as `sldemo_f14_des_drop.mdl`.

Key SimEvents Concepts

The examples in this chapter illustrate various concepts that are important to the way SimEvents works. This section reviews and reinforces those concepts. The topics are

Meaning of Entities in Different Applications

An entity represents an item of interest in a discrete-event simulation. The meaning of an entity depends on what you are modeling. In this chapter, examples use entities to represent abstract customers in a queuing system and instructions from a remote controller to an actuator on the system being controlled.

Entities do not have a graphical depiction in the model window the way blocks, ports, and connection lines do.

Entity Ports and Paths

An entity output port provides a way for an entity to depart from a block . An entity input port provides a way for an entity to arrive at a block.

A connection line indicates a path along which an entity can potentially advance. However, the connection line does not imply that any entities actually advance along that path during a simulation. For a given entity path and a given time instant during the simulation, any of the following could be true:

- No entity is trying to advance along that path.
- An entity is trying and failing to advance along that path. For some blocks, it is normal for an entity input port to be unavailable under certain conditions. This unavailability causes an entity to fail in its attempt to advance along that path, even though the path is intact (that is, even though the ports are connected). An entity that tries and fails to advance is called a pending entity.

- An entity successfully advances along that path. This occurs only at a discrete set of times during a simulation.

Note The simulation could also have one or more times at which one or more entities successfully advance along a given entity path and, simultaneously, one or more different entities try and fail to advance along that same entity path. For example, an entity departs from a queue and, simultaneously, the next entity in the queue tries and fails to depart.

Data and Signals

In time-based dynamics, signals express the outputs of dynamic systems represented by blocks. Event-based blocks can also read and produce signals. One way to learn about signals is to plot them; the discussion in "Creating Additional Plots" on page 2-14 is about visualizing signals that reflect behavior of event-based blocks.

Time-based and event-based dynamics can interact via the data shared by both types of blocks. Attributes of entities provide a way for entities to carry data with them. The subsystem in "Adding Event-Based Behavior" on page 2-23 illustrates the use of attributes in the interaction between time-based and event-based dynamics.

Although signals are common to both time-based and event-based dynamics, event-based dynamics can produce signals that have slightly different characteristics. For more information, see "Working with Signals" online.

3

Creating Entities Using Intergeneration Times

Role of Entities in SimEvents Models

As described in “What Is an Entity?” on page 1-6, entities are discrete items of interest in a discrete-event simulation. You determine what an entity signifies, based on what you are modeling.

Data and Entities

You can optionally attach data to entities. Such data is stored in one or more *attributes* of an entity. You define names and numeric scalar values for attributes. For example, if your entities represent a message that you are transmitting across a communication network, you might assign data called `length` that indicates the length of each particular message. You can read or change the values of attributes during the simulation.

Creating Entities in a Model

SimEvents models typically contain at least one source block that generates entities. Other SimEvents blocks in the model process the entities that the source block generates. One source block that generates entities is the Time-Based Entity Generator block, described in “Introduction to the Time-Based Entity Generator” on page 3-3.

Varying the Interpretation of Entities

A single model can use entities to represent different kinds of items. For example, if you are modeling a factory that processes two different kinds of parts, then you can

- Use two Time-Based Entity Generator blocks to create the two kinds of parts.
- Use one Time-Based Entity Generator block and subsequently assign an attribute to indicate what kind of part each entity represents.

Introduction to the Time-Based Entity Generator

The Time-Based Entity Generator block creates entities. You configure the Time-Based Entity Generator block to customize aspects such as

- The intergeneration times between successive entities. The sections below discuss ways of doing this.
- How the block reacts when it is temporarily unable to output entities. To learn more, see the block's online reference page.
- The relative priority of entity generation events compared to other kinds of events that might occur simultaneously. To learn more, see "Processing Sequence for Simultaneous Events" online.

Accessing the Time-Based Entity Generator

The Time-Based Entity Generator block resides in the Entity Generators sublibrary of the Generators library of SimEvents.

Specifying the Distribution of Intergeneration Times

The intergeneration time is the time interval between successive entities that a Time-Based Entity Generator block generates. You can use the block's dialog box to describe a statistical distribution that governs the intergeneration times. Use this procedure:

1. Set **Generate entities with** to `Intergeneration time from dialog`.
2. Choose a statistical distribution by setting the **Distribution** parameter to one of these values:
   - `Constant`. Then set the **Period** parameter to the constant intergeneration time.

     Note If potential roundoff errors in entity generation times are a concern in your model, then consider using the Event-Based Entity Generator block instead. Set the block's **Generate entities upon** parameter to `Sample time hit from port ts` and connect an input signal whose sample time is the desired constant intergeneration time. See "Detecting Sample Time Hits" online for details.

   - `Uniform`. Then set the **Minimum** and **Maximum** parameters to define the interval over which the distribution is uniform. The uniform distribution has probability density function

$$f(x) = \begin{cases} \dfrac{1}{\textbf{Maximum} - \textbf{Minimum}} & \textbf{Minimum} < x < \textbf{Maximum} \\ 0 & \text{Otherwise} \end{cases}$$

   - `Exponential`. Then set the **Mean** parameter to the mean of the exponential distribution. The exponential distribution with mean 1/λ has probability density function

$$f_\lambda(x) = \begin{cases} \lambda \exp(-\lambda x) & x \geq 0 \\ 0 & x < 0 \end{cases}$$

The random distributions also provide an **Initial seed** parameter that specifies the seed on which the stream of random numbers is based. Typically, you would use a large (for example, five-digit) odd number. For a fixed seed, the random behavior is repeatable the next time you run the simulation. Changing the seed changes the stream of random numbers.

Example: Using Random Intergeneration Times in a Queuing System

Open the model that you created in "Building a Simple Discrete-Event Model" on page 2-2. By examining the Time-Based Entity Generator block's **Distribution** and **Period** parameters, you can see that the block is configured to use a constant intergeneration time of 1 second. To use a random intergeneration time instead, try these variations and see how they affect the plot that the simulation creates:

- Set **Distribution** to `Uniform`, set **Minimum** to `1`, and set **Maximum** to `3`. The first entity, generated at T=0, appears in the plot at T=1 after its service is complete. The second entity, generated at a random time between T=1 and T=3, appears in the plot between T=2 and T=4.
- Set **Distribution** to `Uniform`, set **Minimum** to `1`, and set **Maximum** to `1.5`. The plot probably shows more entities compared to the scenario above because the range of intergeneration times has the same minimum but a smaller maximum.
- Set **Distribution** to `Exponential` and set **Mean** to `0.5`. This system is called an M/D/1 queuing system, where the M stands for Markovian and indicates a Poisson arrival rate. Note that the exponential distribution has no upper bound, so the time between successive entities could be any positive number.

Using Intergeneration Times from a Signal

To indicate intergeneration times explicitly as values from a signal, use this procedure:

1. Set the Time-Based Entity Generator block's **Generate entities with** parameter to `Intergeneration time from port t`. A signal input port labeled **t** appears on the block.
2. Create a signal whose value at each generation time is the time until the next entity generation.

   For examples of how to create such signals, see

   - "Example: Using a Step Function as Intergeneration Time" on page 3-7
   - "Example: Using an Arbitrary Discrete Distribution as Intergeneration Time" on page 3-9.
3. Connect the signal to the signal input port labeled **t**.

Upon generating each entity, the Time-Based Entity Generator block reads the value of the input signal and uses that value as the time interval until the next entity generation.

