# An Improved Events List Algorithm

James O. Henriksen

## ABSTRACT

One of the basic overhead activities necessary to carry out a discrete event simulation is the insertion and deletion of event notices (transactions) in/from the event list (future events chain). The insertion of event notices is of particular interest, because it requires a search of the event list to determine the point at which an event notice should be inserted. If a linear search is used, as is the case with most simulation languages, the time consumed by the search can grow as $N^2$, where N is the average size of the event list. In a paper entitled "A Comparison of Event List Algorithms," authors Jean G. Vaucher and Pierre Duval (1) evaluated three algorithms as alternatives to linear search. They concluded that an ". . .indexed list algorithm provides the best overall performance." This paper describes a new algorithm, based on the indexed list algorithm of Vaucher and Duval. The algorithm is adaptive to the distribution of event interarrival times. Results are shown that indicate the algorithm works well with a number of distributions and that shortcomings of the indexed list algorithms are overcome.

## INTRODUCTION

Among the basic overhead activities necessary to carry out a discrete event simulation are the functions performed by event scheduling routines. Unfortunately, the terminology used to describe event scheduling varies widely among simulation languages. For example, the data structure which represents an event is called an event notice in Simscript II.5 (2), but is called a transaction in GPSS (3). Similarly, the set of future events has several different names: "events set" (Simscript II.5), "future events chain" (GPSS), "time file" (GASP IV (4)), and "sequencing set" (Simula 67 (5)). In this paper, the terms "event notice" and "event list" are used.

The operation of a "next event" simulator may be characterized roughly as follows:

1. At model initialization time, one or more events are scheduled by creating event notices and merging them into the event list, in order of ascending time.
2. Overall control of the simulation is assumed by an executive routine of the simulator (the "timing routine" in Simscript).
3. The executive routine updates the simulator clock to register the time of the first (earliest) event notice in the event list.
4. The first event notice is removed from the event list, and user-provided code corresponding to the particular event is invoked. This code makes appropriate changes to the status of the model and may cause new events to be scheduled. Frequently an event will schedule the next occurrence of itself, reusing the event notice identified in step 3.
5. The user-provided code returns to the executive routine.
6. If the event list is not empty, execution continues with step 3. Otherwise, the simulation is terminated.

This characterization is an oversimplification of more elaborate algorithms actually used in some languages. The Simscript programmer will, for example, note that no mention is made of stratification of the event list into event classes. The GPSS programmer will note that the rather complex current events chain scanning rules of that language are completely glossed over. (For an excellent treatment of this subject, see (6).) Nevertheless, the characterization provides a simple framework for further discussion, with enough generality to be applicable to all discrete event simulation languages.

Of particular interest in the operation of a "next event" simulator is the insertion of event notices into the event list, because it requires a scan of other event notices already in the event list, in order to find the proper point of insertion. Most simulation languages (GPSS, Simscript, and GASP, for example) use a simple linear search algorithm, or some simple variation thereof, scanning the event list in order of decreasing event time. For many distributions of event times, the machine time consumed by carrying out such a search is proportional to N, the average number of event

notices in the list. If K * N events are scheduled during the course of a simulation, the machine time spent scanning the event list can be proportional to $N^2$. For models with large values of N, this time can be significant, if not dominant. Fortunately, in many cases the distribution of event times is "well-behaved" with respect to search time, because of a natural tendency of new events to be scheduled for times very near the end (highest time) of the event list.

In a paper entitled "A Comparison of Simulation Event List Algorithms" (1), authors Jean G. Vaucher and Pierre Duval evaluated three algorithms as alternatives to linear search: the post-order tree, end-order tree, and indexed list algorithms. They established two basic criteria for performance: speed of operation and robustness, i.e., insensitivity of the algorithm to the statistical distribution of event times being scheduled. To test the alternative algorithms, six distributions of event times were used in experiments with each algorithm, with the average size of the event list varied from 1 to 200 event notices. They concluded that the ". . .indexed list algorithm provides the best overall performance." Although the indexed list algorithm provided the best overall performance of the algorithms tested, it has a number of deficiencies. In the sections which follow, a brief description of the indexed list algorithm is given, shortcomings of the algorithm are identified, and an alternative algorithm, based on the indexed list technique, is presented. Results of timing comparisons between the indexed list and new algorithms are presented. Finally, a case study is presented, wherein the new algorithm was incorporated into an implementation of GPSS and used to run a model of a front-end communications computer.

## THE INDEXED LIST ALGORITHM

The indexed list algorithm has its roots in a technique used in simulators of digital logic circuitry (7, 8). This type of simulation is characterized by an integer-valued clock and by clusters of many events occurring at precisely spaced times. The indexed list algorithm implements the event list as a circular list, with an array of pointers to dummy event notices "scheduled" at fixed intervals. The following variables are used in the operation of the algorithm:

- CURRENT - A pointer to the "current" event notice
- DELTAT - The size of the fixed time interval between dummy event notices
- I - Work variable
- ICURRENT - An index into PTRVEC, corresponding to the first (earliest) dummy event notice
- LOWERBOUND - The base value to which multiples of DELTAT are added
- PTRVEC - A vector of pointers to dummy event notices
- SEARCH - A work variable which points to the current event notice being examined
- TIME - The simulator clock
- VECSIZE - PTRVEC indices range from zero to VECSIZE.

