6SENG006W Concurrent Programming

Week 3

Further FSP Language Features
&
Modelling & Defining Concurrent Programs in FSP

Further FSP Language Features, Modelling & Defining Concurrent Programs in FSP

This lecture introduces further features of FSP & concurrent FSP processes, the following topics will be covered:

- FSP processes with: *constants*, *number ranges*, *indexed actions* & *indexed processes*.
- FSP processes that have *parameters*.
- *Conditional* processes, using: `if-then-else`.
- "*Guarded*" processes, using: `when( BExp ) x -> P`
- FSP processes: `END` & `ERROR`.
- *Modelling Concurrency*.
- *Concurrent FSP processes* using: "`||`" parallel operator.
- *Alphabet Diagrams*.
- Modelling Concurrency using *"interleaving"*.

PART I

FSP Processes with: Indexed Actions, Constants, Number Ranges, Indexed (Local) Processes & Parameters

Processes with Indexed Actions

In order to model processes & actions that can take multiple values, both *action labels* & *local processes* may be **indexed** in FSP.

Notation:

- FSP *indexes* are integers.
- Usual *integer arithmetic operators* are supported on indexes: `+, -, *, /, %`.
- The indices must always have a *finite range of values*, this ensures the models are finite & can be analysed.

These features greatly increase the *expressive power* of FSP

To illustrate these indexing features, we shall use: a simple buffer process, a summing two numbers process & one that combines index features.

The processes `BUFF`, `BUFF1`, `BUFF2` & `BUFF3` all represent a buffer that can contain a single value, i.e. a 1 element buffer.

The `SUM` processes inputs two numbers & outputs their sum.

We also use an index features process `INDEXES`.

A Version of BUFF using Indexed Action Labels

The buffer process BUFF inputs a value in the range 0 to 3 & then outputs it.

```
BUFF
BUFF = ( in[ i : 0..3 ] -> out[ i ] -> BUFF ) .
```

"**i**" is the *indexed action label*, resulting in an *indexed action*: "in[**i**]" & the alphabet for BUFF:

```
alphabet( BUFF ) = { in[0],   in[1],   in[2],   in[3],
                     out[0],  out[1],  out[2],  out[3] }
```

Process BUFF1 is an *equivalent definition* in which the *choice* between input values is stated explicitly.

```
BUFF1
BUFF1 = (   in[0] -> out[0] -> BUFF1
          | in[1] -> out[1] -> BUFF1
          | in[2] -> out[2] -> BUFF1
          | in[3] -> out[3] -> BUFF1 ) .
```

Notation: The *scope* of an *action label index*, e.g. "i", is the *choice element in which it occurs*.

State Machine for BUFF & BUFF1

The state machine for both buffer processes BUFF & BUFF1 *is the same* & is given in Fig. 3.1.

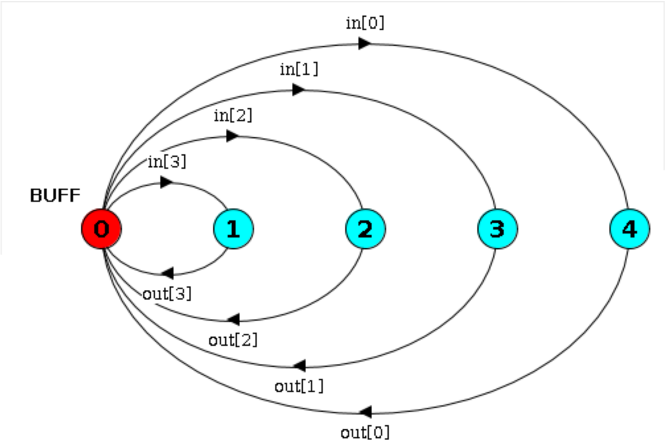

Figure : 3.1 BUFF/BUFF1 state machine.

Defining Constants & Number Ranges in FSP

As with all types of programming it is good *Software Engineering* practice to use *constants* in your program.

Similarly, for the ability to *limit the value range of a variable*, e.g. define a *range of number values* that a variable can have.

