11 Simulation Models—Introduction and Motivation

So far in the course the stochastic models which we have considered have been solved *analytically*. What this means is that by carrying out analysis of the system we have been able to deduce the steady state behaviour of the corresponding stochastic process, expressed as a probability distribution over the possible states. Such models can be regarded as a *mathematical abstraction* of the system. The analytic model is a representation which can be analysed mathematically to deduce the behaviour of the system.

In contrast, a stochastic simulation model can be regarded as an *algorithmic abstraction* of the system—a simulation model gives a representation which when *executed* reproduces the behaviour of the system.

We still assume that the system is characterised by a family of random variables $\{X(t), t \in T\}$. As the value of time increases, and in response to the "environment" (represented by random variables within the model) the stochastic process progresses from state to state. Any set of instances of $\{X(t), t \in T\}$ can be regarded as a path of a particle moving randomly in a state space, S, its position at time t being $X(t)$. These paths are called *sample paths*. Using the analytic approach of Markov processes we characterised all possible sample paths by the global balance equations. Using simulation we investigate the sample paths directly. In other words we allow the model to trace out a sample path over the state space. Each *run* of the simulation model will generate another, usually distinct, sample path.

There are many reasons why simulation may be preferable to analytic modelling.

Level of Abstraction As we have seen, Markovian modelling relies on many assumptions and abstractions—these may not be appropriate for the system being studied. For example, it may be unrealistic to assume that only one event can happen at any time, or that the inter-event times are all exponentially distributed. Simulation models allow us to represent a system at arbitrary levels of detail, although you should remember that there is always a trade-off between how elaborate the model is and how long it takes to produce a statistically significant run.

Transient Analysis In some cases we are not interested in the steady state behaviour of a system, but in its *transient* behaviour. Some systems never reach a steady state. Even those that do will usually have a "warm-up" period while the behaviour settles into the regular pattern which characterises steady state. The analytic solutions we have considered earlier in the course ignore this period since the global balance equations only capture the behaviour after steady state has been reached. However it is not always possible to extract all the information we want about a system from its steady state behaviour. A sample path derived from a simulation model will clearly represent transient behaviour in addition to steady state[1].

Size of State Space In most cases solving a model analytically involves constructing and storing the complete state space of the model. For example, for a Markov process with N states solving the global balance equations involves (at least) an $N \times N$ matrix (the generator matrix) and a vector with N elements (the steady

[1] For Markov processes transient behaviour can be derived analytically but this generally requires more mathematically sophisticated analysis techniques than those we have considered in this course.

state distribution). When N becomes very large this becomes infeasible. In contrast, in a simulation model the state space is generated "on-the-fly" by the model itself during execution so it does not need to be all stored at once.

11.1 Constructing simulation models

Simulation models are complex computer programs. They can be developed in any programming language but there are distinct advantages to using a language or package specifically designed for simulation. Languages with simulation features built-in, such as SimJava, provide facilities for many of the routine features of a simulation model. These features are common to all models, regardless of the system being represented. This allows the performance analyst to concentrate on the issues specific to the system being modelled and to not worry about issues which are general to all simulations. Using a general language the modeller might be able to write a more efficient program. However, this is likely to take much longer to develop because the program will be responsible for the simulation management, as well as the representational aspects of the model itself.

Some of the common features of simulation management are listed below.

Event scheduler The events in the system change the state and therefore drive the simulation. An event scheduler keeps track of the events which are waiting to happen, usually as a linked list, and allows them to be manipulated in various ways. For example,

- schedule event E at time T;
- hold event E for a time interval ∂t;
- cancel a previously scheduled event E;
- hold event E indefinitely (until it is scheduled by another event);
- schedule an indefinitely held event.

The event scheduler is one of the most frequently executed components of a simulation model. It is called before every event, and it may be called several times during one event to schedule other new events. Therefore it is very important that it is implemented efficiently.

Simulation clock and time management Every simulation model must have a global variable representing the *simulated* time. The event scheduler is usually responsible for advancing this time, either one unit at a time or, more commonly, directly to the time of the next scheduled event. This latter approach is called *event-driven* time management.

System state variables Since a simulation model generates a random walk over the state space of the system it is essential that the model has variables to capture the state of the system at each step. If a simulation run is stopped in the middle, it can be restarted later if, and only if, the values of all state variables are known.

Event routines Each event in the system brings about a state change; thus, in the simulation model the effect of each event must be represented in a way which updates the system state variables, and in some cases, schedules other events. How the event routines are generated will depend on the simulation modelling paradigm used to construct the model.

Random number/random variate generator Random numbers play a crucial role in most discrete event simulations. For example the impact of the environment on the system, e.g. inter-arrival times, is usually represented by random variables of some specified distribution. A random number generator is used to generate a sequence of random values between 0 and 1. These values are then transformed to produce a sequence of random values which satisfy the desired distribution. This second step is sometimes called random variate generation.

Report generator Performance measures are derived from a simulation run by observing the values of parameters of interest during the execution. Most simulation modelling languages and packages contain built-in routines to calculate statistics from these observations and generate a report when the run is completed.

Trace routines A trace of the system can be a useful tool for debugging (sometimes called *verifying*) and validating a model. It is a time-ordered list of events, state variable values or output parameter values. Most simulation languages provide routines to generate traces which can be switched on or off in a particular run of the model. Since trace generation is usually very inefficient it is generally only used during model development.

Dynamic memory management The number of active entities during the execution of a simulation model will vary continuously as new entities are created and old ones become obsolete. Most simulation languages provide automatic garbage collection to remove obsolete entities.

11.2 Simulation modelling paradigms

There are a number of approaches to discrete event simulation; the two most commonly used are *event based modelling* and *process based modelling*. Each of these modelling styles is briefly described below but all the example simulation programs which we will consider in the notes are developed using the process based approach.

