

The Threshold of Event Simultaneity

Frederick Wieland

Center for Advanced Aviation Systems Development, The MITRE Corporation, McLean, Virginia; E-mail: fwieland@mitre.org

In this paper we revisit the notion of event simultaneity in the context of parallel and distributed simulation. Although the simulation community has recognized this problem for years, it has focused mainly on the mechanics of breaking event-time ties, and has neither measured its extent nor considered its implications. Extant simulators (both serial and parallel) prohibit simultaneity either by user-specified event priorities or by an arbitrary (but well-documented) tie-breaking mechanism. We shall argue that, in many cases, these strategies lead to an invalid simulation. In doing so, we shall introduce the threshold of event simultaneity and use it to understand the semantics of simultaneity. We shall also present empirical results measuring the magnitude of deviation from logical correctness that occurs when a parallel simulation explicitly allows simultaneous events.

Keywords: Simultaneity, event simultaneity, parallel and distributed simulation

1. Introduction

Simultaneous events—two or more events that are scheduled to occur at exactly the same simulation time—cause a great deal of difficulty in both serial and parallel simulators. The ordering of simultaneous events can dramatically affect the result of the computation, and can lead to logical correctness errors for parallel simulators. Even more importantly, the concept of a *correct answer* is difficult to define in the presence of simultaneous events. In this paper we review the concepts behind simultaneous events, show the mechanisms by which serial and parallel simulators avoid them, define what it means to get a correct answer when events are simultaneous, and draw implications for parallel and distributed simulation. While event simultaneity is not unique to parallel simulation, we shall argue that it is one cause of deviation from logical correctness, and that current methods of resolving simultaneous events need to be re-addressed.

A discrete-event simulation is composed of timestamped events that are scheduled by one entity for another. An “entity” is usually the software realization of some real-world object, such as airports in an aviation simulation or combat units in a wargame. In the context of parallel simulation, the more general term “logical process” is used instead of “entity.” Both terms can be used in this analysis, as they refer to the same computational unit at our level of abstraction, but we will use the term “logical process,” because our focus is mainly on the problems inherent in parallel simulators when dealing with the ordering of simultaneous events. A logical process is composed of a future event list and a current state. Its state is changed whenever an event is executed; events are executed in timestamp order. Simulation time changes *between* the execution of events, rather than *during* event execution.

In some sense the discrete-event model is a dual of the process-oriented model. In the latter, simulation time can change *during* the execution of a process. Process-oriented simulators provide mechanisms for modelers to specify the passing of simulation time; most commonly, it is some form of “wait” statement. Examples include waiting for a certain period of time to elapse before continuing, or waiting for a certain trigger event to happen either at that entity or at a remote entity. Simulation practitioners have successfully employed both the discrete-event and the process-oriented models, and problems regarding event simultaneity arise in both cases. For the remainder of this paper we shall concentrate on the discrete-event model, because the issues are more transparent using that model.

We are now in a position to define the term “simultaneous events” as it applies to this analysis. Events are characterized by their future execution timestamp, by their type, and by the data required to properly execute them. In this paper, simultaneous events will be very narrowly defined to mean *two or more events of identical type with identical timestamps, to be executed by the same logical process*. The only potential difference between otherwise simultaneous events is their data and the logical process that scheduled it (although both of these are frequently identical as well). By defining simultaneous events in this manner, event tie-breaking mechanisms that rely on ordering by event type are inapplicable.

Although this definition of simultaneous events seems to eliminate many of the more common instances of simultaneity, events conforming to this narrow definition do indeed occur in real-world simulations, as will be shown later in this paper. In such cases, it is difficult, if not impossible, for the simulation creator (hereafter, the *simulationist*) to specify tie-breaking rules, because frequently the simulationist is unaware of the cause or even the existence of this type of event time-tie. Therefore, tie-breaking schemes which rely upon model-defined tie-breaking mechanisms also become inapplicable.

Received: January 1998; Revised: March 1999, Accepted: March 1999

TRANSACTIONS of The Society for Computer Simulation International
ISSN 0740-6797/99
Copyright © 1999 The Society for Computer Simulation International
Volume 16, Number 1, pp. 23-31

2. Causes of Event Simultaneity

It is mistakenly believed that simultaneous events can be avoided by using continuous probability distributions for the generation of inter-event times. In theory, the probability that two continuously distributed event times are equal is zero. But because computers necessarily discretize real numbers, it is possible to generate two “very close” random variates whose discrete representations are equal at the level of machine precision. This concept is particularly true when the generated time is the result of a calculation involving a random variate, in which case the cumulative roundoff error can equate two otherwise different times.¹ When the event time distribution is discrete rather than continuous, then the probability of two equal times is significantly greater.

Another example of machine precision causing simultaneous events occurs when the timestamp field associated with events runs out of space to correctly resolve otherwise non-simultaneous events. This situation can occur when the simulation time range (the difference between the smallest and largest timestamp in the underlying model) is so large that small differences in event times are rounded to the same floating-point representation. These types of events are simultaneous only because of limitations of machine precision, but they pose all of the difficulties associated with events purposely scheduled at the same time.

