Timed Discrete Event Modelling and Simulation

- extend State Automata with “time in state”
- equivalent to Event Graphs “time to transition”

⇒ schedule events
(timed) Discrete Event Models

Finite State Automaton

Event (Scheduling) Graph

McGill, October, 2001
hv@cs.mcgill.ca
CS 308-522A Modelling and Simulation
Discrete Event Modelling and Simulation

- Model: objects and relationships among objects
- Object: characterized by attributes to which values can be assigned
- Attributes:
  - indicative
  - relational
- Values: of a type
Time and State Relationships

- Indexing Attribute: enables state transitions
  Time is most common.

- Instant: value of System Time at which the value of at least one attribute
  of an object can be assigned.

- Interval: duration between two successive instants.

- Span: contiguous succession of one or more intervals.

- State of an object: enumeration of all attribute values at a particular
  instant.

- State of the system: all object states at a particular instant.
Single Server Queueing System

**Physical View**

- **Arrival**
- **Queue**
- **Cashier**
- **Departure**

**Abstract View**

- **Arrival**
  - [IAT distribution]
- **Queue**
- **Cashier**
  - [ST distribution]
Queueing System State Trajectory

state = queue_length x cashier_state

Input Events
Arrival

Output Events
Departure

queue_length

cashier_state

Busy
Idle

state =
queue_length x cashier_state
Time and State Relationships

- Activity: state of an object over an interval
- Event: change in object state, occurring at an instant.
  Initiates an activity
  - Determined: occurrence based on time ("time event")
  - Contingent: based on system conditions ("state event")
- Object activity: state of object between two events for that object
- Process: succession of states of object over a span
Event/Object Activity/Process
Event Scheduling

- Identify *objects* and their *attributes*
- Identify *system attributes* (global)
- Define what causes *changes* in attribute value as *event*
- Write *event routine* for each event:
  - *modify state* (attributes)
  - *schedule* event(s) at \( t + \Delta t, \Delta t \geq 0 \)
- Priorities for *tie-breaking*
- Event scheduling logic
Cashier-queue Event Scheduling Model
declare variables:
    t : Time
    queue_length : PosInt
    cashier_state : \{Idle, Busy\}

declare events:
    start, arrival, departure, end

define events:

    start event:
    /* scheduled first automatically by simulator */

    /* initializations */
    queue_length   = 0
    cashier_state  = Idle

    /* schedule end of simulation */
    schedule end absolute end_time
/* schedule first arrival */
schedule arrival relative 0

arrival event:
schedule arrival relative Random(IATmean, IATspread)
if (queue_length == 0)
  if (cashier_state == Idle)
    cashier_state = Busy
    schedule departure relative Random(SERVmean, SERVspread)
  else
    queue_length++
else /* queue_length != 0 */
  queue_length++

departure event:
if (queue_length == 0)
  cashier_state = Idle
else /* queue_length != 0 */
  queue_length--
schedule departure relative Random(SERVmean, SERVspread)
end event:
/* terminates simulation */
/* process/output performance metrics */
print time, queue_length /* current */
print average_queue_length
Event Scheduling Kernel

- **start**
  - initializations (schedule "start" event)
  - time flow mechanism: select next event from event list
  - Event List: [(ev1,t1),(ev2,t2), ...]
  - time
  - state variables; performance variables

- event routine 1
  - event routine k
  - event routine k+1

- event routine "end"
  - output performance metrics; cleanup;

- end
Input Generation

A “model” of input (sequence of Inter Arrival Times):

- Trace driven
- Auto generating (bootstrapping)
Cashier-queue Event List

<table>
<thead>
<tr>
<th>time</th>
<th>Event List</th>
<th>State set: queue_size x cashier_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0, Idle)</td>
<td>Initialized to: empty queue idle cashier arrival pre-scheduled at time 10</td>
</tr>
</tbody>
</table>

- Process current event:
  - set time to current event time
  - update state:
    - cashier busy
    - queue length increases
  - schedule next arrival at t+IAT()
  - schedule departure at t+ST()
  - remove current event from list

| 10   | (0, Busy) | 1020 |

- Process current event:
  - set time to current event time
  - update state:
    - cashier remains busy
    - queue length increases
  - schedule next arrival at t+IAT()
  - remove current event from list

| 20   | (1, Busy) | |

- Process current event:
  - set time to current event time
  - generate departure output
  - update state:
    - cashier remains busy
    - customer from queue
    - queue length decreases
  - schedule departure at t+ST()
  - remove current event from list

| 30   | (0, Busy) | |
Queueing System State Trajectory

Input Events
Arrival

Output Events
Departure

queue_length

cashier_state

state = queue_length \times cashier_state
Termination Conditions

- Empty Event List
  Need to stop generating arrivals after $t_{end}$ when auto-generating arrivals

