Introduction to Communicating Sequential Process (CSP) (Lecture 4)

Mannheim, September 2007

Contents

- Traces
- Concurrency

Traces

- Traces of a process
- Notation
- Operators, After, Initials
- Semantics
- Specifications and satisfaction

Traces of a Process

- If P is a process then the *traces* of P, *traces* P, denotes the set of all finite sequences of events P can perform, in the order in which it can perform them.
- In notation from discrete mathematics,

$$traces\ P \subseteq (\alpha P)^*$$

Traces of a Process: Notation

- Sequence of symbols, separated by commas and enclosed in angular brackets:
  - $\langle\rangle$ - empty sequence containing no events
  - $\langle a \rangle$ - singleton; a is the only event.
  - $\langle a_0, a_1, \ldots, a_n \rangle$ - general case

Traces of a Process: Examples

- *traces STOP = {<>}*
- *traces Student = { <>, <snore>, <snore,snore>, ...,*
  <snore, ..., snore>, ...}
- *traces μX: {coin,choc}. (coin->choc->X) =*
  $\{ t \mid \exists n : N \bullet t \leq (coin, choc)^n \}$
- *traces μX: {a,b}. ((a->X)[] (b->STOP)) =*
  $\{ \langle a \rangle^n \mid n \in N \} \cup \{ \langle a \rangle^n \hat{} \langle b \rangle \mid n \in N \}$

Traces of a Process : Exercises

- 1. *traces (a -> b -> STOP)*
- 2. *traces (a -> STOP* [] *b -> STOP)*

Traces of a Process : Exercises

- 1. *traces (a -> b -> STOP) =*
  {<>,<a>,<a,b>}

- 2. *traces (a -> STOP* [] *b -> STOP) =*
  {<>,<a>,<b>}

Operations on Traces

- Catenation

  The *catenation* of traces *s* and *t* is the trace consisting of *s* followed by *t*, s ^ t .

  – C.f. functional programming's *s* ++ *t* .

- Example: <*m*,*a*,*n*> ^ <*y*,*e*,*a*,*r*,*s*> = <*m*,*a*,*n*,*y*,*e*,*a*,*r*,*s*>.
- Definition:

  <>^*t* = *t*

  (*a* : *s*) ^ *t* = *a* : (*s* ^ *t*).

- Catenation is associative with identity <>.

Operations on Traces

- Repeat Catenation

  Trace t followed by itself, a total of n times, is written t^n

- Example: <ho>^{3} = <ho, ho, ho> .
- Definition:

  $t^0 = < >$

  $t^{n+1} = t \wedge (t^n)$.
- Repeated catenation satisfies $t^n \wedge t^m = t^{n+m}$.

Operations on Traces

- Laws of Catenation

  $s\hat{}\langle\rangle = \langle\rangle\hat{}s = s$

  $s\hat{}(t\hat{}u) = (s\hat{}t)\hat{}u$

  If $s\hat{}t = s\hat{}u$ then $t = u$

  If $s\hat{}t = u\hat{}t$ then $s = u$

  If $s\hat{}t = \langle\rangle$ then $s = \langle\rangle$ and $t = \langle\rangle$

Operations on Traces

- Head

  $head(a: s) = a$

- Tail

  $tail(a: s) = s$

Thus, s and t are equal if

$$s = t \text{ iff } (s = t = <> \vee (head(s) = head(t) \wedge tail(s) = tail(t))$$

Operations on Traces

- Length

  The length of the sequence t is denoted $\# t$.

  Example: $\#\langle x,y,x\rangle = 3$

  – Laws

  $\# \langle\rangle = 0$

  $\#\langle x\rangle = 1$

  $\# (a : t) = 1 + \# t$

  $\#(s \frown t) = (\#s) + (\#t)$ ...

