6SENG006W Concurrent Programming

Week 4

Modelling Interacting Concurrent Processes in FSP

Modelling Interacting Concurrent Processes in FSP

This Week 4 lecture focuses on how FSP models systems of *interacting concurrent processes* & will cover the following topics:

- *Interacting concurrent processes* – "*shared actions*".
- Using *"synchronisation"* to achieve "*Mutual Exclusion*".
- Remaining FSP operators & features
- *Inter-Process Communication*
  – "Synchronised Message Passing"
- *Structure Diagrams*

Note: see Chapter 3 of the recommended book:
Concurrency: State Models and Java Programs, (2nd Edition).

PART I

Interacting Concurrent Processes: Shared & Non-Shared Actions

Systems of Non-Shared Action Processes

The concurrent FSP program examples in the previous lecture were all compositions of processes that:

did not have any actions in common, i.e. no “shared actions”.

For example, recall the following `CONVERSE_ITCH` system:

CONVERSE_ITCH

```
ITCH     = ( scratch -> STOP ) .

CONVERSE = ( think -> talk -> STOP ) .

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

CONVERSE_ITCH's Alphabet Diagram

It is clear from its *alphabet diagram* for the system of two processes that their alphabets are *disjoint*:

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

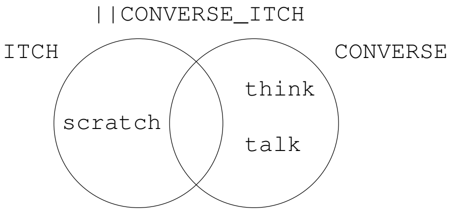


Figure : 4.1 CONVERSE_ITCH's Alphabet Diagram.

So ITCH & CONVERSE **do not share any actions**.

Shared & Non-Shared Actions

Shared & *non-shared* actions are used to model different aspects of a system.

In particular:

Shared actions: are used to model *process interactions*.

Non-Shared actions: are used to model a *process' independent actions*.

The *formal name* for these two types of actions are:

- *Synchronous* for shared actions.
- *Asynchronous* for non-shared actions.

Synchronous & Asynchronous Actions

Synchronous: for shared actions.

- The performance of a shared action must be *synchronised* between the processes that can perform it, i.e. those that have it in their alphabet.
- A *synchronised* action must be executed at the same time by all the processes that participate in it.
- Therefore, if a *synchronised* action can not be executed by at least one of the processes that "share" it, then it can **not be executed**.

Asynchronous: for non-shared actions.

- The performance of a non-shared action is solely dependent on the single process that can perform it, i.e. is performed *asynchronously*.
- Further, *asynchronous* actions may be arbitrarily interleaved.

Sharing/Synchronising Actions Example 1

The following example is a *composition of processes* that *share (synchronise)* the action **meet**.

```
BILL_BEN

BILL = ( play -> meet -> STOP ) .
BEN  = ( work -> meet -> STOP ) .

|| BILL_BEN = ( BILL || BEN ) .
```

BILL_BEN's alphabet diagram shows the *synchronised* actions, given by:

$$\mathtt{sync}(\mathtt{BILL\_BEN}) = \mathtt{alphabet}(\mathtt{BILL}) \cap \mathtt{alphabet}(\mathtt{BEN}) = \{\ \mathbf{meet}\ \}$$

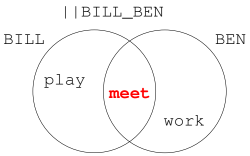


Figure : 4.2 BILL_BEN's Alphabet Diagram.

Execution of Process `BILL_BEN`

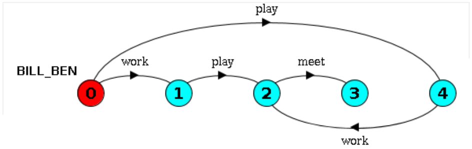


Figure : 4.3 `BILL_BEN`'s LTS Diagram.

The two longest traces for `BILL_BEN` are:

`< play, work,` **`meet`** `>,` `< work, play,` **`meet`** `>`

The *asynchronous* actions: `BILL`'s `play` & `BEN`'s `work`, are *concurrent* & can be *executed in any order*.

However, both of these actions are *constrained* to happen *before* the *synchronised* action **`meet`**.

The **`meet`** action *synchronises* the execution of the processes `BILL` & `BEN`.

NOTE: after one of these process has performed its asynchronous action, i.e. `play` or `work`, it cannot perform the synchronised action **`meet`**, until the other process is ready to perform it, i.e. *"synchronise"* on it.

Sharing Actions Example 2

`MAKER_USER` consists of a process that manufactures (`make`) an item & then *signals* the item is ready for `use` by the *synchronised* **`ready`** action.

A user can only `use` the item *after* the **`ready`** action occurs.