Luckily FSP supports these two features as follows:

```
Constants & Ranges

const MIN_INT = -3                          // Constants
const MAX_INT = 3                           //

range INT      = 0 .. MAX_INT               //
range BIG_INT  = MIN_INT .. MAX_INT         // Ranges
range HUGE_INT = -100 .. 100                //
```

Notation: The constant & range identifiers must start with an *UPPER CASE* letter. The *scope* of these definitions, e.g. `INT` & `MAX_INT`, is the whole of an FSP program, i.e. they can be used in *all the process definitions in a file*.

In other words `const` & `range` declarations are *global*.

A Version of `BUFF` using Indexed Local Process

`BUFF2` is an equivalent definition & uses two *index variables*:

- "`i`" for an *indexed action* – "`in[i]`", e.g. `in[0]`, .., `in[3]`
- "`j`" for an *indexed local process* – "`STORE[j]`", e.g. `STORE[0]`, .., `STORE[3]`

BUFF2

```
const MAX_INT = 3
range INT = 0 .. MAX_INT

BUFF2 = ( in[ i : INT ] -> STORE[ i ] ),

STORE[ j : INT ] = ( out[ j ]  -> BUFF2 ) . // j = i
```

Notation: The *scope* of a *process index variable* e.g. "`j`", is the *process definition*.

Note that all three buffer processes have the **same FSM** & so are **"semantically equivalent"**.

And obviously, they all have the *same alphabet* as `BUFF`.

Example using Multiple Indexes: SUM1

Both process & action labels may have more than one index.

For example, process SUM1 inputs two values, **a** & **b** then outputs their sum.

SUM1 uses one *process index* **s** (TOTAL1[**s**]) & two *action indexes* **a** & **b** (in[**a**][**b**]).

SUM1
```
const N   = 1
range IN  = 0..N
range OUT = 0..2*N

SUM1 = ( in[ a:IN ][ b:IN ] -> TOTAL1[ a + b ] ) ,

TOTAL1[ s : OUT ] = ( out[ s ] -> SUM1 ) . // s = a+b
```

The alphabet for SUM1 is:

```
alphabet( SUM1 ) = { in[0][0], in[0][1], in[1][0], in[1][1],
                     out[0],   out[1],   out[2] }
                 = { in[0..1][0..1],  out[0..2] }
```

FSM for example process `SUM1`

A small value is used for the constant `N` in the definition of `SUM1` so that the graphic representation, see Fig. 3.2, is displayable.

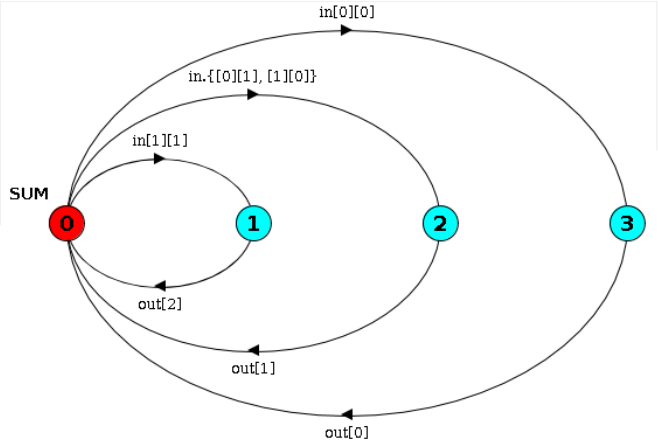

Figure : 3.2 `SUM1` state machine.

Example: SUM2

SUM2 is equivalent to the SUM1 process.

SUM2 uses:

- two *action indexes* **a** & **b** – "in[**a**][**b**]"
- two *process indexes* **x** & **y** – "TOTAL2[**x**][**y**]"

SUM2

```
const N = 1

range IN = 0..N

SUM2 = ( in[ a:IN ][ b:IN ] -> TOTAL2[a][b] ) ,

TOTAL2[ x:IN ][ y:IN ] = ( out[x + y] -> SUM2 ) .
```

Example SUM2: Fully Expanded Indexes

We can give the equivalent *fully expanded indexes* version of SUM2 by making explicit the *action indexes* & the *process indexes* as follows:

