Proceedings of the 1999 Winter Simulation Conference
P. A. Farrington, H. B. Nembhard, D. Sturrock and G.W. Evans, eds.

WHO SERVES WHOM?
DYNAMIC RESOURCE MATCHING IN AN ACTIVITY-SCANNING SIMULATION SYSTEM

Photios G. Ioannou

Civil & Environmental Engineering Department
University of Michigan
Ann Arbor, Michigan 48109-2125, U.S.A.

Julio C. Martinez

Charles E. Via, Jr. Department of Civil Engineering
Virginia Tech
Blacksburg, Virginia 24061-0105, U.S.A.

ABSTRACT

This paper presents an activity-scanning simulation model for the familiar barbershop problem where customers have favorites, barbers may not show up for work, and customers may get impatient and leave. This example illustrates the mechanisms for matching customers to barbers or barbers to customers and argues that either approach may be implemented in an activity scanning system without altering the basic model structure. The solution to this problem is described in detail using a simulation model written in STROBOSCOPE.

1 INTRODUCTION

Perhaps the most important characteristic of any general-purpose discrete-event simulation modeling system is its simulation strategy. The main simulation system strategies in use today for modeling construction operations are *process interaction* and *activity scanning*.

Process interaction simulation models are typically written from the point of view of the moving entities (transactions) that flow through the system. These entities typically arrive, undergo some processing where they capture and release scarce resources, and then exit. Thus, an important characteristic of these models is the explicit classification of resources into those served (moving entities) and those that serve (scarce resources).

In contrast, activity scanning models are written from the point of view of the various activities that are performed and focus on identifying the nature of these activities and the required resources and conditions under which they may take place. In activity scanning, all resources are viewed as prerequisites for activities to start, and as a result, no distinction is made between resources that serve and resources being served.

This difference in resource modeling is of particular importance in applications of simulation modeling to civil engineering construction, and in particular to earthmoving, where the focus is often on the interaction between dissimilar equipment, such as loaders and haulers, or pushers and scrapers. A typical objective of an earthmoving application, for example, may be to investigate resource-matching strategies for assigning equipment of various sizes and capacities, such as large and small pushers and scrapers, to various activities. A detailed discussion of this type of problem appears in (Martinez & Ioannou 1999). An important conclusion of this discussion is that the equitable approach to resource modeling taken by activity scanning makes this simulation strategy particularly suitable for modeling construction operations that are characterized by interdependent components, complex activity startup conditions, and many resources that must work together under highly dynamic rules.

This paper presents an example that supports this conclusion and demonstrates the modeling power and flexibility afforded by activity scanning for modeling complex resource-matching problems relatively easily. This example also shows that in activity scanning it is possible to switch the role of resources (from servers to customers) and still arrive at a correct representation of the underlying operation without having to change the basic structure of the model.

The example selected for this purpose is the classic barbershop problem. Its description has been adapted from that in (Chisman 1996) where the problem is estimated to be of moderate difficulty, requiring 10 to 19 hours of work. This non-construction problem was chosen for a variety of reasons. It is familiar, complex, and yet small enough to be described completely in a limited amount of space. Furthermore, we hope to show that the required number of hours when modeling the problem in an activity-scanning system is substantially less than estimated.

A simulation model for this problem is presented using the notation of STROBOSCOPE (an acronym for STate and ResOurce Based Simulation of COnstruction

ProcEsses). STROBOSCOPE is a simulation language and system designed specifically for modeling construction operations based on three-phase activity scanning and activity cycle diagrams. The STROBOSCOPE language is described in (Martinez 1996). Example applications can be found in (Ioannou & Martinez 1996a, 1996b, 1996c) and (Martinez & Ioannou 1994, 1995, 1999).

2 PROBLEM STATEMENT

A barbershop with three barbers has to contend with customers choosing favorites, customers getting impatient and leaving, and absentee barbers. The mean time between customer arrivals is 12 minutes and it takes 19 minutes to cut a head of hair, both exponentially distributed. Of the customers, 30% prefer Barber B and 10% prefer Barber C (no one prefers A, the owner). There is a 5% chance that a barber will be absent any given day. If a favorite barber is absent, the customer will not wait. A customer will look at the overall queue to determine his impatience—if there are more than six, he will leave. A customer wanting a favorite will not wait if the number of customers stating a preference for his favorite is more than three.

3 SIMULATION MODEL

The activity-based network for the barbershop simulation model is shown in Figure 3. We will use this network to outline the overall structure of the simulation model before we describe any of its statements in detail.