Figure 1 depicts the operation of the algorithm for DELTAT = 1.0 and VECSIZE = 3. To insert an event notice, the following steps are performed:

1. Compute I = (event time - LOWERBOUND) / DELTAT.
2. If I <= VECSIZE, set I = VECSIZE; otherwise set I = (I + ICURRENT) MOD VECSIZE.
3. Set SEARCH = PTRVEC(I) and perform a linear search, in descending time order, down from the dummy event notice pointed to by SEARCH.

To execute the next event (and update the clock) the following steps are performed:

1. Locate the event notice for the current event by setting CURRENT to point to the predecessor of the dummy event notice corresponding to time "infinity."
2. If the current event notice is not a dummy event notice, skip ahead to step 8.
3. Increment LOWERBOUND by DELTAT.
4. Set ICURRENT to (ICURRENT + 1) MOD VECSIZE.
5. Add VECSIZE * DELTAT to the time of the current dummy event notice.
6. Remove the dummy notice from the event list and splice it back into the list at the point appropriate to its new time. To do this, a search must be made down from the dummy event notice corresponding to time "infinity."
7. Continue with step 2.
8. Update TIME to the time of the current event notice.
9. Remove the current event notice from the event list and process the event.

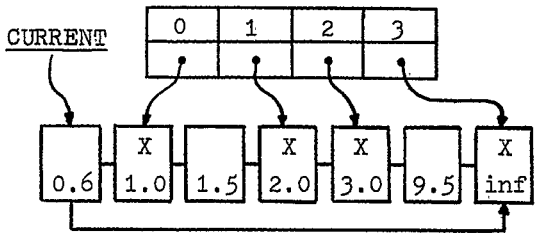
FIGURE 1

Operation of the Indexed List Algorithm

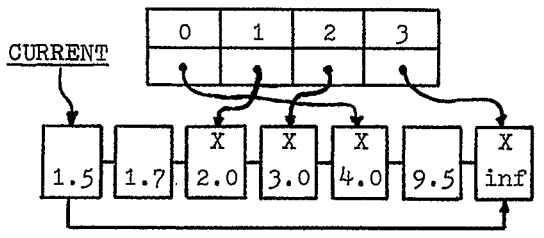
DELTAT = 1.0; VECSIZE = 3

Dummy notices are marked with an "X."

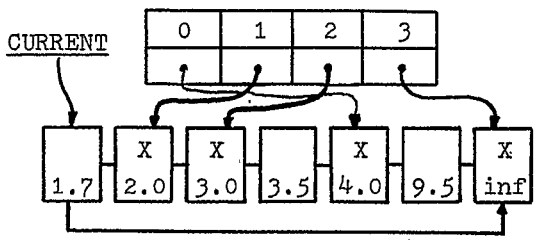
(a) Hypothetical initial conditions
TIME=0.6; ICURRENT=0; LOWERBOUND=0.0




(b) Assume "0.6" rescheduled to "1.7".
TIME=1.5; ICURRENT=1; LOWERBOUND=1.0




(c) Assume "1.5" rescheduled to "3.5".
TIME = 1.7; ICURRENT=1; LOWERBOUND=1.0




## SHORTCOMINGS OF THE INDEXED LIST ALGORITHM

The major shortcoming of the indexed list algorithm is its lack of robustness. Test results (presented below) indicate that the algorithm performs poorly on the "BIMODAL" distribution. Unfortunately, this distribution is representative of a large class of systems. A second shortcoming of the algorithm is that it requires, when implemented as described above, an a priori estimate of DELTAT. Vaucher and Duval present a simple formula for calculating a "good" value for DELTAT, but the formula depends on $\overline{T}$, the average value of "event time - current clock," taken over all event types. This value may not be readily available to the programmer. To avoid having to make an estimate of DELTAT, Vaucher and Duval propose incorporation of an adaptive algorithm to dynamically set DELTAT in accordance with "observed performance." Such an algorithm should also determine the value of VECSIZE. Vaucher and Duval do not specify such an algorithm, but suggest that ". . .dummy notices can be repositioned for a new value of DELTAT with a single pass through the notices, and this reorganization would only be contemplated occasionally, for example, after scheduling 50 notices or passing 50 dummies."