```
MAKER_USER

MAKER = ( make ->  ready -> MAKER ) .
USER  = (  ready -> use -> USER ) .

|| MAKER_USER = ( MAKER || USER ) .
```

The synchronised actions are then given by:

$$\texttt{sync}(\texttt{MAKER\_USER}) = \texttt{alphabet}(\texttt{MAKER}) \cap \texttt{alphabet}(\texttt{USER}) = \{\, \textbf{ready} \,\}$$

The state machine for `MAKER_USER` is:

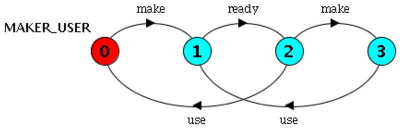


Figure : 4.4 Composition `MAKER_USER`.

MAKER_USER's Traces

From Fig. 4.4, it can be seen that the following are possible *execution traces* of MAKER_USER:

```
< make, ready, use, make, ready, ... >

< make, ready, make, use, ready, ... >
```

After the initial item is manufactured & becomes ready, manufacture & use can *proceed in parallel* since the actions `make` & `use` can occur in any order.

However, it is always the case that an item is made before it is used since the first action is `make` in all traces.

The second trace shows that two items can be made before the first is used.

Exercises:

1. Confirm these are traces using `ltsa`.
2. Investigate the other traces using `ltsa`.
3. Draw the *alphabet diagram* for MAKER_USER.

MAKER_USER Version 2

Suppose that this is undesirable behaviour & we do not wish the MAKER process to get ahead in this way.

The solution is to modify the model so that the USER process indicates that the item is used.

So we add a new action: **used**, this is *synchronised* with the MAKER, who now **cannot proceed to manufacture another item** until the first is used.

MAKER_USERv2

```
MAKERv2 = ( make -> ready -> used -> MAKERv2 ) .

USERv2  = ( ready -> use -> used -> USERv2 ) .

|| MAKER_USERv2 = ( MAKERv2 || USERv2 ) .
```

The *synchronised* actions are then given by:

$$\begin{aligned}\mathtt{sync}(\mathtt{MAKER\_USERv2}) &= \mathtt{alphabet}(\mathtt{MAKERv2}) \cap \mathtt{alphabet}(\mathtt{USERv2}) \\ &= \{\, \mathbf{ready}, \mathbf{used} \,\}\end{aligned}$$

`MAKER_USER` – "Handshake"

The *interaction* between `MAKERv2` & `USERv2` is an example of a *"handshake"*.

That is *"an action which is acknowledged by another action"*.

Handshake protocols are widely used to structure interactions between concurrent processes.

They are usually achieved by means of *two synchronised actions*, e.g. **`ready`** & **`used`**.

Note that the FSP *"model of interaction"* does **not distinguish** which process instigates a shared action.

However, we obviously do.

For example, we quite naturally think of:

- the `MAKERv2` process *instigating* the **`ready`** action &
- the `USERv2` process *instigating* the **`used`** action.

Examples of Synchronisation

The examples of *synchronisation* so far are between *two processes*, i.e. *binary* synchronisation.

In practice this is by far the most common case, but *many processes* can engage in a *synchronised action*.

The next example uses *"multi-party"* (or *"n-way"* or *"n-ary"*) synchronisation.

The example is a small factory manufacturing system which produces two different parts & assembles the parts into a product.

Will need to define three processes:

- `MAKE_A` & `MAKE_B` that each make a part.
- `ASSEMBLE` that assembles the parts into a product.

It will make use of *3-way* synchronisation.

Concurrency Requirements (CRs)

For this system to work correctly we have the following *Concurrency Requirements (CRs)*:

(CR1) Assembly **cannot** take place until both parts are ready.

(CR2) Makers are **not permitted** to get ahead of the assembler.

Now since we must *synchronise* the three processes twice (once each for CR1 & CR2) we need to introduce two *synchronised* actions.

`ready`: for (CR1), the assembler must wait until both components are ready.

`used`: for (CR2), the two makers must wait until the two components have been used by the assembler.

Designing the FSP code for the Factory

The first step is to define the *alphabets* for the individual processes, which are as follows:

```
alphabet(MAKE_A)    = { makeA, ready, used }

alphabet(MAKE_B)    = { makeB, ready, used }

alphabet(ASSEMBLE) = { ready, assemble, used }

alphabet(FACTORY)  = { makeA,  makeB, ready, assemble, used }
```

The *synchronised actions* are then given by:

$$\begin{aligned} \mathtt{sync(FACTORY)} &= (\,\mathtt{alphabet(MAKE\_A)} \cap \mathtt{alphabet(MAKE\_A)} \\ &\qquad \cap\ \mathtt{alphabet(ASSEMBLE)}\,) \\ &= \{\, \mathbf{ready}, \mathbf{used} \,\} \end{aligned}$$

So in the `FACTORY` system there are two synchronised actions **`ready`** & **`used`**, & each is a 3-way synchronised action.

FACTORY's Alphabet Diagram

The *alphabet diagram* for FACTORY is then given in Fig. 4.5.

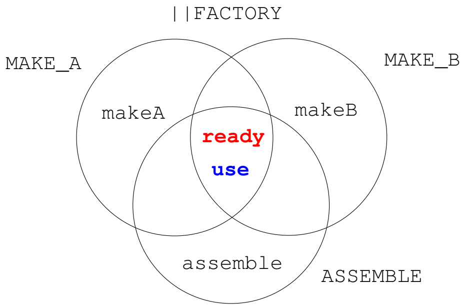


Figure : 4.5 FACTORY's Alphabet Diagram.

From Fig. 4.5, we can see that there are no *binary* synchronous actions (the 3 areas are empty), but only *3-way* synchronous actions (area contains **ready** & **use**).

The FSP code for the Factory

So now the FSP processes for this system are:

```
FACTORY

MAKE_A = ( makeA -> ready -> used -> MAKE_A ) .
MAKE_B = ( makeB -> ready -> used -> MAKE_B ) .