- Schedule Termination Event
  - process statistics
  - cleanup
  - stop
  - *caveat*: process all final events!
    - use reserved priority
    - re-schedule

- Similarly: schedule initialization/setup
Event Scheduling (dis)advantages

- advantage: run-time efficient
- disadvantage: hard to understand model
Activity Scanning (rule-based)

Activity:

- condition: must be satisfied for activity to take place. Becomes true \textit{only} at event times.

- actions: operations performed when condition becomes true

Time-advance mechanism:

- fixed time-step

Also known as Two Phase Approach
Cashier-queue Activity Scanning Model
declare (and initialize) variables:
   t            : Time
queue_length  : PosInt = 0
cashier_state : {Idle, Busy} = Idle
t_arrival     : Time = 0
t_depart      : Time = plusInf

declare activities:
   queue_pay, depart, end

queue_pay activity
condition: t >= t_arrival
actions:
   if (queue_length == 0)
      if (cashier_state == Idle)
         keep queue_length == 0
         cashier_state = Busy
         t_depart = t + Random(SERVmean, SERVspread) /* service time */
      else
         queue_length++
else /* queue_length != 0 */
    queue_length++, keep cashier_state == Busy
    t_arrival = t + Random(IATmean, IATspread) /* inter arrival time */

depart activity
condition: t >= t_departure
actions:
    if (queue_length == 0)
        cashier_state = Idle
    else /* queue_length != 0 */
        queue_length--, keep cashier_state == Busy
        t_depart = t + Random(SERVmean, SERVspread) /* service time */

end activity
condition: t >= t_end
actions:
    print t, queue_length /* current */
    print avg_queue_length /* performance metric */
Activity Scanning

start

initializations

time flow mechanism: discrete time step

activity scan

state variables; performance variables

discrete time variable

activity 1
condition
actions

activity k
condition
actions

activity k+1
condition
actions

activity "end"
condition
output performance metrics; cleanup;

end
Activity Scanning (dis)advantages

- advantage: declarative model

- disadvantages:
  - inaccurate if changes occur in between time-steps
  - run-time inefficient (fixed time-step)
Three Phase Approach

- **Bound to occur activities**: unconditional state changes. Pre-scheduled.
- **Conditional activities**
Three Phase Approach

start
initializations
time flow mechanism: select earliest time on EL
Event List (EL): [(activity B1, t1), (activity B2, t2), ...]
execute all B activities on EL due now
state variables; performance variables
A phase
B phase

activity scan

C phase

activity C1
condition
actions
activity Ck
condition
actions
activity C(k+1)
condition
actions

activity B1
... actions
activity Bk
... actions
activity B(k+1)
... actions

activity C"end"
... condition
actions
... performance metrics; cleanup

activity B"end"
... actions

end

McGill, October, 2001
hv@cs.mcgill.ca
CS 308-522A Modelling and Simulation
Three Phase Approach (dis)advantages

- advantage: performance added to Activity Scanning
- disadvantage: mixing two views
Process Interaction

start

Initializations

Clock Update Phase

- current time = move time of first xact on Future Events List (FEL)
- transfer all xacts with move-time = current time to the Current Events List; order by priority

Scan Phase

- move next object on CEL through as far as possible through its process description

Y more xacts to move ?

N terminate simulation ?

Y output performance metrics; cleanup;

end

FEL: [xact1, x1ct2, ...]

increasing move-time

Increasing move-time

CEL

xact1'
xact2'
xact3'

... more xacts to move ?

... terminate simulation ?

output performance metrics; cleanup;

end

Process Interaction model
(e.g., GPSS block diagram)

GENERATE 10, 5
QUEUE wait
SEIZE cashier
DEPART wait
ADVANCE 5, 3
RELEASE cashier
TERMINATE 1
Process Interaction: Transaction Life

Transaction Creation

Y

IAT=0

N

CEL

ADVANCE

FEL

move time = clock time

TERMINATE

Transaction Destruction
Cashier-Queue: GPSS Process Interaction View

GENERATE 10, 5
QUEUE wait
SEIZE cashier
DEPART wait
ADVANCE 5, 3
RELEASE cashier
TERMINATE 1
Process Interaction (dis)advantages

