Proceedings of the 1986 Winter Simulation Conference
J. Wilson, J. Henriksen, S. Roberts (eds.)

YOU CAN'T BEAT THE CLOCK: STUDIES IN PROBLEM SOLVING

James O. Henriksen
Wolverine Software Corporation
7630 Little River Turnpike
Suite 208
Annandale, VA 22003-2653

ABSTRACT

This paper has a broad purpose and a narrow purpose. The broad purpose is to teach some general techniques of problem solving, and the narrow purpose is to teach a particular approach to modeling. These purposes are attained by presenting a sequence of three problems and solutions to these problems.

Many of the techniques of problem solving presented are taken from, or suggested by, the works of Polya (1957, 1973, 1981). Readers interested in expanding their problem solving abilities should strongly consider reading these works. The particular modeling viewpoint advocated in this paper is that time-oriented approaches to modeling are frequently preferable to space-oriented approaches. Hence, the title of this paper. Unfortunately, for most people, space-oriented approaches to modeling are more natural than time-oriented approaches. By emphasizing the advantages of the time-oriented approach, we hope to broaden the perspective of the reader.

1. ORGANIZATION OF THIS PAPER

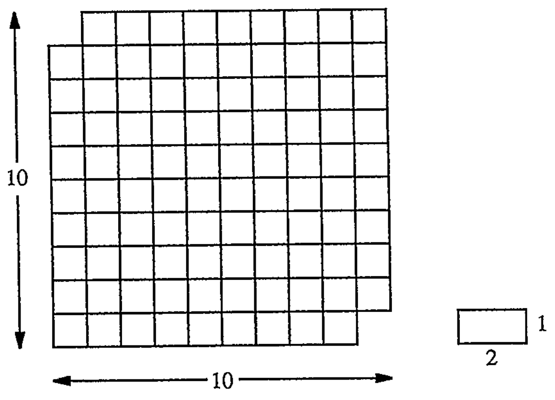
Section 2 is a mental "warmup exercise," not specific to simulation. It presents a problem for which the solution hinges entirely on finding a proper way of viewing the problem. Once the problem is examined from the proper viewpoint, the solution is immediately apparent.

Sections 3 and 4 present examples which illustrate the benefits of modeling system behavior from a time-oriented perspective, rather than a space-oriented perspective. Section 3 presents a classic problem which is often used to teach the concept of recursion in computer science curricula. However, when the problem is examined from the viewpoint of describing behavior of system components over time, as a simulationist would do, an elegant time-oriented solution is immediately apparent. Section 4 presents an example of a conveyor system which is representative of a class of real-world simulation problems. A space-oriented modeling approach which is frequently applied to modeling conveyors is presented. Deficiencies of this approach are discussed, and a time-oriented modeling approach is developed as a sequence of improvements upon the space-oriented approach.

2. FINDING THE PROPER WAY TO VIEW A PROBLEM

2.1 Statement of the Problem

Figure 1 depicts the problem to be discussed in this section: can the grid shown, a 10 X 10 array, with diagonally opposite corners removed, be covered by forty-nine 1 X 2 rectangles? (Rectangles cannot be overlapped; nor can they be sawn in half.) This problem is taken from Kline (1953).



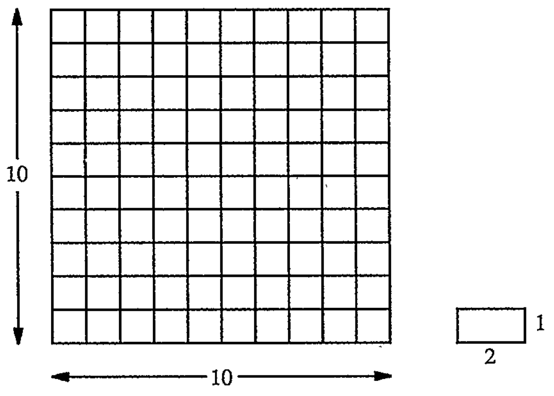
98 SQUARES 49 RECTANGLES

Figure 1: The Problem

The first step in considering this problem is to compare the areas represented by the grid and the rectangles. In both cases, the total area is 98. Thus our first conclusion is that we cannot immediately establish infeasibility; i.e., it <u>might</u> be possible to find a way to cover the grid. At this point, the serious reader should pause to grapple with this problem for several minutes, before proceeding.

2.2 Solving a Simpler Problem

When confronted with a difficult problem, we can sometimes make progress by first considering a simpler problem, then returning to the more difficult problem, applying what we have learned in the process of solving the simpler problem. Figure 2 depicts a simpler problem: if we replace the missing corners of Figure 1 and increase our supply of rectangles to fifty, can the full 10 X 10 grid be covered by fifty rectangles? The answer to this problem is an immediate "yes!" In fact, there are so many obvious solutions to this problem, that one is immediately drawn to the more challenging problem of specifying exactly how many unique solutions exist. Consideration of the latter problem, however interesting, would lead us too far afield.



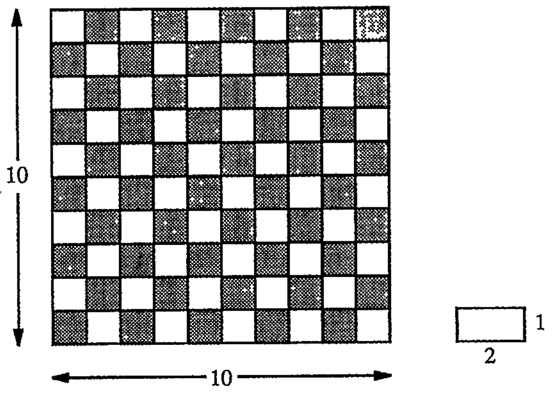
100 SQUARES 50 RECTANGLES

Figure 2: A Simpler Problem

2.3 Does the Problem Remind us of Anything?

For most readers, the grid in Figure 2 is suggestive of a checker board, and the rectangles are suggestive of dominos. (Strictly speaking, the checker board familiar to most of us is an 8 X 8 grid; however, in the game of international checkers, a much more challenging game, a 10 X 10 board is used.) In the rest of this discussion we shall refer to the grid and the rectangles as the checker board and the dominos, respectively.

At this point in our discussion, the checker board lacks one important element of realism: on a real checker board, alternating squares are painted dark and light colors. This shortcoming is rectified in Figure 3.



