Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

TOWARD REAL TIME SIMULATION: PROTOTYPING OF A LARGE SCALE PARALLEL GROUND TARGET SIMULATION

John B. Gilmer Jr.
David W. O'Brien
Jeffery E. Payne

BDM International, Inc.
1300 N. 17th St., Suite 950
Arlington, VA 22209

ABSTRACT

Large scale detailed simulation is used to drive simulations of sensors and systems for development and testing. One such simulation is the Dynamic Ground Target Simulator (DGTS), used at the Rome Air Development Center. However, the current implementation of the simulation is too slow, too small in area covered, and lacks some functionality necessary for long term needs. An analysis of the simulation was conducted, which looked at the performance characteristics of the existing simulation and projected a need for three or more orders of magnitude greater computational performance than that of the existing system. Technologies were surveyed, and a design plan originated which is expected to achieve the desired performance utilizing coarse grain shared memory parallel processing. The technique chosen can be characterized as having consistent time, in that groups of events of the same or nearly the same time are executed together as a set of parallel tasks. The state space management mechanism ensures determinism and task independence by handling state data changes as updates while maintaining a read-only reference copy of the state. This technique is appropriate to this simulation due to the large proportion of time-step like event processing, as for movement, and the requirements of modeling a perception rich environment in which efficient read access to information about one object by another is important.

1. BACKGROUND

A number of large scale simulation problems are of an event stepped architecture, but have large numbers of events or objects to be processed which are simultaneous and regularly sequenced. The combat simulations of interest to the authors are a case in point. Movement, perception and some other processes (which are continuous in the real world) are modeled in a discrete time stepped manner, while messages, decisions, and other mode changes are modeled as events.

Most other parallel event processing strategies such as Time Warp [Jefferson and Sowizval 1985], Moving Time Window [Sokol and Briscoe 1986], and the Misra-Chandy method [Bagrodia et al. 1987] are based on allowing events which are at different times to be processed in parallel. This is at a cost of requiring all object interactions to be by message. Limiting all interactions to the form of messages prevents the use of very fast direct memory accesses of a shared state space, and so fails to take advantage of the principal strength of shared memory machines. For simulations in which events are randomly and continuously distributed in time, simultaneous processing of objects which are at different times is necessary to achieve speedup. This inconsistent time characteristic of the simulation state space has not been addressed by other than object oriented techniques mentioned.

However, for the large scale simulations of interest, events and processing is highly correlated in time, allowing an approach in which the state space is maintained at a consistent time, while all of the events for that time are processed in parallel. The consistency of time allows direct access to state variables without the protocol and overhead of messages (although a message like facade may be used for software engineering reasons). An extreme case of such a simulation is one that is purely timestepped. A benchmark effort has shown very good performance for a large scale simulation used by the U.S. Army, CORBAN, which typically has several hundred objects, and has been run at about 50% efficiency on a 40 node GP-1000 [Gilmer 1987, 1988].

The end result sought in the current effort is high performance for a more general class of simulations that is event driven, and includes a significant mix of both correlated and uncorrelated event types. The particular simulation of interest is DGTS, originally developed by PAR Government Systems Corporation (PGSC), and used at the U.S. Air Force Rome Air Development Center. That simulation currently runs at least a few orders of magnitude too slowly on a VAX11/780. DGTS represents a large number of ground and air vehicles moving on a battlefield and interacting with each other as targets for sensors. A previous paper reported on state space representation issues and methods developed in the course of evaluating that simulation, and other reports report the findings of the initial exploratory effort in [Trott et al. 1986; O'Brien et al. 1989; O'Brien and Gilmer 1989].

In the previous work it was discovered that the event time correlation in DGTS is such as to offer the potential of significant speedup. In the case of singular events which do not occur at times common to many other events, there is potential for moving the processing of these events to another time due to event effect latency. This should allow almost all events to be processed simultaneously with numerous other events. The techniques developed require that all events at a given time be completed before any events for the next time be started, allowing the state space to be consistent in time. This paper looks at the mechanics of dispatching such events and the structuring of the event list. The methods should be applicable to any shared memory machine, or distributed memory machine having virtually shared memory supported by very fast message passing hardware.

An effort to prototype the scenario driver on meaningful scale is underway as of this writing with completion expected by the end of the Summer, 1990. The prototype follows earlier structural prototypes which tested event dispatching and state space representation mechanisms appropriate to the parallel environment. The large scale prototype is incorporating large sections of code from the original version of DGTS, using automated translation to make the transition from the original Modeling Description Language (MDL, a variant of PASCAL) to C, while incorporating the parallel state space reference techniques. A new executive was written that groups events into compatible sets that can be dispatched together in parallel. The prototype is run on a BBN TC-2000, a Motorola 88000 machine having about 10 MIP per processor performance, a low latency interconnection network, shared memory, and scaleability to 504 nodes and 8 GBytes RAM.

2. PARALLEL PROCESSING CONSIDERATIONS

On a shared memory parallel machine, there are a number of hazards that can significantly reduce the benefits of parallelism. The first two of these apply to any MIMD processor. A "serial bottleneck" exists if a significant amount of the code must be executed outside of parallel constructs. For a simulation of the type envisioned, a sequential bottleneck exists in the dispatching of events. A second effect is the fact that if the number of tasks is not much larger than the number of processors, or if task execution processing requirements vary greatly, some processors will be idle while waiting for others to finish as the events at a particular time are mostly finished. This is a "load balance" problem in that the processing loads assigned to the various processors varies under a synchronization constraint.