While these causes of simultaneity are important, simultaneous events are much more common when they are scheduled by a process external to the simulation (such as through a data file). For example, it is common for different airlines to schedule the departure of flights at the same time, so airport simulations must deal with multiple simultaneous departure events. Similarly, periodically scheduled events—such as the generation of status reports in battle simulations, or the updates of positions in networked virtual-reality systems—can generate many coincident event times. Events related to the management of a simulation, such as those representing initialization activities, creation or destruction of entities, or starting and stopping certain processes, are often scheduled at integral times in such a way that simultaneity is also possible.

It is difficult for the simulationist to identify simultaneous events because of the inherent complexity of simulation. Simulations are decomposed into multiple logical processes (or LPs), and the activities of many such LPs are interleaved in simulation time. Simultaneous events often arise from the interleaved execution of different entities simulating different processes during coincident simulation time intervals.

Simultaneous events cause a problem for simulators (both sequential and parallel) because it is not known in what order the events ought to be run. If there are two simultaneous events, A and B, should A be processed before B or vice-versa? As we shall demonstrate, the order in which these events are run can cause a substantial difference in the results.

3. A Simultaneous Event Problem

Consider a simple queueing model where the major observable shall be total wait time in the system, summed over all customers

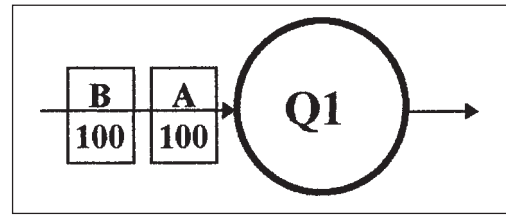


Figure 1. A queue with simultaneous arrivals

in the queue. A simple single-server queue model is shown in Figure 1, where there are two identical and indistinguishable customers, labeled A and B, who wish to enter the same queue (Q1) at the same time (time 100). We shall assume that the service time in all queues in this model is a constant two minutes, with no variance.

The notation {A, B} shall mean that event A is processed before event B. In this simple example, the ordering {A, B} produces no delay for A, but a two-minute delay for B as it waits for the service of A to be completed. The ordering {B, A} produces the opposite result. In either case, the total system waiting time is the same—two minutes—but the assignment of that waiting time is different. The simultaneity problem arises when these two customers are routed to other queues in a network, thereby propagating their delays to different queues. Depending upon the ordering of the events at Q1, the propagated delay will create different situations at the other queues, causing a different pattern of future events and thereby affecting system performance measures.

To illustrate this effect in its simplest form, suppose we introduce two additional single-server queues, Q2 and Q3, such that A, after being serviced by Q1, is routed to Q2, and B is routed to Q3. There are also two additional customers, X and Y. Customer X enters Q2 at time 101, and customer Y enters Q3 at 102; these customers go directly to Q2 and Q3 without first passing through Q1. Throughout this discussion, the travel time between queues will be zero. This example is illustrated in Figure 2.

If the ordering at Q1 is {A, B}, then A exits Q1 at time 102 and (after a zero travel time) enters Q2 at time 102. This situation creates two nonsimultaneous events at Q2, with the ordering {X, A}. X is already one minute into its two-minute service time when A arrives at Q2, so A must wait one minute for service. B, on the other hand, experiences a two-minute wait for Q1, entering at 102 and leaving at 104, and experiences no wait at Q3, because at time 104, Q3 has already completed the service of Y. Note both X and Y experience no wait. Thus the ordering {A, B} at Q1 has produced a total system waiting time of three minutes.

Next consider the ordering {B, A} at Q1. Following the same logic, B exits Q1 at 102, causing another simultaneous event at Q3, where both Y and B enter Q3 at 102. Regardless of the ordering at Q3—either {B, Y} or {Y, B}—one will emerge with a two-minute wait, the other with no wait. Meanwhile, A must wait two minutes at Q1, and has no wait at Q2. Thus the ordering {B, A} at Q1 has produced a total system waiting time of four minutes—two for A at Q1 and two for either Y or B at Q3.

What has happened here is pivotal for understanding the underlying problem: *merely interchanging the order in which*

¹ For the skeptical reader, it should be noted that the author has encountered this effect in developing a parallel simulation.

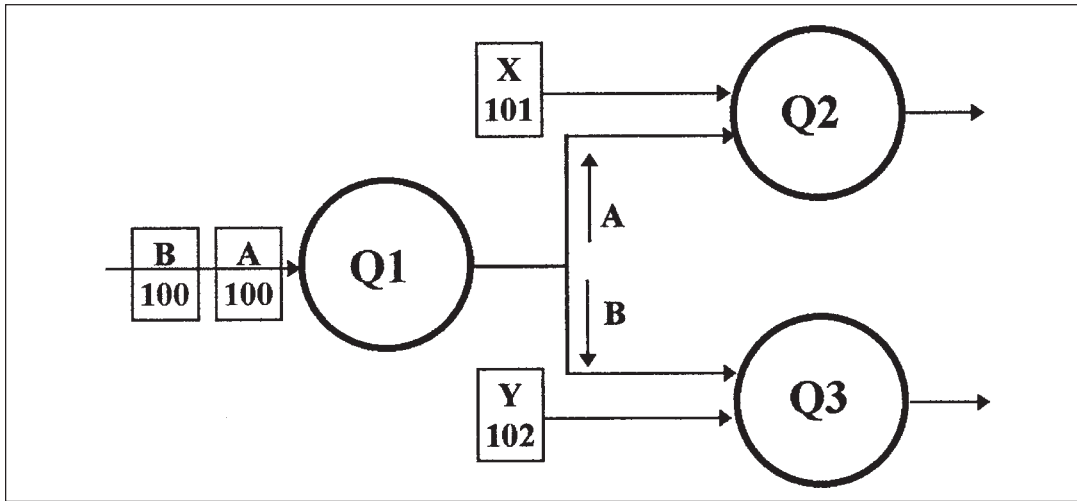


Figure 2. Network of queues with simultaneous arrivals

the customers are processed in Q1 has increased the system waiting time by 33%, from three to four minutes. This simple result has implications for the validity of network-based simulations in the presence of simultaneous events, as well as parallel simulations when there lacks a mechanism for distinguishing between events.

