Proceedings of the 1993 Winter Simulation Conference
G.W. Evans, M. Mollaghasemi, E.C. Russell, W.E. Biles (eds.)

INCREMENTAL STATE SAVING IN SPEEDES USING C++

Jeff S. Steinman

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, Mail Stop 138-310
Pasadena, CA 91109
(818) 354-3793, JSS@PEBBLES.JPL.NASA.GOV

ABSTRACT

Optimistic parallel discrete-event simulation engines require a rollback mechanism to restore the state of a simulation object to an earlier time if its events are accidentally processed out of time order. Traditional approaches save the entire state of an object before its next event is processed so that its original state can be restored. However, this approach can very quickly use up all of a processor's available memory. In addition, the overhead for copying large amounts of memory can be quite high. Incremental state saving helps to overcome these problems by saving only the changes that an event makes to an object.

This paper describes some reversible incremental state saving techniques (i.e., the Delta Exchange Method and the Rollback Queue Mechanism) that are used by the object-oriented C++ SPEEDES (Synchronous Parallel Environment for Emulation and Discrete-Event Simulation) operating system. The need for *reversible* incremental state saving is emphasized by a brief discussion on lazy cancellation techniques which require *roll-forward* as well as *rollback* support.

1. INTRODUCTION

The most challenging goal in the field of parallel discrete-event simulation is to provide general mechanisms for correct event processing (i.e., processing time-tagged events for simulated objects in their ascending time order) while simultaneously achieving high parallel performance (i.e., *speedup*). Ensuring correctness is difficult in a parallel simulation where objects are distributed among multiple processors because each processor tends to advance at its own rate. Because of this characteristic, it is possible for an event on a *slow* node to schedule an event for an object on a *fast* node in the object's past (see Figure 1).

There are two schools of thought on how to solve this problem. Various conservative approaches (Chandy 79, Fujimoto 90) ensure that events are only processed if it is known that no earlier event will be scheduled for that object. Another, and yet very different, approach for solving this problem is to process events optimistically (Jefferson 85, Fujimoto 90) without concern for time accidents (i.e., events that are processed out of order). However, when a time accident occurs, that simulation object, and others that might have been affected from events that were erroneously generated, are rolled back.

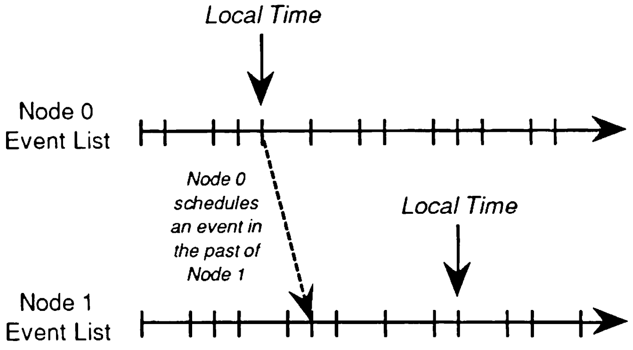


Figure 1: The problem of synchronizing parallel discrete-event simulations. There are two basic approaches: *Conservative* approaches never allow events to be processed if an earlier event for that object might be scheduled. *Optimistic* approaches process events without regard for synchronization, but then roll the simulation back to the time when events were processed out of order to fix the simulation error.

Optimistic parallel discrete-event simulation algorithms, such as those supported by SPEEDES (e.g., Breathing Time Buckets, Time Warp, Breathing Time Warp, COOP, and potentially others), demand such a rollback capability (Steinman 92a). Further, when lazy cancellation (Fujimoto 90, Reiher 90) is used, events must also be able to roll forward as well as backward. SPEEDES uses some new reversible techniques for incremental state saving that support both the rollback and roll-forward requirements. The rest of this paper describes how SPEEDES accomplishes this.

2. MEASUREMENTS

All of the rollback techniques that are mentioned in this paper were carefully measured and are reported in Tables 1 through 5. The measurements were taken on a Silicon Graphics IRIS-4D machine, which has proven to be very reliable for providing repeatable measurements.

The IRIS-4D is a shared-memory machine with four processors (only one processor was used for these measurements), each rated at about 20 million instructions per second (Mips). The optimization option in the C++, Version 2.1 compiler was enabled.

3. EVENTS AS OBJECTS

SPEEDES (Steinman 92a) is a highly object-oriented C++ environment (Stroustrup 86). It differs from other simulation environments in that its events are fully encapsulated objects, separate from the simulation objects (see Figure 2). Because the internal data structures of the simulation objects are hidden from events, modifying the state of a simulation object is only accomplished through method calls. In other words, the object-oriented technique of data encapsulation is built into the philosophy of SPEEDES. There are other benefits for events being C++ objects. They will be discussed shortly.

In SPEEDES, an event is attached to one simulation object. This means that each event has a pointer to its specific simulation object. As an event is processed, it may modify the state variables of the simulation object, but only in a reversible manner using either the Delta Exchange mechanism, and/or the Rollback Queue mechanism (these mechanisms will be elaborated on later in this paper).

Messages (which represent future events that have been scheduled in the course of processing an event) are saved in the event object and released to other nodes according to the synchronization strategy. The Multi-Router object routes messages to their destination nodes where, upon arrival, they create and initialize event objects. Antimessages (in the case of Time Warp) are stored in the event object if messages are sent with a risk of the event's being rolled back.