A third potential problem is one of "memory contention", which is applicable to shared memory machines. If numerous processors all try to simultaneously access the same memory, some will be refused access, resulting in delays. On some machines there can also be "switch contention" in which accesses to different memories can cause problems if the switch is near capacity, or a "hot spot" has developed. An earlier machine, the Butterfly, with a switch architecture similar to that of the TC-2000, was demonstrated not to have significant problems from switch contention [Rettberg and Thomas 1986]. Potential problems with memory contention can be dealt with by scattering data structures across the various memories of the machine, so that accesses made to random blocks of data will be uncorrelated in the memories that they access. This means that, in general, access to such data will be remote rather than local, at some additional cost in processing speed. However, on the TC-2000 the time to access remote data is within a factor of 10 of that for local data [BBN 1989]. Since such data references are a low percentage compared to code and local variable references which are generally local, the problem of remote access time is not severe as long as memory contention does not occur. In fact, projections from benchmarks of the current DGTS prototype indicate that by distributing memory to very large numbers of processors (in which case all shared storage can be considered remote) performance will be 61% of that achieved with 100% local data. These results are prior to any optimizations or intelligent use of cache, both of which are possible. The event list structure is distributed to minimize memory contention by scattering the event blocks among the processors.

A generalized form of contention can take place not only for memory, but for entry into a critical region of code which must be used by only one process at a time. Given the state space management techniques developed for this project, this is not expected to occur during the processing associated with particular events. The generation of tasks for each event does involve some critical regions or contended resources, but the extent is minor.

3. A LAYERED VIEW OF TASK ACTIVATIONS

Four distinct types of events within DGTS have been identified:

1. S-type: Single events which activate an individual entity. Example: "start a radio transmission"
2. O-type: Similar to S-type events, but tend to appear in simultaneous clusters in time. Example: "react to vehicle detection"
3. N-type: Single events which activate large numbers of objects. Example: "move all objects of a given type"
4. M-type: Involve interactions of a single entity with many others. Example: "attempt to detect all other objects"

The possibilities for parallelism occur in four distinct layers, which happen to correspond to the events identified above, as illustrated in Figure 1.

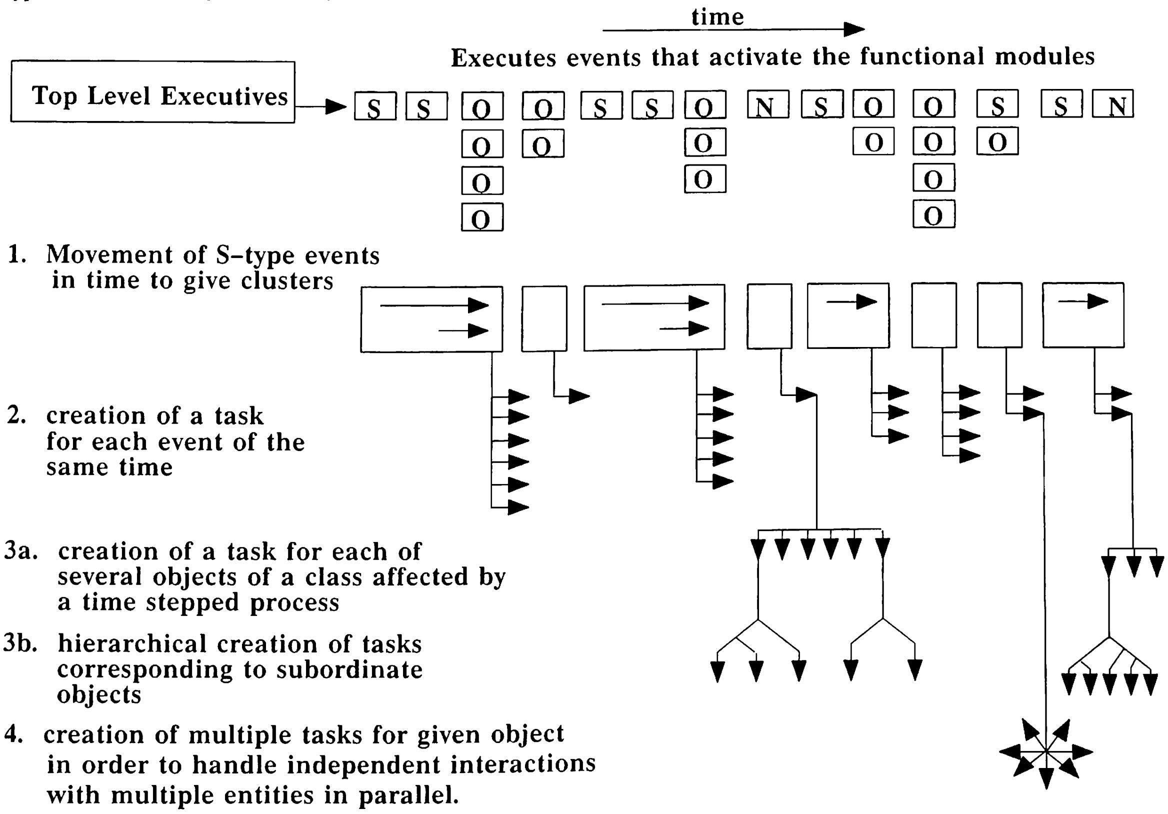

time
Executes events that activate the functional modules
Top Level Executives
S
O
N
1. Movement of S-type events in time to give clusters
2. creation of a task for each event of the same time
3a. creation of a task for each of several objects of a class affected by a time stepped process
3b. hierarchical creation of tasks corresponding to subordinate objects
4. creation of multiple tasks for given object in order to handle independent interactions with multiple entities in parallel.