Operations on Traces

- Restriction

  The traces t when restricted to symbols in the set A is denoted

  $$t \lceil A$$

  Example : $\langle x,y,x \rangle \lceil \langle x \rangle = \langle x,x \rangle$

  Laws: $\langle\rangle \lceil A = \langle\rangle$

  $\langle x \rangle \lceil A = \langle x \rangle$ *if* $x \in A$

  $\langle y \rangle \lceil A = \langle\rangle$ *if* $y \notin A$

  $(s \frown t) \lceil A = (s \lceil A) \frown (t \lceil A)$

  In general,

  $\langle\rangle \lceil A = \langle\rangle$

  $(a : t) \lceil A = (a: (t \lceil A)) \lhd a \in A \rhd (t \lceil A)$

Operations on Traces

- Frequency

  The frequency of event a in trace s, $s{\downarrow}a$, is given by $s{\downarrow}a = \#(s \lceil \{a\})$.

  Example: $s{\downarrow}\langle x,y,x\rangle = 2$

Operations on Traces

- Quantified prefix

  $s \preceq_n t$ iff $(s \leq t) \wedge (\#t - \#s \leq n)$

  Example:

  $\langle m,a \rangle \preceq_1 \langle m,a,n \rangle$

  $\langle m,a \rangle \preceq_2 \langle m,a,n \rangle$

  $\langle m,a \rangle \preceq_0 \langle m,a,n \rangle$ *(false)*

After

If t is a trace of process *P* then

P after t

denotes a process which behaves like *P* after it has already engaged in trace *t*

Examples:

VMS after coin = choc -> VMS

VMS after <coin,choc> = VMS

After

- Laws
  - *P after <> = P*
  - *P after (s ^ t) = (P after s) after t*
  - *(?x : A → P(x)) after <a> = P(a) if a □ A .*

Initials

If P is a process then the initials of P, *initials* P, denotes the set of events which it may initially perform

$$initials\ P = \{\ a \mid \langle a \rangle \in traces\ P\}$$

Examples:

$initials\ VMS = \{coin\}$

$initials\ STOP = \{\}$

Traces of a Process

- Home exercises
  - Laws of initials
  - Laws for the operators given
  - Other operators and laws
  - Hoare's book

Traces: Semantics

- A semantics provides a model of processes: a meaning for each process.
- Denotational semantic: each proces *P* is modelled by its alphabet/universe and set of traces, *traces P*.

Traces Denotations

$traces\ STOP = \{ <> \}$

$traces\ (a \rightarrow P) = \{<>\} \cup \{a{:}t \mid t \in traces\ P\}$

$traces\ (a \rightarrow P \mid b \rightarrow Q) = traces\ (a \rightarrow P) \cup traces\ (b \rightarrow P)$

$traces\ (x{:}\ B \rightarrow P(x)) = \{<>\} \cup \{x{:}t \mid x \in B \wedge t \in traces\ P(x)\}$

$traces\ (P \lceil\rceil Q) = (traces\ P) \cup (traces\ Q)$

$traces\ (P\ [\ ]\ Q) = (traces\ P) \cup (traces\ Q)$

$traces\ (P \triangleleft b \triangleright Q) = (traces\ P) \cup (traces\ Q)$

$traces(\mu X : A.F(X)) = \bigcup_{n \geq 0} traces(F^n(STOP_A))$

Traces Denotation

$$traces(\mu X : A.F(X)) = \bigcup_{n \geq 0} traces(F^n(STOP_A))$$

That formula for recursion formalises the increasing unfolding of the recursion:

Stop represents no iterations

F(Stop) represents one iteration

F^2*(Stop)* represents two iterations

...

F^n Stop represents *n* iterations

...

Traces Denotation

$$traces\left(\mu X : A.F\left(X\right)\right) = \bigcup_{n \geq 0} traces(F^n(STOP_A))$$

- Recall

  $Student = \mu X \bullet (snore \rightarrow X).$

- Now

  $traces\ (F^n\ (Stop)) = \{<snore>^j \mid 0 \leq j \leq n\}$

  hence

  $traces\ Student = \bigcup\{traces\ (F^n\ (Stop)) \mid n \in N\}$

  $= \bigcup\{\{<snore>^j \mid 0 \leq j \leq n\} \mid n \in N\}$

  $= \{<snore>^n \mid n \in N\}.$

Traces Denotation

- **Problem:** the traces semantics does not distinguish the processes we know to have different behaviour. It is too coarse, not fully abstract. For example,

  $traces\ (P\ []\ Q) = (traces\ P) \cup (traces\ Q)$
  $traces\ (P \lceil\rceil\ Q) = (traces\ P) \cup (traces\ Q)$

- **Solution:** to introduce the failures/divergences semantics (we will study later).