The upper half of the simulation network represents a 24-hour clock that controls the day-to-day availability of *Barbers*. Together, the three nodes *Start*, *Work*, and *OffShift* act like a simple clock mechanism to regulate the daily cycle. At the start of simulation, queue *Start* is initialized with one unit of generic resource *Token*. This *Token* allows combi (i.e., conditional) activity *Work* to start at simulation time zero. When *Work* finishes eight-hours later, it releases the *Token* to the normal (i.e., bound) activity *OffShift*. Sixteen hours later, activity *OffShift* ends and releases the *Token* to queue *Start* to repeat the cycle. Thus, *Work* and *Offshift* follow each other in a continuous cycle of 8 hours of work-time and 16 hours of off-time.

At the start of simulation the three *Barbers* are located in queue *AbsentQ*. The queue *NewDay* contains one resource of generic type *Token* to enable combi activity *StartOfDay* to start. *StartOfDay* is a dummy (zero-duration) activity that removes all *Barbers* from *AbsentQ* (except any absentees) and releases them to *BarberQ* where they are available for work. At the end of working time, activity *Work* releases *n Tokens* (where *n* equals the number of *Barbers* working *that day*) to queue *EndDay*

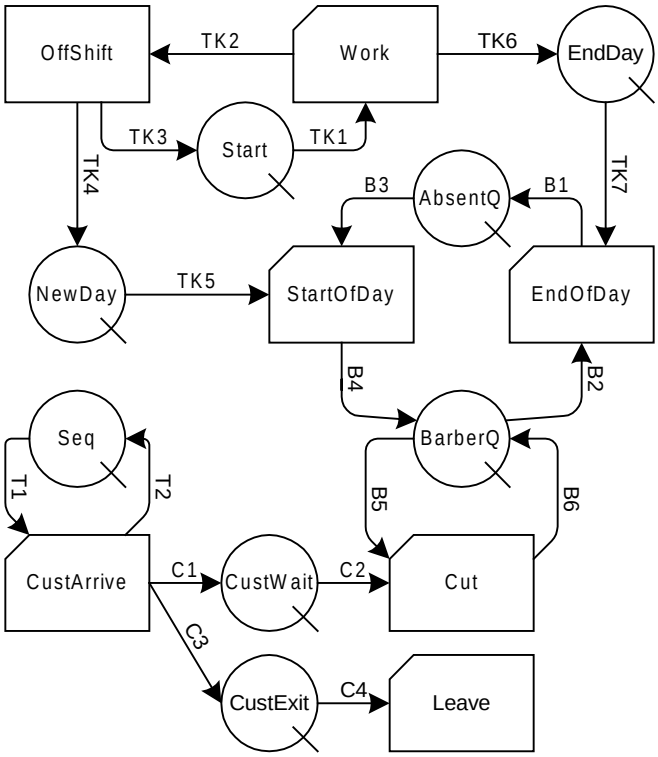


Figure 1: Barbershop Simulation Model Network

to allow dummy activity *EndOfDay* to start *n* times. Each such instance transfers one *Barber* from *BarberQ* to *AbsentQ* for the overnight break. When *OffShift* finishes, it releases one *Token* to queue *NewDay* to allow *StartOfDay* to start again and bring the *Barbers* to work.

The bottom-left part of the network models the arrival of *Customers*. The bottom-right part models what happens to these *Customers* when they arrive—they either get a haircut, or they change their minds and leave.

Queue *Seq* is initialized with one *Token* to allow activity *CustArrive* to start. The duration of *CustArrive* represents the interarrival time between successive *Customers*. When *CustArrive* finishes, it returns the *Token* to *Seq* and the cycle is repeated. Every time activity *CustArrive* finishes, it generates a new *Customer* resource that is released either to queue *CustWait* or to queue *CustExit* depending on the queue of *Customers* waiting for a haircut, the availability of a preferred *Barber*, and the time of day. Each *Customer* routed to *CustExit* is immediately drawn by an instance of dummy activity *Leave* and is destroyed. A *Customer* that enters *CustWait* joins the queue of *Customers* waiting for service. When a compatible pair of *Barber* and *Customer* are available, activity *Cut* starts and the *Barber* gives the *Customer* a haircut. A *Customer* that has a favorite *Barber*, waits in queue until that particular *Barber* is free. If a *Customer* that prefers another *Barber*, or has no

preference, arrives in the meantime, he may bypass those in queue and get served earlier. The mechanism for matching *Customers* to *Barbers* (or *Barbers* to *Customers*) is described in detail below.

3.1 Combi Instantiation and Resource Drawing

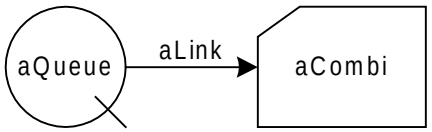
Before we present the STROBOSCOPE statements for the barbershop simulation model shown in Figure 1, it is necessary to explain the mechanism that determines when combi activities can start and how they acquire resources from preceding queues.