Figure 1. A Layered View of Task Activations

Parallel execution of events which are scheduled for different times is in general quite difficult. These events tend to have a unique time, and correspond to the kind of events for which Time Warp etc. are optimized. However, many S-type events have latencies associated with their effects, or have independent effects such as message transmission, or for some other reason may be shifted to some extent in their execution time to allow clustering. This allows S-type events to be converted into O-type events. The methods and conditions under which this kind of time shifting can be done has not yet been fully explored. It is not yet known how critical this will be to overall simulation performance. It is believed that as the simulation scales the proportion of singular S-type events will decrease, and that they will constitute an even smaller proportion of overall processing.

Parallel execution of events which are scheduled for the same time is the most basic source of parallelism. If the state space management issues are dealt with to allow the simultaneous events to be independent in their effects as well, then this kind of parallelism is at surface fairly straightforward. A parallel event task dispatcher is used to create a task for each event. This use of parallelism specifically applies to O-type events, and N and M-types with which they may happen to be clustered.

Another opportunity for parallel execution is for processes that apply to a objects of a large set simultaneously. The creation of a task for each of many objects which are activated together is a most necessary form of parallelism for DGTS. Such N-type events constitute the bulk of the processing in the DGTS simulation as it currently exists. Conceptually, such parallelism should be fairly straightforward: one need only create a task for each of the objects to which an N-type event pertains. In the case of hierarchically organized objects such as formations, however, this is more complicated. Hierarchical activations could be considered a separate level potential parallelism below the parallel activation of "top-level" for independent entities.

Use of parallel tasks within the processing of a single object's event is most directly applicable to M-type events where an entity interacts with numerous other entities in fairly independent fashion. The most obvious example is perception, where an object conducts trials against particular vehicles or regions to determine what can be detected. Such events can be large, and have the potential to be broken up into parallel tasks corresponding to individual targets, regions, or groups of targets or regions. The goal of isolating parallelism to the simulation executive is difficult to reconcile with the very representation specific nature of such tasking. It will probably be necessary to embed parallel structures in functional modules to achieve this form of parallelism. Parallelism of this kind will be deferred unless it proves necessary. Whether immediately utilized or not, it remains a potential source of improved performance.

4. THE EVENT LIST AND THE SHARED MEMORY PARALLELISM CONTEXT

For the class of simulation of interest one can expect to have many events with equal time values, for which ordering is unimportant. It is appropriate to modify the simple sequential list often used for event lists to a binary tree having a branch for every distinct time, as shown in figure 2. The left branch, or time header nodes, are a convenient repository for a count of the number of events for that time. The event selection is similar to that of the sequential event list except that an entire list of events having the same time are selected for execution. (The set can be thought of as one event for selection purposes). Each event selected for that time needs to be initiated as a parallel task.

Many of the techniques of structuring the event list used in sequential simulations to minimize time to add or delete an event are equally applicable here, with the "events" so structured being the event headers in this usage. These optimizations are not inherent to parallelism, are not believed

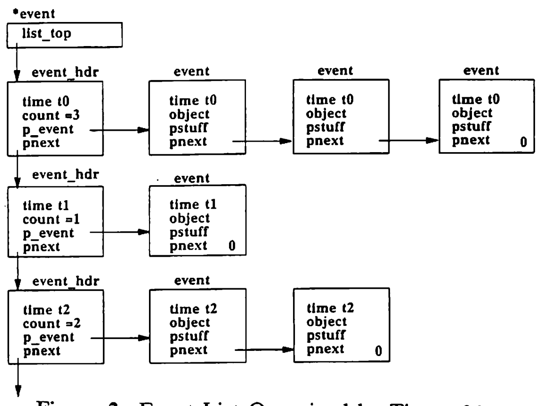


Figure 2. Event List Organized by Time of Event

necessary on the basis of statistics gathered on the sequential version. However, more clever structuring of the event list could be added later.

5. STATE SPACE REPRESENTATION REQUIREMENTS

The requirements listed below are driven by the specific current structures and software methodology of DGTS, the shared memory consistent time design approach to parallelizing it, and the desire to conform to good software practice to the extent we can given the state of knowledge for this context.

The objects in the simulation state must be accessed such that they are available to processes on multiple processors simultaneously. This implies they are in global memory. Objects must be accessible without creating contention for their access structures. This implies duplicated and scattered elements of the access structures similar in principle to the scattered array supported by the Uniform System for programming the TC-2000.

The duplication of objects to create scratch copies and update of the state should be supported by utilities, and should be transparent to the functional event code. This event code should be the same for either scratch or reference copies of objects, and the functional code should not have to be conscious of which is being manipulated. Parallel processing of updates without access structure contention must be supported. If possible, overlap of processing of events and updates should be allowed.

Since on the TC-2000 code is duplicated on every processor while global data (including the state space) is not, software solutions should seek to minimize the code, even at the expense of more data. For example, packing of variables in the state space is counterproductive.

