Extras

Concurrency and Distributed Systems

January 2023

Contents

Sequential composition

A particular kind of hiding occurs—occasionally, in specifications—when we have a transition from one phase of behaviour to another.

This special case has its own operator: the sequential composition operator `;`.

The left-hand and right-hand arguments are both process-valued expressions.

Interpretation: sequential composition

If `P` and `Q` are processes, then

```
P ; Q
```

is a process that behaves as `P` until that process terminates successfully, upon which it behaves as `Q`.

Termination

We write `SKIP` to denote a process that does nothing except terminate successfully.

Step law: sequential composition

If

```
  P = [] e : A @ e -> P(e)
```

then*

```
  P ; Q = [] e : A @ e -> ( P(e) ; Q )
```

For any process Q,

```
  SKIP ; Q = Q
```

*provided that `e` does not appear free in Q

Algebra

```
STOP ; P = STOP

P ; (Q ; R) = (P ; Q) ; R

P ; (Q |~| R) = (P ; Q) |~| (P ; R)

(P |~| Q) ; R = (P ; R) |~| (Q ; R)
```

Distributed termination

```
SKIP [ A || A ] SKIP = SKIP
```

Example

```
Juice = glass -> juice -> SKIP

Cereal = bowl -> cereal -> milk -> SKIP

Coffee = cup -> coffee -> sugar -> SKIP


Breakfast = Juice ; Cereal ; Coffee
```

Example

```
Breakfast =
  Juice ;
    Cereal ;
      Coffee
```

Example

```
Breakfast =
  glass -> juice -> SKIP ;
    bowl -> cereal -> milk -> SKIP ;
      cup -> coffee -> sugar -> SKIP
```

Example

```
Breakfast =
  glass -> ( juice -> SKIP ;
    bowl -> cereal -> milk -> SKIP ;
      cup -> coffee -> sugar -> SKIP )
```

Example

```
Breakfast =
  glass -> juice -> ( SKIP ;
    bowl -> cereal -> milk -> SKIP ;
      cup -> coffee -> sugar -> SKIP )
```

Example

```
Breakfast =
  glass -> juice ->
  ( bowl -> cereal -> milk -> SKIP ;
      cup -> coffee -> sugar -> SKIP )
```

Example

```
Breakfast =
  glass -> juice ->
    bowl -> cereal -> milk ->
      cup -> coffee -> sugar -> SKIP
```

Traces: skip

```
traces(SKIP) = {<>, <tick>}
```

where `tick` is a event name reserved for this purpose

Traces: sequential composition

```
traces(P ; Q) =
  union(
   { trP | trP <- traces(P),
           not member(tick,set(trP)) },
   { trPˆtrQ | trP <- traces(P),
               not member(tick,set(trP)),
               member(trPˆ<tick>,traces(P)),
               trQ <- traces(Q) } )
```

Example

```
traces(Breakfast) =
  { <>,
    <glass>,
    <glass,juice>,
    <glass,juice,bowl>,
    <glass,juice,bowl,cereal>,
    <glass,juice,bowl,cereal,milk>,
    <glass,juice,bowl,cereal,milk,cup>,
    <glass,juice,bowl,cereal,milk,cup,coffee>,
    <glass,juice,bowl,cereal,milk,cup,coffee,sugar>,
    <glass,juice,bowl,cereal,milk,cup,coffee,sugar,tick> }
```

Example

```
ParallelBreakfast =
  ( Juice
    [ {glass,juice} || {bowl,cereal,milk,cup,coffee,sugar} ]
    ( Cereal
      [ {bowl,cereal,milk} || {cup,coffee,sugar} ]
      Coffee ) )
```

Example

```
traces(ParallelBreakfast) =
  { <>, <glass>, <bowl>, <cup>,
    <glass,juice>, <glass,bowl>, <glass,cup>,
    ... }
```

traces of parallel breakfast…

Interrupt

Sometimes we may require a more drastic form of sequential composition.

We may use /\ to describe a composition in which the second process simply interrupts the first—permanently.

After the second process has started, the first makes no further contribution to the pattern of behaviour: it is simply discarded.

Interpretation: interrupt

If `P` and `Q` are processes, then

```
P /\ Q
```

is a process that behaves as `P` while offering, before and after every event, the choice to behave as `Q` instead.

If either process terminates, then the whole process terminates immediately.