Using intergeneration times from a signal might be appropriate if you

- Want to use a statistical distribution that is not directly accessible using the `Intergeneration time from dialog` option, described in "Specifying the Distribution of Intergeneration Times" on page 3-4.
- Want the intergeneration time to depend on the dynamics of other blocks in your model.
- Have a set of intergeneration times in a MATLAB workspace variable or in a MAT-file.

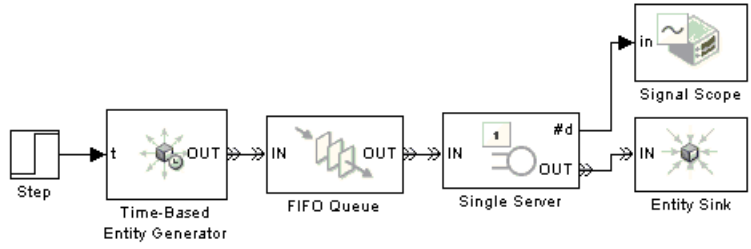
Note The block reads the input signal upon each entity generation, not upon each simulation sample time, so signal values that occur between successive entity generation events have no effect on the entity generation process. For example, if the input signal is `10` when the simulation starts, `1` at T=1, and `10` from T=9 until the simulation ends, then the value of `1` never becomes an intergeneration time.

Example: Using a Step Function as Intergeneration Time

Open the model that you created in "Building a Simple Discrete-Event Model" on page 2-2. To specify intergeneration times using a signal, use this procedure:

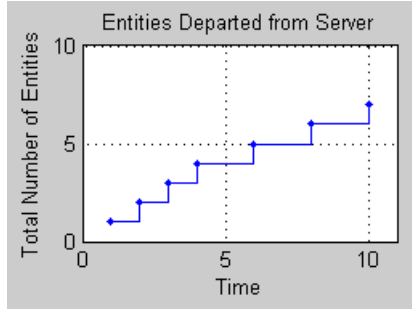
1. Set the Time-Based Entity Generator block's **Generate entities with** parameter to `Intergeneration time from port t`. A signal input port labeled **t** appears on the block.

2. From the Simulink Sources library, drag a Step block into the model and connect it to the **t** input port of the Time-Based Entity Generator block. The model looks like the figure below.



3. Set the Step block's **Step time** parameter to `2.8`, **Initial value** parameter to `1`, and **Final value** parameter to `2`. With these parameters, the block generates a signal whose value is 1 from T=0 to T=2.8, and whose value is 2 thereafter.

4 Run the simulation. You can see from the plot that the entities departing from the server are initially spaced 1 second apart and later spaced 2 seconds apart.



The Time-Based Entity Generator block reads intergeneration times from the Step block each time it generates an entity. The table below shows when the Time-Based Entity Generator block generates entities and which intergeneration time values it reads in each instance. The table also shows when each entity departs from the server, which you can see from the plot. Although the Step block starts producing the value of 2 at T=2.8, the Time-Based Entity Generator block does not read the new value until the next time it generates an entity, at T=3.

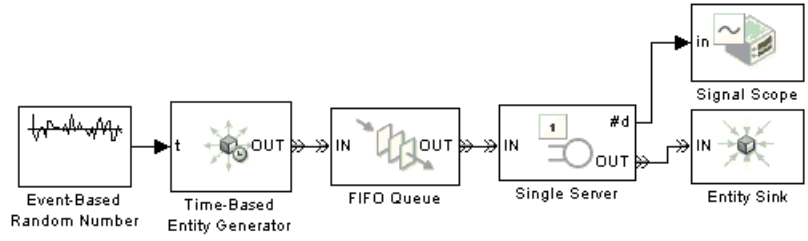
Entity Generation Time	Intergeneration Time Until Next Entity Generation	Departure Time of Entity from Server
0	1	1
1	1	2
2	1	3
3	2	4
5	2	6
7	2	8
9	2	10

Example: Using an Arbitrary Discrete Distribution as Intergeneration Time

Open the model that you created in "Building a Simple Discrete-Event Model" on page 2-2. To specify intergeneration times using a signal, use this procedure:

1. Set the Time-Based Entity Generator block's **Generate entities with** parameter to `Intergeneration time from port t`. A signal input port labeled **t** appears on the block.

2. From the Signal Generators sublibrary of the Generators library, drag the Event-Based Random Number block into the model and connect it to the **t** input port of the Time-Based Entity Generator block. The model looks like the figure below.

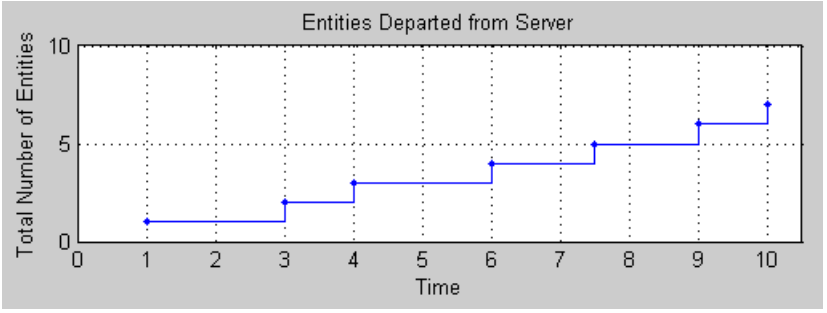


3. Set the Event-Based Random Number block's **Distribution** parameter to `Arbitrary discrete`, **Value vector** parameter to `[1 1.5 2]`, and **Probability vector** parameter to `[0.25 0.5 0.25]`. With these parameters, the block generates intergeneration times Δt such that

   $P(\Delta t = 1) = 0.25$
   $P(\Delta t = 1.5) = 0.5$
   $P(\Delta t = 2) = 0.25$

4. Run the simulation. You can see from the plot that the entities departing from the server are spaced 1, 1.5, or 2 seconds apart. The simulation

time in this example is much too short to verify that the random number generator is applying the specified probabilities, however.



4

Basic Queues and Servers

Role of Queues in SimEvents Models

In a discrete-event simulation, a queue stores entities for some length of time that cannot be determined in advance. The queue attempts to output entities as soon as it can, but its success depends on whether the next block accepts new entities. An everyday example of a queue is a situation where you stand in a line with other people to wait for someone (a bank teller, a retail cashier, etc.) to address your needs and you cannot determine in advance how long you must wait.

Distinguishing features of different queues include

- The queue capacity, which is the number of entities the queue can store simultaneously
- The queue discipline, which determines which entity departs first if the queue stores multiple entities

Physical Queues and Logical Queues

In some cases, a queue in a model is similar to an analogous aspect of the real-world system being modeled. This kind of queue is sometimes called a *physical queue*. For example, you might use a queue to represent a sequence of

- People standing in line
- Airplanes waiting to access a runway
- Messages waiting to be sent
- Parts waiting to be assembled in a factory
- Computer programs waiting to be executed

In other cases, a queue in a model does not arise in an obvious way from the real-world system but instead is included for modeling purposes. This kind of queue is sometimes called a *logical queue*. For example, you might use a queue to provide a temporary storage area for entities that might otherwise have nowhere to go. Such use of a logical queue can prevent deadlocks or simplify the simulation. For example, see “Example of a Logical Queue” on page 4-12.

Accessing Queue Blocks

Queue blocks reside in the Queues library of SimEvents. This chapter focuses on the FIFO Queue block; for more information about other blocks in the library, see “Modeling Queues and Servers” online.

Although queuing theory typically treats a queue-server pair as one component, SimEvents contains queue blocks and server blocks as distinct components. You often attach a queue block directly to a server block, but you might also want to use the blocks in other ways.

