Describing Simulations in the Orc Programming Language

David Kitchin, Adrian Quark, Jayadev Misra

Department of Computer Science
University of Texas at Austin

23rd ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS 2009)
June 22-25, 2009
Lake Placid, New York, USA

Simulation as Concurrent Programming

- A simulation description is a real-time concurrent program.
- The concurrent program includes physical entities and their interactions.
- The concurrent program specifies the time interval for activities.

Features needed in the Concurrent Programming Language

- Describe entities and their interactions.
- Describe passage of time.
- Allow birth and death of entities.
- Allow programming novel interactions.
- Support hierarchical structure.

Orc

- Goal: Internet scripting language.
- Next: Component integration language.
- Next: A general purpose, structured “concurrent programming language”.
- A very late realization: A simulation language.

Internet Scripting

- Contact two airlines simultaneously for price quotes.
- Buy a ticket if the quote is at most $300.
- Buy the cheapest ticket if both quotes are above $300.
- Buy a ticket if the other airline does not give a timely quote.
- Notify client if neither airline provides a timely quote.

Orc Basics

- Site: Basic service or component.
- Concurrency combinators for integrating sites.
- Theory includes nothing other than the combinators.

  No notion of data type, thread, process, channel, synchronization, parallelism $\cdots$

  New concepts are programmed using the combinators.

Examples of Sites

- $+\ -\ *\ \&\&\ ||\ <\ =$ ...
- `println`, `random`, `Prompt`, `Email` ...
- `Ref`, `Semaphore`, `Channel`, `Database` ...
- `Timer`
- External Services: Google Search, MySpace, CNN, ...
- Any Java Class instance
- Sites that create sites: `MakeSemaphore`, `MakeChannel` ...
- Humans

  ...

Sites

- A site is called like a procedure with parameters.
- Site returns at most one value.
- The value is published.

Site calls are strict.

Overview of Orc

- Orc program has
  - a goal expression,
  - a set of definitions.

- The goal expression is executed. Its execution
  - calls sites,
  - publishes values.

Structure of Orc Expression

- Simple: just a site call, $CNN(d)$
  Publishes the value returned by the site.

- Composition of two Orc expressions:

do f and g in parallel	$f \mid g$	Symmetric composition
for all x from f do g	$f > x > g$	Sequential composition
for some x from g do f	$f < x < g$	Pruning

Structure of Orc Expression

- Simple: just a site call, $CNN(d)$
  Publishes the value returned by the site.

- Composition of two Orc expressions:

do f and g in parallel	$f \mid g$	Symmetric composition
for all x from f do g	$f >x> g$	Sequential composition
for some x from g do f	$f <x< g$	Pruning

Structure of Orc Expression

- Simple: just a site call, $CNN(d)$
  Publishes the value returned by the site.

- Composition of two Orc expressions:

do f and g in parallel	$f \mid g$	Symmetric composition
for all x from f do g	$f > x > g$	Sequential composition
for some x from g do f	$f < x < g$	Pruning

Structure of Orc Expression

- Simple: just a site call, $CNN(d)$
  Publishes the value returned by the site.

- Composition of two Orc expressions:

do f and g in parallel	$f \mid g$	Symmetric composition
for all x from f do g	$f > x > g$	Sequential composition
for some x from g do f	$f < x < g$	Pruning

Symmetric composition: $f \mid g$

- Evaluate f and g independently.
- Publish all values from both.
- No direct communication or interaction between f and g. They can communicate only through sites.

  Example: $CNN(d) \mid BBC(d)$

  calls both CNN and BBC simultaneously.
  Publishes values returned by both sites. (0, 1 or 2 values)

Sequential composition: $f \; >x> \; g$

For all values published by f do g.
Publish only the values from g.

- $CNN(d) \; >x> \; Email(address, x)$
  - Call $CNN(d)$.
  - Bind result (if any) to x.
  - Call $Email(address, x)$.
  - Publish the value, if any, returned by $Email$.

- $(CNN(d) \mid BBC(d)) \; >x> \; Email(address, x)$
  - May call $Email$ twice.
  - Publishes up to two values from $Email$.

Notation: $f \gg g$ for $f \; >x> \; g$, if x unused in g.

Schematic of Sequential composition

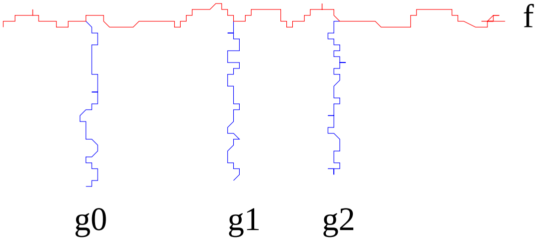

Figure: Schematic of f $>x>$ g

Pruning: $(f \;{<}x{<}\; g)$

For some value published by g do f.

