Structured Orchestration of Data and Computation

William Cook
Jayadev Misra
David Kitchin
John Thywissen
Arthur Peters

Department of Computer Science
University of Texas at Austin

http://orc.csres.utexas.edu

The 10th International Symposium on Formal Aspects of Component Software
Jiangxi Normal University, Nanchang, China.
October 28 - 30, 2013

A Big Vision:
Software Challenge in the next two decades

- Design Methodology
    - Build it cheap
    - Build it correct
    - Build it for evolution

- Reliability
    - Correctness
    - Fault-tolerance in software and hardware

- Security

Orc

- Orc addresses Design: as a component integration system.

  Components:

  - from many vendors
  - for many platforms
  - written in many languages
  - may run concurrently and in real-time

- Preliminary work on Security.

Evolution of Orc

- Web-service Integration
- Component Integration
- Structured Concurrent Programming

Initial Goal: Internet Scripting

- Web Services as primitive operations
- Combinators to orchestrate them:
  1. Sequential Orchestration
  2. Parallel Orchestration
  3. Interruption

Web-service Integration: 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.

-

Enhanced Goal: Component Integration

Components could be:

- Web services
- Library modules
- Custom Applications, including real time

Components could be for:

- Functional Transformation
- Data Object Creation
- Real-time Computation

Component Integration; contd.

- Combine any kind of component, not just web services
- Small components: add two numbers, print a file ...
- Large components: Linux, MSword, email server, file server ...
- Time-based components: for real-time computation
- Actuators, sensors, humans as components
- Fast and Slow components
- Short-lived and Long-lived components
- Written in any language for any platform

Concurrency

- Component integration: typically sequential using objects
- Concurrency is ubiquitous
- Magnitude higher in complexity than sequential programming
- No generally accepted method to tame complexity
- May affect security

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: Structured Concurrent Programming

- A combinator combines two components to get a component
- Combinators may be applied recursively
- Results in hierarchical/modular program construction
- Combinators may orchestrate components concurrently
- Orc is just about 4 combinators

Power of Orc

- Solve all known synchronization, communication problems
- Code objects, active objects
- Solve all known forms of real-time and periodic computaions
- Solve a limited kind of transactions
- and, all combinations of the above

Typical Computing Domains

- Software Integration within an organization
- Workflow
- Mediated Computing
- Perpetual Computing
- Rapid Prototyping

Orc Calculus

- Site: Basic service or component.
- Concurrency combinators for integrating sites.
- Calculus includes nothing other than the combinators.

No notion of data type, thread, process, channel,
synchronization, parallelism $\cdots$

New concepts are programmed using new sites.

Examples of Sites

- $+ \quad - \quad * \quad \&\& \quad || \quad =$ ...
- `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 any number of values.
- The value is published.

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
if f halts without publishing do g	$f \; ; \; g$	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	$f \mid g$	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
if f halts without publishing do g	$f \; ; \; g$	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	$f \mid g$	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
if f halts without publishing do g	$f\ ;\ g$	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	$f \mid g$	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
if f halts without publishing do g	$f \; ; \; g$	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	$f \mid g$	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
if f halts without publishing do g	$f \;;\; g$	Otherwise

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

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 $Email$.

Notation: $f \gg g$ for $f \;>x>\; g$, if x is unused in g.

Right Associative: $f \;>x>\; g \;>y>\; h$ is $f \;>x>\; (g \;>y>\; h)$

Schematic of Sequential composition

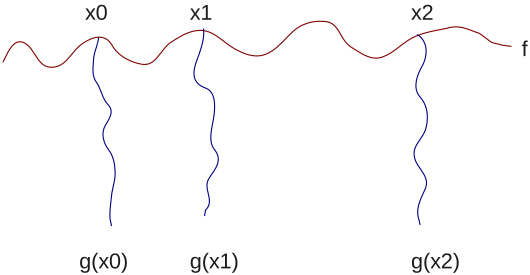


Figure: Schematic of $f >x> g$

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

Notation: $f \ll g$ for $f \; {<}x{<} \; g$, if x is unused in f.

Left Associative: $f \; {<}x{<} \; g \; {<}y{<} \; h$ is $(f \; {<}x{<} \; g) \; {<}y{<} \; h$

Example of Pruning

$$Email(address, x) \;\; <x< \; (CNN(d) \mid BBC(d))$$

Binds x to the first value from $CNN(d) \mid BBC(d)$.
Sends at most one email.

