Proceedings of the 1983
Winter Simulation Conference
S. Roberts, J. Banks, B. Schmeiser (eds.)

EVENT LIST MANAGEMENT - A TUTORIAL

James O. Henriksen

Wolverine Software Corporation
7630 Little River Turnpike - Suite 208
Annandale, VA 22003-2653

Proper management of the event list is critically important for achieving execution efficiency in discrete event modelling of computer, telecommunication, and other frequently occurring classes of systems. For such systems, use of a poor event list algorithm can result in horrendous inefficiency. Conversely, replacement of a poor algorithm with a good algorithm can reduce execution times tremendously. For example, in models of telecommunication systems, total run times can easily differ by as much as 5:1, depending on choice of event list algorithm. Many papers have been written about event list algorithms. Unfortunately, the variance in quality of what has been written closely parallels the variance in performance of the algorithms themselves: everything from scholarly works to utter nonsense has been published.

Among the persistent myths of simulation is the "fact" that major simulation languages fail to make use of advanced event list algorithms. This is patently false. GPSS/H (Henriksen & Crain 1983) has used an improved algorithm (Henriksen 1977) since 1977, and SLAM (Pritsker 1979) has used an improved algorithm (a variant of the GPSS/H algorithm) since the advent of SLAM II in 1981.

Before proceeding, we briefly summarize the sections which follow. Section 1 defines what is meant by the phrase "event list algorithm." Section 2 presents a hypothetical example, to illustrate the spectacular failure of a naive event list algorithm. Section 3 describes a simple alternative algorithm which is effective when applied to the example of Section 2, but lacks generality. Section 4 presents criteria for evaluating event list algorithms. Section 5 describes some alternative general-purpose algorithms which have been devised, and reviews a number of written works about event list algorithms. Section 6 presents a detailed description of the algorithm used in GPSS/H (Henriksen 1977). Finally, Section 7, offers conclusions.

1. EVENT LIST ALGORITHMS DEFINED

In discrete event simulation languages, an event is an instantaneous change in the state of a model at a specific point in simulated time. Most languages include time-ordered lists of events scheduled to take place in future (simulated) time. Depending on the language, such a list may be called the event file, the event set, the future events chain, the sequencing set, etc. We use the phrase "event list" to describe such lists. Specifically excluded from this designation is the GPSS current events chain, which is used to manage events which have the potential to take place at the current instant of simulated time. The elements of an event list may be called event notices, transactions, activation records, etc. We use the phrase "event notice" to refer to such elements. Event notices contain the simulated time at which an event is scheduled to occur, a designation of which event in the model is to occur, and other attributes necessary for occurrence of the event.

The design of an event list representation includes definitions of event notices and other supporting data structures, such as list origins and terminators. An event list algorithm comprises procedures for insertion and deletion of event notices from the list and for manipulation of supporting data structures. In the sections which follow, emphasis is placed on procedures for insertion of event notices. Deletion of event notices is of lesser importance, because in most cases, event notices are deleted in sequential order from the front of the list. However, most languages provide mechanisms for removal of arbitrary event notices from positions other than the start of the event list. For example, the Simscript CANCEL statement and the GPSS PREEMPT and FUNAVAIL statements can cause such removals. While the removal of event notices from arbitrary positions in the event list may require special actions, in most models, such processing occurs much less frequently than "normal" event list processing. Hence, we emphasize procedures for event notice insertion and mention procedures for deletion only where they are important.

2. SPECTACULAR FAILURE OF A POOR ALGORITHM

2.1 The Most Commonplace Poor Algorithm

The most commonplace algorithm for insertion of event notices into the event list is linear search in descending time order. The algorithm works as follows: Event notices are chained together on a doubly linked list (with forward and backward pointers). When an event is scheduled to occur at a simulated time T, the event list is searched in decreasing time order, from the event notice for the event (if any) currently scheduled to occur at the most distant point in simulated time, down to the event notice for the event (if any) currently scheduled to occur at the next imminent instant of simulated time. The event notice is inserted after the first event notice (if any) found with a time value less than or equal to T. To facilitate the search process, dummy event notices with times of -infinity and +infinity may be used to anchor the bottom and top ends of the event list. The use of such dummy notices assures that any "real" event notice will always have a predecessor and a successor.