## THE BINARY SEARCH INDEXED LIST ALGORITHM

The binary search indexed list algorithm is similar to the indexed list algorithm, in that a vector of pointers to event notices is maintained, providing an effective partitioning of the event list. Rather than directly computing the index into the pointer vector, however, a binary search of the pointer vector is performed to find an event notice with the smallest time greater than the time of the event notice to be inserted in to the list. Since the vector is searched, the times represented by the elements of the vector are free to "float," rather than being a fixed increment apart. The pointer vector is dynamically updated, continually adapting to the distribution of event notices in the list. Furthermore, provision is made for expansion of the pointer vector, should overall search performance dictate this action. The following variables are used in the operation of the algorithm:

CURRENT - Pointer to the current event notice

I - Work variable

PTRVEC - A vector of pointers to event notices

TIME - The simulator clock

TIMEVEC - A vector of event times, in one-to-one correspondence with PTRVEC

VECSIZE - The current size of the TIMEVEC and PTRVEC vectors

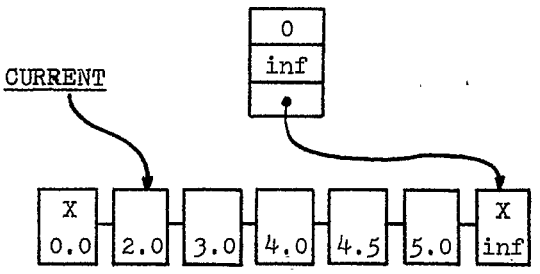
Figure 2 depicts the operation of the binary search indexed list algorithm. To insert an event notice, the following steps are performed:

1. A binary search is made to find a value of I for which TIMEVEC(I) is the smallest value greater than the time of the event notice being inserted. Initially, TIMEVEC(I) is set to zero for all I except TIMEVEC(VECSIZE), which is set to "infinity."

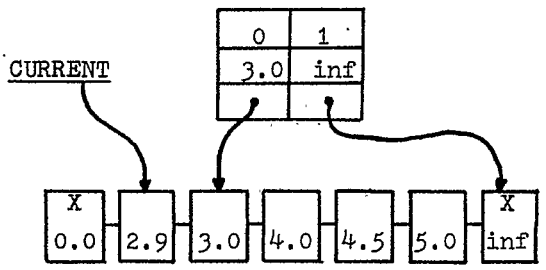
2. A linear search of the event list is initiated, looking "down" from the event notice pointed to by PTRVEC(I). Initially PTRVEC(I) is set to point to a dummy event notice corresponding to time zero, for all I, except PTRVEC(VECSIZE), which is set to point to a dummy event

FIGURE 2
Operation of the Binary Search
Indexed List Algorithm
Dummy notices are marked with an "X."

(a) Hypothetical initial conditions assume that no "pulls" have yet been done.
TIME=2.0; VECSIZE=0




(b) Assume that "2.0" has been rescheduled to "2.9," requiring a "pull."
TIME=2.9; VECSIZE=1




notice for time "infinity." The event list is always anchored by a dummy event notice at each end. These two notices are the only dummies employed.

3. If the linear search examines more than a predetermined number of event notices without finding the insertion point for the new notice, the following steps are taken:

   a. I is set to I - 1.

   b. If I is less than zero, the value of VECSIZE is doubled, PTRVEC and TIMEVEC are reformatted to their initial states (as described in step 2, above), and the entire search is restarted at step 1. Note that dynamic storage allocation may be required to increase the size of the vectors.

   c. If I is greater than or equal to zero, PTRVEC(I) is updated to point to the event notice most recently examined, and TIMEVEC(I) is set to the time of this event notice. This operation is called a "pull."

   d. The linear search is continued at step 3, resetting the count of the number of event notices unsuccessfully examined.

4. The new event notice is spliced into the event list at the point determined by the linear search.

To execute the next event (and update the clock), the following steps are performed:

1. CURRENT is set to point to the successor to the dummy notice at the low end of the event list.

2. This event notice is removed from the list.

3. TIME is updated to the time of the current event notice.

4. The event is processed.

To clarify the operation of the binary search indexed list algorithm, some further explanation is in order. First, the careful reader may question the need for the use of the TIMEVEC vector. It might appear that event times, already contained in their respective event notices, could be referenced indirectly through PTRVEC. This is not possible, because PTRVEC may contain (in its "low time end") pointers to event notices that have been removed from the event list and possibly destroyed. Note that since the binary search always selects a time greater than the time of an event notice being inserted, and since the time of a new event is greater than or equal to the current value of the simulator clock, "stale" pointers are never used, because their corresponding times are always less than or equal to the current value of the simulator clock. A serendipitous benefit of using the TIMEVEC vector is that the binary search is faster, because times are directly addressable, and memory references during the binary search are confined to two vectors, rather than referencing event notices, which may be spread throughout the address space of the computer. In a virtual memory architecture, localized memory references help to cut down on the overhead associated with paging. Another observation that the careful reader might make is that a more intelligent means for setting up pointers could be used following a doubling of the size of the vectors. This was not done,

because (1) doubling the size of the vectors occurs very infrequently, with respect to the time required to carry out an entire simulation; and (2) it is simpler to use a single routine both to initialize PTRVEC and TIMEVEC and to reformat them after a doubling operation.

In the description given above, the phrase "more than a predetermined number of event notices" was used. The exact value of this number is dependent on the ratio of the execution time required to do comparisons in the binary search loop to the time required to do comparisons in the linear search loop. Tests of the algorithm were run on an IBM 370/168 computer to experiment with various values, and the value of 4 was found to be optimal.

