Simulations in the Orc Programming Language

Jayadev Misra

Department of Computer Science
University of Texas at Austin

ICTAC 2011
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.
- **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` ...
- `Mutable Ref, Semaphore, Channel, ...`
- `Timer`
- `External Services: Google Search, MySpace, CNN, ...`
- `Any Java Class instance, Any Orc Program`
- `Factory sites; Sites that create sites: Semaphore, Channel ...`
- `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:
  
  \[
  \begin{align*}
  \text{do } f \text{ and } g \text{ in parallel} & \quad f \mid g & \text{Symmetric composition} \\
  \text{for all } x \text{ from } f \text{ do } g & \quad f \gg x \gg g & \text{Sequential composition} \\
  \text{for some } x \text{ from } g \text{ do } f & \quad f \ll x \ll g & \text{Pruning}
  \end{align*}
  \]
Structure of Orc Expression

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

- **Composition** of two Orc expressions:

  do $f$ and $g$ in parallel  \[ f \mid g \]  Symmetric composition
  for all $x$ from $f$ do $g$ \[ f \gg x \gg g \]  Sequential composition
  for some $x$ from $g$ do $f$ \[ f \ll x \ll g \]  Pruning
Structure of Orc Expression

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

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

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

- **Composition** of two Orc expressions:

  \[
  \text{do } f \text{ and } g \text{ in parallel } \quad f \mid g \quad \text{Symmetric composition}
  \]

  \[
  \text{for all } x \text{ from } f \text{ do } g \quad \quad f \triangleright x \triangleright g \quad \text{Sequential composition}
  \]

  \[
  \text{for some } x \text{ from } g \text{ do } f \quad \quad f \prec x \prec g \quad \text{Pruning}
  \]
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 $g$.

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

- $(\text{CNN}(d) | \text{BBC}(d)) \gg x \gg \text{Email}(\text{address}, x)$
  - May call $\text{Email}$ twice.
  - Publishes up to two values from $\text{Email}$.

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 \((M() \mid N(x)) \prec x \prec g\)
- 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

\[ \text{Email}(\text{address}, x) \; <x< \; (\text{CNN}(d) \mid \text{BBC}(d)) \]

Binds \( x \) to the first value from \( \text{CNN}(d) \mid \text{BBC}(d) \).

Sends at most one email.
Some Fundamental Sites

- $Ift(b)$, $Iff(b)$: boolean $b$. Returns a signal if $b$ is true/false; remains silent otherwise.

- $Rwait(t)$: integer $t$, $t \geq 0$, returns a signal $t$ time units later.

- $stop$: never responds. Same as $Ift(false)$ or $Iff(true)$.

- $signal$: returns a signal immediately. Same as $Ift(true)$ or $Iff(false)$. 
Expression Definition

\[
\text{def } \text{MailOnce}(a) = \\
\quad \text{Email}(a, m) < m < (\text{CNN}(d) \mid \text{BBC}(d))
\]

\[
\text{def } \text{MailLoop}(a, t) = \\
\quad \text{MailOnce}(a) \Rightarrow \text{Rtimer}(t) \Rightarrow \text{MailLoop}(a, t)
\]

\[
\text{def } \text{metronome}() = \text{signal} \mid (\text{Rtimer}(1) \Rightarrow \text{metronome}())
\]

• Expression is called like a procedure.
  It may publish many values. \textit{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**: \texttt{if E then F else G}
- **Data structures**: Tuple and List
- **Pattern Matching**
- **Function Definition; Closure**
Variable Binding; Silent expression

\begin{align*}
  \textit{val } x &= 1 + 2 \\
  \textit{val } y &= x + x \\
  \textit{val } z &= x/0 \quad \text{-- expression is silent} \\
  \textit{val } u &= \text{if } (0 < 5) \text{ then } 0 \text{ else } z
\end{align*}
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 $C[x] < x < F$.

$$1 + 2 | 2 + 3$$  is  $add(1, 2) | add(2, 3)$

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

**Implication:**
Arguments of site calls are evaluated in parallel.
Site is called when all arguments have been evaluated.
Example: Fibonacci numbers

\[
\text{def } H(0) = (1, 1) \\
\text{def } H(n) = H(n - 1) > (x, y) > (y, x + y)
\]

\[
\text{def } Fib(n) = H(n) > (x, _) > x
\]

{- 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 $M$’s response if it arrives before time $t$, Otherwise, publish 0.

\[ z \leftarrow (M() \mid (Rwait(t) \gg 0)), \text{ or} \]
\[
val \ z = M() \mid (Rwait(t) \gg 0)
\]
\[ z \]
Fork-join parallelism

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

$((u, v) < u < M() \land < v < N())$

or,

$(M(), N())$
Simple Parallel Auction

- A list of bidders in a sealed-bid, single-round auction.
- \( b.ask() \) requests a bid from bidder \( b \).
- Ask for bids from all bidders, then publish the highest bid.