The rationale given for use of linear search in descending time order is that as the execution of a model moves forward in simulated time, events are scheduled further and further into the future, and that while the insertion of some event notices may require searches of moderate length, the insertion of most event notices will require relatively short searches. Research reported by McCormack (1979) and McCormack & Sargent (1981) reveals that the distribution of search lengths for event list insertion operations is far more complex than these naive, but seemingly well-motivated assumptions would seem to indicate. As we shall see in Section 2, models which violate these assumptions are commonplace.

While the use of linear search in descending time order is commonplace, it is by no means universal. The "fact" that all major simulation languages use this algorithm is one of the most persistent myths of simulation literature.

2.2 A Model to Illustrate Algorithm Deficiency

Figure 1 is a state-transition diagram which naively characterizes a timesharing computer system. In this naive model, a fixed number of users endlessly cycle through three states: thinking, typing a command, and receiving output as the result of entering a command. Thinking times are distributed uniformly as 5±5 seconds. Command lengths are distributed uniformly as 10±5 characters, and characters are typed at a uniformly distributed speed of 250±100 milliseconds per character. Output message lengths are uniformly distributed as 175±125 characters, and they are assumed to be transmitted at a rate of 120 characters per second (1200 Baud). The host computer is assumed to respond instantaneously to all commands, and interference among users is ignored. These assumptions are, of course, totally unrealistic, but our objective is to illustrate performance of an event list algorithm, not to present a highly detailed model of a computer system.

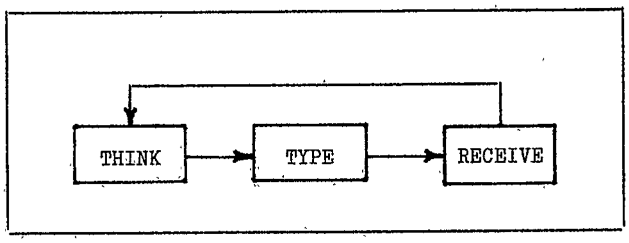

Figure 1: Timesharing User Behavior Pattern

2.3 "Hand Calculator" Analysis of the Model

Using the means of the distributions given above, we can calculate the probabilities that a user will be in each of the three states. The mean length of a cycle is 8.96 seconds (5 seconds thinking + 2.5 (10 / 4) seconds typing + 1.46 (175 / 120) seconds receiving output). Thus, the probabilities of being in the thinking, typing, and receiving states are 0.56 (5 / 8.96), 0.28 (2.5 / 8.96), and 0.16 (1.46 / 8.96), respectively. Since the number of terminal users in the model is fixed, and the user population is homogeneous, the probabilities of a single user being in each of the three states can be used to determine the expected number of users that are in each state at any given time. Thus, at any given time, the expected percentages of users who are thinking, typing, and receiving are 56%, 28%, and 16%, respectively.

Again using distribution means, we can compute the expected percentages of event types that occur. In a cycle, there is exactly one end of thinking time event, an expected value of 10 typing events, and an expected value of 175 receiving events, making an expected total of 186 events per cycle. Thus, the percentages of events of the three types are 0.5%, 5.4%, and 94.1%, respectively.

When a receiving event is scheduled, its event time is always equal to the current time plus 8.3 milliseconds (1/120 second). Given the magnitude of the thinking and typing event interarrival times, we can expect that a linear search in descending time order will "usually" require examination of all ("except perhaps 1 or 2") of the currently scheduled thinking and typing events. Given the constant interarrival time (8.3 milliseconds) between receiving events, each time a receiving event is scheduled, it will become the receiving event with the largest event time. (Ignoring time ties, it must fall after any previously scheduled receiving event.) To summarize our primitive analysis, 94.1% of the time an event is scheduled, we can expect to examine 84% (56% +28%) of the event notices.

2.4 Testing a Poor Algorithm

To test the linear-search-in-descending-order algorithm, the GPSS model shown in Figure 2 was written and run under a version of GPSS/H specially modified to use (and collect statistics upon the operation of) the linear search algorithm. The model was run for configurations of 10, 50, and 100 terminal users. The results are shown in Figure 3. The mean search lengths observed are consistent with the analysis given in Section 2.2, and the dramatic increases in execution times as configuration size increases are readily apparent. In short, performance is wretched.

3. A SOMEWHAT IMPROVED ALGORITHM

The analysis of Section 2.2 suggests a simple alternative to linear search in descending time order: why not use linear search in ascending time order? According to that analysis, if we use linear search in acending order, 94.1% of the events scheduled will require examination of 16% of the event notices.

To test this hypothesis, another specially modified version of GPSS/H was developed to incorporate the linear-search-in-descending-order algorithm. The results shown in Figure 4 are consistent with the foregoing analysis.