Role of Servers in SimEvents Models

In a discrete-event simulation, a server stores entities for some length of time, called the *service time*, and then attempts to output the entity. During the service period, the block is said to be *serving* the entity that it stores. An everyday example of a server is a person (a bank teller, a retail cashier, etc.) with whom you perform a transaction with a projected duration.

The service time for each entity is computed when it arrives, which contrasts with the inherent unknowability of the storage time for entities in queues. If the next block does not accept the arrival of an entity that has completed its service, however, then the server is forced to hold the entity longer.

Distinguishing features of different servers include

- The number of entities it can serve simultaneously, which could be finite or infinite
- Characteristics of, or the method of computing, the service times of arriving entities
- Whether the server permits certain arriving entities to preempt entities that are already stored in the server

---

Tip In the absence of preemption, a finite-capacity server does not accept new arrivals when it is already full. You can place a queue before each finite-capacity server, establishing a place for entities to stay while they are waiting for the server to accept them. Otherwise, the waiting entities might be stored in various different locations in the model and the situation might be more difficult for you to predict or analyze.

What Servers Represent

In some cases, a server in a model is similar to an analogous aspect of the real-world system being modeled. For example, you might use a server to represent

- A person (such as a bank teller) who performs a transaction with each arriving customer
- A transmitter that processes and sends messages
- A machine that assembles parts in a factory
- A computer that executes programs

You might use an infinite-capacity server to represent a delaying mechanism. An example of this is in the subsystem in "Building a Simple Hybrid Model" on page 2-22.

Servers Inserted for Modeling Purposes

In some cases, a server in a model does not arise in an obvious way from the real-world system but instead is included for modeling purposes. A common modeling technique involves a delay of duration zero, that is, an infinite server whose service time is zero, either to break an algebraic loop or to provide a place for an entity to reside while a preceding block updates its output signals. For details and examples, see "Interleaved Operations of Storage and Nonstorage Blocks" and "Loops in Entity Paths Without Storage Blocks" online.

Accessing Server Blocks

Server blocks reside in the Servers library of SimEvents. This chapter focuses on the Single Server block; for more information about other blocks in the library, see "Modeling Queues and Servers" online.

Using FIFO Queue and Single Server Blocks

The example in "Building a Simple Discrete-Event Model" on page 2-2 illustrates how to

- Create a queue-server pair using the FIFO Queue and Single Server blocks
- View statistics from the queue and server blocks, such as blockage of the queue block's entity output port and utilization of the server

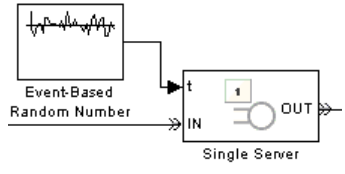
This section discusses additional aspects of the queue and server blocks, in these topics:

Varying the Service Time

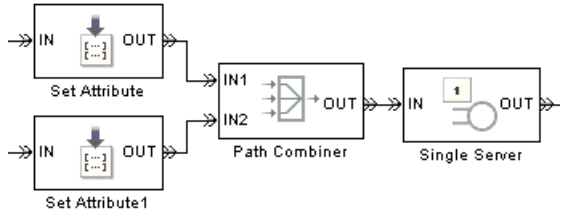
The subsystem described in "Adding Event-Based Behavior" on page 2-23 includes an Infinite Server block that serves each entity for a random amount of time. The random duration is the value of a signal that serves as an input to the Infinite Server block. Similarly, the Single Server block can read the service time from a signal, which enables you to vary the service time for each entity that arrives at the server.

Some scenarios in which you might use a varying service time are as follows:

- You want the service time to come from a random number generator. In this case, set the Single Server block's **Service time from** parameter to `Signal port t` and use the Event-Based Random Number block to generate the input signal for the Single Server block. Be aware that some distributions can produce negative numbers, which are not valid service times.



- You want the service time to come from data attached to each entity. In this case, set the Single Server block's **Service time from** parameter to `Attribute` and set **Attribute name** to the name of the attribute containing the service time. An example is in the figure below.



  To learn more about attaching data to entities, see "Setting Attributes of Entities" online.

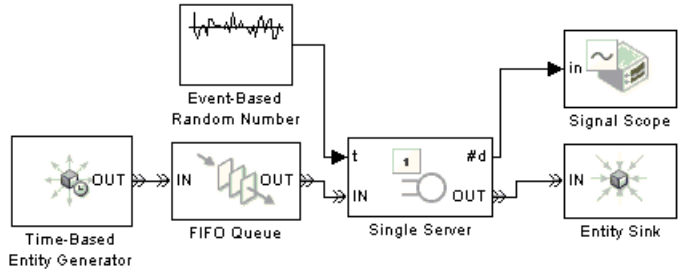
- You want the service time to arise from dynamics of the simulation. In this case, set the Single Server block's **Service time from** parameter to `Signal port t` and create a signal whose value at the time an entity arrives at the server is equal to the desired service time for that entity.

Example: Using Random Service Times in a Queuing System

Open the model that you created in "Building a Simple Discrete-Event Model" on page 2-2. By examining the Single Server block's **Service time from** and **Service time** parameters, you can see that the block is configured to use a constant service time of 1 second. To use a random service time instead, follow these steps:

1. Set **Service time from** to `Signal port t`. This causes the block to have a signal input port labeled **t**.

2. From the Signal Generators sublibrary of the Generators library, drag the Event-Based Random Number block into the model window and connect it to the Single Server block's signal input port labeled **t**.



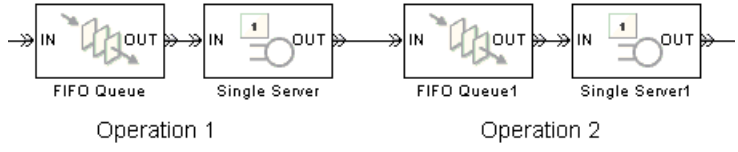
3 Run the simulation and note how the plot differs from the one corresponding to constant service times (shown in "Results of the Simulation" on page 2-13).

Constructs Involving Queues and Servers

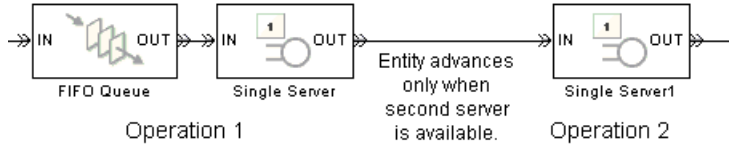
Here are some examples of ways to combine FIFO Queue and Single Server blocks to model different situations:

Serial Queue-Server Pairs

Two queue-server pairs connected in series represent successive operations that an entity undergoes. For example, parts on an assembly line are processed sequentially by two machines.

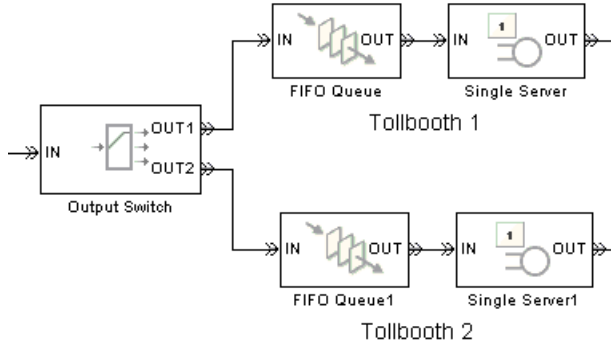


While you might alternatively model the situation as a pair of servers without a queue between them, the absence of the queue means that if the first server completes service on an entity before the second server is available, the entity must stay in the first server past the end of service and the first server cannot accept a new entity for service until the second server becomes available.