Expanded SUM2

```
SUM2 = (    in[ 0 ][ 0 ] -> TOTAL2[0][0]
          | in[ 0 ][ 1 ] -> TOTAL2[0][1]
          | in[ 1 ][ 0 ] -> TOTAL2[1][0]
          | in[ 1 ][ 1 ] -> TOTAL2[1][1] ) ,

TOTAL2[ 0 ][ 0 ] = ( out[ 0 ] -> SUM2 ) ,
TOTAL2[ 0 ][ 1 ] = ( out[ 1 ] -> SUM2 ) ,
TOTAL2[ 1 ][ 0 ] = ( out[ 1 ] -> SUM2 ) ,
TOTAL2[ 1 ][ 1 ] = ( out[ 2 ] -> SUM2 ) .
```

Note: in this version do not need to define the constant N or the range IN.

Check that these two versions are in fact the same using ltsa.

Processes using Indexes

The following processes illustrate various features of using *action indexes*:

INDEX1

```
const N = 5
range INPUT = 0..N

INDEXES_1 = ( in[ i : INPUT ]
              -> i_add_2        [i + 2]
              -> i_subtract_1 [i - 1]
              -> i_multiply_4 [i * 4]
              -> i_divide_2    [i / 2]
              -> i_remainder_3[i % 3] -> STOP ) .
```

INDEX3

```
INDEXES_3 = ( in[ i : INPUT ]
              -> add       [i][2][i + 2]
              -> subtract [i][1][i - 1]
              -> multiply [i][4][i * 4]
              -> divide    [i][2][i / 2]
              -> remainder[i][3][i % 3] -> STOP ) .
```

Process Parameters

> **Definition:** *Process Parameters*
>
> Processes may be *parameterized* so that they may be described in a general form & modelled for a particular parameter value.

For instance, the single-slot buffer `BUFF` process illustrated in Fig. 3.1 can be described as a *parameterized process* `BUFF3` for values in the range `0` to **`N`** as follows:

BUFF3

```
const DEFAULT = 3

BUFF3( N = DEFAULT ) = ( in[ i : 0..N ]
                          -> out[ i ] -> BUFF3 ) .
```

Notation: *parameters* **must have a DEFAULT value** & **must start with an uppercase letter** & the scope of the parameter is the process definition.

Examples of Processes taking Several Parameters

Example using 2 parameters:

PARAMS_2

```
const N = 5

/* PARAMS: LL - lower limit, UL - upper limit */

PARAMS_2( LL = 0, UL = N )
         = ( in[ i : LL..UL ]
               -> add_2        [i][i + 2]
               -> subtract_1   [i][i - 1]
               -> multiply_4   [i][i * 4]   -> STOP ) .
```

Example using 5 parameters:

PARAMS_5

```
/* PARAMS: AN, SN & MN used for arithmetic & indexes */

PARAMS_5( LL = 0, UL = N, AN = 2, SN = 1, MN = 4 )
         = ( in[ i : LL..UL ]
               -> add        [i][AN][i + AN]
               -> subtract   [i][SN][i - SN]
               -> multiply   [i][MN][i * MN]   -> STOP ) .
```

PART II

FSP Process Commands

Using Booleans in FSP

We begin by looking at how the *Boolean* type is represented in FSP.

In FSP there is **no** Boolean type, but to represent the two *truth values*: true & false it uses numbers: *true* is ≥ 1 (non-zero) & *false* is 0.

Notation: FSP supports the usual logical operators: "**&**" (*and*), "**|**" (*or*) & "**!**" (*not*).

The following processes illustrate their use:

Booleans & Logical Operators

```
const TRUE  = 1
const FALSE = 0

range BOOL  = FALSE .. TRUE

BOOL_AND = ( in[ p : BOOL ][q : BOOL ]
               -> and [p][q][ p & q] -> STOP ) .

BOOL_OR = ( in[ p : BOOL ][q : BOOL ]
               -> or [p][q][ p | q] -> STOP ) .

BOOL_NOT = ( in[ p : BOOL ] -> not [p][ !p ] -> STOP ) .
```

Conditional Processes: `if-then-else`

FSP has the usual forms of *conditional* choice statement:

```
if ( BExp ) then P

if ( BExp ) then P else Q
```

With `BExp` the usual *boolean expression*.