One basic operation not discussed above is the removal from the event list of an arbitrary event notice, i.e., a notice not necessarily at the beginning of the list. Such an operation is required to carry out a Simscript "CANCEL" statement and may be required in the execution of a GPSS "PREEMPT" or "FUNAVAIL" block. If the algorithm is implemented as described above, removal of an arbitrary event notice requires that a (binary) search be made to see if any pointers point to the event notice being removed. Any pointers so found must be altered, along with their associated times, to point to the predecessor of the event notice being removed. The use of a dummy notice to anchor the low time end of the list assures that every notice has a predecessor. Several alternative methods were considered. The first alternative was to use dummy event notices in a manner similar to the method employed in the indexed list algorithm of Vaucher and Duval. This method was rejected because the overhead required for manipulation of the dummy notices greatly increases the time required to do a "pull" operation. A second alternative considered was the incorporation of a back pointer in the event notice, pointing from the event notice back to the pointer vector entry associated with the event notice. For event notices not referenced by the pointer vector, the back pointer would be zero. This method provides for rapid removal of an arbitrary event notice, but requires additional storage for the back pointer. Because of the increased storage required and the relative infrequency of arbitrary removals, this method was rejected.

## TESTING THE ALGORITHMS

In order to compare the efficiency of the indexed list and binary search indexed list algorithms, two FORTRAN test programs were written and executed on an IBM 370/168 computer to obtain execution times. The six statistical distributions of Vaucher and Duval were used for interarrival times of event notices, and the size of the event list was varied from 16 to 512 by powers of 2. Results of the timing runs are depicted in figures 3 and 4. The time represented by the Y-axis of these graphs is the time to perform a "hold" operation, i.e., to remove the current event notice from the event list, to reschedule it, and to insert the same event notice back into the event list. The time required to generate random samples and to perform loop counting has been factored out, so the times shown accurately depict the timing of a "hold" operation. Several experiments were conducted using antithetic variates of the statistical distributions. (This method replaces "X" by "1.0-X" in computing interarrival times as a function of "X", where "X" is uniformly distributed over the interval (0,1).) The timing differences between the two forms of calculation were very small. Accordingly, no further attempts were made to incorporate the use of antithetic variates and averaging of results. Each test was run for 4000 repetitions of the "hold" operation.

The indexed list algorithm was implemented, incorporating a priori knowledge of the size of the event list to determine the value of DELTAT, per the technique suggested by Vaucher and Duval. In order to provide a fair comparison, the binary search indexed list algorithm was implemented to incorporate the same information. In each case, VECSIZE was fixed at a value of $\log_2$(list size/4). The adaptive mechanism for doubling the size of the vectors was deliberately suppressed so that relatively short runs could be made and still get meaningful comparisons. Thus, the results represent steady state performance achieved after adaptation.

The distributions of interarrival times tested were as follows:

- EXPO - Negative exponential with a mean of 1.0
- U(0.0,2.0) - Continuous uniform distribution over the interval (0.0,2.0)
- U(0.9,1.1) - Continuous uniform distribution over the interval (0.9,1.1)
- BIMODAL - Uniform over the interval (0.0,S) with 90% probability; Uniform over the interval (10OS, 10lS) with 10% probability. S is chosen so that the mean interarrival time for the distribution time is 1.0.
- D(1) - Interarrival times are a constant 1.0.
- D(0,1,2) - Interarrival times are 0.0, 1.0, or 2.0, with equal probability.

## INTERPRETATION OF TEST RESULTS

The most obvious conclusion that can be made from an examination of figures 3 and 4 is that the binary search indexed list algorithm is far less sensitive to the distribution of interarrival times than is the indexed list algorithm. For the binary search algorithm, the curves are very similarly shaped for all of the distributions tested. The curves for the simple indexed list algorithm, on the other hand, indicate that this algorithm is quite sensitive to the distribution

