# A PROGRAMMING THEORY FOR DISCRETE SIMULATION

**Robert E. Haymond**

## ABSTRACT

Discrete systems are abstracted as Cartesian products of automata operating on a data structure. These abstract systems are then modeled as intuitive automata with natural (internal) updates and unnatural (external) updates. Formal programs which perform these functions as well as necessary information flow between subsystems are presented. A collection of examples illustrate the theory.

## INTRODUCTION

Digital modeling is one of the areas being developed at Clemson University as part of our efforts in the mathematical sciences. We have found in many cases that this modeling occurs as a two-stage process. From the physical system a first model serves to abstract the system and from that, one develops a second model which results in a program. These models are labeled generic model and specific model. A programming theory is a generic-specific pair which models a wide class of problems. For several years we have been designing and testing such a theory for discrete simulation. At this point, we have enough evidence to feel that the theory accomplishes what we want from it. Major factors are:

1. The ability to model any discrete event system which we can imagine with a high degree of certainty of obtaining a running program.

2. Essential immunity to system complexity. Although there is a moderate amount of work to model the simplest system this way, there is little increase in difficulty as the system get indefinitely complex.

3. Flexibility of model design. Any system model can immediately be added as a subsystem of a larger model. Any subsystem can be replaced by a more complex model as more information is available.

4. System polices can be identified in the minutest detail, no "mysterious events."

5. Even on major systems, programs are easy to debug.

We are now preparing a report on our major test of the system thus far: a model of the Southern Railway system [2], [3]. This allowed a direct comparison with a comparable model sponsored by the American Association of Railroads and done in either Simscript or GPSS [1], [6]. We can reproduce results of the earlier model with run time faster by roughly two orders of magnitude. A simulation of thirteen days of system time reduced from six hours to four minutes. We have since demonstrated point 3 above by adding a complete simulation of track segments in terms of siding versus multiple tracks.

Although the theory has been developed to some degree of completeness and we have made some extensions [4], [5] we realize that there are some difficulties in communicating to others what the theory is all about. The purpose of this article is to present a first cut at making the theory available outside the classroom.

There are several principles on which the presentation is based:

1. This is a method of accomplishing running programs, not a method for avoiding them. The novice programmer will find it difficult material.

2. The only way we have found to teach this material is by example. We will cover a sequence of examples, some in detail and some suggested. The total set is what we consider minimal to learn the material. The sequence is selected for various levels of complexity and not necessarily for the value of the finished models.

3. The ideas are essentially (computer) language independent. However, they must be described in some language

and we will use FORTRAN. This means that a reader who prefers a different language, or indeed some changes in programming style, should probably understand what we have done first and then make the appropriate translation.

## THE MODELS

### Generic Model

A discrete system is modeled as an automaton; a digraph of its state space and state-change function which indicates the state at any given time. Of course, producing the state-change functions computationally is at the heart of the difficulty in discrete simulation. Even though any small change could be considered a different state, we have found that most systems have a natural "data structure." Hence, we assume a system digraph of logical states and at each change in state, a set of operations is performed on a data structure. These operations on the data structure correspond to the output function in the algebraic definition of an automaton. In some sense if we think (intuitively) of an automaton as a self-operating entity, then we have a discrete system as an automaton operating on a data structure. At the next step, we consider two or more systems so modeled and join these to make a larger system. The digraph of the system is the Cartesian product of the subsystem digraphs and the data structure is the union. If there are interrelations between the subsystems then the product can furnish inputs to the subsystems. In this fashion a product or product of products (etc.) is a system model. Clearly, a system of many subsystems can be represented as a collection of products in many different ways. When we get to the specific model we will see that it is not too difficult to select product elements for a successful simulation. All of these notions can be specified in algebraic detail and is a good way to abstract any discrete system. On the other hand, it does not furnish any reasonably direct fashion for programming the interrelations and indeed combinatorics on interrelations. This will be furnished by additional modeling.

