Implications of Using the Event Scheduling World View for Parallel Simulation

By D.S. Richardson and M. Abrams

TR 90-44

IMPLICATIONS OF USING THE EVENT SCHEDULING WORLD VIEW FOR PARALLEL SIMULATION

D. S. Richardson
M. Abrams

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061

* To appear in *Proceedings of the 20th Annual Virginia Computer User's Conference*, Blacksburg, Va., Sept 14-16, 1990.

ABSTRACT:

Use of a particular world view, or conceptual framework, makes the programming of many simulation problems easier. Parallel simulation implementations generally have not been based on a particular world view. An open question is what impact a world view has on the execution of a simulation in parallel.

This paper compares the programming interface needed to perform event scheduling simulation sequentially to the interface needed to perform the simulation in parallel. The chief conclusion is that the parallel event scheduling interface looks deceptively similar to the sequential interface. However, the parallel simulation program must be structured to accommodate time and state relationships among events.

CR Categories and Subject Descriptors: I.6.0 **[Simulation and Modeling]:** General; I.6.1 **[Simulation and Modeling]:** Simulation Theory- *model classification, types of simulation;* I.6.2 **[Simulation and Modeling]:** Simulation Languages; D.3.3 **[Programming Languages]:** Language Constructs- *concurrent programming structures;* G.1.0 **[General]:** Parallel Algorithms.

Additional Key Words and Phrases: conceptual frameworks, simulation strategies, parallel processing, distributed simulation.

I. INTRODUCTION

Computer simulation involves organizing and manipulating many simultaneous activities. Highly detailed simulation problems may take hours or days to execute in a uniprocessor environment, and this time factor may be costly or unacceptable to the user. Time may be shortened by making simplifying assumptions to the simulation; however, a favorable alternative to sacrificing detail is to use a multiprocessor or distributed environment to perform the simulation. Different parts of the same simulation can be executed on different processors in parallel, possibly reducing total simulation time [Heidelburger 1986; Jefferson 1987; Baezner 1990].

As simulation execution becomes faster through the use of parallel processing, the problems a simulator is able to solve in a given amount of time will become larger and more complex. Therefore, it becomes increasingly important that a simulation programmer uses all tools available to him to aid in developing, writing and validating a computer simulation.

Basing a simulation model on a particular world view significantly reduces implementation complexity [Balci 1988]. This paper reports on ongoing work to identify the implications of using a particular world view to perform a parallel simulation. The *event scheduling* world view is addressed here, chosen due to its popularity in the United States' simulation community [Balci 1988] and because of its potential to be utilized in the implementation of other world views [Schwetman 1986].

The structure of this paper is as follows: section II gives background information on world views, parallel simulation, and programming interfaces as well as presenting open questions relating to those topics, section III presents the programming interface developed to perform an event scheduling simulation sequentially, section IV explains the corresponding parallel interface and details what features were added to the sequential interface, and section V concludes and gives directions for further research An example contrasting event scheduling simulation using the sequential and parallel interfaces appears in the Appendix.

II. BACKGROUND INFORMATION AND OPEN QUESTIONS

A. SIMULATION WORLD VIEWS

A *world view,* also referred to as a *conceptual framework, simulation strategy,* or *formalism,* is "an underlying structure and organization of ideas which constitute the outline and basic frame that guide a modeler in representing a system" [Derrick 1989]. The use of a world view enables a modeler to implement a simulation with significantly reduced complexity, because the outline for the problem is given by the chosen world view [Balci 1988].

Many world views have been studied (see Derrick [1989]). Three of the most common are the *event scheduling* approach, the *activity scan* approach, and the *process interaction* approach. Briefly these are identified [Nance 1981]:

1) *event scheduling,* where the system state is described in terms of events and their consequences,
2) *activity scan,* where the actions of objects in the system and the conditions for those actions to take place are described, and
3) *process interaction,* which considers a sequence of events and activities through which an object moves.

The choice of world view should be made by understanding the problem domain; some simulation problems match one world view better than another. The choice should also take

into consideration simulation model characteristics such as maintainability, efficiency, modifiability, reusability, and ease of development [Balci 1988].

B. OPTIMISTIC VERSUS CONSERVATIVE PARALLEL SIMULATION METHODS

Parallel simulation mechanisms can be broadly classified as conservative or optimistic.

Conservative approaches, for example the Bryant-Chandy-Misra (BCM) protocol [Bryant 1977; Chandy 1979], strictly avoid the possibility of executing an event out of sequence. Before executing an event, these methods determine whether all events that could possibly affect the event in question have been processed. Only when precursor events are completed will the current event be processed.

The Time Warp method [Jefferson 1985] is the original optimistic approach. Optimistic methods rely on detecting when an error in event sequencing has occurred, and then rolling back the simulation state to an earlier time to correct the error. Rather than waiting for all possible inputs to arrive, a processor optimistically assumes that the input it has at the moment is correct, and proceeds accordingly. The justification is that because the processor would be idle anyway, why not have it working on computations that will be correct some of the time [Fujimoto 1989].

C. PROGRAMMING INTERFACE

A simulation programmer can use a parallel simulation protocol either by:

1) using a simulation language whose run time environment is based on the parallel simulation protocol, or
2) using a traditional programming language that has been augmented by a library of functions and data types. This library provides an interface to all simulation protocols.

This paper assumes the second case. The library helps to accomplish tasks necessary for many applications, so that each programmer is freed from having to reimplement often used functions and data structures. The programming interface presented can be implemented either by conservative or by optimistic protocols.

D. OPEN QUESTIONS REGARDING WORLD VIEWS, PARALLEL SIMULATION LANGUAGES, AND PARALLEL SIMULATION METHODS

Recent parallel simulation languages, including Yaddes [Preiss 1989], ModSim II [Bryan 1989], Sim++ [Lomow 1989], Maisie [Bagrodia 1990], and CPS [Abrams 1989], have generally followed the process interaction world view. However, because messages or events are an integral part of each language, the languages present the user with aspects of the event scheduling world view.