For a combi activity to start at any point during simulation (i.e., to create an instance of itself), it must pass two tests. The first test is the logical expression for its *semaphore* which must return a *true* (i.e., non-zero) value. The second test is that the *enough* expressions for all incoming links must also return *true* values. The default expressions and the statements for changing the *semaphore* for a combi and the *enough* expression for a drawing link are shown in Figure 2 (a *drawing* link is link from a queue to combi).

The default logical expression for the semaphore is 1 (*true*). The default *enough* expression for a drawing link returns the value *true* if the contents of the preceding queue are not zero. Thus, by default, a combi activity starts whenever all queues preceding it are not empty.

The process for *determining* whether a combi can start and create a new instance is completely separate from the process of actually *drawing* resources from the preceding queues to the new activity instance. Here we shall describe the drawing process for a combi preceded by a characterized queue (the process for queues holding generic resources is slightly different).

When an activity passes the *semaphore* and *enough* tests, it creates a new activity instance and by default tries to draw one resource from each preceding characterized queue. In general, the drawing process continues until the DRAWUNTIL logical expression for the corresponding link becomes *true*. In the default case, this occurs when the link is able to draw once. Every time the combi makes an attempt to draw a characterized resource from the preceding queue, it uses the drawing link to cursor (point to) a resource in the queue (starting at the front) and evaluates the logical expression for the DRAWWHERE attribute of the drawing link. If the result is *true*, then the resource is drawn. Otherwise, it is passed over and the link cursors the next resource in line. Thus, the logical expression for DRAWWHERE acts like a filter. The default logical expression for this filter is the value 1 (*true*) and the currently cursored resource is drawn. The drawing process continues until the DRAWUNTIL logical expression becomes *true* or the entire queue of characterized resources has been examined one-by-one.



SEMAPHORE aCombi *LogicalExpression*;
Default LogicalExpression: 1 (i.e., true)
ENOUGH aLink *LogicalExpression*;
Default LogicalExpression: aQue.CurCount
DRAWUNTIL aLink *LogicalExpression*;
Default LogicalExpression: aLink.nDraws>=1
DRAWWHERE aLink *LogicalExpression*;
Default LogicalExpression: 1 (i.e., true)
DRAWDUR aLink *SampleExpression*;
Default SampleExpression: 0

Figure 2: Combi Instantiation and Drawing Statements

The ability to define complex logical expressions for combi instantiation and resource drawing provides very precise control as to when activities can start and the resources they acquire.

3.2 STROBOSCOPE Simulation Code

This section describes the statements for a complete model implemented in STROBOSCOPE to illustrate how resource-matching and the dynamics of this problem may be modeled in an activity-scanning simulation system.

First we define the number of available *Barbers* as a problem parameter to allow for sensitivity analysis:

```
VARIABLE      nBarbers 3;
```

Next we define the resources that flow through the simulation model network.

```
GENTYPE       Token;
COMPTYPE      Barber;
CHARTYPE      Customer          Choice;
SUBTYPE       Customer Cust0        0;
SUBTYPE       Customer Cust2        2;
SUBTYPE       Customer Cust3        3;
```

Token is a generic resource that does not have any attributes and is used to initialize and maintain activity cycles like the clock and the arrival of *Customers*.

Barber is a compound characterized resource type. Each resource that belongs to a compound resource type is an individual entity with its own serial number (*ResNum*). As shown below, queue *AbsentQ* is initialized with three resources of type *Barber* (corresponding to the three barbers in the problem) that will automatically be assigned *ResNum* equal to 1, 2, and 3. The *Barber* whose

ResNum is 1 represents the owner, whereas the ones with *ResNum* equal to 2 and 3 are the ones preferred by 30% and 10% of the *Customers* respectively.

Customer is a characterized resource type for which we define one property called *Choice*. We also define three subtypes of *Customer*, *Cust0*, *Cust2*, and *Cust3* each having a value of *Choice* that reflects their *Barber* preference. The value 0 indicates no preference whereas 2 and 3 point to the *Barbers* with the same *ResNum*.