ASSEMBLE = ( ready -> assemble -> used -> ASSEMBLE ) .

|| FACTORY = ( MAKE_A || MAKE_B || ASSEMBLE ) .
```

The state machine for FACTORY is:

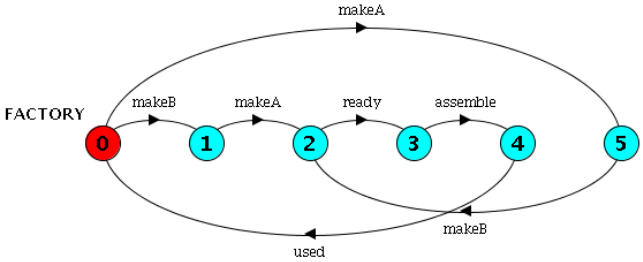


Figure : 4.6 Composition FACTORY.

Nested Parallel Composition

Since a *parallel composition of processes* is itself a *process*, called a *composite process*, it can be used in the definition of further compositions.

We can restructure the previous example by creating a composite process from `MAKE_A` & `MAKE_B` as follows:

```
MAKERS
MAKE_A = ( makeA -> ready -> used -> MAKE_A ) .

MAKE_B = ( makeB -> ready -> used -> MAKE_B ) .

|| MAKERS = ( MAKE_A || MAKE_B ) .
```

`ASSEMBLE` remains the same, but the definition of `FACTORY` is now:

```
FACTORY
|| FACTORY = ( MAKERS || ASSEMBLE ) .
```

The state machine remains that depicted in Fig. 4.6.

PART II

How to Achieve Mutually Exclusive Access to a Shared Resource

Achieving Mutually Exclusive Access to a Shared Resource

Recall, that when we have a concurrent system made up of:

- a *shared resource* &
- several processes that need to *share the resource* in a "*safe*" way,

we generally require that each process has *mutually exclusive access* to the shared resource.

To be able to model these types of systems in FSP we first need to introduce more of FSP's features:

action sets	`set Actions = { a1, ..., an }`
using parameterised processes	`Proc( N = DefaultValue )`
process labelling	`a:P`
process prefix label sets	`{ a1, ..., an } :: P`
relabelling actions	`P/{ new/old }`
hiding actions	`P\{ a1, ..., an }`
interface operator	`P@{ a1, ..., an }`
alphabet extension	`P+{ a1, ..., an }`

Defining Sets of Actions

A very useful feature of FSP is the ability to define a *set of actions* as follows:

```
set SetOfActions = { a1, a2, ..., an }
```

then `SetOfActions` can be used anywhere in an FSP program that an explicit set of actions could be used.

For example, we can define the set of actions that define the *interface* to a process as a set of actions & then use it to define the process:

PROC

```
set Interface = { acquire, release, in, out }

PROC = ( Interface -> do_process -> PROC ) .
```

Expanded PROC

```
PROC1 = (  acquire -> do_process -> PROC
         | release -> do_process -> PROC
         | in      -> do_process -> PROC
         | out     -> do_process -> PROC  ) .
```

Using Parameterised Processes in Parallel

Recall the definition of the single element buffer process `BUFF3` from the previous lecture.

It is a *parameterised* process that takes the *upper limit* for what value the buffer is willing to accept as input:

BUFF3

```
const N = 3

BUFF3( N = 3 ) = ( in[ i : 0..N ] -> out[ i ] -> BUFF3 ) .
```

We can use this definition to define a parallel process that uses two instances of the `BUFF3` process with different parameters as follows:

Buffer System BS

```
const UL1 = 4
const UL2 = 5

|| BS = ( BUFF3( UL1 ) || BUFF3( UL2 ) ) .
```

Parallel Composition of Identical Processes

Given the definition of a process, we often want to use more than one copy of that process in a program or system model.

For example, given the definition for a switch:

```
SWITCH
SWITCH = ( on -> off -> SWITCH ) .
```

We might want a system that is the composition of two switches.

However, if we model this system as:

```
NOT_TWO_SWITCHES
|| NOT_TWO_SWITCHES = ( SWITCH || SWITCH ) .
```

`NOT_TWO_SWITCHES` is indistinguishable from a single `SWITCH`, since the two `SWITCH` processes synchronise on their shared actions `on` & `off`.

Why it Doesn't Work

Remember, processes combined in parallel *synchronise on their shared actions* & obviously here they have the same alphabets:

```
aplhabet( SWITCH ) = { on, off }
```

Therefore, they *synchronise on both actions*, as a result they are always in *"lock-step"*.

Thus, if we want 2 switches then we must ensure that the actions of the `SWITCH` processes are **not shared**, i.e. they must have *disjoint alphabets*.

That is we want two `SWITCH` processes & their two actions must be *asynchronous*, i.e. they must **not** interact at all.

So we must created *two copies* of the `SWITCH` process that have *different alphabets*, i.e. their actions must have *disjoint action labels*.

To do this we need to introduce an FSP feature called: *Process Labelling*.

Process Labelling

To achieve this requirement of processes with *disjoint alphabets*, we use the process labelling construct.

Definition: *Process Label*

`a : P` prefixes each action label in the alphabet of `P` with the label “`a`”.

For example, compare the alphabets of a process `P` & `a:P`:

```
alphabet( P )      = { x, y, z }
alphabet( a : P ) = { a.x, a.y, a.z }
```

A system with two switches can now be defined as:

```
|| TWO_SWITCHES_1  = ( a:SWITCH || b:SWITCH ) .
|| TWO_SWITCHES_2  = ( { a, b } : SWITCH ) .
```

Now it is clear that the alphabets of the two processes are disjoint:

