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 …

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:

  \[
  \begin{align*}
  \text{do } f \text{ and } g \text{ in parallel} & \quad f \mid g \quad \text{Symmetric composition} \\
  \text{for all } x \text{ from } f \text{ do } g & \quad f \triangleright x \triangleright g \quad \text{Sequential composition} \\
  \text{for some } x \text{ from } g \text{ do } f & \quad f \triangleleft x \triangleleft g \quad \text{Pruning}
  \end{align*}
  \]
Structure of Orc Expression

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

- **Composition** of two Orc expressions:
  
  \[
  \text{do } f \text{ and } g \text{ in parallel } \quad f \mid g \quad \text{Symmetric composition}
  \]
  
  \[
  \text{for all } x \text{ from } f \text{ do } g \quad f > x > g \quad \text{Sequential composition}
  \]
  
  \[
  \text{for some } x \text{ from } g \text{ do } f \quad f < x < g \quad \text{Pruning}
  \]
Structure of Orc Expression

- **Simple**: just a site call, $\textit{CNN}(d)$
  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:
  \[
  \text{do } f \text{ and } g \text{ in parallel } \quad f \parallel g \quad \text{Symmetric composition}
  \]
  \[
  \text{for all } x \text{ from } f \text{ do } g \quad \text{for some } x \text{ from } g \text{ do } f \quad \text{Sequential composition}
  \]
  \[
  \text{for some } x \text{ from } g \text{ do } f \quad f < x < g \quad \text{Pruning}
  \]
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: \( f \gg x \gg g \)

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

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

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

**Notation:** \( f \gg g \) for \( f \gg x \gg g \), if \( x \) unused in \( g \).
Schematic of Sequential composition

Figure: Schematic of $f \circ x \circ g$
Pruning: \((f \prec x \prec g)\)

For some value published by \(g\) do \(f\).

- Evaluate \(f\) and \(g\) in parallel.
  - Site calls that need \(x\) are suspended.
  - see \((M()) | N(x)) \prec x \prec g\)

- When \(g\) returns a (first) value:
  - Bind the value to \(x\).
  - Terminate \(g\).
  - Resume suspended calls.

- Values published by \(f\) are the values of \((f \prec x \prec g)\).
Example of Pruning

\[ Email(address, x) \; \langle x \rangle \; (CNN(d) \; | \; BBC(d)) \]

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

```
def MailOnce(a) =
    Email(a, m) < m < (CNN(d) | BBC(d))

def MailLoop(a, t) =
    MailOnce(a) ≫ Rtimer(t) ≫ MailLoop(a, t)
```

```
def metronome() = signal | (Rtimer(1) ≫ metronome())
metronome() ≫ 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**
val x = 1 + 2

val y = x + x

val z = x/0 -- expression is silent

val u = if (0 < 5) then 0 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 \( C[x] <x<F \).

\[
1 + 2 \mid 2 + 3 \quad \text{is} \quad add(1, 2) \mid add(2, 3)
\]

\[
(1 \mid 2) + (2 \mid 3) \quad \text{is} \quad (add(x, y) <x<(1 \mid 2)) <y<(2 \mid 3)
\]
Example: Fibonacci numbers

```python
def H(0) = (1, 1)
def H(n) = H(n - 1) >(x, y)> (y, x + y)

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.
Publish $\mathcal{M}$’s response if it arrives before time $t$, Otherwise, publish 0.

\[
z < (\mathcal{M}() \mid (\text{Rtimer}(t) \gg 0)), \text{ or }
\]

\[
\text{val } z = \mathcal{M}() \mid (\text{Rtimer}(t) \gg 0)
\]

\[
z
\]
Fork-join parallelism

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

$$(u, v) \begin{cases} <u < M() \end{cases} <v < N()$$

or,

$$(M(), N())$$
Recursive definition with time-out

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

\[
\begin{align*}
\text{def } \text{tally}([\ ]) &= 0 \\
\text{def } \text{tally}(M : MS) &= (M() \gg 1 | \text{Rtimer}(10 \gg 0) + \text{tally}(MS)
\end{align*}
\]
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.

\[
val x = M() | u
\]
Mutable Structures

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

\( r\text{.write}(3) \), or \( r := 3 \)
\( r\text{.read()} \), or \( r? \)

\[
\text{def } \text{swapRefs}(x, y) = (x?, y?) \gg (xv, yv) \gg (x := yv, y := xv)
\]
def search(key) = -- return true or false
searchstart(key) >(_, _, q)> (q ≠ null)

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

def delete(key) =
Semaphore

\[ \text{val } s = \text{Semaphore}(2) \quad \text{-- } s \text{ is a semaphore with initial value } 2 \]

\[ s.\text{acquire}() \]
\[ s.\text{release}() \]

Rendezvous:

\[ \text{val } s = \text{Semaphore}(0) \]
\[ \text{val } t = \text{Semaphore}(0) \]

\[ \text{def } \text{send}() = \quad t.\text{release}() \gg s.\text{acquire}() \]
\[ \text{def } \text{receive}() = \quad t.\text{acquire}() \gg s.\text{release}() \]