The state space representation issue is how to configure the state data so that determinism is preserved, routines that implement events operate correctly, and updates can be made efficient. State data falls into the following categories:

1. State data that may only be read or written by a single type of event in the context of the owning object, with only one such event possible at a given time. No synchronization hazard exists for such data, so such events may be executed safely with any others. The limit of movement of such an event in time will depend on the time latency of its effects (for example, the receipt time of a message generated).
2. State data that is private to a given object is safe from synchronization hazards as long as there is no more than one event affecting that object in any given event set. Thus, in such circumstances no special state space structuring is necessary. However, if there is more than one event for that object, some form of update recording or scratch space is needed.
3. Visible state data in the objects for which a given event is called can potentially be perceived, and

possibly written, by other objects. To make matters even more complicated, some of the changes are accessed from within the code that implements the change, so that these data cannot be simply accessed in their original "read only" state. For example, as the movement routine processes formation, after each incremental change of the state data for all of its vehicles, the routine "calculate centroid" is called which accesses the updated locations. Some such state data may be updated more than once.

4. Global variables such as terrain that may simultaneously be accessed by other events, but are not changed by events being processed. For these, a simple read only access to the previous value is sufficient.

In general, for any event, the state data at the beginning of the event may need to be copied to supply a version that can be updated and referenced within the event, leaving the original for read-only reference by other, coincidental events. In addition, when more than one event generates changes for the same object, it will be necessary at the end of event processing to combine their updates into a single copy that becomes the reference for the next set of events. The handling of state space will need to go through a number of steps listed below:

1. When an event set is initiated, it must be determined whether any special update treatment is needed. If there is only one event (of any type besides N), or if all events are independent in the objects they affect (for example O-type events affecting only the scheduled object), then the updates to the state space need no special treatment, since no synchronization hazard exists. However, if there is any possibility of simultaneous write and read of the same object data by two tasks, state space management is necessary. Information on circumstances must be passed to the tasks associated with each event.
2. When a particular event is initiated, and a parallel access hazard has been found to exist as discussed above, the event must cause to be created an update copy of the data structures that may be modified. (Possibly the creation could be deferred until an actual change takes place, but such a complication is beyond the capabilities of the current prototype, since it would need to use address error exception handling to recognize and respond to such a case).
3. The code that implements the event must be caused to reference the proper copies of various structures, both for the object whose structures are to be updated, and perhaps for those other objects merely referenced. In most, but not necessarily all cases, the object being modified is that for which the event is scheduled.
4. Upon completion of an event, the processor handling the event must catalog any state update information for later incorporation into the reference copy of the simulation state, and reset its access paths to reference only the reference state.
5. After the event set completes, the updates must be reincorporated into the reference copy of the state space. In most cases there will be at most one update for a particular structure, in which case either copying or substitution can be used. Substitution is quicker, but cannot be used if there are pointers to the structure from elsewhere (unless a method is included for tracking down and changing those pointers). If more than one update to a given structure exists, the updates must be combined. This is done field by field, with some fields added or concatenated (to give a sum or list), and others replaced, as appropriate to the particular datum.

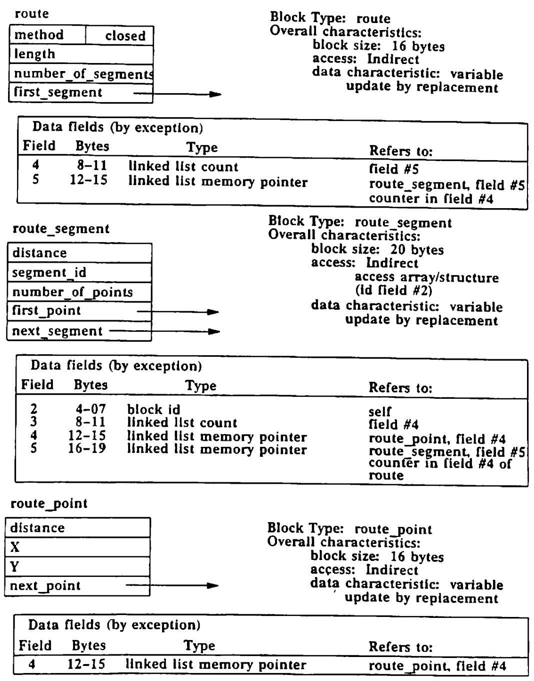
One possible effect of an event is the allocation of a new object. Space does not permit an elaboration of the treatment of this case. The general approach is to provide a global resource protected by a lock for allocation of objects which must have unique identifiers. The newly created or deleted objects and modifications to the numbers of objects are reflected in the reference state space only as updates of various sorts are incorporated.

6. SUPPORTING META-INFORMATION ABOUT EVENTS AND STRUCTURES

To support the above steps, the simulation (or, more properly, its executive) will have to have available to it meta-information about the events, objects, and structures that are used by the simulation. This must include:

1. Event characteristics:
   a. Event type (S,O,M,N) to support reasoning about event combinations
   b. Object access: data to allow an event, including those of type N, to find the objects whose point of view is being processed, and whose state data is accessed, and whose state data is changed.
   c. A means of identifying those data structures and fields to be updated.
   d. A means of identifying how to combine multiple updates on an individual field basis.
2. Data Structure characteristics- needed to support the above
   a. Data structure access characterizations of a general nature.
   b. Data block overall characteristics-length, associated objects.
   c. Links to other Data blocks (characterized)
   d. Identification of fields (that may require update)
   e. Identification of data combiner operations by field.