4. CRITERIA FOR EVENT LIST ALGORITHMS

4.1 Search Lengths

An event list algorithm should keep search lengths within reasonable bounds. As the example of Section 2 illustrated, simple models can easily result in search lengths which account for most of the CPU time consumed executing the model.

4.2 Speed

An event list algorithm should execute rapidly. If the steps taken to reduce search lengths require complicated computations, the advantages attained by reducing search lengths are diluted.

4.3 Robustness

Any event list algorithm intended for general use should operate well across a wide range of event scheduling distributions. While performance in any given model may be less than that which could be obtained by an application-tailored algorithm, performance should not degrade dramatically. The linear search algorithms discussed in Sections 2 and 3 illustrate this point. In models which have large numbers of event notices in the event list, we can expect that where one of the search algorithms performs poorly, the other will perform better, if not well. Thus, neither algorithm can be considered robust enough for general use.

4.4 Automatic Operation

An event list algorithm which automatically adapts to the number and distribution of event notices in the event list is generally preferable to an algorithm which requires user specification of algorithm parameters. A number of improved event list algorithms (some of which are discussed below) require specification of algorithm parameters. The sensitivity of algorithm performance to parameter values is sometimes disputed: authors may claim relative insensitivity, while others provide empirical counterexamples.

4.5 Testing Procedures

Many papers written about event list algorithms have employed the HOLD model for testing purposes. The HOLD model operates as follows: (A) the event list is initialized by filling it with a number of event notices whose interarrival times are all sampled from a single distribution; (B) transient conditions are removed by repeatedly (several thousand times?) removing the next imminent event from the event set and rescheduling it; and (C) steady state statistics are collected by circulating event notices (in the manner of (B)) a large number of times. McCormack & Sargent (1981) point out that the HOLD model differs from most actual simulations in three ways: (A) in real models, the size of the event list is rarely constant; (B) in real models, events are not independent; i.e., occurrence of an event of one type usually influences the occurrence of events of other types; (C) in real models, events of different types sample their interarrival tinmes from different distributions; i.e., the homogeneity inherent in the HOLD model is atypical.

Despite its deficiencies, the HOLD model does produce useful information, and it is easily programmed. However, the HOLD model cannot be used alone; rather, it must be used in conjunction with other testing procedures. The ultimate test of an algorithm is its performance in real-world models. Therefore, an algorithm should be tested against a variety of well-chosen applications. The frustration inherent in such testing is that one can never prove that an algorithm works by testing it. One can only prove that it fails.

In many of the published analyses of event list algorithms, CPU time has been used as the dominant, if not exclusive, measure of performance. While CPU times provide the ultimate measure of performance (cost), they fail to reveal the components of performance. For example, the collection of search length distributions can help to differentiate inherent algorithm performance from speed gains or losses resulting from the quality of the program which implements the algorithm. If an algorithm has acceptably short search lengths, but runs unacceptably slowly, the implementation of the algorithm should be examined carefully to try to find bottlenecks.

```
        SIMULATE
        REALLOCATE     COM,15000           ENOUGH STORAGE FOR XACTS
*
*       SIMPLIFIED TERMINAL MODEL
*
        GENERATE       ,,,100              100 TERMINALS

THINK   ADVANCE        5000,5000           THINK 5 +- 5 SECONDS

        ASSIGN         1,5+RN1@11,PH       WILL TYPE 5-15 CHARACTERS
TYPE    ADVANCE        250,100             TYPE APPROX 4 CHAR/SEC
        LOOP           1PH,TYPE

        ASSIGN         1,50+RN1@251,PH     RECEIVE 50-300 CHARS
RECV    ADVANCE        8                   1200 BAUD => 8.33 MSEC
        LOOP           1PH,RECV
        TRANSFER       ,THINK
*
*       TIMER SEGMENT
*
        GENERATE       ,,100*1000           SIMULATE FOR 100 SECONDS
        TERMINATE      1
        START          1,NP
        END
```

Figure 2: GPSS Model of Timesharing User Behavior

Number of Users	Number of Events Scheduled	Insertion Search Lengths Average	Maximum	Execution CPU Time / Number of Statements Executed
10	20045	9.15	11	182.14
50	98878	42.24	51	375.98
100	196875	82.77	101	581.33