```
alphabet( a : SWITCH ) = { a.on, a.off }
alphabet( b : SWITCH ) = { b.on, b.off }
```

Multiple Switches

Using a *parameterised composite process*, we can describe an *“array”* of switches processes in FSP as follows:

```
|| SWITCHES_1( N = 3 )
     = ( forall[ i : 1..N ]  s[i]:SWITCH ) .
```

An equivalent but shorter definition is:

```
|| SWITCHES_2( N = 3 )  =  ( s[ i : 1..N ] : SWITCH ) .
```

This is also equivalent to:

```
|| SWITCHES_3
     = ( s[1]:SWITCH || s[2]:SWITCH || s[3]:SWITCH ) .
```

We can use `SWITCHES_1` to get any number (4) of switches as follows:

```
|| SWITCHES_4 = SWITCHES_1(4) .
```

See the state machines for these processes using `ltsa`.

Process Prefix Label Sets

Processes may also be labelled by a set of *prefix labels*.

Definition: *Process Prefix Label Sets*

The following process:

```
{ a1, ..., an } ::  P
```

replaces every *action label* `x` in the *alphabet* of `P` with the labels:

```
a1.x, ..., an.x
```

Further, every *transition* `(y -> Q)` in the definition of `P` is replaced with the *transitions*:

```
( { a1.y, ..., an.y } -> Q )
```

which is equivalent to:

```
( a1.y -> Q | a2.y -> Q |...| an.y -> Q )
```

Example: Mutual Exclusion using Process Prefix Label Sets

The control of a *resource* is modelled by the `RESOURCE` process & *users* of the resource are modelled by the `USER` process:

```
RESOURCE & USER

RESOURCE = ( acquire -> release -> RESOURCE ) .

USER = ( acquire -> use -> release -> USER ) .
```

We wish to model a system consisting of two users that *share the resource* such that only one user at a time can be using it.

That is we require that the resource enforces *"mutual exclusion"*.

The two users can be modelled using *process labelling* as: **`a:`**`USER` & **`b:`**`USER`.

These are *equivalent* to the following ones, defined without labelling:

```
A_USER = ( a.acquire -> a.use -> a.release -> A_USER ) .

B_USER = ( b.acquire -> b.use -> b.release -> B_USER ) .
```

The Labelled Processes

Now given the two user processes – `a:USER` & `b:USER`.

The resource has to ensure *mutual exclusive access* between these 2 user processes.

To achieve this the `RESOURCE` must be *prefix labelled* with the label set `{a, b}` to produce the process: `{ a, b }::RESOURCE`.

This is *equivalent* to `AB_RESOURCE`, defined without labelling:

```
AB_RESOURCE = ( { a.acquire, b.acquire } ->
                { a.release, b.release } -> AB_RESOURCE ) .
```

This means that there are:

- 2 distinct actions (`a.acquire` & `b.acquire`) to *obtain the resource*, &
- 2 actions to *free it* (`a.release` & `b.release`).

This results in *"matching"* actions that can be *synchronised* on between the resource & the two users, that result in *synchronised transitions*.

Modelling the Mutually Exclusive Shared Resource

The composition is described below.

```
                              ME_SHARED_RESOURCE

 || ME_SHARED_RESOURCE = (    a : USER
                          || b : USER
                          || { a, b } :: RESOURCE ) .
```

The effect of *process labelling* on RESOURCE can clearly be seen in Fig. 4.7.

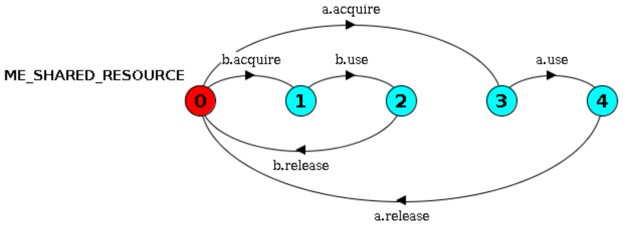


Figure : 4.7 Process labelling in ME_SHARED_RESOURCE..

ME_SHARED_RESOURCE's graph shows that the desired result of allowing only one user to use the resource at a time has been achieved.

A possible problem ensuring Mutual Exclusion?

You might notice that our model of the `RESOURCE` alone would permit *one user to acquire* the resource & the *other to release* it!

For example, it would permit the following **disastrous** trace:

```
< a.acquire, b.release, ... >
```

However, each of the `USER` processes **cannot release the resource** until it has succeeded in *performing an acquire action*.

Hence, when the `RESOURCE` is composed with the `USER` processes, this composition ensures that only the same user that acquired the resource can release it.

This is shown in the composite process `ME_SHARED_RESOURCE` in Figure 4.7.

This can also be confirmed using `ltsa` to run through the possible traces.

NOTE: See the tutorial exercise that compares different versions of the `USER` & `RESOURCE` processes.

Process (Action) Relabelling

Definition: *Process (Action) Relabelling*

Relabelling functions are applied to processes `P` to change the *names* of *action labels*.
The general form of the relabelling function is:

```
    P / { new_1/old_1,  ...,  new_n/old_n }.
```

A *relabelling function* can be applied to both *primitive* & *composite* processes.

However, it is generally used far more often when defining composite processes, because *relabelling* is usually done to ensure that: *composite processes synchronise on certain actions*.

Approach:

1. The individual actions of the processes that have *different names*, but are required to be *synchronised* on, are simply *relabelled* so that they *all have the same name*.
2. When the processes are composed in parallel, the *relabelled actions* will now automatically be *synchronised on*.

Example of Process Relabelling: `CLIENT` & `SERVER`

A *server* process that provides some service & a *client* process that uses the service are defined by:

```
CLIENT & SERVER

CLIENT = ( call -> wait -> continue -> CLIENT ) .

