PK8207 - Lecture memo

Jørn Vatn
Email: jorn.vatn@ntnu.no

Updated 2020-02-03

Discrete event simulation

Introduction

In discrete-event simulation, the operation of a system is represented as a chronological sequence of events. Each event occurs at an instant in time and marks a change of state in the system (Robinson, 2004). For example, if the up- and down times of a system of components are simulated, an event could be "component 1" fails, and another event could be "component 2" is repaired. A third event could be that a customer arrives to a service desk. Rather than explicitly assessing the performance of a system by the laws of probability, we just "simulate" the system, and calculate the relevant statistics to get the performance. This presentation gives the core elements of discrete-event simulation and some indications regarding implementing simple situations. An Excel file,`DiscreteEventSimulation.xlsm` is available. There are basically three ways to perform discrete-event simulation, i.e., by application of:

1. A tailored tool to perform simulation for special situations. Miriam Regina is such a tool to perform simulation of an offshore oil- and gas production system.

2. A dedicated discrete-event simulation program with predefined functions and procedures to handle events, housekeeping of resources etc. SimEvent is such a tool which builds on Matlab. Other tool are Extend-Sim and Arena. Experience shows that many such tools comes with user manuals which emphasize one application area, and that it is not straight forward to apply the tool for a specific application without considerable effort in order to understand the logic of the tool.

3. Ordinary programming languages like FORTRAN, C/C++ and Visual Basic for Application (VBA). With the ordinary programming languages the programmer has full control over everything, but has to develop all the code needed.

In this presentation we will start with the basic, hence the starting point is 3 from the above list. Note that code written in VBA is generally slow to execute compared to e.g., FORTRAN or C++ code. The advantage of VBA code is that it is very easy to integrate the code with model specification either from an MS Excel work sheet, or an MS Access application. In the code listing that follows we only provide pseudo code. Essentially VBA syntax is used. To simplify the code variable declaration is generally omitted. It is, however, good practice to declare all variables used. This will help verifying the code, and also ensure more optimal code. VBA allows use of user defined types by the TYPE statement. To refer to type elements the standard reference by a period (.) is used. Also note that code is optimized by using the following construct

```
With <Variable>
   Code, where an element is referred only by a period (.) and the
      element name, e.g.,
   .Time = .Time + random variable
End With
```

Lists etc. that are used are sometimes linked. Generally the syntax (^Element) is used as a pointer statement. VBA does not support pointers, so here the pointer is just an integer pointing to an element in a table. Linked lists are implemented as arrays of user defined types with one element named Next. To traverse a linked list the following structure is then used:

```
Next = pointer to first element in the list
Do While Next <> NIL
   ... Code ...
   Next = List(Next).Next
Loop
```

Where we exit the loop if some criterium is met.

Components of a Discrete-Event Simulation

In addition to the representation of system state variables and the logic of what happens when system events occur, discrete event simulations include the following concepts:

- Clock. The simulation must keep track of the current simulation time, in whatever measurement units are suitable for the system being modelled. In discrete-event simulations, time advances in discrete jumps, that is, the clock skips to the next event start time as the simulation proceeds.
- Event. A change in state of a system.

- Events List (PES). The simulation maintains at least one list of simulation events. This is sometimes called the pending event set because it lists events that are pending as a result of previously simulated events but have yet to be simulated themselves. In other presentations this list is denoted the future event set. The event list, or pending event set, will in this presentation be denoted PES. The pending event set is typically organized as a priority queue, sorted by event time. That is, regardless of the order in which events are added to the event set, they are removed in strictly chronological order.

- Event notice. An element in the PES describing when an event is to be executed. An event is described by the time at which it occurs and a type, indicating the code that will be used to simulate that event. It is common for the event code to be parametrised, in which case, the event description also contains parameters to the event code. In a programming context the `EventNotice()` is a function that places an event in the PES. Logically an event notice is an event in the PES waiting to be executed at a given point of time. When implementing the EventNotice() function we refer to an event notice as the action to insert the event in the PES.

- Activity. A pair of events, one initiating and the other completing an operation that transform the state of the entity. Time elapses in an activity. Repairing a component is treated as an activity. At the start of the activity the component is in a fault state, and at the end of the activity the state of the component is a functioning state. Another activity could be the service of a customer. We usually assume that no continuous changes are taking place during the activity. If this is the case, it is much more demanding to implement activities in the computer code.

