Proceedings of the 1988 Winter Simulation Conference
M. Abrams, P. Haigh, and J. Comfort (eds.)

Distributed simulation

Brian W. Unger
Jade Simulations International
1833 Crowchild Trail NW
Calgary, Alberta, Canada T2M 4S7
(403)282-5711
unger@calgary.CDN

ABSTRACT

Emerging multicomputer architectures provide a platform upon which to realize substantial performance improvements in simulation. Parallel execution is the means to these improvements, yet hurdles remain: the synchronization of simulation time across multicomputer nodes and whether sufficient model parallelism can be identified to enable substantial speedup. Overcoming these obstacles will make speeding up the execution of simulations by several orders of magnitude practical.

Simulation environments of the future must build on traditional object-oriented methods and the new logic program paradigms in a way which does not hide the parallelism inherent in models. Distributed operating system kernals and language run-time systems must then be able to exploit this parallelism to support concurrent execution of simulations on multicomputers. Finally, these simulation kernels must support this parallelism transparently, that is, without modification at the model source code levels. Without this transparency, models will have to be continually re-written when moving from model develpment, testing, and validation on sequential hardware to parallel hardware for simulation experiments.

INTRODUCTION

Parallelism in the execution of simulations may make substantial performance improvement possible. The emerging multicomputer architectures that consist of hundreds to thousands of powerful node computers have tremendous potential for speeding up simulations. Faster simulations will expand the kinds of problems to which simulation can be applied. The ability to speed up execution by several orders of magnitude would make interactive, faster than real time, simulation possible for problems in urban and transportation planning, for example, which have not previously been feasible.
However, three issues must be resolved before substantial speedup via parallelism can be achieved with real simulation problems. The first involves the synchronisation of simulation time in the concurrent, distributed execution of system model components. The second involves the development of models in a way which identifies potential parallism. The third issue is whether sufficient inherent parallelism exists within system models to make distributed, concurrent execution practical.

In a multicomputer that consists of large numbers of computing nodes, it is difficult to provide a globally available clock. This implies asynchronous execution on different nodes and thus a requirement to coordinate the advance of simulation time across nodes. Recently, a new "optimistic" synchronisation approach called "Virtual Time" has been described by Jefferson that may make greater parallelism possible for models of arbitrary structure [Jefferson 85]. Past approaches have been based on "conservative" mechanisms which are easier to implement but which either restrict model structure or the extent of parallelism achievable.

The second model development issue requires modelling techniques and programming language facilities that do not unnecessarily constrain parallelism. For example, customers arriving at a service facility may be served in order of time of arrival, i.e., first arrived, first served. However, customers need not arrive in this order in real time during a simulation, e.g., C1 and C2 may arrive at respective simulation times t1<t2 thus requiring that C1 be served before C2 but that they may actually arrive at the service facility in either order.

The third issue, whether their exists sufficient inherent parallelism within simulation models to enable substantial speedup, cannot be resolved without further experimental research. Even when there is substantial parallelism in the system being modelled, it is not clear that speedup via parallelism is possible during model execution. A key unknown is the ratio of computation overhead required for the synchronisation of simulation time to the computation required to mimic model behavior.

This paper outlines a process view of simulation built on a message passing model. A simple example is presented to illustrate how a model can be specified using this view without regard for whether the simulation will be executed as a sequential program or as distributed parallel, program. This enables

simulations to be written in a away which is independant of the underlying physical computing architecture. It is also anticipated that greater speedup will be possible than for alternative methods.

A COMMUNICATING PROCESS VIEW OF SIMULATION

Modelling involves decomposing the system under investigation into a set of interacting components. Then each of these components can be considered a system and thus further decomposed into interacting components. Ultimately, the components at the lowest level are represented as modules in a simulation program. One widely used approach involves representing operating component models by "processes".

The Process View

The process view of modelling and simulation originated with the GPSS [Gordon 75] and Simula [Dahl &Nygaard 66] languages. This "object-oriented" approach was further elaborated in Demos [Birtwistle 85] and in [Franta 77]. Currently the process view, or equivalently the object-oriented approach, has been incorporated in some form in most widely used simulation languages,e.g., Simscript (see [Kruetzer 86] for a good discussion of this approach to modelling and related language issues).