The control statements for initializating the network queues with resources at the start of simulation appear at the end of this section.

```
PRIORITY      Work '100';
DURATION      Work '8*60';
DURATION      OffShift '16*60';
```

Every time the simulation system enters the *combi instantiation phase* it sorts combi activities based on their priority and creates a new instance of every activity that can start. In this model, combi activity *Work* is assigned the highest priority because the existence of a *Work* instance is a prerequisite for other combi activities to start. The duration of *Work* (8 hrs) and *OffShift* (16 hrs) are expressed in minutes (the time units for this model).

```
DRAWUNTIL     B3 '0';
DRAWWHERE     B3 'Rnd[]<=0.95';
```

When combi *StartOfDay* starts, it tries to remove all three *Barbers* from *AbsentQ*. Link *B3* does this by pointing to (*cursoring*) and attempting to draw each *Barber* one after another. This process continues until the DRAWUNTIL logical expression becomes *true* (i.e., different from 0). In this case, this never happens (0 is always *false*) and link B3 cursors and attempts to draw all three *Barbers*. For each cursored *Barber*, link *B3* evaluates the logical expression for the DRAWWHERE statement and if the result is *true* it draws the cursored *Barber* to the starting new instance of *StartOfDay*. Otherwise, it cursors the next *Barber* and the process is repeated. A *Barber* is drawn only if the random number returned by function Rnd[] is less than 95%. Thus, on any given day each *Barber* has a 5% chance of staying in *AbsentQ* and not showing up for work.

```
RELEASEAMT TK6 nBarbers-AbsentQ.CurCount;
```

The number of *Tokens* released into queue *EndDay* at the end of activity *Work* equals the number of working *Barbers* (i.e., *nBarbers* - absentees). Each of these *Tokens* allows a separate instance of *EndOfDay* to start and transfer a working *Barber* from *BarberQ* to *AbsentQ* as soon as he is done with his last *Customer* for the day.

```
DURATION      CustArrive 'Exponential[12]';
```

The duration of *CustArrive* is exponential with a mean time of 12 minutes between *Customer* arrivals.

```
BEFOREEND CustArrive GENERATE
 PRECOND Work.CurInst 1
Rnd[]<0.3?Cust2:LastRnd[]<0.4?Cust3:Cust0;
```

When activity *CustArrive* ends, but before it releases any resources through its outgoing links, it generates (i.e., creates) the arriving *Customer* resource. The terms *PRECOND logicalexpression* mean that the action GENERATE takes place only when *logicalexpression* is true. The expression above returns the number of instances of activity *Work* currently going on. Thus, a *Customer* is generated only during working hours. No *Customer* arrivals should be modeled during *OffShift* because the barbershop is closed. The conditional expression at the end of this statement uses random sampling to determine which subtype of *Customer* to create (*Cust0*, *Cust2*, or *Cust3*).

Once a *Customer* resource is generated, it is released either through link *C1* to *CustWaits* or through *C3* to *CustExit*. The order in which links are defined in the simulation input file determines which is used to release resources first. In this model, link *C1* is defined first. Thus, only those resources not allowed through link *C1* will be released through *C3*. The release mechanism for link *C1* is controlled by the following statements.

```
FILTER NewCustChoice Barber
       ResNum==CustArrive.Customer.Choice;
FILTER SameChoiceCusts Customer
       Choice==CustArrive.Customer.Choice;
RELEASEWHERE C1
    'Choice > nBarbers ?  0:
    CustWait.CurCount>6?  0:
    !Choice?              1:
    !AbsentQ.NewCustChoice.Count &
  CustWait.SameChoiceCusts.Count<=3';
```

The first two statements define filters. A filter is applied to a queue to create a subset of the queue's contents. Thus, *AbsentQ.NewCustChoice* is a subset of the *Barbers* currently in queue *AbsentQ*. This subset includes only those *Barbers* whose *ResNum* equals the *Choice* property of the *Customer* in activity *CustArrive*. Since *Customers* arrive (i.e., are generated) only during working hours, this subset should be empty for most cases except when the *Barber* preferred by the new *Customer* is an absentee (in which case, the subset should contain exactly one *Barber* resource).

Similarly, *CustWait.SameChoiceCusts* is the subset of *Customers* in *CustWait* that prefer the same *Barber* as the newly generated *Customer* in activity *CustArrive*.

The RELEASEWHERE statement makes use of these filters and allows only those *Customers* that satisfy its logical condition to flow through link *C1*. This condition first rejects those *Customers* that prefer a *Barber* that does not exist (due to sensitivity analysis on *nBarbers*). Then it examines if there are more than 6 *Customers* in

CustWait. If so, the new *Customer* is not released. Otherwise, if the new *Customer* does not have a favorite (!*Choice* is *true*) then it is released. Otherwise, a *Customer* flows through *C1* only if the preferred *Barber* is not in *AbsentQ* and the *Customers* in *CustWait* that have the same favorite as the newcomer is at most three. A *Customer* that does not flow through *C1* is released through link *C3*.

