

CS 475/575
 Slide set 2

M. Overstreet
 Old Dominion University
 Spring 2005

Simulation Engine

Any sim. lang. (Arena, GPSS, CSIM18, Simula) has simulation engine:

Basic services:

- Manage (set) the simulation clock. sim. time is readable by user, but usually cannot be changed
- Advance time (again normally cannot be done by user)
- Ensure actions occur when they should:
 - time-based: some actions are "scheduled" to occur at a specified simulation time
 - state-based: some actions should occur whenever some condition is true

sim engine (cont)

These ideas provide both **implementation techniques** and **modeling techniques** (just as representing a system as a discrete event system is also both an implementation and modeling technique)

Some simulation languages emphasize scheduling (time-based), others conditions (state-based)

One of the oldest sim languages, CSL, was designed by Esso (became Exxon) to control an oil refinery.

Flavor:

- if tank is full, shut off pump
- if temp > 90, reduce therm to xxx
- if ship is waiting & tank non-empty & ship not full then ...

sim engine - as implementation

- ⌘ Basic language construct:
 - ☒ `schedule(code, time), as in`

```

                schedule( arrival, clock+exp(5.0) )
            
```

 where **clock** is the current time and **exp** is a neg. exp. random var, arrival is some code that, say, checks to see if a server is avail, etc.
- ⌘ Assume model consists of (and only of):
 - ☒ initial event (initialization), including at least one schedule op
 - ☒ set of event routines (like arrival above) that model what the system does when that event occurs.
- ⌘ So the model consists entirely of events (chunks of code to be run at their scheduled times) and a schedule op (for future events)

(aside - while I'm describing implementation, notice how this is a way of thinking about and specifying system behavior: first identify the events (the points of time when something happens), then ...)

sim engine - imp

- ⌘ each event routine has a name. only thing special (making this a sim) is that events are scheduled to occur (run) at a particular clock time. Events change attributes of objects and perhaps schedule additional events by name and time.
- ⌘ one basic data structure to support this is a priority queue (where event time is priority).
 - ☒ schedule op is really a insert into a list structure ordered by the time at which the event is to occur

sim engine - imp

- ⌘ To do simulation, user writes event routines.
- ⌘ Sim engine:


```

                init clock to 0
                run init event
                while (event list not empty)
                    get next event (routine name and time) // delete from list
                    set clock to event time
                    call routine
                end while
                write report (maybe)
            
```
- ⌘ More helpful sim engines will run replications, also)

sim engine - version 2

⌘ What if the modeler thinks in terms of what causes actions rather than in terms of scheduling actions beforehand?

⌘ CSL (for Control and Simulation Language) view:

Model consists of a collection of activities of the form:

- condition
- initial set of actions
- time delay
- final set of actions

Example (from a port sim)

Condition:

(NumShipsWaiting>0) & (TugAvail) & (NumBerths>0) & (highTide)

InitialActions:

NumShipsWaiting--;
TugAvail = false;
NumBerths--;

TimeDelay:

wait TravelTime;

FinalActions:

TugAvail = true;

sim engine - imp 2

⌘ So each "action" typically consists of two events:

- "contingent" not dependent on time
- "bound" dependent on time

⌘ So two lists:

- contingent
- determined

sim engine - imp 2

```
call init
UNTIL end-of-simulation DO
  // Run all possible contingent events before adv. clock
  UNTIL the contingent events list is empty OR every
  condition fails
    remove the first event whose condition is true
    from the list
    call that code
  END UNTIL
  // Advance the clock
  clock = time of first event on determined events list
  // Run determined events
  remove first event from determined events list
  call that code
END UNTIL
```

⌘ Note: this implementation assumes items are added to contingent events list as needed (how?).

How does algorithm need to be changed if not?

