Sādhanā, Vol. 22, Part 5, October 1997, pp. 611–627. © Printed in India.

An overview of discrete event simulation methodologies and implementation

RAJESH MANSHARAMANI

Tata Research Development & Design Centre, Plot #54B, Hadapsar Industrial Estate, Pune 411 013, India
e-mail: rmansha@pune.tcs.co.in

MS received 2 December 1996; revised 12 March 1997

Abstract. Discrete event simulation has been widely used to model and evaluate computer and engineering systems and has been an on-going area of research and development. This paper presents an overview of the field. It covers specifications of discrete event systems, simulation methodology, simulation languages, data structures for event management, and front and back-end support in simulation packages including random number generation and resource management. The emphasis of the survey is on simulation methodology and event scheduling, which forms the core of any simulation package or environment.

Keywords. Discrete event simulation; event scheduling; process interaction; priority queue; simulation languages.

1. Introduction

The design of a system or a process often needs to be evaluated for correctness and engineering properties before its implementation. Simulation is a cost-effective mechanism to evaluate system and process design. Likewise, to study the behaviour of an existing system or process, simulation is often more cost effective than direct system or process measurement. Depending on the underlying system model, the simulation will take the form of solution of a set of equations, as in the case of a continuous system model, or execution of event-based program code, as in the case of a discrete-event system model. In this paper we will consider only simulation of discrete-event systems.

Ever since the sixties, discrete-event simulation has been widely used for modelling and evaluating computer systems, computer networks, real-time systems, distributed systems, database management systems, manufacturing systems etc. For example, to evaluate the configuration of a computer system for a banking application, to evaluate a resource management policy in an operating system, to study the behaviour of a local area network communication protocol, or to examine strategies for job shop scheduling. A number of

text-books and papers describe traditional as well as non-traditional uses of discrete event simulation, for example, Banks & Carson (1984), Jain (1991), Law & Kelton (1993), Banks & Norman (1996).

Given that several engineering disciplines have found the need for discrete event simulation, several languages and packages have been developed, both for general purpose use, as well as for a suitable set of applications. Advances in software development such as object-oriented design, data structures, and graphical user interfaces have caused advances in simulation techniques and software. As a result, discrete event simulation is an active area of research and development. This paper presents a survey of state of the art in sequential discrete event simulation[1]. The focus will be on simulation methodologies, event scheduling, languages and modelling support in software packages.

The rest of this report is organised as follows. Section 2 defines the terms and formalisms for discrete event systems. Section 3 surveys the different strategies that have been used for discrete event simulation. Section 4 presents a survey of simulation languages and packages. A core aspect of discrete event simulation, that is, future event scheduling is reviewed in § 5. Front-end and back-end support issues are discussed in § 6. Finally, § 7 summarises the main aspects of this paper.

2. Discrete event systems and their specification

To model any system we first need to define its state space, i.e., the variables that govern the behaviour of the system with respect to the metrics being estimated. If the variables continuously change with time it is called a continuous system. If the system state *instantaneously* changes at discrete points in time, instead of continuously, it is called a discrete system.

In discrete systems whenever some state variables instantaneously change value, this occurrence is denoted by the term *event*. The general behaviour of a discrete event system is that the system starts out with some initial state. The system remains in that state for a duration of time. Then an event occurs which causes the system to instantaneously transit to a new state. This behaviour repeats with the system transiting from state to state over time but remaining in a specific state for a duration of time. This is opposed to continuous systems where the system continuously changes state with time.

Let us consider a simple example of a discrete event system (DEVS), a single server queue that serves customers in first-come-first-serve order. It is assumed that the system is work-conserving, that is, the server will not be idle if there is a customer waiting for service and the server will not abruptly stop serving a customer. We model the state to be the number of customers in the system. The system starts out as empty. The first event to occur is a customer arrival and the state changes to one customer in the system. At this point the server begins service and the customer's departure is scheduled. The next event to occur can either be the second customer's arrival or the first customer's departure, whichever occurs first in time. In the former case, the state will change to two customers

[1]Though distributed simulation is an active research topic, the majority of industry uses sequential discrete event simulation on account of the simplicity in description

and in the latter case the state will return to the system being empty. The events can also be specified more elaborately, for example, start service and end service. In this case events occur instantaneously. A customer arrival at an empty queue will cause a start service to occur at the same time, without any change in state[2].

To model a DEVS we need to describe the state space of the system, the events in the system, the state transitions upon events and the times at which events will occur. The event space can be classified further as *external input events*, *external output events* and *internal events*. External input events are events that are triggered from outside the system, as in the case of a customer arrival at the queue. External output events are those that are generated as output from the system, as in the case of customer departure. Internal events are changes in state variables that do not affect the system environment, e.g., *start service*. The simulation of the discrete event system is done by means of generating events and executing the actions associated with the events.

A DEVS can be precisely modelled using the abstract specification first presented by Zeigler (1976) (see also Evans 1988 and Fishwick 1993). This formalism has been widely used in the DEVS literature. Let $\mathcal{I}$ denote the set of external input event types to the discrete event system. Let $\mathcal{O}$ denote the set of external output event types. Let $\mathcal{S}$ denote the set of states of the system, where a subset of the state variables includes the list of future-event times at the given time instant. Then the DEVS abstraction is given by the 7-tuple

$$\langle \mathcal{I}, \mathcal{O}, \mathcal{S}, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, \tau \rangle,$$

where $\delta_{\text{int}} : S \rightarrow \mathcal{S}$ is the internal transition function dictating state transitions due to internal events.