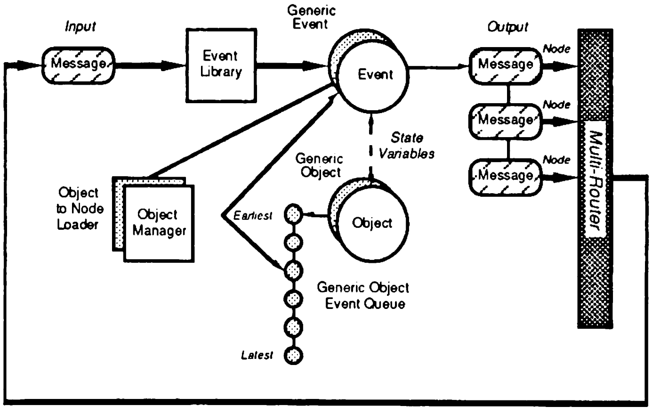


Figure 2: Events as objects.

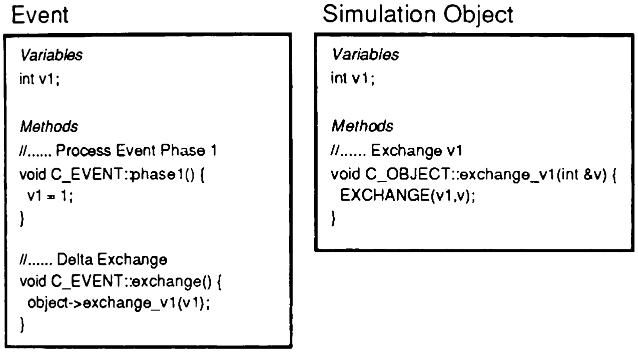
One might think that separating events from their objects is a foolhardy thing to do. However, upon closer inspection, making this separation is very desirable in large, complicated simulations for the following reasons:

1. Simulation objects can be developed independently from the SPEEDES simulation environment if event code is separated from the simulation objects. This separation promotes portability and reusability. Scientists can develop their simulation object models without any knowledge of the SPEEDES operating system.
2. Simulation objects are encapsulated entities that do not know about other simulation objects. By eliminating event code from simulation objects, complexity of inheritance chains (assuming that the same event code works with different top-level objects by using virtual functions) for simulation objects is reduced. This promotes independence of the simulation objects from one another, which in the long run reduces the complexity of large simulations.
3. Events are independent from other events because they are fully encapsulated. When an event schedules another event, the scheduling event does not require knowledge of the methods for the scheduled event, nor does it share any of its variables. Because of the independence of events, multiple simulation developers can work in the same environment without interfering with, or being affected by, each other's codes.
4. Events being separate from simulation objects promotes a high-level capability for object abstraction. Events do not have to know about the top layer of a simulation object's inheritance chain. Virtual functions provided by base classes of the simulation object can be used by events in a very generic way. Similarly, events can be built on top of lower base-class events through inheritance. This again simplifies and promotes reusability of software.
5. Events being separate from simulation objects keeps the size of the simulation object manageable. It would defeat the purpose of encapsulation — one of the most important object-oriented programming techniques — to dump all of the event code into the various simulation objects. Event code is typically very different from the body of code in the simulation object. Since they are fundamentally two different entities, they should be separated.
6. Events contain (in a base class hidden from the application developer) important rollback information. Because rollbacks occur on an event-by-event basis, events, as objects, are the logical place to store this information. In addition, events, as objects, contain their own data structures, which are ideal for supporting rollbacks using the Delta Exchange mechanism, a very efficient way of handling incremental state saving. The Delta Exchange method is discussed in the next section of this paper.
7. Events can store important input information in their internal data structures to support sophisticated optimization techniques such as lazy cancellation, which will be discussed later.
8. Event routines are accessed by the SPEEDES operating system through virtual function calls. Using these calls makes the user interface for "plugging" event code into the simulation very elegant. The user simply informs SPEEDES of all the event objects in the simulation.
9. Special features can be embedded in events. For example, the user may enable lazy cancellation for specific events (hopefully, the ones most likely to benefit from lazy cancellation) while using aggressive cancellation as the default for the rest.
10. Future development and new ideas can be easily supported in an environment that separates events as objects from the simulation objects. There is more room for growth because

events as objects are more powerful than events as method calls in a simulation object. For example, events can be processed in multiple stages that are supported as virtual function calls for the event object. This kind of processing may be very difficult or unwieldy to support if events are not objects.

4. THE DELTA EXCHANGE METHOD

One of the most efficient ways to handle incremental state saving in parallel discrete-event simulations is the Delta Exchange mechanism. This functionality can be supported by using special-purpose hardware (Fujimoto 92), or very efficiently in SPEEDES through software (Steinman 92a). The SPEEDES approach is that when an event wants to change the state of a variable in the simulation object, it does so by exchanging this variable with a similar variable that is stored in the data structure of the event (see Figure 3).



```
Event
Variables
int v1;

Methods
//...... Process Event Phase 1
void C_EVENT::phase1() {
  v1 = 1;
}

//...... Delta Exchange
void C_EVENT::exchange() {
  object->exchange_v1(v1);
}
```

```
Simulation Object
Variables
int v1;

Methods
//...... Exchange v1
void C_OBJECT::exchange_v1(int &v) {
  EXCHANGE(v1,v);
}
```

Figure 3: The Delta Exchange.

State changes of this type can easily be accomplished by dividing event processing into two steps. The first step does the basic event processing (Phase 1 in SPEEDES) while the second step (Delta Exchange) exchanges the new state values with the old state values. After the Delta Exchange method is called, the simulation object has the new state values, and the event has the old values. If the Delta Exchange method is called again, those variables will be restored, and the simulation object will return to its original state.

