CS 345, Programming Languages, Fall 2029; Course Content: ???

William Cook, Jayadev Misra, David Kitchin, Adrian Quark, Andrew Matsuoka, John Thywissen

Department of Computer Science
University of Texas at Austin

http://orc.csres.utexas.edu
Structured Concurrent Programming

- **Structured Sequential Programming**: Dijkstra circa 1968
  Fundamental Question: Component Integration.

- **Structured Concurrent Programming**: Component Integration
  - Concurrency combinators that promote component integration
  - Paradigms for constructing concurrent and distributed programs
  - Orchestration
Some Typical Applications

- **Account management in a bank** (Business process management):
  Workflow lasting over several months
  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

- **Managing a city**: (A proposal to EU)
  Components integrated dynamically
  The scope of software is nebulous
Some Typical Applications, contd.

- Matrix computation in a multi-core machine
- Map-Reduce using a server farm
- Concurrency management in database access
- Thread management in an operating system
- Mashups (Internet Scripting)
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.
Orchestrating Components (services)

Acquire data from services.
Calculate with these data.
Invoke yet other services with the results.

Additionally
Invoke multiple services simultaneously for failure tolerance.
Repeatedly poll a service.
Ask a service to notify the user when it acquires the appropriate data.
Download a service and invoke it locally.
Have a service call another service on behalf of the user.
...

Orc, an Orchestration Theory

- **Site**: Basic service (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.
Sites

- **External Services**: Google spell checker, Google Search, MySpace, CNN, Discovery ...

- **Giant Components**: Linux, Homeland Security Database ...

- **Any Java Class instance**

- **Library sites**
  - + − * & & | | ...
  - println, random, Prompt, Email ...
  - Timer
  - Database, Semaphore, Channel ...
  - Sites that create sites: MakeDatabase, MakeSemaphore, MakeChannel ...

...
Overview of Orc

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

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

- Orc is simple
  - Orc has only 3 combinators.
  - Can handle time-outs, priorities, failures, synchronizations, · · ·
  - Implementation allows writing simple expressions.
    2 + 3 is translated to site call Add(2,3).
Structure of Orc Expression

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

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

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

- **Composition** of two Orc expressions:

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

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

- **Composition** of two Orc expressions:
  - \( f \) 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
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
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.

**Examples**

- \( CNN(d) \mid BBC(d) \): calls both \( CNN \) and \( BBC \) simultaneously. Publishes values returned by both sites. (0, 1 or 2 values)

- \( WebServer() \mid MailServer() \mid LinuxServer() \)
  May not publish any value.
Sequential composition: $f \geq x \geq g$

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

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

- $(CNN(d) \mid BBC(d)) \geq x \geq Email(address, x)$
  - May call $Email$ twice.
  - Publishes up to two values from $Email$. 
Figure: Schematic of $f > x > g$
Pruning: \((f \prec x \prec g)\)

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

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

- When \(g\) returns a (first) value:
  - Assign it 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) \textless{} x \textless{} (CNN(d) \mid BBC(d))

Binds $x$ to the first value from $CNN(d) \mid BBC(d)$. 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 \( if(true) \).
Centralized Execution Model

- An expression is evaluated on a single machine (client).
- Client communicates with sites by messages.
Publish $M$’s response if it arrives before time $t$, Otherwise, publish $0$.

$$z < z < (M() \mid (Rtimer(t) \gg 0))$$
Fork-join parallelism

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

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

$$\ <v< N()$$
Expression Definition

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

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

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

- output n signals -
  
  \[
  \text{def } \text{signals}(n) = \text{if}(n > 0) \gg (signal \mid \text{signals}(n - 1))
  \]

- Publish a signal at every time unit.-
  
  \[
  \text{def } \text{metronome}() = \text{signal} \mid (\text{Rtimer}(1) \gg \text{metronome}())
  \]

- Publish a signal every \( t \) time units.-
  
  \[
  \text{def } \text{tmetronome}(t) = \text{signal} \mid (\text{Rtimer}(t) \gg \text{tmetronome}(t))
  \]

- Publish natural numbers from \( i \) every \( t \) time units.-
  
  \[
  \text{def } \text{gen}(i, t) = i \mid \text{Rtimer}(t) \gg \text{gen}(i + 1, t)
  \]
Recursive definition with time-out

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

```
def tally([]) = 0
def tally(M : MS) = u + v
    u < (M() >> 1) | (Rtimer(10) >> 0)
    v < tally(MS)
```

or, in the current language,

```
def tally([]) = 0
def tally(M : MS) = (M() >> 1) | Rtimer(10) >> 0) + tally(MS)
```
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.

\[
((u, v) \\
< u < M() \\
< v < N()) \\
\gg (f \mid g)
\]
Priority

- Publish \(N\)'s response asap, but no earlier than 1 unit from now. Apply fork-join between \(R\text{timer}(1)\) and \(N\).

\[
\text{def } \text{Delay()} = (R\text{timer}(1) \gg u) < u < N()
\]

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

\[
x < x < (M() \mid \text{Delay()})
\]
Interrupt \( f \)

Evaluation of \( f \) can not be directly interrupted. Introduce two sites:

- \texttt{Interrupt.set}: to interrupt \( f \)
- \texttt{Interrupt.get}: responds after \texttt{Interrupt.set} has been called.

