Semi formal reasoning about distributed systems using Orc

Peter Kilpatrick
Queen's University Belfast.
p.kilpatrick@qub.ac.uk

Content

- Motivation
- Orchestration Language Orc
- Orc + Muskel
- Orc + Metadata
- Other work

Motivation

- Aim: To create an abstract model of grid systems.
- Notation for description of fundamental grid operations: job placement, monitoring, job replacement.
- Orchestration Language

Orc

"Orc is a programming language and system for orchestrating distributed services.

The Orc model assumes that basic services, like sequential computation and data manipulation, are implemented by primitive sites.

Orc provides constructs to orchestrate the concurrent invocation of sites to achieve a goal – while managing time-outs, priorities, and failure of sites or communication."

http://www.cs.utexas.edu/~wcook/projects/orc/

Orc

Jayadev Misra and William R. Cook

Computation Orchestration: A Basis for Wide-Area Computing

"… where the delays associated with communication, unreliability and unavailability of servers, and competition for resources from multiple clients are dominant concerns."

Site

- Site call (possibly with parameters) is simplest Orc expression.
- A call to a site may update the site and respond (publish a *single* value).
- Different calls to the same site may return different values.
- May never respond (remain silent).
- Site call:
  - Function call;
  - Method call of an object;
  - Monitor procedure;
  - …

Special Sites

- if b returns a signal if b is true and remains silent otherwise.
- RTimer(t) always responds with a signal after t time units.
- let always returns (publishes) its argument
- 0 never responds (used to terminate expressions)

Operators

- Parallel Composition: e | f
- Sequential composition: e >x> f
- Asymmetric Composition: e where x :∈ f
- …and recursion

Parallel composition

e| f

evaluates e and f in parallel.

Both evaluations may produce replies.

Evaluation of the expression returns the merged output streams of e and f

Parallel composition - Example

BBC | CNN

May produce 0, 1 or 2 outputs.

Parallel composition - Notation

$$(| \; i: 0 \leq i \leq 2: Pi)$$

is an abbreviation for

$$(P0 \mid P1 \mid P2)$$

Sequential composition

e >x> f(x) evaluates e, receives a result x, calls f with parameter x.

If e produces two results, say x and y, then f is evaluated twice, once with argument x and once with argument y.

The abbreviation

e >> f

is used for e >x> f when evaluation of e is independent of x.

(Cf. Universal quantification)

Sequential composition - Example

BBC >x> Email(a,x)

Email is called 0 or 1 time

(BBC | CNN) >x> Email(a,x)

Email is called 0, 1 or 2 times

where (asymmetric parallel composition)

e where x :∈ f

begins evaluation of both e and f in parallel.

Expression e may name x in some of its site calls.

Evaluation of e may proceed until a dependency on x is encountered; evaluation is then delayed.

The first value delivered by f is returned in x; evaluation of e can proceed and the thread f is terminated.

(Cf. Existential Quantification)

where (asymmetric parallel composition) - example

Email(a,x) where x :∈ (BBC | CNN)

Email is called 0 or 1 time.

where - notational convention

Write

```
(f where x :∈ g) where y :∈ h
```

as

```
f where x :∈ g
  where y :∈ h
```

or as

```
(f where x :∈ g, y :∈ h)
```

Expression Definition

SendOnce(a,d) =

Email(a,x) where x :∈ (BBC(d) | CNN(d))

Expression call

- An expression call is syntactically similar to a site call
- An expression call may publish many values.
- Calling an expression starts a new instance of that expression. Thus, f >> f refers to two different instances of f.
  (Cf. several calls to a site, S, result in the calls being queued at a single instance of the site.)
- Expression calls are non-strict. That is, evaluation begins when it is called, even if some parameters are (as yet) undefined.

Expression call - example

SendOnce(a,d) =

Email(a,x) where x :∈ (BBC(d) | CNN(d))

Sendmail = SendOnce(a,d) where a :∈ getAddress

d :∈ getDate

Example

```
tally([]) = let(0)
tally(x:xs) =
      add(u,v)
            where
                  u :∈ (x(m) >> let(1) | RTimer(10) >> let(0))
                  v :∈ tally(xs)
```

tally(L) publishes the number of sites in L that respond within 10 time units.

m is a (fixed) argument.

if-then-else

if b then S else T

may be coded in Orc as:

if(b) >> S | if(¬b) >> T

(remember: if is a site call)

Example

Repeated Polling

```
TPoll(E,t,f) =def
  (if flag >> let(s)) |  (if ¬ flag >> TPoll(E,f(t),f))
      where (flag,s) :∈ {   E >s> let (true,s)
                          | Rtimer(t) >>let(false,failure)
                        }
```

fork-join parallelism

Spawn two threads, M and N, and resume when both threads complete.

```
(let(u,v)
      where u :∈ M
      where v :∈ N
)
```

Synchronisation

There is no special mechanism for synchronisation in Orc – be careful!

A where expression may be used to effect synchronisation.

Assume M >>f and N >>g are to be executed independently but f and g are to be synchronized by starting them only after both M and N have completed.

```
(let(u,v)
        where u :∈ M
        where v :∈ N
)
>> (f | g)
```

Channels

Orc does not have the notion of a channel. A channel must be implemented by a site (outside Orc).

Assume channels are FIFO and unbounded.

Channel c has two methods: c.get and c.put which are called as site calls from an Orc expression.

c.put(m) adds m to the end of the channel and publishes a signal.

c.get publishes the value at the head of c and removes it from c if the channel is non-empty; otherwise the caller is suspended until the channel is non-empty.

Processes

Orc does not have an explicit notion of process. A process may be represented by an expression that names channels with are shared with other processes (expressions).

Assume c and e are input and output channels, respectively.

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

Note: this processes publishes nothing, although it writes its output on a channel, e.

Processes

To publish each value written on e, define:

```
Q(c,e) = c.get >x> Compute(x) >y>
                    (let(y) | e.put(y) >> Q(c,e))
```