Figure 3: Results Using Linear Search in Descending Order

Number of Users	Number of Events Scheduled	Insertion Search Lengths Average	Maximum	Execution CPU Time / Number of Statements Executed
10	20045	2.85	11	165.76
50	98878	9.75	51	201.39
100	196875	19.20	101	235.22

Figure 4: Results Using Linear Search in Ascending Order

The failure to observe all pertinent statistics can result in incorrect conclusions. The following (admittedly ridiculous) example illustrates this possibility. Assume that an event list algorithm is to be devised for a simulation which entails scheduling 10,000 events over a simulated time ranging from zero to 1,000,000. One possible algorithm might be to preformat the event list with 100,000 "dummy" event notices, equally spaced 10 time units apart. "Real" event notices would be interspersed among the "dummy" notices. In order to insert into the event list an an event notice scheduled to occur at time T, one could simply compute I = T / 10 as the estimated insertion point of the event notice, and perform a linear search from that point to determine the exact insertion point. If the measure of performance were simply search length, one would doubtless conclude that the performance of the algorithm was excellent, since the probability of of an estimated insertion point being close to the actual insertion point is extremely high. Couched in the terms of computer science, the search length would be described as "of order 1." Unfortunately, consideration of search length alone fails to reveal a very large component in the performance of this algorithm: skipping over dummy notices to find the first "real" event notice each time the simulator clock is to be updated. Assuming that "dummy" notices are not recycled, we know that the scheduling of 10,000 events will entail skipping over 100,000 "dummy" notices. Thus for every order 1 insertion, ten "dummy" notices must be skipped over.

4.6 The Best of All Worlds

The ideal event list algorithm is a fully adaptive, high-performance algorithm which performs well over a wide variety of applications without ever seriously degrading. If simulations for which the algorithm is to be used allow removal of arbitrary event notices, the algorithm should work well for such operations (preferably with performance equal to that of the insertion operation.) Arbitrary removal operations are required to support such language constructs as the GPSS PREEMPT and FUNAVAIL blocks and the Simscript II.5 CANCEL statement.

5. OTHER ALGORITHMS AND LITERATURE

5.1 Historical Notes and References

A number of interesting papers and at least one doctoral dissertation have been published on the subject of event list algorithms. One of the earliest event list papers to appear was Vaucher & Duval (1975). Much of what has been done since can be considered as building upon or reacting to the ideas they first expressed, although Wyman (1976) must be given credit for independently proposing a similar algorithm. Vaucher and Duval compared the performance of the linear search algorithm (descending order), two tree-based algorithms, and their own "indexed list" algorithm. The indexed list algorithm drew upon previously published approaches used in digital logic simulation (Szygenda, et. al. (1971) and Ulrich (1969)). Their tests were conducted by using the HOLD model (see Section 4.5 for a description therof), which was apparently first used by Myhrhaug (publication date unknown).

Improved variants of the indexed list algorithm were developed by Franta & Maly (1977) and Henriksen (1977). Henriksen's algorithm is described in detail in Section 6.

Comfort (1981) and Comfort & Miller (1981) have published some interesting work on the use of dedicated event set processors.

Blackstone et. al. (1981) developed a "two-list" procedure that has been described in at least three papers in addition to the cited reference.

McCormack's doctoral dissertation (1979) is a landmark work comparing a variety of event list algorithms. Results obtained are also summarized in McCormack & Sargent (1981).

5.2 Performance: Claims and Counterclaims

Event list algorithm literature is a vast sea of contradictory claims. Sorting out the claims and counterclaims is made difficult by the fact that methodologies for algorithm evaluation vary widely among authors. Even where CPU times are chosen as the primary basis for comparisons, hardware architecture, choice of language, and object code efficiency are difficult to assess, particularly when hardware and software of different suppliers are used. In the paragraphs which follow, a number of the more significant controversies are presented.

5.2.1 Franta & Maly - Franta & Maly (1977, 1978) have made several claims that others have disputed. First, they have claimed that their algorithm (the "two-level" algorithm) is superior to the indexed list algorithm of Vaucher & Duval (1975) and Wyman (1975). McCormack & Sargent found exactly the opposite to be the case, and offered two possible explanations:

> "One is that previous tests used Pascal [McCormack & Sargent used Fortran.] ..., [and] the second reason could be that an error exists in the implementation [Franta & Maly used]. The effect of this error is to lose some inserted records, thus reducing the size of the future event set. This error does not appear in a later implementation..."