\[
\begin{align*}
def \text{auction}([]) &= 0 \\
def \text{auction}(b : bs) &= \max(b.ask(), \text{auction}(bs))
\end{align*}
\]

Notes:
- All bidders are called simultaneously.
- If some bidder fails, then the auction will never complete.
Parallel Auction with Timeout

- Take a bid to be 0 if no response is received from the bidder within 8 seconds.

\[
def \text{auction}([],) = 0
\]

\[
def \text{auction}(b : bs) =
  \text{max}(\text{b.ask()} | (R\text{wait}(8000) \gg 0),
  \text{auction}(bs)
  )
\]
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.
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.
  Shortest path length to node $i$ = time for $i$ to receive its first ray.
Graph structure in function $Succ()$

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

Figure: Graph Structure
Algorithm

```
def eval(u, t) = record value t for u ➞
for every successor v with d = length of (u, v):
    wait for d time units ➞
eval(v, t + d)
```

**Goal:** eval(source, 0) | read the value recorded for the sink

Record path lengths for node u in FIFO channel u.
Algorithm (contd.)

\[ \text{def } \text{eval}(u, t) = \begin{align*} & \text{record value } t \text{ for } u \quad \Rightarrow \\ & \text{for every successor } v \text{ with } d = \text{ length of } (u, v) : \\ & \quad \text{wait for } d \text{ time units } \Rightarrow \\ & \quad \text{eval}(v, t + d) \end{align*} \]

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

\[
\begin{align*}
\text{def } \text{eval}(u, t) = & \quad u.\text{put}(t) \Rightarrow \\
& \quad \text{Succ}(u) > (v, d) > \\
& \quad \text{Rwait}(d) \Rightarrow \\
& \quad \text{eval}(v, t + d) \\
\{- \text{ Goal :-} \} & \quad \text{eval}(\text{source}, 0) \mid \text{sink.get()}
\end{align*}
\]
Algorithm (contd.)

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

{− Goal :− } \text{eval}(\text{source}, 0) | \text{sink.get()}

• 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 \textit{Succ}, \textit{put} and \textit{get} should take no time.
Virtual Timer

Methods:

\[ V\text{wait}(t) \]
- Returns a signal after \( t \) virtual time units.

\[ V\text{time}() \]
- Returns the current value of the virtual timer.
Virtual timer Properties

- Virtual timer value is monotonic.
- \( V_{wait}(t) \) consumes exactly \( t \) units of virtual time.
- A step is started as soon as possible in virtual time.
- Virtual timer is advanced only if there can be no other activity.
Implementing virtual timer

Data structures:

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

At run time:

- A call to $Vtime()$ immediately responds with $n$.
- A call to $Vwait(t)$ is assigned rank $n + t$ and queued.
- **Progress**: If the program is stuck, then:
  
  remove the item with the lowest rank $r$ from $q$,
  
set $n := r$,
  
respond with a signal to the corresponding call to $Vwait()$. 
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 \textit{simtime}. Below, \textit{Bank()} never publishes .

\texttt{val } z = \textit{Bank()} | \textit{Vwait(simtime)}

\texttt{z \gg Stats()}
Description of Bank

\begin{align*}
\text{def } & \quad \text{Bank}() = (\text{Customers}() \mid \text{Teller()} \mid \text{Teller}()) \Rightarrow \text{stop} \\
\text{def } & \quad \text{Customers}() = \text{Source}() > c > \text{enter}(c) \\
\text{def } & \quad \text{Teller}() = \text{next}() > c > \\
& \quad \text{Vwait}(c.\text{ServTime}) \Rightarrow \\
& \quad \text{Teller}() \\
\text{def } & \quad \text{enter}(c) = q.\text{put}(c) \\
\text{def } & \quad \text{next}() = q.\text{get}()
\end{align*}
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

\[
\text{val } z = \text{Restaurant}() \mid \text{Vwait(simtime)}
\]

\[
z \gg \text{Stats}()
\]
def Restaurant() = (Customers() | Cashier() | Cook() | Cook()) ≫ stop

def Customers() = Source() ≫ c ≫ enter(c)

def Cashier() = next() ≫ c ≫
  Vwait(c.ringupTime) ≫
  orders.put(c.order) ≫
  Cashier()

def Cook() = orders.get() ≫ order ≫
  ( prepTime(order.entropy) ≫ t ≫ Vwait(t),
  prepTime(order.side) ≫ t ≫ Vwait(t),
  prepTime(order.drink) ≫ t ≫ Vwait(t) ) ≫ Cook()

def enter(c) = q.put(c)
def next() = q.get()
Collecting Statistics: waiting time

Change

\[
\text{def } \text{enter}(c) \quad = \quad q.put(c) \\
\text{def } \text{next}() \quad = \quad q.get()
\]

to

\[
\text{def } \text{enter}(c) \quad = \quad Vtime() >s> q.put(c, s) \\
\text{def } \text{next}() \quad = \quad q.get() > (c, t) > \\
\quad \quad \quad Vtime() >s> \\
\quad \quad \quad \quad \quad reportWait(s - t) \gg c
\]