To make the Delta Exchange easier for the user, an inline function named EXCHANGE has been defined that is overloaded in SPEEDES. This function accepts various types of arguments (possibly of different types). By using function overloading in C++, EXCHANGE knows how to exchange the values of different types of arguments. For example, in Figure 3, the EXCHANGE function is simple (accepting two integer variables as arguments) and looks like this:

```
void EXCHANGE(int &i1, int &i2) {
  int temp;
  temp = i1;
  i1 = i2;
  i2 = temp;
}
```

As another example, EXCHANGE, accepting two double precision arguments, looks like this:

```
void EXCHANGE(double &d1, double &d2) {
  double temp;
  temp = d1;
  d1 = d2;
  d2 = temp;
}
```

A final note concerning the Delta Exchange mechanism: One might complain that this procedure for exchanging values places too much of a burden on the simulation developer. Those who complain about the burdens of object-oriented programming techniques such as data encapsulation typically say the same thing. However, in reality, there is no more work required by the user for the Delta Exchange method than there is in encapsulating an object's data. As object-oriented programming techniques begin to dominate the world of software development, criticisms of the Delta Exchange method should fall by the wayside.

Table 1 shows the extra overhead required when using the Delta Exchange method for changing the value of an integer variable contained within an object. When running conservatively, the code in the EXCHANGE routine was redefined to be just one line of code, *i1 = i2* (see the example above). In these conditions, there was no measured difference in overhead for storing an integer value. The second column shows the timing for normal optimistic processing. The third column shows the overhead for performing a rollback. The fourth column shows the extra memory overhead for performing the Delta Exchange.

Table 1: Encapsulated assignments and Delta Exchange.

	Conserv.	Optimistic	Rollback	Memory
Assign	0.123 µs	0.123 µs	0.0 µs	0 bytes
Exchange	0.123 µs	0.265 µs	0.265 µs	4 bytes

5. THE ROLLBACK QUEUE

There are times when the simple Delta Exchange mechanism doesn't support what one really wants to do. Some examples of this are dynamic memory allocations, adding or removing an item from a dynamic data structure (such as a list or a tree) contained within an object, memory copies, and methods inside the simulation object (that are called by events) that change values of its internal state variables. Another mechanism is needed to incrementally save the state of the object as it is modified by the processing of an event. This is accomplished by using the Rollback Queue in SPEEDES.

The Rollback Queue saves the changes that occur when an event modifies the state of its simulation object. Because events are rolled back on an event-by-event basis, this queue is encapsulated through a hidden base class in the event object. However, before each event is processed, SPEEDES makes this Rollback Queue available to the simulation object as well, providing a way (although not recommended for reasons of efficiency) for the simulation object to alter its state in its own methods.

When an event does something to change the state of its simulation object, a *rollback item* is pushed onto the top of the Rollback Queue. Specific rollback items (which are C++ objects) inherit from the base class Rollback Item object.

```
//...... Rollback Item base class
//...... Inherits from C_ITEM so that
//...... C_RB_ITEMS can be put into a C_QUEUE object

class C_RB_ITEM : public C_ITEM {
  private:
  protected:
    void *ptr;      // pointer to altered state variable
  public:
    void set_ptr(void *p) {ptr = p;}  // set the pointer
    virtual void store(int v) {;}     // virtual store variable
    virtual void store(float v) {;}   // virtual store variable
    virtual void store(double v) {;}  // virtual store variable
    virtual void store(void *v) {;}   // virtual store variable
    virtual void cleanup() {;}        // virtual cleanup
    virtual void rollback() {;}       // virtual rollback
}
```

The base-class Rollback Item defines three virtual functions: *store*, *rollback*, and *cleanup*. Through the use of these virtual functions, storing values, rolling back an event, or cleaning up the rollback queue is very easy. Adding new types of rollback items into the SPEEDES environment becomes very easy with this approach.

Rolling back an event is accomplished by popping rollback items out of the Rollback Queue, and then calling their rollback virtual function to undo whatever was done. In the code given below, the rollback items are then stored in reverse order in another list called *qreverse*. This is to provide the roll-forward capability needed for lazy cancellation.

After an event is successfully processed (e.g., at garbage collection time in Time Warp), its rollback information must be cleaned up. This is accomplished by popping the rollback items out of the Rollback Queue, and then calling its cleanup virtual function. It should be noted that some of the rollback items do not require any work for cleanup. For these cases, the base-class virtual cleanup function is called, but does no processing.

Shown below is sample code for rolling back and cleaning up the Rollback Queue for an event:

```
//...... rolling back the Rollback Queue

int rollback() {
  C_RB_ITEM *a;
  C_QUEUE qreverse;
  while (Rollback_Queue->length()) {
    a = (C_RB_ITEM *)Rollback_Queue->pop_top();
    a->rollback();
    qreverse.push_top(a);  // save for lazy cancellation
  }
  *Rollback_Queue = qreverse;
  return Rollback_Queue->length();  // return n rollbacks
}


//...... clean up the Rollback Queue

void cleanup() {
  C_RB_ITEM *a;
  while (Rollback_Queue->length()) {
    a = (C_RB_ITEM *)Rollback_Queue->pop_top();
    a->cleanup();
    delete(a);
  }
}
```

6. SIMPLE ASSIGNMENTS

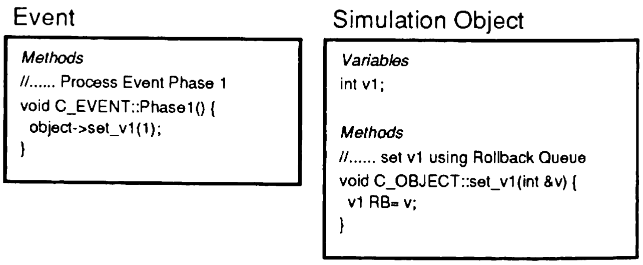
One of the impressive features of C++ is its ability to overload operators. SPEEDES provides a new assignment operator for simple variable assignments. Instead of using the equal sign ("=") for assignments, SPEEDES supports a new operator, "RB=", for changing state variables inside simulation objects while providing rollback capability.