FIGURE 3

Performance of the Indexed List Algorithm

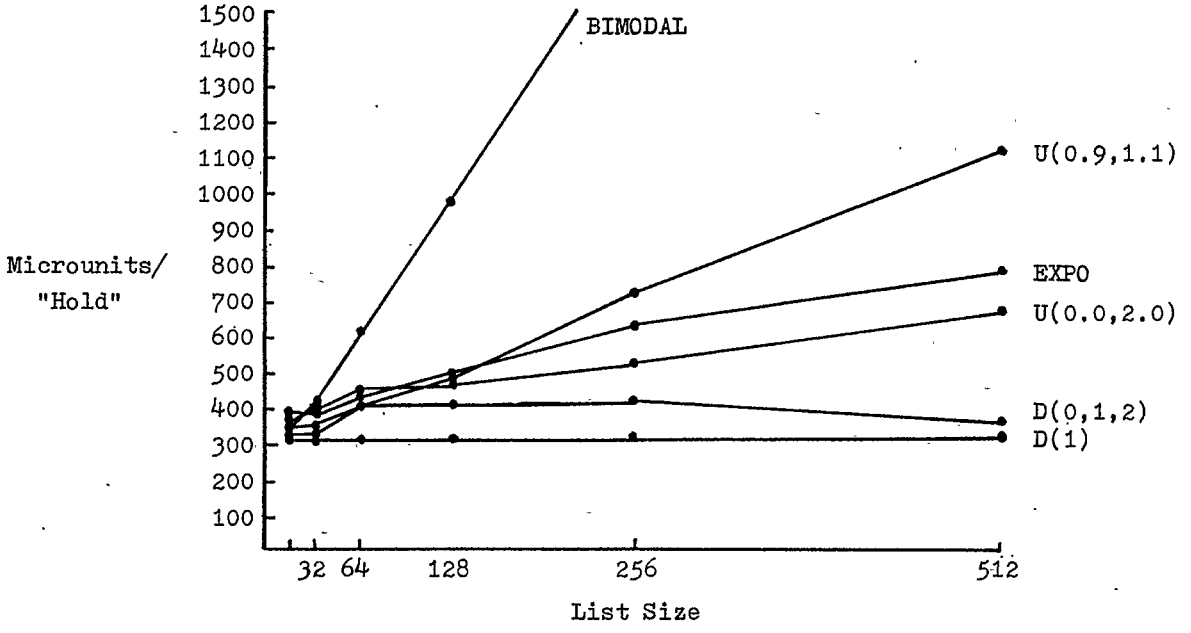
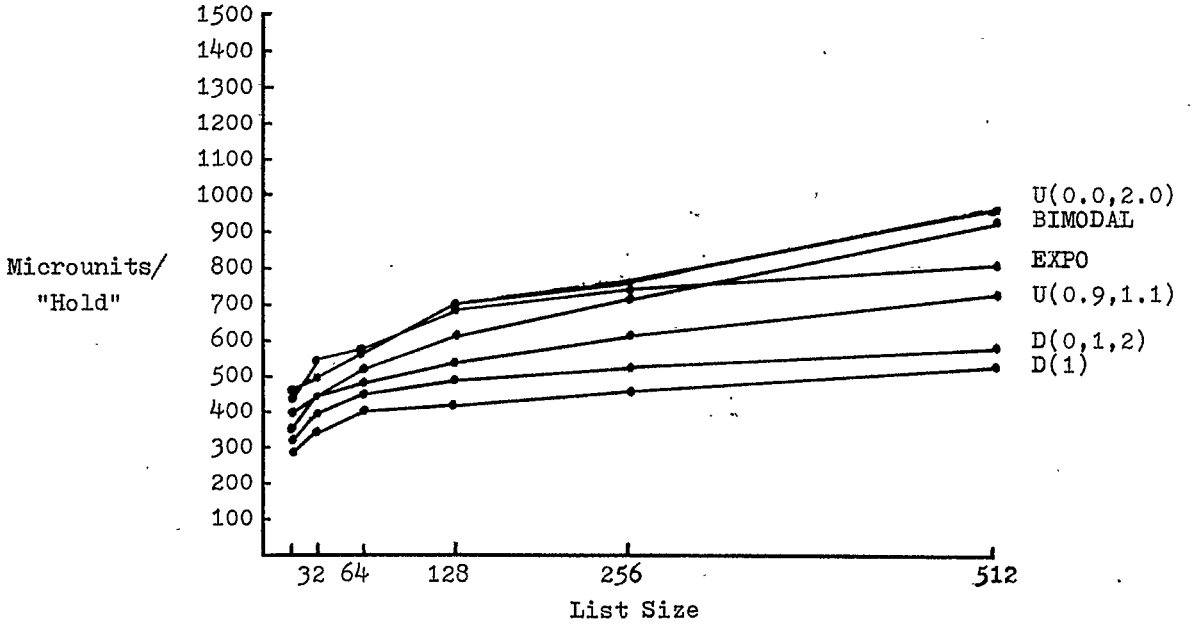



FIGURE 4

Performance of the Binary Search Indexed List Algorithm



of interarrival times, with performance being particularly poor for the BIMODAL distribution.

For the interested reader, detailed statistics of the performance of the two algorithms are presented in Appendix A. These statistics reveal that for large event list sizes, the value of DELTAT, determined by the heuristic formula of Vaucher and Duval, is ill-suited for use with the BIMODAL distribution. For the event list of size 512, only 6 dummy notices were encountered during 4000 "hold" operations. This indicates that DELTAT is far too large. In addition, the size of the overflow class must be quite large, because an average of 283.17 comparisons are required to determine the new placement of a dummy notice, when one is encountered at the front of the list and moved to the back. Since it is apparent that (a) DELTAT is too large and (b) too many event notices fall into the overflow class, more effective performance could be achieved only by decreasing DELTAT and increasing VECSIZE, i.e., the number of dummy notices.