### Specific Model

Here we make additional modeling assumptions on automata and will hereafter refer to them as subsystems. We assume that all state changes occur in two fashions. A natural update is a state change which will occur in a totally internal fashion at a given future time provided no external input occurs. Any subsystem will have a scheduled natural update at any time (even if it is scheduled trivially at infinity). An unnatural update is a state change which occurs to a subsystem due to an external input. Any subsystem will have a set (possibly empty) of inputs with which it can react. The input for a natural update is simply a time. Now imagine a tree structure which corresponds to a multilevel product. We partition this tree as follows:

1. The root is a product but not a factor. This is called the system.

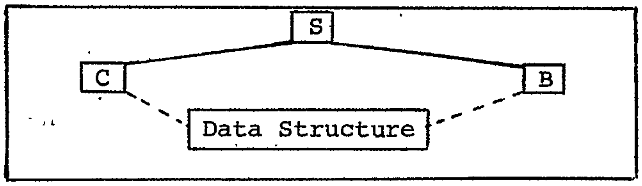
2. The terminal nodes are factors but not products. They are called atoms.

3. All others are factors and products. They are called intermediates.

The natural update time for any product is the minimum natural update time for its factors; the system has only natural updates. The inputs acceptable to any intermediate consist of exactly those it can parcel out to its factors. Vacuously or otherwise an atom has both unnatural and natural updates.

There are formal programs for the system, intermediates and atoms. In FORTRAN each of these is a subprogram with entry points. Like most programming issues, these abstractions are understood only after repeated examples. The remainder of the paper is directed entirely toward techniques of programming. Classroom evidence (graduate level) indicates the necessity of heavy reader participation by way of actual modeling and programming. In particular, to save space, programming will be omitted in any case where the needed computation is theory independent. We will begin with a system which would be easy to handle in any case and progress to some which are quite intricate. In order to concentrate on the theory, we will omit any notion of statistics and output; the information is available for both and can be done as the reader's choice.

Example 1. Barber shop with n barbers; customers either do or do not have a preference for a particular barber. The seating policy should be first come, first served; however, if at the beginning of service a shuffling of those without preference allows additional service, it should be done. This model is done in detail as the least level of the theory. As defined earlier, we have a system and two atoms and the diagram can be thought of as representing the generic model and the specific model. In the generic case: C has a state vector which consists of the specification of arrival times for




customers in n+1 queues and next arrival time for each queue; B has a state vector which contains the finish time for each barber and the number of barbers available; the state vector of S is system time plus the union of the previous states. The simulation would be the sequence of states of S. For the specific case, think of these three as operating automata (or black boxes). The natural updates are: C next arrival; B next finish; S minimum of these two. Unnatural updates are: C release a customer to service; B a barber begins service; S none. The information flow: C to S natural update time; B to S natural update time; S to C & B time. For convenience, we segregate a portion of S called BRANCH which handles unnatural updates. Then C sends to BRANCH a permutation vector P(n+1) which lists the priority (based on arrival time) of the n+1 queues; B sends to BRANCH a logical array A(n) which barbers are available. Program for this specific model:

```
C               Main

       CALL START
       CALL SYSTEM
       CALL STOP
       END
            SUBROUTINE SYSTEM
       COMMON/ALL/ TIME,UPDT(2)
       COMMON/STS/ BIGT
1      TIME=MIN(UPDT)
       CALL STATE
       IF (TIME.GE.BIGT) RETURN
            CALL  C  N
            CALL  B  N
2      CALL BRANCH(&1,&3)
3           CALL  C  U
            CALL  B  U   (&2)
       END
         SUBROUTINE BRANCH  (*,*)
       COMMON/BRC/ P(n+1),WHICH Q
       COMMON/BRB/ A(n),WHICH B
           see notes below for seating
           algorithm then RETURN1 or
           compute WHICH Q and WHICH B
           and RETURN2.
       END
              SUBROUTINE C
       COMMON/ALL/ TIME,UPDATE
       COMMON/BRC/ P(n+1),WHICH
               ENTRY   C  N
1      IF (TIME.NE.UPDATE) RETURN
       CALL PUSH C (UPDATE,P,&1)
               ENTRY   C  U
2      CALL POP C  (UPDATE,P,WHICH)
       RETURN
       END
              SUBROUTINE B
       COMMON/ALL/ TIME,BLANK,UPDATE
       COMMON/BRB/ A,WHICH
               ENTRY  B  N
1      IF (TIME.NE.UPDATE) RETURN
       CALL POP  B  (UPDATE,A,&1)
             ENTRY  B  U  (*)
       CALL  DS  C  (SVT)
       CALL PUSH  B  (UPDATE,A,WHICH,SVT)
       RETURN1
       END
```