This merging of world views raises several issues. First, what are the implications of using a language that implements a pure event scheduling view versus a language that implements a pure process interaction view, versus a language using another world view? Second, the early simulation protocols of BCM and Time Warp were described using a computation model that mixed process interaction and event scheduling world views. What new simulation protocols are enabled by a language that uses only one world view? Finally, research on world views is intended to simplify the task of building a simulation model. What implication do new developments in world views have on design of parallel simulation protocols? This paper reports ongoing research to address these questions.

III. PROGRAMMING INTERFACE FOR SEQUENTIAL EXECUTION OF EVENT SCHEDULING SIMULATIONS

A. MOTIVATION FOR DEVELOPMENT

A workable programming interface for sequential execution of simulations has been developed so that the additions and changes needed to construct a parallel interface could be more easily identified.

B. FUNCTIONS PROVIDED IN INTERFACE

Generally a simulation model is constructed by first specifying event *types,* then specifying how many *instances* of each event exist.* For example, a queuing network model with fifty identical queues may have two event types *Arrival* and *Departure*, and fifty instances of each event type.

To write a simulation, the simulation programmer identifies and defines the effects of the events in the system. He may then use the interface functions to schedule when event instances are to occur. The *Schedule* function will schedule execution according to the simulation time the event instance is to take place. More than one *Schedule* function is offered to the programmer. If the event instance to be scheduled is to be chosen randomly from a list of possibilities, he may use the *RandomSchedule* function. If all the event instances present on a list are to be scheduled, he may use the *MultiSchedule* function. The *AddToOutArc* and *RemoveFromOutArc* functions manipulate the lists of possible event instances that *RandomSchedule* and *MultiSchedule* can choose from.

Another function offered to manipulate events is the *Cancel* function. This will remove an event instance previously scheduled; *Cancel* is useful in scenarios that allow preemption.

The main driver of a simulation program will call function *ExecuteReplicativeSim* or *ExecuteBatchSim*. These functions are provided to allow a parallel simulator to run multiple replications in parallel. Parameters for the functions determine how many replications to perform, when steady state is reached, how many events instances in each simulation replication, and a pointer to the function that schedules initial event instances to be placed on the empty event list. *ExecuteReplicativeSim* performs a simulation by the method of replication, which clears the event list between simulation replications. *ExecuteBatchSim* performs a simulation by the method of batch means, which maintains the event list from one simulation replication to the next [Powlikowski 1990]. Both will repeatedly take the most immediate event instance from the event list and perform the operations associated with that event, until it is time to terminate. Termination is decided by a boolean function *Done*, which is called by one or more event procedure(s) in the user's program.

The remaining interface functions are: *GetClock* to return the present value of the simulation time, *ClearEvList* and *ResetClock* to prepare for a new simulation replication, boolean functions *ListEmpty* and *EventOnList* to determine the status of the event list, and *Identify* which may be used to identify events without removing them from the list.

Refer to the SIMEXECUTOR class in the Appendix for a description of the interface in C++.

IV. REQUIREMENTS FOR PARALLEL EXECUTION OF EVENT SCHEDULING SIMULATIONS

This section enumerates the additions to the sequential interface needed to perform a simulation in parallel, and explains why the additions are necessary.

A. IDENTIFICATION OF DEPENDENCIES

The concept behind parallel programming is to allow different processors to work on different parts of the program at the same time. Ideally, N processors could complete the program in 1/Nth of the time. Practically, this is never accomplished due to dependencies.

---

* The simulation literature historically has referred to event instances simply as events.

Parts of programs are often dependent on other parts of the program: they may share common variables (state dependency), and some parts must occur before others (time dependency). These dependencies must be identified and not violated, because arbitrary violations may cause arbitrary results.

i) STATE DEPENDENCIES

To execute a parallel program, the components of the program that cannot be run at the same time due to access to common variables must be identified. Because a variable occupies a certain space in memory, and two processors cannot write to the same space at the same time, the possibility of concurrent access must be avoided.

To execute a simulation in parallel, the simulation programmer must make structuring decisions not required for sequential execution. When using the sequential interface as described in section III, a simulation programmer describes *events*, and all information needed for the identification and execution of an event is contained in that description. For parallel execution, the programmer must separate the information into *events* and *states*. The events are the same as in the sequential case, such as arrival to a queue and departure from the queue, but now all variables that are shared by more than one event are placed in a data structure called the state: in this example, the state of a queue. The state of a queue contains information such as the number of jobs in the queue (which arrival increments and departure decrements) and the function to generate a random service time to calculate a departure time (which both arrival and departure access). This is illustrated in Figure 1.

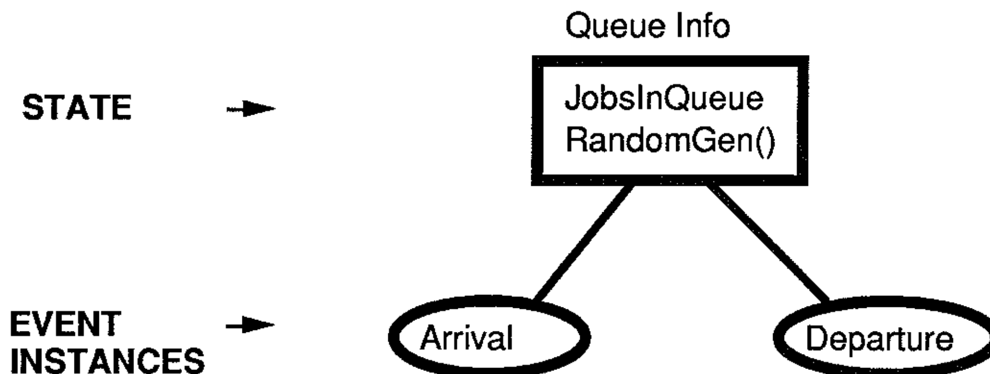

Figure 1: Example of the Relationship Between States and Event Instances

When a simulation programmer creates an event instance, he enumerates which states this event instance can affect, and maintains a list of pointers to states for each event instance. The parallel system uses this knowledge of dependencies to avoid scheduling two event instances at the same time if they both access the same state.

