General Simulation Principles

Christos Alexopoulos and Dave Goldsman

Georgia Institute of Technology, Atlanta, GA, USA

5/27/18

Outline

Steps in a Simulation Study

(i) Problem Formulation — statement of problem.

Profits are too low. What to do?

Customers are complaining about the long lines. Help!

(ii) Objectives and Planning — what specific questions should be answered?

How many workers to hire?

How much buffer space to insert in the assembly line?

(iii) Model Building — both an art and science.

M/M/k queueing model?

Need physics equations?

(iv) Data Collection — what kinds of data, how much?

Continuous? Discrete?

What exactly should be collected?

Budget considerations.

(v) Coding — decide on language, write program.

Modeling paradigm — Event-Scheduling or Process-Interaction?

100s of languages out there.

(vi) Verification — is the code OK? If not, go back to (v).

(vii) Validation — is the model OK? If not, go to (iii) and (iv).

(viii) Experimental Design — what experiments need to be run to efficiently answer our questions?

Statistical considerations

Time / budget

(ix) Run Experiments — press the "go" button and make your production runs, often substantial. May require a lot of time.

(x) Output Analysis — statistical analysis, estimate relevant measures of performance.

Often iterative with (viii) and (ix).

Almost always need more runs.

(xi) Write Reports, Implement, and Make Management Happy.

Outline

- A *system* is a collection of *entities* (people, machines, etc.) that interact together to accomplish a goal.
- A *model* is an abstract representation of a system, usually containing math/logical relationships describing the system in terms of states, entities, sets, events, etc.
- *System state:* A set of variables that contains enough information to describe the system. Think of the state as a system "snapshot."

  E.g., in a single-server queue, all you might need to describe the state are: $L_Q(t) =$ number of people in queue at time t, and $B(t) = 1$ [0] if server is busy [idle] at time t.
- Entities can be permanent (e.g., a machine) or temporary (e.g., customers), and can have various properties or *attributes* (e.g., priority of a customer or average speed of a server).

- A *list* (or *queue*) is an ordered list of associated entities (e.g., a linked list, or a line of people).
- An *event* is a point in time at which the system state changes (and which can't be predicted with certainty beforehand). E.g., an arrival event, a departure event, a machine breakdown event. "Event" technically means the time that the thing happens, but loosely refers to "what" happens (e.g., an arrival).
- An *activity* is a duration of time of *specified length* (aka an *unconditional wait*). E.g., an exponential customer interarrival time or a constant service time (since we can explicitly generate those).
- A *conditional wait* is a duration of time of *unspecified length*. E.g., a customer waiting time — we don't know that directly (we just know arrival and service times).

Outline

- The *simulation clock* is a variable whose value represents simulated time (which doesn't equal real time).
- Time-Advance Mechanisms — how does the clock move?
  - Always moves forward (never goes back in time). Two ways:
  - *Fixed-Increment Time Advance*: Update the state of the system at fixed times, nh, $n = 0, 1, 2, \ldots$, where h is chosen appropriately. This is used in continuous-time models (e.g., differential equations) and models where data are only available at fixed times (e.g., at the end of every month). Not really emphasized in this course!
  - *Next-Event Time Advance*: The simulation clock is initialized at 0. Times of all known future events are determined and placed in the *future events list* (FEL), ordered by time. The clock advances to the *most imminent* event, then to the next most imminent event, etc. At each event, the system state is updated, and the FEL is updated.

Notes on FELs:

- The system state can only change at event times. Nothing really happens between events.
- The simulation progresses by sequentially executing (dealing with) the most imminent event on the FEL.
- What do we mean by "dealing with"?
  - The clock advances to the most imminent event.
  - The system state is updated, depending on what type of event it is (arrival, departure, etc.). For example, if it's an arrival event, you may have to turn on an idle server, or add a new customer to the queue of a busy server.
  - Update the FEL.

- What do we mean by "Update the FEL"? Any time there's an event, the simulation may update the chronological order of the FEL's events by inserting new events, deleting events, moving them around, or doing nothing.
  - Example: After guy arrives at a queue, typical simulation programs will immediately spawn the next arrival time. If this arrival time is significantly far enough in the future, we'll simply put it at the "end" of the FEL (in terms of ordered execution times of the events).
  - If that next arrival turns out to happen before another event, e.g., a slow server finishing his current customer, then that next arrival has to be *inserted* in the interior of the FEL (in terms of ordered execution times).
  - What if the arrival is a nasty-looking guy, and a number of customers currently in line get disgusted and leave or switch to other lines? Then you have to *delete or move* entries in the FEL.

- Need efficient list processing (e.g., *linked lists*) for the FEL.
  - Singly and doubly linked lists intelligently store the events in an array that allows the chronological order of the events to be accessed.
  - Such lists easily accommodate insertion, deletion, switching of events, etc.
  - Take a Computer Science course to learn more.
- Be careful about events that take place (or are scheduled to take place) at the same time, e.g., a guy shows up at the same time that another guy finishes getting served. It is important to establish ground rules for how to deal with ties.
- Every discrete-event simulation language maintains a FEL somewhere deep in its heart.
- Good news: Most commercial simulation packages take care of the FEL stuff for you, so you don't have to mess around at all with FEL logic. It's completely transparent!