This new assignment operator (through various techniques employed, which are not described in this paper) automatically creates a new rollback item, saves assignment information, and then pushes it onto the top of the Rollback Queue. When running sequentially (or when running conservative protocols), the "RB" symbol can be defined as null so that "RB=" reverts back to the simple "=" assignment operator, thus removing all of the rollback overhead.

```
//...... Rollback Item for integer assignments

class C_RB_ITEM_INT : public C_RB_ITEM {
  private:
    int oldvalue;  // old value saved here
  protected:
  public:
    void set_oldvalue(int v) {oldvalue = v;}
    virtual void store(int v) {*(int *) ptr = v;}
    virtual void store(float v) {*(int *) ptr = (int)v;}
    virtual void store(double v) {*(int *) ptr = (int)v;}
    virtual void rollback() {
        int temp;
        temp = oldvalue;
        *oldvalue = *(int *)ptr;
        *(int *)ptr = temp;
    }
}
```

Figure 4 shows how a simple assignment might be made in processing an event (compare this with Figure 3).



```
//...... Process Event Phase 1
void C_EVENT::Phase1() {
  object->set_v1(1);
}
```

```
int v1;

//...... set v1 using Rollback Queue
void C_OBJECT::set_v1(int &v) {
  v1 RB= v;
}
```

Figure 4: Simple assignments using the Rollback Queue with the overloaded "RB=" assignment operator.

There is much more overhead using this method of altering the state of the simulation object compared to the Delta Exchange method. Using the Rollback Queue requires creating a rollback item, storing the address of the state variable to be modified in the rollback item, saving the old state value in the rollback item, storing the new value in the simulation object's state variable, and then pushing the rollback item onto the top of the Rollback Queue. This amount of overhead is considerably larger than that required for using the simple EXCHANGE code.

Further, the memory consumption when making assignments using the Rollback Queue is much larger than when using the Delta Exchange mechanism. Therefore, it is strongly recommended to choose the Delta Exchange approach whenever possible. Table 2 compares the overhead between "=" vs. "RB=".

Table 2: Assignments using "=" and "RB=".

	Conserv.	Optimistic	Rollback	Memory
=	0.002 µs	0.002 µs	0.0 µs	0 bytes
RB=	0.002 µs	8.39 µs	2.15 µs	20 bytes

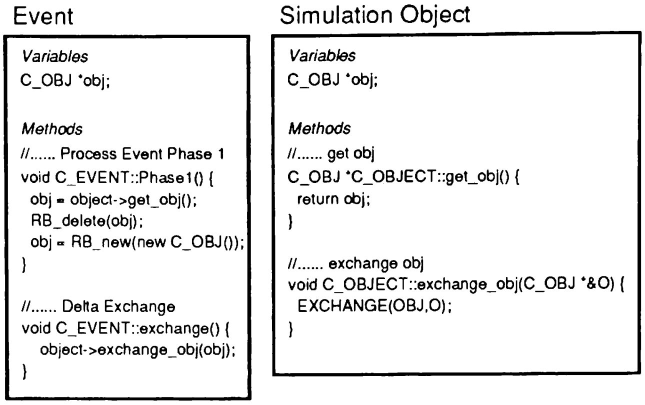
7. DYNAMIC MEMORY ALLOCATIONS

One very important requirement for an optimistic simulation environment is to allow the user to dynamically create or destroy memory. Performing this function in an incremental state-saving mechanism is simple. If an event creates memory, then upon rollback, this memory must be deleted. Similarly, if an event deletes memory, then upon rollback, the memory should be freed again (this means that, in reality, the memory is actually only deleted during cleanup). This is all easily accommodated by the Rollback Queue.

```
//...... Rollback Item for dynamic memory

class C_RB_ITEM_MEMORY : public C_RB_ITEM {
  private:
    int flag;        // flag for creation/deletion
  protected:
  public:
    void set_flag(int f) {flag = f;}
    virtual void cleanup() {
        if (!flag) delete ptr;
    }
    virtual void rollback() {
        if (flag) {
          flag = 0;
        }else{
          flag = 1;
        }
    }
}
```

Figure 5 shows how an event might delete a C++ object stored in a simulation object, and then create a new one and save it in the simulation object. This example combines both the Delta Exchange Mechanism and the Rollback Queue in the same event.



```
Variables
C_OBJ *obj;

Methods
//...... Process Event Phase 1
void C_EVENT::Phase1() {
 obj = object->get_obj();
 RB_delete(obj);
 obj = RB_new(new C_OBJ());
}

//...... Delta Exchange
void C_EVENT::exchange() {
   object->exchange_obj(obj);
}
```

```
Variables
C_OBJ *obj;

Methods
//...... get obj
C_OBJ *C_OBJECT::get_obj() {
 return obj;
}

//...... exchange obj
void C_OBJECT::exchange_obj(C_OBJ *&O) {
 EXCHANGE(OBJ,O);
}
```

Figure 5: Dynamic memory.

