Architecture as coordination: the Orc perspective

L.S. Barbosa

Dept. Informática,
Universidade do Minho
Braga, Portugal

DI-CCTC, UM, 2009

- Introduction
- Basic calculus
- Functional core
- Orc(hestration) examples
- Conclusion

Introduction

Architectural design as a coordination problem

A typical scenario: Applications acquire data from services, compute over 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.
- ...

Orc — orc.csres.utexas.edu/

A process calculus for service orchestration, ie

- A model for expressing coordination of independent services using the following *rationale*: a Orc expression invokes multiple (external or local) services to achieve a goal while managing time-outs, priorities, and failures of services or communications;
- assuming the form of a process calculus, with an operational semantics based on a lts labelled by pairs (event, time),
- but, unlike classical concurrency models, introduces an asymmetric relationship between a program and the services that constitute its environment: An orchestration invokes and receives responses from the external services, which do not initiate communication.

Orc — orc.csres.utexas.edu/

A full language for structured concurrent programming

- Structured programming: sequential component composition (Dijkstra, 1968) vs concurrent component composition (cf, paralelism, asynchrony, failures, timeouts, ...)
- functional flavour (yet handling many non-functional issues: spawning of concurrent threads, time-outs, etc);
- particularly suitable to express workflows, internet scripting, and, in general, service orchestration at large scale;
- efficient implementation, with easy integration with Java

- Introduction
- Basic calculus
- Functional core
- Orc(hestration) examples
- Conclusion

Sites

A site represents a service or component, local or remote, that can be invoked

`add(3,4)`	Add the numbers 3 and 4.
CNN(d)	Get the CNN news headlines for date d.
`Prompt("Name:")`	Prompt the user to enter a name.
swap(l_0,l_1)	Swap the values stored at locations l_0 and l_1.
`Weather("Austin, TX")`	Find the current weather in Austin.
`random(10)`	Get a random integer in the range 0..9.
invertMatrix(m)	Find the inverse of matrix m.

- called like procedures, but with a strict calling discipline
- a call returns at most one value, which is publisehd

Sites

A site may respond, halt (ie, report it will not respond, eg, when facing an invalid operation, system error or non data availability) or neither respond nor halt

Special sites

- *let* (the identity site: publishes its own argument
- *if* (conditional): responds with a signal if its argument is true, and otherwise halts.
- *signal* (equivalent to *if*(*true*))
- *stop* (equivalent to *if*(*false*))
- *Rtimer*(t), for t an integer: responds with a signal t milisecs later
- ...

Combinators

A Orc program consists of a set of definitions and a goal expression which calls sites and publishes values.

Sites are orchestrated in an expression through a set of 4 combinators (ordered by decreasing precedence):

- pipelining: $f > x > g$
- parallel composition: $f \mid g$
- pruning: $f < x < g$
- sequential composition: $f ; g$

... no notions of thread, channel, process, synchronisation, etc.

Parallel composition: $f \mid g$

example:
$CNN(d) \mid BBC(d)$

- f and g are evaluated independently
- publish all values from both
- no direct interaction between f and g (can communicate only through sites).
- (commutative and associative)

Pipelining: $f > x > g$

example:
$(CNN(d) \mid BBC(d)) > r > email(addr, r)$

- ie, for all values published by f, invoke g
- publish only values. if any, returned by g
- execution of f continues in parallel with those of g
- (left associative)

Pruning: $f < x < g$

example:
$email(addr, r) < r < (CNN(d) \mid BBC(d))$

- ie, for some value published by g, invoke f
- f and g evaluate in parallel
- calls (in f) depending on x are suspended
- when g returns a first value, binds it ti x, terminates and resume suspended calls.
- (right associative)

Sequential composition: $f; g$

example:
$(CNN(d); BBC(d)) > x > email(addr, x)$

- first invoke f
- if f publishes no values and then halts, then g executes.
- f halts if all site calls in f have either responded or halted, f will never call any more sites and will never publish any more values
- (associative)

Definitions

example:
$def\ metronome(t) = signal \mid Rtimer(t) >> metronome(t)$

- similar to declaration of functions
- unlike a site call, a function call does not suspend if one of its arguments is a variable with no value
- a function call may publish more than one value: it publishes every value published by the execution of f
- definitions may be recursive

The calculus

(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

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

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

$$(f\ <x<\ M) = f \mid (M \gg stop)$$

- bisimulation equalities (wrt to the lts sematics [Wehrman et al 2008])