This separation of events and states lends itself well to statistics collection. The information needed for the statistics is placed in a separate state, and only those event instances which affect the statistic will access it. For example, consider a queuing network of two queues, and the calculation of the average number of jobs in the system over some interval of time is desired. The event instances which create jobs and the event instances which destroy jobs will both affect the statistics collection state. This is easily identified when coding the system, and the parallel system will not schedule both event instances at the same time. See Figure 2 for a diagram of this scenario.

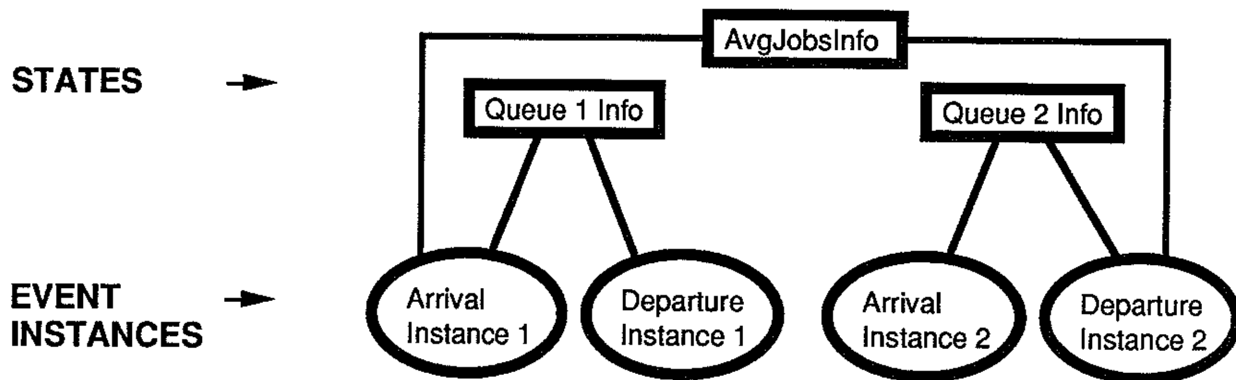

Figure 2: Event Instances Accessing Multiple States

ii) TIME DEPENDENCIES

Conservative parallel simulation methods will proceed with an operation only if all event instances that can possibly affect the operation have been completed (see section II B). Therefore, it must be known by the system which event instances can cause other event instances to occur.

To maintain time dependencies, each event instance needs to contain a list of *input arcs*. If event instance A schedules or cancels event instance B, then B has an input arc from A. This information is used by the conservative parallel implementation to determine when execution of an event instance can proceed, having received information from all possible inputs.

Figure 3 illustrates the use of input arcs. Suppose in our queueing network model three independent servers can schedule an arrival at a fourth server. The event instance corresponding to an arrival at the fourth server maintains a list of input arcs identifying all event instances which can schedule it; this list contains the departure event instances of the first three servers. Before proceeding, the list of input arcs is examined so that the correct choice of which event instance to execute can be made based on the simulation protocol rules.

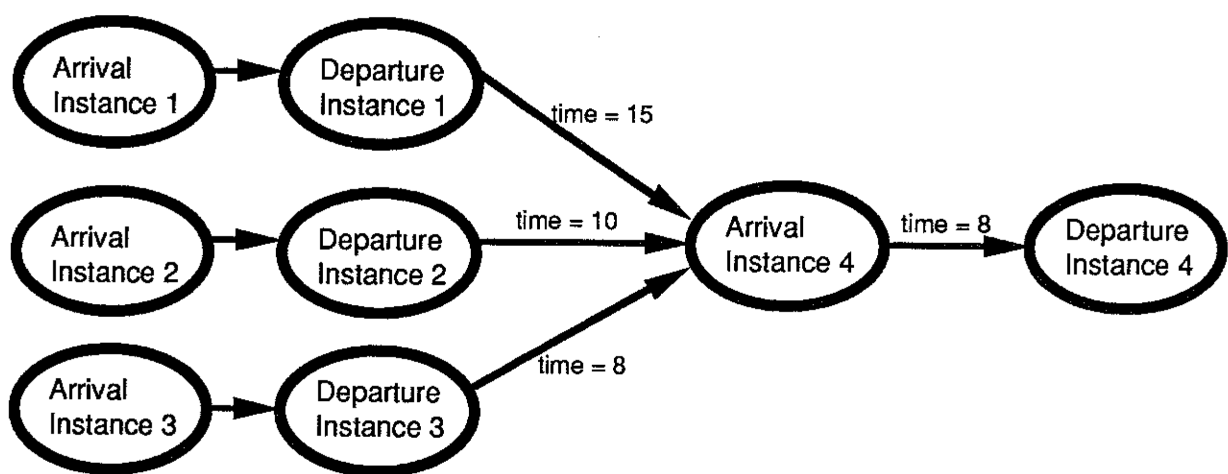

Figure 3: Dependency Graph of Event Instances

The event instance at the tail of an arrow schedules the event instance at the head of the arrow. For example, arrival instance 4 is scheduled to occur at times 8, 10, and 15. A conservative method will not execute arrival instance 4 at time 8 until departure instances 1 and 2 have scheduled arrival instances at times greater than or equal to 8.

Creating the list of input arcs can be done transparently to the simulation programmer. Because he must already include the information of output paths for the various scheduling methods, the code to add to an output arc can also add to the input arc of the given output event. This list of input arcs is not necessary for sequential simulation execution, because event instances are strictly executed in order of increasing time.

iii) SUMMARY OF DEPENDENCIES

Three conditions must be satisfied before an event instance E can be executed. First, all event instances that might schedule or cancel E must have already been executed. Second, all event instances that might modify states accessed by E must have already been executed. Third, no event instance that accesses states accessed by E can be concurrently executed. See Cota and Sargent [1989] for an algorithm to identify these dependencies during simulation execution.

B. DEADLOCK AVOIDANCE AND LOOKAHEAD

The possibility of deadlock exists in conservative parallel simulation methods. Since no event will proceed until all possible inputs to it have been completed, the possibility exists that two or more processors are waiting for output from the other(s). To avoid this situation, the simulation programmer must add code to each event to calculate a lower bound on the time at which it could schedule an event the next time it is executed. For example, an event in a queueing network model could pre-sample the service time distribution. The simulation interface provides a function *UpdateLowerBound* to inform the parallel simulation protocol of this bound. Use of the function is sometimes required to avoid deadlock in conservative protocols (e.g. to generate null messages), and usually will decrease simulation running time.