Notes: (1) Seating algorithm. A relevant queue is one with customers waiting and the corresponding barber is available. If there are none of these then no action is taken. Otherwise compare the priority of relevant preference queues with that of the non-preference queue. Then service the customer in the top preference queue or service the non-preference queue with the barber corresponding to queue of least priority. It is important that only one service occurs on any given pass through BRANCH and the product logic cycles until there is no additional service to begin.

(2) The PUSH (put in) and POP (take out) routines are operations and ENTRY points on a data structure, DS. In case PUSH C, the START routine reads the appropriate arrival and service times for the queues and furnishes this information (by COMMON) to DS. This means that DS can maintain full queue information including next arrival. It also maintains full service information including next finish. The CALL statements are self explanatory, it being assumed that the information in each is correct on RETURN. Atoms, here and in general, are operations on data structures. The calling statements should contain only essential information from above and essential information to pass along. The data structure should contain all possible details. In this fashion, the models are data structure independent and in larger, more complex systems this allows the freedom to employ any of the very powerful data structure techniques desired.

(3) The service time is called by B much as a real system might use a terminal. This allows a partition of the data structure corresponding to the various subsystems.

For those interested in doing simulations this way, it is recommended that this model be run before proceeding. Now we abstract these ideas to a general product including system, intermediates, and atoms. The discussion parallels the program and they should be studied together.

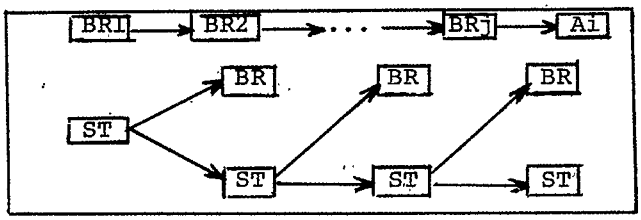
We will concentrate, inductively, on the operation of an intermediate automaton and then see how that alters for a system and for an atom. The intermediate has 3 major sections: natural update, unnatural update, and data processing.

Natural Update. The natural update time of any product is the minimum of the natural update times of its factors. Logically this minimum must be computed after each natural update and after each unnatural update. This could lead to excessive computation and as the natural update times are forever changing much of the minimization is wasted. Linked lists could eliminate some of the computation but there would still be much ado for update times which are likely to change before they are used. The method used was suggested by Mertens [5]; it transfers responsibility to the lower routines. The information is kept in an UPDATE array with a pointer NEXT. The value UPDATE(NEXT) is always correct in a given routine and is maintained by factor routines with calls of the form N x (I,UPDATE). It is assumed, inductively, that after any natural update call on a routine it returns with a natural update time which is later than TIME.

Unnatural Update. This calls a branch (BR) routine which goes to data processing if there is null input or makes a sequence of calls to unnatural updates of factor routines. There is a cycle through BR until such time as inputs are null.

Data Processing. We have already discussed the CALL N x (I,UPDATE(NEXT)) which sends up this routine's contribution to update. There is also a state (ST) routine which receives processed information. The one difficult algorithm in the barber shop was left in intentionally. It was a little messy because contributing routines supplied only logical information instead of processed information. There is enough information to make a decision but the earlier fashion forces all of the complexity to one spot. The ST routines should be carefully constructed to send information in its most useful form. In the barber shop, for example, C ST might provide a linked list of customer priorities.