The statements that follow at this point deal with the matching of *Barbers* to *Customers* (or *Customers* to *Barbers*) to determine whether combi activity *Cut* can start and which resources it draws. Since a detailed description of this portion of the model is one of the main objectives of this paper, the statements and their detailed explanation is deferred until the next section.

```
INIT Seq 1;
INIT Start 1;
INIT NewDay 1;
INIT AbsentQ nBarbers;
SIMULATEUNTIL SimTime>=100*24*60;
REPORT;
```

The *INIT* statements define the contents of queues at the start of simulation. Queues *Seq*, *Start*, and *NewDay* are initialized with one *Token*. Queue *AbsentQ* receives three *Barbers* (that are automatically assigned *ResNum* values of 1, 2, and 3). The *SIMULATEUNTIL* statement runs the actual simulation until its logical expression becomes true, i.e., until the simulation clock time (*SimTime*) exceeds 100 24-hour days (expressed in minutes). The *REPORT* statement produces the standard simulation output report.

3.3 Matching Customers to Barbers

The following statements match *Customers* in queue *CustWait* to *Barbers* in queue *BarberQ* to allow activity *Cut* to start and draw the correct resources.

```
FILTER MatchedCustomers Customer 1; /dummy
FILTER MatchedBarbers    Barber
         CustWait.MatchedCustomers.Count;
VARIABLE CurrentlyCursoredBarber
   'MatchedBarbers.HasCursor?
    MatchedBarbers.ResNum : B5.ResNum';
FILTEREXP MatchedCustomers
  !Choice|Choice==CurrentlyCursoredBarber;

PRIORITY     Cut '10';
ENOUGH       B5
           'BarberQ.MatchedBarbers.Count';
DRAWWHERE    B5
        'CustWait.MatchedCustomers.Count';
DRAWWHERE    C2
  '! Choice | Choice ==Cut.Barber.ResNum';
DURATION     Cut 'Exponential[19]';
```

These statements initiate the resource matching and drawing process by first examining the available *Barbers* and then trying to find the matching *Customers*. Similar statements that examine the *Customers* first and then try to find the matching *Barbers* are presented below.

Combi activity *Cut* can start when the *enough* expressions for links *C2* and *B5* are *true*. *C2* has the default *enough* expression that returns true when queue *CustWait* is not empty. Resource matching is performed by the *enough* expression for link *B5*.

The *enough* expression for B5 makes indirect use of two filters, *MatchedCustomers* and *MatchedBarbers*, that cross-reference each other and create a chicken-and-egg problem. This problem is solved by first defining *MatchedCustomers* with a dummy filter expression that is redefined later. Once the name *MatchedCustomers* is defined, it may be used to define the filter expression for *MatchedBarbers*. The actual filter expression for *MatchedCustomers* (that references *MatchedBarbers*) is defined later by the FILTEREXP statement. This way it is possible for two filters to reference each other. It is this ability to define cross-referencing filters that makes extremely complex resource-matching possible.

The expression *BarberQ.MatchedBarbers.Count* returns the number of *Barbers* in *BarberQ* that belong to the subset *BarberQ.MatchedBarbers.* The filter *MatchedBarbers* creates this subset by cursoring each *Barber* in *BarberQ* to determine whether it should be included. This decision is made by the value of *CustWait.MatchedCustomers.Count* which counts the number of *Customers* in *CustWait* that can be served by the *Barber* cursored by *MatchedBarbers*.

To see how this process works let us consider an example. Assume that *Barbers* B-1 and B-3 are in queue *BarberQ* and that the *Customers* in *CustWait* are as shown in Figure 3.

As indicated by the arrow, filter *MatchedBarbers* cursors the first *Barber* in *BarberQ* (i.e., B-1). Then it evaluates its filter expression and counts the number of *Customers* in *CustWait.MatchedCustomers*. This requires filter *MatchedCustomers* to cursor each *Customer* in *CustWait* and evaluate its own filter expression. The results are shown next to each *Customer* in Figure 3. Notice that *Choice* refers to the property of the *Customer* cursored by *MatchedCustomers*, whereas the expression *MatchedBarbers.ResNum* refers to the *ResNum* of the *Barber* cursored by *MatchedBarbers* (i.e., B-1). Thus, the subset formed by *CustWait.MatchedCustomers* is *tailored* to B-1 and represents those *Customers* that this *Barber* can serve. In this case, *Barber* B-1 can only serve *Customer* C-14 (the last *Customer* in queue *CustWait*). The expression *CustWait.MatchedCustomers.Count* for *Barber* B-1 returns #{C-14} = 1 (*true*), and thus B-1 is included in the subset *BarberQ.MatchedBarbers.*

BarberQ
(front of queue)

ResNum
→ 1
3

(back of queue)