SERVER = ( request -> service -> reply -> SERVER ) .
```

But `CLIENT` & `SERVER` have *disjoint alphabets* & do not interact in any way.

```
alphabet(CLIENT) = { call, wait, continue }
alphabet(SERVER) = { request, service, reply }
```

But by using the following *relabelling*, we can make them *interact*, i.e. *synchronise*:

- `CLIENT / { reply/wait }` – `CLIENT`'s `wait` action relabelled to `reply`, so will synchronise with `SERVER`'s `reply` action.
- `SERVER / { call/request }` – `SERVER`'s `request` action relabelled to `call`, so will synchronise with `CLIENT`'s `call` action.

Example of Process Relabelling: CLIENT_SERVER System

The system is then:

```
                         CLIENT_SERVER
|| CLIENT_SERVER = (    ( CLIENT / { reply/wait } )
                     || ( SERVER / { call/request } ) ) .
```

The effect of applying the *relabelling function* can be seen in the state machine representation of Fig. 4.8.

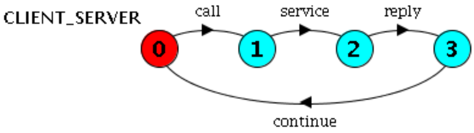


Figure : 4.8 Relabelling in CLIENT_SERVER.

- reply replaces wait in the description of CLIENT
- call replaces request in the description of SERVER

Action Prefix Relabelling: Alternative Client/Server System

An alternative client/server system using *qualified* or *prefixed labels*:

```
                         CLIENT_SERVERv2

CLIENTv2 = ( call.request -> call.reply
                          -> continue -> CLIENTv2 ) .

SERVERv2 = ( accept.request -> service
                            -> accept.reply -> SERVERv2 ) .

||CLIENT_SERVERv2 = ( CLIENTv2 || SERVERv2 )
                                  / { call/accept } .
```

The relabelling function {**`call`**/**`accept`**} replaces *any label prefixed* by **`accept`** with the same label *prefixed* by **`call`**.

Thus **`accept`**`.request` becomes **`call`**`.request` & **`accept`**`.reply` becomes **`call`**`.reply` in the composite process `CLIENT_SERVERv2`.

Relabelling by *prefix* is useful when a process has *more than one "interface"*.

Each *interface* consists of a *set of actions* & can be related by having a *common prefix*.

In *compositions* an *interface's prefix* is often *relabelled*.

Hiding Actions

Definition: *Process Hiding*

When applied to a process `P`, the *hiding* operator “\”:

```
P \ { a1, .., an }
```

removes the action names `a1, .., an` from the alphabet of `P` & makes these *concealed* actions *“silent”*.
These silent actions are labelled “**`tau`**”. (Greek letter τ.).
Silent actions in different processes are **not shared**.

Hidden actions become *“unobservable”* in that they **cannot be shared** with another process & so **cannot** affect the execution of another process.

Hiding is essential in reducing the complexity of large systems for analysis purposes.

It is possible to *minimise* the size of state machines by removing **`tau`** actions, see the `ltsa` “**Minimise**” operation.

Hiding can be applied to both *primitive* & *composite* processes.

Example: Hiding an Action

As an example consider the `USER` process:

```
USER = ( acquire -> use -> release -> USER ) .

alphabet( USER ) = { acquire, use, release }
```

Now we can *hide* the **use** action as follows:

```
set HideAction = { use }

USER = ( acquire -> use -> release -> USER ) \ HideAction .

alphabet( USER \ HideAction ) = { acquire, release }
```

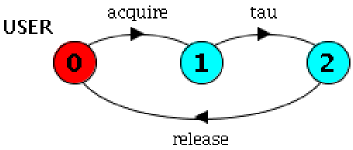


Figure : 4.9 Hiding action **use** of `USER`.

Interface Operator

Sometimes it is more convenient to state the set of action labels which are *visible* & hide all other actions, i.e. a processes *interface*..

Definition: *Interface Operator*

When applied to a process `P`, the *interface* operator "`@`":

```
P @ { a1, .., an }
```

hides all of `P`'s actions **not** in the set: `{ a1, .., an }`.

For example, this version of `USER` is equivalent to the previous version:

```
USER
set Interface   = { acquire, release }

USER = ( acquire -> use -> release -> USER ) @ Interface .
```

Using `ltsa`'s "*minimisation*" feature on `USER` removes the hidden **`tau`** action, resulting in a state machine with equivalent "*observable behaviour*", but fewer states & transitions. (Confirm this using `ltsa`.)

Alphabets & Traces of Process using Hiding & Interface Operators

When a process is defined using either the hiding (\ ) or interface (`@`) operators, the actions that are *hidden* by either of these operators **no longer appear** in either the *alphabet* or *traces* of the process.

However, these hidden actions appear as *"`tau`" actions* in `ltsa`'s state machines, they do not appear in either the process's alphabet or traces, e.g.

```
P7 & P8

set HideActions = { b, c }
set Interface   = { b, c }

P7  = ( a -> b -> c -> d -> STOP ) \  HideActions .

