SWEN90004
Modelling Complex Software Systems

Concurrency in FSP

Nic Geard

Lecture Con.06

Semester 1, 2019

Introduction

In the last lecture we used FSP to model individual processes.
However, FSP is not usually used for modelling individual processes.
Its power comes about from a simple view of interacting processes.

In this lecture, we look at how to model processes running in parallel, and the semantics for interpreting this in FSP.

FSP—Parallel composition

The parallel composition operator allows us to describe the concurrent execution of two processes.

> *"If* `P` *and* `Q` *are processes, then* `(P || Q)` *represents the concurrent execution of* `P` *and* `Q`*."*

The following is a (silly) example that specifies that one can scratch and think "at the same time":

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

FSP—Parallel composition

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

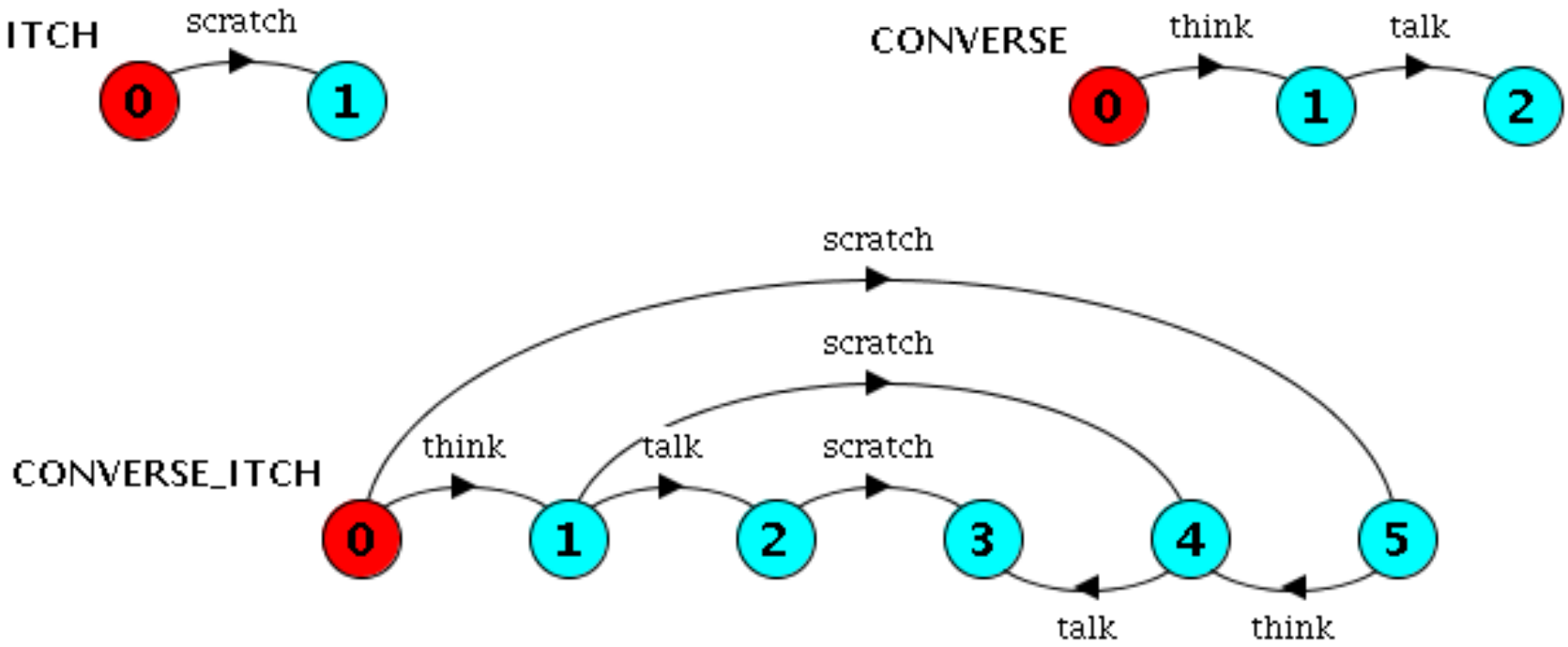
The FSP semantics specify that the two processes will interleave. That is, only a single atomic action from either will execute at one time. As a result, the possible interleavings are:

```
think -> talk -> scratch
think -> scratch -> talk
scratch -> think -> talk
```

FSP insists that when a process P is defined by parallel composition, the name of the composite process is prefixed with vertical bars: ||P.