C. ABILITY TO SAVE STATES

Optimistic parallel simulation methods require state information to be stored to allow for rollbacks and reexecution of event instances (see section II B). The separation of events and states as explained in section IV A lends itself well to this requirement. Only the information that varies upon execution of an event instance is contained in a state, thus information that does not change is not repeatedly stored with the numerous savings of state required by the optimistic implementation.

The function *SetFrequency* allows the simulation programmer to alter how often optimistic simulation protocols save a copy of states to use for rollbacks. This frequency can have significant effects on simulation performance.

A state may not always be affected when an event instance that can influence it is executed. The function *Clean* may be used by the simulation programmer to indicate that that state has not been altered and need not be stored. If *Clean* is not called, the fields in the state are assumed to have been changed, and they will be restored.

D. SUMMARY OF DIFFERENCES BETWEEN USING THE SEQUENTIAL INTERFACE AND THE PARALLEL INTERFACE

Parts A and B of the Appendix show code written in C++ [Stroustrup 1986] that represents the sequential and parallel interfaces, and the code that a user would write to perform a simple simulation using the interfaces.

The example illustrates an unexpected result: there is little difference in the functions offered by the two interfaces. Everything offered by the sequential interface is offered by the parallel interface, and the only extra functions that a user of the parallel interface needs are *Clean, UpdateLowerBound,* and *SetFrequency* that were mentioned earlier in this section. However, a significant difference in the structure of the two programs is noticed, chiefly due to the separation of *events* and *states*. The parallel interface provides additional classes EVENT and STATE, from which all user structures can be derived. Additional data structures present

in class EVENT include information to identify input arcs to an event instance, and information regarding the states the event instance can affect.

Refer to the Appendix part C for a summary of specific C++ differences between the sequential and the parallel interfaces.

V. CONCLUSION

This paper begins to address the topic of utilizing particular world views to perform a simulation in parallel, concentrating on the event scheduling approach. To accomplish this, a programming interface to perform a sequential simulation using the event scheduling world view has been developed and described, so that the changes needed to accomplish the simulation in parallel could be more easily identified.

To perform the simulation in parallel, the user must identify state dependencies by separating the events from the state(s) they affect. By identifying event-state dependencies, the parallel implementation can avoid scheduling conflicting event instances to occur concurrently. This separation of events and states lends itself well to statistics collection, since information needed for the statistic can be held in a separate state. Additional interface functions *Clean* and *SetFrequency* are used by optimistic parallel simulation methods to store states, and the function *UpdateLowerBound* is needed by conservative parallel simulation methods to avoid deadlocks.

Future research will deal with other world views, and identify what is necessary to perform parallel simulations using those world views. Immediate plans include addressing the *process oriented* world view. The feasibility of using a structured approach to development by layering the *process oriented* interface onto the *event scheduling* interface will be explored.

Acknowledgements

The authors wish to thank the members of the Simulation Research Group at Virginia Tech for their input regarding this paper.

References

Abrams, M. (1989), "A Common Programming Structure for Bryant-Chandy-Misra, Time-Warp, and Sequential Simulators," In *Proceedings of the 1989 Winter Simulation Conference* (Washington D.C., Dec.), IEEE, Piscataway, N.J., pp 661-670.

Baezner, D. , G. Lamow, and B.W. Unger (1990), "Sim++: The Transition to Distributed Simulation," In *Distributed Simulation* (San Diego, Calif., Jan.), Soc. for Comp. Sim., San Diego, Calif., pp 211-218.

Bagrodia, R.L. and W. Liao (1990), "Maisie: A Language and Optimizing Environment for Distributed Simulation, " In *Distributed Simulation* (San Diego, Calif., Jan.), Soc. for Comp. Sim., San Diego, Calif., pp 205-210.

Balci, O. (1988), "The Implementation of Four Conceptual Frameworks for Simulation Modeling in High-Level Languages," In *Proceedings of the 1988 Winter Simulation Conference* (San Diego, Calif., Dec.), IEEE, Piscataway, N.J., pp 287-295.

Bryan, O.F. Jr. (1989), "ModSim II: An Object Oriented Simulation Language for Sequential and Parallel Processors," In *Proceedings of the 1989 Winter Simulation Conference* (Washington, D.C., Dec.), IEEE, Piscataway, N.J., pp 172-179.

Bryant, R.E. (1977), "Simulation of Packet Communication Architecture Computer Systems," Tech Rep. MIT,LCS,TR-188, M.I.T., Cambridge, MA.

Chandy, K.M., V. Holmes, and J. Misra (1979), "Distributed Simulation of Networks," *Computer Networks 3,* pp 105-113.

Cota B.A. and R.G. Sargent (1989), "An Algorithm for Parallel Discrete Event Simulation Using Common Memory," *Simuletter, 20 (1),* pp 23-31.

Derrick, E.J., O. Balci, R.E. Nance (1989), "A Comparison of Selected Conceptual Frameworks for Simulation Modeling," In *Proceedings of the 1989 Winter Simulation Conference* (Washington D.C., Dec.), IEEE, Piscataway, N.J., pp 711-718.

Fujimoto, R.M. (1989), "Parallel Discrete Event Simulation," In *Proceedings of the 1989 Winter Simulation Conference* (Washington D.C., Dec.), IEEE, Piscataway, N.J., pp 19-28.

Heidelberger, P. (1986), "Statistical Analysis of Parallel Simulations," In *Proceedings of the 1989 Winter Simulation Conference* (Washington D.C., Dec.), IEEE, Piscataway, N.J., pp 290-295.

Jefferson, D. and H. Sowizral (1985), "Fast Concurrent Simulation Using the Time Warp Mechanism," In *Distributed Simulation* (San Diego, Calif., Jan.), Soc. for Comp. Sim., San Diego, Calif., pp 63-69.

Jefferson, D. (1987), "Distributed Simulation and the Time Warp Operating System," In *Proceedings of the 11th ACM Symposium on Operating System Principles* (Austin, Tex., Nov.) ACM, New York, pp 77-93.