P8  = ( a -> ( b -> STOP | c -> d -> STOP ) ) @ Interface .
```

then the *alphabets* & *traces* are:

```
alphabet(P7) = { a, d }
alphabet(P8) = { b, c }

traces(P7) = { <>, <a>, <a, d> }
traces(P8) = { <>, <b>, <c> }
```

Process Alphabet Extensions

Recall that the alphabet of a process is the set of actions in which it can engage.

However, it may have actions in its alphabet that it never performs.

For example, a program that writes to an integer variable may potentially write any (32/64-bit) integer value to the variable.

E.g. 32 bits: -2,147,483,648 — 2,147,483,647,
64 bit: -9,223,372,036,854,775,808 — 9,223,372,036,854,775,807.

But, in practice it will usually only write a few different integers out of this large set of values.

Question: How do we deal with the situation where the set of actions *required to be in the alphabet* is larger than the set of actions *used or referenced* in its definition?

Answer: Use the *alphabet extension* operator "**+**" provided by FSP.

Process Alphabet Extension Definition

Definition: *Process Alphabet Extension*

The *process alphabet extension* operator "**+**":

```
P + { a1, a2, ..., an }
```

extends process `P`'s alphabet, by adding the set of actions "`{ a1, a2, ..., an }`" to it:

```
  alphabet( P + { a1, a2, ..., an } )
= alphabet( P ) ∪ { a1, a2, ..., an }
```

This does **not** mean that process `P` will perform any of the added actions: `a1, a2, ..., an`, only that they are *now part of its alphabet*.

Notation: if a process is defined using one or more local process definitions, the *alphabet of each local process is exactly the same as that of the enclosing process*.

The *alphabet of the enclosing process* is simply the *union of the set of actions referenced in all local definitions* together with *any explicitly specified alphabet extension*.

Example of a Process Alphabet Extension

Consider the following process `WRITER_13` defined as follows:

WRITER_13

```
WRITER_13 = ( write[1] -> write[3] -> WRITER_13 ) .
```

Since actions `write[1]` & `write[3]` are in its definition its alphabet is:

```
alphabet(WRITER_13) = { write[1], write[3] }
```

However, we can extend this process's alphabet by using an *alphabet extension* of the actions `write[0..3]` as follows:

WRITER_0123

```
WRITER_0123 = ( write[1] -> write[3] -> WRITER_0123 )
                                       +{ write[ 0..3 ] } .
```

Using the definition of +, the alphabet of `WRITER_0123` is:

```
alphabet(WRITER_0123) = { write[0], write[1],
                          write[2], write[3] }
```

QUESTION: When is Alphabet Extension Needed?

Consider the following FSP processes & their *alphabets*:

```
AC & BC

set BLOCK = { c, d }

A = ( a -> b -> A ) .
B = ( a -> b -> B ) + BLOCK .
C = ( {a, b, c, d } -> C ) .

||AC = ( A || C ) .
||BC = ( B || C ) .