- Random-Number Generators. The simulation needs to generate random variables of various kinds, depending on the system model. This is accomplished by one or more pseudorandom number generators. To generate pseudorandom numbers a two-step procedure may be used, (i) generate a pseudorandom number from a uniform distribution on the interval [0,1], and then (ii) use this number (or more such numbers) as a basis for generating a pseudorandom number from the required distribution for example by a call to the inverse cumulative distribution function (CDF). Modern programming languages provide at least a simple subroutine to generate [0,1] variables, but in most cases the user need to provide subroutines to transform this number to the required distribution. Various subroutine libraries exist for more efficient random-number generation.

- Statistics. The simulation typically keeps track of the system's statis-

tics, which quantify the aspects of interest. In the example above, it is of interest to track the relative portion of time the system is in a fault, or a degenerated state.

- Ending Condition. A discrete-event simulation could run forever. So the simulation designer must decide when the simulation will end. Typical choices are “at time t" or ‘after processing n number of events".

Simulation Engine Logic

The main loop of a discrete-event simulation is basically:

- Start of simulation.
- Initialize Ending Condition to FALSE.
- Initialize system state variables.
- Initialize Clock (usually starts at simulation time zero).
- Schedule one or more events in the PES.
- “Do loop" or “While loop", i.e., While (Ending Condition is FALSE) then do the following:
  - Get the NextEvent from the PES.
  - Set clock to NextEvent time.
  - Execute the NextEvent code and remove NextEvent from the PES.
  - Update statistics.
- Generate statistical report.
- End of simulation.

Although the main program is very simple it is usually much harder to write the code for processing the events. It is informative to use the notation `On<Event>` as function name of the functions to be called when the NextEvent is retrieved and to be executed by running the event function. This emphasise that the `Event` is what is happening. For example if a “failure” is put in the PES, and when the time comes to “execute” this failure, the `OnFailure` event is called. The `OnFailure` code then has to programmatically change state variables and “post" new events, for example when the repair is completed. An other `On<Event>` function could be `OnArrival` which is a function to call when a new customer arrives into a waiting room.

Implementing the pending event set (PES)

Logically the PES may be seen as a queue of events waiting for execution from left to right. As simulation proceeds we may think of the queue as a number of Post-it notes placed on the blackboard sorted in chronological order with respect to time of execution. The main simulation loop will then fetch the leftmost Post-it note and execute the code described. The code to be executed will typically add one or more new Post-it notes on the blackboard. When a new Post-It note is added it is placed at it's appropriate time. There are several ways to manage the PES in a computer program. Examples are binary search trees, splay trees, skip lists and calendar queues. The following properties need to be balanced:

- It should be fast to insert a new event in the PES
- It should be fast to delete an event from the PES
- It should be fast to get the first event in the PES
- The PES should not occupy too much memory
- The code to implement the PES should not be too complex

A very simple implementation of a PES will be to use a double-linked list. For double linked list it is easy both to insert and delete records. If the list is rather short searching for events is also very efficient. However, if the number of elements in the PES becomes large, searching time is proportional to the number of elements. Since we for each execution of an event typically add a new event, this means that searching the appropriate place to insert the new event will require much computer time. In the following we will provide a rather simple approach for implementation of the PES. The implementation comprises the following elements:

- A linked list to represent the PES.
- An indexed list (Indx) to enable fast access to the PES