- Evaluate f and g in parallel.
  - Site calls that need x are suspended.
  - see $(M() \mid N(x)) \;{<}x{<}\; g$
- When g returns a (first) value:
  - Bind the value to x.
  - Terminate g.
  - Resume suspended calls.
- Values published by f are the values of $(f \;{<}x{<}\; g)$.

Example of Pruning

$$Email(address, x) \;\; <x< \; (CNN(d) \;\; | \; BBC(d))$$

Binds x to the first value from $CNN(d) \;\; | \; BBC(d)$.
Sends at most one email.

Some Fundamental Sites

- $if(b)$: boolean b,
  returns a signal if b is true; remains silent if b is false.
- $Rtimer(t)$: integer t, $t \geq 0$, returns a signal t time units later.
- $stop$: never responds. Same as $if(false)$.
- $signal$: returns a signal immediately. Same as $if(true)$.

Expression Definition

$$\mathit{def}\ \ MailOnce(a) = $$
$$Email(a, m) \ \ {<}m{<}\ (CNN(d)\ \mid BBC(d))$$

$$\mathit{def}\ \ MailLoop(a, t) = $$
$$MailOnce(a) \ \gg\ Rtimer(t) \ \gg\ MailLoop(a, t)$$

$$\mathit{def}\ \ metronome() = \ signal\ \mid (Rtimer(1) \gg metronome())$$
$$metronome() \gg stockQuote()$$

- Expression is called like a procedure.
  It may publish many values. *MailLoop* does not publish.
- Site calls are strict; expression calls non-strict.

Functional Core Language

- Data Types: Number, Boolean, String, with usual operators
- Conditional Expression: if E then F else G
- Data structures: Tuple and List
- Pattern Matching
- Function Definition; Closure

Variable Binding; Silent expression

$val\ \ x = \ 1 + 2$

$val\ \ y = \ x + x$

$val\ \ z = \ x/0$ - - expression is silent

$val\ \ u = \ \ \text{if } (0 < 5) \text{ then } 0 \text{ else } z$

Comingling with Orc expressions

Components of Orc expression could be functional.
Components of functional expression could be Orc.

$$(1 + 2) \mid (2 + 3)$$

$$(1 \mid 2) + (2 \mid 3)$$

Convention: whenever expression F appears in context C where a single value is expected from F, convert it to $C[x] <x< F$.

$1 + 2 \mid 2 + 3$ is $add(1, 2) \mid add(2, 3)$

$(1 \mid 2) + (2 \mid 3)$ is $(add(x, y) <x< (1 \mid 2)) <y< (2 \mid 3)$

Example: Fibonacci numbers

$$
\begin{aligned}
&\textit{def}\;\; H(0) = (1,1)\\
&\textit{def}\;\; H(n) = H(n-1) \; >(x,y)> \; (y, x+y)
\end{aligned}
$$

$$\textit{def}\;\; Fib(n) = H(n) \; >(x,\_)> \; x$$

{- Goal expression -}

$Fib(5)$

Some Typical Applications

- Adaptive Workflow (Business process management):
  Workflow lasting over months or years
  Security, Failure, Long-lived Data

- Extended 911:
  Using humans as components
  Components join and leave
  Real-time response

- Network simulation:
  Experiments with differing traffic and failure modes
  Animation

Some Typical Applications, contd.

- Grid Computations
- Music Composition
- Traffic simulation
- Computation Animation

Some Typical Applications, contd.

- Map-Reduce using a server farm
- Thread management in an operating system
- Mashups (Internet Scripting).
- Concurrent Programming on Android.

Time-out

Publish M's response if it arrives before time t,
Otherwise, publish 0.

$$z \ {<}z{<} \ (M() \ | \ (Rtimer(t) \ \gg 0)),$$ or

$$\begin{array}{l} val \ z = M() \ | \ (Rtimer(t) \ \gg 0) \\ z \end{array}$$

Fork-join parallelism

Call M and N in parallel.
Return their values as a tuple after both respond.

$$
\begin{array}{l}
((u, v) \\
\qquad < u < M()) \\
\qquad < v < N()
\end{array}
$$

or,

$$(M(), N())$$

Recursive definition with time-out

Call a list of sites simultaneously.
Count the number of responses received within 10 time units.