In the previous section, it was indicated that the adaptive mechanism of the binary search algorithm, for doubling the sizes of the vectors, was suppressed for testing purposes. The detailed statistics of Appendix A reveal that, in certain cases, this puts the binary search algorithm at a disadvantage, when compared to the simple indexed list algorithm. For example, the number of comparisons performed to schedule an event notice for the D(1) distribution is $\log_2$ (list size) -1. If the adaptive mechanism had been incorporated, the number of comparisons would have been a constant value of 2 (1 comparison in a single-level binary search and 1 comparison in the linear search). This would compare more favorably with the simple indexed list algorithm, which achieves an average number of comparisons of 1.

One final note should be made in comparing the two algorithms. The test programs for the two algorithms were written in FORTRAN, for the sake of expediency. Since FORTRAN has no pointer variables nor data structuring facilities, the algorithms were implemented by means of vectors and integer subscripts. This has a more adverse effect on the performance of the binary search algorithm than on the simple indexed list algorithm. The major difference between the algorithms is, of course, in the determination of the point at which the linear search of the events list should start. In the simple indexed list algorithm, this is accomplished as a direct computation comprised of a floating point subtraction, a floating point division, and a conversion from floating point to integer. In the binary search algorithm, the determination of the starting point of the linear search is carried out as a binary search of the vector. The FORTRAN implementation of the search manipulates integer subscripts used to access vectors. The code generated by the IBM FORTRAN/H compiler (9) is much better for the simple indexed list computations than for the binary search. If the test programs had been written in assembly language, the relative performance of the binary search algorithm would have been somewhat better. An examination of instruction times for the IBM 360/65 computer (10) indicates that, for that machine, approximately three binary search comparisons could be made in the time required to perform the initial index calculation of the indexed list algorithm. The binary search can be coded very efficiently, using only additive indexing, i.e., no shift or divide instructions. A technique for doing this is given in Appendix B.

## APPLICATION OF THE BINARY SEARCH ALGORITHM - A CASE STUDY

The binary search indexed list algorithm, in its fully adaptive form, has been incorporated into GPSS/H (11), a new implementation of GPSS. Interest in improved events list algorithms was prompted by surprisingly poor performance of GPSS/H on a user's model of a "front-end" communications computer. The program models the performance of a PDP/11 communications computer connected to an Amdahl 470/V6 mainframe. The model incorporates actual historical data for distributions of user "thinking" time, average input message length and typing rate, average response time of the Amdahl 470/V6 to an input request, and the average number and length of output lines resulting from an input line. The range of times that must be represented in the model is quite large. The time that it takes the PDP/11 to respond to an input/output interrupt is roughly on the order of one millisecond, while user "thinking" times are about three orders of magnitude larger, i.e., on the order of several seconds. The distributions of these times are such that very poor performance is achieved with a simple linear search events list algorithm. Consider, for example, the number of users in the model receiving output at any given time. This number is considerably less than half of the total number of users connected to the simulated system. To simulate the behavior of the PDP/11, an input/output interrupt must be scheduled for each character of an output message. Since the interarrival time between such interrupts (for a single user) is quite small, a large number of event notices must be examined in order to determine the insertion point of an event in the "near future."

The results obtained by incorporation of the binary search indexed list algorithm into GPSS/H are quite dramatic. Timing runs were conducted for a configuration of the model which simulated the interactions of 50 terminal users for a period of 60 seconds of simulated time. Prior to the incorporation of the new algorithm, the execution rate of the model was 235 microunits per GPSS block. (The term "microunits" is used because the system on which the tests were run provides "fudged" figures, rather than true CPU times.) After the incorporation of the binary search indexed list algorithm, the execution rate was 158 microunits per GPSS block, a savings of about 30%. This was accomplished, of course, with absolutely no change to the model itself.

## CONCLUSIONS

A new algorithm has been described for the efficient manipulation of events lists in a discrete event simulator. The algorithm has been shown to adapt itself very well to a variety of distributions of event interarrival times. The algorithm has been incorporated into a new implementation of GPSS and has resulted in greatly enhanced performance for classes of models which require a more sophisticated event scheduling algorithm.

## BIBLIOGRAPHY

1. Vaucher, Jean G. and Duval, Pierre, "A Comparison of Event List Algorithms," Communications of the ACM, April, 1975, Volume 18, Number 4.

2. Kiviat, P. J., Villanueva, R., and Markowitz, H. M., Simscript II.5 Programming Language, CACI, Inc., Los Angeles, Cal.

3. GPSS/V User's Manual, IBM Publication Number SH20-0851.

4. Pritsker, A. Alan B., The GASP IV Simulation Language, John Wiley and Sons, 1974.

5. Dahl, O-J, Myhrhaug, B., and Nygaard, K., SIMULA 67 Common Base Language, Publication S22, Norwegian Computing Center, Forskningsveien 1B, Oslo 3, Norway.

6. Schriber, Thomas J., Simulation Using GPSS, John Wiley and Sons, New York, 1974.

7. Szygenda, S., Hemming, S. W., and Hemphill, J. M., "Time Flow Mechanisms for Use in Digital Logic Simulation," Proceedings of the 1971 Winter Simulation Conference, New York.