The basic idea is to decompose the system being modelled into a set of components while identifying subsets of similar components that have common characteristics and operations. Similar components form the basis for defining different component "types". Here, instances of component types will be called "objects" and the component types will be referred to as "object types", or just types. The operation, or execution, of an object will be defined as a "process".

Objects & Processes

An object type is nicely represented as an "abstract type" within a programming language which supports this concept. Part of the implementation of a type is a set of attributes which all objects of that type have in common, i.e., variables whose values define characteristics of specific instances of that type and procedures which define common operations. For example, a type "car" may have an attribute "color", and a specific instance of car might have blue as the value of color; a car may also have a procedure fill_with_gas which when invoked on behalf of a specific car would increase the value of a variable called fuel_available.

DEFINITION - *attributes* (tp)- the attributes of type tp are a set of variables and procedures that all type tp objects have in common.

The progressive decomposition of components and the definition of types leads naturally to hierarchies of types and objects if each new type can be defined as an extension of at most one previously defined type. Then the new type can be said to inherit some or all of the parent type's attributes. Other inheritance rules are also possible, as when a new type is defined as a refinement of two or more previously defined types, e.g., as in Lisp Flavors [Kreutzer 86, pg 103]. The hierarchical approach to software design originated with Simula 67 and has since blossomed into the object-oriented programming and language methodologies of Smalltalk, C++ and Ada [Kreutzer 86].

A crucial element of the definition of an object type is a sequence of instructions that defines the behavior of all objects of that type. This sequence of instructions is essentially a main program, possibly including local procedure calls, which read and write attribute variables and possibly invoke operations associated with the corresponding type.

DEFINITION - *program*(tp) - a sequence of instructions: $\{I_i, i = 1, ..., n\}$ performed by all instances of object type tp where each instruction I_i may involve reading and possibly changing the values of variables in attributes(tp).

A program instruction may involve interactions with the object's external world. This can be accomplished by reading and writing the attribute variables, or invoking local procedure attributes, of different objects. It could also be accomplished via message passing.

An object type tp is defined by the sets attributes(tp) and by the program(tp). A number of specific instances of this type can be created, e.g., $object_1(tp)$, $object_2(tp)$, ..., $object_m(tp)$. The operation of a specific object is represented by the execution of its program which gives rise to a sequential "process".

DEFINITION - *process* (tp)- a process p of type tp is a sequence of tuples: $\langle state_i(p), I_i \rangle$, $i = 0,1,...,n$ where $state_i(p)$ is a set that contains a value for each of the variables in attributes(tp). Each $state_i(p)$ is generated by the execution of the ith instruction, I_i, defined in program(tp) as follows. The initial state, $state_0(p)$, is the set of initial values of variables in attributes(tp), $state_i(p)$ is the set of values for variables in

attributes(tp) that result from performing I_i on $state_{i-1}(p)$.

Saving the state of a process enables its execution to be suspended and later continued. For example, the first k instructions for process p could be executed producing $state_k(p)$. Then this state could be saved along with the value of k, another process executed, and at some later time $state_k(p)$ and the value of k could be restored and the k+1st through nth instruction of p's main program executed. Notice that k must be saved so that the next instruction to be executed, i.e., k+1, is remembered. This is exactly what happens in a multi-tasking operating system, such as Unix, when the execution of different user processes are interleaved in time.

The fact that process execution may have been temporarily suspended by saving its state, then executing some other process, and later restoring the saved state and continuing execution, is completely "transparent" to that process. That is, the successive process states and results of any computation are identical to what they would have been without suspension and subsequent continuation.

Simulation Time

A simulation always involves associating a time with the occurrance of object operations. Here this means that each process instruction is executed at some simulation time.

DEFINITION - *simulation_time* - a unique non-negative real variable whose value increases monotonically during a simulation. Two atomic operations, called *time* and *hold*, are defined on simulation_time which can be performed by any process in a simulation. A *time* call returns the current value of simulation_time to the calling process and *hold*(t) suspends the execution of the calling process until the value of simulation_time increases by t.

The *time* operation can be implemented as a simple read of the value of a globally available simulation_time variable. The *hold* operation is more complex. A call to *hold*(t) by process p at simulation_time s could be implemented by first saving the current state(p), determining which process to execute next, and insuring that the execution of p will be resumed when simulation_time becomes s+t. Later we will discribe an alternative implementation of hold.