$\delta_{\text{ext}} : \mathcal{Q} \times \mathcal{I} \rightarrow \mathcal{S}$ is the external transition function dictating state transitions due to external input events. $\mathcal{Q} = \{(s, e) | s \in \mathcal{S}, 0 \leq e \leq \tau(s)\}$ is the total state set of the model; (s, e) represents the state of having been in state s for elapsed time e.

$\lambda : S \rightarrow \mathcal{O}$ is the output function generating external events, and

$\tau : \mathcal{S} \rightarrow \mathcal{R}_0^+$ is the time advance function. If the system is in state s at time t then the system will remain in that state until time $t + \tau(s)$. In other words, $\tau(\ ) \geq 0$ is the minimum of future event times.

Composite models are constructed in Zeigler (1987) by coupling models (either atomic or composite) by means of *external input coupling*, *external output coupling*, and *internal coupling*. External input coupling specifies how input events of the composite model are identified with the input events of the components. External output coupling specifies how output events of the composite model are identified with the output events of the components. Internal coupling specifies how the components inside the coupled model are interconnected by means of connections from output events of components to input events of others.

[2] If the modeller so desires he can specify the state as the tuple ⟨number of customers in queue, number of customers in server⟩ so that even instantaneous events are associated with a change in state

3. Methodologies of discrete event simulation

Section 2 defined discrete event systems. We now consider how to simulate discrete event systems, specifically, from a programmer's viewpoint. We describe the most common strategies for designing and implementing discrete event simulation programs. (See Evans 1988 for more details.) Three such methodologies are followed in the literature: event scheduling, activity scanning, and process interaction.

3.1 *Event scheduling*

In this strategy a list of all events in the system is first constructed. Each event is taken individually and described in terms of the particular interaction between entity (say customer) and resource (say server). Associated with each event is the corresponding action or procedure to be invoked when the event occurs. Consider for example a single server queue: the events of interest are customer arrival, start service, end service, and leave system. The processing required at say end service is to compute system metrics, e.g., response time, for the departing customer, clear its resource allocation (if needed), check if there is another customer waiting in queue. If so, then schedule a start service event for that customer.

While the approach of event scheduling is straightforward, it involves programming at a low level. All events have to be enumerated in one place. Care must be taken to ensure that reactivations are scheduled, as in the case of the start service event being scheduled as soon as the previous end-service is completed. Likewise, the start service event must also be scheduled if an arriving customer finds the queue empty. These reactivations typically occur in zero time, but if they are not explicitly specified the simulation will terminate. Thus, the responsibility of event scheduling lies entirely on the programmer, which is why this strategy is so called. However, because of the explicit specification the efficiency of this strategy is best as compared to the other strategies.

3.2 *Activity scanning*

The purpose of this strategy is to overcome the reactivation problem of event scheduling. As before, the list of events is first drawn up. But now the events are classified into two types: B-activities and C-activities. B-activities are activities that are Bound to occur, whereas C-activities are those that are Conditional. For example, the arrival of a customer is a B-activity since it is unconditional. However, the start service activity is a C-activity since it occurs only if the previous event was *end-service* and the queue size is greater than zero, or if the previous event was a customer arrival and the queue is empty.

In this strategy the programmer does not have to explicitly specify reactivations. The system automatically handles them. The generic structure of a simulation written in this strategy consists of three phases. The A-phase advances the time for simulation, the B-phase checks the type of the B-activity that has occurred at that time and executes the procedure associated with it, and the C-phase checks the C-activity (or activities), if any, that need to be executed at that time, and executes the corresponding event-handling procedure. Note that the A, B and C phases must proceed in sequential order. Each C-activity has a head

part that is the condition under which it is true and a procedure to execute if the condition evaluates to true.

While this strategy relieves the programmer from explicitly programming reactivations, the execution is inefficient because all the C-activities need to be checked in the C-phase. If many C-activities are rare-events, then most of the effort spent is wasted. Such inefficiency was absent in event scheduling because of the explicit invocation of C-activities. To make the processing efficient and yet retain the simulation strategy, attempts have been made to group C-activities and B-activities that share the same entity. Thus, a set of C-activities is grouped with all the B-activities from which they obtain their entities. During the C-phase only those C-activities need to be examined which are grouped with the B-activity that executed.

3.3 *Process interaction*

In the preceding discussion it seems natural to group B and C-activities pertaining to a common entity, and to order them in their actual sequence of succession. Thus, rather than view the system as a set of event modules it is more natural to view the system as flows of entities.

The process interaction methodology describes the system's workings from the viewpoint of an entity flowing through the system. The model is thus described as a projected life-history of a typical entity, called a process (Evans 1988). To start with, the resources and entities are identified. Then the entity's resource requirements and interaction with resources, duration of activities etc., are all described in the process.

Each new instance of an entity is a separate process instance. The execution of a process simulation takes a single process instance and executes it until deactivation. That is, if the process has to wait for a resource, or if the process is to spend a period of time in a given activity (e.g., service time). At that time another process instance is taken and reactivated from the position where it left off.