An important result arising from this problem is that the discrepancy in total system waiting time can become arbitrarily large as more simultaneous events are added to the system. For example, consider a third simultaneous event at Q1 (call it event C) which is routed to a fourth queue, Q4. The events are now A, B, and C at Q1 at time 100, X at Q2 at 101, Y at Q3 at 102, and Z at Q4 at 103. This situation is illustrated in Figure 3. There are now $3! = 6$ possible orderings of the three events at Q1, and they generate the total system waiting times listed in Table 1. These times span a factor of two, from three minutes to six minutes, with an average of $4\frac{1}{3}$.

Table 1. Performance metrics for different simultaneous event resolutions for the situation in Figure 3

Ordering	Total System Waiting Time (Mins.)
{A,B,C}	4
{A,C,B}	5
{B,A,C}	3
{B,C,A}	6
{C,B,A}	4
{C,A,B}	4
Average	$4\frac{1}{3}$

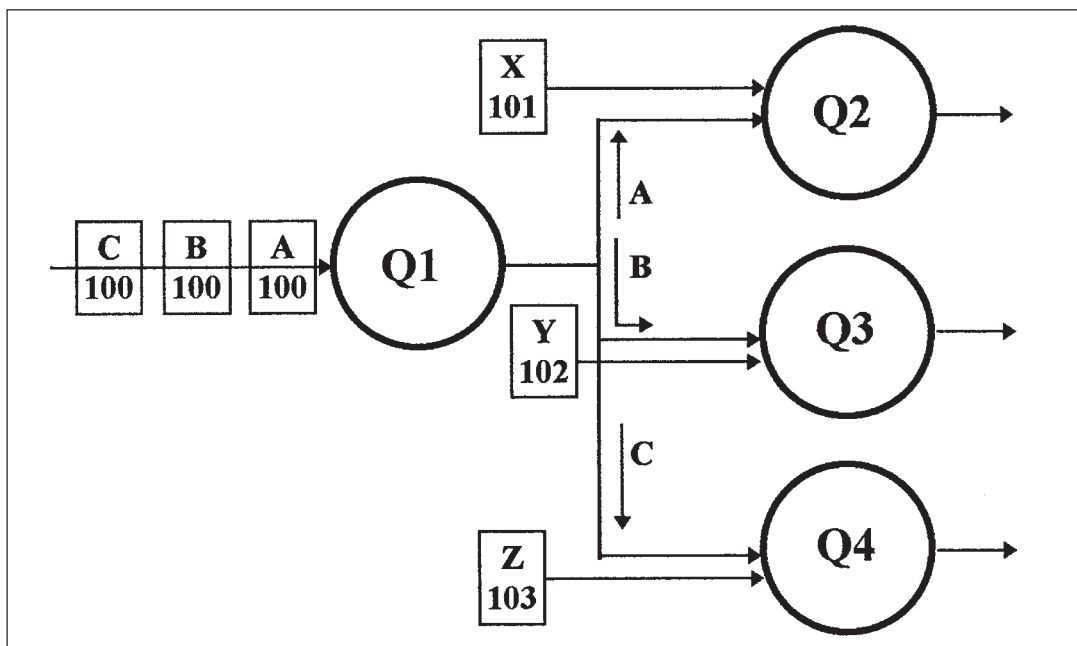


Figure 3. A more complicated network of queues with simultaneous arrivals

It can be shown that by extending this pattern, the deviation between the longest and shortest total system waiting times among the $N!$ orderings of the N simultaneous events *can become arbitrarily large*. There is no limit to the amount of “damage” that can be done to the results as the number of simultaneous events grows.

3.1 Implications for Parallel Discrete-Event Simulation

The computational model for parallel discrete-event simulation systems typically involves a set of logical processes which exchange timestamped event messages representing the transactions of the simulation (for a detailed discussion of this model, see [1]). The logical processes can be represented as queues, and the timestamped event messages are the customers that are processed by the queues. In the situation where there are two or more simultaneous, indistinguishable event messages waiting for a logical process, the order of processing becomes nondeterministic.

The nondeterminism is peculiar to the parallel execution of the simulation. Indistinguishable simultaneous events are typically processed in the order that they appear in the future event list that is attached to the logical process. In sequential execution, all computations are sequenced identically each time the system is run; therefore the order in which simultaneous events appear in each future event list is identical, leading to repeatable, dependable results. In parallel execution, the computations are carried out on different processors simultaneously, leading to an interwoven stream of instructions and events that behaves as if the simultaneous events are randomly ordered in the future event list. Each time the system is run, this random function can potentially produce a different ordering of simultaneous events. Following the example above, subsequent executions of otherwise identical simulations (input data and random number streams are unchanged) can produce simulation results which are not identical. Such behavior challenges the sensibilities of simulation analysts, who would otherwise expect the results from subsequent identical runs to agree.