The usual comparison relations can be used: `<, <=, >, >=, ==, !=`

But instead of `P` & `Q` being program statements they are FSP *processes* in the true & false clauses.

Definition: *if-then-else*

The choice

```
if ( BExp ) then P else Q
```

means that when the *boolean expressions* `B` is:

- *true* the process `P` is then selected,
- *false* the process `Q` is then selected.

Example if-then-else Process

As an example consider the following IS_ZERO process, that tests if the number input x is zero:

if–Conditional

```
const ZERO    = 0
const MAX_INT = 3

range INT = ZERO .. MAX_INT

IS_ZERO  = ( in[ x : INT ] ->

              if ( x == ZERO )
              then
                   ( isZero    -> IS_ZERO )
              else
                   ( isNotZero -> IS_ZERO )
          ) .
```

Exercise: animate IS_ZERO in ltsa.

Guarded Actions

It is often useful to define particular actions as conditional, depending on *the current state of the machine*.

We use "*boolean guards*" to indicate that a particular action can:

ONLY be "selected" if its GUARD is true.

> **Definition:** *Guarded Actions*
>
> The choice
>
> ```
> (   when ( BExp ) x -> P
>   | y -> Q              )
> ```
>
> means that **when** the *boolean guard* `BExp` is:
>
> - *true* – the actions `x` & `y` are both eligible to be chosen,
> - *false* – the action `x` **cannot** be chosen, but only `y` is available.

The concept of *guarded commands* was invented by E.W. Dijkstra.

Guarded Command Laws

The following laws illustrate how a guarded command is evaluated:

```
GUARDED Laws

const TRUE  = 1
const FALSE = 0

 ( when(TRUE) x -> P | y -> Q )  =  ( x -> P | y -> Q )

( when(FALSE) x -> P | y -> Q )  =  ( y -> Q )

           ( when(TRUE) x -> P ) = ( x -> P )

          ( when(FALSE) x -> P ) = STOP
```

Example Guarded Action Process

As an example consider the following `GUARDED` process, which uses the value input "`b`" as the *boolean guard*:

GUARDED

```
const FALSE = 0
const TRUE  = 1

range BOOL  = FALSE .. TRUE

GUARDED
 = ( in[ b : BOOL ] ->
     (   when ( b  )  guardTrue  -> GUARDED

       | when ( !b )  guardFalse -> GUARDED
     )
   ) .
```

Example with “Overlapping Guards”: Counter

The `COUNT` process encapsulates a count variable, it can be increased by `inc` operations & decreased by `dec` operations.

The count is not allowed to exceed `N` or be less than zero.

But if `0 < i < N` it offers a choice of either action, as *both guards are true*.

BUFF

```
COUNT ( N = 3 ) = CT[0],

CT[ i : 0..N ]  = (  when( i < N )  inc -> CT[i+1]
                   | when( i > 0 )  dec -> CT[i-1] ) .
```

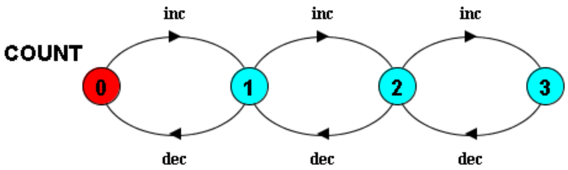

Figure : 3.3 `COUNT` state machine.

Example: Countdown Timer

COUNTDOWN models a timer which counts down to zero & then beeps.

Once started, it outputs a tick sound each time it decrements the count & a beep when it reaches zero.

At any point, the countdown may be aborted by a stop action.

```
COUNTDOWN (N = 3) = ( start -> CTDN[N] ),

CTDN[ i : 0..N ]  = (   when( i > 0  ) tick -> CTDN[i-1]
                      | when( i == 0 ) beep -> STOP
                      | stop -> STOP                      ) .
```

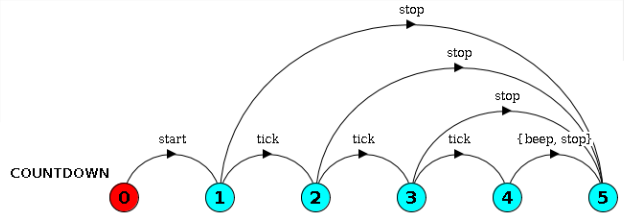

Figure : 3.4 COUNTDOWN (N = 3) state machine.