Multiple Pruning happens concurrently

$add(x,y) < x < f \;\; < y < g$ is $(add(x,y) < x < f\,) < y < g$

$(add(x,y) < x < f\,)$ is computed concurrently with g

$(add(x,y)$, f and g computed concurrently.

Otherwise: f ; g

Do f. If f halts without publishing then do g.

- An expression halts if
  - its execution can take no more steps, and
  - all called sites have either responded, or will never respond.
- A site call may respond with a value, indicate that it will never respond (helpful), or do neither.
- All library sites in Orc are helpful.

Examples of $f \; ; \; g$

$1 \; ; \; 2$ publishes 1

$(CNN(d) \mid BBC(d)) \;\; {>}x{>} \; Email(address, x) \;\; ; \;\; Retry()$

If the sites are not helpful, this is equivalent to

$(CNN(d) \mid BBC(d)) \;\; {>}x{>} \; Email(address, x)$

Orc program

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

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

Some Fundamental Sites

- $Ift(b)$, $Iff(b)$: boolean b,
  Returns a signal if b is true/false; remains silent otherwise.
  Site is helpful: indicates when it will never respond.
- $Rwait(t)$: 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 $Iff(false)$.

Use of Fundamental Sites

- Print all publications of h. When h halts, publish "done".

  $h \; >x> Println(x) \gg stop \; ; \; "done"$

- Timeout:
  Call site M.
  Publish its response if it arrives within 10 time units.
  Otherwise publish 0.

  $x \; <x< (M() \mid Rwait(10) \gg 0)$

Interrupt f

- Evaluation of f can not be directly interrupted.
- Introduce two sites:
  - *Interrupt.set*: to interrupt f
  - *Interrupt.get*: responds only after *Interrupt.set* has been called.

  - *Interrupt.set* is similar to *release* on a semaphore;
    Interrupt.get is similar to *acquire* on a semaphore.
- Instead of f, evaluate

$$z \;{<}z{<}\; (f \;|\; \mathit{Interrupt.get}())$$

Site Definition

$$\mathit{def}\ \ \mathit{MailOnce}(a) =$$
$$\quad \mathit{Email}(a, m)\ \ <\!m\!<\ (\mathit{CNN}(d) \mid \mathit{BBC}(d))$$

$$\mathit{def}\ \ \mathit{MailLoop}(a, t) =$$
$$\quad \mathit{MailOnce}(a)\ \gg\ \mathit{Rwait}(t)\ \gg\ \mathit{MailLoop}(a, t)$$

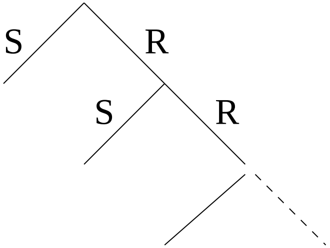
$$\mathit{def}\ \ \mathit{metronome}() =\ \mathit{signal} \mid (\mathit{Rwait}(1)\ \gg \mathit{metronome}())$$

- Expression is called like a procedure.
  It may publish many values. *MailLoop* does not publish.

Example of a Definition: Metronome

Publish a signal every unit.

$$\mathit{def}\ \ \mathit{Metronome}() = \underbrace{\mathit{signal}}_{S}\ \ |\ \ (\underbrace{\mathit{Rwait}(1) \gg \mathit{Metronome}()}_{R})$$



Unending string of Random digits

$Metronome() \gg Random(10)$ – one every unit

$def\ \ rand\_seq(dd) =$ – at a specified rate
$Random(10) \mid Rwait(dd) \gg rand\_seq(dd)$

Concurrent Site call

- Sites are often called concurrently.
- Each call starts a new instance of site execution.
- If a site accesses shared data, concurrent invocations may interfere.

Example: Publish each of "tick" and "tock" once per second, "tock" after an initial half-second delay.

$$\begin{array}{lll} & \mathit{Metronome}() \gg {''}\mathit{tick}{''} \\ \mid \mathit{Rwait}(500) \gg & \mathit{Metronome}() \gg {''}\mathit{tock}{''} \end{array}$$

Orc Language vs. Orc Calculus

- Data Types: Number, Boolean, String, with Java operators
- Conditional Expression: *if* E *then* F *else* G
- Data structures: Tuple, List, Record
- Pattern Matching; Clausal Definition
- Closure
- Orc combinators everywhere
- Class for active objects

Subset Sum

Given integer n and list of integers xs.

$parsum(n, xs)$ publishes all sublists of xs that sum to n.