Notice that a simulation_time can be associated with the execution of every object instruction. The simulation_time at which any given operation I_i is performed is defined by the value that would be returned by *time* if it were called just before execution of I_i. We can now define the concept of an event.

DEFINITION - *event*(p,t) - an event occurs at simulation_time t and at process p: $\{state_0(p), ..., state_{k-1}(p), state_k(p), ..., state_n(p)\}$ with program(tp): $\{I_j, j=1,...n\}$ if simulation_time<t when I_{k-1} is executed and simulation_time=t when instruction I_k is executed.

This definition implies that simulation_time remains constant between successive events, i.e., simulation_time can only advance upon the occurrance of an event. In general, there will be a sequence of state changes, $\{state_j(p), ..., state_k(p)\}$, associated with each event(p,t) where simulation_time =t when I_k is executed, and simulation_time is some constant value <t when I_j, ..., I_{k-1} are executed.

A Process View based on Message Passing

Interactions among system components must also be represented within objects and processes. This can be accomplished in several ways. Past approaches realize the exchange of information among processes by enabling program instructions to read and write global variables and to directly access the attributes of another object, e.g., as in Simula. Another possiblity is to implement all process interactions as the exchange of messages, i.e., object operations could also invoke message send and receive primitives. The latter approach is taken here.

Here we restrict all process interactions to calls to two message passing primities named "send" and "receive":

DEFINITION - send(p,t,m) - a send is an atomic operation that initiates the transfer of message m to process p at receive time t. If t' is the value of simulation_time when send is called and t" is the simulation_time when this call returns then t"=t'. Further, t must be >=t'. The value of t specifies the simulation_time at which process p must receive m.

Notice that send can be implemented as a synchronous, non-blocking procedure which is invoked with values specified for all three of its parameters. The value of p identifies a specific destination process, t defines the earliest simulation time that m can be acquired by p and the message m can be an arbitrary sequence of data values.

DEFINITION - receive(p,m) - a receive is an atomic operation which, when called by process q at simulation_time t', returns at simulation_time t" with values for p and m such that process p called send(q,t,m) with t'<=t"=t, where t is also less than or equal to the receive times of all other messages sent to q but not yet received by q.

This receive primitive is a synchronous, blocking procedure which is invoked without values for either of its two parameters. The procedure returns to its caller with message m which has been sent by some process p.

It can be shown that all process interactions can be accomplished via these two primitives. For example, one way to implement hold(t) is as a "send(self, time+t, null)" followed by "repeat receive(src, msg) until src=self". This implementation unfortunately throws away any messages that arrive during the hold time t. However, hold can also be implemented in a way which retains messages which arrive for the process during the hold. This latter implementation is assumed here which corresponds to the familiar meaning of the hold operation.

Discrete Event Simulation

A discrete event simulation can be defined as:

DEFINITION - *simulation* - a simulation on [0,T] is a set of processes $p_1, ..., p_2$ that are executed from simulation_time 0 up to simulation_time T which produces result R. Let the processes $p_1, ..., p_2$ be represented by:
$p1: \{state_0(p_1), I1_1, t1_0>, ..., <state_{n1}(p_1), I1_{n1}, t1_{n1}>\}$
...
$pn: \{<state_0(p_n), In_1, tn_0>, ..., <state_{nn}(p_n), In_{nn}, tn_{nn}>\}$
where $tj_{i-1}<T<=tj_i$ for j=1, ...n and i=n1, ..., nn
Then $R= \{state_{n1}(p_1), ..., state_{nn}(p_n)\}$.

Thus a simulation is defined by the set of final states of its constituent processes after their execution from simulation_time 0 up to T. Intermediate states may be different between two simulations but if their final states are equivalent, then the two simulations are equivalent. Cases where intermediate results are important can be represented by including these intermediate results within process attributes in a way that carries such results to the final state.

It is important to realize that the process view of simulation arose primarily because it is a natural way to develop models, not because of any underlying machine structure on which these models were to be executed. The communicating process view defined above is essentially equivalent to the original process view of modelling and thus retains this advantage. In fact, the execution of a sequential simulation that consists of a set of processes must be translated, or mapped, into a single sequential process before it can be executed. This is the purpose of the sequential simulation scheduling algortihm described in the following paragraphs.

SEQUENTIAL / PARALLEL EXECUTION OF SIMULATIONS

The following example is used to illustrate the process view for simulation models. After presenting this example, its execution as a sequential simulation, and then as a distributed simulation are described.