Example

```
  Pinball = start -> (Game /\ Tilt) ; Pinball

     Game = launch -> out -> BallTwo

  BallTwo = launch -> out -> BallThree

BallThree = launch -> out -> SKIP


     Tilt = tilt -> SKIP
```

Example

```
Game /\ Tilt =
  ( launch -> out -> SKIP ) /\ (tilt -> Tilt)
```

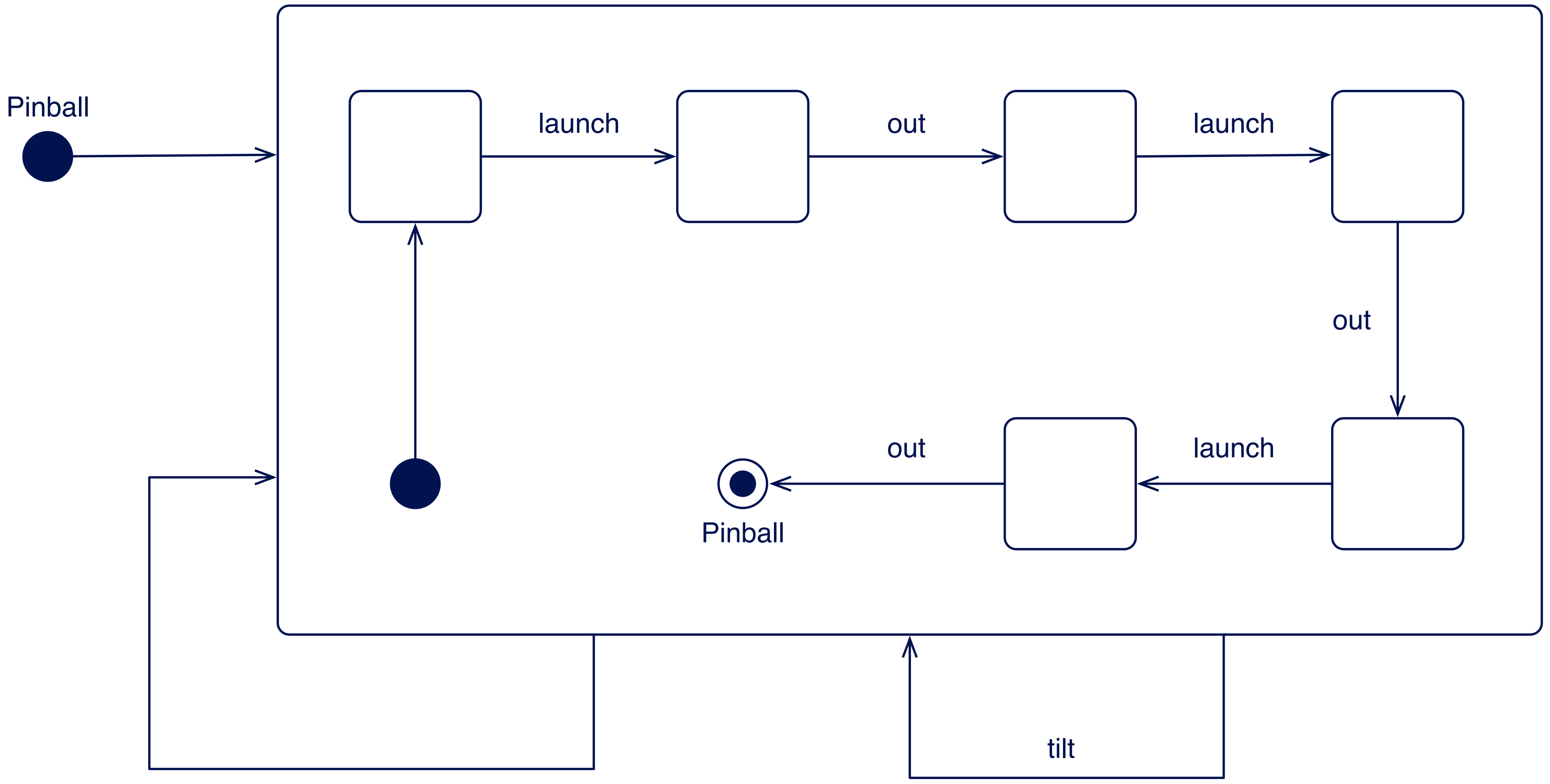
Example

```
Game /\ Tilt =
  launch ->
    ( out -> SKIP ) /\ (tilt -> Tilt)
  []
  tilt -> SKIP
```

Example

```
Game /\ Tilt =
  launch ->
    ( out -> SKIP
      []
      tilt -> SKIP ) /\ (tilt -> Tilt)
  []
  tilt -> SKIP
```

```
Game /\ Tilt =
  launch ->
    ( out ->
        ( launch ->
            ( out ->
                ( launch ->
                    ( out -> SKIP
                      []
                      tilt -> SKIP )
                  []
                  tilt -> SKIP )
              []
              tilt -> SKIP )
          []
          tilt -> SKIP )
      []
      tilt -> SKIP )
  []
  tilt -> SKIP
```


Pinball
launch
out
launch
out
launch
out
Pinball
tilt

Interleaving

We may find it useful to use the same name for two different transactions.