The basic approach proposed for the data structures is a system of block descriptors which characterize each type of data structure component which is stored in contiguous memory. Each block descriptor includes a list of field descriptors that characterize the relevant information about the fields. The block descriptor is illustrated by example in figure 3.



route

method	closed
length	
number_of_segments	
first_segment	

Block Type: route
Overall characteristics:
block size: 16 bytes
access: Indirect
data characteristic: variable
update by replacement

Data fields (by exception)

Field	Bytes	Type	Refers to:
4	8-11	linked list count	field #5
5	12-15	linked list memory pointer	route_segment, field #5 counter in field #4

route_segment

distance
segment_id
number_of_points
first_point
next_segment

Block Type: route_segment
Overall characteristics:
block size: 20 bytes
access: Indirect
access array/structure
(id field #2)
data characteristic: variable
update by replacement

Data fields (by exception)

Field	Bytes	Type	Refers to:
2	4-07	block id	self
3	8-11	linked list count	field #4
4	12-15	linked list memory pointer	route_point, field #4
5	16-19	linked list memory pointer	route_segment, field #5 counter in field #4 of route

route_point

distance
X
Y
next_point

Block Type: route_point
Overall characteristics:
block size: 16 bytes
access: Indirect
data characteristic: variable
update by replacement

Data fields (by exception)

Field	Bytes	Type	Refers to:
4	12-15	linked list memory pointer	route_point, field #4

Figure 3. Block Descriptors used for a Route Structure

The illustrated route structure block descriptors provide sufficient information to allow a process to traverse and make a scratch or update copy of the entire structure. When a structure is copied, the copy will be at different memory locations, so the pointers will necessarily have different values from those in the original copy. The simulation executive uses the above structure information, together with an expression from the event descriptor to guide traversal, in order to create any necessary scratch/update copies of objects for a particular event. An expression that allows the entire route structure to be traversed is:

sp5p4p4*^(5(4pr*^))*^^

where s=traverse using the route structure pointer (start), 5=traverse using field 5, 4=traverse using field 4, a "p" following a number implies a stack push of the block (pointer), ^ means a stack pop, and * means iteration for an arbitrary number of times (possibly zero). It is possible that for some purposes, for example an event that may change the segments but not the points, the entire data structure does not have to be traversed, copied, or updated. In such cases a different and simpler expression would apply.

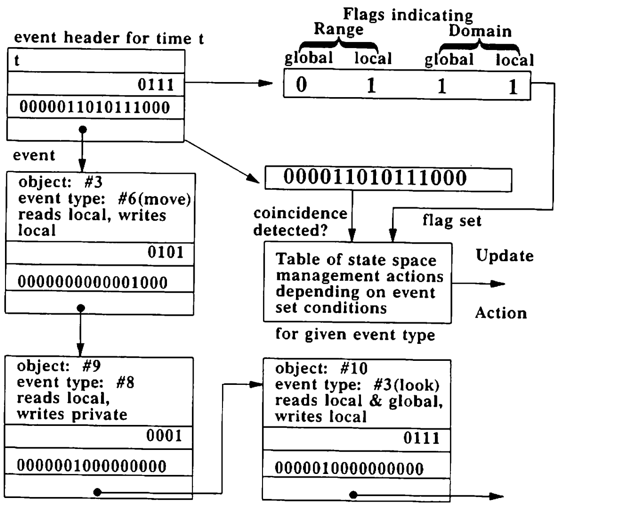
The traversal of data structures using such expressions can also be used as a way to implement modeled events. For example, an event to change the speed of a formation in fact simply traverses the structure, setting the new value for speed in all of the components. With an appropriate addition to the language, this can be done at the same time as object copying, saving the overhead of an extra traversal. If the event set is such that the changes can be made to the reference copy, the expression would still be used, for variable setting but not object copying. Traversal based on such expressions can also be used to spawn hierarchical tasks for subordinate object processing.

The event descriptors contain information on what kinds of data may be referenced or written by events of that type, in the form of a series of flags for reads and writes of private data that might be referenced by multiple events, globally visible data belonging to the object, and data belonging to other objects. A logical OR over such flag arrays for all events in the event set characterizes the bounds of interactions for the event set as a whole. The flag set indexes into an array for each particular event type to return the action that must be taken to set up the state space for that event. Figure 4 illustrates this process. This method provides flexibility to allow the overhead for state space setup to be minimized, especially for small event sets whose processing would be a parallelism bottleneck. The flag operations needed over all events are performed when events are scheduled, thus are part of the parallel task doing the scheduling and hence not a serial bottleneck. (Cancellation of an event cannot undo the flagged implications, but this only causes unnecessary state space setup at worst).

An additional optimization is a second set of flags into which all object identifiers would hash. Absence of collisions in this flag space would indicate the absence of multiple events for the same object. Such a condition allows avoidance of special state space setup when all events change only private data.

7. STATE SPACE VARIABLES AND ACCESS STRUCTURES

The basic form of access structures proposed is illustrated in figure 5. Objects are commonly referenced by side and identifier in DGTS. An identifier, or id, is comparable to a serial number- it uniquely identifies an instance of an object of a given type and side. The access structure adds one additional level of pointer beyond those for side and identifier to indicate the type of data block (object) to which access is sought. This enables objects to be accessed using utility type code applicable to all kinds of objects. Access to a particular attribute variable of an object would look like the following, exclusive of bounds checking that is omitted in the interest of clarity:



Notes:

Private data are defined such that multiple events cannot possibly access the same data simultaneously.

Local data are possibly accessible by multiple events, but only those associated with that particular object.

Global data are accessible by other objects.

The object identity flags refer to local (but not private) data accesses, to detect the possibility of multiple object events for the same object.

Event set header contains flag fields that are the union of indicators for all events in the set (logical or).

