



Figure 9-4. Fixture Required for Entity Transfer

Of course, you could include all kinds of embellishments for this activity. Suppose that only a portion of the entities need this fixture. We could easily include a Decide module between them to check for this condition, as shown in Figure 9-5.

Our two examples using the Allocate and Move modules both resulted in the transporter being moved to the location of the entity. Now let's assume that whenever your transporter is freed, you want it to be roving around the system looking for work. You have a predefined path that the transporter should follow. This path would be very similar to a sequence of stations, except it would form a closed loop. Each time the unallocated transporter reached the next station on its path, it would check to see if there is a current request from somewhere. If there is no request, the transporter continues on its mindless journey. If there is a request, the transporter proceeds immediately to the location of that request.

To model this, you would create a single entity (let's call it the loop entity) that would attempt to allocate the transporter with a very low priority (high number for the priority). Once allocated, the empty transporter is moved to the next station on its path. Upon arrival at this station, the transporter is freed so it can respond to any current request. The loop entity is directed to the initial Allocate module where it once again tries to allocate the transporter. This assumes that any current request for the transporter has a higher priority than the loop entity. Remember that the default priority is 1, with the lowest value having the highest priority.

Once a transporter arrives at its destination, it must be freed. The Free module provides this function. You only need to enter the transporter name in the dialog box, and in some cases, the unit number. Two additional modules, Halt and Activate, allow you to control the number of active or available transporters. The Halt module causes a single transporter unit to become inactive or unavailable to be allocated. The Activate module causes an inactive transporter to become active or available.

9.3 Entity Reneging

9.3.1 Entity Balking and Reneging

Entity *balking* occurs when an arriving entity or customer does not join the queue because there is no room, but goes away or goes someplace else. Entity *reneging* occurs when an entity or customer joins a queue on arrival but later decides to jump out and



Figure 9-5. Checking for Fixture Requirement

leave, probably regretting not having balked in the first place. Let's consider the more complex case where it's possible to have both customer balking and customer reneging. First, let's define the various ways that balking and reneging can occur.

In most service systems, there is a finite system capacity based on the amount of waiting space. When all the space is occupied, a customer will not be able to enter the system and will be balked. This is the simplest form of entity balking. Unfortunately, most systems where balking can occur are much more complicated. Consider a simple service line where, theoretically, the queue or waiting-line capacity is infinite. In most cases, there is some finite capacity based on space, but a service line could possibly exit a building and wind around the block (say the waiting line for World Series or Super Bowl tickets). In these cases, there is no concrete capacity limit, but the customers' decisions to enter the line or balk from the system are based on their own evaluation of the situation. Thus, balking point or capacity is often entity-dependent. One customer may approach a long service line, decide not to wait, and balk from the system—yet the next customer may enter the line.

Entity reneging is an even more complicated issue in terms of both modeling and representation in software. Each entity or customer entering a line has a different tolerance threshold with respect to how long to wait before leaving the line. The decision often depends on how much each customer wants the service being provided. Some customers won't renege regardless of the wait time. Others may wait for a period of time and then leave the line because they realize that they won't get served in time to meet their needs. Often the customers' decisions to remain or leave the line are based on both the amount of time they have already spent in the line as well as their current place in the line. A customer may enter a line and decide to wait for ten minutes; if he is not serviced by that time, he plans to leave the line. However, after ten minutes have elapsed, the customer may choose to stay if he is the next in line for service.

Line switching, or *jockeying*, is an even more complicated form of reneging that often occurs in supermarket checkout lines, fast-food restaurants, and banks that do not employ a single waiting line. A customer selects the line to enter and later re-evaluates that decision based on current line lengths. After all, we invariably enter the slowest-moving line, and if we switch lines, the line we left speeds up and the line we enter slows down. We won't cover the logic for jockeying.

9.3.2 Model 9-3: A Service Model with Balking and Reneging

Let's look at balking and reneging in the context of a very simple model. Customers arrive with $EXPO(5)$ interarrival times at a service system with a single server—service time is $EXPO(4.25)$. All times are in minutes. Although the waiting line has an infinite capacity, each arriving customer views the current length of the line and compares it to his *tolerance* for waiting. If the number in the line is greater than his tolerance, he'll balk away from the system. We'll represent the customer-balking tolerance by generating a sample from a triangular distribution, $TRIA(3, 6, 15)$. Since our generated sample is from a continuous distribution, it will not be an integer. We could use one of the Arena math functions to convert it to an integer, but we're only interested if the number in the waiting line is greater than the generated tolerance.