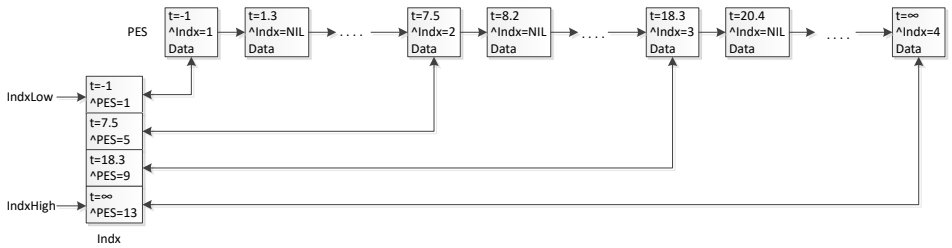

Figure 1: Implementation of the PES with a supporting indexed list

Figure 1 shows the structure. The PES contains the pending events. In addition it has a header and a tail with corresponding times equal -1 and infinity respectively. Since the header may be considered as a dummy event, we will always have access to the header, and this element will never be removed from the list. This makes it easy to maintain the list as simulation proceeds. The indexed list is a list of representative times sorted in chronological order. Since this list is sorted, it is very fast to access a given point of time. This indexed list has pointers to the PES. There are fewer elements in the indexed list than in the PES. This means that when we search for an event in the PES we will only get a rough position where to start searching. For example if we will like to insert a new element in the PES at time $t = 9.1$ we first search the indexed list and finds that 9.1 is between entry 2 and 3 in the indexed list. Entry 2 points to the entry corresponding to $t = 7.5$ in the PES. Hence, we start searching in the PES from that point until we find the place where to insert the new element with $t = 9.1$. Searching time now consist of (i) the time to (binary) search the indexed list which is proportional to $\log_2 N_{\mathrm{IL}}$, where N_{IL} is the number of events in the indexed list, and (ii) the time to search the PES from the starting point until we find an event, or the place where to insert a new event. For example, if the size of the PES, say N_{PES}, is $50N_{\mathrm{IL}}$ we need in average 25 comparisons in the PES. As elements are inserted and deleted from the PES, the indexed list becomes out of date. We therefore now and then update the indexed list. This will require approximately N_{PES} operations. Obvious there will be a need to optimize the parameter N_{IL} and the frequency of re-indexing of the indexed list. This is not discussed further here.

The following functions are now required to operate on the PES and the indexed list

```
Function FindLow(t)
Search Indx to find i where Indx(i).t  <= t < Indx(i+1).t
Return a pointer to PES, i.e. FindLow = Indx(i).^PES
End Function

Function EventNotice(time, functionPointer, parameters)
EventNotice = InsertElementInList(time, Data=[functionPointer, parameters])
End Function

Function InsertElementInList(t, Data)
Start = FindLow(t)
Traverse PES from PES(Start) until PES(i).t  <= t < PES(i+1).t
Insert a new element in PES at position i+1
PES(i+1).Data = Data
If NeedToUpdate > IndexFrequency
   NeedToUpdate = 0
```

```
   IndexPES()
Else
   NeedToUpdate = NeedToUpdate + 1
End If
InsertElementInList = i + 1
End Function

Function ReleaseEventNotice(EventNotice)
i = FindLow(t)
Traverse PES from PES(i) until i = EventNotice
If PES(EventNotice)^Indx <> NIL Then SetFlag
Disconnect and Free EventNotice
If SetFlag Then IndexPES()
End Function

Function IndexPES()
Make Indx be an empty list
Run through PES
For every k'th element, insert new entry in Indx, and make links
End Function

Function GetNxtEvent()
PrevClock = Clock
Clock = PES(PES(1).Next).t
GetNxtEvent = PES(PES(1).Next).Data
ReleaseEventNotice PES(1).Next
End Function

Function InitPES()
Empty tables
Create dummy events for header (t = -1), and tail (t = infinity)
End Function

Function GetClock()
GetClock = Clock
End Function

Function TimeElapsed()
TimeElapsed = Clock - PrevClock
End Function
```

The above listed functions have been implemented in the `DiscreteEventSimulationLib` VBA module of the `DiscreteEventSimulation.xlsm` Excel file. The user only needs to care about the following functions:

- InitPES()
- EventNotice(time, functionPointer, [P1], [P2], [P3], [P4])
- GetNxtEvent()
- ReleaseEventNotice(eventNotice)
- GetClock()
- TimeElapsed()

Also an "Exectue" function is required to execute the `On<event>` got by the `GetNxtEvent` function. The `On<event>` function needs to work with parameters, for example which component failed since the same functional call is used for all component failures. Up to 4 parameters of integer type can be passed to the `EventNotice` function. The `GetNxtEvent` returns a `Variant` type data, which could be passed to the `Execute()` function. The `Execute()` function will execute the subroutine pointed to by the `functionPointer`, and where the parameters `P1` to `P4` are passed to the subroutine.

