Process algebra in motion: Orc - a coordination language

Luís S. Barbosa

DI-CCTC
Universidade do Minho
Braga, Portugal

May 2010

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

- 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: to be called all its parameters must have values
- a call 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
- ...

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

- 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 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)

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

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

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

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

- 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:
  $def\ sumto(n) = ifn < 1\ then\ 0\ else\ n + sumto(n-1)$
- 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
$(add(x, y) < x < (1 \mid 2)) < y < (2 \mid 3)$

example: $[1, 2]$ translates to
$cons(1, t) < s < cons(2, t) < t < nil()$

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

example: $[1, 2]$ translates to
$cons(1, t) < s < cons(2, t) < t < nil()$

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

example: $[1, 2]$ translates to
$cons(1, t) < s < cons(2, t) < t < nil()$

- 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):

$def\ callCount([]) = 0$

$def\ callCount(H : T) =$

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

Taking time seriously

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

example (count replies within a time interval):
$def\ callCount([]) = 0$
$def\ callCount(H : T) =$
$\quad (H() >> 1 \mid Rtimer(10) >> 0)\ +\ callCount(T)$

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):

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

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: 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

$$
\begin{array}{l}
let(\\
val\ a = P \\
val\ b = Q \\
(a||b) \mid if(a) >> true \mid if(b) >> true \\
)
\end{array}
$$

- 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):

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

Timeout

$$let(F \mid Rtimer(1000) >> 0)$$

Routing

example (generalised time-out):

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

- 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, $\mathit{Rtimer}(1000)$ responds, triggering the call $c.\mathit{closenb}()$ 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):

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

- 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*() 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 is value by one.

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

From process algebra 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

From process algebra 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
- 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.

... two invitations

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.

... two invitations

Invitation

MFES MI/MEI - http://wiki.di.uminho.pt/twiki/bin/view/Education/MFES/

Métodos Formais em Engenharia de Software

Mestrado de [Engenharia] Informática (2009/10)

Login | Print

TÓPICOS

- Benvindo a MFES
- Docentes
- Contacto
- Página principal
- Alunos
- Funcionamento
- Programa
- Sumários
- Material
- Projecto (PI)
- UCE

AVISOS

24 Mai Atenção à data da segunda Milestone do PI(II): NEW 17-Junho. A terceira (e última, com participação da indústria) será em meados de **Julho**.

22 Abr Atenção à data da primeira Milestone do PI(II): 6-Maio. A segunda será em meados de **Junho** e a terceira (e última) em meados de **Julho**.

3 Fev Atenção à calendarização das provas de avaliação e milestones (ver sumários da UCE).

9 Dez O relatório referente à primeira

Bem vindo a MFES

Bem vindo à página da edição de 2009/10 da UCE de **Métodos Formais em Engenharia de Software**. O meu nome é José Nuno Oliveira e sou o responsável por esta unidade curricular, que conta com uma equipa de docentes altamente qualificados na investigação e ensino de métodos formais aplicados ao desenvolvimento de *software*. Todos fazemos parte do Laboratório HASLab (*Formal Methods for High-Assurance Software*), em que se vem consolidando *know-how* em métodos formais desde há mais de 25 anos.

Nas suas (cerca de) 300 horas anuais de ensino em métodos científicos de programação, incluindo (cerca de) 75 horas de acompanhamento de projectos propostos por parceiros nacionais e estrangeiros, esta UCE é porventura uma das mais expressivas unidades curriculares na área, à escala europeia.

Os módulos que compoem MFES corporizam os principais vectores de que depende o projecto fiável de aplicações à escala industrial. Na sua componente teórica, a visão é a de abordar problemas de software segundo uma autêntica perspectiva de engenharia, criando modelos matemáticos sobre os quais é possível raciocinar e calcular.

Na sua componente prática, a UCE ensina a conceber e animar modelos de problemas, testando-os atempada e exaustivamente antes de se proceder à fase de cálculo e implementação, por forma a evitar erros de perspectiva ou infantilidades de concepção. Em suma: ensina-se a saber modelar e calcular, sim, mas também a saber testar e avaliar.

No seu conjunto, os conteúdos desta UCE pretendem realizar o desígnio de que é possível afixar, nos artefactos de software desenvolvidos segundo os seus princípios, o carimbo

Invitation

FACS'10 - `www.di.uminho.pt/facs2010/`

Formal Aspects of Component Software (FACS 2010)

Universidade do Minho

7th International Workshop on Formal Aspects of Component Software

October 14-16, 2010 - Guimarães (Portugal)

United Nations University

- Home
- Scope & Topics
- Committees
- Submission
- Program
- Registration
- Venue
- Accommodation
- Past Workshops

Guimarães - Portugal

Important Dates

Abstract submission
July 2, 2010

Paper submission
July 9, 2010

Acceptance Notification
August 22, 2010

Camera Ready
September 25, 2010

Workshop
October 14-16, 2010

News

21-05-2010 Sanjit Seshia and Luis Caires will be the invited speakers at FACS 2010.

03-05-2010 Proceedings will be published by Springer in the Lecture Notes in Computer Science series

23-04-2010 Committees have been published

23-04-2010 **Call for Papers:** txt