If we do not need to distinguish between them, for the purposes of the model, then giving them the same name will simplify both our description and our reasoning.

In doing so, we may need to allow them to happen independently, on each side of a parallel combination. To do this, we need a different operator: $|||$

Interpretation: interleaving

If P and Q are processes, then

```
P ||| Q
```

is a process that behaves as the interleaved parallel composition of P and Q.

In this kind of parallel composition, P and Q may perform any event without reference to the other, even if the other process is able—or ready, even—to engage in an event with the same name.

Step law: interleaving

If

```
P = [] e : A @ e -> P(e)
Q = [] e : B @ e -> Q(e)
```

then

```
P ||| Q = [] e : union(A,B) @ e ->
             if member(e,diff(A,B)) then
               P(e) ||| Q
             else if member(e,diff(B,A)) then
               P ||| Q(e)
             else
               (P(e) ||| Q) |~| (P ||| Q(e))
```

Algebra

```
        P ||| Q = Q ||| P
P ||| (Q ||| R) = (P ||| Q) ||| R

P ||| (Q |~| R) = (P ||| Q) |~| (P ||| R)
(P |~| Q) ||| R = (P ||| R) |~| (Q ||| R)

  SKIP ||| SKIP = SKIP
```

Example

```
    JuiceA = glass -> Juice -> SKIP
   CerealA = bowl -> cereal -> milk -> SKIP
   CoffeeA = cup -> coffee -> milk -> SKIP

BreakfastA = JuiceA ||| CoffeeA ||| CerealA

  milk -> milk -> SKIP [FD=
    BreakfastA \ { glass, juice, bowl, cereal, cup, coffee }
```

Partial interleaving

It may be that some of the event names appearing on both sides of a parallel combination are intended to denote the same transaction, and should be shared, to happen at the same time for both processes, while others denote different transactions and should be interleaved.

We may achieve this effect using the partial interleaving operator

```
[|  |]
```

The operator will be useful even if there are no events to be interleaved, as a way to describe a parallel combination in terms of the intersection of alphabets, rather than the alphabets themselves.

Interpretation: partial interleaving

If P and Q are processes, and S is a set of events, then

```
P [| S |] Q
```

is a process that behaves as the partially-interleaved parallel combination of P and Q, in which:

- events from S are allowed only when allowed by both P and Q; they happen for both processes, or not at all
- events outside S can be performed independently by either P or Q; they can happen for either process

Step law: partial interleaving

If

```
P = [] e : A @ e -> P(e)
Q = [] e : B @ e -> Q(e)
```

then

```
P [| S |] Q =
  ([] e : diff(A,union(B,S)) @ e -> (P(e) [| S |] Q))
   []
  ([] e : diff(B,union(A,S)) @ e -> (P [| S |] Q(e)))
   []
  ([] e : Inter({A,B,S}) @ e -> (P(e) [| S |] Q(e))
   []
  ([] e : diff(inter(A,B),S) @ e ->
      ((P(e) [| S |] Q) |~| (P [| S |] Q(e))))
```

Renaming

We may wish to use the same name for (two transactions currently described by) two different events. We can achieve this using the renaming operator `[[ ]]`.

This operator is useful also in creating replicated parallel combinations: re-labelling events to produce many different instances of the same process.

Interpretation: renaming

If `P` is a process and `L` is a list of renamings of the form

```
from <- to | s, t
```

where `from` and `to` are channels and `s` and `t` are generators and constraints (as in a set comprehension), then

```
P [[L]]
```

is a process that behaves as `P` except that every `from` event now appears as `to`.

Algebra

```
     STOP [[L]] = STOP
  (P ; Q) [[L]] = P [[L]] ; Q [[L]]
 (P [] Q) [[L]] = P [[L]] [] Q [[L]]
(P |~| Q) [[L]] = P [[L]] |~| Q [[L]]
```

Step law: functional renaming

If

```
  P = [] e : A @ e -> P(e)
```

and `L` corresponds to the function `f` on events, then

```
  P [[L]] =
    [] e' : { f(e) | e <- A } @ e' ->
      |~| e : { e | e <- A,  f(e) = e' } @ P(e)[[L]]
```

Example

If `[[L]]` corresponds to the functional renaming `f`, such that `f(a) = c` and `f(b) = c` then

```
(a -> P [] b -> Q)[[L]]

=

c -> (P[[L]] |~| Q[[L]])
```

Renaming and parallel composition

Renaming does not distribute through parallel composition.

We may think of renaming as an operation that is applied at the external interface to a parallel composition of processes.

At the interface, some of the transactions have new names; inside the interface, they are constrained as before.

Summary

- Sequential composition
- Interleaving
- Renaming

Index