MatchedBarbers
Filter logical expression:
CustWait.MatchedCustomers.Count
#{C-14} = 1 = *true*
#{C-12, C-13, C-14} = 3 = *true*
→ indicates *Barber* cursored by filter

CustWait
(front of queue)

ResNum	*Choice*
11	2
12	3
13	3
14	0

(back of queue)

MatchedCustomers
Filter logical expression
!Choice | Choice==MatchedBarbers.ResNum
(!2) | (2 == 1) = *false* | *false* = *false*
(!3) | (3 == 1) = *false* | *false* = *false*
(!3) | (3 == 1) = *false* | *false* = *false*
(!0) | (0 == 1) = *true* | *false* = ***true***
(the above example values assume B-1 is cursored by *MatchedBarbers)*

Figure 3: Evaluation of *BarberQ.MatchedBarbers.Count*

Filter *MatchedBarbers* then cursors B-3 and the entire process is repeated again as shown in Figure 4. The *Customers* that can be served by B-3 are C-12, C-13, and C-14. Expression *CustWait.MatchedCustomers.Count* for *Barber* B-3 returns #{C-12, C-13, C-14} = 3 (*true*). As a result, B-3 is also included in the subset formed by *BarberQ.MatchedBarbers.*

Thus, the subset *BarberQ.MatchedBarbers* includes both *Barbers*. The *enough* logical expression for link *B5* *BarberQ.MatchedBarbers.Count* = #{B-1, B-3}=2 (true). This means that there are *enough* matched *Barbers* and *Customers* to create a new instance of activity *Cut*.

The new instance of activity *Cut* must now draw the

BarberQ
(front of queue)

ResNum
→ 1
3

(back of queue)

Link B5
DRAWWHERE logical expression:
CustWait.MatchedCustomers.Count
#{C-14} = 1 = *true*
→ 1 = *Barber* cursored by link B5

CustWait
(front of queue)

ResNum	*Choice*
11	2
12	3
13	3
14	0

(back of queue)

MatchedCustomers
Filter logical expression
!Choice | Choice==B5.ResNum
(!2) | (2 == 1) = *false* | *false* = *false*
(!3) | (3 == 1) = *false* | *false* = *false*
(!3) | (3 == 1) = *false* | *false* = *false*
(!0) | (0 == 1) = *true* | *false* = ***true***
(the above example values assume B-1 is cursored by link *B5)*

Figure 5: Evaluation of DRAWWHERE for Link *B5*

BarberQ
(front of queue)

ResNum
1
→ 3

(back of queue)

MatchedBarbers
Filter logical expression:
CustWait.MatchedCustomers.Count
#{C-14} = 1 = *true*
#{C-12, C-13, C-14} = 3 = *true*
→ indicates *Barber* cursored by filter

CustWait
(front of queue)

ResNum	*Choice*
11	2
12	3
13	3
14	0

(back of queue)

MatchedCustomers
Filter logical expression
!Choice | Choice==MatchedBarbers.ResNum
(!2) | (2 == 3) = *false* | *false* = *false*
(!3) | (3 == 3) = *false* | *true* = ***true***
(!3) | (3 == 3) = *false* | *true* = ***true***
(!0) | (0 == 3) = *true* | *false* = ***true***
(the above example values assume B-3 is cursored by *MatchedBarbers)*

Figure 4: Evaluation of *BarberQ.MatchedBarbers.Count*

appropriate resources through its incoming links. This process begins by first drawing a *Barber* through link *B5*. This is not necessarily the first *Barber* in line, but rather the first one that can be matched with a *Customer* in *CustWait*. Once the appropriate *Barber* is drawn, *Cut* draws the first matching *Customer* through link *C2*.

The DRAWWHERE logical expression for link B5 is identical to the filter expression for *MatchedBarbers*. As shown in Figure 5 when link B5 cursors *Barber* B-1 and evaluates its DRAWWHERE logical expression it returns #{C-14} = 1 (*true*). Thus, Barber B-1 is drawn to the newly created instance of activity *Cut* and the drawing process through link *B5* ends.

The main difference between Figures 3 and 5 is that in the former the *Choice* property of *Customers* is compared to the *ResNum* of the *Barber* cursored by filter *MatchedBarbers* (i.e., *MatchedBarbers.ResNum*). In the latter it is compared to the *ResNum* of the *Barber* cursored by link *B5* (i.e., *B5.ResNum*). Variable *CurrentlyCursoredBarber* is defined to return the correct *ResNum* depending on whether filter *MatchedBarbers* is active or not (see variable *MatchedBarbers.HasCursor*).