8. Ulrich, E. G., "Exclusive Simulation Activity in Digital Networks," Communications of the ACM, February, 1969, Volume 12, Number 2.

9. IBM System/360 and System/370 FORTRAN IV Language, IBM Publication Number GC28-6515.

10. IBM System/360 Model 65 Functional Characteristics, IBM Publication Number GA22-6884.

11. Henriksen, James O., "Building a Better GPSS: A 3:1 Performance Enhancement," Proceedings of the 1975 Winter Simulation Conference.

## APPENDIX A

| Distribution | List Size | Average Time | Average Search | Max Search | Dummy Moves | Avg. Dummy Search |
|---|---|---|---|---|---|---|
| EXPO | 16 | 401.87 | 2.86 | 11 | 414 | 1.00 |
| | 32 | 401.87 | 3.30 | 15 | 378 | 1.00 |
| | 64 | 426.99 | 3.82 | 18 | 312 | 1.15 |
| | 128 | 496.48 | 5.02 | 33 | 228 | 3.26 |
| | 256 | 627.93 | 7.51 | 41 | 145 | 12.29 |
| | 512 | 778.63 | 12.52 | 70 | 79 | 37.73 |
| U(0.0,2.0) | 16 | 389.90 | 2.84 | 12 | 410 | 1.00 |
| | 32 | 416.57 | 3.24 | 16 | 375 | 1.00 |
| | 64 | 452.11 | 3.94 | 19 | 310 | 1.00 |
| | 128 | 452.11 | 5.03 | 27 | 227 | 1.00 |
| | 256 | 539.22 | 7.06 | 37 | 145 | 1.00 |
| | 512 | 686.43 | 11.02 | 52 | 81 | 1.00 |
| U(0.0,1.1) | 16 | 376.76 | 1.47 | 11 | 412 | 1.00 |
| | 32 | 376.76 | 2.00 | 17 | 378 | 1.00 |
| | 64 | 405.93 | 3.02 | 37 | 313 | 1.00 |
| | 128 | 480.45 | 5.19 | 79 | 230 | 1.00 |
| | 256 | 728.40 | 10.56 | 125 | 147 | 1.00 |
| | 512 | 1130.27 | 22.35 | 192 | 82 | 1.00 |
| BIMODAL | 16 | 373.39 | 2.41 | 13 | 383 | 1.00 |
| | 32 | 425.23 | 3.95 | 28 | 340 | 1.00 |
| | 64 | 627.93 | 7.24 | 58 | 263 | 25.48 |
| | 128 | 979.57 | 14.39 | 116 | 163 | 74.91 |
| | 256 | 1728.44 | 33.28 | 224 | 100 | 169.10 |
| | 512 | 3435.85 | 82.13 | 459 | 6 | 283.17 |
| D(1) | 16 | 326.52 | 1.00 | 1 | 411 | 1.00 |
| | 32 | 326.52 | 1.00 | 1 | 376 | 1.00 |
| | 64 | 325.87 | 1.00 | 1 | 313 | 1.00 |
| | 128 | 325.09 | 1.00 | 1 | 232 | 1.00 |
| | 256 | 326.52 | 1.00 | 1 | 146 | 1.00 |
| | 512 | 326.52 | 1.00 | 1 | 80 | 1.00 |
| D(0,1,2) | 16 | 352.92 | 1.28 | 96 | 418 | 1.00 |
| | 32 | 366.46 | 1.43 | 96 | 383 | 1.00 |
| | 64 | 401.87 | 1.97 | 96 | 318 | 1.00 |
| | 128 | 401.87 | 2.78 | 96 | 232 | 1.00 |
| | 256 | 427.06 | 3.94 | 96 | 146 | 1.00 |
| | 512 | 346.44 | 1.00 | 1 | 80 | 1.00 |