alphabet(A)   = { a, b }

alphabet(B)   = { a, b, c, d }

alphabet(C)   = { a, b, c, d }

alphabet(||AC) = { a, b, c, d }

alphabet(||BC) = { a, b, c, d }
```

Note that **c** & **d** are in `B`'s alphabet by *alphabet extension*.

Alphabet Diagrams for `||AC` & `||BC`

`||AC` & `||BC` alphabet diagrams shows the effect of **+BLOCK** on their actions: **synchronised**, **asynchronous**, **blocked**.

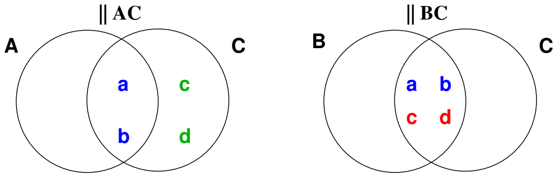


Figure : 4.10 Alphabet Diagrams for `||AC` & `||BC`

For ||`AC`:

$$\begin{aligned}
\texttt{sync(||AC)} &= \texttt{alphabet(A)} \cap \texttt{alphabet(C)} &= \{\, \texttt{a},\ \texttt{b} \,\} \\
\texttt{async(A)} &= \texttt{alphabet(A)} - \texttt{alphabet(C)} &= \{\, \} \\
\texttt{async(C)} &= \texttt{alphabet(C)} - \texttt{alphabet(A)} &= \{\, \texttt{c},\ \texttt{d} \,\}
\end{aligned}$$

For ||`BC`:

$$\begin{aligned}
\texttt{sync(||BC)} &= \texttt{alphabet(B)} \cap \texttt{alphabet(C)} &= \{\, \texttt{a},\ \texttt{b},\ \texttt{c},\ \texttt{d} \,\} \\
\texttt{async(B)} &= \texttt{alphabet(B)} - \texttt{alphabet(C)} &= \{\, \} \\
\texttt{async(C)} &= \texttt{alphabet(C)} - \texttt{alphabet(B)} &= \{\, \}
\end{aligned}$$

ANSWER: When We Need to Force Synchronisation

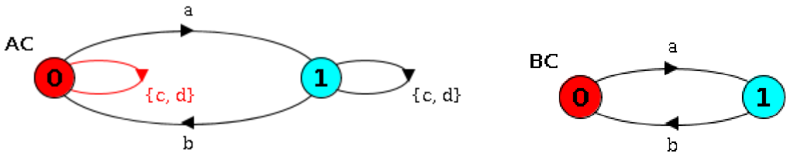


Figure : 4.11 `||AC` & `||BC` State Machines

So in `||AC`, `C` can perform **`c`** & **`d`** *asynchronously*, but in `||BC`, `C` can **not** perform **`c`** or **`d`** because they are **blocked**, as they must be *synchronised* with `B`, which is impossible since `B` can not perform them.

Here, the use of the *alphabet extension* "**`+BLOCK`**" to extend `B`'s alphabet, **blocks** or **inhibits** `C` from performing the two actions: **`c`** & **`d`**.

This is the most common use of an *alphabet extension*, i.e. to extend a *client* process's alphabet to **block** a *resource* process from performing actions *independently* of the *client* that is currently using it.

PART III

Inter-Process Communication using "Synchronised Message Passing"

Inter-Process Communication: Passing Data between Processes

Previously we have seen how FSP processes can *synchronise* with each other via *shared actions*, e.g. `acquire`, `release`.

However, one of the most important abilities that the processes in a concurrent system must have, is to *communicate with each other*.

This means that processes can *pass data* between themselves & other processes & not just synchronise their actions.

The mechanism that provides this ability is known as an *Inter-Process Communication (IPC)* mechanism.

The IPC mechanism used in FSP is known as *“synchronised message passing”*.

In standard *synchronised message passing*:

- a *synchronised message* is passed from 1 *sender* process to 1 *receiver* process,
- the contents of the *“message”* is the *data* that is being communicated between the two processes.

Synchronised Message Passing in FSP

To see how *synchronised message passing* works in FSP consider the following basic case in Fig. 4.12.

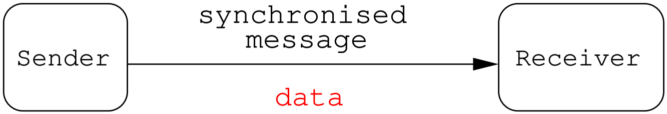


Figure : 4.12 Inter-Process Communication (IPC) between two processes

So, if two processes `Sender` & `Receiver` want to pass data between them then:

- `Sender` & `Receiver` must perform a *synchronised* action.
- The *data transmission* between them is achieved because the data is part of the synchronised action, i.e. message.
- One process, the `Sender` will *send the data*.
- The other process, `Receiver` will *receive the data*,
- An important point to note about this is that they do not *share any common memory*.

Synchronised Message Passing Example: “Distributed Assignment”

Synchronised message passing can be used to achieve *“distributed assignment”* – one process *“assigns”* a value to a variable in another process.

Distributed Assignment

```
const MAX    = 3
range DATA   = 1..MAX

set ALL_DATA = { message[DATA] }

Sender(VAL = 1) = ( message[ VAL ] -> assignedToX[ VAL ]
                                   -> END ) +ALL_DATA .

Receiver = ( message[ x : DATA ] -> valueOfX[ x ] -> END ) .