100 SQUARES 50 RECTANGLES

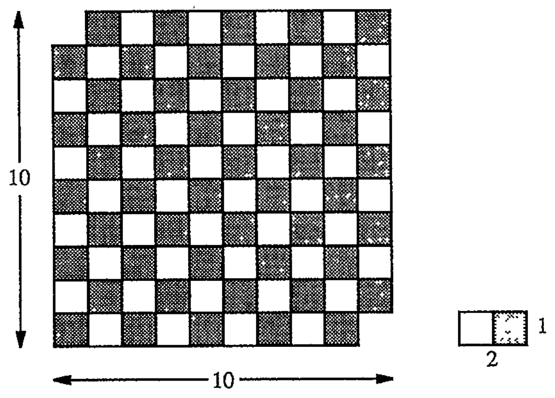
Figure 3: Adding Evocative Shading

2.4 Solving a More General Problem

By examining Figure 3, we can make a general observation about the process of placing dominos on the checker board. (Whether the observation is useful remains to be demonstrated.) No matter what strategy is chosen for placing dominos, every domino covers a pair of adjacent light and dark squares of the checker board. To cover two light squares or two dark squares would require sawing a domino in half, a clear violation of the rules. Whether we've solved anything to this point is arguable; however, at the very least we have made an interesting observation about the general problem of placing dominos on the checker board.

2.5 Applying What We've Learned

At this point, we're ready to apply what we've learned. The first thing we must do is to paint the modified checker board shown in Figure 1. The apparently unremarkable results of this effort are shown in Figure 4.



98 SQUARES 49 RECTANGLES

Figure 4: The Solution Becomes Apparent

Let us examine Figure 4 in light of the general observation we made about domino placement in the previous section. We know that every domino must cover one light square and one dark square. Therefore, forty-nine dominos must cover forty-nine light squares and forty-nine dark squares. The modified checker board in Figure 4 contains forty-eight light squares and fifty dark squares. Alternatively, the board could have been painted so that it had forty-eight dark squares and fifty light squares. In either case, the diagonally opposite squares removed from the full checkerboard are of the same color.

We now have all the information we need to solve the problem. There is no way the forty-nine dominos can be placed on the modified checker board, because of the unequal numbers of dark and light squares. The key to solving this problem was finding the proper way of viewing the problem. Once the problem was properly viewed, the solution was immediately apparent.

3. A TIME-BASED SOLUTION TO A LEGENDARY PROBLEM

3.1 The Legend

Figure 5 depicts a legendary problem, the Towers of Hanoi. The rules of the game are as follows:

(1) A group of N disks of graduated diameters is initially situated on peg number 1.
(2) The objective of the game is to move the disks, one at a time, until all disks have been moved to peg number 2 (or 3).
(3) At no point can a larger disk be placed on top of a smaller disk.

According to an ancient Brahmin legend, a group of monks works around the clock, moving golden disks in a game where N=64. When all disks have been moved to peg 2, the world will come to an end. This is a game of cosmic proportions.

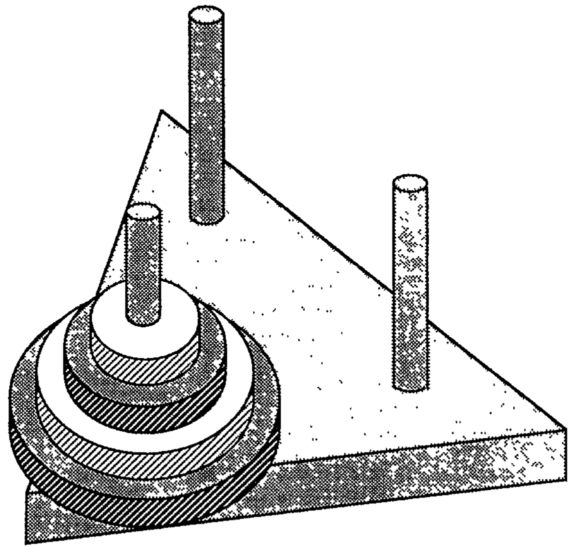


Figure 5: The Towers of Hanoi Problem

With a little experimentation, one soon discovers that when disks are moved optimally, completing an N-disk game requires 2**N-1 moves; i.e., a 6-disk game requires 63 moves. A 64-disk game requires 2**64-1 (18,446,744,073,709,551,615) moves. If the monks are capable of making one move per second, completing the game will take them approximately 584,000,000,000 years.

3.2 The Problem

The problem to be considered in this section is to design a computer program to print the steps of an N-disk Towers of Hanoi game.

3.3 The Traditional Recursive Approach

The Towers of Hanoi problem is frequently used in computer science curricula as a vehicle for illustrating the utility of recursive algorithms. (See Dromey (1982) for a typical treatment.) A summary of the recursive approach follows. Suppose we face the problem of moving N disks from peg 1 to peg 2. This problem can be decomposed into three subproblems:

(1) Move N-1 disks from peg 1 to peg 3.
(2) Move the disk remaining on peg 1 to peg 2.
(3) Move N-1 disks from peg 3 to peg 2.

Subproblem (2) is trivial. Subproblems (1) and (3) are alike, in that they both require moving N-1 disks. Thus, we can reduce an N-disk problem to an N-1-disk problem. To solve the N-disk problem, we pause to solve an N-1-disk problem. To solve the N-2-disk problem, we must pause to solve an N-3-disk problem. We continue this pattern until we have reduced the original problem to a 1-disk problem, for which the solution is trivial. When we have solved the 1-disk problem, we can resume the deferred 2-disk problem. As we work our way back up the list of deferred problems, additional problems will have to be deferred, but eventually, we work our way back to the original problem, having completed all deferred problems, and the game is over.

Figure 6 traces the pattern used to solve the problem of moving four disks from peg 1 to peg 2. Disks are numbered from top to bottom; i.e., disk number 1 is the smallest. Underscored steps are steps which are decomposed into subproblems. Disk moves are numbered from 1 through 15. (Recall that a 4-disk game requires 2**4-1 moves.)

3.4 Non-Recursive Solutions

Despite the age of the Towers of Hanoi Problem, a great deal has been published about the problem in the past several years (Eggers (1985), Franklin (1984), Floriani (1984), Hudson (1984), Konopasek (1985), Mayer and Perkins (1984), Meyer (1984), and Stadel (1984)). The articles cited all present alternative, non-recursive solutions to the problem.

3.5 A Time-Oriented Approach

In this section, a time-oriented approach to the Towers of Hanoi problem is presented. To the author's knowledge, this approach was first suggested by Birtwistle (1985).