Event simultaneity can cause the execution to produce different results because there exist two problems which can create nondeterminism. The first problem arises if the two events are not *commutative*, meaning that the state produced by the LP is different depending upon the order of execution of the two identical, indistinguishable events. In this case the LP's state can differ every time the system is executed, which is one cause of nondeterminism. The second problem arises if the LP scheduled subsequent events such that the timing of the subsequent events is dependent upon the order of processing of the incoming events. In this case, the future event trace of the simulation will differ depending upon which order the simultaneous events are processed, leading to a second cause of nondeterminism. If both causes are present, the simulation has little hope of achieving deterministic results with otherwise identical input.

An interesting question arises with this problem. If the results are dependent upon the execution order of simultaneous events, then which of the various event orderings produces results that are correct? Equivalently, is it even possible to define one of the results as being the correct one? More generally,

how does this whole problem affect simulation validity? We will address these questions after first considering how extant sequential and parallel simulators handle this problem.

3.2 Resolving Simultaneous Events in Sequential and Parallel Simulators

This problem has already been noted by both the sequential and parallel simulation community, and various work-arounds have been proposed. In this section we shall review how first sequential, and then parallel, simulators address this problem, and then we shall expose weaknesses in these current methods with regard to simulation validity.

Beginning with the sequential community, Schriber's book on GPSS contains an excellent discussion about simultaneous events, beginning on page 70 and running, on and off, throughout the examples in the book [2]. GPSS solves the problem by executing simultaneous events in an arbitrary order, unless the simulationist has explicitly specified the priority of events. Schriber emphasizes throughout the book the necessity of specifying such tie-breaking priorities, and illustrates the technique with numerous examples.

SLAM II uses two different tie-breaking techniques depending upon whether the simultaneous events are “current events” or “future events” [3]. Current events are defined as those with zero delay (that is, events where the simulation time at which the event is scheduled and when it is scheduled for are identical). Current event ties are executed in a LIFO (last-in, first-out) basis, with no provision for the simulationist to override the default. Future event ties are executed in a FIFO (first-in, first-out) basis, with a provision for the user to override the default with a LIFO ordering or one based upon a user-specified attribute of the event. Pritsker notes that the two tie-breaking mechanisms exist for reasons of event-list manipulation efficiency, rather than for some deeper theoretical reason.

SIMAN resolves event ties differently depending upon the type of activity that created the tie [4]. In the creation and queuing of entities, ties are broken based upon the order the events occur in the event list. In the case of cloning entities during branch instructions, the simultaneous events created by the clones are executed in the order they are specified in the branch instruction. Finally, in the case of dequeuing operations or explicit non-branch cloning operations, simultaneous events are run in the *reverse* order from which they are created. There does not appear to be any procedure for overriding these defaults.

SIMSCRIPT II.5 also handles ties in a context-dependent fashion. Multiple instances of an identical process all scheduled for the same time are executed in a FIFO fashion. For ties which occur as a result of scheduling from two different processes, priority is given to the event scheduled by the process which appears earlier in the SIMSCRIPT preamble. Events scheduled with zero-delay are executed before any other events occurring at the same time, and when there are several tied zero-delay events they use the same rules as above. Finally, the simulationist can override any of these defaults by explicitly declaring attributes of the processes by which ties will be broken.

The problem has been noted in the parallel simulation community as well. Cota and Sargent describe an algorithm whereby

priorities can be assigned to events in a sequential simulator such that ties are resolved in a manner consistent with a distributed simulation [5]. They claim that the latter breaks ties by simulating events in *dependency order*, which means that whenever the simulation of one event affects another, the first event is executed first. Their automatic tie-breaking algorithm makes use of both dependency order as well as *influence* order. Apparently they do not consider simultaneous events which affect the result of the computation but are not dependent in a graph-theory sense, such as customers A and B in the example we cited above. The simulationist typically schedules such events, so the dependent event is *external* to the simulation itself.

Horst Mehl presents an algorithm for the global ordering of all events such that optimistically synchronized parallel simulations and their sequential counterparts will be consistent [6]. The algorithm is robust in the face of anti-message generation, simulationist-initiated message cancellation, and lazy cancellation. Mehl also reviews other parallel tie-breaking schemes, such as Agre and Tinker’s technique, the method used in JADE’s Time Warp, and that used in the Time Warp Operating System (TWOS). TWOS broke ties by ordering all events on the following key fields: receiver, receive time, sender, send time, and the message text. The message text was used when all other fields were identical, and a lexicographic ordering of the text was employed.

Another technique for resolving simultaneous events was introduced by A. Chow in the Parallel DEVS environment [7]. Chow distinguishes between internal events (those scheduled by entities in the model for each other) and external events (those scheduled by processes outside of the simulation). When two internal events are simultaneous, the modeler is required to serialize them using a “select” statement. Ambiguity arises when internal and external events are simultaneous: should the external events be processed before or after the internal events trigger the model’s state change? Chow introduces the notion of a confluent transition function, which allows the modeler to specify the processing of the external events as if they were one logical event, and furthermore allows the parallelism between simultaneously scheduled internal and external events to be exploited.