Health Care System Example

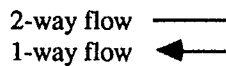
The governments of many developing countries provide health care for the majority of their populations through a multi-tiered system of services and referrals. The pyramid structure of the health system in each province of a country might include the following facilities: health posts, health centers, local hospitals, regional hospitals, and one provincial, university teaching hospital.

At each level in the system, certain services are provided and designated problem situations are referred to the next level. A simplified view of this system where people living in "villages" travel to and from health care "clinics" for services is shown in Figure 1. In this simplified example, people always first go to the clinic attached to their village. Most of them are assessed and treated at this nearest clinic and then return to their home village. Some of them, however, have problems which can't be treated at this local clinic and so are sent to the local clinic at the next level up in the hierarchy, e.g., from the local health post to the local health center, or local hospital to regional hospital.

Clinic & Village Objects

The objects in this model are specific clinics and villages. Clearly, we can identify two types of objects, the types *clinic* and *village*. Attributes of a village might include total population, frequency of different kinds of health problems, and lists of people currently recovering from different health problems or treatments. Attributes of a clinic might include distributions of assessment and treatment times.

The programs for these simple clinic and village object types are outlined in Figures 2 & 3. The execution of



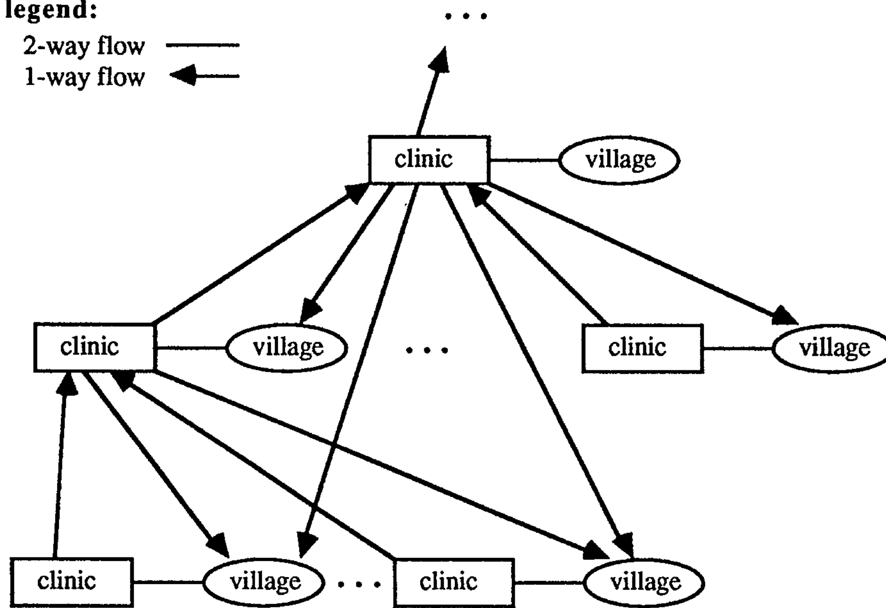


Figure 1 Structure of a Health Care System

The program for a clinic:	Comments:
1) receive(v_or_c, np)	: wait to receive the next person, np, either from the local village or a lower level clinic.
2) <assess problems for np, determine assessment time, at>	
3) hold(at)	: wait for at simulation time
units.	
4) if <treatable or this clinic is a provincial hospital> then hold(treatment_time) send(home_village, t, np) else send(parent_clinic, t, np)	: either treat np & send home or send to next higher level clinic where t is time+travel_time.
5) repeat	

Figure 2

The program for a village:	Comments:
1) <determine next time, nt, and next person, np>	: determine the next time and person that needs to go to a clinic.
2) send(self, time+nt, null) receive(src, t, m)	: send null message to self and wait until either nt time passes or a person returns from a clinic.
3) if m is null then send (np, t, my_clinic) else <update local attributes>	: t = time + travel_time : person returning, record service completion statistics and any necessary followup treatment.
4) repeat	

Figure 3

the clinic program, for example, gives rise to a specific clinic process. A health system simulation would consist of multiple interacting clinic and village processes.