A second claim made by Franta & Maly (1978) has been disputed by several others. Their experimentation and analysis led them to conclude that their two-level algorithm was superior to the heap method. Their 1978 paper was written in response to a suggestion by Gonnet (1978) that the heap algorithm was superior to that of Vaucher & Duval (1975). Once again, the results of McCormack & Sargent contradicted the findings of Franta & Maly.

A third claim made by Franta & Maly (1978) is that "... average complexity for the TL [two-level] structure is [of order 1], experimentally determined." Alternatively stated, they claim that their algorithm is insensitive to the number of event notices in the event list. At complete odds with the results of others, this bold assertion cries out for further investigation.

5.2.2 Blackstone, Hogg, Phillips, and Rodriguez - Blackstone, Hogg, and Phillips (1981a, 1981b); Rodriguez, Blackstone, and Hogg (1982); and Blackstone & Hogg (1982) have developed an event list algorithm called the two-list algorithm. In the various descriptions of their algorithm, the authors have made claims which are erroneous, and they have offered questionable advice to simulation practitioners. The following statements are made in Blackstone, Hogg, and Phillips (1981b):

> "To date, no major general purpose simulation language has adopted an advanced synchronization procedure, probably because of the requirement for external optimization. ... GEMS [a language developed by Phillips and others] is thus the first general purpose simulation language to adapt [sic] an advanced synchronization procedure."

Henriksen (1977) reported results of incorporating an improved algorithm into GPSS/H, and a variant of the GPSS/H algorithm has been used in SLAM (Pritsker 1979) since the advent of SLAM II in 1981. The GPSS/H algorithm (described in Section

6, below) is totally adaptive and has no requirements for external optimization.

The following statement is made in Blackstone, Hogg, and Phillips (1981a, 1981b):

> "The authors believe that the two-list procedure is ideal for general purpose languages and that such languages should adopt the structure."

Neither of these papers present any empirical results to support this recommendation. Furthermore, some potentially excellent algorithms, e.g., the GPSS/H algorithm and the heap algorithm are ignored. The authors do admit their that algorithm is "probably" slower than the Franta & Maly algorithm and the indexed list algorithm. McCormack & Sargent found both of these algorithms inferior in performance to the GPSS/H and heap algorithms. While Phillips, et. al.'s advocacy of their algorithm is based, in part, upon considerations other than performance, their arguments are weakened by their failure to consider alternative approaches which appeared in the literature long prior to their work.

6. THE GPSS/H ALGORITHM

6.1 How the GPSS/H Algorithm Works

In this section, a detailed description of the GPSS/H algorithm (Henriksen (1977)) is given. The presentation herein differs somewhat from the original presentation, reflecting considerations that arose when implementing the algorithm in a high-level language. (The original version was written in assembly code.)

The GPSS/H algorithm operates as follows:

A. The event set is bracketed by dummy event notices on the left (low time) and right (high time) ends, as shown in Figure 6. These dummy notices serve as a virtual predecessor of the first (lowest time) "real" event notice and as a successor to the last (highest time) "real" event notice, respectively. Thus, every "real" event notice has a successor and a predecessor, allowing a symmetry of treatment.

B. A binary tree with the following properties is built "on top" of the event list:

1. Each node in the tree has the format shown in Figure 5. Each node contains a pointer to an event notice, the simulated time at which the event corresponding to the event notice is scheduled to occur, left son and right son pointers, and a pointer to the next lower time node in the tree. For leaf nodes (at the bottom of the tree), the left and right son pointers are zero. For the leftmost (minimum time) node of the tree, the "next lower time" pointer is zero. A three-node tree is shown in Figure 6.

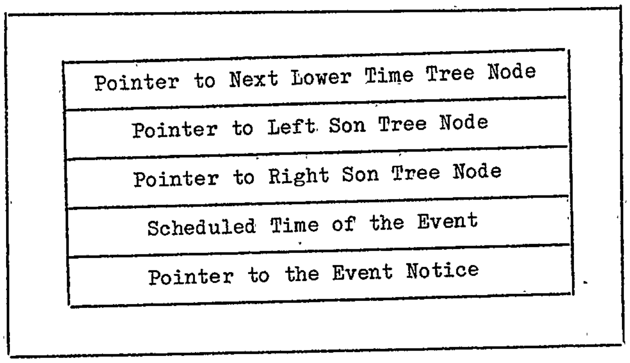
Pointer to Next Lower Time Tree Node
Pointer to Left Son Tree Node
Pointer to Right Son Tree Node
Scheduled Time of the Event
Pointer to the Event Notice

Figure 5: Format of a Binary Tree Node

2. Initially the tree has only a single node; i.e., it is degenerate. That node points to the dummy node at the high end of the event set.