Each of these tie-breaking mechanisms relies upon either some arbitrary (but well-documented) rule imposed by the simulation engine, or by relying upon the simulationist to specify the model or assign event priorities in such a way that simultaneous events are eliminated. These techniques all either prevent or deterministically order simultaneous events, but (as we have shown) it is possible to have a valid simulation in which time ties do occur, and yet arbitrarily choosing one of the many possible tie-breaking orderings as the “correct” one seems unreasonable, and quite possibly incorrect. How can the simulationist be sure that the particular tie-breaking order in which events are executed leads to a valid result? What is a valid result anyway? How can simulators ensure that data representing “outliers” are not presented to the analyst as the correct answer?

4. Actual Examples

Before addressing these questions, one might be tempted to ask whether this effect happens in actual practice and, if so, what is

its magnitude. To answer this question, we have used an actual parallel simulation of air traffic control as reported by [8]. This simulation is called the Detailed Policy Assessment Tool (DPAT), and has been used extensively for analysis of operations of the National Airspace System (NAS), both domestically and worldwide. The model views the NAS as a collection of queues, at both the airport level and in en-route airspace. The model is well suited for computing delay and throughput numbers, and for propagating delays from one airport to another within a geographical region.

During model development, it was noted that nonrepeatable results often arise in subsequent executions of otherwise identical scenarios. This problem was traced to the handling of simultaneous events in the model. Simultaneous events arise, in this case, when departures from an airport are scheduled by a process external to the simulation itself, in this case a data file of intended commercial aviation departures. The data file would often contain event ties of the type considered in this paper: two or more departure events, from the same airport LP, at the same simulation time. The only difference among the events is the data associated with the flight (the airline, flight number, gate assignment, and so on).

Table 2 indicates how often this situation arose in an actual run. The data used was for a day in May 1993 with fairly heavy traffic. DPAT was configured to run 500 airports in the continental United States, and to handle all other traffic in one of 20 other super source/sink airports. DPAT ran the simulation for 24 simulated hours (which only consumes about one minute of wall-clock time on a 4-processor Sun SPARC machine with 120 MHz processors). Table 2 reveals that there are 1,444 distinct instances where there are exactly two departure events scheduled for the same airport at the same simulation time, 245 instances where there are three tied departure events, and so on. There are approximately 400,000 total events in the model, so

Table 2. Number of instances of simultaneous events in one of the DPAT scenarios

Number Departure Events at the Same Simulation Time	Number of Instances When this Occurred
2	1444
3	245
4	87
5	46
6	24
7	10
8	9
9	5
10	1
11	4
12	3
13	1

Table 3. Simultaneous event disparity in a 4-processor parallel simulation

Run #	Throughput (Number)	Total Delay (Mins.)	Delay/Operation (Mins.)
1	1215	35,637	29.3
2	1222	41,970	34.3
3	1226	40,357	32.9
4	1223	43,053	35.2
5	1226	44,668	36.4
6	1228	43,761	35.6
7	1210	29,318	24.2
8	1224	43,832	35.8
9	1217	39,103	32.1
10	1223	41,692	34.1
Average	1221.4	40,339	33.0
Std. Dev.	5.7	4,709	3.7

the number of simultaneous events is a small fraction (about 0.4%) of the total number of events executed.

Despite the relatively small (0.4%) number of simultaneous events, their effect on the model results are dramatic. Table 3 shows the throughput and delay numbers at one of the large airports in the model when DPAT was executed 10 times with identical input (same data files, same random number seeds). These numbers (delay and throughput) are what aviation analysts expect from a simulation such as DPAT, and yet otherwise identical runs show a coefficient of variation (COV) in throughput of 0.4% and in delay of 11.7%. An analyst would expect the COV to be zero in both instances.

DPAT is nonrepeatable because it exhibits both of the problems mentioned earlier. Simultaneous departure events in DPAT are not commutative—the LP’s final state differs depending upon which order the aircraft depart. In addition, the scheduling of future events at the airport LP depends upon the timing of the departures—which in turn is dependent upon which of the simultaneous events is processed first.

The system is nondeterministic to such a degree that the first reaction is that there must be a bug in the software at some level. This explanation is incorrect. The source of the nondeterminism is merely the different results that are obtained when the simultaneous events are processed in a different order on subsequent executions. Confirmation of this fact occurs when the simultaneous events are eliminated, by adding a small amount of “noise” to the departure times, such that the number of simultaneous departures at each airport becomes zero. When that feature is enabled in DPAT, all runs suddenly become identical.

If the runs become identical when we eliminate simultaneous events, then it appears that we have solved the problem, but that is not the case. The problem is not solved merely by mechanically ordering the events in some deterministic manner, because we are left with the question, what ordering is correct? In the case of DPAT, assume that there are departure ties at airport X for flights from airline A and airline B. If we arbi-

trarily choose airline A to depart first, we will get one answer from the model. If we then arbitrarily choose airline B to depart first, then we will get another answer from the model. Which answer is correct?