$$
\begin{aligned}
&\mathit{def}\ \ \mathit{tally}([\,]) = 0 \\
&\mathit{def}\ \ \mathit{tally}(M : MS) = (M() \gg 1 \mid \mathit{Rtimer}(10) \gg 0) + \mathit{tally}(MS)
\end{aligned}
$$

Barrier Synchronization in $M() \gg f \mid N() \gg g$

f and g start only after both M and N complete.
Rendezvous of CSP or CCS; M and N are complementary actions.

$$(M(), N()) \gg (f \mid g)$$

Priority

- Publish N's response asap, but no earlier than 1 unit from now. Apply fork-join between $Rtimer(1)$ and N.

$$val \ \ (u, \_) = \ (N(), Rtimer(1))$$

- Call M, N together. If M responds within one unit, publish its response. Else, publish the first response.

$$val \ \ x = \ M() \ \mid u$$

Mutable Structures

$\text{val } r = Ref()$

$r.write(3)$, or $r := 3$
$r.read()$, or $r?$

$def \;\; swapRefs(x, y) = \;\; (x?, y?) \;\; >(xv, yv)> \;\; (x := yv, y := xv)$

Binary Search Tree; Pointer Manipulation

$def\ \ search(key) =$ -- return true or false
$\quad searchstart(key)\ >(\_,\_,q)>\ (q \neq null)$

$def\ \ insert(key) =$ -- true if value was inserted, false if it was there
$\quad searchstart(key)\ >(p,d,q)>$
$\quad \text{if}\ \ q = null$
$\quad\quad \text{then}\ \ Ref()\ >r>$
$\quad\quad\quad r := (key, null, null)\ \gg update(p,d,r)\ \gg true$
$\quad\quad \text{else}\ \ false$

$def\ \ delete(key) =$

Semaphore

$val\ \ s = \ Semaphore(2)$ - - s is a semaphore with initial value 2

$s.acquire()$
$s.release()$

Rendezvous:

$val\ \ s = \ Semaphore(0)$
$val\ \ t = \ Semaphore(0)$

$def\ \ send() = \quad t.release() \gg s.acquire()$
$def\ \ receive() = t.acquire() \gg s.release()$

n-party Rendezvous using $2(n-1)$ semaphores.

Readers-Writers

$$\mathit{val}\ \ \mathit{req} = \ \mathit{Buffer}()$$
$$\mathit{val}\ \ \mathit{cb} = \ \mathit{Counter}()$$

$$\begin{array}{l}
\mathit{def}\ \ \mathit{rw}() = \\
\quad \mathit{req.get}()\ \ >(b, s)> \\
\qquad (\ \ \mathit{if}(b) \quad \gg\ \mathit{cb.inc}() \qquad \gg \mathit{s.release}()\ \gg \mathit{rw}() \\
\qquad \ \mid \mathit{if}(\neg b) \ \gg\ \mathit{cb.onZero}() \ \gg \\
\qquad \quad \mathit{cb.inc}()\ \gg\ \mathit{s.release}() \quad \gg \mathit{cb.onZero}()\ \gg \mathit{rw}() \\
\qquad)
\end{array}$$

$$\begin{array}{l}
\mathit{def}\ \ \mathit{start}(b) = \\
\quad \mathit{val}\ \ s = \ \mathit{Semaphore}(0) \\
\quad \mathit{req.put}((b, s))\ \gg \mathit{s.acquire}()
\end{array}$$

$$\mathit{def}\ \ \mathit{quit}() = \ \mathit{cb.dec}()$$

Shortest path problem

- Directed graph; non-negative weights on edges.
- Find shortest path from source to sink.

We calculate just the length of the shortest path.

Algorithm with Lights and Mirrors

- Source node sends rays of light to each neighbor.
- Edge weight is the time for the ray to traverse the edge.
- When a node receives its first ray, sends rays to all neighbors. Ignores subsequent rays.
- Shortest path length = time for sink to receive its first ray.

Algorithm

$$
\begin{array}{ll}
\textit{def}\ \ eval(u,t) = & \text{if } t \text{ is the first value for } u\text{, record it else stop} \gg \\
& \text{for every edge } (u,v) \text{ of length } d \text{ do} \\
& \text{wait for } d \text{ time units} \gg \\
& eval(v, t+d) \\
\\
\textit{Goal}: & eval(source, 0) \mid \\
& \text{read the value recorded for the } sink
\end{array}
$$

record and read sites

$write(u, t)$: Write value t for node u. If already written, block.

$read(u)$: Return value for node u. If unwritten, block.