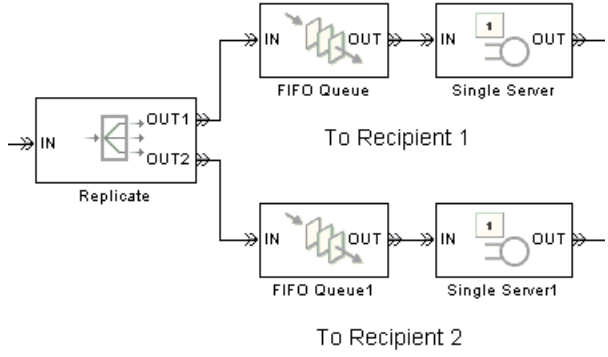
Parallel Queue-Server Pairs as Alternatives

Two queue-server pairs connected in parallel, in which each entity arrives at one or the other, represent alternative operations. For example, vehicles wait in line for one of several tollbooths at a toll plaza.



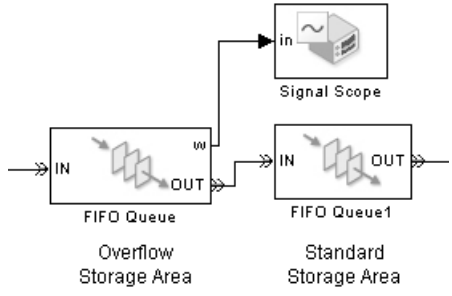
Parallel Queue-Server Pairs in Multicasting

Two queue-server pairs connected in parallel, in which a copy of each entity arrives at both, represent a multicasting situation such as sending a message to multiple recipients. Note that copying entities might not make sense in some applications.



Serial Connection of Queues

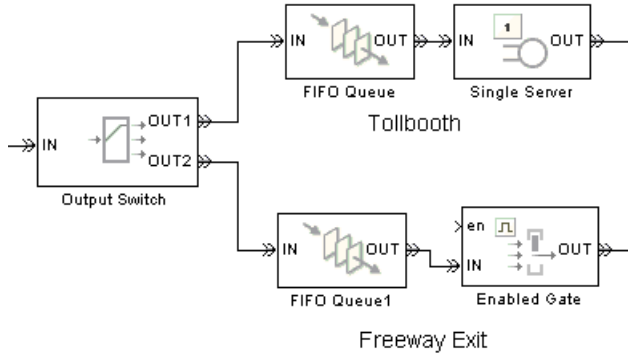
Two queues connected in series might be useful if you are using entities to model items that physically experience two distinct sets of conditions while in storage. For example, additional inventory items that overflow one storage area have to stay in another storage area in which a less well-regulated temperature affects the items' long-term quality. Modeling the two storage areas as distinct queue blocks facilitates viewing the average length of time that entities stay in the overflow storage area.



A similar example is in “Example of a Logical Queue” on page 4-12, except that the example there does not suggest any physical distinction between the two queues.

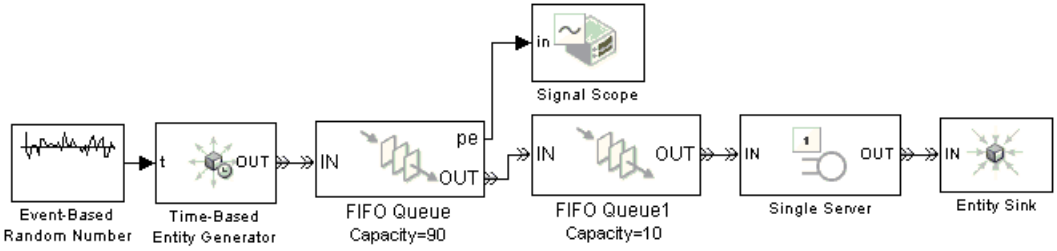
Parallel Connection of Queues

Two queues connected in parallel, in which each entity arrives at one or the other, represent alternative paths for waiting. The paths might lead to different operations, such as a line of vehicles waiting for a tollbooth modeled and a line of vehicles waiting on a jammed exit ramp of the freeway. You might model the tollbooth as a server and the traffic jam as a gate.



Example of a Logical Queue

Suppose you are modeling a queue that can physically hold 100 entities and you want to determine what proportion of the time the queue length exceeds 10. You can model the long queue as a pair of shorter queues connected in series. The shorter queues have length 90 and 10.



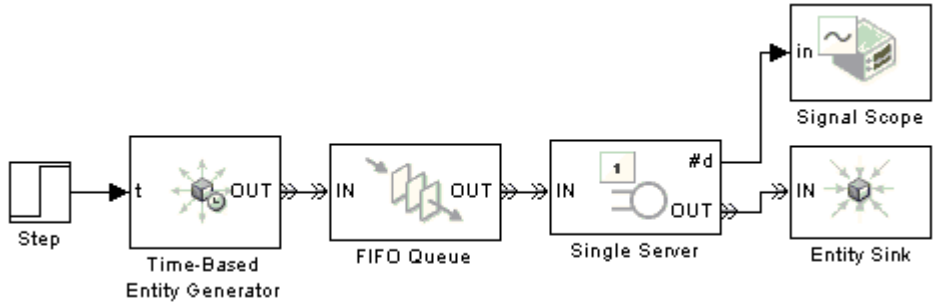
Although the division of the long queue into two shorter queues has no basis in physical reality, it enables you to gather statistics specifically related to one of the shorter queues. In particular, you can view the pending-entity status signal of the queue having length 90. If the signal is 1 over a nonzero time interval, then the length-90 queue contains an entity that cannot advance to the length-10 queue. This means that the length-10 queue is full. As a result, the physical length-100 queue contains more than 10 items. Determining the proportion of time the physical queue length exceeds 10 is equivalent to determining the proportion of time the pending-entity status signal of the logical length-90 queue equals 1.

5

Designing Paths for Entities

Role of Paths in SimEvents Models

An entity path is a connection from an entity output port to an entity input port, depicted as a line connecting the entity ports of two SimEvents blocks. An entity path represents the equivalence between an entity's departure from the first block and arrival at the second block. For example, in the model shown below, any entity that departs from the FIFO Queue block's **OUT** port equivalently arrives at the Single Server block's **IN** port.

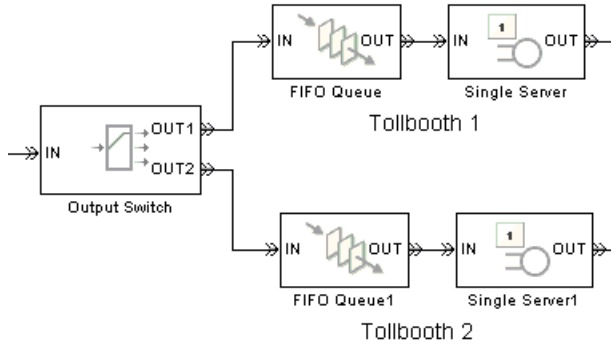


The existence of the entity path does not guarantee that any entity actually uses the path; for example, the simulation could be so short that no entities are ever generated. Even when an entity path is used, it is used only at a discrete set of times during the simulation.

Implications of Entity Paths

In some models, you can use the entity connection lines to infer the full sequence of blocks that a given entity arrives at, throughout the simulation.

In many discrete-event models, however, the set of entity connection lines does not completely determine the sequence of blocks that each entity arrives at. For example, the model below shows two queues in a parallel arrangement, preceded by a block that has one entity input port and two entity output ports.



By looking at the entity connection lines alone, you cannot tell which queue block's **IN** port an entity will arrive at. Instead, you need to know more about how the one-to-two block (Output Switch) behaves and you might even need to know the outcome of certain run-time decisions.

