Structured Concurrent Programming

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

Outline

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 $\cdots$

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, $\cdots$
  - 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:

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.

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

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 $Email$.

Schematic of Sequential composition

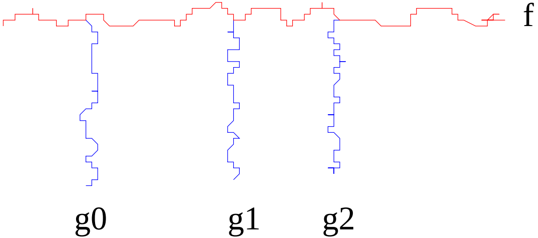


Figure: Schematic of f $>x>$ g

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

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 $(M() \mid N(x)) \ {<}x{<}\ g$
- When g returns a (first) value:
  - Assign it to x.
  - Terminate g.
  - Resume suspended calls.
- Values published by f are the values of $(f \ {<}x{<}\ g)$.

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 $if(true)$.

Centralized Execution Model

- An expression is evaluated on a single machine (client).
- Client communicates with sites by messages.

Time-out

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

$$z \; <z< \; (M() \; | \; (Rtimer(t) \gg 0))$$

-

Fork-join parallelism

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

$$((u, v) \quad <u< M()) \\ \qquad\qquad\qquad <v< N()$$

-

Expression Definition

$$\begin{array}{l} def\ \ MailOnce(a) = \\ \qquad Email(a, m) \ \ <m< \ (CNN(d) \ \mid BBC(d)) \end{array}$$

$$\begin{array}{l} def\ \ MailLoop(a, d) = \\ \qquad MailOnce(a) \ \gg Rtimer(d) \ \gg MailLoop(a, d) \end{array}$$

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

Expression Definition

- output n signals -

$def \quad signals(n) = if(n > 0) \gg (signal \mid signals(n-1))$

- Publish a signal at every time unit.-

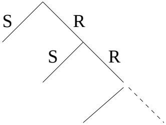
$def \quad metronome() = signal \mid (Rtimer(1) \gg metronome())$

- Publish a signal every t time units.-

$def \quad tmetronome(t) = signal \mid (Rtimer(t) \gg tmetronome(t))$

- Publish natural numbers from i every t time units.-

$def \quad gen(i, t) = i \mid Rtimer(t) \gg gen(i + 1, t)$



Recursive definition with time-out

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

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

or, in the current language,

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

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.

$$
\begin{array}{l}
\quad ((u, v) \\
\qquad <u< M() \\
\qquad <v< N()) \\
\gg (f \mid g)
\end{array}
$$

Priority

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

$$def\ \ Delay() = (Rtimer(1) \gg u)\ \ {<}u{<}\ N()$$

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

$$x\ \ {<}x{<}\ (M()\ \mid Delay())$$

Interrupt f

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

- $Interrupt.set$: to interrupt f
- $Interrupt.get$: responds after $Interrupt.set$ has been called.

Instead of f, evaluate

$$z \ <z< (f \ \mid Interrupt.get())$$

Parallel or

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

$$
\begin{array}{l}
if(x) \gg true \mid if(y) \gg true \mid (x||y) \\
\quad <x< M() \\
\quad <y< N()
\end{array}
$$

To return just one value:

$$
\begin{array}{l}
z \\
\quad <z< if(x) \gg true \mid if(y) \gg true \mid (x||y) \\
\quad <x< M() \\
\quad <y< N()
\end{array}
$$

-

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

$$
\begin{array}{l}
z \\
\quad\quad <z< threshold(x) \;\; | \;\; threshold(y) \;\; | \;\; Min(x,y) \\
\quad\quad <x< A() \\
\quad\quad <y< B()
\end{array}
$$

Backtracking: Eight queens

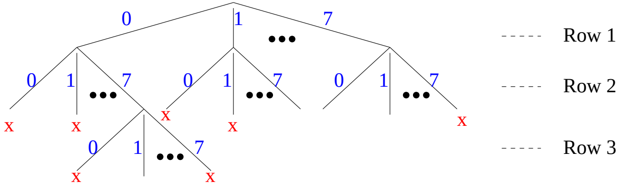


Figure: Backtrack Search for Eight queens

Eight queens; contd.

$$def\ \ extend(z, 1) = valid(0{:}z)\ |\ valid(1{:}z)\ |\ \cdots\ |\ valid(7{:}z)$$

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

- 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:

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

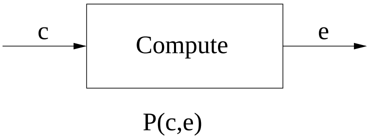


Figure: Iterative Process

Process Network

Process (network) to read from both c and d and write on e:

$$def \;\; Net(c, d, e) = \;\; P(c, e) \;\; | \; P(d, e)$$

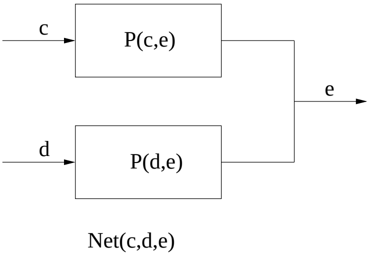


Figure: Network of Iterative Processes

Workload Balancing

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

$$
\begin{aligned}
def\ \ bal(c, c', d') = \quad & c.get()\ \ >x>\ random(2)\ \ >t> \\
& (\ \ if(t = 0) \gg c'.put(x) \\
& \ \ |\ if(t = 1) \gg d'.put(x) \\
&) \gg bal(c, c', d')
\end{aligned}
$$

$$
\begin{aligned}
def\ \ WorkBal(c, e) = \quad & Buffer()\ \ >c'>\ Buffer()\ \ >d'> \\
& (bal(c, c', d')\ \ |\ Net(c', d', e))
\end{aligned}
$$

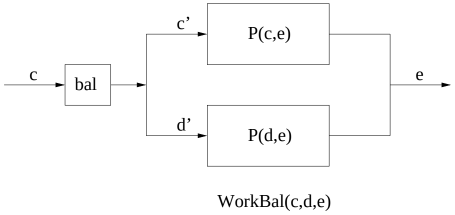


Figure: Workload Balancing in a network of Processes

Network of Services: Insurance Company

$$def \;\; insurance() = apply() \;|\; join() \;|\; payment()$$

$$\begin{aligned} def \;\; apply() = \; & inApply.get() \; >x> quote(x) \; >y> Email(x.addr, y) \gg \\ & apply() \end{aligned}$$

$$\begin{aligned} def \;\; join() = \; & inJoin.get() \; >(id,p)> validate(id,p) \gg \\ & (\; add\_client(id,p) \gg Email(id.addr, welcome) \\ & \;\; |\; renew(id) \\ &) \gg \\ & join() \end{aligned}$$

$$\begin{aligned} def \;\; payment() = \; & inPayment.get() \; >(id,p)> validate(id,p) \gg \\ & update\_client(id,p) \gg \\ & payment() \end{aligned}$$

Laws of Kleene Algebra

(Zero and $\mid$)	$f \mid 0 = f$
(Commutativity of $\mid$)	$f \mid g = g \mid f$
(Associativity of $\mid$)	$(f \mid g) \mid h = f \mid (g \mid h)$
(Idempotence of $\mid$)	$f \mid f = f$
(Associativity of $\gg$)	$(f \gg g) \gg h = f \gg (g \gg h)$
(Left zero of $\gg$)	$0 \gg f = 0$
(Right zero of $\gg$)	$f \gg 0 = 0$
(Left unit of $\gg$)	$Signal \gg f = f$
(Right unit of $\gg$)	$f >x> let(x) = f$
(Left Distributivity of $\gg$ over $\mid$)	$f \gg (g \mid h) = (f \gg g) \mid (f \gg h)$
(Right Distributivity of $\gg$ over $\mid$)	$(f \mid g) \gg h = (f \gg h \mid g \gg h)$

Laws which do not hold

(Idempotence of $\mid$)	$f \mid f = f$
(Right zero of $\gg$)	$f \gg 0 = 0$
(Left Distributivity of $\gg$ over $\mid$)	$f \gg (g \mid h) = (f \gg g) \mid (f \gg h)$

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 $<<$) if g is y-free

$$\begin{aligned} & ((f <x< g) <y< h) \\ = \quad & ((f <y< h) <x< g) \end{aligned}$$

(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

$succ(u)$: Publish all (v, d), where edge (u, v) has weight d.

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

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

First Algorithm

$$\begin{aligned} \mathit{def}\ \ eval1(u,t) = \quad & write(u,t) \gg \\ & Succ(u)\ >(v,d)> \\ & Rtimer(d) \gg \\ & eval1(v, t+d) \end{aligned}$$

$$\mathit{Goal}: \quad eval1(source, 0) \mid read(sink)$$

First call to $eval1(u,t)$:

- The relative time in the evaluation is t.
- Length of the shortest path from source to u is t.
- $eval1$ 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