The system routine only looks down so it has the natural and unnatural updates but no data processing; the atoms only look up so they have all three sections but no BR. The atoms have no factors and accomplish the natural and unnatural updates by operations on various components of data structure. The cycle of natural updates in each atom completes the induction of each routine completing all natural updates before return. The information flow is summarized as follows:




The following abstract program is intended for study much as one would study mathematics. Here are some notational conventions:

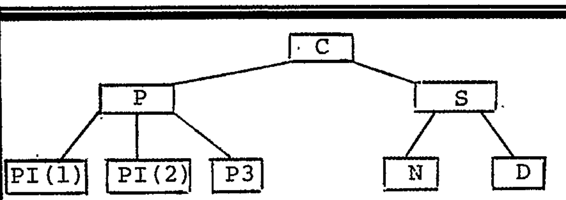
```
N   x      natural update processing for x
           example CALL N F2 (3,UPDATE)
           #3 factor of F2.
ENTRY F1 N    natural update on F1
ENTRY A1 U    unnatural update on A1
CALL  S  BR   branch routine for S
CALL  A1 ST   state routine for A1
CALL  DS x i  entry #i on the data
              structure for automaton x.

          SUBROUTINE SYSTEM
      COMMON/ALL/ TIME
      COMMON/SN/ UPDATE (n), NEXT
1     TIME=UPDATE(NEXT)
      IF (TIME.GE.BIGT) RETURN
2     IF (TIME.NE.UPDATE) GO TO 10
      GO TO (3,4,5,...), NEXT
3         CALL F1  N  (&2)
4         CALL F2  N  (&2)
 ..       ...............
zz        CALL Fn  N  (&2)
10    CALL S  BR  (&11,&1)
11        CALL F1  U  (&12)
12        CALL F2  U  (&13)
 ..       ................
ww        CALL Fn  U  (&10)
      END
            SUBROUTINE Fi
      COMMON/ALL/ TIME
      COMMON/FiNUP/ UPDATE (n), NEXT
1     IF (TIME.NE.UPDATE(NEXT)) GO TO 100
          ENTRY  Fi  N  (*)
      GO TO (2,3,4,...), NEXT
2         CALL  Fi  1  N  (&1)
3         CALL  Fi  2  N  (&1)
 ..       ...................
qq        CALL  Fi  n  N  (&1)
          ENTRY  Fi  U  (*)
10    CALL  Fi  BR  (&11,&100)
11        CALL  Fi  1  U  (&12)
12        CALL  Fi  2  U  (&13)
 ..       .....................
vv        CALL  Fi  n  U  (&10)
100   CALL  N  S  (i,UPDATE(NEXT))
      CALL  Fi  ST
      RETURN1
      END
A typical N  x  routine:
         SUBROUTINE  N  Fj  (I,U)
      COMMON/FjNUP/ UPDATE(n), NEXT
      IF (U-UPDATE(NEXT)) 1,2,3
1         NEXT=I
2         UPDATE (NEXT)=U
          RETURN
3     UPDATE (I)=U
```

```
      IF (I.NE.NEXT) RETURN
           DO 4 L=1,n
4          IF(UPDATE(L).LET.UPDATE(NEXT))
     *     NEXT=L
      RETURN
      END
```

The only case which requires a search is that of replacing the previous UPDATE (NEXT) with a greater value and I=NEXT.

```
           SUBROUTINE Ai
       COMMON/ALL/TIME
1      IF (TIME.NE.UPDATE) GO TO 100
           ENTRY  Ai  N  (*)
       CALL  DS  x  1
       CALL  DS  y  2  (UPDATE,etc.)
       CALL  DS  z  3  (&1)
           ENTRY  ai  U  (*)
       GO TO (200,10,11,12,...), REQ
10
   data structure calls corresponding to
11 various requests to which this atom
   can respond; as above but in groups
12 each group should return to 100.
       ..................................
100    CALL  N  Fj  (i,UPDATE)
       CALL  Ai  ST
200    RETURN1
       END
```