Overview of Routing Library for Designing Paths

You design entity paths by choosing or combining entity paths using some of the blocks in the Routing library of SimEvents. These blocks have extra entity ports that let you vary the model's topology (that is, the set of blocks and connection lines).

Typical reasons for manipulating entity paths are

- To describe an inherently parallel behavior in the situation you are modeling — for example, a computer cluster with two computers that share the computing load. You can use the Output Switch block to send computing jobs to one of the two computers. You might also use the Path Combiner or Input Switch block if computing jobs share a common destination following the pair of computers.
- To design nonlinear topologies, such as feedback loops — for example, repeating an operation if quality criteria such as quality of service (QoS)

are not met. You can use the Path Combiner block to combine the paths of new entities and entities that require a repeated operation.

- To incorporate logical decision making into your simulation — for example, determining scheduling protocols. You might use the Input Switch block to determine which of several queues receives attention from a server.

Using the Output Switch

The Output Switch block in the Routing library selects one among a number of entity output ports. The selected port can change during the simulation. You have several options for criteria that the block uses to select an entity output port.

When the selected port is not blocked, an arriving entity departs through this port. When the selected port is blocked, however, the entity cannot arrive at the Output Switch block, even if an unselected entity output port is not blocked.

Sample Use Cases

Here are some scenarios in which you might use an output switch:

- Entities advance to one of several queues based on efficiency or fairness concerns. For example, airplanes advance to one of several runways depending on queue length, or customers advance to the first available cashier out of several cashiers.

  Comparing different approaches to efficiency or fairness, by testing different rules to determine the selected output port of the output switch, might be part of your goal in simulating the system.

- Entities advance to a specific destination based on their characteristics. For example, parcels advance to one of several delivery vehicles based on the locations of the specified recipients.

- Entities use an alternate route in case the preferred route is blocked. For example, a communications network drops a packet if the route to the transmitter is blocked and the simulation gathers statistics about dropped packets.

The topics listed below illustrate the use of the Output Switch block.

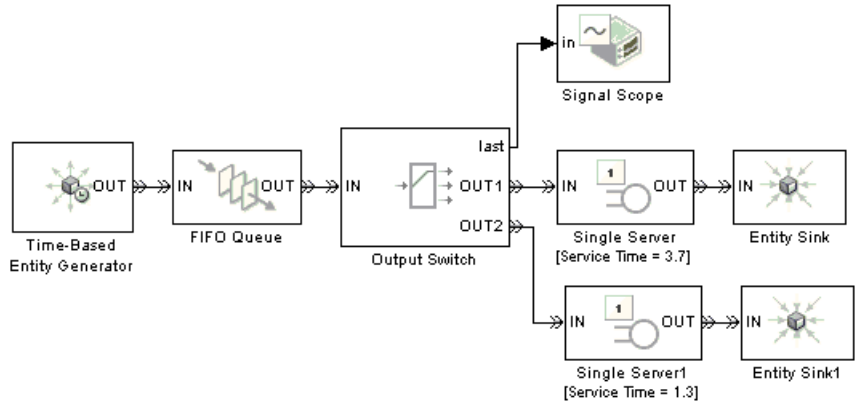
Topic	Features of Example
"Example: Selecting the First Available Server" on page 5-6	`First port that is not blocked` switching criterion
"Example: Using an Attribute to Select an Output Port" on page 5-8	Attribute-based switching, where the attribute value is random
"Example: A Packet Switch" on page 5-16	Attribute-based switching in conjunction with a Path Combiner block
"Example: Choosing the Shortest Queue" online	Switching based on an event-based computation
"Example: Using Servers in Shifts" online	Switching based on a time-based computation

To learn about all supported switching criteria, see the online reference page for the Output Switch block.

Example: Selecting the First Available Server

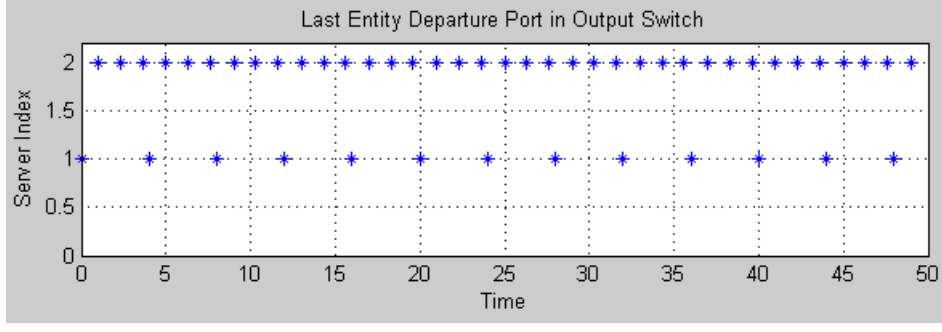
In this example, entities arriving at the Output Switch block depart through the first entity output port that is not blocked, as long as at least one entity output port is not blocked. An everyday example of this approach is a single queue of people waiting for service by one of several bank tellers, cashiers, call center representatives, etc. Each person in the queue wants to advance as soon as possible to the first available service provider without preferring one over another.

You can implement this approach by setting the **Switching criterion** parameter in the Output Switch block to `First port that is not blocked`.



This deterministic model creates one entity every second and attempts to advance the entity to one of two servers. The two servers have different service times, both greater than 1 second. The server with the longer service time becomes available less frequently and has a smaller throughput. The FIFO Queue block stores entities while both servers are busy. After any server becomes available, an entity in the queue advances to the Output Switch, which outputs that entity to that server.

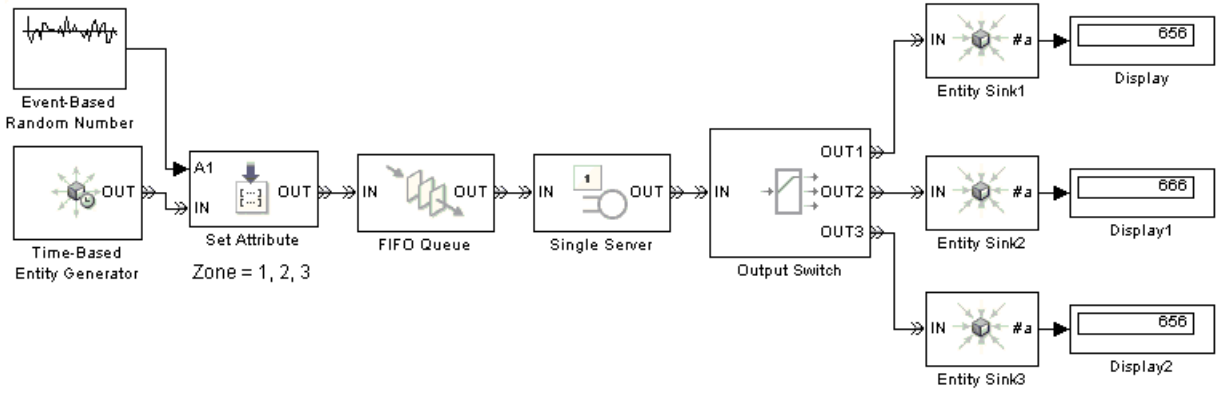
The Output Switch block also outputs a signal containing the index of the entity output port through which the most recent entity departure occurred. The Signal Scope block plots the values of this signal. You can see from the plot that, compared to the first server, the second server processes more entities because its service time is shorter.



Example: Using an Attribute to Select an Output Port

Consider the situation in which parcels are sorted among several delivery vehicles based on the locations of the specified recipients. If each parcel is an entity, then you can attach data to each entity to indicate the location of its recipient. To implement the sorting, set the **Switching criterion** parameter in the Output Switch block to `From attribute`.

