Language-based Discrete-Event Simulators
Key Ideas

In a language-based DES we create objects/libraries/classes that are used within a “normal” program. For example

– Event lists
– Base classes for events
– Statistics gathering/analyzing infrastructure

Long history of embedding simulations in C, C++, and Java
Good News and Bad News

The Good News

– You can implement just about anything you want
– Clearly constrained path to integration with other code systems and libraries

The Bad News

– Anything you want you have to build yourself or use something built by others
– There can be implementation issues where simulation view conflicts with language view, e.g. threads
DES World Views

Two most common DES world views

Event-oriented: Identify points in time where model state changes ("events").

- Create a data structure that describes all variations of events
- For each event define handler code called to execute that event
Event Oriented Paradigm

Initialize event list

while (terminating condition not met) {

• remove event with least time-stamp from list

• Look up event handler for this event
  – execute handler, using parameters in event data structure
    • Execution may create new events and insert into event list

}
Example : G/G/1 queue

Define event data structure to have “time” and “type” {arrival, departure}

Define state of model : number of jobs Q in queue

Arrival event handler :

• sample inter-arrival time, schedule next arrival event
• Increment Q
• If (Q==1) sample service time, schedule next departure
Example G/G/1 queue

Departure event handler
- Update statistics
- Decrement Q
- If(Q>0) then sample service time and schedule next departure event

Easy to imagine embedding in programming language, BUT there are subtleties, e.g. events with the same time-stamp
- imagine the DES for a SAN with immediate transitions
- Writing so as to support modularity
Source Code Simulations
Simulations embedded in Programming Language : Examples

• gg1.cc - G/G/1 example in C++
• Event.java - class for DES event
• EventList.java - splay tree event list
• oo_sim.h, oo_sim.cc - framework for object-oriented simulation
Process Oriented View

Describe evolution of state in terms of processes—events are implicit.

Example: csim models of “airplanes” through an airport.

Note library-specific data structures, implicit blocking.

View is of airplane process:

```c
facility_ms *gates;
...
Void airplaneGenerator() {
    create("gen");
    while(1) {
        airplane();
        hold(exponential(airplaneInterarrivalTm));
    }
}

void airplane() {
    create("airplane");
    gates->use(
        uniform(minGateTm,mxGateTM));
}
```
SSF : Scalable Simulation Framework

API developed 10 years ago for developing simulations with provable parallel performance

Like an assembly language for simulators

Key concepts:

communication channels
  – declared minimal latencies

processes: code statements that suspend them
  – waitFor, waitOn

timelines and alignment
Illustration

Entity

Process

outChannel

inChannel

Event

mapto

Minimal latency graph
SSF Entity

Methods:
void init(), void wrapup();
ltme_t now();
void * alignment();
ltme_t alignto(Entity* s);
ltme_t makeIndependent() 
Entity** coalignedEntities();
Process** processes();
inChannel** inChannels();
outChannel** outChannels();
void startAll(), joinAll()
SSF Process

void action();
Boolean isSimple();
void waitOn(inChannel**); void waitOn(inChannel *);
waitForever();
waitFor(ltime_t);
waitUntil(ltime_t);
boolean waitOnFor(inChannel**, ltime_t);
boolean waitOnFor(inChannel*, ltime_t);
boolean waitOnUntil(inChannel*, ltime_t);
boolean waitOnUntil(inChannel**, ltime_t);
inChannel** activeChannels();
Timer object in SSF

• See Timer.java
Process-orientation Language-based simulators

Key issue: saving and recovering state before and after a statement that suspends the thread of control

e.g., “hold”, “use” in csim
waitFor, waitOn in SSF

Native C/C++ does not support threading, always need some add-on
Options

Use standard threading package, e.g. Posix threads

– Pros: well developed and standard infrastructure
– Cons: pre-declared # threads, pre-declared thread state size. Potentially costly thread-switching cost. Thread switching logic that has to be adapted for simulation
Options

Simulator defined threads

What is the core of the problem? Memory

A threading package will allocate a block of memory on the heap for a thread

But a source-based simulation routine has local variables on the stack
  – function parameters, local variables

csim copies stack onto heap at suspension, restores on re-animation
  – Tricky business, need assembly language for each different supported architecture
Process state in SSF