There are two ways to model the balking activity. Let's assume that we create our arrivals, generate our tolerance value from the triangular distribution, and assign this value to an entity attribute. We could send our arrival to a Decide module and compare our sample value to the current number in the waiting line, using the Arena variable NQ. If the tolerance is less than or equal to the current number in queue, we balk the arrival from the system. Otherwise, we enter the waiting line. An alternative method is to assign our tolerance to a variable and also use this same variable for the server queue capacity. We then send our arrival directly to the server. If the current number in the queue is greater than or equal to the tolerance, which is equal to the queue capacity, the arrival will be balked automatically. Using the second method, it's possible for the queue capacity to be assigned a value less than the current number in queue. This method works because Arena checks only the queue-capacity value when a new entity tries to enter the queue. Thus, the current entities remain safely in the queue, regardless of the new queue-capacity value. We will use this second method when we develop our model.

To represent reneging, assume that arriving customers who decide not to balk are willing to wait only a limited period of time before they renege from the queue. We will generate this renege tolerance time from an ERLA(15, 2) distribution (Erlang), which has a mean of 30, and assign it to an entity attribute in an Assign module. Modeling the mechanics of the reneging activity can be a challenge. If we allow the arrival to enter the queue and the renege time is reached, we need to be able to find the entity and remove it from the queue. At this point in our problem description, you might want to consider alternative methods to handle this. For example, we could define a variable that keeps track of when the server will next be available. We generate the entity-processing time first and assign it to an attribute in an Assign module. We then send our entity to the Decide module where we check for balking. If the entity is not balked, we then check (in the same Decide module) to see if the entity will begin service before its renege time. If not, we renege the entity. Otherwise, we send the entity to an Assign module where we update our variable that tells us when the server will become available, then send the entity to the queue. This model logic may seem complicated, but can be summarized as follows:

```

Define Available Time = Time in the future when server will be available

Create arrival
  Assign Service Time
  Assign the Time in the future the activity would renege, which is equal
    to Tolerance Time + TNOW
  Assign Balk Limit

Decide
  If Balk Limit > Number in queue
    Balk entity
  If Renege Time < Available Time
    Renege entity
  Else
    Assign Available Time = MX(Available Time , TNOW) + Service Time
    Send entity to queue

```

Note the use of the "maximum" math function, MX.

There is one problem with this logic that can be fixed easily—our number in queue is not accurate because it won't contain any entities that have not yet reneged. We can fix this by sending our reneged entities to a Delay module where they are delayed by the renege time; we also specify a Storage (e.g., *Reneged_Customers*). Now we change our first Decide statement as follows:

```
If Balk Limit > Number in queue + NSTO(Reneged_Customers)
```

We have to be careful about our statistics, but this approach will capture the reneging process accurately and avoid our having to alter the queue.

Let's add one last caveat before we develop our model. Assume that the actual decision of whether to renege is based not only on the renege time, but also on the position of the customer in the queue. For example, customers may have reached their renege tolerance limit, but if they're now at the front of the waiting line, they may just wait for service (i.e., renege on reneging). Let's call this position in the queue where the customer will elect to stay, even if the customer renege time has elapsed, the customer *stay zone*. Thus, if the customer stay zone is 3 and the renege time for the customer has expired, the customer will stay in line anyway if they are one of the next three customers to be serviced.

We'll generate this position number from a Poisson distribution, POIS(0.75). We've used the Poisson distribution because it provides a reasonable approximation of this process and also returns an integer value. For those of you with no access to Poisson tables (you mean you actually sold your statistics book?), it is approximately equivalent to the following discrete empirical distribution: DISC(0.472, 0, 0.827, 1, 0.959, 2, 0.993, 3, 0.999, 4, 1.0, 5). See Appendix D for more detail on this distribution.