The model shown below illustrates the sorting process (but not the delivery process itself), partitioning the delivery area into three geographic zones. An entity generator represents sources of parcels addressed to one of the zones. After being marked with a randomly chosen zone 1, 2, or 3 via the Set Attribute block, the parcels advance to the queue to wait for sorting. The Single Server block models the small delay incurred in the sorting process and sends each parcel through the Output Switch block to one of three entity output ports. From there, the example merely counts the sorted entities destined for each zone, but your own simulation might do something interesting with the outputs from the switch.



Using the Input Switch

The Input Switch block in the Routing library chooses among a number of entity input ports. This block selects exactly one entity input port for potential arrivals and makes all other entity input ports unavailable. The selected entity input port can change during the simulation. You have several options for criteria that the block uses for selecting an entity input port.

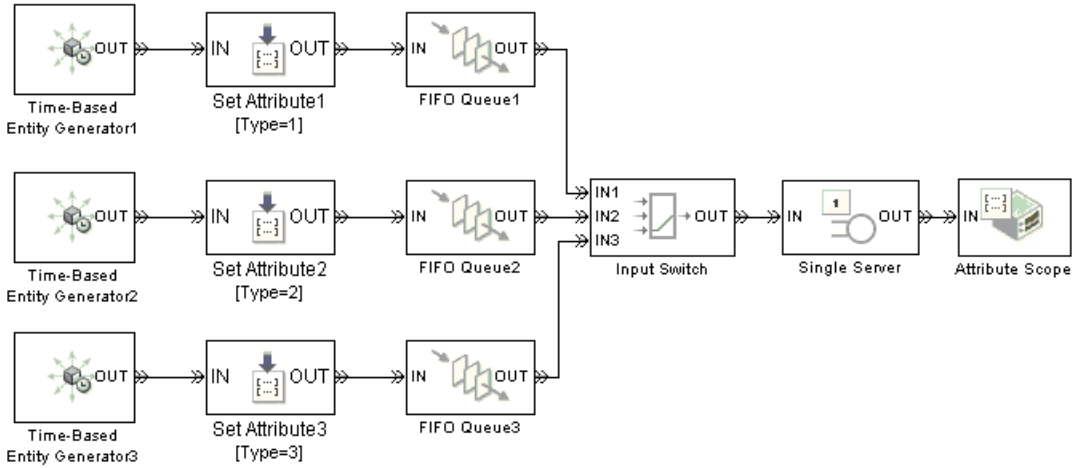
A typical scenario in which you might use an input switch is when multiple sources of entities feed into a single queue, where the sequencing follows specific rules. For example, users of terminals in a time-shared computer submit jobs to a queue that feeds into the central processing unit, where an algorithm regulates access to the queue so as to prevent unfair domination by any one user.

Example: Round-Robin Approach to Choosing Inputs

In a round-robin approach, an input switch cycles through the entity input ports in sequence. After the last entity input port, the next selection is the first entity input port. The switch selects the next entity input port after each entity departure. When the switch selects an entity input port, it makes the other entity input ports unavailable, regardless of how long it takes for an entity to arrive at the selected port.

You can implement a round-robin approach by setting the **Switching criterion** parameter in the Input Switch block to `Round robin`.

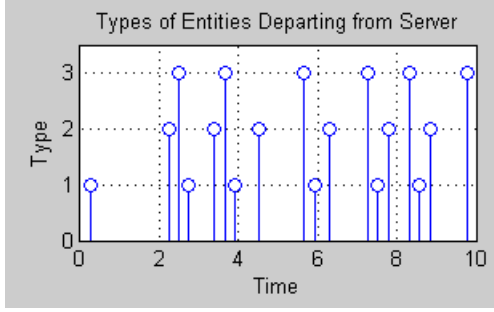
Consider the following model, in which three sets of entities attempt to arrive at an Input Switch block with the round-robin switching criterion.



The three Set Attribute blocks assign a `Type` attribute to each entity, where the attribute value depends on which entity generator created the entity. FIFO Queue blocks store entities that cannot enter the Input Switch block yet because either

- The Input Switch is waiting to receive an entity at a different entity input port, according to the round-robin switching criterion.
- The Single Server block is busy serving an entity, so its entity input port is unavailable.

The Attribute Scope block creates a a stem plot of the `Type` attribute values over time. Because the `Type` attribute identifies the source of each entity that arrives at the scope, you can see the effect of the round-robin switching criterion. In particular, the heights of the stems in the plot cycle among the values 1, 2, and 3.

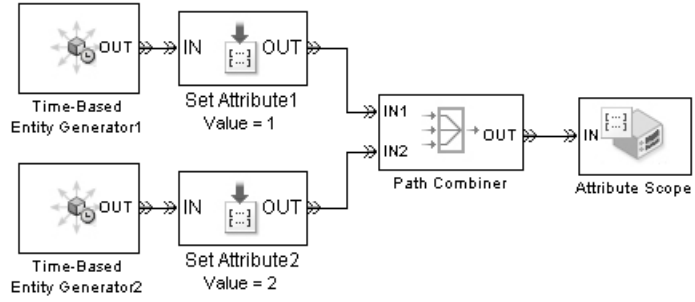


Combining Entity Paths

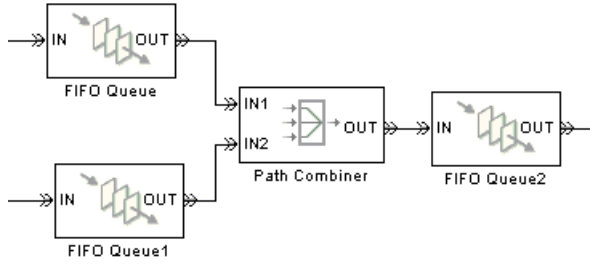
You can merge multiple paths into a single path using the Path Combiner block. Merging entity paths does not change the entities themselves, just as merging lanes on a road does not change the vehicles that travel on it. In particular, the Path Combiner block does not create aggregates or batches.

Here are some scenarios in which you might combine entity paths:

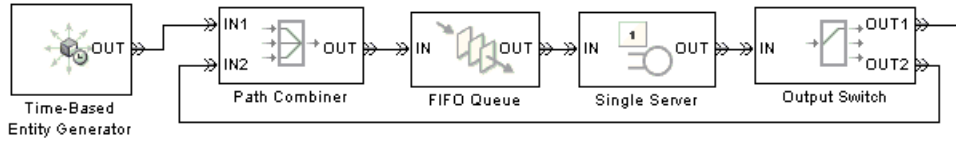
- Multiple entity generator blocks create entities having different values for a particular attribute. The entities then follow a merged path but might be treated differently later based on their individual attribute values.



- Multiple queues merge into a single queue. (You might also use an Input Switch block for this, depending on the desired sequencing of entities in the single queue.)



- A feedback path enters the same queue as an ordinary path.