Let us consider the recursive solution to the Towers of Hanoi Problem in broad philosophical terms. The problem was viewed as one of describing the behavior of a complicated system, as a whole. The complexity of the problem was dealt with by systematically reducing large problems into smaller problems, until easily solvable problems remained. The recursive solution is a top-down approach. By contrast, the time-based approach to the Towers of Hanoi problem is a bottom-up approach. Rather than taking as an initial goal the task of describing the behavior of the system as a whole, we take as an initial goal the task of describing the behavior of an individual disk. If we can describe the behavior of each disk, we can collectively describe the behavior of the system.

```
    Move disks 1...4 from peg 1 to peg 2

      Move disks 1...3 from peg 1 to peg 3

        Move disks 1 & 2 from peg 1 to peg 2

1.        Move disk 1 from peg 1 to peg 3

2.        Move disk 2 from peg 1 to peg 2

3.        Move disk 1 from peg 3 to peg 2

4.      Move disk 3 from peg 1 to peg 3

        Move disks 1 & 2 from peg 2 to peg 3

5.        Move disk 1 from peg 2 to peg 1

6.        Move disk 2 from peg 2 to peg 3

7.        Move disk 1 from peg 1 to peg 3

8.    Move disk 4 from peg 1 to peg 2

      Move disks 1...3 from peg 3 to peg 2

        Move disks 1 & 2 from peg 3 to peg 1

9.        Move disk 1 from peg 3 to peg 2

10.       Move disk 2 from peg 3 to peg 1

11.       Move disk 1 from peg 2 to peg 1

12.     Move disk 3 from peg 3 to peg 2

        Move disks 1 & 2 from peg 1 to peg 2

13.       Move disk 1 from peg 1 to peg 3

14.       Move disk 2 from peg 1 to peg 2

15.       Move disk 1 from peg 3 to peg 2

Game Complete.
```

Figure 6: Trace of Optimal Solution for a 4-Disk Towers of Hanoi Problem

In Figure 6, disk moves are numbered from 1 through 15. Let us assume that disks can be moved at a rate of one per second. Using this assumption, move numbers can be interpreted in the time domain; e.g., the Nth move takes place N seconds into the game. Following a common practice in simulation, let us assume an implicit time unit (seconds) and an implicit time origin (time zero). Under these assumptions, the Nth move is described as "taking place at time N." Finally, we choose to view the disks as active objects, rather than passive objects; i.e., we say "disk N moves," rather than "disk N is moved." The active-object world-view is used in transaction-flow and network simulation languages. For a discussion of world-views in simulation languages, see Henriksen (1981).

Let us consider the behavior of disk 1 over time. By examining Figure 6, we see that it first moves at time 1, from peg 1 to peg 3. At time 3, it moves from peg 3 to peg 2. At time 5, it moves from peg 2 to peg 1. At time 7, it moves from peg 1 to peg 3. The behavior of disk 1 can be summarized as follows:

(1) Disk 1 first moves at time 1.
(2) Disk 1 moves at every odd-numbered time; i.e., the time between moves of disk 1 is a constant 2.
(3) Disk 1 moves in a clockwise direction; i.e., it moves from peg 1 to peg 3 to peg 2 to peg 1, *ad infinitum*.

Next, we consider the behavior of disk 2 over time. By examining Figure 6, we see that it first moves at time 2, from peg 1 to peg 2. At time 6, it moves from peg 2 to peg 3. At time 10, it moves from peg 3 to peg 1. At time 14, it moves from peg 1 to peg 2. The behavior of disk 2 can be summarized as follows:

(1) Disk 2 first moves at time 2.
(2) The time between moves of disk 2 is a constant 4.
(3) Disk 2 moves in a counterclockwise direction; i.e., it moves from peg 1 to peg 2 to peg 3 to peg 1, *ad infinitum*.

In general, the behavior of disk N over time is as follows:

(1) Disk N first moves at time 2**(N-1).
(2) The time between moves of disk N is a constant 2**N.
(3) If N is odd, it moves clockwise; otherwise it moves counterclockwise. Note that the direction of motion depends on the destination peg. In table 6, the destination peg was peg 2. If the destination peg had been peg 3, odd-numbered disks would have moved counterclockwise, and even-numbered disks would have moved clockwise. The direction of motion also depends on the number of disks. For example, if we wanted to move seven disks from peg 1 to peg 2, odd-numbered disks would move counterclockwise, and even-numbered disks would move counterclockwise.

From the above description, an active object simulation program for the Towers of Hanoi problem can be constructed as follows:

(1) Create N active objects, one per disk.
(2) Schedule the first move of each disk I to take place at time 2**(I-1).
(3) Set the constant time between moves for each disk I to 2**I.
(4) If the number of disks is even, and the destination peg is peg 2, or if the number of disks is odd and the

destination peg is peg 3, set the direction of motion for all odd-numbered disks to be counterclockwise, and set the direction of motion for all even-numbered disks to be clockwise. If the number of disks is odd, and the destination peg is peg 2, or if the number of disks is even and the destination peg is peg 3, set the direction of motion for all odd-numbered disks to be clockwise, and set the direction of motion for all even-numbered disks to be counterclockwise.

(5) Schedule an end-of-simulation event at time 2**N-1.

A Towers of Hanoi program written in GPSS/H (Henriksen & Crain (1983)) is shown in Appendix A. The inner loop of the program is four statements long. One of the four statements prints a trace of the current move. The elegance of this approach is apparent in the compactness of the program. You can't beat the clock!

The GPSS/H program was executed on an Amdahl 5860 computer, a very powerful mainframe machine. The program required about 7.5 microseconds of CPU time per move (with trace output suppressed). If the monks of the legend were replaced by the GPSS/H program, the world would end after 4,380,000 years of program execution. In fairness to the monks, it must be pointed out that even if a computer could be made to run non-stop for 4,380,000 years, hardware limitations would prevent the program from completing a 64-disk game. To work properly, the program requires 64 bits of precision in its time calculations. GPSS/H uses a double precision floating point clock. On the Amdahl 5860, double precision arithmetic affords 56 bits of precision. Once the simulator clock reaches a value of 2**56, the program will go into an infinite loop, because adding a time increment of 1 to 2**56 yields a result of 2**56, due to limited precision. World without end, amen!

4. MODELING A RANDOM ENTRY CONVEYOR SYSTEM

4.1 DESCRIPTION OF THE SYSTEM

Figure 7 portrays a hypothetical conveyor system. The operation of the system is as follows:

(1) Four pickers work along adjacent, non-overlapping 25-foot sections of a 100-foot conveyor.
(2) Pickers process random orders which require them to pick (retrieve) cartons from a storage area alongside the conveyor. Cartons are placed on the conveyor, to be sent to a loading dock.
(3) The random locations from which cartons are picked are uniformly distributed along the 25-foot area manned by each worker.
(4) Cartons are 11, 17, or 23 inches long, with equal probability. (The curious reader might wonder why lengths of 12, 18, and 24 inches were not chosen. The method behind this madness will be revealed below.)