This new decision process means that the above logic is no longer valid. We must now place the arriving customer in the waiting line and evaluate the reneging after the renege time has elapsed. However, if we actually go ahead and place the customer in the queue, there's no mechanism to detect that the renege time has elapsed. To overcome this problem, we'll make a duplicate of each entity and delay it by the renege time. The original entity, which represents the actual customer, will be sent to the service queue. After the renege-time delay, we'll have the duplicate entity check the queue position of the original entity. If the customer is no longer in the service queue (that is, the customer was served), we'll just dispose of the duplicate entity. If the customer is still in the queue, we'll check to see if that customer will renege. If the current queue position is within the customer stay zone, we'll just dispose of the duplicate entity. Otherwise, we'll have the duplicate entity remove the original entity from the service queue and dispose of both itself and the original entity. This model logic is outlined as follows:

```

Create Arrivals
  Assign Renege Time = ERLA(15,2)
  Assign arrival time: Enter System = TNOW
  Assign Stay Zone Number = POIS(0.75)
  Assign Balking Tolerance = TRIA(3,6,15)

Create Duplicate entity
  Original entity to server queue
  If Balk from queue
    Count balk
    Dispose
  Delay for Service Time = EXPO(4.25)
  Tally system time
  Dispose

Duplicate entity
  Delay by Renege Time
  Search queue for position of original entity
  If No original entity
    Dispose
  If Queue position <= Stay Zone Number
    Dispose
  Remove original entity from queue and Dispose
  Count Renege customer
  Dispose

```

In order to implement this logic, we obviously need a few new features, such as the ability to search a queue and remove an entity from a queue. As you would suspect, Arena modules that perform these functions can be found in the Advanced Process and Blocks panels. Our completed Arena model (Model 9-3) is shown in Figure 9-6.

We start our model with a Create module that creates arriving customers, which are then sent to the following Assign module where the values we'll need later are assigned, as in Display 9-7.

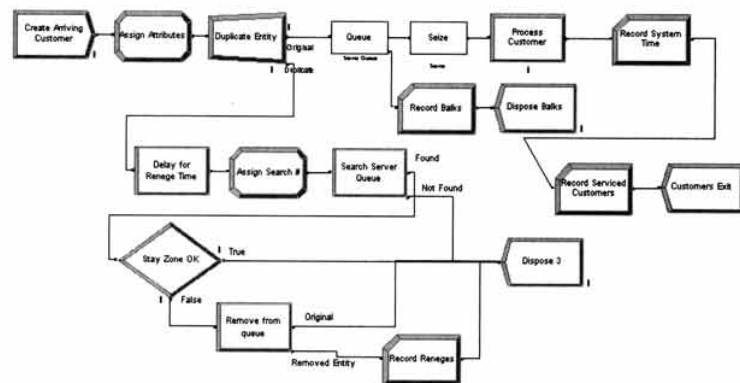


Figure 9-6. The Service Model Logic

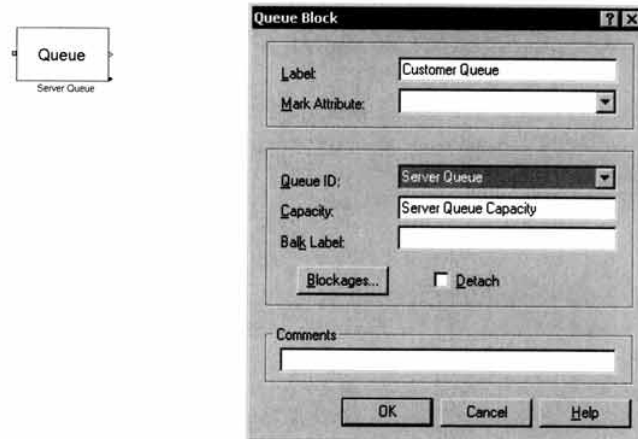
Name	Assign Attributes
Type	Attribute
Attribute Name	Enter System
New Value	TNOW
Type	Attribute
Attribute Name	Reneg Time
New Value	ERLA(15, 2)
Type	Variable
Variable Name	Server Queue Capacity
New Value	TRIA(3, 6, 15)
Type	Attribute
Attribute Name	Stay Zone
New Value	POIS(0.75)
Type	Variable
Variable Name	Total Customers
New Value	Total Customers + 1
Type	Attribute
Attribute Name	Customer #
New Value	Total Customers

Display 9-7. The Assign Module

We send the new arrivals to a Separate module. This module allows us to make duplicates (clones) of the entering entity. The original entity leaves the module by the exit point located at the right of the module. The duplicated entities leave the module by the exit points below the module. In this case, we only need to enter the name for our module, accepting the remaining data as the defaults.