Sequencing Simultaneous Pending Arrivals

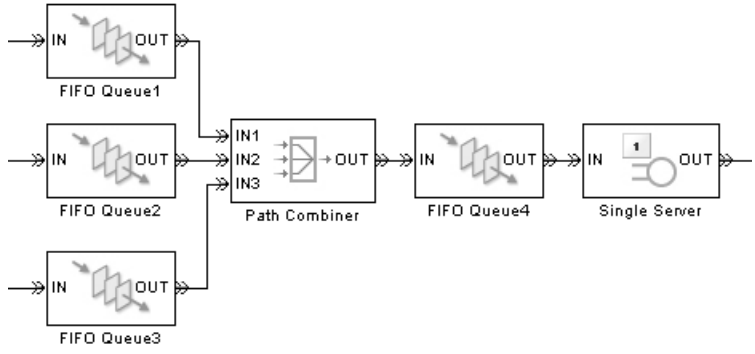
The Path Combiner block does not experience any collisions, even if multiple entities attempt to arrive at the same time. The categories of behavior are as follows:

- If the entity output port is not blocked when the entities attempt to arrive, then the sequence of arrivals depends on the sequence of departure events from blocks that precede the Path Combiner block. Although the departure time is the same for all such entities, the sequence might affect the system's behavior. For example, if the entities advance to a queue, the departure sequence determines their positions in the queue.
- If pending entities are waiting to advance to the Path Combiner block when its entity output port changes from blocked to unblocked, then the entity input ports are notified of the change sequentially. The change from blocked to unblocked means that an entity can advance to the Path Combiner block.

  If at least two entities are waiting to advance to the Path Combiner block via distinct entity input ports, then the notification sequence is important because the first port to be notified of the change is the first to advance an entity to the Path Combiner block. The **Input port precedence** parameter determines which of the block's entity input ports is first in the notification sequence. For the list of options, see the online reference page for the Path Combiner block.

Example: Significance of Input Port Precedence

Consider the sequence of blocks in the figure below, in which a Path Combiner block merges three small queues into a single large queue.



Suppose the server is busy serving an entity, the single large queue (FIFO Queue4) is full, and each of the three small queues is nonempty. In this situation, the Path Combiner block's entity output port is blocked. When the entity in the server departs, an entity from the large queue advances to the server. The large queue is no longer full, so its entity input port becomes available. As a result, the Path Combiner block's entity output port changes from blocked to unblocked. The Path Combiner block uses the **Input port precedence** parameter to determine the sequence by which to notify entity input ports of the change. The sequence of notifications determines which of the three small queues is the first to advance an entity to the large queue via the Path Combiner block.

The **Input port precedence** parameter is relevant only when the entity output port changes from blocked to unblocked; the parameter is irrelevant in other situations. For example, if the large queue has infinite capacity, or if at most one of the three small queues is nonempty at any given time during the entire simulation, then all settings for the **Input port precedence** produce the same behavior.

Difference Between Path Combiner and Input Switch

The Input Switch block, described in "Using the Input Switch" on page 5-9, has multiple entity input ports and one entity output ports. The same is true for the Path Combiner block. These two blocks differ in that

- The Path Combiner block's acceptance of an entity arrival does not depend on which port the entity arrives at. By contrast, the Input Switch block accepts only those entities that arrive at the block's selected entity input port.
- The Path Combiner block's **Input port precedence** parameter is relevant only in the specific situation described in "Input Port Precedence" online. By contrast, the Input Switch block's **Switching criterion** parameter governs the block's behavior throughout the simulation.

When deciding whether to use a Path Combiner or Input Switch block in your model, consider how you want the simulation to behave when one source of entities is dormant for a long time but another source of entities is not. If you want the routing block to wait until an entity finally departs from the dormant source, then an Input Switch block would be more appropriate. If you want the routing block to accept arrivals from other entity sources that are not dormant, then a Path Combiner block would be more appropriate.

Example: A Packet Switch

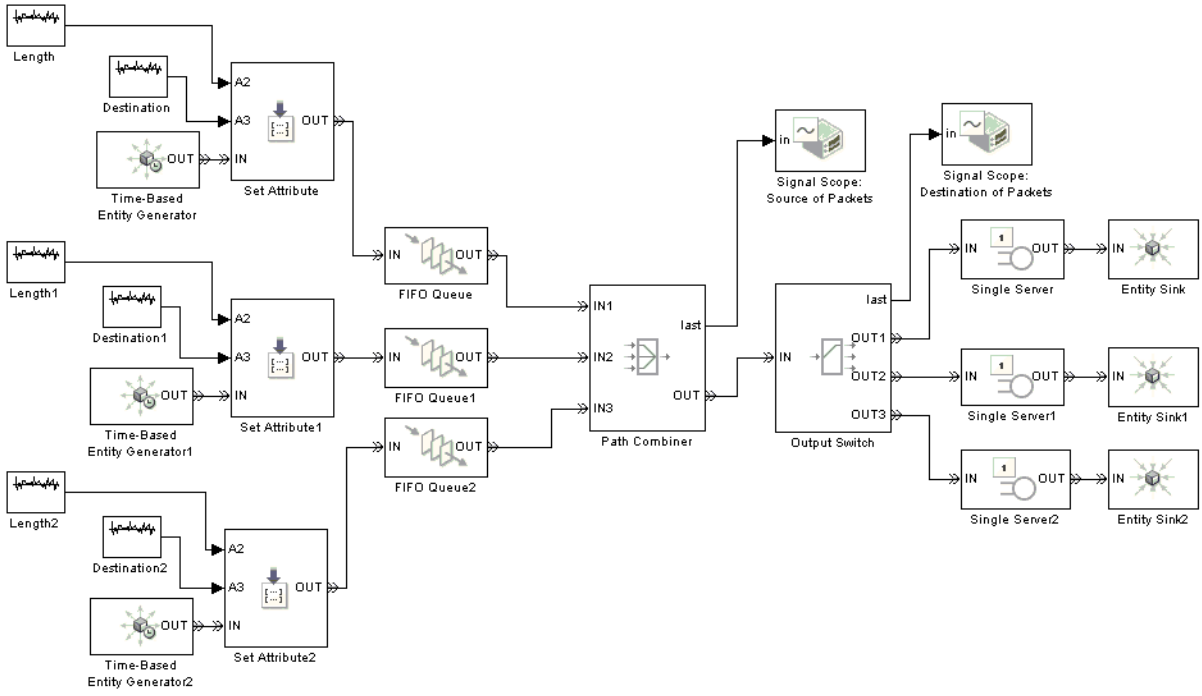
A packet switch is a component of a communication network that connects several sources of data packets with several destinations. The switch directs each packet to the correct destination, processing packets as they arrive. The switch must be able to handle the situation in which several data packets have the same arrival time and the same destination. Packet switches can use a variety of architectures and algorithms, and this section describes how to construct one particular packet switch model.

The goal in this example is to construct a switch that

- Connects three data sources to three destinations
- Holds arriving packets in a buffer (that is, a queue) for each of the data sources
- Randomly resolves contention if two or more simultaneous packets at the head of their respective queues share the same intended destination, with no bias to any particular source of packets

The example model accommodates variable-length packets and assumes that packets from each data source have exponential interarrival times.

The figure below shows an overview of the block diagram.



The following sections describe portions of the model in detail:

- “Generating Packets” on page 5-18
- “Storing Packets in Input Buffers” on page 5-19
- “Routing Packets to Their Destinations” on page 5-20
- “Connecting Multiple Queues to the Output Switch” on page 5-20
- “Modeling the Channels” on page 5-21

Generating Packets

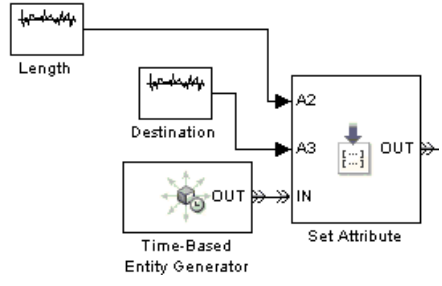
The packet switch example models each packet as an entity. The Time-Based Entity Generator block creates entities. To implement exponentially distributed intergeneration times between successive entities from each source, the block has its **Distribution** parameter set to `Exponential`.