\(n\)-party Rendezvous using \(2(n - 1)\) semaphores.
val req = Buffer()
val cb = Counter()

def rw() =
    req.get() > (b, s)>
    ( if(b) => cb.inc() => s.release() => rw() |
    if(!b) => cb.onZero() =>
    cb.inc() => s.release() => cb.onZero() => rw() )

def start(b) =
    val s = Semaphore(0)
    req.put((b, s)) => s.acquire()

def quit() = 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{align*}
def \text{eval}(u, t) &= \begin{cases} 
\text{if } t \text{ is the first value for } u, \text{ record it else stop} & \Rightarrow \\
\text{for every edge } (u, v) \text{ of length } d \text{ do} & \\
\text{wait for } d \text{ time units} & \Rightarrow \\
\text{eval}(v, t + d) & 
\end{cases} \\
\text{Goal} : \quad & \text{eval(}\text{source}, 0) | \\
& \text{read the value recorded for the } \text{sink}
\end{align*}
record and read sites

\textit{write}(u, t): \text{Write value } t \text{ for node } u. \text{ If already written, block.}

\textit{read}(u): \text{Return value for node } u. \text{ If unwritten, block.}
Graph Structure: Function \textit{Succ()}

\textit{Succ}(u) \text{ publishes } (x, 2), (y, 1), (z, 5).
Algorithm (contd.)

\[
\text{def eval}(u, t) = \begin{cases} 
\text{if } t \text{ is the first value for } u, \text{ record it else stop} & \Rightarrow \\
\text{for every edge } (u, v) \text{ of length } d \text{ do} & \\
\text{wait for } d \text{ time units} & \Rightarrow \\
\text{eval}(v, t + d) & 
\end{cases}
\]

Goal:
\[\text{eval}(\text{source}, 0) \mid \text{read the value recorded for the sink}\]

\[
\text{def eval}(u, t) = \begin{cases} 
\text{write}(u, t) & \Rightarrow \\
\text{Succ}(u) > (v, d) > & \\
\text{Rtimer}(d) & \Rightarrow \\
\text{eval}(v, t + d) & 
\end{cases}
\]

Goal:
\[\text{eval}(\text{source}, 0) \mid \text{read(sink)}\]
Algorithm (contd.)

\[
def \text{eval}(u, t) = \begin{align*}
&\text{write}(u, t) \\
&Succ(u) > (v, d) > \\
&R\text{timer}(d) \\
&\text{eval}(v, t + d)
\end{align*}
\]

Goal: \( \text{eval}(\text{source}, 0) \mid \text{read}(\text{sink}) \)

- Any call to \( \text{eval}(u, t) \): Length of a path from source to \( u \) is \( t \).
- First call to \( \text{eval}(u, t) \): Length of the shortest path from source to \( u \) is \( t \).
- \( \text{eval} \) 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.

\[
def \text{eval}(u, t) = \text{write}(u, t) \gg Succ(u) > (v, d) > Ltimer(d) \gg eval(v, t + d)
\]

**Goal:** \[eval(source, 0) \mid read(sink)\]
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:

- $L\text{timer}(t)$ 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 \( c.get() \)?

• \( \text{stop} \mid Ltimer(2) \)
  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 $\text{Ltimer.time()}$, initially $n = 0$.
- $q$: queue of calls to $\text{Ltimer()}$ whose responses are pending.

At run time:

- A call to $\text{Ltimer.time()}$ immediately responds with $n$.
- A call to $\text{Ltimer}(t)$ 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 $\text{Ltimer()}$. 
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.
Run the simulation for \textit{simtime}.
Below, \textit{Bank()} never publishes.

\begin{verbatim}
val \ z \ = \ Bank() \mid Ltimer(simtime)

\ z \ \gg \ Stats()
\end{verbatim}
Description of Bank

```python
def Bank() = (Customers() | Teller() | Teller()) >> stop
def Customers() = Source() >c> enter(c)
def Teller() = next() >c>
                  Ltimer(c.ServTime) >>
                  Teller()
def enter(c) = q.put(c)
def next() = q.get()
```
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 ⇒ Stats()
def Restaurant() = (Customers() | Cashier() | Cook() | Cook()) \[\text{stop}\]
def Customers() = Source() >c> enter(c)
def Cashier() = next() >c> 
    Ltimer(c.ringupTime) \[\text{stop}\]
    orders.put(c.order) \[\text{stop}\]
    Cashier()
def Cook() = orders.get() >order> 
    ( 
        prepTime(order.entrée) >t> Ltimer(t), 
        prepTime(order.side) >t> Ltimer(t), 
        prepTime(order.drink) >t> Ltimer(t) 
    ) \[\text{stop}\]
    Cook()
def enter(c) = q.put(c)
def next() = q.get()
Collecting Statistics: waiting time

Change

\[
\begin{align*}
\text{def } \text{enter}(c) & \quad = \quad q.\text{put}(c) \\
\text{def } \text{next}() & \quad = \quad q.\text{get}() \\
\end{align*}
\]

to

\[
\begin{align*}
\text{def } \text{enter}(c) & \quad = \quad L\text{timer}.\text{time}() >s> q.\text{put}(c, s) \\
\text{def } \text{next}() & \quad = \quad q.\text{get}() >(c, t)> \\
& \quad \quad \quad \quad L\text{timer}.\text{time}() >s> \\
& \quad \quad \quad \quad \text{reportWait}(s - t) \Rightarrow \\
& \quad \quad \quad \quad c
\end{align*}
\]
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
  $$sw[i].\text{stop} \mid sw[i + 1].\text{start}$$
- After removing an item if the queue length is $i$, $0 \leq i < N$, do
  $$sw[i].\text{start} \mid sw[i + 1].\text{stop}$$
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.