Thus, the process oriented approach is modular. All events and activities comprising a process are described in the form of the entity model in one place. Note that processes may also apply to resources. For example, a server process. In the single server queue example, the process oriented simulation can be expressed by means of the entity customer. The sequence of description is

generate customer every inter_arrival
acquire server (*waits in queue if necessary*)
get serviced for t time units
release server
destroy customer

In this case events are implicitly handled by the language. The modular approach makes it very easy to concisely specify the customer's behaviour. To implement the process oriented strategy the language needs to provide support for process suspension and reactivation. For example, when the customer requests for the server and the server is busy then that customer instance needs to be deactivated. When the server is released then the first waiting customer in the queue needs to be reactivated. This is usually handled by means of a restricted form

of coroutine called *semi-coroutine* (Dahl 1968) or *generator* (Marlin 1980) construct provided in the simulation language. In this form, the choice of coroutine to be resumed is determined by a master module that decides on the basis of the next event that is to be activated.

The process oriented approach allows for high level programming but requires that the simulation language provide coroutine support, which is typically not the case with general purpose languages such as C and C++. (See § 4 for extensions to general purpose languages.) Further, by its very nature it can cause the system to deadlock if two process instances are waiting for each other to release their resources before acquiring those of the other. This may not be the intended purpose in cases where more than one resource is accessed at the same time, but the language does not support atomic access of multiple resources. (See Evans 1988 for examples.) However, in most cases this is either a non-issue or special constructs are provided in the language.

3.4 *Summary*

As described in this section, there are three prevalent forms of expressing discrete event simulation programs: Event scheduling, activity scanning and process interaction. While the expressive power is the same in each case, they differ in the level of programming. Event scheduling is the most efficient mechanism in terms of execution speed but involves explicit specification of events and their actions globally across all parts of the system, as well as reactivations of conditional events. Activity scanning implicitly handles reactivations by explicitly stating which events are unconditional and which are conditional. However, it incurs a loss of efficiency due to explicit checks for conditional events at time advances.

Process interaction or process orientation offers a more natural modelling environment to the programmer since the focus is on the entity and its sequence of activities rather than on a global perspective of the entire system. It is efficient at run time but requires support for coroutines in the specification language. With the emergence of distributed systems some authors have proposed a message based approach to discrete event simulation (Bagrodia *et al* 1987) where the physical system is modelled as a set of communication processes. Events are modelled by message communications. An entity is modelled as a message communicating process.

4. Discrete event simulation languages and APIs

Since discrete event simulation is an important field in its own right, a number of languages have been designed specifically for the purpose. In other cases general purpose languages have been enhanced. There is also a common trend to provide application programming interfaces (APIs) for discrete event simulation in a general purpose high level language. In this section we provide a survey of languages, enhancements and APIs. Our aim is not be exhaustive or provide a critical review but to give a flavour of what exists in the discrete event simulation world.

4.1 *Discrete event simulation languages*

Historically, discrete event simulation languages were promoted since the 60's. Simscript used the event scheduling strategy, GSP the activity scanning and GPSS the process interaction strategy. Later with the popularity of Algol60, Simula was proposed as a preprocessor to perform simulation programming.

Currently, GPSS is generally regarded as the most popular simulation language. It is a block-structured language that was conceived to run on mainframes and supported by IBM. The GPSS/H processor, a product of Wolverine Software, USA, was first released in the late 1970s and is available on a variety of hardware platforms ranging from PCs to Sun workstations (Scher 1991). The GPSS model is that transactions (i.e., entities) flow through systems to produce dynamic effects. They interact with resources by flowing into *blocks* or program statements. A block remains inactive until a transaction attempts to enter it. Over 40 different blocks describe resource requirements, conditional branching, queueing, data collection, report generation and attribute control (Evans 1988).

Simscript started out as a language for event scheduling but the modern version Simscript II.5 incorporates a process interaction strategy. Like GPSS, Simscript is commercially available (Markowitz *et al* 1987). On the other hand Simula, which is more of an object oriented language, has received less commercial support as a simulation language. SLAM (Pritsker 1986) and SIMAN (Pegden *et al* 1990) have also been used over the last decade. Recently, there is a trend towards object oriented simulation languages as in the case of MODSIM III which is commercially available from CACI, and HSL which was proposed in Sanderson *et al* (1991).

In general, event scheduling languages provide features for specifications of events and associated actions. The actions will be the code necessary to update the system's state and generation of a random time for a future event. The process-oriented languages, typically, include a richer set of language constructs which apart from time delays include interaction of processes with resources, such as suspend at a specific queue, resume once the head of queue is reached, or suspend until is specific condition is satisfied in the form of a *wait-until* construct. The constructs in GPSS, for example, include GENERATE an entity, ENTER into a resource, ADVANCE the time spent at a resource, QUEUE at a resource, LEAVE the resource and TERMINATE an entity.

A number of other languages have been proposed in the literature (see Evans 1988 for example) and many comparisons have been done (cf. Tocher 1965, Dahl 1968 and Virjo 1972). Simulation languages commercially available for personal computers have also been compared, e.g. the comparison between *Simple*$_1$ and *Simian* in Houten (1988).

4.2 *Extensions and APIs*

The primary disadvantage of a special purpose simulation language is that it has to be learned and compilers for it have to be bought. To overcome this problem a number of extensions and application programming interfaces (APIs) for simulation have been proposed for general purpose languages.

Such an effort started with GASP in the 60s. GASP is a collection of FORTRAN subroutines for simulation, and has been very popular among FORTRAN users. Another popular choice has been to extend Pascal for quasi-parallel programming (Kaubisch *et al* 1976; Kriz & Landmayr 1980) or provide APIs in it (Hac 1982; Marsden 1984). Extensions to Algol68 have been proposed in Shearn (1975). Use of PL/I for discrete event simulation has been proposed by Hac (1984), of Ada by Bruno (1984), of Modula-2 in HPSIM (Sharma & Rose 1988) and of SR by Olsson (1990).