Attached to each entity are these pieces of data, stored in attributes:

- The source of the packet, an integer between 1 and 3
- The destination of the packet, a random integer between 1 and 3
- The length of the packet, a random integer between 6 and 10

Note The entity does not actually carry a payload because this example models the transmission of data at a level of abstraction that includes timing and routing behaviors but that is not concerned with the specific user data in each packet.

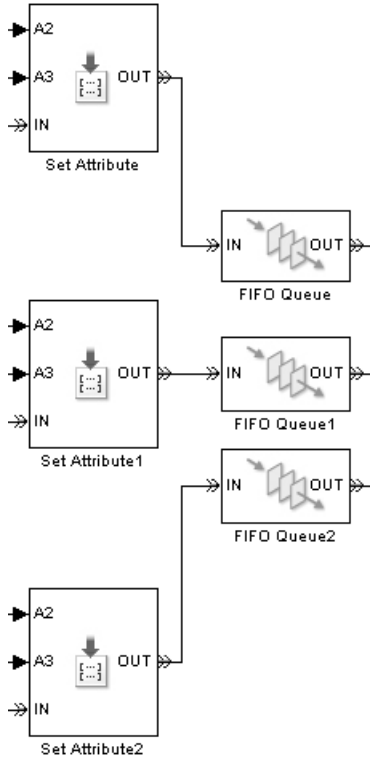
Copies of the Event-Based Random Number block produce the random destination and length data. The Set Attribute block attaches all the data to each entity. The Set Attribute block is configured so that the destination and length come from input signals, while the source number comes from a constant in the dialog box.



The packet generation processes for the different sources differ only in the initial seeds for the random number generators and the values for the source attribute.

Storing Packets in Input Buffers

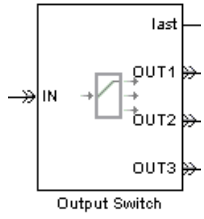
The packet switch example uses one FIFO Queue block as a buffer following each data source's Set Attribute block.



The queue uses a FIFO queuing discipline, which does not take into account the destination of each packet. Note that such a model can suffer from "head-of-queue blocking," which occurs when a packet *not* at the head of the queue is forced to wait even when its destination is available, just because the packet at the head of the queue is aiming for an unavailable destination.

Routing Packets to Their Destinations

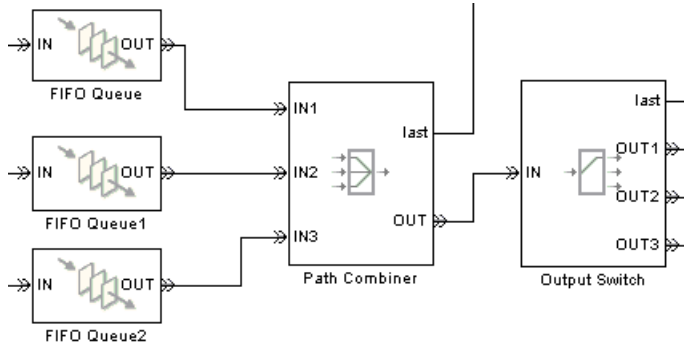
A core block in the packet switch example is the Output Switch block. This block sorts arriving entities so that they depart at the appropriate entity output port based on the entities' `Destination` attribute.



This part of the example is similar to the model shown in "Example: Using an Attribute to Select an Output Port" on page 5-8.

Connecting Multiple Queues to the Output Switch

The packet switch model needs a way for entities to advance from the three queues to the single entity input port of the Output Switch block. Candidate blocks are Input Switch and Path Combiner. The Path Combiner block is more appropriate because it processes entities as they arrive from any of the entity input ports, whereas the Input Switch block would restrict arrivals to a specific selected entity input port.



Contention among packets can occur in these ways:

- Multiple packets from different sources with the same intended destination arrive simultaneously at an *empty* queue and immediately attempt to arrive at the path combiner.

  Although the arrivals occur at the same simulation time value, the processing sequence depends on

  - The priorities of the entity generation events. In this example, all Time-Based Entity Generator blocks share the same **Generation event priority** parameter value.
  - The **Execution order of simultaneous events** parameter in the model's Configuration Parameters dialog box. In this example, the parameter is set to `Randomize`.

  As a result, when two packets are generated at the same time, the sequence of generation events in this example is random.

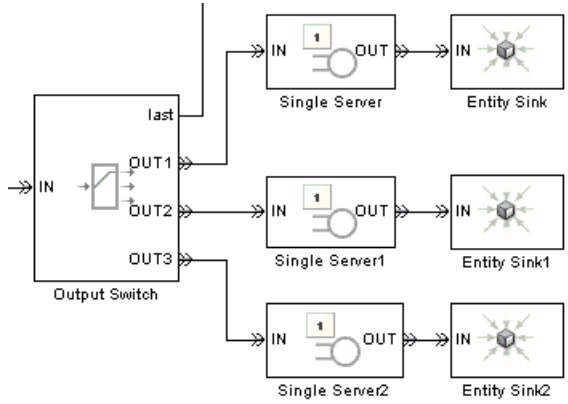
- Multiple packets with the same intended destination are at the head of their respective queues precisely when the Path Combiner block's entity output port changes from blocked to unblocked.

  For example, suppose all of the queues have leading packets destined for the first server, which is busy serving an earlier packet. The Path Combiner block's entity output port is blocked. When the server completes service on the earlier packet, the Path Combiner block's entity output port becomes unblocked. At that moment, the Path Combiner block notifies its entity input ports of the change in status, in a sequence determined by the **Input port precedence** parameter. In this example, the parameter is set to `Equiprobable`. As a result, when packets waiting at the head of their queues have the same intended destination that changes from unavailable to available, the sequence in which these packets are selected for advancement is random.

Modeling the Channels

The packet switch examples does not model the channel in detail. The channel's key purpose is to process one packet at a time, for a duration that depends on the length of the packet. During processing, other packets bound for the same destination must wait, which introduces resource contention into the simulation.

Each channel is modeled as a Single Server block that delays each entity by an amount of time stored in the entity's `Length` attribute. Each destination is modeled as an Entity Sink block.



6

Selected Bibliography

[1] Banks, Jerry, John Carlson, and Barry Nelson, *Discrete-Event System Simulation*, Second Ed., Upper Saddle River, N.J., Prentice-Hall, 1996.

[2] Cassandras, Christos G., *Discrete Event Systems: Modeling and Performance Analysis*, Homewood, Illinois, Irwin and Aksen Associates, 1993.

[3] Cassandras, Christos G., and Stéphane Lafortune, *Introduction to Discrete Event Systems*, Boston, Kluwer Academic Publishers, 1999.

[4] Fishman, George S., *Discrete-Event Simulation: Modeling, Programming, and Analysis*, New York, Springer-Verlag, 2001.

[5] Gordon, Geoffery, *System Simulation*, Second Ed., Englewood Cliffs, N.J., Prentice-Hall, 1978.

[6] Kleinrock, Leonard, *Queueing Systems, Volume I: Theory*, New York, Wiley, 1975.

[7] Law, Averill M., and W. David Kelton, *Simulation Modeling and Analysis*, 3rd Ed., New York, McGraw-Hill, 1999.

[8] Moler, C., "Floating points: IEEE Standard unifies arithmetic model," Cleve's Corner, The MathWorks, Inc., 1996. You can find this article at http://www.mathworks.com/company/newsletter/clevescorner/

[9] Watkins, Kevin, *Discrete Event Simulation in C*, London, McGraw-Hill, 1993.

[10] Zeigler, Bernard P., Herbert Praehofer, and Tag Gon Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, Second Ed., San Diego, Academic Press, 2000.

Index

U

W