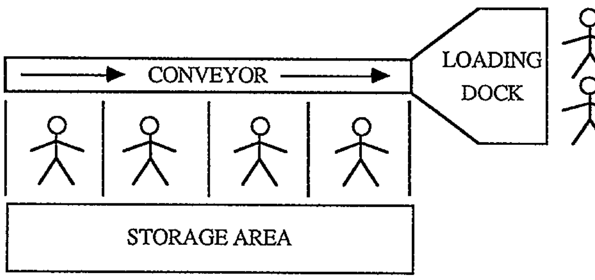


Figure 7: A Hypothetical Conveyor System

(5) Adjacent cartons require a minimum of 12 inches of clearance between them on the conveyor.
(6) Pickers are lazy, in the sense that once they have picked a carton from storage, they move directly to the conveyor, along a path perpendicular to the conveyor, and then wait for a space large enough to accommodate the carton to pass by their current position. Pickers do not roam to the left or right along the conveyor, looking for free conveyor space, even if the conveyor is temporarily stopped due to blockage.
(7) Cartons move along the conveyor until they reach the loading dock.
(8) The loading dock can accommodate eight cartons, independent of carton size.
(9) Two workers remove cartons from the loading dock and place them onto trucks.
(10) The conveyor is a non-accumulating conveyor; i.e., it cannot slide under blocked cartons. If a carton reaches the end of the conveyor, and the loading dock is full, the conveyor is halted until the next carton is removed from the loading dock.

For the most part, details of system timing are irrelevant to the discussion in this paper; however, for those enterprising readers who wish to construct models of this system on their own, hypothetical timing information is provided. The timing has been chosen to provide interesting results, e.g., frequent blockages. The timing of the system is as follows:

(1) The conveyor moves at a speed of one foot per second. (Starts and stops are assumed to be instantaneous.)
(2) The time required for a picker to move from his current position to the location of his next order (in the storage area) is uniformly distributed from 1 to 10 seconds, independent of the actual distance traveled.
(3) The time required for a picker to pick a carton and carry it from the storage area to the conveyor is uniformly distributed between 5 and 10 seconds.
(4) When sufficient free space is available at a picker's current position, he places his carton on the conveyor instantaneously.
(5) It is assumed that cartons are removed from the loading dock instantaneously.

(6) The time required for a worker to move a carton from the loading dock to a truck, and to return to the loading dock is uniformly distributed between 8 and 12 seconds.

4.2 TERMINOLOGY AND CONVENTIONS

The term "leading edge" is used to refer to the right-hand (downstream) end of (1) cartons on the conveyor or alongside the conveyor, ready for placement onto the conveyor and (2) intervals of unoccupied conveyor space. Similarly, the term "trailing edge" is used to refer to the left-hand (upstream) end. When applied to cartons and intervals of unoccupied space, "current position" means "the space from leading through trailing edges." When applied to pickers, "current position" means "current position of the carton he is loading onto the conveyor." The term "free space" is used to refer to intervals of unoccupied space on the conveyor (between cartons, to the right of the first carton, and to the left of the last carton.) If the conveyor is empty, the entire conveyor is free space.

TRUE is used as a synonym for "1", and FALSE is used as a synonym for "0". Saying that a switch is "set" means "set to a TRUE-value," and saying that a switch is "cleared" means "set to a FALSE-value."

Distances along the conveyor are measured as distances from the loading dock.

4.3 MODELING DIFFICULTIES

The discussion in the remainder of Section 4 will focus on two major modeling difficulties presented by the system described above: (1) accounting for the conveyor space consumed by cartons and (2) implementing the rules by which a picker waits for an adequate interval of free conveyor space to flow by his current position. Both of these difficulties are exacerbated by the fact that the conveyor starts and stops due to causes beyond the control of the pickers.

4.4 SOME CONVENIENT MODELING TRICKS

This problem deals with spatial requirements which are expressed in inches, and it includes a conveyor which moves at twelve inches per second. If we were to choose a time unit of one second, the time required for the conveyor to move N inches would be N/12; i.e., move distances would have to be divided by 12 to get move times. Instead, we will use a time unit of 1/12 second. By employing this convention, no scaling is required to convert move distances into move times.

Modeling the 12-inch minimum separation rule can be done by artificially extending the length of each carton by six inches on both ends of the carton; i.e., a 17-inch carton is treated as requiring 29 inches of space. For the rest of this paper, all cartons are assumed to be extended on each end. Thus "leading edge" really means "six inches ahead of the true leading edge," and "trailing edge" really means "six inches behind the true trailing edge."

4.5 A SPACE-ORIENTED MODELING APPROACH

4.5.1 Representing Conveyor Space

A commonly used approach to modeling conveyor systems of the type we are discussing is to partition the surface of the conveyor into small segments. Since we are dealing with packages whose dimensions are expressed in inches, we will divide the conveyor into 1-inch segments. (If we had chosen carton lengths of 12, 18, and 24 inches, all lengths would have been evenly divisible by 6, allowing the use of 6-inch segments.) Using this approach, the surface of a 100-foot conveyor is treated as 1200 1-inch segments.

A simulation program to implement the above approach to accounting for conveyor space could use an array of 1200 TRUE/FALSE switches, with a FALSE-value indicating a free inch and a TRUE-value indicating an occupied inch. In GPSS, for example, a collection of 1200 contiguous Logic Switches could be used. Using this representation, placing a 17-inch carton on the conveyor would require 29 consecutive FALSE-values, beginning at the leading edge of the picker's current carton and extending upstream. (Remember the 12-inch clearance rule.)

4.5.2 Simulating Carton Placement

An algorithm for simulating the placement of a carton onto the conveyor is shown in Figure 8. The description is language-independent (although evocative of the C language), and the algorithm is easy to implement in most simulation languages. Only a "wait-until" capability is required; e.g., GATE LR in GPSS. The detection of delays, for which explicit logic is shown in Figure 8, can be modeled conveniently in GPSS by using a SIM-mode TRANSFER Block.

Several alternative algorithms for searching the switch array were implemented in GPSS. No significant improvements were obtained over the algorithm of Figure 8. In one case, a more sophisticated algorithm (using GPSS SELECT Blocks to search ranges of switches in one fell swoop) performed worse than the simple algorithm. Apparently, the setup costs for the "improved" search were not warranted.

4.5.3 Simulating Conveyor Motion

An algorithm for simulating the motion of the conveyor is shown in Figure 9. Once again, the description is language-independent. The handling of conveyor stoppage is admittedly inelegant, but probably sufficiently accurate. (This is discussed in detail in Section 4.5.4.)