The duplicates are exact replicas of the original entity (in terms of attributes and their values) and can be created in any quantity. If you make more than one duplicate of an entity, you can regard them as a batch of entities that will all be sent out of the same (bottom) exit. Note that if you enter a value n for # of duplicates, $n+1$ entities actually leave the module— n duplicates from the bottom connection point and 1 original from the top.

The original entity (customer) is sent to a Queue – Seize module sequence (from the Blocks panel) where it tries to enter queue Server Queue to wait for the server. In order to implement this method, we need to be able to set the capacity of the queue that precedes the Seize to the variable Server Queue Capacity. We accomplish this by attaching the Blocks panel to our model and selecting and placing a Queue module. (Note that when you click on the Queue module, there is no associated spreadsheet view; this will be the case for any modules from the Blocks or Elements panels.) Before you open the Queue dialog box, you might notice that there is a single exit point at the right of the module. Now we double-click on the module and enter the label, name, and capacity, as shown in Display 9-8. After you close the dialog box, you should see a second exit point near the lower right-hand corner of the module. This is the exit that the balked entities will take.



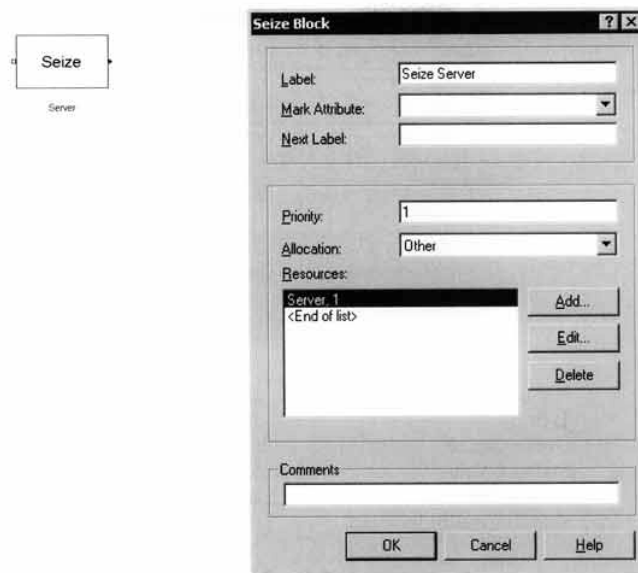
Label	Customer Queue
Queue ID	Server Queue
Capacity	Server Queue Capacity

Display 9-8. The Queue Module from the Blocks Panel

The capacity of this queue was entered as the variable `Server Queue Capacity`, which was set by our arriving customer as his tolerance for not balking, as explained earlier. If the current number in queue is less than the current value of `Server Queue Capacity`, the customer is allowed to enter the queue. If not, the customer entity is balked to the Record module where he increments the balk count and is then sent to a Dispose module, where he exits the system (see Figure 9-6). A customer who is allowed to enter the queue waits for the resource `Server`.

We need to follow this Queue module with a Seize module. When you add your Seize module, make sure it comes from the Blocks panel and not from the Advanced Process panel. The module in the Advanced Process panel automatically comes with a queue and Arena would become quite confused if you attempted to precede a seize construct with two queues. We then double-click on the module and make the entries shown in Display 9-9.

When the customer seizes the server, it is sent to the following Process module. This Process module uses a Delay Release Action, which provides the service delay and releases the server resource for the next customer. A serviced customer is sent to a Record module where the system time is recorded, to a second Record module where the number is counted, and then to the following Dispose module.