Lomow, B. and D. Baezner (1989), "A Tutorial Introduction to Object-Oriented Simulation and Sim++," In *Proceedings of the 1989 Winter Simulation Conference* (Washington D.C., Dec.), IEEE, Piscataway, N.J., pp 140-146.

Nance, R.E. (1981), "The Time and State Relationships in Simulation Modeling," *Communications of the ACM, 24,* pp 173-179.

Pawlikowski, K. (1990), "Steady-State Simulation of Queueing Processes: A Survey of Problems and Solutions," *ACM Computing Surveys, 22 (2),* pp 123-170.

Preiss, B.R. (1989), "The Yaddes Distributed Discrete Event Simulation Specification Language and Execution Environments," In *Distributed Simulation* (San Diego, Calif., Jan.), Soc. for Comp. Sim., San Diego, Calif., pp 139-144.

Schwetman, H. (1986), "CSim: A C-Based, Process-Oriented Simulation Language," In *Proceedings of the 1986 Winter Simulation Conference* (Washington D.C., Dec.), IEEE, Piscataway, N.J., pp 387-396.

Stroustrup, B. (1986), *The C++ Programming Language.* Addison Wesley, Reading, MA.

West, J. and A. Mullarney (1988), "ModSim: A Language for Distributed Simulation," In *Distributed Simulation* (San Diego, Calif., Jan.), Soc. for Comp. Sim., San Diego, Calif., pp 155-159.

APPENDIX : IMPLEMENTATION EXAMPLE USING C++

This appendix gives examples of code written to perform event scheduling simulations using the interfaces developed. The code is written in C++. The object oriented principles of data abstraction and inheritance provide structure and flexibility of design well suited to interface development.

The example considered is a simple queueing network of two first come first served queues in tandem. The first job in the queue is considered to be in service by the processor. When a job departs the first queue it immediately enters the second queue. A statistic of the average number of jobs in the system is calculated.

This example consists of two event types, and two instances of each event: arriving at the first queue, departing from the first queue, arriving at the second queue, and departing from the second queue. How this event scheduling simulation would be accomplished using the sequential interface is shown in part A of this section, the corresponding parallel example is shown in part B, and an investigation of the specific differences between the two is in part C.

The random number generators in the code are based on those in the GNU g++ 1.37 library, with the function "fetch()" used to return the next random number from the stream.

A. EXAMPLE USING SEQUENTIAL INTERFACE

i) STRUCTURE PROVIDED BY SEQUENTIAL INTERFACE
See Section III for English description of functions.

```
Class SIMEXECUTOR
{ public:
        SIMEXECUTOR ();                  // void constructor to create instance of class
        eventid Schedule ( double aTime, INSTANCELIST* aArcList, functptr aMembFunct);
                                         // the 1st param is when the event is to take place,
                                         // INSTANCELIST is a linked list of ptrs to event instances,
                                         //    functptr points to a member function of a class
        eventid RandomSchedule ( double aTime, INSTANCELIST* aArcList, functptr aMembFunct);
                                         // both functions return 'eventid', which is a
                                         //    pointer to an event
        eventid MultiSchedule ( double aTime, INSTANCELIST* aArcList, functptr aMembFunct );
        bool Cancel (eventid aEvent);
        bool Cancel (eventid aEvent, double aTime, INSTANCELIST* aArcList, functptr aMembFunct);
                                         // both Cancels return false if the event to cancel
                                         //   was not on the list of events. The second Cancel
                                         //   returns all the fields of the cancelled event.
        bool Identify (eventid aEvent, double aTime, INSTANCELIST* aArcList,  functptr aMembFunct);
                                         // identifies the fields of an event w/o cancelling it
        double GetClock ();              // returns the present value of simulation clock
        double ResetClock ();            // sets sim clock to 0 for new replication
        bool ListEmpty ();               // checks to see if event list is empty
        bool EventOnList (eventid aEvent); // sees if a particular event is on the event list
        bool Done ();                    // tests to see if simulation replication should terminate
        bool AddToOutArc (int aArcList, nodeptr aNodeList, nodeptr aToAdd);
                                         // the first param indicates which instance list to add to,
                                         //   nodeptr is a ptr to a class (e.g. FCFS)
        void ExecuteReplicativeSim (int Repititions, int SteadyState, int JobsToDo, funptr First);
                                         // this will execute a regenerative simulation
                                         //   funptr is a ptr to a user written function
        void ExecuteBatchSim (int Repititions, int SteadyState, int JobsToDo, funptr First);
                                         // this will execute a batch simulation
};
```

ii) USER WRITTEN STRUCTURES TO PERFORM NETWORK SIMULATION

--Class Declarations: These would appear in C++ header files

```
class AvgJobsHist                          // this class used to calc  the statistics histogram
{ private:                                 // variables used to calc histogram
        double TimeSpan, BottomSpan, LastValue, Area, TotalArea;
  public: int NumInSystem;                 // the jobs present in the system at any time
        AvgJobsHist ();                    // void constructor
        void Update (double Time);         // updates the histogram when Num changes
        double Mean (double Time);         // calcs the mean # jobs over that period of time
        void Reset ();                     // clears variables for new sim replication
};

class FCFS                                 // this class used to represent a node in the network
{ private:
        int  Id;                           // unique id for node, useful in tracing
        int  InQ;                          // number of jobs present in the queue
        MLCG*  MultGen;                    // multiplicative linear congruential generator
        NegativeExpntl* StRandgen;         // random number generator to produce numbers
                                           //    with a negative exponential distribution for the
                                           //    service time of a job at this node
        NegativeExpntl* InterArrTime;      // to produce the distr for inter arrival time of jobs
        int NumOutArcs;                    // the number of different OutArcs for this node
        INSTANCELIST** OutArcs;            // array  of ptrs to lists of possible Output Arcs
        int*  NumEventsOutArc;             // array of integers of # of events in each OutArc
 public:
        FCFS (int aId, int aNumOutArcs);   // constructor with unique Id parameter  & number
                                           //    of output arcs this node will have
        bool Arrival ();                   // function to represent the event  arriving at queue
        bool Departure ();                 // function to represent the event departing queue
                                           //   termination of sim replication occurs if any function
                                           //   returns true
};
```