4.5.4 Comments on The Space-Oriented Approach

The approach outlined above has two deficiencies: (1) it consumes enormous amounts of computer time, and (2) it contains

```
delay_incurred = TRUE;                    /* Loop setup */

while (delay_incurred == TRUE)            /* Until every inch of conveyor space required
                                             to hold the carton is free at a single
                                             instant in time */
    {
    leading_edge = displacement (inches from the loading dock)
                   of the leading (downstream) edge of the current
                   carton location;

    trailing_edge = displacement (inches from the loading dock)
                    of the trailing (upstream) edge of the current
                    carton location;

    delay_incurred = FALSE;                     /* No delays this pass */

    for (i=leading_edge to trailing_edge)       /* Scan N contiguous inches */

        if (switch[i] == TRUE)                  /* This inch is occupied */
            {
            delay_incurred = TRUE;              /* Not instantaneous */
            wait until (switch[i] == FALSE);    /* Wait for this inch to clear */
            }
    }
```

Figure 8: A Space-Oriented Algorithm for Simulating Carton Placement

```
while (leading_edge > 1)                    /* Until leading edge of carton occupies
                                               the very last inch of the conveyor */
    {
    wait 1 time unit;                       /* Move 1 inch */

    wait until (conveyor_stopped == FALSE); /* Conveyor may be stopped */

    leading_edge = leading_edge - 1;        /* One inch of motion */
    switch[leading_edge] = TRUE;            /* Inch ahead */

    switch[trailing_edge] = FALSE;          /* Vacated inch */
    trailing_edge = trailing_edge - 1;      /* One inch of motion */
    }

if (loading dock is full)
    {
    conveyor_stopped = TRUE;                /* Stop the conveyor */
    wait until (loading dock is not full);
    }

while (trailing_edge > 0)                   /* Until the trailing edge
                                               is off the conveyor */
    {
    wait 1 time unit;                       /* Move 1 inch */
    switch[trailing_edge] = FALSE;          /* Vacated inch */
    trailing_edge = trailing_edge - 1;      /* One inch of motion */
    }
```

Figure 9: A Space-Oriented Algorithm for Simulating Conveyor Motion

inaccuracies. Enormous amounts of computer time are consumed, because modeling the inch-by-inch progress of cartons down the conveyor requires scheduling an event for every inch of motion for every object on the conveyor. Inaccuracies are of two forms: (1) accounting for space to only the nearest inch and (2) recognizing conveyor stoppages at points in time which are after their actual points of occurrence. Both types of errors are acceptable for the hypothetical system. Accounting for carton positions to the nearest inch is almost certainly acceptable. Recognizing conveyor stoppages within 1/12 second of their actual time is probably also acceptable, unless stoppages occur at an extremely high frequency, in which case the cumulative error may become unacceptable. (If the conveyor stops this frequently, redesign of the real system is probably in order.)

Although the inaccuracies inherent in the space-oriented approach are probably acceptable, the enormous amount of computer time it consumes is probably unacceptable. In the next section, we present some ways of improving performance.

4.6 IMPROVING UPON THE SPACE-ORIENTED APPROACH

4.6.1 Problems With the Space-Oriented Approach

When simulations using the modeling approach of Section 4.5 are run, most of the computer time used is spent updating the data structure used to represent conveyor space. For example, assume that at time 100, a 23-inch carton is to be placed on the conveyor, with its leading edge 301 inches upstream from the loading dock. Further assume that sufficient free space is available at the carton's current position. The space-oriented approach would result in the setting of switches 301...335. (Remember, we add six inches on each end, for clearance.) At time 101, switch 300 would be set and switch 335 would be cleared; at time 102, switch 299 would be set and switch 334 would be cleared; etc. This pattern would continue until the leading edge of the carton reached the loading dock, at which point switches 1...35 would be set. The excessive time spent setting and clearing switches is required because the switch array is a time-dependent data structure; i.e., it must be updated over time.

4.6.2 Making Switch Values Time-Independent

Let's set an optimistic goal: to develop a data structure that requires updating only when a carton is placed on or removed from the conveyor. If switch settings representing a carton are to remain unchanged for the period of simulated time the carton occupies space on the conveyor, the positions represented by the switches must be made time-independent. To do so requires that we modify or augment that data structure. In trying to develop a time-independent representation, we will proceed as follows:

(1) We will make some simplifying assumptions.
(2) We will design a modeling approach which takes advantage of our simplifying assumptions.
(3) We will go back and deal with our simplifying assumptions, one-by-one.

Our simplifying assumptions are as follows:

(1) Assume that the switch array is of unlimited size.
(2) Assume that conveyor stoppage can be ignored.
(3) Assume that the simulator clock is integer-valued.

Using these assumptions, the following modeling approach achieves time-independent interpretation of switch values:

(1) Let switch number I represent the status of inch number I at time zero.
(2) Let switch number I represent the status of inch number I-1 at time 1. In other words, let switch number I+1 represent inch number I at time 1.
(3) Generalizing upon (2), let switch number I represent the status of inch number I-T at time T. In other words, let switch number I+T represent inch number I at time T.

To illustrate this modeling approach, let us reconsider the example of section 4.5.1. At time 100, a distance of 301 inches from the loading dock is represented by switch number 401. Thus, switches 401...435 would be set to represent the space occupied by the carton on the conveyor. At time 401 (assuming no conveyor stoppage), switch 401 would represent the first inch of the conveyor, i.e., the inch immediately to the left of the loading dock. By revising the use of the switch array, we have eliminated the need for updating the array between placements and/or removals of cartons from the conveyor.

We now turn our attentions to assumptions (1), (2), and (3). An unlimited array (assumption (1)) is needed, because as time increases, successively higher indices are used to access the array. For the 1200-inch conveyor of the hypothetical system, if the simulation is to run to time T, an array of size 1200+T is needed. We can get around this problem by noting that after time T, elements 1...T of the array are unused, i.e., available for reuse. This suggests treating the switch array as a circular array, as follows:

> Let switch number (I+T) modulo L represent the status of inch number I at time T, where L is the length of the conveyor, in inches.

X modulo Y is defined as X minus the integer portion of X / Y. If X and Y are integer values, X modulo Y is the remainder of X / Y. Most programming languages provide a primitive for performing modulus division operations upon integer operands, e.g., the MOD built-in function of Fortran. If X and/or Y are floating point values, an analogous floating point operation must be used, e.g., FMOD in Fortran. Note that for integer operands, the expression I modulo J has values ranging from 0...J-1; i.e., the remainder after dividing by J is at most J-1. When implementing the switch array in languages which do not allow zero-valued array indices, the expression (I+N) modulo L + 1 can be used. By introducing circular use of the switch array, we have eliminated the need for assumption (1) above.

We now turn our attention to assumption (2), (ignoring conveyor stoppages). At any given point in simulated time, the length of time the conveyor has been moving is the current simulated time, minus the amount of time the conveyor has been stopped. In other words, down-time doesn't count. Down-time can be incorporated into our revised modeling approach as follows:

> Let switch number (I+T-D) modulo L represent the status of inch number I at time T, where L is the length of the conveyor, in inches, and D is the cumulative down-time for the conveyor.

To illustrate the incorporation of down-time, let us once again reconsider the example of section 4.5.1. Leaving all other assumptions as before, assume that a conveyor blockage of 20 time units' duration occurs at time 150. Further assume that this is the first blockage to occur. At time 150, the leading edge of the carton is 251 inches from the loading dock. The status of inch 251 is represented by switch (251+150-0) modulo 1200, i.e., switch 401. As long as the conveyor is blocked, simulated time and cumulative down-time increase and decrease, respectively, at identical rates. For example, at time 160, the status of inch 251 is represented by switch (251+160-10) modulo 1200, i.e., (still) switch 401. Assuming resumption at time 170, and no further blockages, the leading edge of the carton will reach inch 1 of the conveyor at time 420. At this time, the status of inch 1 will be represented by switch (1+420-20) modulo 1200, i.e., switch 401.