PART III

Special FSP Processes: END *&* $ERROR$

FSP Processes: END & ERROR

Definition: *Process END*

is a special predefined process that *"successfully terminates"* then engages in no further actions.

END is similar to STOP, i.e. neither *can perform any actions & just terminates*.

However, END is used to represent a process that *successfully terminates*, whereas STOP is used to represent a process that does **NOT successfully terminate**, i.e. has **deadlocked**.

The SKIP process does nothing except successfully terminates.

```
SKIP = END .
```

Fig. 3.5 depicts the state machine for SKIP, it is denoted by **E**, not a number, but like STOP has **no** actions.

Figure : 3.5 SKIP (END) state machine.

Example of END: "Just 1 Doughnut"

We can use END in our FSP process definitions in the same way as we have used STOP.

For example, in the unusual situation that Homer is satisfied with just 1 Doughnut:

```
WEIRD_HOMER = ( pickup_Doughnut -> eat -> mmmmmm -> END ) .
```

However, the more usual case is:

```
NORMAL_HOMER = ( pickup_Doughnut -> eat
                                 -> mmmmmm
                                 -> NORMAL_HOMER ) .
```

Exercise: try the 2 Homers out in the LTSA tool.

Faulty Process: `ERROR`

Definition: *Process* *`ERROR`*

is a special predefined process that represents a *"faulty"* or *"broken"* process.

`ERROR` is similar to `END` & `STOP`, i.e. *none of them can perform any actions*.

However, `ERROR` is used to represent a process that behaves **"chaotically"**.

The following process represents a faulty clock:

```
FAULTYCLOCK = ( tick -> tock -> ERROR ) .
```

In `FAULTYCLOCK`'s state machine the `ERROR` state is denoted by "**−1**".

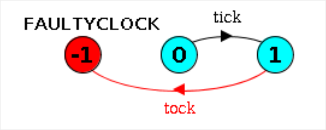

Figure : 3.6 `FAULTYCLOCK`'s state machine.

PART IV

Modelling Concurrency in FSP

“Real” Concurrent Execution

The execution of a concurrent program consists of multiple processes active at the same time.

Where each process is executing a sequential program.

A process progresses by submitting a sequence of instructions to a processor for execution.

If the computer has multiple “*physical*” processors then instructions from a number of processes, equal to the number of physical processors, can be executed at the same time.

For example, Fig. 3.7, shows three processes A, B & C executing in parallel on 3 processors.

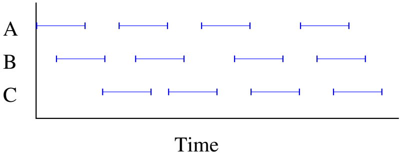

Figure : 3.7 Real Concurrent Processes

This is sometimes referred to as *parallel* or *real concurrent* execution.

“Interleaving” (or “Pseudo”) Concurrent Execution

Usually there are *more active processes* than *processors*, in this case, the *processors* are *switched* between *processes*.

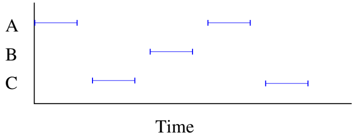

Figure : 3.8 Process Switching

In Fig. 3.8 a *single processor* is switching between three processes A, B & C; each makes progress, but *instructions from only one process at a time can be executed.*

The switching between processes occurs voluntarily or in response to interrupts, such as I/O completion.

Processor switching does not affect the *order* of instructions executed by each process.

The processor executes a sequence of instructions which is an *interleaving* of the instruction sequences from each process, also known as *pseudo-concurrency*.

Concurrent Systems Modelling Issues

We use the terms *parallel* & *concurrent* interchangeably.

Usually do not distinguish between *real* & *pseudo-concurrent* execution.

Since, in general, the same programming principles & techniques are applicable to both physical (real) concurrent & interleaved (pseudo) execution.

In the previous lecture, we modelled a process abstractly as a state machine that proceeds by executing atomic actions, which transform its state.