SSF requires annotations

– Uses these to do a source to source translation

– Process turned a large “switch” statement
  • Each suspension/re-entry point given a label
  • “state” of process includes switch label---controls entry point on execution

– Process state record created on the heap
  • All references to variables on the stack are transformed to references to their implementation on the heap
    – So transformed code does not use stack except for subroutine calls
Example-blocking process in SSFNet

// SSF PROCEDURE
bool BlockingSocketMaster::connect(int sock, IPADDR ip, uint16 port)
{
    Socket* mysocket; // SSF STATE

    { mysocket = 0;
        SSFNET_MAP(int, Socket*)::iterator iter = bound_socks.find(sock);
        if(iter != bound_socks.end()) mysocket = (*iter).second;
        if(!mysocket || // if the socket is not bound
            !mysocket->session || // this shouldn't happen
            mysocket->session->connected() || // if the socket has been connected
            (mysocket->state & Socket::CONNECT_ACTIVE)) // // if the socket is connecting
            return false; }

    SOCK_DUMP(printf("[host="\%s\"] %g: connect(): socket=%d (state=\%08x), ip="\%s", port=%d.\n",
        inHost()->nhi.toString(),
        getNow().second(), sock, mysocket->state, IPPrefix::ip2txt(ip), port));

    // initiate the connection
    if(!mysocket->session->connect(ip, port)) return false;

    // set the connect active flag
    mysocket->state |= Socket::CONNECT_ACTIVE;

    // SSF CALL
    block_till(sock, Socket::OK_TO_SEND|Socket::CONN_RESET, false);
More on implementing process orientation

In current SSF system, files with annotated procedures/state have .cxx extension

– A perl-based source-to-source translator creates an equivalent .cc file

For every annotated procedure a specialization of base class Procedure is created

– An instance of the derived class is a “p-frame”

  • p-frames hold the procedure’s state
  • p-frame constructor is called by the routine that calls the procedure, passing the input arguments
Transformation

Before

// call blocking procedure
action(x,y);

After

// call blocking procedure
pf*= pfconst_action(x,y);
pf->mangled_action();
// check blocking state
If( pf->suspended() ) {
    return;
    _label;
}

See example file language-based-support.txt
p-frame memory is released once the call it was created for returns w/o suspended state
“Simple” Processes

All the aggravation with p-frames is to recover state following a suspension.

What if a process never needed prior state after a suspension???

How could we tell?

What could we do to make use of that?
SSF Simple Process

In SSF the base class for Process contains virtual function boolean is_simple(), which can be defined in derived class to return 1

The scheduler can check a process it will envoke, and if simple, just call the process body.

“State” here includes place in body to return after suspending call
Simple Process Example

//! SSF SIMPLE PROCESS

void action() {
    Event e = new wakeupcall(now());
    room[uniform(0,N)]->write(e);
    waitFor(expon(10.0));
}

Native threads and scheduling

Issue – garden variety threads (posix, java) provide locking mechanisms not immediately suited to simulation

How can we build a thread scheduler for simulation on top of native threads?

Example : Java and SSF
Java threads

Java defines Thread class, derived classes define “execute” method

Each object in Java has a lock (and a method to synchronize on it)

A thread tries to execute a code fragment protected by a lock for object obj by the call

Synchronized(obj) { /* code fragment here */ }  
Only one thread has lock at a time, released on exit from fragment

Methods wait, notify also defined: obj.wait(), obj.notify()
Using Java threads for simulation

Each SSF process derives from Java Thread class
  – Each has built-in native lock, and scheduling one called “sched”
Additional thread for scheduler
Scheduling thread maintains data structures to order the reanimation times of suspended threads
  – Applies notify() to “sched” of the selected process
  – Applies wait() to native lock of selected process
When the reanimated process suspends it first puts a re-animation event in queue, then blocks on itself
Implementation of waitFor

Public void waitFor(long interval) {
    time = owner.owner.clock + interval;
    owner.owner.insertProcess(this); // release time
    if(! isSimple()) {
        synchronized(sched) { // get both locks for self
            synchronized(this) {
                notify(); // this releases scheduler
            }
        }
        try(sched.wait(); } // block on sched lock
        catch(InterruptedException e) {} 
    }}}