We now turn our attention to assumption (3), (an integer-valued clock). If the simulator clock is implemented as a floating point variable, this assumption can be dealt with by simply truncating time values used to index into the switch array. For example, switch 1 represents the status of inch 1 from time 0 through time 0.99999...

An algorithm which implements the modeling approach developed above is shown in Figure 10. The corresponding algorithm for simulating conveyor motion is shown in Figure 11. An important assumption is implicit in these algorithms: <u>all time delays must be increased by any additional down-time experienced after the start of the time delay</u>. Depending on the language used, this can be difficult to do straightforwardly. For example, in GPSS, scheduled time delays can be lengthened by using the FUNAVAIL Block. These statements are capable of interrupting Transactions (units of traffic), but only if the Transactions are in control of a single server entity, called a Facility; i.e., interruptions are server-based. In the hypothetical system under consideration, a GPSS program could easily be modified to have the pickers acquire (SEIZE) a Facility prior to initiating their delay (ADVANCE) to wait for free space to come by. The Facility could be freed (RELEASEd) upon completion of the delay. Whenever the conveyor experienced blockage, a FUNAVAIL Block could be used to extend the delay times of all active pickers currently waiting for free space.

The GPSS approach works well for extending time delays for the pickers, but is difficult to implement for extending travel time delays experienced by cartons. The difficulty stems from the fact that FUNAVAIL operates on Facilities. To use FUNAVAIL would require SEIZEing a Facility for each carton on the conveyor. Since cartons come and go, a dynamic pool of Facilities would have to be managed by the GPSS program.

To overcome the difficulties in extending delay times, the technique of <u>optimistic scheduling</u> can be used. This technique works as follows:

(1) Record any data necessary for subsequent computation of the amount of time by which a scheduled event must be delayed; e.g., record the current cumulative down-time.
(2) Schedule an event <u>optimistically</u>, hoping that no changes will occur which affect its scheduled time.
(3) At the scheduled time, test to see whether current conditions match those recorded in step (1). For example, if the current cumulative down-time exceeds that recorded in step (1), additional delay must be accounted for. If a change has occurred, calculate the amount by which the time delay must be extended, and return to step (1). Otherwise, the scheduled event can take place at the present time.

A language-independent algorithm implementing this technique is shown in Figure 12.

4.6.3 The Payoff

GPSS models were constructed to test the space-oriented and improved approaches to modeling the hypothetical conveyor system. For runs simulating 10 minutes' operation of the system, the space-oriented approach consumed 54 seconds of CPU time on a VAX 11/750, and the improved approach consumed 5 seconds.

4.7 A TIME-ORIENTED APPROACH

4.7.1 Eliminating the Switch Array

Although the improved modeling approach yielded a tenfold improvement in execution time, there exists an even better modeling approach. The major advantage of the improved approach is the drastic reduction in the number of times switches have to be modified. However, even in the improved approach, large numbers of switches must be manipulated for each carton. Following our "frontal assault" style of solving problems, let us assume that the switch array is to be completely eliminated, and consider what data structures would be required to take its place.

An interval of free space on a conveyor can be represented by two quantities: its position and its length. Accordingly, we propose the following general approach to representing free space:

(1) Assume that initially, a huge interval of free space extends from the loading dock to beyond the left end of the conveyor. The interval must be large enough so that its trailing edge never gets close to the left end of the conveyor during a simulation run.

```
FOREVER
    {
    leading_edge = displacement (inches from the loading dock)
                   of the leading (downstream) edge of the current
                   carton location;

    trailing_edge = displacement (inches from the loading dock)
                    of the trailing (upstream) edge of the current
                    carton location;

    FOREVER
        {
        /* Compute the index of the switch representing the leading edge of the carton */

        ifirst = (leading_edge + current_time - conveyor_down_time) MOD conveyor_length;

        /* Compute the index of the switch representing the last inch of the conveyor */

        ilim = (conveyor_length + current_time - conveyor_down_time) MOD conveyor_length;

        if (ifirst < ilim)                                  /* No wrap-around */
            {
find_start: for (i = ifirst to ilim)
                if (switch[i] == FALSE)                     /* A free inch has been found */
                    {
                    jlim = i + carton_size - 1;             /* Last inch */
                    for (j = i to jlim)
                        if (switch[j] == TRUE)              /* An occupied inch */
                            {
                            ifirst = j + 1;                 /* Next inch */
                            goto find_start;
                            }

                    exit the loop on i;
                    }
            }
        else
            {
            Perform search in two pieces, one scanning from ifirst to the end of the
            switch array, and the other (if necessary) from the start of the array
            to ilim.
            }

    delta_t = (i - ifirst + conveyor_length) MOD conveyor_length;   /* Allow wrap-around */

    if (delta_t == 0)
        exit the FOREVER loop;                              /* Space available NOW */

    wait for delta_t time units to elapse;                  /* See note in text */

    }     /* End of FOREVER loop => go retry the scan */
```

Figure 10: An Improved Algorithm for Simulating Carton Placement

```
wait leading_edge-1 time units;           /* Move to inch 1.  (See note in text.) */

wait until (loading dock is not full);

while (trailing_edge > 0)                 /* Until the carton is off the conveyor */
    {
    wait 1 time unit;                     /* Move 1 inch.  (See note in text.) */
    switch[trailing_edge] = FALSE;        /* Vacated inch */
    trailing_edge = trailing_edge - 1;    /* One inch of motion */
    }
```

Figure 11: An Improved Algorithm for Simulating Conveyor Motion

```
FOREVER
    {
    saved_down_time = cumulative_down_time;

    wait scheduled_delay time units;                            /* Optimistic delay */

    if (cumulative_down_time == saved_down_time)
        exit the FOREVER loop;

    scheduled_delay = cumulative_down_time - saved_down_time;   /* Make up difference */
    }
```

Figure 12: The Optimistic Scheduling Algorithm

(2) Represent each interval of free space with an object having position and length attributes. Position can be defined as either the leading or trailing edge position.
(3) Place these objects in a list sorted by position.
(4) When a carton is to be placed on the conveyor, search the list to find an interval whose trailing edge is at, or to the left of, the trailing edge of the carton, and whose leading edge is at, or to the right of, the leading edge of the carton. Remember that leading and trailing edges have been adjusted by six inches, to implement the 12-inch clearance rule. By assumption (1), an interval of free space will always exist beyond the left end of the conveyor.
(5) If the leading edge of the selected interval has already reached or passed the leading edge of the current carton, proceed to step (8).
(6) Wait until the leading edge of the interval passes the position of the current carton.
(7) Return to step (4), to retest the selected interval. (During the time delay of step (6), other cartons may have been placed into the interval of free space.)
(8) Alter the length attribute of the object representing the interval of free space, to take into account the space consumed by the current carton. If the current carton is being placed in the middle of an interval of free space, create a new object to represent the interval of free space ahead of the current carton, and place the new object into the list of intervals of free space.