- advantage: declarative model, high-level “process view”
- disadvantage: rather inefficient
General disadvantages

- (here) not formally defined, is possible
- non-modular, is possible

⇒ DEVS formalism
World Views: Classification

“Discrete” Formalisms

Discrete Time Formalisms
- Activity Scanning
- Difference Equations
- Finite State Automata

Discrete Event Formalisms
- Statecharts
- Process Interaction
- Event Scheduling
- Three Phase Approach

DEVS
(Pseudo-) Random-number Generators

- SYS model is deterministic + random constructs
- randomness ≡ not enough detail known or don’t care
- randomness: characterized by distribution
- In SYS: draw from distribution and
  Monte-Carlo run multiple deterministic simulations.
- Alternatives:
  - Transform to deterministic.
  - Markov Chains (analytical).
Probability Distributions

- Continuous vs. discrete
- Probability Density Function ($f(x)$)
- Cumulative Probability Function ($F(X)$)
- see probability course: Poisson, Erlang, …
Pseudo-random

• Sample from distribution \( U(0, 1) \)

• Reproducability/comparison of experiments!
  – science needs reproducible results
  – makes debugging easier
  – \textit{identical} random numbers to compare \textit{different} systems

• Quality of generator:
  – appear uniformly distributed
  – non-correlated
  – fast and doesn’t need much storage
  – long period, dense (full) coverage
  – provision for \textit{streams} (subsegments)
Linear Congruential Generators

\[ Z_i = (aZ_{i-1} + c) \mod m \]

- \( m \) is modulus
- \( a \) is multiplier
- \( c \) is increment
- \( Z_0 \) is seed
- \( c = 0 \) is called \textit{multiplicative} LCG
Generators ctd.

- Composite Generators
- Tausworthe generators (operate on bits)
- L’Ecuyer, Devroye (non-uniform)
- Testing RNG: empirical vs. theoretical
- References: Knuth, Law & Kelton
Marse and Roberts’ portable RNG

\[ Z[i] = (630360016 \times Z[i - 1]) \mod (2^{31} - 1) \]

- Prime modulus multiplicative linear congruential generator.
- Based on Fortran UNIRAN code.
- Multiple (100) streams are supported with seeds spaced 100,000 apart.
- Include file: rand.h
- C file: rand.c
- Example use: randtest.c
Non-uniform continuously distributed RNG

Inverse Transformation Method
Gathering Statistics (report generation)

1. counters
2. summary measures
3. utilization
4. occupancy
5. distributions and transit times
Counters

In all previous examples: keep/update counters (as state vars)!

- numbers of entities of different types in the system
- number of times a particular event occurred
- basis for statistics (performance metrics)
Summary Measures

- minima and maxima:
  compare new values to current min and max, update when necessary

- mean of a set of $N$ observations $x_i, i = 1, 2, \ldots, N$

$$m = \frac{1}{N} \sum_{i=1}^{N} x_i$$
Summary Measures (ctd.)

- standard deviation (from mean)

\[
s = \sqrt{\frac{1}{N - 1} \sum_{i=1}^{N} (m - x_i)^2}
\]

- need to calculate \( m \) first → need to keep all observations
- sum of squares may grow very large (accuracy ↓)

\[
\sum_{i=1}^{N} (m - x_i)^2 = \sum_{i=1}^{N} x_i^2 - Nm^2
\]
Utilization

The fraction (or %) of time each *individual* entity is engaged

\[
U = \frac{1}{t_{end} - t_{start}} \sum_{i=1}^{N} (t_e - t_b)_i
\]
Average Use and Occupancy

for groups and classes of entities
Average Use and Occupancy (ctd.)

- Average use over time ($t_i$ are times of change)

\[
A = \frac{1}{t_{end} - t_{start}} \sum_{i=1}^{N} n_i(t_{i+1} - t_i)
\]

Example use: average queue length.

- Occupancy: average number in use with respect to $MAX$

\[
O = \frac{A}{MAX}
\]

No bookkeeping of individual entity information required, only total use ($n_i$) and when change occurs. This, as opposed to for example average transit time computation where individual times must be kept.
Distributions and Transit Times

Number of intervals $N$, Uniform interval size $\Delta$, Lower tabulation limit $L$.
Implementation: table of interval *counters*.
Global accumulation: number of entries, sum of entries, sum of squares.
Distributions and Transit Times (ctd.)

- Transit times: use clock as *time stamp*, enter in table at end of transit.
- Distribution of number of entities: measure at uniform intervals of time.