The calculus

- almost a Kleene algebra

(Zero and $\mid$)	$f \mid stop = f$
(Commutativity of $\mid$)	$f \mid g = g \mid f$
(Associativity of $\mid$)	$(f \mid g) \mid h = f \mid (g \mid h)$
(Idempotence of $\mid$) NO	$f \mid f = f$
(Associativity of $\gg$)	$(f \gg g) \gg h = f \gg (g \gg h)$
(Left zero of $\gg$)	$stop \gg f = stop$
(Right zero of $\gg$) NO	$f \gg stop = stop$
(Left unit of $\gg$)	$signal \gg f = f$
(Right unit of $\gg$)	$f >x> let(x) = f$
(Left Distributivity of $\gg$ over $\mid$) NO	$f \gg (g \mid h) = (f \gg g) \mid (f \gg h)$
(Right Distributivity of $\gg$ over $\mid$)	$(f \mid g) \gg h = (f \gg h \mid g \gg h)$

- Introduction
- Basic calculus
- Functional core
- Orc(hestration) examples
- Conclusion

The functional core

- function definitions:
  $def\ sumto(n) = if n < 1\ then\ 0\ else\ n + sumto(n-1)$
- variable bindings:
  $val\ x = 1 + 2$
  $val\ y = x + x$

  $val\ x = 1/0$
  $val\ y = 4 + 5$
  $if\ false\ then\ x\ else\ y$
- patterns:
  $val\ ((a, b), c) = ((1, true), (2, false))$

Translation into the basic calculus

- Operators become site call:
  $1 + (2 + 3)$ to $add(1, x) < x < add(2, 3)$
  if t then f else g to $(if(b)f \mid not(b) > c > if(c)g) < b < t$
- Bidings become combinator expressions:
  $val\ x = g\ \ f$ to $f < x < g$
- Function definitions become ... standard Orc definitions

Translation into the basic calculus

```
def throw() = random(6) + 1

def exp(0,_) = 0
def exp(n,c) =
   (if throw() + throw() = c then 1 else 0)
   + exp(n-1,c)
```

Translation into the basic calculus

```
def throw() = add(x,1) <x< random(6)

def exp(n,c) =
    ( if(b) >> let(0)
    | not(b) >nb> if(nb) >>
      ( add(x,y)
          <x< ( ( if(bb) >> 1 | not(bb) >nbb> if(nbb) >> 0 )
                                    <bb< equals(p,c)
                                      <p< add(q,r)
                                        <q< throw()
                                        <r< throw() )
          <y< ( exp(m,c) <m< sub(n,1) ) )
    ) <b< equals(n,0)
```

Translation into the basic calculus

Orc expressions may contain functional expressions and vice-versa

example: $(1 + 2) \mid (2 + 3)$ becomes

$((let(x) \mid let(y)) < x < add(1, 2)) < y < add(2, 3)$

example: $(1|2) + (2|3)$ becomes

$(add(x, y) < x < (1 \mid 2)) < y < (2 \mid 3)$

- Introduction
- Basic calculus
- Functional core
- Orc(hestration) examples
- Conclusion

Taking time seriously

example (interrupt):

$email(addr, x) < x < (BBC(d) \mid Rtimer(5000) >> \text{"}error\text{"})$

example (count replies within a time interval):

$def\ \ callCount([]) = 0$

$def\ \ callCount(H : T) =$

$\quad (H() >> 1 \mid Rtimer(10) >> 0)\ +\ callCount(T)$

Fork-Join pattern

is expressed just as (P, Q), which equivales to
$((x, y) < x < P) < y < Q$

example (electronic auction):
$def\ \ auction([]) = 0$
$def\ \ auction(b : bs) = max(b.ask(), auction(bs))$

Note that all bidders are called simultaneously.
But what if one of them fails to reply?

Fork-Join pattern

example (electronic auction with time-out):

$$
\begin{array}{l}
def\ \ auction([]) = 0 \\
def\ \ auction(b : bs) = \\
\quad val\ bid = b.ask() \mid Rtimer(5000) >> 0 \\
\qquad max(bid, auction(bs))
\end{array}
$$

Synchronization barrier

from

$$P() > x > F \mid Q() > x > G$$

to

$$(P(), Q()) > (x, y) > (F \mid G)$$

Sequential Fork-Join pattern

example (print lines, signal the end):
$F > x > println(x) \;>>\; stop \;;\; signal$

- A recursive fork-join solution requires lines be stored in a traversable data structure like a list, rather than streamed as publications from F
- Here, since ; only evaluates its RHS if the LHS does not publish, suppress the publications on the LHS using stop
- Need to assume detection of F halting (what if the sending party never closes the socket?)