Some argue that at this point a more detailed model is necessary. For example, if DPAT simulated all the events leading up to a departure—the pre-flight checklist, passenger loading, baggage handling, ticket operations, fueling, and so on—then it could accurately distinguish departure times, thereby eliminating the simultaneity. This simplistic solution is also incorrect, for several reasons. First, what happens if there are simultaneous events at this finer level of detail? Secondly, suppose all events are nonsimultaneous at the finer detail level. Where does an aviation analyst get data about the length of the pre-flight checklist activity? About the number of passengers aboard each aircraft? About the amount of fuel loaded on each aircraft and the time it takes to load such fuel? Even if these numbers could be accurately determined (doubtful, in most cases), this level of detail requires an analyst to solve a problem which is an order of magnitude more complicated merely to resolve event ties which occur 0.4% of the time (in this example). In an era of tight deadlines, it is unrealistic to expect an analyst to undertake such an activity.

Here we have a real-world example, DPAT, which is used by analysts to study the operations of air traffic control situations worldwide, and yet we are still plagued with the question that we have been asking throughout this paper: what should the correct answer be in the presence of simultaneous events? How can we determine which of the various possible tie-breaking rules generates the correct result? In other words, what is the *meaning* of simultaneous events?

5. The Semantics of Event Simultaneity

The central thesis of this paper is that, lacking any clues from the simulationist as to how to resolve simultaneous events, they become yet another source of uncertainty (randomness) in the model

and should be treated identically to all other sources of uncertainty—using statistical techniques to characterize its output distribution. We shall build this argument from fundamental principles regarding the semantics of event simultaneity.

Returning to the simple three-queue system diagrammed in Figure 2, note the events A and B. They both occur at time 100 at Q1, and the ordering is either {A, B} or {B, A}. Now consider a slightly different simulation, in which event A occurs at time $100 - \delta$, and event B occurs at time $100 + \delta$; the two events are no longer simultaneous. Now suppose $\delta = 0.01$. The ordering at Q1 is now deterministically {A, B}, and A enters Q1 at 99.99 and B enters Q1 at 100.01. This timing causes A to wait 1.01 minutes for Q2 (because it arrives at Q2 0.01 minutes earlier), while B waits 1.98 minutes for Q1 (as opposed to 2.0 minutes) and 0.01 minutes for Q3 (as opposed to 0 minutes). The effect of A being scheduled earlier causes A to wait slightly longer, and B to wait slightly less, such that the total wait time (three minutes) is identical to the simultaneous events case.

However, if B is scheduled at time $100 - \delta$, and A at $100 + \delta$ ($\delta = 0.01$), then A waits 1.98 minutes at Q1 (as opposed to 2) and has no wait at Q2. On the other hand, B has no wait at Q1, but waits 1.99 minutes at Q3. The total system waiting time becomes 3.97 minutes, not quite the four minutes observed with the {B,A} ordering in the simultaneous events case.

Although neither modified simulation contains any event time ties, neither do their results match: one claims a system waiting time of 3 minutes, the other 3.97 minutes. These two simulations can be characterized as specific instances of a third, more general simulation, in which the event times of A and B are chosen from a uniform random distribution on the interval $[100 - \delta, 100 + \delta]$.² Now suppose this more general simulation happens to be the one of interest: how would we determine the “correct” answer? A simulationist would determine the answer to this (more general) problem by running it multiple times, varying the random number seed each time, and generating statistics (mean, variance) on the results. In this example, where $\delta = 0.01$, the procedure would yield an average of 3.485 minutes $((3 + 3.97) / 2)$. In the limit as $\delta \rightarrow 0$, the procedure will converge to 3.5 minutes $((3+4)/2)$. But at $\delta = 0$ we have an exact representation of the original simultaneous-events problem. This reasoning leads to the following assertion:

Event Simultaneity Assertion. *If there are N simultaneous events centered at time T , then an estimate of the correct answer to the simulation is equivalent to the limit as $\delta \rightarrow 0$ of the average answer computed by drawing K samples where, for each sample, the N time ties are broken by independently choosing their times from a uniform random distribution on the interval $[T - \delta, T + \delta]$.*

The parameter K , of course, should be a statistically significant sampling of the $N!$ tie-breaking combinations. A corollary to this assertion is that the exact answer can be computed by averaging over *all* possible event orderings. The assertion above is merely a statistical procedure for estimating this answer, the pro-

cedure having been derived by considering the more general simulation of two events “close” but not identical in their event times.

5.1 The Threshold of Event Simultaneity

We shall call the parameter δ the *threshold of event simultaneity*. The observant reader might note that our assertion reduces to drawing a sample of the $N!$ combinations where in each run of the sample we break the ties in a random manner, and then we average all of these runs together; why, then, do we need the parameter δ ? One reason is that the parameter δ is useful in answering the *opposite* question: if breaking ties is equivalent to perturbing the time ties by a small amount δ , then does that mean that events scheduled within δ time units are, in fact, simultaneous events?