In the late 80's and in the 90's, several libraries for discrete event simulation are available in C and C++. In C++ it is possible to write class libraries that support coroutines (cf. Stroustrup & Shapiro 1987) and this allows for process oriented simulation. Process oriented packages in C or C++ include CSIM (Schwetman 1988, 1990), SIM++ (Lomov & Baezner 1990) and YacSim from Rice University. Event scheduling packages include SMPL (MacDougall 1987), and SimPack (Fishwick 1992). More recent packages in C++ provide object-oriented simulation features, for example, C++SIM (Little & McCue 1994) and Awesime (Grunwald 1991).

Apart from extensions and libraries some packages use only diagrams as a means of modelling and simulation. This obviates the need to learn a special purpose language, but it can be rather cumbersome if there are a number of diagrammatic blocks as in GPSS. On the other hand specific tools such as Petri-net analysers (cf. Evans 1988) or queueing network simulators (cf. Melamed & Morris 1985, Funka-Lea *et al* 1991) are useful within the range of applications they model. To model applications beyond their limits one must opt for a programming language. More recently, Shanbagh & Gopinath (1997) have proposed a C++ simulator generator from graphical specifications.

5. Future event management

No matter what be the simulation strategy or the simulation language, in discrete event simulation the underlying mechanism is scheduling of events. Events are generated for future times. At any time the next scheduling instant is the minimum of the future event times. In practice, the number of future events may range from a handful, say in a single server queueing system at light to moderate load, to a huge number of events, say in the simulation of a wide area network.

The following operations are needed on the *future event list* data structure: *insert* an event (represented by its time and a pointer to associated event information), *delete minimum* time event and return the information pointer, *delete* or *cancel* any arbitrary event from the list. The first two are the most common operations, the last one is used occasionally in some simulation applications, e.g., resource preemption.

The abstract data type appropriate for future event scheduling is the *priority queue*, which can be implemented in many ways. The performance of the implementation is subject to the operational profile of *inserts*, *delete-mins*, and *cancels*. Not surprisingly, this has led to a significant number of proposals in the literature for future event scheduling data structures. In this section we discuss the various implementation techniques and follow it with comparisons of the techniques that have appeared in the literature. For more detailed surveys see Srikanth (1996) and Evans (1988).

5.1 *Future event list implementations*

The simplest representation of the future event list is an array or linked list ordered by time. Though simple to implement it is rather inefficient for large list sizes since search time is linear in the size of the list. For this reason researchers have proposed index structures over linked lists (Wyman 1976; Franta & Maly 1977, 1978; Henriksen 1977, 1983; Comfort 1979; Nevalainen & Teuhola 1979; Davey & Vaucher 1980; Blackstone *et al* 1981; O'Keefe 1985), a popular scheme among these being that of Henriksen (1977, 1983), which has been incorporated into GPSS (Henriksen & Crain 1982).

Another common approach is to use a *d-heap*. The *2-heap* was proposed by Williams (1964) and then generalised to $d > 2$ by Johnson (1975). In a *d-heap* the nodes are maintained in *heap-order*, where the value of a node is no less than the value of its parent. The *d-heap* is a complete *d-ary* tree satisfying heap order. Using breadth first search the nodes can be indexed into a single array. Other heap-based implementations include the *leftist tree* (Crane 1972, Knuth 1973b), *pagoda* (Francon *et al* 1978), *skew heap* (Sleator & Tarjan 1983, 1986), *binomial queue* (Vuillemin 1978), *pairing heap* (Fredman *et al* 1986, Stasko & Vitter 1987), *Fibonacci heap* (Fredman & Tarjan 1987), *relaxed heap* (Driscoll *et al* 1988) and *radix heap* (Ahuja *et al* 1990).

Search tree-based structures have been popular as well. Binary search trees are the natural choice and have been analysed for future event scheduling in Evans (1983) and Vaucher & Duval (1975). A variant, called *p-tree*, to combine the advantage of linear list and efficiency of tree structures was proposed in Jonassen & Dahl (1975). Among balanced trees or partially balanced trees the most popular version for priority queues is the *splay tree* (Sleator & Tarjan 1985).

A particular type of implementation called *calendar queue* was proposed by Brown (1988) and also independently proposed by Davidson (1989). In this representation time is split into buckets and keys fall within bucket ranges. Indices are wrapped around for the 'next year'. A more recent structure called *fishspear* (Fischer & Paterson 1994) has worst case performance as the d-heap but is oriented towards better performance in the common case and can also be implemented for sequential storage.

5.2 *Analyses and performance comparisons*

Ordinary linked lists sorted by time require $O(n)$ time for insert and $O(1)$ time for delete-min and delete. On the other hand *d-heaps* require $O(\log\ n)$ for insert, delete and delete-min (Tarjan 1983). Bollobas & Simon (1985) analyse repeated random insertions into a heap where each ordering of the inserted elements is equally likely. They obtain that the number of exchanges per insertion is bounded by a constant of about 1.76. Fibonacci heaps on the other hand have $O(\log\ n)$ amortised time for delete and delete-min and $O(1)$ for insert (Fredman & Tarjan 1987). Driscoll *et al* (1988) show that these times hold in the worst case for relaxed heaps.