Priority

- publish Qs response asap, but no earlier than 1 unit from now:
  $val\ (u, \_) = (Q(), Rtimer(1))$
- call P, Q together and publish P´s response if obtained within one unit; other wise publish the first response to come:
  $val\ x = P() \mid u$

Parallel Disjunction pattern

$let($
$val\ a = P$
$val\ b = Q$
$(a||b) \mid if(a) >> true \mid if(b) >> true$
$)$

Network of iterative processes

example (iterative process: input from c, output to e):

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

example (network: input from c, d, output to e):

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

Routing

example (generalised time-out):

$$
\begin{array}{l}
val\ c = Buffer() \\
repeat(c.get) << \\
\quad P > x > c.put(x) >> stop \\
\quad |\ Rtimer(1000) >> c.closenb()
\end{array}
$$

- allows P to execute for one second and then terminates it
- each value by P is routed through channel c to avoid end P
- after one second, $Rtimer(1000)$ responds, triggering the call $c.closenb()$ which closes c and publishes a signal
- function *repeat* repeatedly take and publish values from c until it is closed

Routing

example (interrupt based on a signal from elsewhere):

$$
\begin{array}{l}
val\ c = Buffer() \\
val\ done = Semaphore(0) \\
repeat(c.get) << \\
\quad P > x > c.put(x) >> stop \\
\quad | \ done.acquire() >> c.closenb()
\end{array}
$$

- dot notation
- instead of waiting for a timer wait for the semaphore *done* to be released
- any call to *done.release* will terminate the expression, because it will cause $done.acquire() topublish$
- but otherwise P executes normally and may publish any number of values

- Introduction
- Basic calculus
- Functional core
- Orc(hestration) examples
- Conclusion

Our approach to Software Architecture

Architectural design as a coordination problem

- The main architectural challenge is to coordinate multiple, heterogeneous, distributed, loosely-coupled, autonomous entities with limited access through (often fragmentary) *published interfaces*.
- recall web-service orchestration, choreography, etc.

The scenario:

- a palette of computational units treated as black boxes
- and a canvas into which they can be dropped
- connections are established by drawing wires

Our approach to Software Architecture

Recall our programme:

- Express architectural designs as coordination patterns for service-based designs: interfaces as sets of ports through which data flows; interaction is anonymous and handled by complex connectors; clear separation between computation and coordination
- Introduce two complementary perspectives:
  - Orc: focus on action patterns, with ephemeral interaction (in the tradition of process algebra)
  - Reo: focus on (continued) interaction as a first-class citizen
- Emphasise formal models for the key notions: interaction, behaviour, concurrency, building on top of process calculi and automata theory.

Conclusion

Where shall I go from here, please Your Majesty?
asked Alice
That depends a great deal on where you want to get to
said the Cat.

time for the mini-projects!

Big projects

- Orc as an Haskell domain-specific language (paper at [FOCLASA09])
- Reasoning about architectural patterns
  - Express typical architectural patterns in Orc or Reo
  - Build a "patter deployer" wrt applications
  - Establish patterns properties using the underlying calculi (may involve some effort in developing suitable calculi)
- A calculus of coordination schemes
  - motivation: how many catas are there in a ... cata?
  - ... start revisiting the typical functional recursive patterns

Revisiting recursive schemes

(from [Kitchin et al, 2009])

```
def fold(_, [x])   = x
def fold(f, x:xs) = f(x, fold(xs))
```

with associative reduction

```
def afold(b, [x]) = x
def afold(b, xs) =
  def step([]) = []
  def step([x]) = [x]
  def step(x:y:xs) = b(x,y):step(xs)
  afold(b, step(xs))
```

Revisiting recursive schemes

with associative and commutative reduction

```
def cfold(b, xs) =
  val c = Buffer()

  def xfer([])     = stop
  def xfer(x:xs)   = c.put(x) >> stop | xfer(xs)

  def combine(1) =  c.get()
  def combine(m) =  c.get() >x> c.get() >y>
                    ( c.put(b(x,y)) >> stop | combine(m-1))

  xfer(xs) | combine(length(xs))
```

Small projects

- A folder from two stacks in Reo and Orc; variants.
- Express typical architectural styles in both Reo and Orc Illustrate with an application.
  - Publish-Subscribe
  - Event-Bus
  - Peer-2-Peer
  - Blackboard
- Mapping Reo connectors to Orc (`homepages.cwi.nl/ proenca/webreo/`)
  - Feedback loop, Or-selector and Discriminator
  - Ordering, Sequencer and Inhibitor