Figure 4. Method to Determine the State Space Management Measures for a Given Event of an Event Set

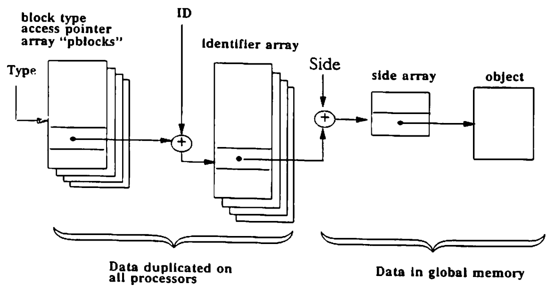

block type access pointer array "pblocks"
ID
Identifier array
Side
side array
object
Type
Data duplicated on all processors
Data in global memory

Figure 5. Basic State Space Access Method for References

value = (structure*) pblocks[structure_id][identifier][side]->field;

The first two arrays used in accessing an object are local, an identical copy residing on every processor. Thus, these copies cannot be contention bottlenecks. The pointers for each of the types of object, pblocks[], will never need to change while the simulation is running, and the size of the array is known at compile time. The lists of pointers (indexed by identifier) for each object type will be more dynamic. Since the maximum number of objects of different types is very scenario dependent, the sizes of these arrays is best determined at the start of a simulation run. Once allocated, these blocks should be static. Each of the elements points to an array, indexed by side, that resides in global memory. All of the lists indexed by identifier on the respective processors would remain identical. If there is a contention problem for the globally located side arrays, it will actually be memory or object contention rather than an access structure contention, since there would be at most two objects associated with each such array. This is quite unlikely given wide scattering of these side indexed access arrays across the various memories of the machine. The array indexed by side points to the copy of the object itself, which is also in global memory.

The above discussion concerns the access of objects in the reference, or read-only, copy of the state space. Some functional routines will need to have access to a scratch copy of an object, which will be modified, eventually becoming part of the update set. The proposed method for accessing such objects is illustrated in figure 6. Note that the access path is very similar to that for the reference copy, the only difference being that the side array is located in the local memory of the processor executing the event requiring the scratch copy of the object. The scratch object it points to is dynamically allocated and initialized with the same data as the reference copy when the event is initiated. It is necessary to point the other side pointer back to the reference copy of the other object. The pointer to the local (scratch) copy of the side array replaces in the identifier array the original pointer to the reference copy. Thus, there is no change in the code, and the functional event code need not be conscious of whether it is accessing a scratch or reference copy. The simulation executive, as it dispatches and event, will decide whether a scratch copy is needed, and generate it if necessary.

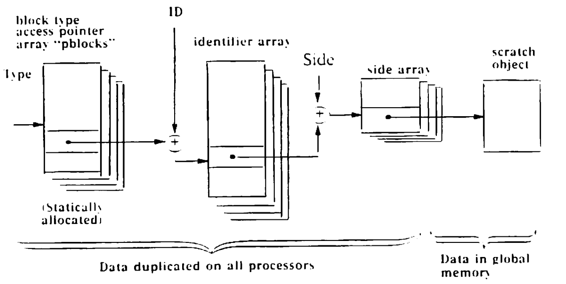


Figure 6. State Space Access Method for Scratch Objects

The scratch copy exists for the duration of the event. The pointer in the identifier array, which was changed to point to the scratch copy, is changed back at the end of the event, and the scratch copy will (usually) be turned into an update. To support changing pointers in the identifier list, arrays having the unchanging pointers to the reference copy side arrays and the scratch copy side arrays need to be maintained. These arrays are accessed through object descriptors, which contain various information about the different kinds of objects. An additional array is used to indicate, on each processor, the status of the reference path for each block: reference, scratch, or reference with write updates to be made on a field basis.

When a write is made to an object from within a functional module, it may do so within one of a number of contexts, which differ in how they are handled:

1. The write is made to the reference copy: This will normally happen only for single events or sets of data-indeoendent events for which no synchronization problem can occur. The overhead of the normal update processing is avoided by allowing writes directly to the reference version of the state space. The object access path is the same as for a read reference.
2. The write is made to a scratch copy: This is the "normal" case. The reference is exactly like the accessing of a scratch copy for read purposes, and has the same access sequence as for the above case.
3. The write is stored in the form of a field update, which will modify the reference object later. This is appropriate when a scratch copy is not needed, and only one or a few variables are changed. The advantage in this case is that the overhead of allocating and redirecting to a scratch copy of the object is avoided. Also, if several processes are modifying the same field of a given object, the field update method will be easier and quicker to process. The disadvantage is that the variable changing process as seen from the functional module is quite different form simply writing to a copy of an object.

The method appropriate to a given invocation of the procedure which writes an object variable may be different at different times. For a singular event, the write may be directed to the reference copy, while for the next discrete time an event may generate a field update. This complicates the state variable storage for any variable which may be updated by a field event. The following code fragment illustrates variable setting:

```
if (pstatus[structure_id][identifier][side] = FIELD_UPDATE)
   update(structure_id, side,identifier,field_id,value);
else
   (structure*)pblocks[structure_id][identifier][side]->field =
value;
```

For procedures that set more than one value, or set attributes having more than one component, the above can be easily expanded in like manner to the translation of read access procedures. Again, bounds checking has been omitted for clarity.