Detailed Statistics for Performance of the Indexed List Algorithm

| Distribution | List Size | Average Time | Average Search | Max Search | Pulls | Notices/ Pull |
|---|---|---|---|---|---|---|
| EXPO | 16 | 441.97 | 5.40 | 14 | 1098 | 3.64 |
| | 32 | 552.58 | 6.82 | 23 | 1487 | 2.69 |
| | 64 | 577.70 | 8.25 | 36 | 1917 | 2.09 |
| | 128 | 692.31 | 9.73 | 60 | 2354 | 1.70 |
| | 256 | 753.52 | 11.21 | 81 | 2842 | 1.41 |
| | 512 | 828.87 | 12.77 | 152 | 3390 | 1.18 |
| U(0.0,2.0) | 16 | 452.11 | 5.17 | 16 | 1079 | 3.71 |
| | 32 | 497.58 | 6.68 | 23 | 1493 | 2.68 |
| | 64 | 572.43 | 8.26 | 41 | 2003 | 2.00 |
| | 128 | 703.28 | 9.88 | 57 | 2559 | 1.56 |
| | 256 | 778.63 | 11.63 | 99 | 3283 | 1.22 |
| | 512 | 884.49 | 13.54 | 139 | 4174 | 0.96 |
| U(0.9,1.1) | 16 | 321.60 | 3.51 | 8 | 22 | 181.82 |
| | 32 | 452.17 | 5.05 | 13 | 259 | 15.44 |
| | 64 | 477.23 | 6.68 | 19 | 706 | 5.67 |
| | 128 | 550.87 | 8.23 | 39 | 1133 | 3.53 |
| | 256 | 643.91 | 9.88 | 40 | 1696 | 2.36 |
| | 512 | 753.52 | 11.76 | 118 | 2490 | 1.61 |
| BIMODAL | 16 | 401.87 | 4.61 | 18 | 383 | 10.44 |
| | 32 | 456.00 | 5.98 | 34 | 677 | 5.91 |
| | 64 | 530.79 | 7.41 | 65 | 1112 | 3.60 |
| | 128 | 627.93 | 8.98 | 98 | 1611 | 2.48 |
| | 256 | 728.40 | 10.72 | 130 | 2315 | 1.73 |
| | 512 | 858.98 | 12.67 | 175 | 3249 | 1.23 |
| D(1) | 16 | 282.45 | 3.00 | 3 | 0 | 0.0 |
| | 32 | 351.64 | 4.00 | 4 | 0 | 0.0 |
| | 64 | 401.87 | 5.00 | 5 | 0 | 0.0 |
| | 128 | 430.82 | 6.00 | 6 | 0 | 0.0 |
| | 256 | 473.85 | 7.00 | 7 | 0 | 0.0 |
| | 512 | 552.58 | 8.00 | 8 | 0 | 0.0 |
| D(0,1,2) | 16 | 351.64 | 4.10 | 18 | 412 | 9.71 |
| | 32 | 400.23 | 5.29 | 30 | 565 | 7.08 |
| | 64 | 449.95 | 6.15 | 44 | 451 | 8.87 |
| | 128 | 502.34 | 7.10 | 81 | 386 | 10.36 |
| | 256 | 552.53 | 7.94 | 147 | 346 | 11.56 |
| | 512 | 607.81 | 9.03 | 278 | 246 | 16.26 |

Detailed Statistics for Performance

of the Binary Search Indexed List Algorithm

## Appendix B

This appendix describes a means of constructing TIMEVEC in such a fashion that a binary search of the vector can be performed entirely with additive indexing, i.e., no shifts or divisions. To construct a vector of $2^N$ entries, do the following:

1. Assume the entries are indexed 0. . .$2^N - 1$.

2. The offset within the vector corresponding to entry number I is determined by reversing and inverting the rightmost N bits in the binary representation of I. For example, in an eight entry table (N=3), entry number 4 is stored at offset 6 of the vector. ($4_{10} = 100_2$. Inverting and reversing the bits yields a value of $110_2 = 6_{10}$.) Note that as a consequence of this scheme, the "top" logical entry of the vector, which has an index of the form $2^N-1$, will always be located at offset zero of the vector. Reversing and inverting a number whose binary representation is all 1's yields a zero result.

3. If FORTRAN array indexing is used, add 1 to all indices, so that offset zero maps into index 1.

4. It is highly convenient to construct a vector of predecessor pointers, such that $pointer_i$ points to $entry_{i-1}$ of the table. (Because of the strange way in which the vector is ordered, this facilitates an easy determination of the location of a predecessor during a "pull" operation.)

A search of the vector can be conducted as follows:

1. Start with I=1 and J=1 and assume FORTRAN array indexing.

2. For each compare:

   a. If TIMEVEC(I+J) > Time to be inserted

      Set I = I + J

   b. Set J = J + J, i.e., double the increment

   c. The loop is completed when $J=2^N$.

3. At loop completion, I is the index of the "right" entry.

Note that if the search "falls through," with no times > the time being inserted, I = 1, corresponding to a logical index of $2^N-1$, the "top" entry of the table. By definition of the algorithm, this entry always points at a dummy notice with infinite event time.

Finally, note that on a machine which has an "add-compare-conditional branch" instruction, such as the BXLE instruction of the IBM 360/370 architecture, the doubling of the increment and closing of the loop can be accomplished in a single machine instruction.

To convince yourself that this algorithm really works, consider the following hypothetical example:

| i | $time_i$ | i(binary) | offset(binary) |
|---|---|---|---|
| 0 | 100 | 000 | 111 |
| 1 | 200 | 001 | 011 |
| 2 | 300 | 010 | 101 |
| 3 | 400 | 011 | 001 |
| 4 | 500 | 100 | 110 |
| 5 | 600 | 101 | 010 |
| 6 | 700 | 110 | 100 |
| 7 | inf. | 111 | 000 |

| I | TIMEVEC(I) |
|---|---|
| 1 | inf. |
| 2 | 400 |
| 3 | 600 |
| 4 | 200 |
| 5 | 700 |
| 6 | 300 |
| 7 | 500 |
| 8 | 100 |

Verify that for insertion of an event notice with time = 250, the binary search will converge to I=6.