Various comparisons of priority queue implementations have been reported in the literature. Several comparisons have been done under the *hold model* (Vaucher & Duval 1975) where a *hold* operation is one that removes an event from the priority queue and schedules a new event after an interval of time d from a specific distribution $\mathcal{F}$. The hold model

consists of a sequence of hold operations and is parameterised by the number of events in the event queue and the distribution $\mathcal{F}$. It has been used in many of the early studies including Comfort (1979), Davey & Vaucher (1980), Englebrecht-Wiggans & Maxwell (1978), Franta & Maly (1977, 1978), Henriksen (1977), Jonassen & Dahl (1975), Vaucher & Duval (1975), and Vaucher (1977).

Jones (1986) compared several implementations under the hold model and showed many to outperform heaps. The splay tree was shown to have best performance in his study. In a later study Brown (1988) showed that under the hold model the calendar queue performs better than the linear linked list and the splay tree. Chung *et al* (1993) used a Markov hold model to evaluate 14 implementations. Also, using a token ring simulation for comparison they recommend using the splay tree and the calendar queue while stating that heaps are quite 'stable' albeit with lower performance.

McCormack & Sargent (1981) compare several implementations from Comfort (1979), Davey & Vaucher (1980), Franta & Maly (1977), Henriksen (1977), Taneri (1976), Ulrich (1978), Vaucher & Duval (1975) and Wyman (1976), and show that results from real simulation runs are different from that when the hold model is used. They show that Henriksen's method (Henriksen 1977) and the modified heap perform well and are less sensitive to scheduling distributions.

Thus, in general, it is not readily apparent which implementation works best for a given simulation application. Worst case analyses do not reflect performance accurately for the average case. Average case analyses have been done under restrictive assumptions or special cases of applications. It will be desirable therefore for simulation packages to adopt a variety of future event management mechanisms as in SimPack (Fishwick 1992). A knowledgeable user can select the right mechanism but what would be more desirable is a high level interface to select the right mechanism for the simulation's operational profile. Typically, in simulations one needs to run several experiments in the debugging stage itself and during this phase the various priority queue implementations can be compared.

6. Front end and back end support

Though scheduling of future events forms the core of discrete event simulation, there are a number of other features that are desirable in a simulation environment. They can be classified into front end requirements in the form of diagram editing and graphical output, and back end support in the form of random number generation, resource management, statistical libraries. This section first specifies desirable features of simulation front ends and then desirable features of back ends.

6.1 *Simulation front end*

The simulation front end must in the least capture the system topology and possibly simplify the model description in terms of user input, and display graphical output of simulation results.

The most general case of user input should allow for a diagram editor to specify user defined icons to represent processes or resources, connectivity across icons, and connectivity constraints if any. Few packages allow this, however. GPSS has a cumbersome diagram

notation with over 40 different diagram types representing equivalence to program statements. Typically, no package allows for general purpose model specifications. However, specific application domains such as queueing network simulations capture all user specifications through the front end as in the case of PAW (Melamed & Morris 1985) and its successor $Q+$ (Funka-Lea *et al* 1991; also see Shanbagh & Gopinath 1997).

6.2 *Simulation back end*

All simulation environments include support for random number generation, and resource management. Some also include statistical libraries. We elaborate on each of these features below. Note that our emphasis is only on what is provided in standard packages. There are other important aspects such as variance reduction of output and rare-event simulation, which are not covered in this paper[3].

6.2a *Random number generation:* Random number generation forms an integral part of a simulation environment. The underlying model of a discrete event system assumes that the system remains for a given time in each state. This duration of time is modelled using a random number distribution, for example, the inter-arrival time at a queue is often modelled as an exponential distribution. Likewise, service time of a customer in a queue, number of database items that a query will access can be modelled using random number distributions.

Every operating system is usually equipped with a random number generator that generates uniform random variates. In simulation we additionally require generation of non-uniform random variates. Generation of random numbers is more difficult than what one might expect. As Knuth (1973a) says, "Random numbers should not be generated by a method chosen at random."

Several packages and studies have used defective random number generators. For instance, the study of Majumdar *et al* (1988) that simulated performance of parallel processor allocation policies used a defective technique for random number generation which led to incorrect policy comparisons. This was later corrected in Leutenegger & Vernon (1990). The 1988 version of CSIM (Schwetman 1988) used the *rand*( ) function which is well known to have poor random number generation as given in the UNIX system manual page for *random*( ). A survey of more than 50 computer science text books that contained software for random number generation revealed that most of these generators are unsatisfactory (Park & Meller 1988). This shows the importance of using reliable random number generators as given in Knuth (1973a), Park & Meller (1988), and L'Ecuyer (1988).

Any simulation environment must support a variety of distributions, both discrete and continuous. There should be support for multiple random number streams. The package should allow for transformations on random variables to support practical distributions as well as allow for empirical distributions as obtained from measured data. Typical discrete distributions include uniform, Bernoulli, binomial, geometric, Poisson and typical continuous distributions include uniform, exponential, Erlang, hyper-exponential, normal

[3]The interested reader can find details in the July 1993 issue of *ACM Transactions on Modelling and Computer Simulation*

and gamma. Devroye (1986) provides several methods of generating non-uniform random numbers for many distributions.

The general techniques that are used for non-uniform random number generation are the *inverse method* and *acceptance-rejection.* In the former, a random number is generated by first generating a uniform number between 0 and 1, and then using it as an argument to the inverse of the distribution. In the latter method, the required density is bounded by that of a scaled version of another distribution for which it is known how to generate random samples. The samples from the known distribution are repeatedly taken until one falls under the required distribution.