In the next example, representative code is shown for RB_new and RB_delete. In actuality, SPEEDES optimizes the performance of all dynamic memory allocations by using free lists. When free lists are used, memory blocks are never deleted. Rather, these memory blocks are saved in a list to be reused later when the memory is needed again. Using free lists accelerates memory allocation and also reduces the problem of memory fragmentation. How SPEEDES supports free lists is not discussed in this paper.

```
//...... allocate dynamic memory

void RB_new(void *obj) {
  C_RB_ITEM_MEMORY *a;
  a = new C_RB_ITEM_MEMORY();
  a->set_ptr(obj);
  a->set_flag(1);
  Rollback_Queue->push_top(a);
}
//...... delete dynamic memory

void RB_delete(void *obj) {
  C_RB_ITEM_MEMORY *a;
  a = new C_RB_ITEM_MEMORY();
  a->set_ptr(obj);
  a->set_flag(0);
  Rollback_Queue->push_top(a);
}
```

Table 3 shows the overhead for dynamic memory allocations. The measurements were taken by creating, and then destroying, the same object 1,000,000 times. The timings given here are for the combined *new* and *delete* (or RB_new and RB_delete) operations. The memory overhead, however, is for a single RB_new or a single RB_delete operation.

Table 3: Dynamic memory allocations.

	Conserv.	Optimistic	Rollback	Memory
new delete	6.42 µs	6.42 µs	0.0 µs	0 bytes
RB_new RB_delete	7.84 µs	18.83 µs	5.87 µs	20 bytes

8. MEMORY COPIES

Some simulation applications may require events to copy blocks of memory into the state of the simulation object. This is easily accomplished in SPEEDES through the Rollback Queue. A rollback item for memory copying is defined as follows:

```
//...... Rollback Item for dynamic memory

class C_RB_ITEM_MEMCPY : public C_RB_ITEM {
  private:
    char *oldvalue          // old value of buffer
    int size;               // size of the memory copy
  protected:
  public:
    void set_oldvalue(char *ov) {oldvalue = ov;}
    void set_size(int s) {size = s;}
    virtual void cleanup() {
        delete oldvalue;
    }
    virtual void rollback() {
        char *temp;
        temp = new char[size];
        memcpy(temp,ptr,size);
        memcpy(ptr,oldvalue,size);
        memcpy(oldvalue,temp,size);
        delete temp;
    }
}
```

The SPEEDES function RB_memcpy provides this service.

```
//...... memory copy using Rollback Queue

void RB_memcpy(char *p1, char *p2, int size) {
  C_RB_ITEM_MEMCPY *a;
  char *oldvalue;
  a = new C_RB_ITEM_MEMCPY();
  a->set_ptr(p1);
  a->set_size(size);
  oldvalue = new char[size];
  memcpy(oldvalue,p1,size);
  a->set_oldvalue(oldvalue);
  Rollback_Queue->push_top(a);
  memcpy(p1,p2,size);
}
```

Table 4 describes the overhead for performing memory copies.

Table 4: Memory copies of 1000 bytes.

	Conserv.	Optimistic	Rollback	Memory
memcpy	46.7 μs	46.7 μs	0.0 μs	0 bytes
RB memcpy	47.7 μs	108.5 μs	150.2 μs	1024 bytes

9. THE EXCHANGE QUEUE

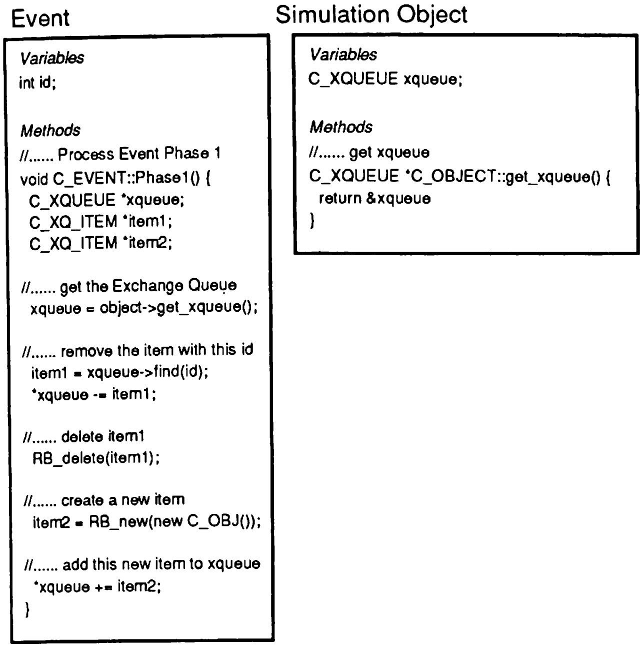
So far, this paper has described how SPEEDES currently supports incremental state saving for various types of operations. Simple assignments are supported by using either the Delta Exchange mechanism or the "RB=" operator. Dynamic memory allocation is supported through the RB_new and RB_delete functions. Memory copies are supported through the RB_memcpy function. In this section, management of dynamic data structures such as lists or trees contained within simulation objects is described. Here, a very useful data structure supported by SPEEDES, called the Exchange Queue, is discussed. The results can be generalized for other dynamic data structures. This section completes the list of incremental state-saving techniques that are currently supported by the SPEEDES operating system.