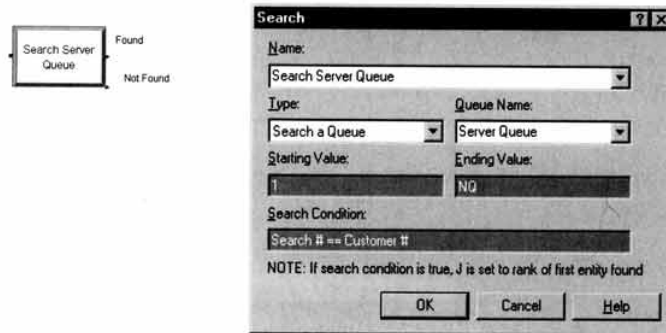


Label	Seize Server
Resource ID	Server
Number of Units	1

Display 9-9. The Seize Module from the Blocks Panel

The duplicate entity is sent to a Delay module where it is delayed by the renege time that was assigned to the attribute *Reneged Time*. After the delay, the entity enters an Assign module where the value of the attribute *Customer #* is assigned to a new variable named *Search #*. The attribute *Customer #* contains a unique customer number assigned when the customer entered the system. The entity is then sent to the following Search module, from the Advanced Process panel. A Search module allows us to search a queue to find the *rank*, or queue position, of an entity that satisfies a defined search condition. A queue rank of 1 means that the entity is at the front of the Queue (the next entity to be serviced). In our model, we want to find the original customer who created the duplicate entity performing the search. That customer will have the same value for its *Customer #* attribute as the variable *Search #* that we just assigned.

The Search module (Display 9-10) searches over a defined range according to a defined condition. Typically, the search will be over the entire queue contents, from 1 to NQ. (Note that the search can be performed backward by specifying the range as NQ to 1.)



Name	Search Server Queue
Type	Search a Queue
Queue Name	Server Queue
Starting Value	1
Ending Value	NQ
Search Condition	Search # == Customer #

Display 9-10. The Search Module

However, you may search over any range that your model logic requires. If you state a range that exceeds the current number in the queue, Arena will terminate with a runtime error. Arena will assign to the variable *J* the rank of the first entity during the search that satisfies the condition and send the entity out the normal exit point (labeled *Found*). If the condition contains the math functions *MX* or *MN* (maximum or minimum), it will search the entire range. If attributes are used in the search condition, they will be interpreted as the attribute value of the entity in the search queue. If the queue is empty, or no entity satisfies the condition, the entity will be sent out the lower exit point (labeled *Not Found*). Ordinarily, you're interested in finding the entity rank so you can remove that entity from the queue (Remove module) or make a copy of the entity (Separate module). The Search module can also be used to search over entities that have been formed as a temporary group using a Batch module. In addition, you can search over any Arena expression.

For our model, we want to search over the entire queue range, from 1 to NQ, for the original entity that has the same *Customer #* value as the duplicated entity initiating the search. If the original entity, or customer, is no longer in the queue, the entity will exit via the *Not Found* exit point and be sent to the following Dispose module. If the original entity is found, its rank in the queue will be saved in the variable *J*. The entity is then sent to the following Decide module. The check at this module is to see if the value of *J* is less than or equal to the value of the attribute *Stay Zone*. If this condition is *True*, it implies that the position of the customer in the queue is good enough that he chooses to remain in the line. In this case, we dispose of the duplicate entity (see Figure 9-6). If the condition is *False*, we want to renege the original customer. Therefore, we send the duplicate entity to the following Remove module.



Name	Remove from queue
Queue Name	Server Queue
Rank of Entity	J

Display 9-11. The Remove Module

The Remove module allows us to remove an entity from a queue and send it to another place in our model. It requires that you identify the entity to be removed by entering the queue identifier and the rank of that entity. If you attempt to remove an entity from an undefined queue or to remove an entity with a rank that is greater than the number of entities in the specified queue, Arena will terminate the run with an error. In our model, we want to remove the customer with rank J from queue *Server Queue*, as shown in Display 9-11.

If you look at the Remove module, you'll see two exit points on the right side. The entity that entered the Remove module will depart from the upper exit point; in our model, it is sent to the same Dispose module we used for the first branch of our Decide module. The customer entity removed from the server queue will depart by the lower exit point and is sent to a Record module to count the number of customers that renege. The entity is then sent to the Dispose module.

We set our Replication length to 2000 minutes. The result of the summary report for this model shows that we had 8 balking and 41 reneging customers with 332 serviced customers.

9.4 Holding and Batching Entities

In this section, we'll take up the common situation where entities need to be held up along their way for a variety of reasons. We'll also discuss how to combine or group entities and how to separate them later.

9.4.1 Modeling Options

As you begin to model more complex systems, you might occasionally want to retain or hold entities at a place in the model until some system condition allows these entities to progress. You might be thinking that we have already covered this concept, in that an entity waiting in a queue for an available resource, transporter, or conveyor space allows us to hold that entity until the resource becomes available. Here we're thinking in more general terms; the condition doesn't have to be based on just the availability of a resource, transporter, or conveyor space. The conditions that allow the entity to proceed