-- FCFS Function Code: these would appear in C++ code file

```
FCFS::FCFS (int aId, int aNumOutArcs)      // constructor called to create instance of node
{
   Id = aId;                               // set identification variables
   NumOutArcs = aNumOutArcs;
   InQ = 0;                                // set number present in queue to zero
   MultGen = new MLCG ( Seed1, Seed2 );    // creates gen with two random seeds
   StRandgen = new NegativeExpntl ( MeanServiceTimeForNode,   MultGen);
   InterArrTime = new NegativeExpntl ( MeanInterArrivalTimeOf Jobs,  MultGen);
                                           // Seeds and Means defined as constants
   OutArcs  = new DATLIST[NumOutArcs];
                                           // creates array to hold ptrs to linked lists
                                           //    for each output arc
   NumEventsOutArc = new int[NumOutArcs];
                                           // creates array of integers for # nodes in each arc
   for (int i = 0; i < NumOutArcs; ++i)
   {                                       // initializes OutArc data
      OutArcs[i] = new INSTANCELIST();
       NumEventsOutArc[i] = 0;
   }
}

bool FCFS::Arrival ()                      // to perform when job arrives at queue
{
   ++InQ;                                  // increment number of jobs at the queue
```

```
    double PresClock = SimRun->GetClock();
                                                // find present value of simulation clock
                                                //  Simrun is an instance of SimExecutor
    if  (InQ == 1)                              // if only one job at node begin processing
    {
       double ServTime = StRandgen->fetch();
                                                // find random service time for this job
       eventid JobInProcess = SimRun->Schedule (PresClock + ServTime,  OutArcs[0],  &FCFS::Departure);
                                                // schedule departure of this job
    }

    if (Id == FirstNode)                        // if this is the first node in the network
       {                                        //   update statistics collection and create new job
                                                //   FirstNode defined as a constant
        ++AvgJobs->NumInSystem;                 // AvgJobs is instance of AvgJobsHist
        AvgJobs->Update ( PresClock);

        double Iat = InterArrTime->fetch();     // find random inter arrival time for next job
        eventid NewSched = SimRun->Schedule (PresClock + Iat, OutArcs[0],  &FCFS::Arrival);
                                                // schedule arrival of that event
    }
    return false;                               // this event will never end the simulation replication
};

bool FCFS::Departure ()
                                                // this fnt  represents the event of a job leaving the q
{
    -- InQ;                                     // decrement number of jobs present in queue
    double PresClock = SimRun->GetClock();

    if (InQ > 0)                                // there are jobs waiting to be processed
    {
       double ServTime = StRandgen->fetch();
                                                // find random service time for this job
       eventid JobInProc = SimRun->Schedule (PresClock + ServTime, OutArc[0], &FCFS::Departure);
                                                // schedule next departure
    }
    if (Id == LastNode)                         // if this is last node, update stats & delete job data
    {                                           //     LastNode defined as constant
       --AvgJobs->NumInSystem;
       AvgJobs->Update (PresClock);
    }
    else                                        // this is not the last node so schedule arrvl at next
       eventid Ret = SimRun->Schedule ( PresClock, OutArc[1],  &FCFS::Arrival);
    return SimRun->Done();                      // determines if should end the simulation
                                                //    Done will return true if # jobs completed is equal to
                                                //     termination #  (JobsToDo) given in parameter
                                                //     of ExecuteBatchSim or ExecuteReplicativeSim
}
```

iii) DRIVER PROGRAM

```
void Initialize ()                              // to create all classes needed and initialize outarcs
{
   SIMEXECUTOR* SimRun = new SIMEXECUTOR();
                                                // to use interface functions
    FCFS*  Fcfs1  = new FCFS ( FirstNode, 2 );
    FCFS*  Fcfs2  = new FCFS ( LastNode,  1 );
                                                // instances of FCFS class with Id = a constant and
                                                //  fcfs1  will utilize 2 output arcs(one to itself and
                                                //  one to fcfs2)  and fcfs2 will use 1 (to schedule
                                                //  departures  to itself-  similar to using "this")
    SimRun->AddToOutArc ( 0,  Fcfs1, Fcfs1);
    SimRun->AddToOutArc ( 1, Fcfs1, Fcfs2);
```

```
    SimRun->AddToOutArc ( 0, Fcfs2, Fcfs2);
}

void FirstEvent()                          // to start off each new simulation replication
{
   INSTANCELIST* TempList = new INSTANCELIST(Fcfs1);
                                           // since schedule requires a list as a parameter
    eventid First = SimRun->Schedule (0.0, TempList, &FCFS::Arrival);
}                                          // schedule first  event to take place

main()
{
    Initialize();
    ExecuteReplicativeSim ( 20, 3000, 15000, &FirstEvent);
                                           // this runs a regenerative sim of 20 repititions
                                           //   steadystate reached after 3000 jobs completed,
}                                          //   steadystate consists of 15000 jobs.
```

B. EXAMPLE USING PARALLEL INTERFACE

i) STRUCTURES PROVIDED BY PARALLEL INTERFACE

See section IV for English descriptions.

```
class SIMEXECUTOR   {...};                // this will remain the same as in the Sequential
                                          //   Interface with the following exceptions:
                                          //   - functptr aMembFunct is no longer needed
                                          //       in Schedule, Cancel, and Identify because
                                          //       each member function performed now has
                                          //       the same name "EventBody"
                                          //   - AddToOutArc is moved to class EVENT
class STATE                               // base class from which all states are derived
{ private:
        int Id;                           //  unique identification for instance of class
  public:
        STATE (int aId);                  // one parameter constructor
};

class EVENT                               // base class from which all events are derived
{ private:
        int Id;                           // unique identification for instance of class
        INSTANCELIST* InArcs;             // list of all possible input arcs
        int NumEventsInArc;               // number of input arcs on the list
        int NumOutArcs;                   // number of output arcs
        int Frequency = 0;                // used in opt methods, how often states saved
  public:
                                          // since these vary with each derived event, must be
                                          //     initialized there
        INSTANCELIST** OutArcs;           // array of ptrs to lists of output arcs
        int*  NumEventsOutArc;            // number of output arcs in each list
        STATE** CRWS;                     // Current Read Write State:
                                          //   array of ptrs to each state this event effects
        int*  CRWSize;                    // array of ptrs to the corresponding size of  states

        EVENT (int aId, int aNumOutArcs);
                                          // constructor
        virtual bool EventBody ()  {}
                                          // this must be redefined by all derived classes
        void AddToOutArc (int aOutArcList, EVENT* aArcEvent);
        void AddToInArc (EVENT* aArcEvent);
                                          // this is called by AddToOutArc, never needs to
                                          //   be called by the user
        void UpdateLowerBound ( double aTime);
        void Clean ( int aState );
        void SetFrequency (int aFreq );
```

```
};
```

ii) USER WRITTEN STRUCTURES TO PERFORM NETWORK SIMULATION
--Class Declarations: These would appear in C++ header files