As part of its state, a simulation object may contain a list of items. This list may grow (or shrink) as events add (or remove) items to (from) the list. The Exchange Queue uses reversible operations that make rollback easy. For example, rolling back an event that inserts an item into the Exchange Queue requires removing that item from the Exchange Queue to restore the state. Similarly, rolling back an event that removes an item from the Exchange Queue requires reinserting that item into the Exchange Queue. For the example below, it is assumed that the Exchange Queue is a doubly linked list that holds to the first-in, first-out (FIFO) property.

```
//...... Rollback Item for the Exchange Queue

class C_RB_ITEM_XQUEUE : public C_RB_ITEM {
  private:
    C_XQUEUE *xq            // pointer to Exchange Queue
    int flag;               // flag for insertion or removal
    int top_flag;           // flag for top or bot
  protected:
  public:
    void set_xq(C_XQUEUE *x) {xq = x;}
    void set_flag(int f) {flag = f;}
    void set_top_flag(int f) {top_flag = f;}
    virtual void rollback() {
        if (flag) {
          xq->remove((C_XQ_ITEM *)ptr);
          flag = 0;
        }else{
          if (top_flag) {
            xq->push_top((C_XQ_ITEM *)ptr);
          }else{
            xq->push_bot((C_XQ_ITEM *)ptr);
          }
          flag = 1;
        }
    }
}
```

To make the Exchange Queue easier to use, the "+=" operator was overloaded to mean — add an item to the Exchange Queue, and the "-=" operator was overloaded to mean — remove an item from the Exchange Queue. Figure 6 shows how an event in SPEEDES might remove an item that is specified by an integer ID from an Exchange Queue, delete it, and then add a new item to it. Note that in Figure 6, the FIFO requirement was bypassed. In this example, the order of the items in the Exchange Queue does not matter since items are removed based on their ID. The overloaded operators for "+=" and "-=" are shown in the code below:



Event

```
Variables
int id;

Methods
//...... Process Event Phase 1
void C_EVENT::Phase1() {
 C_XQUEUE *xqueue;
 C_XQ_ITEM *item1;
 C_XQ_ITEM *item2;

//...... get the Exchange Queue
 xqueue = object->get_xqueue();

//...... remove the item with this id
 item1 = xqueue->find(id);
 *xqueue -= item1;

//...... delete item1
 RB_delete(item1);

//...... create a new item
 item2 = RB_new(new C_OBJ());

//...... add this new item to xqueue
 *xqueue += item2;
}
```

Simulation Object

```
Variables
C_XQUEUE xqueue;

Methods
//...... get xqueue
C_XQUEUE *C_OBJECT::get_xqueue() {
 return &xqueue
}
```

Figure 6: Exchange Queue.

```
//...... add an item to the Exchange Queue

void operator += (C_XQUEUE &xq, C_XQ_ITEM *it) {
  C_RB_ITEM_XQUEUE *a;
  a = new C_RB_ITEM_XQUEUE();
  a->set_xq(&xq);
  a->set_ptr(it);
  a->set_flag(1);
  a->set_top_flag(0);
  Rollback_Queue->push_top(a);
  xq.push_bot(it);
}
```

```
//...... remove an item from the Exchange Queue

void operator -= (C_XQUEUE &xq, C_XQ_ITEM *it) {
  C_RB_ITEM_XQUEUE *a;
  a = new C_RB_ITEM_XQUEUE();
  a->set_xq(&xq);
  a->set_ptr(it);
  a->set_flag(0);
  a->set_top_flag(1);
  Rollback_Queue->push_top(a);
  xq.remove(it);
}
```

Table 5 describes the overhead for adding and removing items from the exchange queue. These measurements were taken by adding and removing the same object from the exchange queue 1,000,000 times. The exchange queue, ignoring the object that was added and removed, was otherwise empty. The timing measurements represent the combined operations while the memory overhead represents overhead for a single operation.

Table 5: Adding and removing items from the xqueue.

	Conserv.	Optimistic	Rollback	Memory
add remove	1.29 µs	1.29 µs	0.0 µs	0 bytes
+= -=	2.5 µs	14.06 µs	8.49 µs	28 bytes

10. ROLLBACK AND LAZY CANCELLATION

This section on Lazy Cancellation may seem out of place to some readers. However, it is not, because of the following three reasons.

First of all, one of the important benefits derived by events being C++ objects is that lazy cancellation can very efficiently be supported. This reinforces the notion that events as objects provides a powerful way to build parallel discrete-event simulations. If one accepts the benefits of events being objects, then it becomes easier to embrace the Delta Exchange mechanism which requires events to be objects.

Secondly, by understanding the utility of Lazy Cancellation, the reader will appreciate why incremental state saving techniques require roll-forward as well as rollback capabilities.

Thirdly, by going through the Lazy Cancellation mechanisms described in this section, the reader will get a practical set of examples showing how rollback and roll-forward systems work. Questions accumulated by the reader may be answered in this section through the examples provided.

How SPEEDES handles lazy cancellation is rather interesting and unique. It has been well known in the simulation community that accidentally processing events for a simulation object in the wrong order might not matter (Jefferson 85, Reiher 90, Fujimoto 90). It may turn out that when a rolled-back event is reprocessed, it gets the same answer. In other words, the event makes the same state changes to the simulation object while also generating the same events. This allows for the possibility that a simulation will beat the critical path (Wieland 91). The critical path determines the minimum time for a conservative discrete-event simulation to be completed (ignoring all overhead other than event processing). It is not possible for conservative simulations to beat the critical path because they require events to be processed in their correct time order.

For events that are rolled back and yet have this interesting property, it would be wasteful to send antimessages (if Time Warp is used) only to later generate the same messages all over again. Also, it would be wasteful to reprocess the event if it makes the same changes to the state of its simulation object. It would be better to roll the state of the simulation object forward using the state changes made by the event the first time it was processed. This, then, requires all of the incremental state-saving techniques to be reversible.

Conventional lazy-cancellation approaches for determining if reprocessing a rolled-back event will get the same answer require saving the entire state of the simulation object and then making a byte-for-byte memory comparison with its previous state (i.e., when the event was first processed) with the new state. If the results are identical, then, obviously, processing the event again will give the same answer. This approach is not feasible in an incremental state-saving environment.