6.2b *Resource management:* Discrete event simulation is widely used to study the behaviour of resource contention. For example, contention for CPU and disk in computer systems, contention for database in DBMS, contention for machines in job shops, contention for toll booths on highways, etc. Associated with each resource is a resource handler and *contention queue(s)* to store contending entities. The resource handler decides how to schedule entities from the contention queue(s) on to the resource.

The resource by itself may contain multiple servers, as in the case of a parallel processor or a petrol bunk. The entities may all contend in a single contention queue or may be split across several queues each contending for a subset of the servers. Most simulation environments provide support for single server single queue resources. Some provide support for multiple server single queue resources.

The resource handling discipline can be preemptive or non-preemptive depending on the application in hand. In computer systems preemption is very common at the CPU (but not at disk) whereas in manufacturing systems preemption of executing jobs at plants is typically absent. Among non-preemptive disciplines the most common ones are first come first serve (FCFS) and fixed priority. Some systems also provide support for first fit and best fit. Among preemptive disciplines the most common one is fixed priority with preemptive resume. In CPU scheduling round robin is a common preemptive discipline where preemption occurs on every time quantum.

Having built-in resource scheduling disciplines simplifies the work of the programmer who is now given access to insertion and deletion of entities in contention queues. If the programmer desires to use a very specific discipline, the interface for using the discipline must be the same as that for ones provided by the simulation environment. The simulation environment should provide the facility to integrate custom resource schedulers.

6.2c *Statistical libraries:* The purpose of discrete event simulation is to study the behaviour of a given system. The behaviour of interest to the user is usually captured in the form of metrics such as average and variance of response time, throughput and resource utilisation. To correctly estimate these metrics the programmer needs to insert *measurement probes* at appropriate places in the program. Good simulation environments provide support for probes, that is, creation and initialisation, sampling, determining averages, variance and distributions of accumulated data, as well as confidence intervals. A sophisticated package will provide support for different types of probes, e.g., space average and time average.

Accumulation of samples can vary according to the estimation method being used. Two common techniques for estimating results are regenerative simulation and the method of batch means (Law & Kelton 1993). Regenerative simulation is widely applicable and produces *correct* results (Welch 1983). It essentially estimates metrics at system regeneration points[4] and determines confidence intervals (i.e., estimates of the variance of the metric being analysed) across regeneration points. When enough regeneration points have been encountered to meet the desired confidence interval the simulation stops. While this method produces correct results as per renewal theory, it can be rather costly in large systems to generate regeneration points. For this purpose the method of batch means is a more efficient approach, where metrics are estimated at the end of a given batch size of samples and confidence intervals are calculated across batches. The batch size must be chosen with caution since a small batch size can cause correlation between successive batches. For more details on statistical techniques to estimate steady state behaviour see Pawlikowski (1990).

7. Summary

We have surveyed the field of discrete event simulation on uniprocessors. We have summarised the formal specifications for discrete event systems. Various strategies for simulation, that is, event scheduling, activity scanning and process interaction have been reviewed. Discrete event simulation languages and extensions and APIs of general purpose languages for discrete event simulation have been briefly covered. Data structures for future event management have been surveyed. These include simple linked lists, heaps and variants and a variety of search trees and assorted data structures. Front end and back end support for simulation have been described. Note that topics such as output analysis, variance reduction techniques, rare event simulation are specialised topics that deserve a separate survey in their own right, and have been treated as outside the scope of this paper. Likewise, emerging technologies such as object oriented simulation have not been covered.

Currently, the trend has been to enable the user to build a simulation model of the system under consideration and to efficiently run the simulation code. Not much emphasis has been given on separating modelling from simulation as is prevalent in the continuous simulation world, e.g., simulation of chemical process plants. It would be desirable to create model libraries of resources or of subsystems which can be used for a variety of applications to be simulated. Typically, the approach is to rewrite code from simulation to simulation. This is not only wasteful in terms of development time but also incurs greater chances of bugs in the simulation.

The author would like to thank the anonymous referees for their valuable comments that improved the quality of the paper. The author would also like to thank S Hanumantha Rao for his valuable feedback on an earlier version of this report that greatly helped in improving the exposition.

[4] A regeneration point is a state from which the system stochastically evolves afresh. For example, an empty queue in a single server queueing system

References

Ahuja R K, Melhorn K, Orlin J B, Tarjan R E 1990 Faster algorithms for the shortest path problem. *J. Assoc. Comput. Mach.* 37: 213–223

Bagrodia R L, Chandy K M, Misra J 1987 A message-based approach to discrete event simulation. *IEEE Trans. Software Eng.* 13: 654–665

Banks J, Carson J S 1984 *Discrete-event system simulation* (Englewood Cliffs, NJ: Prentice Hall)

Banks J, Norman V 1996 Second look at simulation software. Non-traditional uses can lead to unexpected benefits. *OR/MS Today* 23: 4

Blackstone J H, Hogg G L, Phillips D T 1981 A two-list synchronization procedure for discrete event simulation. *Commun. ACM* 24: 825–829

Boas P V E, Kaas R, Zijlstra E 1977 Design and implementation of an efficient priority queue. *Math. Syst. Theory* 10: 99–127

Bollobas B, Simon J 1985 Repeated random insertions into a priority queue. *J. Algorithms* 6: 466–477