Example 2. Medical clinic with n doctors and m nurses; patients require a doctor, a nurse, or both. Again, first come, first served with the exception: if (a) a patient needs both a doctor and a nurse, (b) a doctor is available, and (c) no nurse is available but a nurse is serving a patient alone, then the nurse patient is bumped back to the queue with remaining service time and the nurse joins the doctor on the new service. It is this "undoing of events" that can generally cause difficulty. An indexed product is introduced as a product of identical factors. The structure is given below:




| | |
|---|---|
| C | clinic |
| P | patients |
| S | servers |
| PI(1) | need doctor |
| PI(2) | need both |
| P3 | need nurse |
| N | nurses |
| D | doctors |

Natural updates: PI(1), PI(2), P3 patient arrivals; N,D finish service.
Unnatural updates: PI(1), PI(2) release patient to service; P3 release patient to service or accept patient back in queue; D accept patient for service; N accept patient for service or bump least priority P3 back to queue.
Information passed: PI(1), PI(2), P3 to P (through ST) whether patients are there and arrival times for first two queues; P to C whether patients are there and which of the first two queues has priority; D to S 0 or 1; N to S 0, 1,2 where 2 means no nurses free but some serving P3's; S to C which types of patients can be served.

```
           SUBROUTINE CLINIC
       COMMON/C N/ UPDATE(2), NEXT
       COMMON/CST/ BIGT
1      TIME=UPDATE(NEXT)
       IF (TIME.GE.BIGT) RETURN
2      IF (TIME.NE.UPDATE(NEXT)) GO TO 10
       GO TO (3,4), NEXT
3          CALL  P  N  (&2)
4          CALL  S  N  (&2)
10     CALL  C  BR  (&11,&1)
11         CALL  P  U  (&12)
12         CALL  S  U  (&10)
       END
         SUBROUTINE  C  BR  (*,*)
       COMMON/CBRPS/ REQ(2)
       COMMON/CBRS/ REQS
       COMMON/CBRSTP/ HERE(3),FIRST
       COMMON/CBRSTS/ SERV(3)
       DIMENSION ORDER(3)/1,2,3/
       LOGICAL HERE, SERV
       REQ(1)=1
       REQ(2)=1
       ORDER(1)=FIRST
       ORDER(2)=3-FIRST
       DO 10 I=1,3
       IF(
      *HERE(ORDER(I)).AND.SERV(ORDER(I))
      *) GO TO 1
10     CONTINUE
       RETURN2
1      DO 2 J=1,2
2      REQ(I)=ORDER(I)+1
       IF(.NOT.REQ(1).EQ.3.OR.SERV(3))
      *RETURN1
       REQ(1)=5
       REQ(2)=5
       RETURN1
       END
           SUBROUTINE  S
       COMMON/ALL/ TIME
       COMMON/CBRS/ REQ
       COMMON/SN/ UPDATE(2), NEXT
1      IF (TIME.NE.UPDATE(NEXT)) GO TO 100
           ENTRY  S  N  (*)
       GO TO (2,3), NEXT
2          CALL  D  N  (&1)
3          CALL  N  N  (&1)
           ENTRY  S  U  (*)
10     CALL  S  BR(&11,&200)
11         CALL  D  U  (&12)
12         CALL  N  U  (&100)
100    CALL  N  C  (2,UPDATE)
       CALL  S  ST
```