8. UPDATE LISTING AND CONSOLIDATION

When an event concludes, the scratch object or objects that were created for it must be cleared out of the way for the next event, and pointers in the identifier array restored. The scratch copies will in most cases contain the data or data changes that will need to be incorporated into the next time reference state. To avoid the need to traverse all possible objects and id's to find the updates for any given event set, it is necessary to list update blocks in a separate scattered array. The array is scattered to prevent contention by the parallel tasks doing updating. The details of these update list structures are illustrated in figure 7. As events execute (or conclude), updates are placed on the update lists. The number of header blocks, N, can be adjusted to the number of processors to trade off both potential contention and the number of lists that must be searched to complete updates. (N <= the number of processors). The lock in the header must be used to gain access to the list to allow adding updates during event set processing or removing them to make the updates top the object during update processing. Access to the reference copy of an object is made as usual using the same access structures as for references during event set processing, as shown in figures 5 and 6. To ensure synchronization, the list should not be unlocked until the update is completed for object updates that replace the previous copy of an object.

Since the update process will be generic, it will have to have available information about where the various fields are in the object being updated. This information will be in the object descriptors. For each field, its offset and (if packed) length and shift will be given. Thus, the update process will be able to extract from the update block the object type and field to be updated, the identity and side of the object instance, and will be able to make the change for a field update.

If an object update can be by replacement of the entire object by the update object, the task is simply and quickly accomplished, and the earlier copy can be thrown away. For the case of object types that may be referenced by pointer from within the state space, as indicated by flag in their object descriptors, object replacement will actually be accomplished by copying the data into the original block rather than by switching blocks. Other special cases concern the destruction or creation of an object. The destruction would be signaled by an additional kind of update, so that the reference copy of an object is disposed of without replacement. For creation an object starts as scratch and then update copies, with the update copy becoming the reference copy upon update.

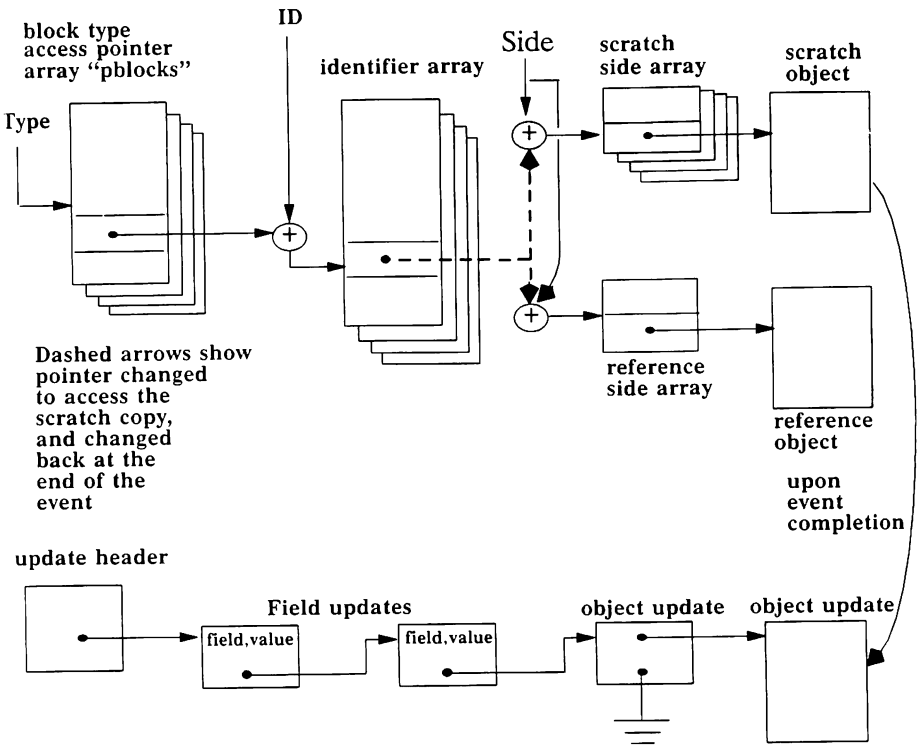


Figure 7. Scratch Objects as made into Update Objects at the end of Event Processing

An object update may be more involved than mere replacement. If two object updates are made, the two have to be combined into the replacement object in some fashion. The field descriptors in the object descriptor will include instructions on how this is to be done. In most cases it will involve going through the objects field by field. Thus, this mode will generally be slower than field updates, although it is the most general.

When an object update is to be made by replacement, it is important that it occur before the field updates for that object, or the latter will be lost. This order may be ensured when appending update lists by hashing the object by identifier to a particular scattered list, so that updates to it will always be in the same list, and appending object updates to the head of the list and field updates after all object updates.

9. EARLY RESULTS

An initial prototype subset of DGTS using the state space methods described and parallelism for O and N type events is operational as of August 1990. Run times for about a two division scenario, 5315 vehicles, have been about 40 times real time on an eight processor TC-2000, with five processors executing the simulation, one generating graphics output, and the others performing system services and public access. This reflects the effects of high performance for the individual 88000 based processors as well as parallelism. Benchmarks with up to twelve processors dedicated to the simulation showed no appreciable serial bottlenecking. A pronounced performance drop when scaling from one to two nodes is due to the fact that all of the state space is no longer local to the processor accessing it. Switch delays in memory access are a significant factor. Processing times and speedups are plotted in figures 8 and 9. In these tests, the state space was distributed among all of the processors even though a smaller number was actually used for processing (cluster = 12; processors = N in Figure 8.) In the case of a single processor, the state was confined to the numbers of processors being applied to the program (cluster = N, processors = 1 in Figure 8). These results reflect the porting of nearly 22K lines of the existing VAX/VMS Pascal code to the TC2000. So far, no special efforts have been made toward optimization, such as correlating tasking with the location of object data. The prototype will continue to be developed and its performance analyzed.