Traces: Healthiness Conditions

- For any process P

  $traces\ P \neq \{\}$ because $\langle\rangle \in traces\ P$

  $(s \frown t) \in traces\ P$, implies $s \in traces\ P$

  $traces\ P \subseteq (\alpha P)^*$

Traces

Specify and verify *safety properties*

nothing bad will happen

It does not ensure *liveness*.

cannot ensure that an event will occur

Specification

- A *specification* of a product is a description of the way it is intended to behave.
- Process – the relevant observation of its behaviour is the trace of events that occur up to a given moment in time (denoted *tr*).

Specification: Examples

- The owner of a vending machine requires it not to make a loss:.

  $NOLOSS = (\#(tr \lceil \{choc\}) \leq \#(tr \lceil \{coin\})$

- The user of the vending machine requires it to be fair: no more than one coin's credit is ever required

  $FAIR = ((tr{\downarrow}coin) \leq (tr{\downarrow}choc) + 1)$

- Observe that a traces specification cannot demand that an event occurs (e.g. choc). It specifies safety but not liveness.

Specification: Examples

- The vending-machine maker must satisfy both owner and user and so must meet the specification.

$$VMSPEC = NOLOSS \wedge FAIR$$

$$= 0 \leq ((tr{\downarrow}coin) - (tr{\downarrow}choc)) \leq 1$$

Specification: Satisfaction

- If *P* is a product which meets a specification *S*, we say that *P satisfies S,* denoted *P sat S.*
- This means that every possible observation of the bahaviour of *P* is described by *S.*

$$P \; sat \; S(tr) = \forall \, tr \in traces(P) \bullet S(tr)$$

Satisfaction: Example

- A specification that requires that it cannot happen more events *down* than events *up* is expressed by:

  $$S(tr) = tr \downarrow down \leq tr \downarrow up$$

  Let P a process:

  $$up \rightarrow up \rightarrow down \rightarrow Stop$$

  $traces\ (up \rightarrow up \rightarrow down \rightarrow Stop) =$

  $$\{<>, <up>, <up,up>, <up,up,down>\}$$

  Observe that any trace tr of P satisfies $S(tr)$.

Satisfaction Laws

1. *P sat true*
2. *STOP sat (tr=<>)*
3. If *P sat S* and *P sat T* then *P sat (S and T)*
4. If *P sat S* and *S => T* then *P sat T*
5. If *P sat S(tr)* then *(c->P) sat (tr = <> $\vee$ (head(tr) = c $\wedge$ S(tail(tr)))*

...

Satisfaction Proof: Example

Show that $b \text{ -> } a \text{ -> } STOP$ sat $tr\downarrow a \leq tr\downarrow b$

1. $a \text{ -> } STOP$ sat $(tr = \langle\rangle \vee (head(tr) = a \wedge (tail\ (tr) = \langle\rangle)))$

   (laws 2 and 5)

2. $a \text{ -> } STOP$ sat $tr\downarrow a \leq 1$

   (law 4, as $(tr = \langle\rangle \vee (head(tr) = a \wedge (tail\ (tr) = \langle\rangle))) \Rightarrow tr\downarrow a \leq 1$)

3. $b \text{ -> } a \text{ -> } STOP$ sat $(tr = \langle\rangle \vee (head\ (tr) = b \wedge tail(tr)\downarrow a \leq 1))$