An intriguing question indeed. Suppose that in a particular queueing simulation, customer A arrives at Q1 at time 99.9995, and customer B arrives at time 99.9996. Are we to believe that the underlying simulation model is precise enough to differentiate between two events that differ by only 0.0001 units of time? Suppose the answer to this question is “no.” If so, we have two events that might as well be simultaneous, leading to the event threshold assertion:

Event Threshold Assertion. *If two or more events are all scheduled within the simulationist-defined parameter δ , then those events are simultaneous and should be treated as such. The parameter δ is called the threshold of event simultaneity.*

5.2 Implications for Analytic Simulation

In analytic simulation, whether serial or parallel, the analyst is interested in drawing the same conclusions from the simulated data as s/he would draw if analyzing actual data. If the physical system being simulated contains simultaneous events, and these events are sequenced in an arbitrary manner by the simulator, then there is a possibility that the analyst would be presented a statistical “outlier,” i.e., a representation of the system that would occur only rarely. A better procedure is for the simulator to present an average answer to the analyst, averaged over all possible tie-breaking combinations. Such a procedure would produce an answer closer to the actual behavior of the physical system.

In such simulations, the threshold parameter δ should be chosen to be equal to the precision of the model. The model precision dictates how close in time two events can be before they become simultaneous. The precision depends upon the precision of the input data, variance of the random number distributions, machine floating-point precision, and so on. For example, if the model is precise enough to differentiate between events that occur only 0.01 time units apart, then two events scheduled within 0.001 time units should be considered simultaneous. The simulator needs to average the answer obtained from the two possible orderings of these two events, treating them as if they had occurred at exactly the same time.

5.3 Implications for Distributed Real-Time Simulation

In unconstrained real-time distributed simulations, events are processed in the real-time order they are received. Simulators on different processors might process the same event stream in

² There is nothing special about a uniform distribution; it is employed here merely for simplicity of thought.

a different order. If events are separated by *less than* a real-time interval δ , then they need to be considered simultaneous. Unlike the case for analytical simulations, real-time simulations are typically used for training or amusement (entertainment), in which case these simultaneous events can be executed in *any* arbitrary order, because the end result is not rigorously scrutinized.

Events that are separated by *more than* δ units of real time need to be properly sequenced with respect to each other. These events can no longer be considered simultaneous. The parameter δ can therefore be considered the degree that logical correctness is enforced on the distributed simulation. If $\delta = 0$, we have strict logical correctness and the full theory of synchronization in parallel simulation is needed. As δ becomes larger and larger, the need for resequencing out-of-order events lessens, because there are more simultaneous events (which can be processed in any order, rather than averaged) and fewer strictly sequenced events. In the limit that $\delta \rightarrow \infty$, then the entire simulation is composed of one giant set of simultaneous events, which can be processed in any arbitrary order; synchronization would be unnecessary.

For human-in-the-loop (HITL) simulation, the argument is advanced (correctly, as it appears to us) that δ should be small enough that the participants are unlikely to notice any adverse effects, but large enough to accommodate differences in event routing due to network communication latency. Typically, a δ of 200 milliseconds or less is used.

One researcher is currently examining the notion of generalizing simulation timestamps so that a simulationist can specify a range of times during which an event may be executed [9]. The idea is that a simulation model often contains enough uncertainty in the input data or the algorithm (or both) such that precise determination of event times is not necessary. Instead, a range of valid times for each event is specified; the simulation engine is free to execute the event anytime within the range. When event times are no longer single points, but rather a range of points, there are more simultaneous events and a larger space of potentially correct answers.

5.4 Implications for Parallel Simulation

Fast-time parallel simulators are typically used for analytic simulations, and as we noted before, such simulators need to present the average of a statistically significant sample of all tie-breaking combinations. The parallel simulator will achieve logical correctness if and only if, upon rollback, the average is taken over the same sample of event orderings. The sample also needs to be consistent with the sample taken by the sequential execution. These observations beg for a mechanism to generate a rollback-robust, statistically significant sampling of event orderings, which is beyond the scope of this paper.

There is yet a second implication for parallel simulation. Running a statistically significant subset of the many different orderings of simultaneous events can be a computationally challenging task. One technique is to parallelize the simulation by branching at each simultaneous event point. If there are $N!$ possible simultaneous event orderings, and a sample of K is chosen, then the simulation can create K clones of itself, each of

which follows a different simultaneous event sequence. This procedure should be repeated every time a set of simultaneous events is encountered.

Branching through dynamic clone creation is reported in [10]. In the context of simultaneous events, the main advantage of dynamic clone creation is that, with enough processors, the various branches will complete in roughly the same time, yielding an estimate for the answer without having to wait for K separate individual executions. This procedure becomes particularly advantageous if there are several different sets of simultaneous events at different points in the simulation. Its chief disadvantage is that it requires many (potentially a combinatorially large) number of processors, as well as back-end statistical calculations to compute the final answer. It may be one area where otherwise serial simulations can effectively exploit idle processors.

5.5 Another Technique for Resolving Event Simultaneity

In many cases the *behavior* of the logical process may change in the presence of simultaneous events. If so, it is necessary for the simulationist to specify a separate behavior for handling simultaneous events. Returning to our queueing example, suppose the arrival of customer A and B within the time interval δ causes A and B to behave differently than if they had arrived separately. For example, if the server is idle, either A or B might speed up to try to “beat” the other to the queue. Or perhaps the two represent data packets that will cause contention on a communication line such that re-transmission is required. In such cases, one procedure is necessary for the simultaneous occurrence of the two events (which includes occurrences where the two events are at different times but within the threshold δ), and a separate procedure for isolated occurrences of the events (in which the two are separate in time by more than δ).