FSP—Parallel composition

The LTSs for the two processes and the composite process are:



The composite LTS is generated in LTSA with *Build* $\rightarrow$ *Compose*. Looking at the composite LTS and the possible interleavings on the previous slide, we can see that the parallel composition of processes results in a process itself.

FSP—Parallel composition rules

There are two algebraic laws for the parallel composition operator. For all processes `P`, `Q`, and `R`, the following laws hold:

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

So the brackets do not matter, nor does the order of composition.

Composite processes are first-class citizens and can be interleaved with other processes.

That is, it is not just simple processes that can be interleaved with other simple processes.

FSP—Parallel composition rules

Here we compose `ITCH || CONVERSE` with a new process `LAUGH`:

```
ITCH = (scratch->STOP).
CONVERSE = (think->talk->STOP).
LAUGH = (laugh -> LAUGH).
||CONVERSE_ITCH = (ITCH || CONVERSE).
||CONVERSE_ITCH_LAUGH = (CONVERSE_ITCH || LAUGH).
```

Running composite processes in LTSA

When you say "compile" in LTSA, sequential processes are compiled.

Composite processes are only created once you say "compose".

At that point, LTSA will also tell you how large the state space is for the composite process.

Back to the traffic light example

Returning to the traffic light example, let us compose the traffic light with a pedestrian who does not really interact with the traffic light. The pedestrian just wanders around:

```
TRAFFIC_LIGHT = (button -> YELLOW | idle -> GREEN),
GREEN = (green -> TRAFFIC_LIGHT),
YELLOW = (yellow -> RED),
RED = (red -> TRAFFIC_LIGHT).

PEDESTRIAN = (wander -> PEDESTRIAN).

||LIGHT_PEDESTRIAN = (TRAFFIC_LIGHT || PEDESTRIAN).
```

In the composition, the wander action can occur at any time (except at the same time as another action), while the traffic lights must still obey their strict sequence.

FSP—Shared actions

How can the pedestrian interact with our traffic light?

Interaction happens through shared actions:

> *"If the processes in a composition have actions in common, these actions are said to be shared. This is how process interaction is modelled. While unshared actions may be arbitrarily interleaved, a shared action must be executed at the same time by all processes that participate in that shared action."*

FSP—Shared actions

In the next traffic light example, the shared action is `button`. This models the action of the pedestrian pushing the traffic light button, so it is part of both processes:

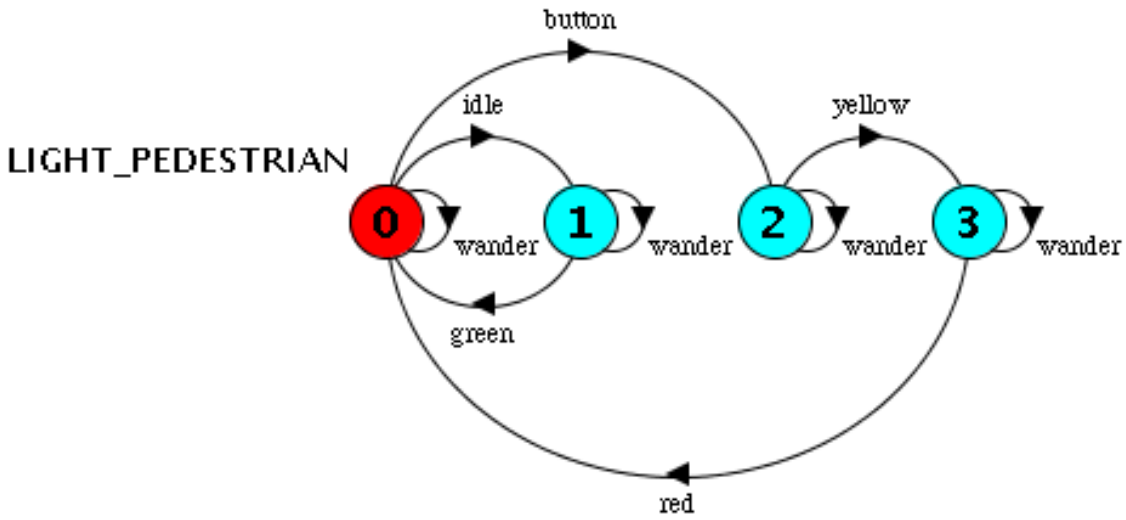
```
TRAFFIC_LIGHT = (button -> YELLOW | idle -> GREEN),
GREEN = (green -> TRAFFIC_LIGHT),
YELLOW = (yellow -> RED),
RED = (red -> TRAFFIC_LIGHT).

PEDESTRIAN = ( button -> PEDESTRIAN
             | wander -> PEDESTRIAN
             ).

||LIGHT_PEDESTRIAN = (TRAFFIC_LIGHT || PEDESTRIAN).
```

FSP—Shared actions

Here is the LTS for the example:



This composite process has no identification of which process is performing the `button` action, because it is shared: both processes participate in it.