C. When an event notice is to be inserted into the event set, the binary tree is searched to find the node which has the lowest time greater than the scheduled event time of the event notice being inserted.

D. A linear search (in descending time order) is initiated, starting with the event notice to the left of the event notice pointed to by the node selected in step C.

E. The linear search continues until one of two events occurs:

1. If the appropriate insertion point is found by examining four or fewer event notices, the event notice is spliced into the event list at the insertion point. (The insertion point is determined by encountering an event notice with a time less than or equal to the time of the event notice being inserted.)

2. If the insertion point is not found after examining four event notices, an attempt is made to perform a "pull" operation. The "next lower time" pointer of the binary tree node selected in step C is examined. (Note: the "next lower time" pointer is not the left son pointer.) If it is non-zero, the node it points to is modified by changing its event notice pointer to point to the most recently examined event notice. The linear search continues with step D, using the modified tree node in place of the node originally selected in step C. If the "next lower time" pointer is zero, there are no lower time binary tree nodes to be modified. In this case, the depth of the tree is increased by adding a level. The expanded tree is initialized by setting its leftmost leaf to point to the dummy event notice at the right (high time) end of the event list, and all other nodes are set to point to the dummy event notice at the left (low time) end of the event list. Execution continues with step C.

No subsequent attempts are made to reduce the size of the binary tree. It is assumed that once the tree has attained a given size, even if the tree size could be subsequently reduced in accordance with some "reasonable" criteria, it would probably have to be increased again. Thus, reducing tree size could become counterproductive, and, unless extreme care were taken in designing the criteria for reduction of the tree size, oscillation could take place, with the adaptive mechanisms of the algorithm continually reducing and increasing the size of the tree.

The "pulling" of event notice pointers from left to right is an attempt to keep the binary tree balanced. In practice, the tree can occasionally become badly imbalanced, so while the average performance of the algorithm is good, maximum search lengths can be spectacularly long. Test results presented in Section 6.2, below, confirm this tendency.

F. Removal of event notices from the left end of the event list requires no modifications to the binary tree; i.e., it makes no difference that the tree contains pointers to an event notice removed from the left end. Because the simulator clock cannot move backwards, we are guaranteed that the time of any event notice inserted into the event list must be greater than (or at worst, equal to) the current value of the simulator clock, which in turn must be greater than or equal to the time of the most recently removed event notice. Since the binary search of step C searches for a tree node with a time greater than the event time of any event notice to be subsequently inserted, nodes which point to "stale" times (less than or equal to the current value of the simulator clock) can never be selected as the result of the search. Event times are stored directly in the nodes of the binary tree for this very reason; i.e., it is essential to the opetration of step C that time comparisons be made without examining event notices.

G. When an arbitrary event notice is to be removed from the event list, the position of the event notice in the event list is known *a priori*; however, because the binary tree may contain one or more nodes which point to the notice to be removed, a search must be conducted, and any nodes found to point to the event notice must be modified. The search is conducted as follows:

1. The tree is searched (as in step C) to find the node with the lowest time greater than the time of the event notice being removed.

2. If the "next lower time" pointer of the selected node is non-zero, the node to which it points is examined to see whether it points to the event notice being removed. If it does, its event notice pointer is revised to point to the predecessor of the notice being removed, and its time is updated to the time of the predecessor notice.

3. Step 2 is repeated for all tree nodes (if any) which have an event time equal to the event notice being removed. The scan loop is exited upon (a) encountering a zero "next lower time" pointer or (b) encountering the first node with an event time less than that of the event notice being removed. This secondary scan is necessary, because occurrence of "time ties" can result in more than one node with a given time value. If deletion of arbitrary event notices is allowed, more than one tree node can point to a given event notice. In practice, this is occurs very rarely.

The GPSS/H algorithm has the following desirable attributes:

A. It is totally adaptive; i.e., it requires no estimation of algorithm parameters on the part of the user.

B. Its performance is excellent, as reported in McCormack & Sargent (1981).

C. It appears to be robust across a wide range of event scheduling distributions.

D. Its performance for removal of arbitrary event notices is equal to its performance for insertion operations.

E. While the storage of event times in the binary tree nodes is necessary (as described in paragraph F, above), it has a fortuitous consequence for operation in virtual memory systems. In general, event notices tend to occupy wide ranges of virtual memory; hence, direct searching of event notices has the potential for creating a large number of page faults. By comparison, the binary tree occupies a limited amount of virtual memory: typically it is contained in a single page. Hence, searching of the binary tree causes limited page faults.