Another lazy-cancellation trick that is used to avoid sending unnecessary antimessages is to compare (after reprocessing the event), byte-for-byte, the newly generated messages with the old ones (from the first time the event was processed). If they are identical, the new messages do not have to be sent. However, if they are not identical, antimessages to cancel the old messages must be sent along with the new messages that have just been generated.

SPEEDES does not use either of these conventional methods because they require too much overhead. Instead, SPEEDES uses an object-oriented approach that very quickly accomplishes nearly the same thing (lazy-cancellation) without the overhead. (Sometimes, this technique is also called Lazy Re-evaluation.)

Because events are objects in SPEEDES, the inputs from the simulation object that are required for processing an event can be saved in the data structures of the event object. Before reprocessing the event (after it has been rolled back), a virtual function called *Check_Lazy* is called. This virtual function, which is supplied by the user, compares the previous inputs from the simulation object still stored in the event object with the new values in the simulation object. If they are the same or if it is determined that the event would still get the same answer, then this virtual function returns a 1. Otherwise, it returns a 0.

If the Check_Lazy virtual function returns a 1, the event is rolled forward. Otherwise, antimessages are sent, and the event is reprocessed.

Before looking at sample code that describes this approach, one must first understand that events are processed in SPEEDES by calling two virtual functions supplied by each event object. These virtual functions are *Phase 1* and *Exchange*. Processing an event in SPEEDES then looks something like this:

```
event->init_Rollback_Queue();
event->Phase1();
event->exchange();
event->send_messages();
```

Rolling back a single event (ignoring antimessages, etc.) is done in the reverse order of processing an event. In SPEEDES, it looks something like this:

```
event->exchange();
event->init_Rollback_Queue();
Rollback_Queue->rollback();
if (event->get_lazy_flag()) {
  event->send_antimessages();
}else{
  event->set_lazy_processed();
}
```

After an event has been rolled back, SPEEDES checks a flag stored in the event object that tells if the event is participating in lazy cancellation. This allows events to participate on a selective basis in lazy cancellation. When it is time to process that event again, SPEEDES checks if the event would have gotten the same answer.

An example of how the user may participate in lazy cancellation in SPEEDES is given in Figure 7.

Event

```
Variables
int input1;
double input2;
double output;

Methods
//...... Process Event Phase 1
void C_EVENT::Phase1() {
//...... get inputs
 input1 = object->get_input1();
 input2 = object->get_input2();

//...... process stuff
 output = input1 + 2.0 * input2;
}

//...... Delta Exchange
void C_EVENT::exchange() {
 object->exchange_output(output);
}

//...... Check Lazy
int C_EVENT::check_lazy() {
 if (input1 != object->get_input1())
    return 0;
 if (input2 != object->get_input2())
    return 0;
 return 1;
}
```

Simulation Object

```
Variables
int input1;
double input2;
double output;

Methods
//...... get Input1
int C_OBJECT::get_input1() {
 return input1;
}

//...... get input2
double C_OBJECT::get_input2() {
 return input2;
}

//...... exchange output
void C_OBJECT::exchange_output(double o) {
 EXCHANGE(output, o);
}
```

Figure 7: Lazy cancellation.

Sample code for how SPEEDES reprocesses a lazy event that has been rolled back is given below.

```
//...... Roll-forward or reprocess an event

if (event->get_lazy_processed() {
  if (event->check_lazy() {
    event->init_Rollback_Queue();
    Rollback_Queue->rollback();      // Roll forward
    event->exchange();
  }else{
    event->send_antimessages();
    event->init_Rollback_Queue();
    event->cleanup_Rollback_Queue();
    event->Phase1();                 // Reprocess event
    event->exchange();
    event->send_messages();
  }
}
```

11. A PROPER PERSPECTIVE

So far, various methods for incremental state-saving have been described, each with its measured performance. The natural question is, "In a practical sense, how efficient is this approach for parallel simulations?"

First of all, to understand this question, one must first have a feel for the other areas of overhead in a parallel discrete-event simulation. Consider, for example, the task of managing the list of pending future events. Figure 8 shows timings for various event-list data structures (Steinman 92b). The SPEEDES TREE (actually used in conjunction with the SPEEDES Queue in a hybrid data structure within SPEEDES) easily outperforms the Splay Tree (which is typically thought of as being the most efficient general-purpose data structure for handling the future-event list).

Figure 8 shows that a simulation consisting of about 1,000 objects with 10 events per object, has about 70 μs of overhead for managing the event list (these measurements were also taken on the Silicon Graphics IRIS-4D machine to be consistent with the other measurements).

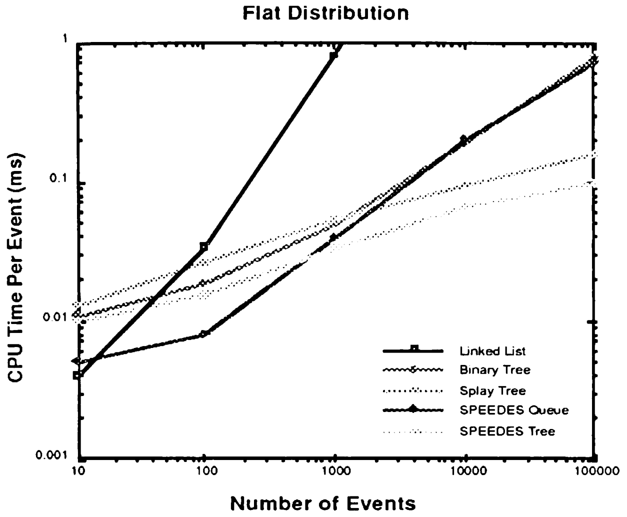


Figure 8: Event-list management overhead for events generated by a flat distribution.