Graph Structure: Function $Succ()$

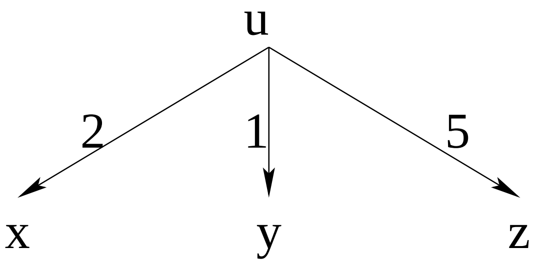

Figure: Graph Structure

$Succ(u)$ publishes $(x, 2)$, $(y, 1)$, $(z, 5)$.

Algorithm(contd.)

$$\mathit{def}\ \ \mathit{eval}(u,t) = \quad \text{if } t \text{ is the first value for } u\text{, record it else stop} \gg$$
$$\text{for every edge } (u,v) \text{ of length } d \text{ do}$$
$$\text{wait for } d \text{ time units} \gg$$
$$\mathit{eval}(v, t+d)$$

$$\mathit{Goal}: \quad \mathit{eval}(\mathit{source}, 0) \mid$$
$$\text{read the value recorded for the } \mathit{sink}$$

$$\mathit{def}\ \ \mathit{eval}(u,t) = \quad \mathit{write}(u,t) \gg$$
$$\mathit{Succ}(u) > (v,d) >$$
$$\mathit{Rtimer}(d) \gg$$
$$\mathit{eval}(v, t+d)$$

$$\mathit{Goal}: \quad \mathit{eval}(\mathit{source}, 0) \mid \mathit{read}(\mathit{sink})$$

Algorithm(contd.)

$$\begin{aligned} \textit{def}\ \ eval(u,t) = \quad & write(u,t) \gg \\ & Succ(u)\ >(v,d)> \\ & Rtimer(d) \gg \\ & eval(v, t+d) \end{aligned}$$

$$\textit{Goal}: \quad eval(source, 0) \mid read(sink)$$

- Any call to $eval(u,t)$: Length of a path from source to u is t.
- First call to $eval(u,t)$: Length of the shortest path from source to u is t.
- $eval$ does not publish.

Drawbacks of this algorithm

- Running time proportional to shortest path length.
- Executions of *Succ*, *read* and *write* should take no time.

Solution: Replace calls to Real-timer by calls to Logical-timer.

$$
\begin{array}{ll}
\mathit{def}\ \ \mathit{eval}(u,t) = & \mathit{write}(u,t) \gg \\
& \mathit{Succ}(u)\ \ >(v,d)> \\
& \mathit{Ltimer}(d) \gg \\
& \mathit{eval}(v, t+d)
\end{array}
$$

$$
\mathit{Goal}: \qquad \mathit{eval}(\mathit{source}, 0) \ \mid\ \mathit{read}(\mathit{sink})
$$

Logical Timer

Methods:

$Ltimer(t)$	Returns a signal after t logical time units.
$Ltimer.time()$	Returns the current value of the logical timer.

Logical timer Implementation

Must guarantee:

- $Ltimer(t)$ consumes exactly t units of logical time.
- No other site call consumes logical time once its execution starts (its execution may depend on site calls that consume time).
- Logical timer is advanced only if there can be no other activity.

Examples

- $$Rtimer(10) \mid Ltimer(2)$$
  Should logical timer be advanced with passage of real time?

- $$Rtimer(10) \gg c.put(5) \mid Ltimer(2)$$
  Does $Rtimer(10) \gg c.put(5)$ consume logical time?

- $$c.get() \mid Ltimer(2) \gg c.put(5)$$
  What are the values of $Ltimer.time()$ before and after $c.get()$?

- $$stop \mid Ltimer(2)$$
  Can the logical timer be advanced?

- $$Google() \mid Ltimer(2)$$
  Advance logical timer while waiting for $Google()$ to respond?
  What if $Google()$ never responds?

Implementing logical timer

Data structures:

- n: current value of $Ltimer.time()$, initially $n = 0$.
- q: queue of calls to $Ltimer()$ whose responses are pending.

At run time:

- A call to $Ltimer.time()$ immediately responds with n.
- A call to $Ltimer(t)$ is assigned rank $n + t$ and queued.
- Progress: If the program is stuck without advancing the logical time, then:
  remove the item with lowest rank r from q,
  set $n := r$,
  respond with a signal to the corresponding call to $Ltimer()$.

Simulation: Bank

- Bank with two tellers and one queue for customers.
- Customers generated by a *source* process.
- When free, a teller serves the first customer in the queue.
- Service times vary for customers.
- Determine
  - Average wait time for a customer.
  - Queue length distribution.
  - Average idle time for a teller.