|| DistributedAssignment  =  ( Sender(1) || Receiver ) .
```

The overall effect is that the `Sender` & `Receiver` synchronise on the **`message[1]`** action, & the result is that 1 is assigned to `Receiver`’s `x`.

Alphabet Diagram for `||DistributedAssignment`

The alphabet diagram for `||DistrubutedAssignment` is given in Fig. 4.13, where the **blue** actions are the only ones that can be performed.

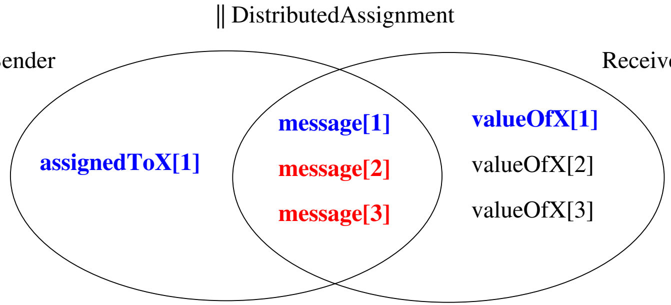


Figure : 4.13 Alphabet Diagram for `||DistrubutedAssignment`

Notes on Distributed Assignment Example

- `Sender` can do 1 “output” action: `message[1]`.
- But “**`+ALL_DATA`**” in `Sender` adds `message[2]` & `message[3]` to its alphabet.
- `Receiver` offers a *choice* of 3 “input” actions: `message[1]`, `message[2]`, `message[3]`.
- This means `Receiver` & `Sender` *must synchronise* on all possible “`message[DATA]`” actions, i.e. `message[1]`, `message[2]`, `message[3]`.
- So `Sender` is blocking `Receiver` doing any `message[DATA]` actions that `Sender` is *not able or willing to do*, i.e. **`message[2]`** & **`message[3]`**
- `Sender` & `Receiver` perform **`message[1]`** *synchronously*, because that is the **only action** that the `Sender` is able to do.
- So the result is that 1 is assigned to `Receiver`’s `x`.

PART IV

FSP Structure Diagrams

Structure Diagrams

Used *state machine* diagrams to depict the *dynamic behaviour* of processes.

State machine diagrams represent the *dynamic process operators*, action prefix & choice.

However, these diagrams **do not capture** the *static structure* of a model.

A *parallel composition* can be described as a state machine since it is a process, but the parallel composition expression itself is not represented.

Parallel composition, relabelling & hiding are *static operators* that describe the *"structure" of a model* in terms of *primitive processes* & their *interactions*.

So to see this *static structure* of an individual process or a parallel system we use a *Structure Diagram*, see Fig. 4.14.

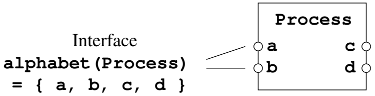
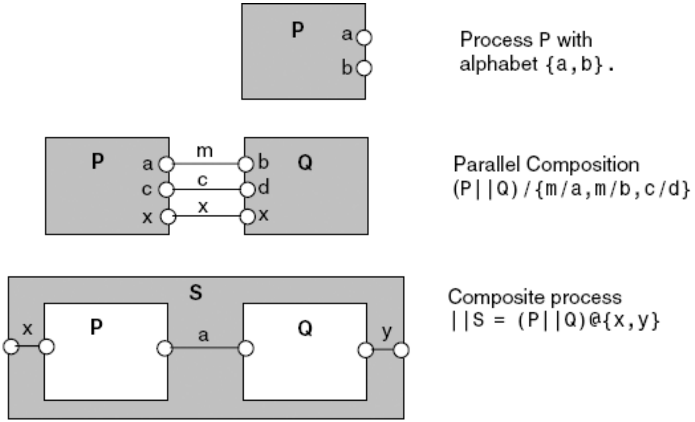


Figure : 4.14 Structure Diagram for a Process.

Structure Diagram Composition Expressions

Composition expressions can be represented graphically using *Structure Diagrams* as shown in Fig. 4.15.



Process P with alphabet `{a,b}`.

Parallel Composition `(P||Q)/{m/a,m/b,c/d}`

Composite process `||S = (P||Q)@{x,y}`

Figure : 4.15 Structure Diagram conventions.

Structure Diagram Conventions I

- A *process* is represented as a box with its *interface* (its *alphabet of visible actions*) shown as circles on the perimeter.
- A *shared synchronous action* is depicted as a line connecting two action circles, with relabelling if necessary.
- A line joining two actions with the same name indicates *only a shared synchronous action* since relabelling is not required.
- A *composite process* is the box enclosing a set of process boxes.
- The *alphabet of the composite* is again indicated by action circles on the perimeter.
- Lines joining these circles to internal action circles show how the composite's actions are defined by primitive processes.
  These lines may also indicate relabelling functions if the *composite* name for an action differs from the *internal* name.

Structure Diagram Conventions II

- Those actions that appear internally, but are not joined to a composite action circle, are *hidden actions*.
  This is the case for action `a` inside `S` in Fig. 4.15.

- The processes inside a composite may, of course, themselves be composite & have structure diagram descriptions.

- We sometimes use a line in a structure diagram to represent a *set of shared actions* that have a *common prefix label*.
  The line is *labelled with the prefix* rather than *explicitly by the actions*.

- Sometimes we omit the label on a connection line where it does not matter how relabelling is done since the label does not appear in the alphabet (interface) of the composite.
  For example, in Fig. 4.16, it would not matter if we omitted the label `a.out` & used `b.in/a.out` instead of `a.out/b.in` as shown, as this is **not** in the interface of the composite process.

Structure Diagram Example 1

The following example uses a single-slot buffer to construct a two slot buffer, its *structure diagram* is given in Fig. 4.16.

```
                              TWOBUFF
range T = 0..3

BUFF = ( in[ i : T ] -> out[ i ] -> BUFF ) .

||TWOBUFF = ( a:BUFF || b:BUFF )
                / { in/a.in, a.out/b.in, out/b.out }
                  @ { in, out } .
```

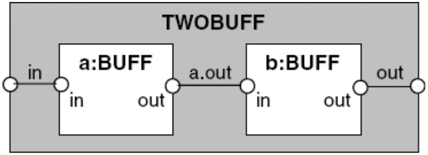


Figure : 4.16 Two-slot buffer `TWOBUFF`.

Each of the labels in the diagram of Fig. 4.16 `in`, `out` & `a.out` represents the set of labels `in[i:T]`, `out[i:T]` & `a.out[i:T]` respectively.

Structure Diagram Example 2

Lastly, we use a diagrammatic convention, see Fig. 4.17, to depict the common situation of *resource sharing*.

The resource is not anonymous as before; it is named `printer`.

```
PRINTER_SHARE

USER = ( printer.acquire -> use -> printer.release -> USER ) .

RESOURCE = ( acquire -> release -> RESOURCE ) .

|| PRINTER_SHARE = (    a : USER ||   b : USER
                     || { a, b } :: printer : RESOURCE ) .
```

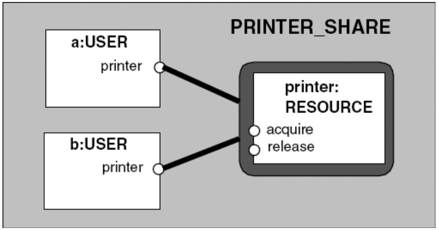


Figure : 4.17 Resource-sharing `PRINTER_SHARE`.

Structure Diagram Conventions III

- *Sharing* is indicated by enclosing the resource process in a *rounded rectangle* & the processes that share it are connected to it by *thick lines*.
- These lines could be labelled, e.g. `a.printer` & `b.printer` in `PRINTER_SHARE`.
- However, usually these labels are omitted as a relabelling function is not required.
- We also omit labels where all the labels are the same, i.e. no relabelling function is required.