There are other sources of overhead involved in parallel discrete-event simulations. Additional overhead is associated with message sending, memory allocation, event initialization, global synchronization, error checking and handling, etc. The overhead of these things will not be discussed here. However, it has been observed in low-grained sequential simulations running under SPEEDES that event times rarely get below 200 μs per event (Steinman 93). With this in mind, low-grained events that do very little work in their event processing probably do not require very much incremental state-saving either. The few extra microseconds spent supporting incremental state saving are not significant, especially when the Delta Exchange is predominantly used.

One last thing to consider is the overhead for saving the full state of a simulation object (the way normal Time Warp systems work). Table 4 shows the overhead for copying large chunks of memory. For a 10,000-byte object, it would take 477 μs to save the entire state of the object in addition to requiring 10,000 bytes of memory. This does not consider dynamic data structures within the simulation object's state. Full state saving would be very inefficient if the object's next event required only a simple variable assignment that could have been supported with a Delta Exchange (using only four bytes of memory, and taking only 0.142 μs of additional overhead).

12. SUMMARY

Methods for supporting incremental state saving in SPEEDES, (an object-oriented C++ parallel discrete-event simulation environment) were discussed. The SPEEDES operating system is unique in that it separates events from simulation objects. Ten important reasons were given why events should be fully encapsulated C++ objects, separate from the simulation objects.

The Delta Exchange mechanism easily, and very efficiently, allows for incremental state-saving while at the same time holding true to the object-oriented principles of data encapsulation. The Delta Exchange mechanism requires no more work than the standard methods of data encapsulation. It also requires the least amount of overhead for providing reversible incremental state-saving information.

The Rollback Queue can accommodate various types of state-changing requirements such as dynamic memory allocation, memory copies, methods in the simulation object that change the state of its internal data, and dynamic data structures such as the Exchange Queue.

Lazy cancellation may be performed in a very efficient manner that saves the event-processing inputs from the simulation object in the event object, and then later before reprocessing the event, checks with the simulation object to see if these values have changed (or would produce a different result). This approach is much more efficient than making byte-for-byte comparisons of the old state of the simulation object with its new state (as most other approaches do). It is also more flexible. It must be remembered that supporting lazy cancellation requires reversible incremental state-saving techniques. Developers of parallel discrete-event-simulation engines who are unaware of this fact may find that their designs fall short of the state-of-the-art.

Putting everything into perspective, the additional overhead for incremental state saving is small for typical parallel discrete-event simulations. This overhead is normally much smaller than the full state-saving techniques that save the entire state of a simulation object before processing each event. If the techniques for incremental state saving do provide low overhead, as claimed, the need for special-purpose rollback hardware is reduced.

ACKNOWLEDGMENTS

I would like to thank Peggy Li, member of the technical staff at the Jet Propulsion Laboratory, for motivating parts of this paper with some of her own ideas for incremental state saving. Her contribution has greatly improved the user friendliness of the SPEEDES operating system. I would also like to thank Dave Curkendall, manager of the Advanced Laboratory for Parallel High-performance Applications (ALPHA) Project, for his continuing support of the SPEEDES operating system.

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the U.S. Air Force Electronic-Systems Center/XRP "Advanced Programs and Analysis," Hanscom Air Force Base, Massachusetts, through an agreement with the National Aeronautics and Space Administration.

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

REFERENCES

Chandy, K. and Misra, J. 1979. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs." *IEEE Transactions on Software Engineering.* Vol. SE-5, No. 5, pages 440–452.

Fujimoto, R. 1990. "Parallel Discrete Event Simulation." *Communications of the ACM.* Vol. 33, No. 10, pages 30–53.

Fujimoto, R. 1992. "Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp." *IEEE Transactions of Computers.* Vol. 41, No. 1, pages 68–82.

Jefferson D. 1985. "Virtual Time." *ACM Transactions on Programming Languages and Systems.* Vol. 7, No. 3, pages 404-425.

Reiher, P.L., Fujimoto, R.M., Bellenot, S., and Jefferson, D.R. 1990. "Cancellation Strategies in Optimistic Execution Systems", *Proceedings of the SCS Multiconference on Distributed Simulation.* Vol. 22, No. 1, pages 112-121.

Steinman, J. 1992a. "SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete-Event Simulation." *International Journal in Computer Simulation.* Vol. 2, pages 251-286.

Steinman, J. 1992b. "The Event Horizon." *JPL Internal Document D-10029.*

Steinman, J. 1993. "Breathing Time Warp." To be published in *Proceedings of the Federated Computing Research Conference 7'th Workshop on Parallel and Distributed Simulation (PADS93).*

Stroustrup, B. 1986 "The C++ Programming Language", Addison-Wesley Publishing Company, Reading Massachusetts.

Wieland, F. 1991. "A Critical Path Tool for Parallel Simulation Performance Optimization." *Proceedings of the International Conference on System Sciences.*

BIOGRAPHY

Jeff Steinman received B.S. degrees in computer science and in mathematical physics from California State University Northridge in 1980. He then worked at Hughes Aircraft Company in the Radar Systems Group for four years while studying physics at UCLA. In 1988, he received his Ph.D. in experimental high-energy particle physics from UCLA, where he measured the quark content of virtual photons generated at the Stanford Linear Accelerator Center. Since then, he has been a member of the technical staff working at the Jet Propulsion Laboratory building simulations for strategic missile and air defense. Jeff Steinman is the principle developer of the SPEEDES operating system.