Structure of bounded simulation

Run the simulation for $simtime$.
Below, $Bank()$ never publishes .

$$val\ \ z = \ Bank() \ \mid Ltimer(simtime)$$

$$z \ \gg\ Stats()$$

Description of Bank

$$
\begin{aligned}
&\mathit{def}\ \mathit{Bank}() &&= (\mathit{Customers}() \mid \mathit{Teller}() \mid \mathit{Teller}()) \gg \mathit{stop}\\
&\mathit{def}\ \mathit{Customers}() &&= \mathit{Source}()\ >c>\ \mathit{enter}(c)\\
&\mathit{def}\ \mathit{Teller}() &&= \mathit{next}()\ >c>\\
& && \quad \mathit{Ltimer}(c.\mathit{ServTime}) \gg\\
& && \quad \mathit{Teller}()\\
&\mathit{def}\ \mathit{enter}(c) &&= q.\mathit{put}(c)\\
&\mathit{def}\ \mathit{next}() &&= q.\mathit{get}()
\end{aligned}
$$

Fast Food Restaurant

- Restaurant with one cashier, two cooking stations and one queue for customers.
- Customers generated by a *source* process.
- When free, cashier serves the first customer in the queue.
- Cashier service times vary for customers.
- Cashier places the order in another queue for the cooking stations.
- Each order has 3 parts: main entree, side dish, drink
- A cooking station processes parts of an order in parallel.

Goal Expression for Restaurant Simulation

$$val \;\; z = \; Restaurant()() \;\; | \; Ltimer(simtime)$$

$$z \;\; \gg \; Stats()$$

Description of Restaurant

$$
\begin{array}{lll}
\mathit{def}\ \mathit{Restaurant}() & = & (\mathit{Customers}()\ \mid \mathit{Cashier}()\ \mid \mathit{Cook}()\ \mid \mathit{Cook}()) \gg \mathit{stop} \\
\mathit{def}\ \mathit{Customers}() & = & \mathit{Source}()\ >c>\ \mathit{enter}(c) \\
\mathit{def}\ \mathit{Cashier}() & = & \mathit{next}()\ >c> \\
 & & \mathit{Ltimer}(c.\mathit{ringupTime}) \gg \\
 & & \mathit{orders.put}(c.\mathit{order}) \gg \\
 & & \mathit{Cashier}() \\
\mathit{def}\ \mathit{Cook}() & = & \mathit{orders.get}()\ >\mathit{order}> \\
 & & (\\
 & & \quad \mathit{prepTime}(\mathit{order.entree})\ >t>\ \mathit{Ltimer}(t), \\
 & & \quad \mathit{prepTime}(\mathit{order.side})\ >t>\ \mathit{Ltimer}(t), \\
 & & \quad \mathit{prepTime}(\mathit{order.drink})\ >t>\ \mathit{Ltimer}(t) \\
 & &)\ \gg\ \mathit{Cook}() \\
\mathit{def}\ \mathit{enter}(c) & = & q.\mathit{put}(c) \\
\mathit{def}\ \mathit{next}() & = & q.\mathit{get}()
\end{array}
$$

Collecting Statistics: waiting time

Change

$$
\begin{array}{lcl}
\mathit{def}\ \mathit{enter}(c) & = & q.put(c) \\
\mathit{def}\ \mathit{next}() & = & q.get()
\end{array}
$$

to

$$
\begin{array}{lcl}
\mathit{def}\ \mathit{enter}(c) & = & Ltimer.time()\ >s>\ q.put(c, s) \\
\mathit{def}\ \mathit{next}() & = & q.get()\ >(c, t)> \\
 & & \quad Ltimer.time()\ >s> \\
 & & \quad reportWait(s - t)\ \gg \\
 & & \quad c
\end{array}
$$

Stopwatch

A stopwatch is aligned with some timer, real or virtual.
Supports 4 methods:

- reset
- read
- start
- stop

Histogram: Queue length

- Create $N+1$ stopwatches, $sw[0..N]$, at the beginning of simulation.
- Final value of $sw[i]$, $0 \leq i < N$, is the duration for which the queue length has been i.
- $sw[N]$ is the duration for which the queue length is at least N.
- On adding an item to queue of length i, $0 \leq i < N$, do

$$sw[i].stop \mid sw[i+1].start$$

- After removing an item if the queue length is i, $0 \leq i < N$, do

$$sw[i].start \mid sw[i+1].stop$$

Simulation Layering

- A simulation is written a set of layers.
- Lowest layer represents the abstraction of the physical system.
- Next layer may collect statistics, by monitoring the layer below it.
- Further layers may produce reports and animations from the statistics.