It is natural to think of `button` as an output from `PEDESTRIAN` and input to `TRAFFIC_LIGHT`, but to FSP it is just a shared action.

FSP—Action relabelling

In the traffic light example, the pedestrian can engage in `button` or `wander`. However, the action of wandering represents the action of the traffic light button not being pushed. As such, the `wander` and `idle` actions can be considered as the same action.

We can change the model of the `TRAFFIC_LIGHT` or `PEDESTRIAN` so that the actions are named the same. However, this is not ideal if we use them elsewhere—that name change could break another composite process. Instead, we can use the relabelling operator when composing the two.

> *"Given a process* `P`*, the process*
>
> `P/{new1/old1, ..., newN/oldN}`
>
> *is the same as* `P` *but with action* `old1` *renamed to* `new1`*, etc."*

FSP—Action relabelling

We can re-label `wander` in `PEDESTRIAN` to `idle`:

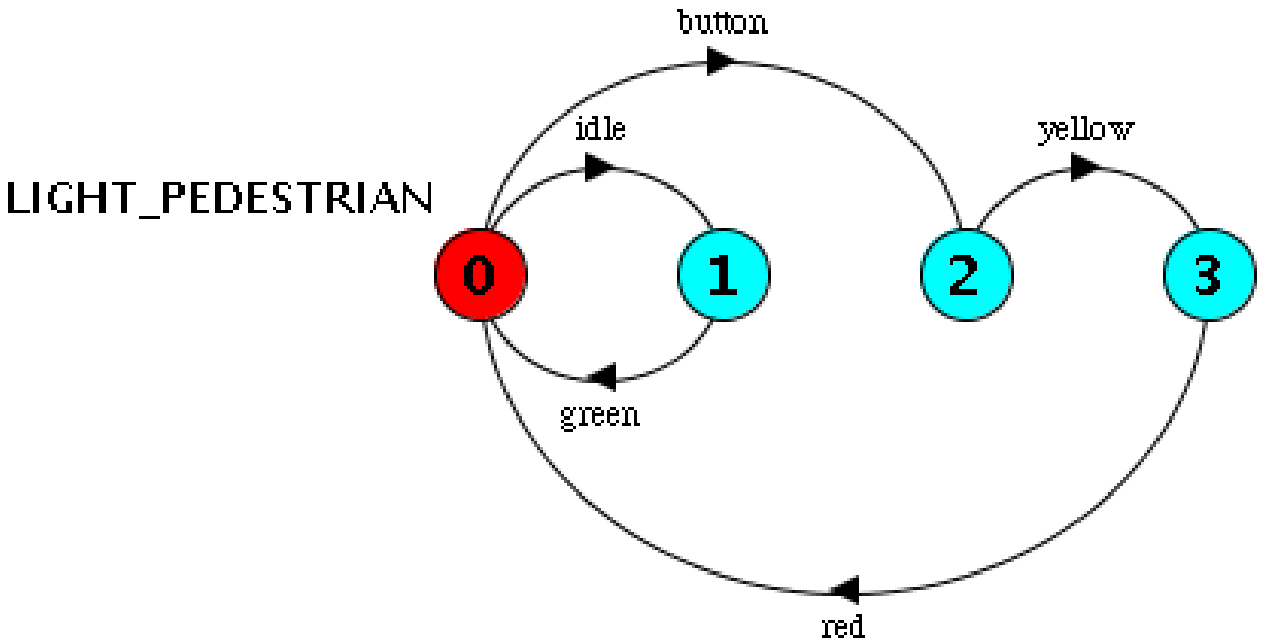
```
TRAFFIC_LIGHT = (button -> YELLOW | idle -> GREEN),
GREEN =  (green -> TRAFFIC_LIGHT),
YELLOW = (yellow -> RED),
RED = (red -> TRAFFIC_LIGHT).

PEDESTRIAN = ( button -> PEDESTRIAN
             | wander -> PEDESTRIAN
             ).

||LIGHT_PEDESTRIAN =
    (TRAFFIC_LIGHT || PEDESTRIAN/{idle/wander}).
```