6.2 Testing the GPSS/H Algorithm

The GPSS program of Figure 2, used to test the two linear search algorithms presented in Sections 2 and 3, was also run using the GPSS/H algorithm. Test results are shown in Figure 7. While the superiority of the GPSS/H algorithm is readily apparent, several observations are in order.

The CPU time per statement-execution increases only slightly as the size of the event list increases from 10 to 100, indicating relative insensitivity to the size of the list.

Although average search lengths are quite short, maximum search lengths are quite long. Indeed, for the 100-user case, the maximum search length exceeds the number of event notices in the event list. This is explained by the fact that the adaptive process built into the algorithm (doubling the size of the binary tree when necessary) can cause major disruptions. Fortunately such disruptions occur rarely.

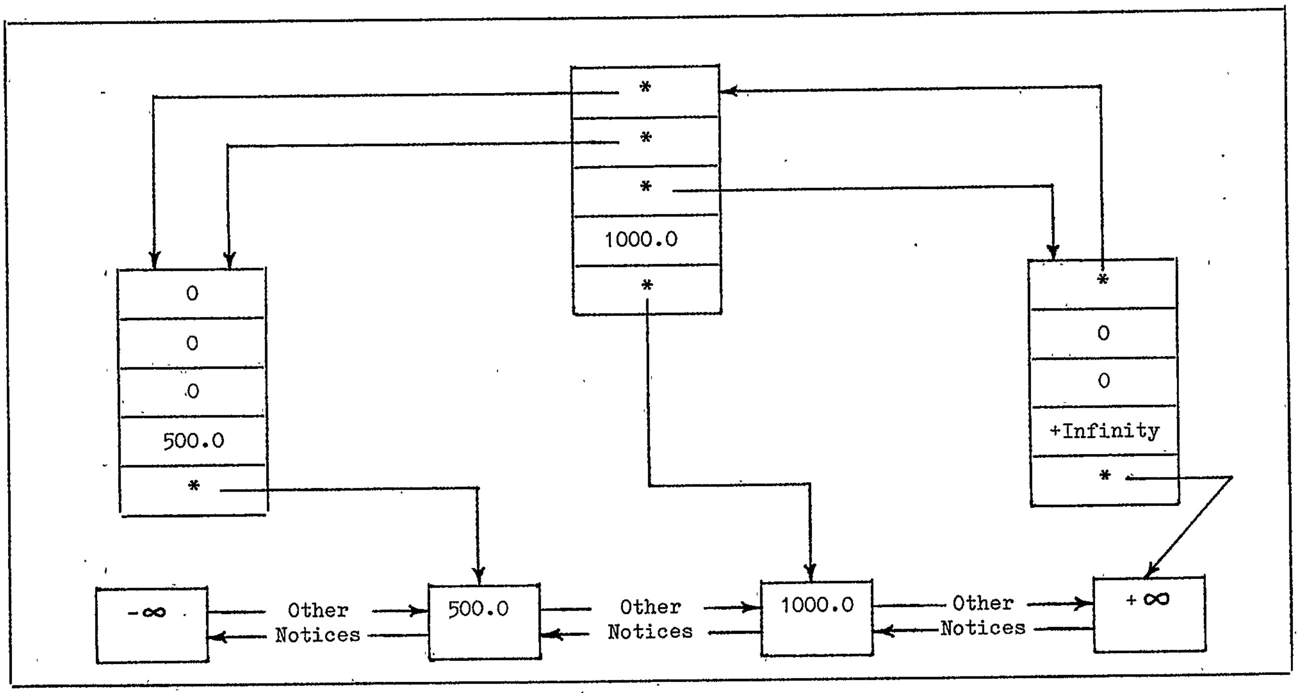

Figure 6: Example Binary Tree

Number of Users	Number of Events Scheduled	Insertion Search Lengths Average	Insertion Search Lengths Maximum	Maximum Tree Depth	Execution CPU Time / Number of Statements Executed
10	20045	4.42	12	2	187.85
50	98878	6.47	49	4	187.71
100	196875	7.55	107	5	192.86

Figure 7: Results Using The GPSS/H Algorithm

7. CONCLUSIONS

The selection of an appropriate event list algorithm can be vitally important to the execution performace of a simulation. The quality of the literature on event list algorithms has a large variance, and it is loaded with claims and counterclaims, correct and incorrect, supported and unsupported. While considerable disagreement exists among those who have analyzed evenyt list algorithms, a number of attractive algorithms are available. Works published as recently as 1982 have erroneously claimed that general purpose languages fail to incorporate advanced event list algorithms; however, several implementations of general-purpose simulation languages now include advanced event list algorithms, and at least one such implementation has been in use since 1977.

