

From modeling to programming: Orc - a coordination language

Luís S. Barbosa

DI-CCTC
Universidade do Minho
Braga, Portugal

May 2011

Introduction

From an **Internet scripting language** to a **component integration language** and

A programming language for concurrency

Requirements for ...

- describe entities and their interactions
- model the passage of time
- allow birth and death of entities
- allow the construction (programming) of novel interactions
- support hierarchical composition

Introduction

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

- 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

<code>add(3, 4)</code>	Add the numbers 3 and 4.
<code>CNN(d)</code>	Get the CNN news headlines for date d .
<code>Prompt("Name:")</code>	Prompt the user to enter a name.
<code>swap(l₀, l₁)</code>	Swap the values stored at locations l_0 and l_1 .
<code>Weather("Austin, TX")</code>	Find the current weather in Austin.
<code>random(10)</code>	Get a random integer in the range 0..9.
<code>invertMatrix(m)</code>	Find the inverse of matrix m .

- called like procedures, but with a [strict](#) calling discipline: to be called all its parameters must have values
- it returns [at most](#) one value, which is [published](#)

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
- ...

Sites

Special sites

- external services: Google Search, MySpace, CNN, ...
- any Java class instance
- any Orc program
- factory sites — sites that create sites: Semaphore, Buffer ...
- ...

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, synchronization, 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:

$(CNN(d) | BBC(d)) > r > email(addr, r)$

- ie, for all values published by f , initiate a separate execution of g wherein x is bound to that published value
- 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:

$email(addr, r) < r < (CNN(d) | BBC(d))$

- 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 to x , terminates and resume suspended calls
- it is the only mechanism available to block or terminate parts of a computation.
- (right associative)

Otherwise: $f; g$

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)

Examples

$h > x > \text{Println}(x) >> \text{stop}; "done"$

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

$x < x < (M()|Rwait(10) >> 0)$

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

Definitions

example:

```
def metronome(t) = signal | 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

$$((f \gg g) \langle x \rangle h) = (f \langle x \rangle h) \gg g$$

(Distributivity over $|$) if g is x -free

$$((f | g) \langle x \rangle h) = (f \langle x \rangle h) | g$$

(Distributivity over $\langle\langle$) if g is y -free

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

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

$$(f \langle x \rangle M) = f | (M \gg stop)$$

- bisimulation equalities (wrt to the Its semantics [Wehrman et al 2008])

The calculus

- almost a Kleene algebra

(Zero and $|$)

(Commutativity of $|$)

(Associativity of $|$)

(Idempotence of $|$) **NO**

(Associativity of \gg)

(Left zero of \gg)

(Right zero of \gg) **NO**

(Left unit of \gg)

(Right unit of \gg)

(Left Distributivity of \gg over $|$) **NO**

(Right Distributivity of \gg over $|$)

$$f | stop = f$$

$$f | g = g | f$$

$$(f | g) | h = f | (g | h)$$

$$f | f = f$$

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

$$stop \gg f = stop$$

$$f \gg stop = stop$$

$$signal \gg f = f$$

$$f \gg x \gg let(x) = f$$

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

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

- 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))

The functional core

Functions

```
def MailOnce(a) = Email(a, m) < m < (CNN(d)|BBC(d))
```

```
def MailLoop(a, t) = MailOnce(a) >> Rwait(t) >> MailLoop(a, t)
```

Notes

- A function is called like a procedure: it may publish many values
- Site calls are strict; Function calls non-strict
- Functions are often called concurrently: each call starts a new instance of function execution.
- If a function accesses shared data, concurrent invocations may interfere.

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 $(if(b)f \mid not(b) > c > if(c)g) < b < t$
- Bindings 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

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

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

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

example: $[1, 2]$ translates to

$\text{cons}(1, t) < s < \text{cons}(2, t) < t < \text{nil}()$

Translation into the basic calculus

Orc expressions may contain functional expressions and vice-versa

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

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

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

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

example: $[1, 2]$ translates to

$\text{cons}(1, t) < s < \text{cons}(2, t) < t < \text{nil}()$

Translation into the basic calculus

Orc expressions may contain functional expressions and vice-versa

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

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

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

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

example: $[1, 2]$ translates to

$\text{cons}(1, t) < s < \text{cons}(2, t) < t < \text{nil}()$

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

Taking time seriously

example (interrupt):

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

example (count replies within a time interval):

$def\ callCount([]) = 0$

$def\ callCount(H : T) =$
 $(H() >> 1 \mid Rtimer(10) >> 0) + callCount(T)$

Taking time seriously

example (interrupt):

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

example (count replies within a time interval):

$def\ callCount([]) = 0$

$def\ callCount(H : T) =$
 $(H() >> 1 \mid Rtimer(10) >> 0) + callCount(T)$

Fork-Join pattern

is expressed just as (P, Q) , which equivaless 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):

```
def auction([]) = 0
def auction(b : bs) =
  val bid = b.ask() | Rtimer(5000) >> 0
  max(bid, auction(bs))
```


Synchronization barrier

from

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

to

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

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 Q 's response asap, but no earlier than 1 unit from now:
 $val (u, -) = (Q(), Rtimer(1))$
- call P, Q : P result is published immediately, but Q 's result is held until the time interval has elapsed. If neither P or Q publishes a result within one second, then the first result from either is published

$val x = P$

$val y = Q$

$let(x|Rtimer(1000) >> y)$

Parallel Disjunction pattern

```
let(  
  val a = P  
  val b = Q  
  (a||b) | if(a) >> true | if(b) >> true  
)
```

- expression $(a||b)$ waits for both a and b to become available and then publishes their disjunction
- however if either a or b is true publish true immediately regardless of whether the other variable is available

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) | P(d, e)
```

Timeout

```
let(F | Rtimer(1000) >> 0)
```

Routing

example ([generalised time-out](#)):

```
val c = Buffer()
repeat(c.get) <<
  P > x > c.put(x) >> stop
  | Rtimer(1000) >> c.closenb()
```

- 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):

```
val c = Buffer()
val done = Semaphore(0)
repeat(c.get) <<
  P > x > c.put(x) >> stop
| done.acquire() >> c.closeb()
```

- 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()* to publish
- but otherwise *P* executes normally and may publish any number of values

Note

Semaphore

- absence of any locking mechanisms built into the language
- resort the *Semaphore* site to create semaphores which enable synchronization and mutual exclusion
- *Semaphore(k)* creates a semaphore with the initial value k (i.e. it may be acquired by up to k parties simultaneously)
- Given a semaphore s , $s.acquire()$ attempts to acquire s , blocking if it cannot be acquired yet because its value is zero. The call $s.release()$ releases s , increasing its value by one.

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

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.