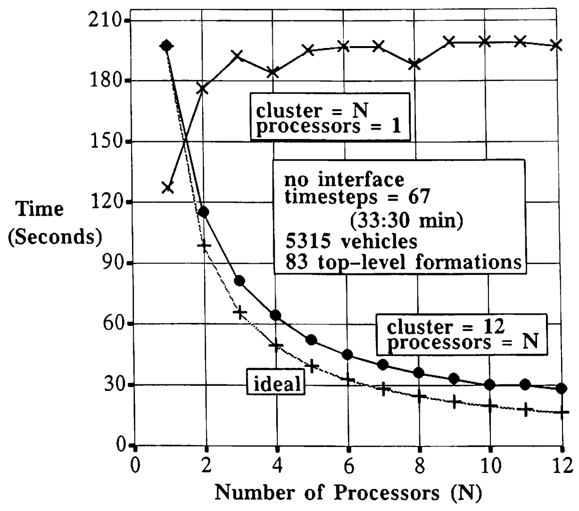


Figure 8. DGTS 2 Division Scenario on TC2000 - Time Benchmark

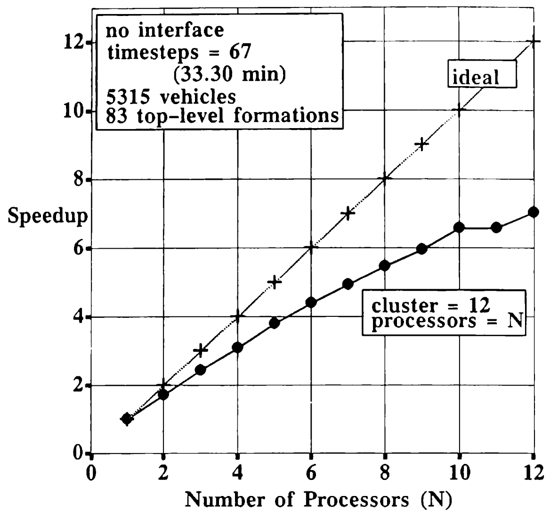


Figure 9. DGTS 2 Division Scenario on TC2000 - Speedup Benchmark

10. CONCLUSION

In summary, the approach adopted is a pessimistic event stepped scheme in which information about event compatibility is used to allow processing to proceed in parallel. All events having the same time are assumed to be, and must be, independent, though they can affect the same variables, and can be executed in parallel. Thus, the method is basically a consistent-time approach. However, some singular events which are compatible with an event set at a nearby time may be shifted to process in that event set as a means of increasing parallelism. Information structures support reasoning about event compatibility and the state space measures needed to avoid synchronization and determinism hazards to allow overhead to be minimized for small event sets.

The approach taken seems well suited to the type of simulation representing continuous as well as discrete processes on large scale. Much of the focus of this work has been directed toward event dispatching and state space structures to allow removal of synchronization hazards while allowing efficient, direct access to variables. A prototyping effort is underway to test the approach using the more important parts of the actual DGTS simulation with full scale data.

ACKNOWLEDGMENT

The work reported here was supported by the U.S. Air Force Rome Air Development Center (RADC) under contract F30602-89-C-0114. We would also like to acknowledge the contributions made to this project by Kevin Trott of PGSC, who has helped with our understanding of the DGTS simulation, and Chris Fortier and J. Richard Mills of BDM, who have contributed greatly to the simulation translation, prototype development, and display processes.

REFERENCES

BBN Advanced Computers Inc. "TC2000 Technical Product Summary," (1989), Cambridge, MA., for a general description, and conversations with BBN ACI concerning memory latency.

Bagrodia, R., K.M. Chandy, and J. Misra (1987), "A Message-Based Approach to Discrete-Event Simulation," *IEEE Transactions on Software Engineering, SE-13,* 6, 654-665. Numerous other articles by Misra and Chandy also describe their method and variations of it.

Gilmer, J.B., (1987), "Parallel Combat Simulation Research to Support The AirLand Battle Management Program," *In Proceedings, 57th Military Operations Research Society Symposium, MORS.*

Gilmer, J.B. (1988), "Shared Memory Parallel Combat Simulation," *The BDM Corporation, BDM/ROS-88-00360-TR*

Jefferson, D. and H. Sowizval (1985), "Fast Concurrent Simulation using the Time Warp Mechanism," In *Proceedings of the SCS Distributed Simulation Conference,* San Diego, CA.

O'Brien, D.W., J.B. Gilmer, and P.A. Merkel (1989), "Real Time Simulation Study," *RADC-TR-89-43,* Rome Air Development Center, NY.

O'Brien, D.W. and J.B. Gilmer (1989), "Mixed Event and Time-Stepped Parallel Simulation," to be published in the *Proceedings, 1989 CSC Multiconference on Distributed Simulation.* Describes conclusions reached after examination of a large Air Force simulation, and a strategy for parallelism.

Rettberg, R. and R. Thomas (1986), "Contention is no Obstacle to Shared Memory Multiprocessing", *Communications of the ACM, 29,* 12, 1202-1212.

Sokol, L. and D. Briscoe (1986), "A Time Window Solution to Scheduling Simulation Events on Parallel Processors," In *Proceeding Expert Systems in Government Symposium.* This is one alternative approach for event stepped parallelism.

Trott, K.C., L.L. Ross, and C.C. Brown (1986) *Program Maintenance Manual, 1, DGTS System Maintenance Manual,* PAR Government Systems Corporation, New Hartford, NY.