Next-Event Time Advance Example (adapted from BCNN): Consider the usual single-server FIFO queue that will process exactly 10 customers. The arrival times and service times are:

customer	1	2	3	4	5	6	7	8	9	10
arrival time	1	3	4	10	17	18	19	20	27	29
service time	5	4	1	3	2	1	4	7	3	1

Let's construct a table containing entries for event time, system state, queue, future events list, and cumulative statistics (cumulative server busy time and customer time in system).

Let A denote an arrival event and D a departure (e.g., 2A is the customer 2 arrival). Suppose that services have priorities over arrivals in terms of updating the FEL; so we'll break ties by handling a service completion first — get those guys out of the system!

Next-Event Time Advance Example (cont'd):

Clock	System State		Queue	FEL	Cumul Stats	
t	$L_Q(t)$	$B(t)$	(cust, arr time)	(event time, event type)	busy time	time in sys
0	0	0	$\emptyset$	(1, 1A)	0	0
1	0	1	$\emptyset$	(3, 2A), (6, 1D)	0	0
3	1	1	(2, 3)	(4, 3A), (6, 1D)	2	2
4	2	1	(2, 3), (3, 4)	(6, 1D), (10, 4A)	3	4
6	1	1	(3, 4)	(10, 2D), (10, 4A)	5	10
10	0	1	$\emptyset$	(10, 4A), (11, 3D)	9	18
$\vdots$						

Tedious as this may appear, the computer handles all of this stuff in a flash. $\square$

Outline

I hinted in the last section that commercial discrete-event simulation packages help you avoid a lot of grief by automatically handling the FEL and carrying out the drudgery themselves. How is that possible?

It comes down to a choice between two modeling "world views" — the event-scheduling approach (☹) vs. the process-interaction approach (☺).

Event-Scheduling Approach. Concentrate on the events and how they affect the system state. Help the simulation evolve over time by keeping track of every event in increasing order of time of occurrence. This is a bookkeeping hassle.

You might use this approach if you were programming in C++ or Python from scratch. If you think that you'd like to do so, see the next few pages outlining a generic simulation program. . .

Generic Simulation Program Flowchart (from Law's book).

Main Program

0. Initialization Routine
   - Set clock to 0
   - Initialize system state and statistical counters
   - Initialize FEL
1. Invoke Timing Routine — what/when is the next event?
   - Determine next event type — arrival, departure, end simulation event
   - Advance clock to next time
2. Invoke Event Routine — for event type i
   - Update system state
   - Update statistical counters
   - Make any necessary changes to FEL. E.g., if you have an arrival, schedule the next arrival or maybe schedule end of simulation event. If simulation is now over, write report and stop. If simulation isn't over, go back to the Timing Routine.
3. Go back to 1

Details on Arrival Event

- Schedule next arrival
- Is server busy?
  - If not busy,
    - Set delay (wait) for that customer = 0
    - Gather appropriate statistics (e.g., in order to ultimately calculate average customer waiting times)
    - Add one to the number of customers processed
    - Make server busy
    - Return to main program
  - If busy,
    - Add one to number in queue.
    - If queue is full, panic (or do something about it)
    - If queue isn't full, store customer arrival time
    - Return to main program

Details on Depature Event

- Is the queue empty?
  - If empty,
    - Make server idle
    - Eliminate departure event from consideration of being the next event
    - Return to main program
  - If not empty,
    - Subtract one from number in queue
    - Compute delay of customer entering service and gather statistics
    - Add one to the number of customers delayed (processed)
    - Schedule departure for this customer
    - Move each remaining customer up in the queue
    - Return to main program

Putting together and coding the previous three charts (and how they interact with each other) gives us a nice but tedious event-scheduling solution. But this is really quite a mess!

Luckily, to accomplish this in a process-interaction language (like Arena), all you have to do is:

- `Create` customers every once in a while
- `Process` (serve) them, maybe after waiting in line
- `Dispose` of the customers after they're done being processed

This is soooo easy...

Process-Interaction Approach. We use this approach in class. Concentrate on a generic customer (entity) and the sequence of events and activities it undergoes as it progresses through the system. At any time, the system may have many customers interacting with each other as they compete for resources. You do the *generic* customer modeling in this approach, but you don't deal with the event bookkeeping — the simulation language handles the event scheduling deep inside for all of the generic customers. Saves lots of programming effort.

Example: A customer is generated, eventually gets served, and then leaves. `CREATE-PROCESS-DISPOSE`. Easy, Easy, Easy!

Arena or any other good simulation language adopts this approach.

Outline

- More than 100 commercial languages out there.
  - Examples (from low- to high-level): FORTRAN, SIMSCRIPT, GPSS/H, Extend, Arena, FlexSim, Simio, AnyLogic, Automod.
  - 5–10 major players at schools
  - Huge price range from <$1,000 to $100,000
- Freeware
  - Several nice packages in Java, Python (e.g., SimPyl).
  - Little bit higher learning curve, but not too bad.
- When selecting a language, you should take into account:
  - Cost considerations: $$, learning curve, programming costs, run-time costs, etc.
  - Ease of learning: documentation, syntax, flexibility
  - World view: E-S, P-I, continuous models, combination
  - Features: random variate generators, statistics collection, debugging aids, graphics, user community
- Where to learn simulation languages?
  - Here
  - Conferences such as the Winter Simulation Conference
  - Vendor short courses