The execution of a process generates a sequence (trace) of atomic actions.

We now examine how to model systems consisting of *multiple processes*.

Before we can model a concurrent program, we need to resolve how to deal with the following two issues:

- the *relative speed of execution of each process*;
- *"concurrency"*.

Relative Speed of Execution

The first issue to consider is how to model:

> *"The speed at which one process executes relative to another".*

This depends on many factors, such as:

- the *number of processors* – 1 or several;
- the *scheduling strategy* – how the operating system chooses the next process to execute.

FSP Approach to Relative Speed of Execution

We choose **not** to model relative speed but simply state that processes execute at *arbitrary relative speeds*.

This means that a process can take an arbitrarily long time to proceed from one action to the next, i.e. we abstract away from execution time.

Advantages:

- ensures that the concurrent programs we design *work correctly independently of these factors*;
- increases the *portability of concurrent programs*;
- *verify properties independently* of the particular hardware & operating system.

Disadvantage:

- can say *nothing about the real-time properties of programs*.

How to Model Concurrency

The next issue is how to model concurrency or parallelism.

Definition: *Concurrent Actions*

An action `a` is *concurrent with another action* `b` if a model permits the actions to occur in either order:

`a -> b` or `b -> a`.

Question: Is it necessary to model *real* & *pseudo* concurrency differently?

Answer: **No**, we will model all concurrent execution using *interleaving*, whether or not implementations run on a single or multiple processors.

We will illustrate what we mean by an *"interleaving model of concurrency"* after we have introduced the FSP parallel operator.

PART V

Concurrency in FSP

Parallel Composition – "`P || Q`"

Definition: *Parallel Composition*

If `P` & `Q` are processes then

`( P || Q )`

represents the concurrent execution of `P` & `Q`.
The operator "`||`" is the parallel composition operator.

For example, the following two process can be used to define a composite (parallel) process:

```
ITCH     = ( scratch -> STOP ) .
CONVERSE = ( think -> talk -> STOP ) .

|| CONVERSE_ITCH = ( ITCH || CONVERSE ) .
```

Notation: *Composite process definitions* are always preceded by "`||`" to distinguish them from primitive process definitions.

CONVERSE_ITCH's State Diagrams

The state diagrams for the three processes ITCH, CONVERSE & CONVERSE_ITCH are given in Fig. 3.9.

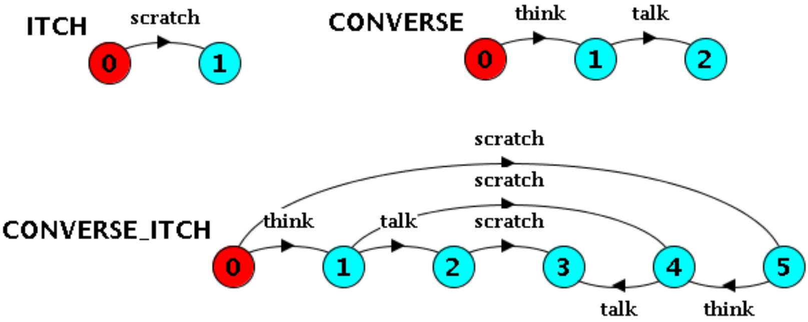

Figure : 3.9 Composition CONVERSE_ITCH.

From Fig. 3.9, it can be seen that the action scratch is concurrent with both think & talk as the model permits these actions to occur in any order while retaining the constraint that think must happen before talk.

Composite Process State Machines

The state machine representing a parallel composition `P||Q`, is formed by the *Cartesian product* (*"all possible combinations"*) of the state machines of its constituent processes `P` & `Q`.

`ITCH`, `CONVERSE` state machines are related to `CONVERSE_ITCH`'s as follows:

CONVERSE_ITCH		ITCH	CONVERSE
0	<0,0>	0	0
1	<0,1>	0	1
2	<0,2>	0	2
3	<1,2>	1	2
4	<1,1>	1	1
5	<1,0>	1	0

For example, if `ITCH` is in **state(i)** & `CONVERSE` is in **state (j)**, then this combined state is represented by `CONVERSE_ITCH` in **state (<i, j>)**.