Brown R 1988 Calendar queues: a fast O(1) priority queue implementation for the simulation event set. *Commun. ACM* 31: 1220–1227

Bruno G 1984 Using Ada for discrete event simulation. *Software Pract. Exper.* 14: 685–695

Chung K, Sang J, Rego V 1993 A performance comparison of event calendar algorithms: an empirical approach. *Software Pract. Exper.* 23: 1107–1138

Comfort J C 1979 A taxonomy and analysis of event set management algorithms for discrete event simulation. In *Proc. 12th Annu. Simulation Symposium,* pp 115–146

Crane C A 1972 Linear lists and priority queues as balanced binary trees. Tech. Rep. STAN-CS-72-259, Comput. Sci., Stanford, CA

Dahl O J 1968 Discrete event simulation languages. In *Programming Languages* (ed.) F Genuys (London: Academic Press)

Davey D, Vaucher J 1980 Self-optimizing partition sequencing sets for discrete event simulation. *INFOR J.* 18: 21–41

Davidson G A 1989 Calendar P's and queues. *Commun. ACM* 32: 1241–1243

Devroye L 1986 *Non-uniform random variate generation* (New York: Springer Verlag)

Driscoll J R, Gabow H N, Shrairman R, Tarjan R E 1988 Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation. *Commun. ACM* 31: 1343–1354

Englebrecht-Wiggans R, Maxwell W L 1978 Analysis of the time indexed list procedure for synchronization of discrete event simulations. *Manage. Sci.* 24: 1417–1427

Evans J B 1983 *Investigations into the scheduling of events and modelling of interrupts in discrete event simulation.* Ph D thesis, Dept. of Operations Research, Univ. of Lancaster

Evans J B 1988 *Structures of discrete event simulation: An introduction to the engagement strategy* (Chichester: Ellis Horwood)

Fischer M J, Paterson M S 1994 Fishspear: a priority queue algorithm. *J. Assoc. Comput. Mach.* 41: 3–30

Fishwick P A 1992 SimPack: getting started with simulation programming in *C* and C++. In *Proc. Winter Simulation Conference,* Arlington, VA, pp 154–162

Fishwick P A 1993 A simulation environment for multimodeling. *Discrete Event Dynamic Syst.: Theor. Appl.* 3: 151–171

Francon J, Viennot G, Vuillemin J 1978 Description and analysis of an efficient priority queue representation. In *Proc. 19th Annual Symp. on Foundations of Computer Science,* Piscataway, NJ, pp 1–7

Franta W R, Maly K 1977 An efficient data structure for the simulation event set. *Commun. ACM* 20: 596–602

Franta W R, Maly K 1978 A comparison of HEAPS and the TL structure for the simulation event set. *Commun. ACM* 21: 873–875

Fredman M L, Tarjan R E 1987 Fibonacci heaps and their uses in improved network optimization problems. *J. Assoc. Comput. Mach.* 34: 596–615

Fredman M L, Sedgewick R, Sleator D D, Tarjan R E 1986 The pairing heap: a new form of self-adjusting heap. *Algorithmica* 1: 111–129

Funka-Lea C A, Kontogiorgos T D, Morris R J, Rubin L D 1991 Interactive visual modeling for performance. *IEEE Software* 8(5): 58–68

Grunwald D 1991 A users guide to Awesime: an objected oriented parallel programming and simulation system. Tech. Report CU-CS-552-91, University of Colorado, Boulder.

Hac A 1982 Computer system simulation in Pascal. *Software Pract. Exper.* 12: 777–784

Hac A 1984 PL/I as a discrete event simulation tool. *Software Pract. Exper.* 14: 692–702

Henriksen J O 1977 An improved events list algorithm. In *Proc. Winter Simulation Conference*, pp 554–557

Henriksen J O 1983 Event list management – a tutorial. In *Proc. Winter Simulation Conference*, pp 543–551

Henriksen J O, Crain R C 1982 *GPSS/H user's manual* 2nd edn (Annandale: Wolverine Software Corp.)

Houten 1988 Simulation languages for PCs take different approaches. *IEEE Software* 5: 91–94

Jain R 1991 *The art of computer system performance analysis: Techniques for experimental design, measurement, simulation and modelling* (New York: Wiley)

Johnson D B 1975 Priority queues with update and finding minimal spanning trees. *Info. Proc. Lett.* 4: 53–57

Jonassen A, Dahl O J 1975 Analysis of an algorithm for priority queue administration. *BIT* 15: 409–422

Jones D W 1986 An empirical comparison of priority queue and event set implementations. *Commun. ACM* 29: 300–311

Kaubisch W H, Perrott R H, Hoare C A R 1976 Quasiparallel programming. *Software Pract. Exper.* 6: 341–356

Knuth D E 1973a *The art of computer programming: Vol. 2/Seminumerical algorithms* (Reading, MA: Addison-Wesley)

Knuth D E 1973b *The art of computer programming: Vol. 3/Sorting and searching* (Reading, MA: Addison-Wesley)

Kriz J, Landmayr H 1980 Extensions of Pascal by coroutines and its application to quasiparallel programming and simulation. *Software Pract. Exper.* 10: 773–789

L'Ecuyer P 1988 Efficient and portable random number generation. *Commun. ACM* 31: 742–749, 774

Law A M, Kelton W D 1993 *Simulation, modeling and analysis* (New York: McGraw-Hill)

Leutenegger S T, Vernon M K 1990 The performance of multiprogrammed multiprocessor scheduling policies. In *Proc. ACM SIGMETRICS* 18: 226–236