Sequential Discrete Event Simulation

As defined in the previous section, a simulation can be viewed as the execution of a set of communicating sequential processes. A sequential simulation requires scheduling the execution of these processes so that object operations are performed in the appropriate order. For example, in the above health care system example, a person must be sent by a village to a clinic before the clinic can receive this person. Unix process scheduling, in contrast, attempts to minimise user process waiting times and maximise throughput. Here, scheduling must be based on the concepts of "simulation time" and "event".

A sequential simulation schedules the execution of processes so that all events occur in non-decreasing simulation_time order. A simple algorithm that implements sequential simulation can be defined as folows. An events_list is maintained which contains an ordered list of pending events, i.e., events_list: {event(p1,t1), event(p2,t2), ...} where t1<=t2, etc. The value returned by *time* will be t1 and process p1 is scheduled for execution until p1 either calls *hold*(t) or requests suspension of its execution for some other reason. If p1 calls hold(t) then t1 is changed to t1+t and this first event in events_list is re-inserted into the events_list so that the list remains in event time order.

In the health care system simulation, we can define two different types of system state changes, i.e., types of events, that can occur. One type of event occurs when a village sends a person to a health clinic. This "arrival" event may also involve waking up the clinic if it is idle when the person arrives. The only other type of event in this model is the "completion" of service, or treatment, for a person. This completion event may also involve waking up an idle village object to put the treated person in a list of people recuperating from surgery, for example.

An example execution sequence for one clinic and one village process is shown in Figure 4. At time 0, the village process executes for one unit of real time at the end of which it schedules person p1 to arrive at the clinic object at time 1 by calling "send(clinic, 1, p1)". The village object is then idle for 2 units of real execution time. At simulation_time 1 person p1 arrives at the clinic which then begins execution to assess and treat this person. After 2 units of real time, the clinic calls hold(4) which suspends its execution until simulation_time 5. The village then executes for 1 unit of real time at simulation_time 2, and again at simulation_time 4, scheduling the arrival of persons p2 and p3 at the clinic.

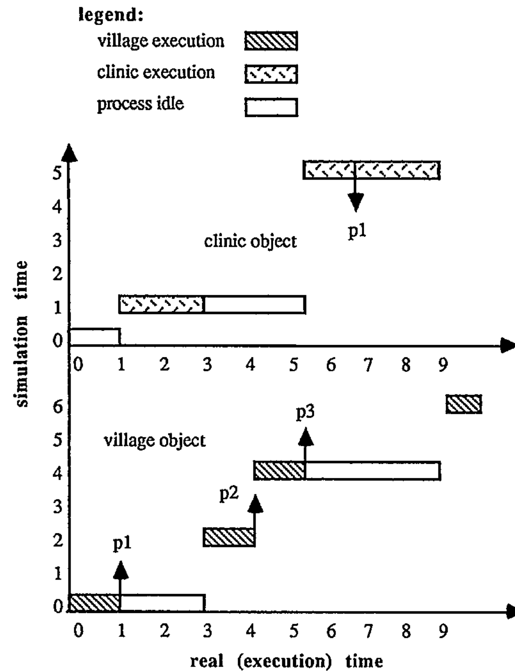


Figure 4 Process Scheduling in Sequential Simulation

The process view of simulation was partly an attempt to eliminate the necessity for the modeller to identify events at this level. At the lowest level of a discrete-event simulation we have the scheduling and execution of events. The clinic and village processes define the operation of these types of objects at a level above the occurrance of events, as in Figure 2. The execution of an object process, e.g., steps (1) to (5) above, may, in general, involve arbitrarily many events and event types.

Potential Parallelism

Since we have conveniently defined a simulation to be the execution of a set of communicating processes, it seems quite natural to attempt to execute these processes concurrently. This is the intent of distributed simulation. We may observe in the seqential health care

system simulation above that all the events do not have to be executed in simulation_time order. For example, the clinic could serve person p1 while the village is generating p2 and p3. The clinic can serve p2 and p3 while the village receives and processes the return of p1, and so on.

A valid execution schedule for the two clinic and village objects is shown in Figure 5 where it is assumed that two processors are available, one for each object. From the figure we see that each object has its own version of simulation_time, i.e., at a real time of 3 the village and clinic simulation_times are 4 and 1, respectively. Notice that the village object can proceed to generate p2 and p3 immediately after producing p1. Further, the clinic object need not wait to complete p1's service before its simulation time advances to 5 when this service period is completed. The village does have to wait until the clinic sends p1 back home before it can receive this person and the clinic does have to wait for the village to send another person if doesn't already have a person waiting for service.