So if `CONVERSE` has performed the `think` action & is in **state (1)** & `ITCH` performs its `scratch` action & is in **state (1)** then the state representing this in the composition is **state (<1,1>)**, i.e. `CONVERSE_ITCH`'s state 4.

(Check this using `ltsa`.)

Alphabet Diagrams

An *alphabet diagram* displays the *alphabets of processes* in a *Venn* diagram.

This is extremely useful when the processes are *composed in parallel*, e.g.

```
alphabet( P ) = ( a1, .., an, c1, .., ck }
alphabet( Q ) = ( b1, .., bm, c1, .., ck }
||SYSTEM = ( P || Q ) .
```

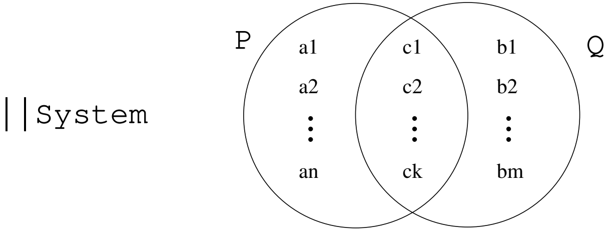

Figure : 3.10 Alphabet Diagram for two Processes `P` & `Q` in Parallel.

If the processes in a composition have actions in common, these actions are said to be *shared*. E.g. `P`'s & `Q`'s `ci`'s. (Covered in the next lecture.)

If a process in a composition has an action that is only in its alphabet, i.e. only it performs it, this action is said to be *non-shared*. E.g. `P`'s `ai`'s & `Q`'s `bi`'s.

CONVERSE_ITCH's Alphabet Diagram

The alphabets for the three processes are:

$$\begin{aligned} \mathtt{alphabet(ITCH)} &= \{\, \mathtt{scratch} \,\} \\ \mathtt{alphabet(CONVERSE)} &= \{\, \mathtt{think, talk} \,\} \\ \mathtt{alphabet(ITCH)} \cap \mathtt{alphabet(CONVERSE\,)} &= \{\,\} \end{aligned}$$

The alphabet diagram for ITCH, CONVERSE & CONVERSE_ITCH is:

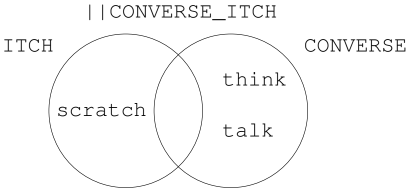

Figure : 3.11 CONVERSE_ITCH's Alphabet Diagram.

From Fig. 3.11, it can be seen that the action scratch is only performed by ITCH, think & talk are only performed by CONVERSE.

So in this concurrent process CONVERSE_ITCH there are **no shared actions**, therefore the actions can be performed concurrently.

Clock Radio Example

The following processes model a clock radio which incorporates two independent activities: a clock which `tick`'s & a radio which can be switched `on` & `off`.

```
CLOCK = ( tick -> CLOCK ) .

RADIO = ( on -> off -> RADIO ) .

|| CLOCK_RADIO = ( CLOCK || RADIO ) .
```

The state machine for the composition is depicted in Fig. 3.12.

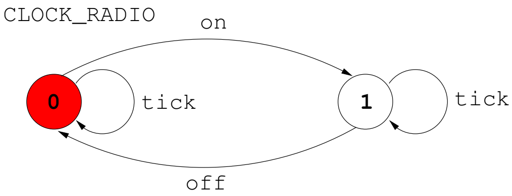

Figure : 3.12 Composition CLOCK_RADIO.

Some Properties of Parallel Composition ||

The *parallel composition* operator `||` obeys some simple but important *algebraic laws*:

```
Commutative:        (P||Q) = (Q||P)

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

Commutative: means that the order in which the processes (e.g. `P` & `Q`) are combined using `||` does not matter.
Maths Example: $2 + 3 \;=\; 3 + 2$

Associative: means that the order in which the processes (e.g. `P`, `Q` & `R`) are combined in pairs using `||` & brackets does not matter.
Maths Example: $(2 + 3) + 4 \;=\; 2 + (3 + 4)$

Taken together these mean that the *brackets can be dispensed with* & the *order that processes appear in the composition is irrelevant*.

Maths Example: $(2 + 3) + 4 \;=\; (4 + 3) + 2 \;=\; 3 + 4 + 2$

PART VI

Modelling Concurrency using Interleaving

Modelling Concurrency using Interleaving

One of the simplest ways to model the concurrent execution of two or more processes is to use the notion of *interleaving*.

So consider the two *sequential* FSP processes `P` & `Q` that perform the following sequence of events:

```
P = ( a1 -> a2 -> ... -> an -> STOP ) .