Little M C, McCue D L 1994 Construction and use of a simulation package in C++. *C User's J.* 12: 3

Lomow G, Baezner D 1990. A tutorial introduction to object-oriented simulation and Sim++. In *Proc. Winter Simulation Conference*, pp 149–153

MacDougall M H 1987 *Simulating computer systems: techniques and tools* (Boston: MIT Press)

Majumdar S, Eager D, Bunt R 1988 Scheduling in multiprogrammed parallel systems. In *Proc. ACM SIGMETRICS* 16: 104–113

Markowitz H M, Kiviat P J, Villaneuva R 1987 *Simscript II.5 programming language* (Los Angeles: CACI)

Marlin 1980 Coroutines. In *Lecture notes in computer science 95* (Berlin: Springer-Verlag)

Marsden B W 1984 A standard pascal event simulation package. *Software Pract. Exper.* 14: 659–684

McCormack W M, Sargent R G 1981 Analysis of future event set algorithms for discrete event simulation. *Commun. ACM* 24: 801–812

Melamed B, Morris R J 1985 Visual simulation: the performance analysis workstation. *IEEE Comput.* 18: 87–94

Nevalainen O, Teuhola J 1979 Priority queue administration by sublist index. *Comput. J.* 22: 220–225

Olsson R A 1990 Using SR for discrete event simulation. *Software Pract. Exper.* 20: 1187–1208

O'Keefe R M 1985 Comment on "Complexity Analysis of Event Set Algorithms". *Comput. J.* 28: 245–272

Park S K, Meller K W 1988 Random number generators: good ones are hard to find. *Commun. ACM* 31: 1192–1201

Pawlikowski K 1990 Steady state simulation of queueing processes: a survey of problems and solutions. *ACM Comput. Surv.* 22: 123–170

Pegden C D, Sadowski R P, Shannon R E 1990 *Introduction to simulation using SIMAN* (Sewickley: System Modeling)

Pritsker A A 1986 *Introduction to simulation and SLAM II* (New York: Halstead)

Sanderson P, Sharma R, Rozin R, Treu S 1991 The hierarchical simulation language HSL: a versatile tool for process-oriented simulation. *ACM Trans. Modeling Comput. Simulation* 1: 113–153

Scher J M 1991 Reworked GPSS/H book is a strong standard. *IEEE Software* 8(4): 105–106

Schwetman H 1988 Using CSIM to model complex systems. In *Proc. Winter Simulation Conference*, pp 246–253

Schwetman H 1990 Introduction to process-oriented simulation and CSIM. In *Proc. Winter Simulation Conference*, pp 154–157

Shanbagh V K, Gopinath K 1997 A C++ generator from graphical specifications. *Software Pract. Exper.* 27: 395–424

Sharma R, Rose L L 1988 Modular design for simulation. *Software Pract. Exper.* 18: 945–966

Shearn D C 1975 Discrete event simulation in ALGOL68. *Software Pract. Exper.* 5: 279–293

Sleator D D, Tarjan R E 1983 Self-adjusting binary trees. In *Proc. ACM SIGACT Symp. on Theory of Computing*, pp 235–245

Sleator D D, Tarjan R E 1985 Self-adjusting binary search trees. *J. Assoc. Comput. Mach.* 32: 652–686

Sleator D D, Tarjan R E 1986 Self-adjusting heaps. *SIAM J. Comput.* 15: 52–69

Srikanth S 1996 A software tool for performance analysis of data structure representations. M Tech thesis, Dept. of Comput. Sci. & Eng., Regional Engineering College, Warangal

Stasko J T, Vitter J S 1987 Pairing heaps: experiments and analysis. *Commun. ACM* 30: 234–249

Stroustrup B, Shapiro J E 1987 A set of C++ classes for co-routine style programming. In *Proc. USENIX C++ Workshop*, pp 417–439

Taneri D 1976 The use of subcalendars in event driven simulations. In *Proc. Summer Simulation Conference*, pp 63–66

Tarjan R E 1983 *Data structures and network algorithms* (Philadelphia: SIAM)

Tocher K D 1965 Review of simulation languages. *Oper. Res. Q.* 16: 189–217

Ulrich E G 1978 Event manipulation for discrete simulations requiring large numbers of events. *Commun. ACM* 21: 777–785

Vaucher J G 1977 On the distribution of event times for the notices in a simulation event list. *INFOR J.* 15: 171–182

Vaucher J G, Duval P A 1975 A comparison of simulation event list algorithms. *Commun. ACM* 18: 223–230

Virjo A 1972 A comparative study of some discrete-event simulation languages. In *Proc. Nordata Conference*, Helsinki, pp 1532–1564

Vuillemin J 1978 A data structure for manipulating priority queues. *Commun. ACM* 21: 309–314

Welch P 1983 Statistical analysis of simulation results. In *Computer performance modeling handbook* (ed.) S S Lavenberg (New York: Academic Press)

Williams J W J 1964 Algorithms 232: Heapsort. *Commun. ACM* 7: 347–348

Wyman F B 1976 Improved event scanning mechanisms for discrete event simulation. *Commun. ACM* 19: 350–353

Zeigler B P 1976 *Theory of modelling and simulation* (New York: Wiley) (Reissued by Krieger, Malabar, FL in 1985)

Zeigler B P 1987 Hierarchical, modular, discrete-event modelling in an object-oriented environment. *Simulation* 49: 219–230