Modeling views

- ⌘ Model consists of collection of **events**, each occurs in zero time and each has to be scheduled before hand (sometimes with 0 delay)
- ⌘ Model consists of collections of **activities**, the beginning of which depends on state (not time) and which normally have a time-based duration.
- ⌘ Model consists of a collection of **processes** each process describing the life cycle of one object.
 - ☒ Each object performs state changes (in 0 time), and may wait for a specified amount of time to pass (**wait**) or until a specified condition is met (**wait until**) then makes more state changes or waits or loops The object does nothing while it is waiting.

Process Example (partial)

```

☒ Customer object:
{ // Enters shops
  Wait ( random( 10 ) ) // shops
  if ( clock < 4:00 )
    wait( random( 5 ) // shops some more if time available
  wait until ( clerk==free )
  clerk = busy
  wait ( random( 2 ) ) // checks out
}
☒ Customer generation object
{// customer creation object
  while ( clock < ent sim time ){
  create customer object
  wait ( random( interarrival time )
  }
    
```

sim engine imp: more services

- ⌘ Some sim prog langs only advance the clock due to schedule ops. Others provide more:
 - ☒ wait(specified amt of time).
 - ☒ how implemented?
 - ☒ wait until (specified condition)
 - ☒ how implemented?
 - naive imp?
 - efficient imp?

(In "olden" times, everybody took the compiler construction course, so a common game was to figure how how a compiler could efficiently implement a clever (but perhaps useless) language construct.

Wirth argued against including any construct in a language unless an efficient implementation was available for fear of "tempting" programmers.)

Implementations

- ⌘ How are events implemented?
 - Already discussed
- ⌘ How are activities implemented?
 - Already discussed
- ⌘ How are processes implemented?
 - Based on Algol co-routines
 - Remember FORTRAN's "computed go-to"?

Living without "wait until"

- ⌘ All (well, lots of) models need a "wait until" and it is not provided in most simulation programming languages.
 - It does exist in Arena and GPSS, but a programmer cannot create arbitrary conditions
 - Did I say: "wait until" and conditions are (almost) the same
 - How's it done anyway?
 - It's really a condition
 - How would the harbor model be implemented in C++ in regards to conditions?

sim. engine - entity flow

- ⌘ Used by Arena, GPSS, many others (influenced by GPSS)
- ⌘ Models consist of **temporary** entities and **permanent** entities. The permanent entities are stationary and the temporary entities move from one permanent entity to the next causing the permanent entity to do something. Sometimes the permanent entities are resources that the temp entities compete for.
 - Think computer network. temp. ent. might be messages, perm. ent. are CPU, switch, coax cable
 - Think manufacturing shop; parts arrive, a processed by machines, etc.
 - Sometimes the temp. entities are failures, the perm entities are machines (or breaks of the machine operator or whatever)
 - Think customers arriving a bank, with tellers, etc.
- ⌘ Temp. entities are passive & mobile.
- ⌘ Perm. entities are active & stationary.

sim. engine - entity flow

- ⌘ Each temp. entity follows a prescribed path (maybe with perm ents. directing their path) through locations which change the state of a perm. entities, advance time, or produce statistics
- ⌘ These "locations" are called statements, blocks, or modules
- ⌘ When temp entity arrives at a block:
 - the perm entity may disallow entrance of the temp entity (so it queues in front of the perm entity)
 - if the temp entity "enters" the perm entity, a set of actions occur based on the type of the module, perhaps dependent on the state of the temp entity.
- ⌘ To advance the clock, some temp entity must enter a time delay module

entity flow sim engine

```

init
till end of simulation
  for each temp entity in system
    until (a complete pass has been made in which
           no entity changes position)
      try to move chosen entity "down its path" until
        1) entity is blocked by some permanent entity
        2) it's in a location which requires a time
           delay (sch time it departs)
      end until
    end for
  pick entity with earliest sch depart time
  advance time to its depart time
  move it till its stuck (condition or time)
end till
    
```

Reading assignment

- ⌘ Read Leemis, Chapter 5, pages 185-194 (though section 5.1.4)
- ⌘ Problems 5.1.3 and 5.1.4 will be due Feb. 9.