Note that the high level modelling paradigms which we have already considered in the course—SPN, GSPN, PEPA and queueing networks—can be used to generate simulation models as well as Markov processes.

11.2.1 Event Based Simulation

Event based simulation focusses the modeller's attention on the individual events which can occur within the system. An event within the system may generate several actions in the model—these are grouped together in an event routine. The event scheduler maintains a pointer to the appropriate event routine, and this is executed when the event reaches the head of the event list.

A good example of a system suited to event based simulation would be a simple queue with deterministic service times and customer defections. The events are a customer arrives, A; a customer defects, D; a customer begins service, B; and a customer ends service, E.

At each event time as well as the processing of the event to represent the behaviour of the system some processing internal to the model might be done. This can be event tracing or statistical collection—whatever has been specified in the model description. For example, if the queue above were being modelled to calculate the average queue length and throughput the events would be implemented as follows.

An event A **at time** t will result in the following actions

- add one to the state variable representing queue length, and record the time at which the change occurred,
- schedule an event D at the time $t+d$, where d is the length of time a customer will wait without defecting,
- schedule an event B to occur as soon as possible, depending on the availability of the server,
- schedule another event A at time $t+a$ where a is the inter-arrival time.

An event D **at time** $t+d$ will

- decrease the queue length by one and the record the time at which the change occurred,
- de-schedule event B on hold since time t.

An event B **at time** $t+w$ $(w<d)$ will

- decrease the queue length by one and record the time at which the change occurred,
- de-schedule the event D at time $t+d$,
- schedule an event E at time $t+w+s$, where s is the service time,

An event E **at time** $t+w+s$ will

- increment the busy time of the service centre by s,
- add one to the total number of customers served,
- activate the first event B waiting in the event list.

11.2.2 Process Based Simulation

The process based approach to simulation modelling collects events together into related sequences which are ordered by time. These sequences are related in the sense that they all involve the same entity within the system; they are termed *processes*. For example, above we could consider each customer to be a process within the system, since it generates a sequence of related events, and track its progress through the queue.

The process based approach views a system as a web of concurrent, interacting entities—the processes. All the actions associated with an entity's behaviour are grouped together to form a life-cycle for entities of that type. The event scheduler still maintains a list of scheduled events centrally but this will now generally be in the form of a pointer to a process/object. The process will maintain a record of its current state and which action it should perform when next scheduled. This style of modelling maps very well onto object-oriented programming where a class is associated with each type of entity; objects belonging to the class then represent instances of the entity.

For the example of a queue with deterministic inter-arrival and service times (but no defections), we would define a class to represent the arrival process. This class (`Source`) would generate the event representing a customer entering the queue and then delay until the arrival time of the next customer. A second class would represent the server (`Server`). It is passive in the sense that it first waits to be notified of an event (the arrival of a customer) and then represents the service of the customer as a delay. This simple deterministic queue represented in SimJava is shown in Figure 24.

In this case each entity class is instantiated just once in the body of the simulation program, which is itself declared as a class `Queue`. We will go into the SimJava language and the facilities it provides in more detail in the next lecture note.

11.3 Common mistakes in simulation studies

Inappropriate level of detail Because simulation modelling allows the modeller to incorporate arbitrary levels of detail it is sometimes tempting to represent too much in the model. This will have a cost in terms of execution time. The best strategy is to start with the simplest possible model and only add detail as it becomes necessary (sensitivity analysis).

Unverified models As already mentioned, simulation models are complex programs and as such are prone to bugs in the same way that any complex program is. Verification is intended to make sure that the model behaves as it was intended to.

Invalid models Validation is intended to make sure that the model is a good representation of the system. A model may behave correctly in the sense that it contains no bugs, but if the assumptions which have been made during its construction are inappropriate the results obtained from the model will still be invalid.

Too short simulation runs Especially when a model has a large state space, the model must be executed for a long (simulated) time to ensure that the sample path which is generated is statistically valid.

Single simulation runs Each run of the simulation represents only one sample path through the state space of the model, corresponding to a particular sequence of random numbers. In order for results to be statistically valid they should be based on several sample paths obtained using different sequences of random numbers.

Poor random-number generators Random number generators are used extensively in simulation models to produce values for random variables modelling the environment or internal working of the system. A poor random-number generator may introduce correlation and/or bias into the value of those random variables.

Jane Hillston ⟨jeh@inf.ed.ac.uk⟩. September 19, 2003.

```
class Source extends Sim_entity {
  private Sim_port enqueue;

  public Source(String name) {
    super(name);
    enqueue = new Sim_port("enqueue");
    add_port(enqueue);
  }

  public void body() {
    while (Sim_system.running()) {
      sim_schedule(enqueue,0.0,0);
      sim_pause(10.0);
    } } }

class Server extends Sim_entity {
  private Sim_port arrival;

  public Server(String name) {
    super(name);
    arrival = new Sim_port("arrival");
    add_port(arrival);
  }

  public void body() {
    Sim_event next = new Sim_event();
    while(Sim_system.running()) {
      sim_wait(next);
      sim_process(8.5);
      sim_completed(next);
    } } }

class Queue {
  public static void main(String args[]) {
    Sim_system.initialise();
    Source customers = new Source("Customers");
    Server service = new Server("Service");
    Sim_system.link_ports("Customers", "enqueue", "Service", "arrival");
    Sim_system.set_termination_condition(Sim_system.EVENTS_COMPLETED,
                                            "Service", 0, 10, false);
    Sim_system.run();
  }
}
```

Figure 24: A simple SimJava program representing the deterministic queue in the process-oriented style