Q = ( b1 -> b2 -> ... -> bn -> STOP ) .
```

Then an *interleaving* of `P` & `Q` is *any sequence of* `ai`'s & `bi`'s such that:

1. `a1` *precedes* `a2` *precedes* `a3` ...
2. Similarly for the `bi`'s.
3. The `ai`'s & `bi`'s can be in *any order*.
   For example

   ```
   < a1, b1, a2, b2, ... >
   < a1, a2, b1, a3, ... >
   ```

An *interleaving* is a *trace* of the parallel composition of the (two) processes.

Interleavings of `P||Q` as a Trace Tree

Consider the following two FSP processes `P` & `Q`:

```
P = ( a1 -> a2 -> STOP ) .
Q = ( b1 -> b2 -> STOP ) .
||PQ = ( P || Q ) .
```

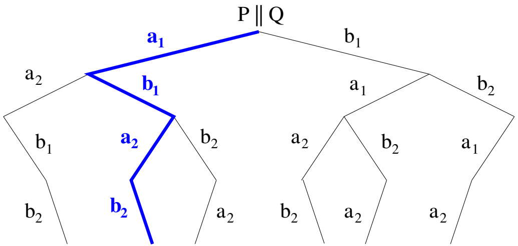

Figure : 3.13 Trace tree for `P||Q`.

The list of longest traces is:

`< a1, a2, b1, b2 >,` **`< a1, b1, a2, b2 >`**`,` `< a1, b1, b2, a2 >,`
`< b1, a1, a2, b2 >,` `< b1, a1, b2, a2 >,` `< b1, b2, a1, a2 >`

Exercise: verify that these are traces using `ltsa`.

Example 2: Interleaving Traces

Consider the following two FSP processes `P` & `Q`:

```
P = ( a1 -> a2 -> a3 -> a4 -> STOP ) .
Q = ( b1 -> b2 -> b3 -> b4 -> STOP ) .
|| PQ = ( P || Q ) .
```

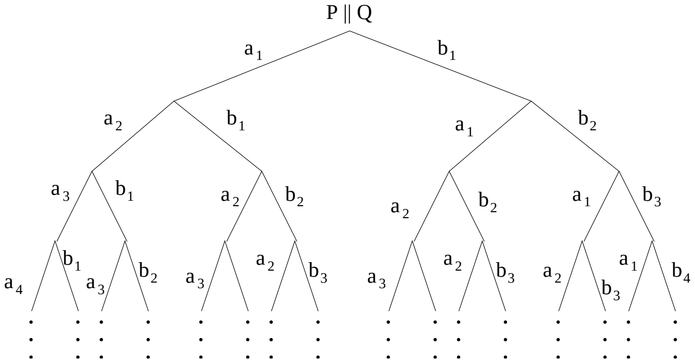

Figure : 3.14 Trace tree for `P||Q`.

Exercise: Construct some of the longest traces by animating it in `ltsa`.

Example 3: CONVERSE_ITCH Traces

See Fig. 3.9 for the state machine representing CONVERSE_ITCH.

```
ITCH     = ( scratch -> STOP ) .
CONVERSE = ( think -> talk -> STOP ) .

|| CONVERSE_ITCH = ( ITCH || CONVERSE ) .
```

The composition generates all possible interleavings of the traces of its constituent processes:

```
traces( ITCH ) = { <>, <scratch> }

traces( CONVERSE ) = { <>, <think>, <think, talk> }

traces( CONVERSE_ITCH )
  = { <>,
      <scratch>, <scratch, think>, <scratch, think, talk>,
      <think>,   <think, scratch>, <think, scratch, talk>,
                 <think, talk>,    <think, talk, scratch>   }
```