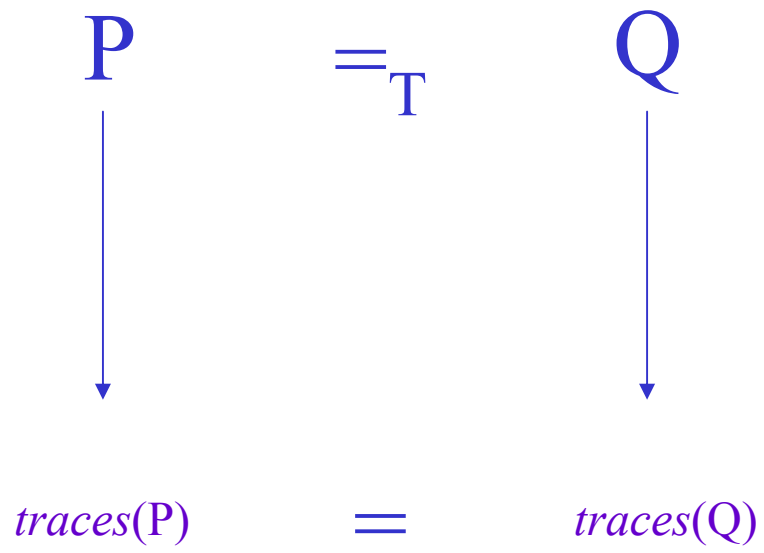
   (law 5)

4. $b \text{ -> } a \text{ -> } STOP\ sat\ (tr\downarrow a \leq tr\downarrow b) \vee (tr\downarrow b \geq 1 \wedge tr\downarrow a \leq 1)$

   (law 4)

5. $b \text{ -> } a \text{ -> } STOP\ sat\ tr\downarrow a \leq tr\downarrow b$ (law 4)

Traces Model: Equality



Traces Model: Refinement

For universe Σ, the traces model of processes over Σ is defined to consist of the partial order

$(\{E \mid E \text{ is a nonempty prefix-closed set of traces over } \}, \sqsubseteq)$.

There

$P \sqsubseteq_{\text{traces}} Q$ is modelled by *traces* $P \supseteq$ *traces* Q

Refinement => removal of nondeterminism

Refinement

A process Q satisfies a specification described by P if any trace of Q is also a trace of P

We say that P *is refined* by Q or Q *refines* P

$P \sqsubseteq_T Q$ means $traces(Q) \subseteq traces(P)$

The FDR tool verify refinement.

Refinement: Example

COPY refines $B^{\infty}_{<>}$

$B^{\infty}_{<>} \sqsubseteq$ COPY

COPY = left ? x:T -> right !x -> COPY

$B^{\infty}_{<>}$ = left ? x:T -> $B^{\infty}_{<x>}$

$B^{\infty}_{s^{\wedge}<y>}$ = (left ? x:T -> $B^{\infty}_{<x>^{\wedge}s^{\wedge}<y>}$

| right!y -> B^{∞}_{s})

Both $B^{\infty}_{<>}$ and COPY satisfy a buffer specification

buffer: tr ↓ right ≤ tr ↓ left

Observe the traces of COPY is a particular case of possible traces of $B^{\infty}_{<>}$

Refinement: Exercises

Let *DOUBLECOPY = left ? x:T -> right !x -> right !x ->DOUBLECOPY*
Verify if *Q* refines *P*, or vice-versa, in the following cases:

1. *P = COPY, Q = DOUBLECOPY*

2. *P = B^{∞}, Q = DOUBLECOPY*

Laws of Refinement

$P \sqsubseteq P$ — $\sqsubseteq$-reflex

$P \sqsubseteq Q \wedge Q \sqsubseteq R \Rightarrow P \sqsubseteq R$ — $\sqsubseteq$-trans

$P \sqsubseteq Q \wedge Q \sqsubseteq P \Rightarrow P = Q$ — $\sqsubseteq$ -anti-sim

$RUN \sqsubseteq_T P$ — $\sqsubseteq$ -bottom

$P \sqsubseteq_T STOP$ — $\sqsubseteq$-miracle

$P\ sat\ S(tr) \wedge P \sqsubseteq Q \Rightarrow Q\ sat\ S(tr)$ — $\sqsubseteq$-spec

Refinement

- Transitivity property:

  Allows a gradual generation of the implementation from the specification

  $Spec \sqsubseteq P_1 \sqsubseteq P_2 \sqsubseteq \ldots \sqsubseteq P_n \sqsubseteq Impl$

- Does not ensure *liveness*

  $P \sqsubseteq STOP \qquad RUN \sqsubseteq P$

Concurrency

- Interaction
- Alphabetised parallel composition
- Deadlock: the dining philosophers
- Interleaving
- Generalised parallel composition

Parallelism

- Represents the parallel execution of two or more processes:
  - Independent control flow
  - Eventual interactions
- Used to connect components of a distributed system

Parallelism

CSP offers several possibilities that vary according to the way the interaction happens:

- Interaction

  P || Q

- Alphabetised parallel composition

  P [X||Y] Q

- General parallel composition

  P [|X|] Q

- Interleaving

  P ||| Q

Interaction

- If P and Q are processes with common alphabet then their interaction

$$P \parallel Q$$

is the process which performs an event iff that event occurs simultaneously in both P and Q.

Interaction

$P \parallel Q$

- ofers the initial events common to P and Q, and wait until the synchronization happens;
- after the synchronization of event a, the process behaves as $P' \parallel Q'$, where P' and Q' behaves, respectively, as P and Q after the ocurrence of event a.

Interaction: Example

Assume that *P* and *Q* has the same alphabet and

P = μX: {coin,choc,toffee}. (coin->(choc->X | toffee -> X))

Q = μX: {coin,choc,toffee}. (choc->X | toffee -> X)

Então

P || Q = STOP *Deadlock!*

Interaction: Example

But considering process *R*

$P = \mu X: \{coin,choc,toffee\}. (coin->(choc->X \mid toffee -> X))$

$R = \mu X: \{coin,choc,toffee\}. (choc->X \mid toffee -> X \mid coin->choc->X)$

Then

$P \parallel R = \mu X: \{coin,choc,toffee\}. (coin->choc->X)$

Laws of Interaction

- Interaction is commutative and associative.

  $P \,||\, Q = Q \,||\, P$

  $P \,||\, (Q \,||\, R) = (P \,||\, Q) \,||\, R)$

- *STOP* is a unit: $P \,||\, STOP = STOP$
- The synchronization is expressed by:

  $(a \rightarrow P) \,||\, (a \rightarrow Q) = a \rightarrow (P \,||\, Q)$

  $(a \rightarrow P) \,||\, (b \rightarrow Q) = Stop, \text{ if } a \neq b,$

  Can be generalized as

  $(?a : A \text{ -> } P(a) \,||\, (?b : B \text{ -> } Q(b))$

  =

  $?c: A \cap B \text{ -> } (P(c) \,||\, Q(c))$

Alphabetised Parallel Composition

- For subsets A and B of Σ and processes P and Q over Σ, their alphabetised parallel composition

$$P\ [A \parallel B]\ Q$$

  is the process in which P may perform events only in A, Q may perform events only in B, but P and Q must agree on events in the intersection $A \cap B$.

- Interaction is a particular case of this composition where $A = B$.

Alphabetised Parallel Composition

- Example:

  (a - > P) [{a} || {a,b}] (a -> Q | b -> R)

  may result in:

  a-> *(P [{a} || {a,b}] Q)*

  or *(b->((a -> P) [{a} || {a,b}] R)*

  Therefore,

  (a - > P) [{a,b} || {a,b}] (a -> Q | b -> R)

  is *a->(P [{a,b} || {a,b}] Q)*

Alphabetised Parallel Composition

(a->b->b->STOP)[{a,b} || {b,c}] (b->c->b->STOP)

= <only a can occur>

a -> ((b -> b-> STOP) [{a,b} || {b,c}] (b->c->b->STOP))

= <event b occurs, because is a common event >

a -> b ->((b-> STOP) [{a,b} || {b,c}] (c->b->STOP))

= <only c happens>

a -> b ->c ->((b-> STOP) [{a,b} || {b,c}] (b->STOP))

=<event b occurs, because is a common event >

a -> b ->c ->b->(STOP [{a,b} || {b,c}] STOP)

= <finishes the execution>

a -> b ->c ->b->STOP

Alphabetised Parallel Composition Example

$A = \alpha NOISYVM = \{coin, choc, clink, clunk, toffee\}$

clink – sound of the coin dropping into the machine

clunk – sound of the machine after delivering a chocolate

This machine has run out off toffee.