`parsum(5,[1,2,1,2]) = [1,2,2], [2,1,2]`

`parsum(5,[1,2,1])` is silent

```
$def\ \ parsum(0, [\,]) = [\,]$

$def\ \ parsum(n, [\,]) = stop$

$def\ \ parsum(n, x : xs) =$
    $parsum(n - x, xs)\ \ >ys>\ x : ys$
  $|\ \ parsum(n, xs)$
```

Subset Sum (Contd.), Backtracking

Given integer n and list of integers xs.

$seqsum(n, xs)$ publishes the first sublist of xs that sums to n.

“First” is smallest by index lexicographically.
`seqsum(5,[1,2,1,2]) = [1,2,2]`

`seqsum(5,[1,2,1])` is silent

$$def \;\; seqsum(0, [\,]) = [\,]$$

$$def \;\; seqsum(n, [\,]) = stop$$

$$\begin{aligned} def \;\; & seqsum(n, x : xs) = \\ & \quad x : seqsum(n - x, xs) \\ & \quad ; \; seqsum(n, xs) \end{aligned}$$

Subset Sum (Contd.), Concurrent Backtracking

Publish the first sublist of xs that sums to n.

Run the searches concurrently.

```
def parseqsum(0, []) = []

def parseqsum(n, []) = stop

def parseqsum(n, x : xs) =
    (p ; q)
        <p< x : parseqsum(n − x, xs)
        <q< parseqsum(n, xs)
```

Note: Neither search in the last clause may succeed.

Process Networks

- A process network consists of: processes and channels.
- The processes run autonomously, and
  communicate via the channels.
- A network is a process; thus hierarchical structure.
  A network may be defined recursively.
- A channel may have intricate communication protocol.
- Network structure may be dynamic, by adding/deleting
  processes/channels during its execution.

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.
- We consider only FIFO channels.
  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:

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

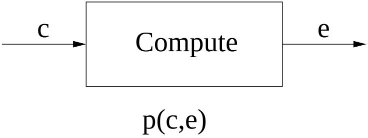


Figure: Iterative Process

Composing Processes into a Network

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

$$def \;\; net(c, d, e) = \;\; p(c, e) \mid p(d, e)$$

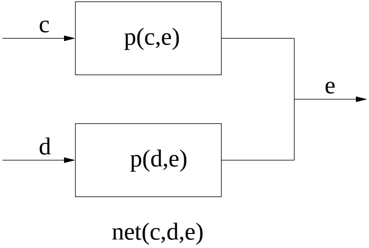


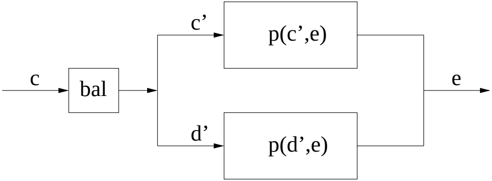
Figure: Network of Iterative Processes

Workload Balancing

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

$$
\begin{array}{ll}
def\ \ bal(c,c',d') = & c.get()\ \ {>}x{>}\ random(2)\ \ {>}t{>} \\
 & (if\ \ t=0\ then\ \ c'.put(x)\ else\ \ d'.put(x)) \gg \\
 & bal(c,c',d')
\end{array}
$$

$$
\begin{array}{ll}
def\ \ workbal(c,e) = & val\ \ c' = \ Channel() \\
 & val\ \ d' = \ Channel() \\
 & bal(c,c',d') \mid net(c',d',e)
\end{array}
$$



workBal(c,e)

Packet Reassembly Using Sequence Numbers

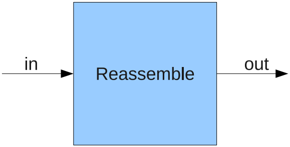


Figure: Packet Reassembler

- Packet with sequence number i is at position p_i in the input channel.
- Given: $|i - p_i| \leq k$, for some positive integer k.
- Then $p_i \leq i + k \leq p_{i+2\times k}$. Let $d = 2 \times k$.

Packet Reassembly Program

```
def reassembly(read, write, d) =   – d must be positive
  val ch = Table(d, lambda(_) = Channel())

  def input() = read() >(n, v)> ch(n%d).put(v) ≫ input()

  def output(i) = ch(i).get() >v> write(v) ≫ output((i + 1)%d)

input() | output(0)        – Goal expression

{- With Multiple Readers -} read() | read() | write(0)
```

Next Steps: Large Scale Deployment

- Industrial strength Implementation
- Distributed Implementation
- Partnering