4.7.2 Making the New Data Structure Time-Independent

The above approach to representing free space has a major disadvantage: it is time-dependent; i.e., the data which represents intervals of free space must be updated over time. We've temporarily taken a giant step backward. Our new data structure incorporates the worst feature of the switch array of the space-oriented modeling approach presented in Section 4.5.

To find a way to make the list representation time-independent, let us reconsider our choice of attributes used to characterize intervals of free space. Because the position of an interval of free space is time-dependent, the "position" attribute is not what we really need. What we'd like to have is a current position attribute. We can calculate the current position of an interval of free space if we know a past position and how much time has elapsed since it was at that position. This suggests adding a "time-stamp" attribute to our data structure, where the time-stamp records the time at which the position attribute was recorded. Current position can then be expressed as old position - (current time - time stamp). (As time increases, position decreases, following the convention of measuring distance as distance from the loading dock.)

The above formula for determining current location fails to take conveyor stoppage into account. This is because our statement of how to calculate current position was overly simplistic. We should have said that current position can be calculated if we know a past position and how far the conveyor has moved since it was at that position. The time the conveyor has actually been moving since it was at a previous position is the elapsed time, less down-time incurred since it was at the previous position. This suggests adding a "down-time-stamp" attribute to our data structure, where the down-time-stamp records the cumulative down-time observed at the time at which the position and time-stamp attributes were recorded. Using all three attributes, the current position of an interval of free space can be expressed as old position - (current time - time stamp) + (cumulative down time - down time stamp). Since neither the attributes used to compute position nor the length attribute require updating over time, the data structure for describing intervals of free space is time-independent.

4.7.3 Improving the Search

The search of the list structure requires the following comparison:

```
Compare:  required_space_position

to:       free_interval_old_position
        - (current_time - time_stamp)
        + (cumulative_down_time
             - down_time_stamp)
```

Removing parentheses and regrouping yields the following equivalent comparison:

```
Compare:  required_space_position

to:         free_interval_old_position
          - current_time
          + time_stamp
          + cumulative_down_time
          - down_time_stamp
```

Regrouping once more yields the following equivalent comparison:

```
Compare:  required_space_position
          + current_time
          - cumulative_down_time

to:         free_interval_old_position
          + time_stamp
          - down_time_stamp
```

Note that "free_interval_old_position," "time_stamp," and "down_time_stamp" are all known at the time a position is recorded. Therefore, rather than storing these three attributes separately, we can store the expression free_interval_old_position - time_stamp + down_time_stamp. Thus, a time-independent representation of an interval of free space can be entirely characterized by two values, the above expression and the length of the interval.

The list structure for representing intervals of free space can be implemented in virtually any simulation language. In the worst case, lists of objects could be built "by hand," using two-dimensional arrays. In Simscript II.5 (TM) (Russell 1983), the list structure could be implemented directly as a ranked set. In GPSS, User Chains could be used; however, User Chains are used for holding active objects, called Transactions. As presented above, the search algorithm is designed for searching passive objects. The approach is easily modified to allow use of Transactions to represent intervals of free space. Furthermore, the BV-form of the UNLINK Block can be used to locate a suitable interval of free space in a single statement.

4.7.4 Optimizing the Search

Despite our improvements, the use of a list to represent free space has one disadvantage, compared to the use of a switch array. The switch array is a data structure which is directly indexable. Thus the starting position for any search can be calculated. When the list representation is used, it would appear that the list must be searched from the beginning. A list of objects ordered by a particular common attribute is referred to in computer science literature as a priority queue. Event lists in simulation languages are an example of this class of problems. A great deal has been written about priority queue and event list algorithms. (See Jones (1986) and Henriksen (1983), for example.) By replacing the linear search with an appropriate algorithm, search times can be significantly reduced.

4.7.5 Eliminating the Free Interval List, Per Se

An alternative to maintaining a separate data structure for intervals of free space is to augment the set of attributes used to characterize cartons on the conveyor, to include additional attributes which characterize intervals of free space. In other words, every carton on the conveyor can carry a description of the free space ahead of it. To avoid special cases, an imaginary carton can be placed on the conveyor at an arbitrarily large distance from the loading dock, with an "infinite" interval of free space in front of it. The large distance guarantees that the imaginary carton will never reach the real conveyor, and the infinite size guarantees that there will always be at least one carton with space in front of it sufficient to hold any carton.

4.7.6 The Payoff

A GPSS model was constructed to evaluate the effectiveness of the approaches presented in Section 4.7. For a run simulating 10 minutes' operation, the model consumed 1.75 seconds of CPU time on a VAX 11/750 computer. Recall that the space-oriented model of Section 4.5 consumed 54 seconds of CPU time, and the improved model of Section 4.6 consumed 5 seconds.

Due to space limitations, program listings have not been included herein; however, listings of the models are available from the author on request.

Section 4.7 has presented a detailed analysis of one aspect of modeling conveyor systems. Readers interested in further exploring techniques for modeling conveyor systems efficiently should see (Henriksen & Schriber, 1986).

5. CONCLUSIONS

Section 2 demonstrated the importance of finding the right way of looking at a problem. Section 3 presented a problem for which viewing the problem from a simulationist's time-oriented perspective led directly to a solution. The example of Section 3 is remarkable, in that the time-oriented perspective has rarely been employed to solve this problem. In Section 4, the problem-solving philosophy developed in Sections 2 and 3 was applied to a representative simulation problem. In simulations of the type presented in Section 4, one models the behavior of objects in space and time. A modeling approach commonly used to simulate such systems was illustrated in Section 4.5. The modeling approach overemphasized the spatial aspects of describing system behavior and underemphasized the timing aspects. The major shortcoming in the modeling approach was the use of time-dependent data structures which required frequent updating, resulting in intolerably large execution times for simulation runs. In Section 4.6, time-independent data structures were derived from the data structures developed in Section 4.5, resulting in a significant improvement in execution time. In Section 4.7, a modeling approach was presented, for which the major objective was to develop the best possible time-independent data structures. Expressions including time-dependent variables were carefully constructed so as to minimize the amount of computation required to compare positions of objects in space. The superiority of this time-oriented approach was clearly demonstrated.

6. ACKNOWLEDGEMENTS

Thanks go to Dan Brunner, Bob Crain, Holly Price, and Elizabeth Tucker for their thoughtful suggestions and assistance in the preparation of this paper.