$NOISYVM = (coin \rightarrow clink \rightarrow choc \rightarrow clunk \rightarrow NOISYVM)$

The consumer prefer toffee, and he utters (*curse*) when he fails to get it; he then has to take a chocolate instead.

$B = \alpha CUST = \{coin, choc, curse, toffee\}$

$CUST = (coin \rightarrow (toffee \rightarrow CUST \mid curse \rightarrow choc \rightarrow CUST))$

Alphabetised Parallel Composition Example

NOISYVM = (coin→clink →choc →clunk →NOISYVM),
A =αNOISYVM = {coin,choc,clink,clunk,toffee}
CUST = (coin →(toffee →CUST | curse →choc →CUST)),
B =αCUST = {coin,choc,curse,toffee}

(NOISYVM[A || B] CUST)
= <coin belongs to both alphabet>
(coin→((clink →choc →clunk →NOISYVM)
[A||B] (toffe -> CUST | curse →choc →CUST))
= <toffee is common so cannot occur, but clink and curse can occur>
(coin→(clink →(choc →clunk →NOISYVM) [A||B] (curse →choc →CUST)
| curse ->(clink →choc →clunk →NOISYVM) [A||B] (choc →CUST))
= <choc is common so cannot occur, but clink and curse can occur>
(coin→(clink →curse ->(choc →clunk →NOISYVM) [A||B] (choc →CUST)
| curse ->clink ->(choc →clunk →NOISYVM) [A||B] (choc →CUST))
= <choc é is common so occurs>

Alphabetised Parallel Composition Example

$NOISYVM = (coin \rightarrow clink \rightarrow choc \rightarrow clunk \rightarrow NOISYVM),$

$A = \alpha NOISYVM = \{coin, choc, clink, clunk, toffee\}$

$CUST = (coin \rightarrow (toffee \rightarrow CUST \mid curse \rightarrow choc \rightarrow CUST)),$

$B = \alpha CUST = \{coin, choc, curse, toffee\}$

$= (coin \rightarrow (clink \rightarrow curse \rightarrow (choc \rightarrow clunk \rightarrow NOISYVM)\ [A||B]\ (choc \rightarrow CUST)$

$\mid curse \rightarrow clink \rightarrow (choc \rightarrow clunk \rightarrow NOISYVM)\ [A||B]\ (choc \rightarrow CUST))$

= <choc is common so occurs>

$(coin \rightarrow (clink \rightarrow curse \rightarrow choc \rightarrow (clunk \rightarrow NOISYVM)\ [A||B]\ CUST$

$\mid curse \rightarrow clink \rightarrow choc \rightarrow (clunk \rightarrow NOISYVM)\ [A||B]\ CUST)$

=<coin is common so does not occur, but clunk can occur>

$(coin \rightarrow (clink \rightarrow curse \rightarrow choc \rightarrow clunk \rightarrow (NOISYVM\ [A||B]\ CUST)$

$\mid curse \rightarrow clink \rightarrow choc \rightarrow clunk \rightarrow (NOISYVM\ [A||B]\ CUST))$

= <definition of recursion>

$\mu X.(coin \rightarrow (clink \rightarrow curse \rightarrow choc \rightarrow clunk \rightarrow X)$

$\mid (curse \rightarrow clink \rightarrow choc \rightarrow clunk \rightarrow X))$

Alphabetised Parallel Composition Exercise

- 1. A pay and display parking permit machine accepts cache and issues ticket and change. Use alphabetised parallel composition to model the machine behaviour.

Alphabetised Parallel Composition Exercise

- 1. A pay and display parking permit machine accepts cache and issues ticket and change. Use alphabetised parallel composition to model the machine behaviour.