ACKNOWLEDGEMENT

Thanks go to Bob Crain for his thoughtful suggestions which are reflected herein.

BIBLIOGRAPHY

Blackstone, J. H. and Hogg, G. L. (1982), Improved File Handling for Discrete-Event Models with Event Cancellation, Simulation, Volume 39, Number 6, (December, 1982), pp. 201-204.

Blackstone, J. H., Hogg, G. L., and Phillips, D. T. (1981a), A Two-List Method for Synchronization of Event Driven Simulation, Proceedings of the Fourteenth Annual Simulation Symposium, Tampa, FL, (March, 1981), pp. 95-101.

Blackstone, J. H., Hogg, G. L., and Phillips, D. T. (1981b), A Two-List Synchronization Procedure for Discrete Event Simulation, Communications of the ACM, Volume 24, Number 12, (December, 1981), pp. 825-829.

Comfort, J. C. (1981), The Simulation of a Microprocessor-Based Event Set Processor, Proceedings of the Fourteenth Annual Simulation Symposium, Tampa, FL, (March, 1981), pp. 17-33.

Comfort, J. C. and Miller, A. (1981), The Simulation of a Pipelined Event Set Processor, Proceedings of the Winter Simulation Conference, Atlanta, GA (December, 1981), pp. 591-597.

Franta, W. R. (1977), The Process View of Simulation, North-Holland, New York, NY.

Franta, W. R. and Maly, K. (1977), An Efficient Data Structure for the Simulation Event Set, Communications of the ACM, Volume 20, Number 8, (August, 1977), pp. 596-602.

Franta, W. R. and Maly, K. (1978), A Comparison of Heaps and the TL Structure for the Simulation Event Set, Communications of the ACM, Volume 21, Number 10, (October, 1978), pp. 596-602.

Gonnet, G. H. (1976), Heaps Applied to Event-Driven Mechanisms, Communications of the ACM, Volume 19, Number 7, (July, 1976), pp. 417-418.

Henriksen, J. O. (1977), An Improved Events List Algorithm, Proceedings of the Winter Simulation Conference, Gaithersburg, MD, December, 1977, pp. 547-557.

Henriksen, J. O. and Crain, R. C. (1982), GPSS/H User's Manual (Second Edition), Wolverine Software Corporation, 7630 Little River Turnpike, Annandale, VA 22003.

McCormack, W. M. (1979), Analysis of Future Event Set Algorithms for Discrete Event Simulation, Ph.D.#Dissertation, Syracuse University, Syracuse, NY, 1979.

McCormack, W. M. and Sargent, R. G. (1981), Analysis of Future Event Set Algorithms for Discrete Event Simulation, Communications of the ACM, Volume 24, Number 12, (December, 1981), pp. 801-812.

Pritsker, A. (1979) and Pegden, C. D. (1979), Introduction to Simulation and SLAM, Halstead Press (Division of John Wiley and Sons, Inc.), New York and Systems Publishing Corpoation, P. O. Box 2161, West Lafayette, IN

Rodriguez, L.C., Hogg, G. L., and Blackstone, J. H. (1982), An Empirical Comparison of Advanced Event File Synchronization Structures, Proceedings of the Winter Simulation Conference, San Diego, CA (December, 1982), pp. 189-194.

Syzgenda, S., Hemming, C. W., and Hemphill, J. M. (1971), Time Flow Mechanisms for Use in Digital Logic Simulation, Proceedings of the Winter Simulation Conference, New York, NY, 1977, pp. 488-495.

Ulrich, E. G. (1969), Exclusive Simulation Activity in Digital Networks, Communications of the ACM, Volume 12, Number 2, (February, 1969), pp. 102-110.

Vaucher, J. G. and Duval, P. (1975), A Comparison of Simulation Event List Algorithms, Communications of the ACM, Volume 18, Number 4, (April, 1975), pp. 223-230.

Wyman, F. P. (1975), Improved Event Scanning Mechanisms for Discrete Event Simulation, Communications of the ACM, Volume 18, Number 6, (June, 1975), pp. 350-353.