STATES:

```
class FCFSQueueState : public STATE       // class to contain all variables
                                          //   corresponding to the state of a FCFS queue
{ public:
        int InQ;                          // number of events present in queue
        MLCG* MultGen;                    // multiplicative linear congruential generator
        NegativeExpntl* StRandgen;        // random # generator for service times
        NegativeExpntl* InterArrTime;     // random # generator for interarrival times
        double NextServTime;              // used for lookahead and deadlock avoidance
        double NextArrvTime;              // used for lookahead and deadlock avoidance

        FCFSQueueState (int aId);         // constructor, initializes all fields in state
        void Reset ();                    // resets state for new sim replication
};

class AvgJobsState : public STATE         // class to contain all variables corresponding
                                          //   to the state of a statistics histogram
{ public:
        int NumInSystem;                  // jobs in system at any time
        double TimeSpan, BottomSpan, LastValue, Area, TotalArea;

        AvgJobsState ( int aId);          // one parameter constructor
        void Update (double aTime);       // updates area of histogram
        double Mean (double aTime);       // calcs the avg # jobs over aTime
        void Reset();                     // resets state for new sim replication
};
```

EVENTS:

```
class FCFSArrival : public EVENT          // class to implement arrival at FCFS queue
{public:
        FCFSArrival (int aId, int NumOutArcs, STATE* aQState,    STATE* aAvgState);
                                          // four parameter constructor, shows that
                                          //   this event can possible affect two states
        bool EventBody ();
                                          // what is performed when event occurs
};

class FCFSDeparture : public EVENT        // class to implement departure from FCFS q
{ public:
        FCFSDeparture (int aId, STATE* aQState, STATE* aAvgState);
        bool EventBody ();
};
```

-- FCFS Function Code: these would appear in C++ code file

```
FCFSArrival::FCFSArrival(int aId, int NumOutArcs, STATE* aQState,
                         STATE* aAvgState ) : EVENT ( aId,NumOutArcs)
                                          // constructor, passes fields to base class EVENT
{
   CRWS = new STATE* [NumStates];         // array of ptrs to each state this event effects
   CRWS[QState] = aQState;                // QState constant defined to 0, AvgState 1
   CRWS[AvgState] = aAvgState;

   CRWSize = new int[NumStates];          // array of integers for size of each state
   CRWSize[QState] = sizeof (FCFSQueueState);
   CRWSize[AvgState] = sizeof (AvgJobsState);
```

```
  OutArcs = new INSTANCELIST*[NumOutArcs];
  NumEventsOutArc = new int[NumOutArcs];
                                          // intialize all output arcs and number in each
  for (int i = 0; i < NumOutArcs; ++i)
  {
    OutArcs[i] = new INSTANCELIST();
    NumEventsOutArc[i] = 0;
  }
}

bool FCFSArrival::EventBody ()
{                                         // function for event occurance
  ++ ((FCFSQueueState*) CRWS[QState]) -> InQ;
                                          // increments # jobs in queue.  Must convert
                                          //    STATE* to FCFSQueueState* to access InQ
  timetype PresClock = SimRun->GetClock();
                                          // find current simulation time
  if ( (( FCFSQueueState*) CRWS[QState] ) -> InQ == 1)
  {                                       // only one job in queue, begin processing
    double ScheduleTime =  ((FCFSQueueState*) CRWS[QState])-> NextServTime  + PresClock;
                                          // determine when event instance should occur
                                          //   by using previously stored random service time
    eventid JobInProc = SimRun->Schedule (ScheduleTime, OutArc[0]);
                                          // schedule departure of this job
    ((FCFSQueueState*) CRWS[QState])-> NextServTime =
                        ((FCFSQueueState*) CRWS[QState]) ->StRandgen->fetch() ;
                                          // find random service time for next instance
    UpdateLowerBound (  ((FCFSQueueState*) CRWS[QState])-> NextServTime  + PresClock );
                                          // update time for deadlock avoidance and lookahead
  }

  if (Id == FirstNode)                    // if this is the first node in network
  {
     ++ ((AvgJobsState* ) CRWS[AvgJobs] ) -> NumInSystem;
                                          // increment jobs in system counter
     ((AvgJobsState*) CRWS[AvgJobs] ) -> Update (PresClock);
                                          // update   statistics histogram
    double ScheduleTime = ((FCFSQueueState*) CRWS[QState])->NextArrvTime  + PresClock;
                                          // determine when event instance should occur
                                          //   by using previously stored random service time
    eventid NewSched = SimRun->Schedule (ScheduleTime,  OutArc[1]);
    ((FCFSQueueState*) CRWS[QState])->NextArrvTime =
                        ((FCFSQueueState*) CRWS[QState]) -> InterArrTime->fetch();
                                          // find inter arrival time for next job
    UpdateLowerBound (  ((FCFSQueueState*) CRWS[QState])-> NextArrvTime  + PresClock );
                                          // update time for deadlock avoidance and lookahead
  }
  else                                    // avg jobs state not altered
      Clean (AvgState);

  return false;                           // this event will never cause sim replication to end
}

FCFSDeparture::FCFSDeparture (int aId,  int NumOutArcs, STATE* aQState,
                              STATE* aAvgState  )  : EVENT ( aId,NumOutArcs)
{...}                                     // same code as FCFSArrival constructor

FCFSDeparture:: EventBody ()
{
  -- ((FCFSQueueState*) CRWS[QState]) ->InQ;
                                          // decrement number of jobs in queue
  double PresClock = SimRun->GetClock();

  if  (  ((FCFSQueueState*) CRWS[QState] ) -> InQ   > 0 )
  {                                       // there are jobs waiting to be processed
```

```
        double ScheduleTime =  ((FCFSQueueState*) CRWS[QState])-> NextServTime  + PresClock;
                                              // determine when event instance should occur
                                              //   by using previously stored random service time
        eventid JobInProc = SimRun->Schedule (ScheduleTime, OutArc[0]);
                                              // schedule departure of this job
        ((FCFSQueueState*) CRWS[QState])-> NextServTime =
                                   ((FCFSQueueState*) CRWS[QState]) ->StRandgen->fetch() ;
                                              // find random service time for next instance
        UpdateLowerBound (  ((FCFSQueueState*) CRWS[QState])-> NextServTime  + PresClock );
                                              // update time for deadlock avoidance and lookahead
      }
      if ( Id == LastNode)                    // if this is last node in network, update stats
      {
        -- ((AvgJobsState*) CRWS[AvgJobs])->NumInSystem;
        ((AvgJobsState*) CRWS[AvgJobs]) -> Update (PresClock);
      }
      else                                    // this is not last node so schedule to next one
      {
        eventid Ret = SimRun->Schedule (PresClock, OutArc[1]);
        Clean (AvgState);                     // avg jobs state not altered
      }
      return SimRun->Done();                  // test to see if sim replication should end
    }
```