APPENDIX A: A GPSS/H PROGRAM FOR THE TOWERS OF HANOI PROBLEM

```
GPSS/H VAX/VMS RELEASE 0.96 (UG206)      14 AUG 1986    17:49:27     FILE: HANOI.GPS

LINE# STMT#  IF DO  BLOCK#  *LOC      OPERATION        A,B,C,D,E,F,G   COMMENTS

    1     1                           SIMULATE
    2     2                 *
    3     3                 *         TOWERS OF HANOI - TIME-DOMAIN SOLUTION
    4     4                 *
    5     5                 TARGET    SYN              2                            DESTINATION PEG
    6     6                 NDISKS    SYN              4                            NUMBER OF DISKS
    7     7                 DONE      SYN              15                           2**4-1

    9     9              1            GENERATE         AC1,,1,,1,4PF                AT TIME 1,2,4,8,...
   10    10              2  TOP       ASSIGN           IMT,AC1+AC1,PF               TIME BETWEEN MOVES
   11    11              3            ASSIGN           DISKNO,N$TOP,PF              DISK NUMBER

   13    13              4                             (PF$DISKNO_                  ALTERNATES
   14    14              4                             +(TARGET-2)_                 ALLOW FOR TARGET DISK #
   15    15              4                             +(NDISKS@2))_                ALLOW FOR ODD OR EVEN N
   16    16              4                             @2+1                         EVEN/ODD: 1/2 OR 2/1

   18    18              5  DLOOP     BPUTPIC          FILE=SYSPRINT,(_
   19    19              5                             AC1,_                        TIME = MOVE NUMBER
   20    20              5                             PF$DISKNO,_                  DISK NUMBER
   21    21              5                             PF$PEGNO+1,_                         CURRENT PEG
   22    22              5                             (PF$PEGNO+PF$PEGINC)@3+1)            NEXT PEG
   23    23              5   MOVE NUMBER *:  MOVE DISK * FROM PEG * TO PEG *

   25    25              6            ASSIGN           PEGNO,(PF$PEGNO+PF$PEGINC)@3,PF NEXT PEG
   26    26              7            ADVANCE          PF$IMT                       TIME UNTIL NEXT MOVE
   27    27              8            TRANSFER         ,DLOOP                       LOOP FOREVER
   28    28                 *
   29    29                 *         TIMER SEGMENT
   30    30                 *
   31    31              9            GENERATE         ,,DONE,1,,4PF                MOVE ALL N DISKS
   32    32             10            TERMINATE        1                            SHUT DOWN
   33    33                 *
   34    34                 *         RUN CONTROLS
   35    35                 *
   36    36                           START            1,NP                         SUPPRESS DEFAULT OUTPUT
   37    37                           END
```

REFERENCES

Birtwistle, G. (1985). The Coroutines of Hanoi, SIGPLAN NOTICES 20.1, 9-10.

Dromey, R.G. (1982). How to Solve It by Computer. Prentice/Hall International, Englewood Cliffs, N.J., 391-403.

Eggers, B. (1985). The Towers of Hanoi: Yet Another Nonrecursive Solution, SIGPLAN NOTICES 20.9, 32-42.

Franklin, W.R. (1984). A Simpler Iterative Solution to the Towers of Hanoi Problem, SIGPLAN NOTICES 19.8, 87-88.

Floriani, P.J. (1984). Letter to the Editor, SIGPLAN NOTICES 19.12, 7.

Henriksen, James O. (1981). GPSS - Finding the Appropriate World-View. In: Proceedings of the 1981 Winter Simulation Conference (T.I. Oren, C.M. Delfosse, and C.M. Shub, eds.). Institute of Electrical and Electronics Engineers, San Francisco, California, 505-515.

Henriksen, James O. (1983). Event List Management - A Tutorial. In: Proceedings of the 1983 Winter Simulation Conference (S. Roberts, J. Banks, B. Schmeiser, eds.). Institute of Electrical and Electronics Engineers, San Francisco, California, 543-551.

Henriksen, James O. and Crain, Robert C. (1983). GPSS/H User's Manual, Second Edition. Wolverine Software Corporation, Annandale, Virginia.

Henriksen, James O. and Schriber, Thomas J. (1986). Simplified Approaches to Modeling Accumulating and Nonaccumulating Conveyor Systems. In: Proceedings of the 1986 Winter Simulation Conference (J. Wilson, S. Roberts, J. Henriksen, eds.). Society for Computer Simulation, San Diego, California.

Hudson, T.F., Jr. (1984). Letter to the Editor, SIGPLAN Notices 19.8, 18-20.

Jones, Douglas W. (1986). An Empirical Comparison of Priority-Queue and Event-Set Implementations, Communications of the ACM 29.4, 300-311.

Kline, Morris (1953). Mathematics in Western Culture. Oxford Press, New York.

Konopasek, M. (1985). The Towers of Hanoi from a Different Viewpoint, SIGPLAN NOTICES 20.12, 39-46.

Mayer, H., and Perkins, D. (1984). Towers of Hanoi Revisited: A Nonrecursive Surprise, SIGPLAN Notices 19.2, 80-84.

Meyer, B. (1984). A Note on Iterative Hanoi, SIGPLAN NOTICES 19.12, 38-40.

Polya, George (1957). Mathematics and Plausible Reasoning. Princeton University Press, Princeton, New Jersey.

Polya, George (1973). How to Solve It, Second Edition. Princeton University Press, Princeton, New Jersey.

Polya, George (1981). Mathematical Discovery, Combined Edition. (Previously published in two volumes, 1962.) John Wiley & Sons, New York.

Russell, E.C. (1983). Building Simulation Models with Simscript II.5, C.A.C.I., Los Angeles, California.

Stadel, M. (1984). Another Nonrecursive Algorithm for the Towers of Hanoi, SIGPLAN NOTICES 19.9, 34-36.

AUTHOR'S BIOGRAPHY

JAMES O. HENRIKSEN is the president of Wolverine Software Corporation, located in Annandale, Virginia (a suburb of Washington, D.C.) Wolverine Software was founded in 1976 to develop and market GPSS/H, a state-of-the-art version of the GPSS language. Since its introduction in 1977, GPSS/H has gained wide acceptance in both industry and academia. Mr. Henriksen is an Adjunct Professor in the Computer Science Department of the Virginia Polytechnic Institute and State University. He teaches courses in simulation and compiler construction at the university's Northern Virginia Graduate Center. Prior to forming Wolverine Software, he worked for CACI, Inc., where he served as project manager for development of the Univac 1100 Series version of Simscript II.5. Mr. Henriksen is a frequent contributor to the literature on simulation. He has given invited presentations at the Winter Simulation Conference, the Summer Simulation Conference, and at the Annual Simulation Symposium. Mr. Henriksen is a member of ACM, SIGSIM, SCS, the IEEE Computer Society, ORSA, and SME.

James O. Henriksen
Wolverine Software Corporation
7630 Little River Turnpike - Suite 208
Annandale, VA 22003-2653
(703) 750-3910