The next step after drawing a *Barber* through *B5* is to draw a matching *Customer* through link *C2*. This does not require the use of filters because the new instance of activity *Cut* has already drawn a *Barber* and makes it easy to refer to its *ResNum* (*i.e.*, *Cut.Barber.ResNum*). The process is illustrated in Figure 6.

Link *C2* cursors the first *Customer* in queue *CustWait* and evaluates its DRAWWHERE expression. Since the result is *false*, it cursors the next Customer in line. This process is repeated until a *Customer* can be drawn (i.e.,

CustWait (front of queue)

ResNum	*Choice*
11	2
12	3
13	3
14	0

(back of queue)

Link C2

DRAWWHERE logical expression:

!Choice | Choice==Cut.Barber.ResNum

(!2) | (2 == 1) = *false* | *false* = *false*

(!3) | (3 == 1) = *false* | *false* = *false*

(!3) | (3 == 1) = *false* | *false* = *false*

(!0) | (0 == 1) = *true* | *false* = ***true***

(the above example values assume *Cut.Barber.ResNum* = 1*)*

Figure 6: Evaluation of DRAWWHERE for Link C2

C-14). (If the DRAWWHERE expression is *false* for every cursored resource then no drawing takes place.) After the new instance of *Cut* receives *Barber* B-1 and *Customer* C-14, it proceeds to determine its duration.

For the resource situation assumed for the example above, activity Cut is able to start a second time and create another new instance. It is not hard to see that the second *Cut* instance will draw *Barber* B-3 and *Customer* C-12. When these resources are drawn, *BarberQ* becomes empty and no more *Cut* instances can be created at this time.

3.4 Matching Barbers to Customers

The statements below illustrate how the modeling perspective can be reversed to match *Barbers* in queue *BarberQ* to *Customers* in queue *CustWait.*

```
FILTER MatchedBarbers Barber 1; /dummy
FILTER MatchedCustomers Customer
          BarberQ.MatchedBarbers.Count;
VARIABLE ChoiceOfCursoredCustomer
   'MatchedCustomers.HasCursor?
     MatchedCustomers.Choice : C2.Choice';
FILTEREXP MatchedBarbers
        '!ChoiceOfCursoredCustomer |
        ResNum==ChoiceOfCursoredCustomer';
ENOUGH C2 CustWait.MatchedCustomers.Count;
DRAWWHERE C2 BarberQ.MatchedBarbers.Count;
DRAWWHERE B5 '!Cut.Customer.Choice |
            ResNum==Cut.Customer.Choice';
PRIORITY  Cut  '10';
DURATION  Cut  'Exponential[19]';
```

These statements are very similar to those used earlier for matching *Customers* to *Barbers.* Figure 7 shows the same example as above to illustrate how they work.

The *enough* expression for link *C2* returns the value #{C-12, C-13, C-14} = 3 (*true*). In other words, there are three *Customers* that can be matched with a *Barber,* so there are *enough* to create a new instance of activity *Cut.* As shown in Figure 8, the new instance will first draw

CustWait (front of queue)

ResNum	*Choice*
→11	**2**
12	3
13	3
14	0

(back of queue)

MatchedCustomers

Filter logical expression

BarberQ.MatchedBarbers.Count

#{} = 0 = *false*

#{B-3} = 1 = *true*

#{B-3} = 1 = *true*

#{B-1, B-3} = 2 = *true*

→indicates *Customer* cursored by filter

BarberQ (front of queue)

ResNum
1
3

(back of queue)

MatchedBarbers

Filter logical expression:

!MatchedCustomers.Choice | ResNum==MatchedCustomers.Choice

(!2) | (1 == 2) = *false* | *false* = *false*

(!2) | (3 == 2) = *false* | *false* = *false*

(these example values assume C-11 is cursored by *MatchedCustomers*)

Figure 7: Value of CustWait.*MatchedCustomers.Count*

Customer C-12 and then *Barber* B-3 (to draw in this order, link *C2* should be defined before link *B5*).

At this point, the *enough* expression for link *C2* is reevaluated and returns #{C-14} = 1 (*true*). Thus, *Cut* creates a second activity instance that draws *Customer* C-14 first and then *Barber* B-1. No more instances of *Cut* can be created because queue *BarberQ* is now empty. Notice that the two instances of *Cut* are created (i.e., draw resources) in reverse order from that in the previous section, i.e., #1:{B-1, C-14}, #2:{B-3, C-12} vs. #1:{C-12, B-3}, #2:{C-14, B-1}.

4 CONCLUSION

The two barbershop models have been animated *using PROOF Animation* to verify that they are indeed correct. They also produce identical results, even though the roles of *Barbers* and *Customers* are reversed. In the actual models, it is impossible for more than one instance of *Cut* to start at the same simulation time. (At most, only one instance can start when a *Customer* arrives, or when a *Barber* becomes free).