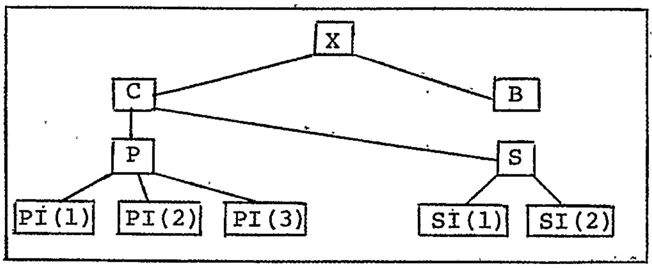
```
200    RETURN1
       END
         SUBROUTINE  S  BR  (*,*)
       COMMON/CBRPS/ REQ
       COMMON/SBRDN/ REQD,REQN
       REQD=1
       REQS=1
       GO TO (1,2,3,4,5), REQ
1      RETURN2                     nothing
2      REQD=2                      doctor
       RETURN1
3      REQD=2                      both
       REQN=2
       RETURN1
4      REQN=2                      nurse
       RETURN1
5      REQN=3                      bump
       RETURN1
       END
           SUBROUTINE  S  ST
       COMMON/CBRSST/  S(3)
       COMMON/SSTDN/ DA,NA
       LOGICAL  S
       S(1)=DA.EQ.1
       S(3)=NA.EQ.1
       S(2)=S(1).AND.(S(3).OR.NA.EQ.2)
       RETURN
       END
           SUBROUTINE  D
       COMMON/ALL/ TIME
       COMMON/SBRDN/ REQ
1      IF (TIME.NE.UPDATE) GO TO 100
           ENTRY  D  N  (*)
       CALL  DS  D  1  (UPDATE,&1)
           ENTRY  D  U  (*)
       GO TO (200,10), REQ
           CALL  DS  P  1  (SVT)
10         CALL  DS  D  2  (SVT,UPDATE)
100    CALL  N  S  (1,UPDATE)
       CALL  D  ST
200    RETURN1
       END
           SUBROUTINE  N
       COMMON/ALL/ TIME
       COMMON/SBRDN/ BLANK,REQ
1      IF (TIME.NE.UPDATE) GO TO 100
           ENTRY  N  N  (*)
       CALL  DS  N  L  (UPDATE, &1)
           ENTRY  N  U  (*)
       GO TO (200,10,11), REQ
10         CALL  DS  P  1  (SVT)
           CALL  DS  N  2(SVT,UPDATE,&100)
11         CALL  DS  N  3  (UPDATE,&100)
100    CALL  N  S  (2,UPDATE)
       CALL  N  ST
       RETURN1
       END
           SUBROUTINE  P
       COMMON/ALL/ TIME
       COMMON/PN/ UPDATE(2), NEXT
1      IF(TIME.NE.UPDATE(NEXT)) GO TO 100
           ENTRY  P  N  (*)
       GO TO (2,3,4,...), NEXT
2          CALL  PI(1)  N  (&1)
3          CALL  PI(2)  N  (&1)
4          CALL  P3     N  (&1)
           ENTRY  P  U  (*)
10     CALL  P  BR  (&11,&100)
11          CALL  PI(1)  U  (&12)
12          CALL  PI(2)  U  (&13)
13          CALL  P3     U  (&100)
100    CALL  N  P  (1,UPDATE(NEXT))
       CALL  P  ST
       RETURN1
       END
         SUBROUTINE  P  BR  (*,*)
       COMMON/CBRPS/ BLANK,REQ
       COMMON/PPIBR/ REQI(2), REQ3
       REQI(1)=1
       REQI(2)=1
       REQ3=1
       GO TO (1,2,3,4,5), REQ
1      RETURN2
2      REQI(1)=2
       RETURN1
3      REQI(2)=2
       RETURN1
4      REQ3=2
       RETURN1
5      REQ3=3
       RETURN1
       END
           SUBROUTINE  PI(J)
       COMMON/ALL/ TIME
       COMMON/PPI/ REQ(2)
1      IF (TIME.NE.UPDATE(J)) GO TO 100
           ENTRY  PI  N  (*)
       CALL  DS  PI1  (J,UPDATE(J),&1)
           ENTRY  P1  U  (*)
       GO TO (200,10), REQ(J)
10     CALL  DS  PI  2  (J)
100    CALL  N  P  (J,UPDATE(J))
       CALL  PI  ST  (J)
200    RETURN1
       END
         SUBROUTINE  PI  ST  (J)
       COMMON/STP/ A(2), HERE(3)
       CALL  DS  PI  (J,A,HERE)
       RETURN
       END
           SUBROUTINE  P3
       COMMON/ALL/ TIME
       COMMON/PPI/ BLANK(2),REQ
1      IF (TIME.NE.UPDATE) GO TO 100
           ENTRY  P3  N  (*)
       CALL  DS  P3  1  (UPDATE,&1)
           ENTRY  P3  U  (*)
       GO TO (200,10,11), REQ
10     CALL  DS  P3  2  (&100)
11     CALL  DS  N  3  (SVT)
       CALL  DS  P3  3  (SVT)
100    CALL  N  P  (3,UPDATE)
       CALL  PI  ST(3)
200    RETURN1
       END
           SUBROUTINE  P  ST
       COMMON/CBRST/ H(3), F
       COMMON/STP/ A(2), HERE(3)
       LOGICAL HERE, H
       F=1
       IF (A(1).GT.A(2)) F=2
       DO 1 I=1,3
1      H(I)=HERE(I)
       RETURN
       END
```

