Structured Wide-Area Programming

Jayadev Misra

Department of Computer Science
University of Texas at Austin

http://orc.csres.utexas.edu

April 12, 2010
Rennes, France
Features needed in a 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.
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.
Structured Concurrent Programming

- **Structured Sequential Programming**: Dijkstra circa 1968
  Component Integration in a sequential world.

- **Structured Concurrent Programming**: Component Integration in a concurrent world.
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, Any Orc Program
- 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,  \textit{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} & & f \mid g & \text{Symmetric composition} \\
  \text{for all } x \text{ from } f \text{ do } g & & f \succ x \succ g & \text{Sequential composition} \\
  \text{for some } x \text{ from } g \text{ do } f & & f \prec x \prec g & \text{Pruning} \\
  \text{if } f \text{ halts without publishing do } g & & f ; g & \text{Otherwise}
  \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:

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

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

  \[ \text{for some } x \text{ from } g \text{ do } f \]
  \[ f <x< g \quad \text{Pruning} \]

  \[ \text{if } f \text{ halts without publishing do } g \]
  \[ f; g \quad \text{Otherwise} \]
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  
  for all $x$ from $f$ do $g$  
  for some $x$ from $g$ do $f$  
  if $f$ halts without publishing do $g$

  $f | g$  
  $f >x> g$  
  $f <x< g$  
  $f; g$

  Symmetric composition  
  Sequential composition  
  Pruning  
  Otherwise
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: $f | g$
    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
  - if $f$ halts without publishing do $g$: $f ; g$
    Otherwise
Orc Notation

Structure of Orc Expression

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

- **Composition** of two Orc expressions:
  
  \[
  \begin{align*}
  & \text{do } f \text{ and } g \text{ in parallel} & f | g & \text{Symmetric composition} \\
  & \text{for all } x \text{ from } f \text{ do } g & f > x > g & \text{Sequential composition} \\
  & \text{for some } x \text{ from } g \text{ do } f & f < x < g & \text{Pruning} \\
  & \text{if } f \text{ halts without publishing do } g & f ; g & \text{Otherwise}
  \end{align*}
  \]
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: $f \gg g$

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 $\text{Email}$.

- $(\text{CNN}(d) \mid \text{BBC}(d)) \gg x \gg \text{Email}(\text{address}, x)$
  - 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 $f > x > g$
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 \((M() \ | N(x)) \ <x< \ g\)

- 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

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

Binds \( x \) to the first value from \( \text{CNN}(d) \ | \ \text{BBC}(d) \).
Sends at most one email.
Some Fundamental Sites

- \textit{if}(b): boolean \( b \),
  returns a \textit{signal} if \( b \) is true; remains \textit{silent} if \( b \) is false.

- \textit{Rtimer}(t): integer \( t, \ t \geq 0 \), returns a signal \( t \) time units later.

- \textit{stop}: never responds. Same as \textit{if}(false).

- \textit{signal}: returns a signal immediately. Same as \textit{if}(true).
Expression Definition

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

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

\[
def \text{metronome}() = \text{signal} | (\text{Rtimer}(1) \gg \text{metronome}())
\text{metronome}() \gg \text{stockQuote}()
\]

- Expression is called like a procedure.
  It may publish many values. \textit{MailLoop} does not publish.
- Site calls are strict; expression calls non-strict.
Orc Notation

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, Clausal Definition**
- **Function Definition; Closure**
Variable Binding; Silent expression

\[
\text{val} \ x = 1 + 2 \\
\text{val} \ y = x + x \\
\text{val} \ z = x/0 \quad \text{-- expression is silent} \\
\text{val} \ u = \text{if} \ (0 < 5) \ \text{then} \ 0 \ \text{else} \ z
\]
Comingling with Orc expressions

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

\[(1 + 2) | (2 + 3)\]

\[(1 \mid 2) + (2 \mid 3)\]