Instead of \( f \), evaluate

\[
z < z < (f \mid \texttt{Interrupt.get}())
\]
Parallel or

Sites $M$ and $N$ return booleans. Compute their parallel or.

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

To return just one value:

\[
z \\
<z< \text{if}(x) \gg true \mid \text{if}(y) \gg true \mid (x||y) \\
<x< M() \\
<y< N()
\]
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 $y$.

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

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

```python
def extend(z, 1) = valid(0:o) \mid valid(1:o) \mid \cdots \mid valid(7:o)

def extend(z, n) = extend(z, 1) > y > extend(y, n - 1)
```

- `z`: partial placement of queens (list of values from 0..7)
- `extend(z, n)` publishes all valid extensions of `z` with `n` additional queens.
- `valid(z)` returns `z` if `z` is valid; silent otherwise.
- Solve the original problem by calling `extend([], 8)`. 
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.
**Typical Iterative Process**

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

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

![Iterative Process Diagram](image-url)
Process (network) to read from both $c$ and $d$ and write on $e$:

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

![Diagram of Process Network](attachment:image.png)

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

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

\[
\text{def } \text{bal}(c, c', d') = c\text{.get()} > x > \text{random}(2) > t > (\text{if}(t = 0) \Rightarrow c'\text{.put}(x) \mid \text{if}(t = 1) \Rightarrow d'\text{.put}(x)) \Rightarrow \text{bal}(c, c', d')
\]

\[
\text{def } \text{WorkBal}(c, e) = \text{Buffer()} > c' > \text{Buffer()} > d' > (\text{bal}(c, c', d') \mid \text{Net}(c', d', e))
\]

Figure: Workload Balancing in a network of Processes
Network of Services: Insurance Company

```python
def insurance() = apply() | join() | payment()

def apply() = inApply.get() >x> quote(x) >y> Email(x.addr, y) >>
               apply()

def join() = inJoin.get() > (id, p) > validate(id, p) >>
             ( add_client(id, p) >> Email(id.addr, welcome)
               | renew(id)
             ) >>
             join()

def payment() = inPayment.get() > (id, p) > validate(id, p) >>
                 update_client(id, p) >>
                 payment()
```
Laws of Kleene Algebra

(Zero and $|$)

(Commutativity of $|$)

(Associativity of $|$)

(Idempotence of $|$)

(Associativity of $\gg$)

(Left zero of $\gg$)

(Right zero of $\gg$)

(Left unit of $\gg$)

(Right unit of $\gg$)

(Left Distributivity of $\gg$ over $|$)

(Right Distributivity of $\gg$ over $|$)

$f \mid 0 = f$

$f \mid g = g \mid f$

$(f \mid g) \mid h = f \mid (g \mid h)$

$f \mid f = f$

$(f \gg g) \gg h = f \gg (g \gg h)$

$0 \gg f = 0$

$f \gg 0 = 0$

Signal $\gg f = f$

$f \gg x> \text{let}(x) = f$

$f \gg (g \mid h) = (f \gg g) \mid (f \gg h)$

$(f \mid g) \gg h = (f \gg h \mid g \gg h)$
Laws which do not hold

(Idempotence of $|\quad)$ $f | f = f$

(Right zero of $\gg\quad)$ $f \gg 0 = 0$

(Left Distributivity of $\gg$ over $|\quad)$ $f \gg (g | h) = (f \gg g) | (f \gg h)$
Additional Laws

(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

$$((f \ll g) < y < h) = ((f < y < h) < x < g)$$

(Elimination of where) if $f$ is $x$-free, for site $M$

$$(f < x < M) = f \mid (M \gg 0)$$
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.
Expressions and Sites needed for Shortest path

\textit{succ}(u): \quad \text{Publish all } (v, d), \text{ where edge } (u, v) \text{ has weight } d.

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

\textit{read}(u): \quad \text{Return value for node } u. \text{ If unwritten, block.}
First Algorithm

\[
\text{def } \text{eval1}(u, t) = \text{write}(u, t) \gg \\
\quad \text{Succ}(u) > (v, d) > \\
\quad \text{Rtimer}(d) \gg \\
\quad \text{eval1}(v, t + d)
\]

\textbf{Goal: } \text{eval1(source, 0) } | \text{read(sink)}

First call to \text{eval1}(u, t):

- The relative time in the evaluation is \( t \).
- Length of the shortest path from source to \( u \) is \( t \).
- \text{eval1} does not publish.
Current Status

- A prototype implementation; robust, non-optimized.
- An extensive site library.
- Several small to medium applications coded.
Where we are heading

- Coding large distributed applications.
- Transactions
- Logical time
- Secure workflow.
- Adaptive workflow.

See [http://orc.csres.utexas.edu](http://orc.csres.utexas.edu)