Notes: D ST and N ST simply set DA=0,1 and NA=0,1,2 for unavailable, available, and in the latter case all nurses busy but some are serving P3's.
DS D 1 has a doctor finishing; furnishes possibly new update and DA=1 through D ST.
DS P 1 gets service time from P data structure.
DS D 2 has a doctor begin service; furnishes possibly new update and DA.
DS N 1 similar to DS D 1.
DS N 2 similar to DS D 2.
DS N 3 releases a nurse from last P3 to begin service; furnishes possibly new update time, sets NA=1, and makes remaining service time for bumped patient available.
DS PI 1 has patient arrival, furnishes possibly new update and appropriate HERE and arrival time.
DS PI 2 takes patient out of queue and makes service time available.
PI(1) and PI(2) illustrate an indexed product. This means that every operation in the subsystem or its data structure requires an index. In case where large numbers of routines occur, DO loops can be used. The PI ST routine is used by P3 as a convenience.

Whether the data structures are simple arrays or linked lists, there are no decisions to be made in any of the entry points. Again, the way to understand the model is to make one run. Our experience has been that there is a leveling off in complexity at about this difficulty. In our model of the Southern Railroad we were never forced to have more simulation complexity at any one point. The data processing needed to operate on trains in a given terminal required appropriate expertise to achieve efficiency but in any case it was always at points in the simulation where it was well known what had to be done.

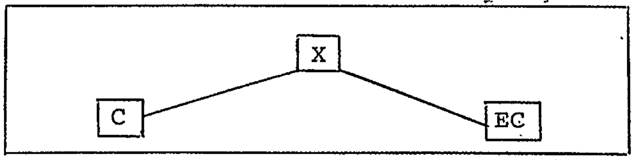
The next two examples indicate the kind of overall complexity which can be achieved with simple BR routines between complex subsystems.

Example 3. Totally Bumping Clinic. This is a clinic which operates like the one in the previous example but in addition, a doctor or nurse can be added or deleted from the system at any time. It can almost be copied from the previous one. The structure is




Each PI is modeled after P3 in the Clinic and each SI is modeled after N of the Clinic. B has all of the bumping requests and C becomes an intermediate. In X BR first check to see if personnel is to be added--this case is easy. Then check if personnel, one at a time, is to be deleted. This would require an indexed bumping. Finally, there would be a command to operate as usual.

Example 4. Emergency Clinic. This has a regular clinic and an emergency clinic, each operating as the TBC. In any case where the emergency clinic needs personnel which exists in the clinic then the transfer is made. When the need no longer exists the reverse transfer is made. The structure is




where C and EC are isomorphic totally bumping clinics. They can be indexed if desired. The X BR simply monitors needs and existence.

## References

[1] AAR Network Simulation System, The: A Tool for the Analysis of Railroad Network Operations. February 1971.

[2] Hodges, H.E., "A Characterizing Model for Discrete Simulations and its Application to a Major Railway System." Ph.D. dissertation, Clemson University, 1977.

[3] Jones, A.F., "A Theory for Discrete Simulations: An Application to Major Railway Systems." Ph.D. dissertation, Clemson University, 1976.

[4] Matar, M.A., "A Comparison of Programming Theories and Languages for Discrete Simulation." Ph.D. dissertation, Clemson University, 1977.

[5] Mertens, G.T., "A Programming Theory for Continuous-discrete Hybrid Simulation." Ph.D. dissertation, Clemson University, 1977.

[6] SIMTRAN: Preliminary User's Manual. Operations Research Department. Southern Railway System. 1973.