Library of functions for generating pseudorandom numbers

In order to run a probabilistic discrete-event simulation we need a library of pseudorandom number generators. In the `rndLib` VBA module of the `DiscreteEventSimulation.xlsm` Excel file some standard functions are provided. Among these are:

```
Function rndExponential(MU)
Returns a random number exponentially distributed with mean MU
End Function

Function rndWeibull(A, B)
Returns a random number, Weibull distributed with shape parameter
 A and location parameter B, i.e., parametrization is
f(x) = ( A/(B^A) ) * x^ (A-1) * EXP( -(x/B) ^A ) which is different from the textbook
End Function

Function rndErlang(K, B)
Returns a random number, Erlang-k distributed with shape parameter
K and scale parameter B
End Function

Function rndNormal(MU, SIGMA)
Returns a random number, Normally distributed with
mean MU and standard deviation SIGMA.
End Function
```

Note that many of the standard life time distributions exist with different parametrization. It is therefore required to check the parametrization used in the library of random number generators.

A simple failure and repair model

We are now able to write our first discrete-event simulation program. We consider a very simple model where there is one component that either is in a fault state, or it is functioning. A global variable DownTime is used to store accumulated down time. To run the program, we need two subroutines, OnFailure() is used to handle the situation when component fails, and OnRepair() is used to handle the situation when the component is repaired:

```
Sub OnFailure()
EventNotice GetClock() + rndExponential(10), AddressOf OnRepair
compStatus = Down
End Sub

Sub OnRepair()
EventNotice GetClock() + rndExponential(1000), AddressOf OnFailure
compStatus = Up
End Sub
```

In each of these subroutines the `EventNotice` is called with a point of time generated by adding an exponentially distributed number and a function pointer to a subroutine. In (Microsoft) VBA we use the prefix `AddressOf` to get the memory address of the subroutine that should be executed at the point of time given. The main program now looks like:

```
Sub MainProgram
downTime = 0
prevClock = 0
compStatus = Up
InitPES
EventNotice rndExponential(1000), AddressOf OnFailure
Do While getClock() < 100000
  nxtEvent = getNxtEvent()
  If CompStatus = Down Then downTime = downTime + (getClock() - prevClock)
  prevClock = getClock()
  Execute nxtEvent
Loop
Debug.Print "U = " & DownTime / GetClock()
End Sub
```

Note that it is not straight forward to implement the steps required to execute a function. In the example above we have passed a so-called function pointer

to the EventNotice() function. How this is actually implemented will vary from one implementation language to another. In for example VBA (Visual basic for Application used by MS Office) we can write an `Execute()` function to execute the subroutine pointed to by the data structure returned by the `getNxtEvent()` function.

In Windows VBA it is rather easy to implement an `Execute()` that uses the address of the function to be called when an event is fetched from the pending event set. In MAC Excel this is not so easy. Therefore to have VBA code that runs under Windows as well as on a MAC a slower implementation may be used where we take advantages of the `Application.Run` command. This command can run a subroutine specified by it's name. Using such an approach we then rather pass the name of the function to the `EventNotice` routine.

In the example we have considered exponentially distributed failure and repair times. In a more general setting we may use any distribution for both time to failure and time to repair as long as we are able to generate the corresponding pseudorandom numbers.

Exercise 1

Write a VBA code that calls the `EventNotice` with point of times 5,4,6,1,3. Use the `getNxtEvent` to retrieve the events and verify that they are picked in the correct order. Then repeat the same procedure, but use the `ReleaseEventNotice` to remove point of time 4 before retrieving the events.

Exercise 2

Implement the simple failure and repair model described above in a VBA program and find the unavailability. Compare with the analytical result.

Exercise 3

In a workshop there are two production lines in parallel. Each production line has a critical machine with constant failure rate $\lambda = 0.01$ failures per hour. There is one (common) spare machine that can replace a failed machine. We assume that switching time can be ignored. The repair rate of the machines is assumed constant and equal to $\mu = 0.2$ per hour. If a production line is down the loss is assumed to be $c_U = 10\ 000$ NOKs per hour. Only one repair man is available.

- Write a VBA code that simulates the situation.
- Find the expected loss due to downtime.

- If production is not 24/7 but runs from 07:00 to 15:00 it is reasonable to assume that we do not lose production during night, but that repair may continue. Implement this in the VBA code.
- Repeat the analysis, but assume that two repair men are available.
- How much should one be willing to pay per hour for having this extra backup on repair resources?