iii) DRIVER PROGRAM

```
    void Initialize()
    {
      SIMEXECUTOR* SimRun = new SIMEXECUTOR();
                                              // allows use of interface functions
      STATE*  Fcfsq1 = new FCFSQueueState ( 1 );
      STATE*  Fcfsq2 = new FCFSQueueState ( 2 );
                                              // a state for each queue in network
      STATE* AvgJobsSt = new AvgJobsState ( 3 );
                                              // a state to calc avg # of jobs in system
      EVENT* FcfsArrvl1  = new FCFSArrival ( 1, 2, Fcfsq1, AvgJobsSt);
                                              // event arrival to fcfs queue #1,
                                              //    will have 2 output arcs, and can effect the
                                              //     two states listed
      EVENT* FcfsDepart1 = new FCFSDeparture (1, 2, Fcfsq1, AvgJobsSt);
      EVENT* FcfsArrvl2  = new FCFSArrival (2, 1, Fcfsq2, AvgJobsSt);
      EVENT* FcfsDepart2 = new FCFSDeparture (2, 1, Fcfsq2, AvgJobsSt);
                                              // set output paths
      FcfsArrvl1   ->AddToOutArc ( 0,  FcfsArrvl1);
                                              // allows schedule  to itself, similar to using "this"
      FcfsArrvl1   ->AddToOutArc ( 1, FcfsDepart1);
                                              // to schedule completions
      FcfsDepart1 -> AddToOutArc (0, FcfsDepart1);
      FcfsDepart1 -> AddToOutArc (1, FcfsArrvl2);
      FcfsArrvl2   -> AddToOutArc (0, FcfsArrvl2);
      FcfsDepart2 -> AddToOutArc (1, FcfsDepart2);
    }
    void FirstEvent()                         // to start off each new simulation replication
    {
      INSTANCELIST* TempList = new INSTANCELIST (FcfsArrvl1);
                                              // create a list, since schedule param requires it
        eventid First = SimRun->Schedule (0.0, TempList);
    }
    main()
    {
      Initialize();
      ExecuteReplicativeSim (20, 3000, 15000, &FirstEvent);
                                              // this runs a regenerative sim of 20 repititions
                                              //   steadystate reached after 3000 jobs completed,
    }                                         //   steadystate consists of 15000 jobs.
```

C. DIFFERENCES BETWEEN THE SEQUENTIAL AND THE PARALLEL INTERFACES, AND A FEW EXPLANATIONS

A significant difference exists in how the simulation is structured as a set of classes. In the sequential case, it was convenient to combine event types *Arrival* and *Departure* into a single class because together they correspond to a queue, a logical component of the system being simulated. This approach was not suitable for parallel implementation, due to the problem of state dependencies. However, the parallel implementation needs to distinguish which events use which states. As Figure 2 illustrates, different instances of event types *Arrival* and *Departure* use different states. Thus, the separation of each event into a separate class was considered necessary for the parallel implementation, because there can exist an arbitrary graph of event-state relationships.

The differences between the two interfaces are not extensive. The SIMEXECUTOR class which performs most of the interface functions has only two small differences:

1) the pointer to a member function needed by the sequential interface to determine which function of the class to perform is unnecessary in the parallel implementation, for each member function has the same name "EventBody", and
2) the function "AddToOutArc" is removed from SIMEXECUTOR for the parallel implementation; having the function in EVENT instead allows easier identification of which event the arc is to be added to.

The extra classes offered by the parallel interface are:

1) STATE - a base class that all states can be derived from; in this example contains only an identification field and a constructor.
2) EVENT - a base class that all events can be derived from; contains fields for identification, input arcs, output arcs, frequency of state saving, and pointers to states. Member functions include a constructor, functions to manage input and output arcs, avoid deadlocks, indicate state changes, and alter frequency of state saving.

The extra data structures contained in EVENT needed in the parallel interface are:

1) INSTANCELIST* InArc and int NumEventsInArc. InArc represents a linked list of all event instances that can possibly effect the instance that this data structure belongs to, and NumEventsInArc is a counter of how many input arcs exist. This is needed by conservative methods to determine time dependencies.
2) STATE** CRWS and int* CRWSize. CRWS (Current Read Write State) is an array of pointers to each state this event instance can affect. CRWSize is an array of the corresponding sizes of each state. These are needed by optimistic methods that require the state be periodically saved during simulation execution.

Parts of the interfaces, as well as utility functions developed, have been omitted due to length considerations. Work has been done on the subjects of prioritizing, typing of events, other types of scheduling, other statistics collections, termination, and output. For additional information and/or access to C++ code, please contact the authors.