This approach is different than explicitly setting priorities for A and B. For the priority approach, the simulationist would specify which of the two (A or B) would be processed first. For this approach, the simulationist would specify two completely different routines, representing two different behaviors of the system. In our queueing example, the simulationist would specify one routine in which customers enter the queue alone, and another in which they enter simultaneously with other customers. To realize this approach, the simulation engine must recognize simultaneous events and call the appropriate user-supplied simultaneity routine.

This approach suggests yet a third meaning for δ . The event simultaneity threshold can be interpreted as the span of time during which two or more occurrences of an event can affect each other in such a way that all need to be considered before any one is processed.

6. Conclusions

While seemingly mundane, the concept of simultaneous events actually provides a deep understanding into issues of logical correctness in parallel simulators, as well as “correct” results for simulations in general. It has been the thesis of this paper that arbitrarily choosing a tie-breaking strategy to resolve simultaneous events, which most serial and parallel simulators currently do, is the incorrect way of handling the problem. The

method advocated in this paper is either to treat simultaneous events as a separate type of event, or to simulate all possible orderings of simultaneous events (or some sample of them) and average them to determine the “correct” result. In the latter case, parallel techniques such as dynamic cloning of a simulation are particularly useful.

In addition to proposing a “correct answer” for simulations containing simultaneous events, we have also effectively generalized the concept of simultaneous events to include any two events whose time tags are within an interval δ . How the parameter δ is interpreted depends upon the context of the simulation. For analytic simulations, δ can represent the precision of the model; for HITL simulations, the participant’s threshold of perception; and for any type of simulation, it can represent the interval during which two events reciprocally affect each other. The main implication of this paper is that future simulation engines, both sequential and parallel, need to be restructured to account for multiple simultaneous event sequencing. Although this conclusion may be controversial, we hope that it begins to stimulate a discussion on the *meaning* of simultaneous events.

7. Acknowledgments

The author wishes to thank Bill Niedringhaus for suggesting the title of the paper, and Eric Blair for his insights. Georgia Tech provided the GTW parallel discrete-event simulation software. For further information on GTW, contact Professor Richard Fujimoto, Georgia Tech College of Computing, Georgia Institute of Technology, Atlanta, GA, USA, 30332-0280, fujimoto@cc.gatech.edu, <http://www.cc.gatech.edu/fac/Richard.Fujimoto>.

The contents of this paper reflect the views of the author. Neither the Federal Aviation Administration nor the Department of Transportation makes any warranty or guarantee, or promise, expressed or implied, concerning the content or accuracy of the views expressed herein.

8. References

- [1] Fujimoto, R. “Parallel Discrete-Event Simulation.” *Communications of the ACM*, Vol. 33, No. 10, pp 30-53.
- [2] Schriber, T.J. *Simulation Using GPSS*, John Wiley & Sons, 1974.
- [3] Pritsker, A.B. *Introduction to Simulation and SLAM II*, Third Edition, Halsted Press, John Wiley & Sons, 1986.
- [4] Pegden, C.D. *Introduction to SIMAN*, Systems Modeling Corporation, December 1982.
- [5] Cota, B.A., Sargent, R.G. “Simultaneous Events and Distributed Simulation.” *Proceedings of the 1990 Winter Simulation Conference*, December 1990.
- [6] Mehl, H. “A Deterministic Tie-Breaking Scheme for Sequential and Distributed Simulation.” *Proceedings of the 6th workshop on Parallel and Distributed Simulation*, January 1992.
- [7] Chow, A.C.-H. “Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism and its Distributed Simulator.” *TRANSACTIONS of the Society for Computer Simulation International*, Vol. 13, No. 2, pp 55-67, February 1996.
- [8] Wieland, Frederick “Parallel Simulation for Aviation Applications.” *Proceedings of the 1998 Winter Simulation Conference*, Washington, DC, December 1998.

- [9] Fujimoto, R. Private communication, October 1998.
- [10] Hybinette, M., Fujimoto, R. “Dynamic Virtual Logical Proceses.” *Proceedings of the 12th Workshop on Parallel and Distributed Simulation Conference*, (IEEE, SCS) Banff, Canada, May 1998.

9. Additional Reading

- Blair, E., et al. “A Distributed Simulation Model of Air Traffic in the National Airspace System.” *Proceedings of the 1995 Winter Simulation Conference*, December 1995.
- Das, S., Fujimoto, R., Panesar, K., Allison, D., Hybinette, M. “GTW: A Time Warp System for Shared Memory Multiprocessors.” *Proceedings of the 1994 Winter Simulation Conference*, December 1994.
- Russell, E.C. *Building Simulation Models with SIMSCRIPT II.5*, CACI Inc., Los Angeles, 1983.
- Zeigler, B.P., Kim, T.G., Praehofer, H. *Theory of Modeling and Simulation*, Academic Press, New York; second edition to be published, 1999.
- Zeigler, B.P., Moon, Y., Kim, D., Ball, G. “The DEVS Environment for High Performance Modeling and Simulation.” *IEEE Computational Science and Engineering*, pp 61-71, July-September 1997.



Frederick Wieland holds an MSc Degree in Information Systems from the Claremont Graduate School, and a BS degree in Astrophysics from Cal Tech. He is currently finishing doctoral work in Information Technology. For most of his career, Mr. Wieland has worked on parallel modeling and simulation, and most recently has developed a widely-used parallel simulation of global air traffic.