In general, the execution of events in a simulation need only be partially ordered. Sequential scheduling algorithms, as defined here, enforce a partial order on event execution which is free only in the ordering of simultaneous events, i.e., events which have identical simulation_times can be executed in arbitrary order. However, as illustrated in Figure 5, this can be overly restrictive. We only need to order events which are causally related. In the health care system example, the village object causes a person to arrive at a clinic, and completion of service by the village causes the person served to return to the village. These two constraints still leave room for substantial parallelism.

The potential for parallelism is much greater when hundreds of clinics and villages are simulated. Many simulation experiments of interest will involve very few interactions outside of the nearest village/clinic pairs. Then, not only can we acheive partly parallel execution of the nearest clinic/village pairs as shown in Figure 5, but we may also be able to execute hundreds of these process pairs in parallel. The difficulty of course is how do we schedule these processes in a way which orders causally related events but still allows substantial parallelism.

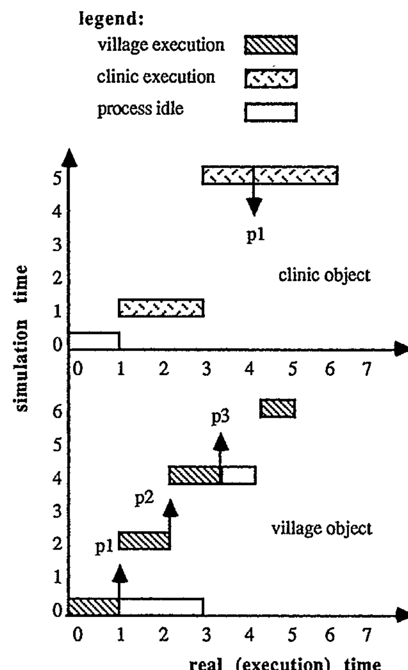


Figure 5 Distributed Process Scheduling

The Synchronization of Simulation Time & Speedup

Two classes of mechanisms for scheduling simulation process execution have been defined: optimistic mechanisms and conservative mechanisms [Unger & Jefferson 88]. There is a clear demarcation between these two mechanisms in that an optimistic mechanism risks synchronization errors which require correction by rolling back the process to a previous state and again computing forward [Jefferson 85]. Conversely, a conservative mechanism never risks synchronization errors but instead risks deadlock by waiting to be sure its safe to continue computing [Chandy & Misra 81].

Although one can clearly define the line between optimistic and conservative synchronisation methods, it is not so easy to define a purely optimistic mechanism. It seems possible to increase optimism by having a process make more and more assumptions about its environment , e.g., by guessing the source and contents of messages it has not yet received. It is also easy to introduce conservatism into an optimistic mechanism by having a process wait before taking a risk.

Our limited experience with mixed optimistic & conservative mechanisms suggest that greater optimism results in greater execution speedup [West 88], [Baezner 89].

REFERENCES

Baezner, D., Cleary, J., Lomow, G., Unger,B.(1989) Algorithmic Optimizations of Simulations on Time Warp. submitted to *Distributed Simulation, Part of the 1989 SCS Eastern Multiconference* March 28-31, Tampa, Florida

Birtwistle, G.M. (1985) *DEMOS: A Process Based Package,* Part 4 of 'An Approach to Discrete Event Modelling' (Birtwistle,G., Lomow,G. & Unger,B.) Submitted for publication to Transactions of the SCS.

Chandy, K. and Mistra, J.(1981) Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM* , April, pp.198-206.

Dahl, O.J. and Nygarrd, K. (1966) SIMULA - An ALGOL-based Simulation Language *CACM* 9(9), pp152-155.

Franta, W.R. (1977). *The Process View of Simulation.* New York: Elsevier.

Gordon, G. (1975) *The Application of GPSS V to Discrete Event Simulation.* Englewood Cliffs, NJ: Prentice Hall.

Jefferson, D. (1985) Virtual Time *ACM Transactions on Programming Languages and Systems* , 7(3) July.

Kruetzer, W. (1986) *System Simulation, Programming Styles and Languages* . Sydney, Addison-Wesley.

Unger, B., General Chairman & Jefferson, D., Program Chairman (1988) SCS Multi'88 Distributed Simulation Conference, San Diego, February.

West, D. (1988) *Optimising Time Warp* , MSc thesis, University of Calgary.