**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 \( add(1, 2) | add(2, 3) \)

\[(1 \mid 2) + (2 \mid 3)\] is \( (add(x, y) <x<(1 \mid 2)) <y<(2 \mid 3) \)
Orc Notation

Example: Fibonacci numbers

\[
\begin{align*}
\text{def } H(0) &= (1, 1) \\
\text{def } H(n) &= H(n - 1) > (x, y) > (y, x + y) \\
\text{def } Fib(n) &= H(n) > (x, _) > x
\end{align*}
\]

{- 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.

\[ z \triangleleft (M() \mid (Rtimer(t) \gg 0)) \], or

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

$z$
Orc Notation

Fork-join parallelism

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

\[
\left( \left( u, v \right) \\
\begin{align*}
<u & < M() \\
<v & < N()
\end{align*}
\]

or,

\[
\left( M(), N() \right)
\]
Recursive definition with time-out

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

\[
\begin{align*}
\text{def } \operatorname{tally}([[]]) &= 0 \\
\text{def } \operatorname{tally}(M : MS) &= (M() \gg 1 \mid \text{Rtimer}(10) \gg 0) + \operatorname{tally}(MS)
\end{align*}
\]
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 $N$.
  \[\text{val } (u, \_ ) = (N(), Rtimer(1))\]
- Call $M, N$ together.
  If $M$ responds within one unit, publish its response. Else, publish the first response.
  \[\text{val } x = M() \mid u\]
Parallel or

Sites \( M \) and \( N \) return booleans. Compute their parallel or.

\[
\text{val } x = M() \\
\text{val } y = N() \\
\text{if}(x) \gg true \mid \text{if}(y) \gg true \mid (x \| | y)
\]

To return just one value:

\[
\text{val } x = M() \\
\text{val } y = N() \\
\text{val } z = \text{if}(x) \gg true \mid \text{if}(y) \gg true \mid (x \| | y) \\
\text{z}
\]
Airline quotes: Application of Parallel or

Contact airlines $A$ and $B$.
Return any quote if it is below $300$ as soon as it is available, otherwise return the minimum quote.

$\text{threshold}(x)$ returns $x$ if $x < 300$; silent otherwise.
$\text{Min}(x, y)$ returns the minimum of $x$ and $y$.

\[
\begin{align*}
\text{val } x &= A() \\
\text{val } y &= B() \\
\text{val } z &= \text{threshold}(x) \mid \text{threshold}(y) \mid \text{Min}(x, y) \\
\end{align*}
\]
Backtracking: Eight queens

Figure: Backtrack Search for Eight queens
Eight queens; contd.

- \textit{xs}: partial placement of queens (list of values from $0..7$)
- \textit{extend}($xs$) publishes all solutions that are extensions of $xs$.
- \textit{open}($xs$) publishes the columns that are open in the next row.
- Solve the original problem by calling \textit{extend}([]).

\begin{verbatim}
\textbf{def} extend(xs) =
  if (length(xs) = 8) then xs 
  else
    (open(xs) >j> extend(j : xs))
\end{verbatim}
Mutable Structures

```latex
val \textcolor{black}{r} = \textcolor{black}{Ref()}\\
\textcolor{black}{r.write}(3) \quad \text{or} \quad \textcolor{black}{r} := 3\\
\textcolor{black}{r.read()} \quad \text{or} \quad \textcolor{black}{r}\?
```

```latex
def \textcolor{black}{swapRefs}(x, y) = \quad (x?, y?) > (xv, yv) > (x := yv, y := xv)
```
def search(key) =  --  return true or false
searchstart(key) \(\gg (\_, \_, q)\gg (q \neq \text{null})\)

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

def delete(key) =
Semaphore

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

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

Rendezvous:

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

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

\(n\)-party Rendezvous using \(2(n-1)\) semaphores.
Readers-Writers

val \texttt{req} = \texttt{Buffer()}

val \texttt{cb} = \texttt{Counter()}

val (r, w) = (Semaphore(0), Semaphore(0))

def \texttt{rw()} = \texttt{req.get()} >b>

( if(b) \Rightarrow \texttt{cb.inc()} \Rightarrow \texttt{r.release()} \Rightarrow \texttt{rw()}

| if(\neg b) \Rightarrow \texttt{cb.onZero()} \Rightarrow

| \texttt{cb.inc()} \Rightarrow \texttt{w.release()} \Rightarrow \texttt{cb.onZero()} \Rightarrow \texttt{rw()}

)

def \texttt{start}(b) = \texttt{req.put}(b) \Rightarrow

if(b) then \texttt{r.acquire()} else \texttt{w.acquire()}

def \texttt{end()} = \texttt{cb.dec()}
Processes

- Processes typically communicate via channels.
- For channel $c$, treat $c.put$ and $c.get$ as site calls.
- In our examples, $c.get$ is blocking and $c.put$ is non-blocking.
- Other kinds of channels can be programmed as sites.
Orc Notation

**Typical Iterative Process**

*Forever:* Read $x$ from channel $c$, compute with $x$, output result on $e$:

$$
\text{def } P(c, e) = c\.get() \triangleright x \triangleright \text{Compute}(x) \triangleright y \triangleright e\.put(y) \triangleright P(c, e)
$$

![Diagram](image)

**Figure:** Iterative Process
Process (network) to read from both $c$ and $d$ and write on $e$:

$$
def \text{Net}(c, d, e) = P(c, e) \parallel P(d, e)$$

**Figure:** Network of Iterative Processes
Workload Balancing

Read from \( c \), assign work randomly to one of the processes.

\[
def \text{bal}(c, c', d') = c\.get() > x > \text{random}(2) > t >
\]

\[
(\text{if } t = 0 \text{ then } c'.\text{put}(x) \text{ else } d'.\text{put}(x)) \Rightarrow
\]

\[
\text{bal}(c, c', d')
\]

\[
def \text{WorkBal}(c, e) = \text{val } c' = \text{Buffer()}
\]

\[
\text{val } d' = \text{Buffer()}
\]

\[
\text{bal}(c, c', d') \mid \text{Net}(c', d', e)
\]

Figure: Workload Balancing in a network of Processes
### Laws Based on Kleene Algebra

<table>
<thead>
<tr>
<th>Zero and</th>
<th>Commutativity of</th>
<th>Commutativity of $\gg$</th>
<th>Associativity of $\mid$</th>
<th>Associativity of $\gg$</th>
<th>Left zero of $\gg$</th>
<th>Right zero of $\gg$</th>
<th>Left unit of $\gg$</th>
<th>Right unit of $\gg$</th>
<th>Left Distributivity of $\gg$ over $\mid$</th>
<th>Right Distributivity of $\gg$ over $\mid$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f \mid stop = f$</td>
<td>$f \mid g = g \mid f$</td>
<td>$(f \mid g) \mid h = f \mid (g \mid h)$</td>
<td>$f \mid f = f$</td>
<td>$(f \gg g) \gg h = f \gg (g \gg h)$</td>
<td>$stop \gg f = stop$</td>
<td>$f \gg stop = stop$</td>
<td>$signal \gg f = f$</td>
<td>$f \gg x \gg let(x) = f$</td>
<td>$f \gg (g \mid h) = (f \gg g) \mid (f \gg h)$</td>
<td>$(f \mid g) \gg h = (f \gg h \mid g \gg h)$</td>
</tr>
</tbody>
</table>
(Distributivity over $\gg$) if $g$ is $x$-free
\[( (f \gg g) <x<h) = (f <x<h) \gg g \]

(Distributivity over $|$) if $g$ is $x$-free
\[( (f \mid g) <x<h) = (f <x<h) \mid g \]

(Distributivity over $\ll$) if $g$ is $y$-free
\[( (f \ll g) <y<h) = (f <y<h) \ll g \]

(Elimination of where) if $f$ is $x$-free, for site $M$
\[( f <x<M) = f \mid (M \gg stop) \]