This example illustrates the power of an activity scanning system to model complex resource matching and activity startup-up conditions, and even reverse the role of resources without changing the basic structure of the simulation model (Figure 1). As shown in (Martinez 1996) many resource-matching problems such as this can be classified into a few standard types. Thus, the time to develop and verify a model for this problem is less than half that estimated by (Chisman 1996).

CustWait
(front of queue)

ResNum	*Choice*
11	2
→12	3
13	3
14	0

(back of queue)

Link C2
Filter logical expression
BarberQ.MatchedBarbers.Count
#{} = 0 = *false*
#{B-3} = 1 = ***true***
#{B-3} = 1 = *true*
#{B-1, B-3} = 2 = *true*
→ shows *Customer* cursored by C2

BarberQ
(front of queue)

ResNum
1
3

(back of queue)

MatchedBarbers
Filter logical expression:
!C2.Choice | ResNum==C2.Choice
(!3) | (1 == 3) = *false* | *false* = *false*
(!3) | (3 == 3) = *false* | *true* = ***true***
(these example values assume C-12 is cursored by link *C2*)

Figure 8: Evaluation of DRAWWHERE for Link *C2*

STROBOSCOPE, its documentation, and several solved examples are available at http://grader.engin.umich.edu and http://strobos.ce.vt.edu.

ACKNOWLEDGMENTS

The authors wish to thank the National Science Foundation (Grants CMS-9415105, CMS-9733267) for supporting portions of the work presented here. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

Chisman, J.A. 1996. *Industrial Cases in Simulation Modeling*, Duxbury Press, Wadsworth Publishing Co., International Thomson Publishing, p.29.

Ioannou, P.G. and J.C. Martinez. 1996a. Comparison of Construction Alternatives Using Matched Simulation Experiments. *Journal of Construction Engineering and Management*, ASCE, (122)3:231-241.

Ioannou, P.G. and J.C. Martinez. 1996b. Simulation of Complex Construction Processes. *Proceedings, 1996 Winter Simulation Conference*. Society for Computer Simulation, San Diego, CA, 1321-1328.

Ioannou, P.G. and J.C. Martinez. 1996c. Scaleable Simulation Models for Construction Operations. *Proc. 1996 Winter Simulation Conference*. Society for Computer Simulation, San Diego, CA, 1329-1336.

Martinez, J.C. 1996. STROBOSCOPE: State and Resource Based Simulation of Construction Processes. Ph.D. Dissertation, Department of Civil and Environ. Engineering, University of Michigan, Ann Arbor, MI.

Martinez, J.C. and P.G. Ioannou. 1994. General Purpose Simulation with STROBOSCOPE. *Proceedings, 1994 Winter Simulation Conference*, Society for Computer Simulation, San Diego, CA, 1159-1166.

Martinez, J.C. and P.G. Ioannou. 1995. Advantages of the Activity Scanning Approach in the Modeling of Complex Construction Processes. *Proceedings of the 1995 Winter Simulation Conference*, Society for Computer Simulation, San Diego, CA, 1024-1031.

Martinez, J.C. and P.G. Ioannou. 1999, General Purpose Systems For Effective Construction Simulation, *J. of Construction Engineering and Management*, ASCE, (125)4, July-August.

AUTHOR BIOGRAPHIES

PHOTIOS G. IOANNOU is Associate Professor in the Dept. of Civil and Environmental Engineering at the Univ. of Michigan. He has received a Civil Engineer's degree from the National Technical University, Athens, Greece, in 1979; and a SMCE and Ph.D. in Civil Engineering from MIT in 1981 and 1984. From 1989-1995 he has served as Chairman of the Computing in Construction Technical Committee of the ASCE. He co-developed the UM-CYCLONE construction process simulation system with R.I. Carr, supervised the design and development of COOPS by L.Y. Liu, and chaired J.C. Martinez's Ph.D. dissertation on STROBOSCOPE. His primary research interests are in the areas of decision support systems and construction process modeling.

JULIO C. MARTINEZ is an Assistant Professor in the Via Department of Civil Engineering at Virginia Tech. He received his Ph.D. in Civil Engineering at the University of Michigan in 1996; an MSE in Construction Engineering and Management from the University of Michigan in 1993; an MS in Civil Engineering from the University of Nebraska in 1987; and a Civil Engineer's degree from Universidad Catolica Madre y Maestra (Santiago, Dominican Republic) in 1986. He designed and implemented the STROBOSCOPE simulation language as part of his Ph.D. dissertation research. His research interests include discrete event simulation, scheduling of complex and risky projects, and construction management information systems.