TICKET = cash -> ticket -> TICKET

CHANGE = cash -> change -> CHANGE

MACHINE =

TICKET [{cash,ticket}||{cash,change}] CHANGE

Alphabetised Parallel Composition Exercise

- 2. Use alphabetised parallel composition to express the board exercise already done.

$\alpha C = \{up,down,left,right\}$

$C = (up \rightarrow D \mid left \rightarrow A \mid right \rightarrow E)$

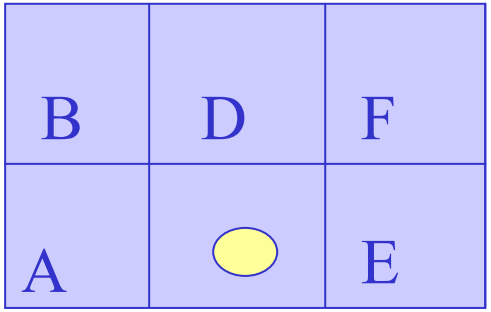
$D = (down \rightarrow C \mid left \rightarrow B \mid right \rightarrow F)$

$E = (up \rightarrow F \mid left \rightarrow C)$

$F = (down \rightarrow E \mid left \rightarrow D)$

$A = (up \rightarrow B \mid right \rightarrow C)$

$B = (down \rightarrow A \mid right \rightarrow D)$



Alphabetised Parallel Composition Exercise

- 2. Use alphabetised parallel composition to express the board exercise done before.

$\alpha P = \{up,down\}$

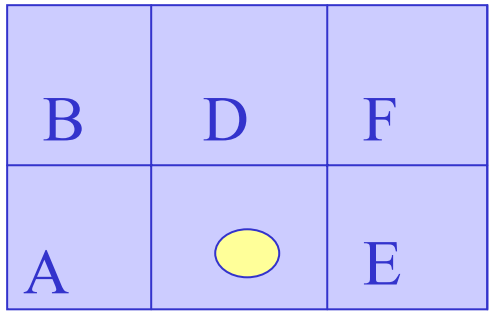
$P = (up \rightarrow down \rightarrow P)$

$\alpha Q = \{left,right\}$

$Q = (left \rightarrow right \rightarrow Q \mid right \rightarrow left \rightarrow Q)$

$\alpha C = \{up,down,left,right\}$

$C = P \, [\{up,down\} \| \{left,right\}] \, Q$



Alphabetised Parallel Composition: Laws

- Similarly to interaction is associative and commutative.
- It distributes through conditional choice

  $P\ [A \parallel B]\ (Q \sqcap R) = (P\ [A \parallel B]\ Q) \sqcap (P\ [A \parallel B]\ R)$

  $P\ [A \parallel B]\ (Q \lhd b \rhd R) = (P\ [A \parallel B]\ Q) \lhd b \rhd (P [A \parallel B]\ R)$
- The general law is:

  Assume $C = (A \cap (X \setminus Y)) \cup (B \cap (Y \setminus X)) \cup (A \cap B \cap X \cap Y)$

  $(?a : A \rightarrow P(a)[X \parallel Y]\ (?b : B \rightarrow Q(b))$

  =

  $?c: A \cap B \rightarrow (P(c)\ [X \parallel Y]\ Q(c)) \lhd c \in X \cap Y \rhd (P(c)[X \parallel Y]\ (?b : B \rightarrow Q(b))$

  $\lhd c \in X \rhd (?a: A \rightarrow P(a)[X \parallel Y] Q(b))$

Alphabetised Parallel Composition and Interaction: Traces

- Which are the traces expressions of:
  - $P \parallel Q$?
  - $P \; [A \parallel B] \; Q$?

Indexed Alphabetised Parallel Composition

- The binary parallel composition operator may be generalised to model the interaction of several components.

- If *I* is a finite set of indices such that *P(i)* and *A(i)* are defined for each *i* in *I* then

  || *i:{1..N}* @ *[A(i)] P(i)*

  describes the combination of components where each *P(i)* has alphabet *A(i)*.

  So,

  || *i:{1..2}* @ *[A(i)] P(i) = P(1) [A(1) || A(2)] P(2)]*