FSP—Action relabelling

The resulting LTS of `LIGHT_PEDESTRIAN` is:

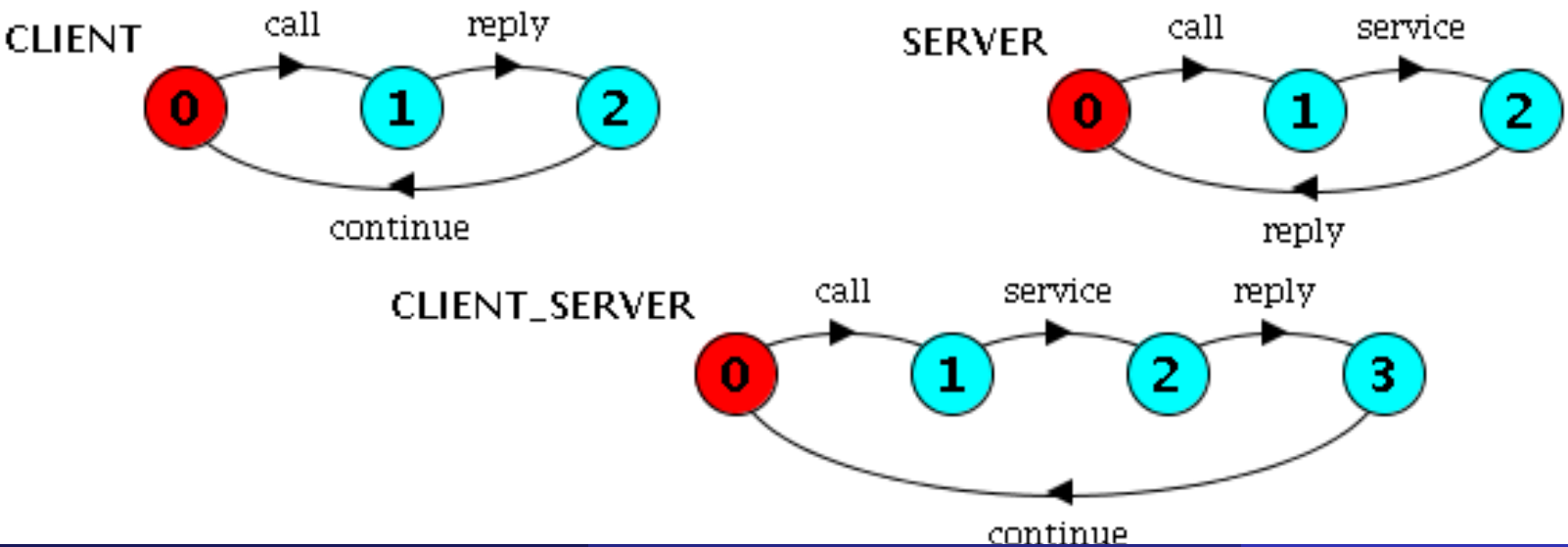


This is (not coincidentally) exactly the same as last week's `TRAFFIC_LIGHT`.

Client-server example

```
CLIENT = (call -> wait -> continue -> CLIENT).
SERVER = (request -> service -> reply -> SERVER).
||CLIENT_SERVER
  = (CLIENT || SERVER) /{call/request, reply/wait}.
```

The client makes a call, waits for the reply, and continues. The server waits for a request, services it, and replies. The client's call and the server's request are the same action, and similarly the reply and wait.

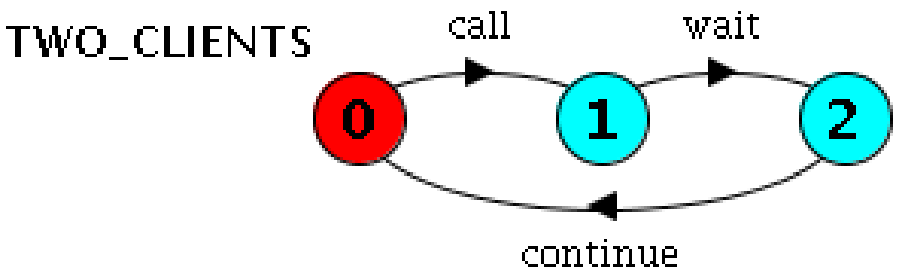


FSP—Process labelling

Given the definition of CLIENT, we may want to describe a process that involves more than one client executing in parallel, instead of just one. It may seem natural to describe this as:

```
||TWO_CLIENTS = (CLIENT || CLIENT).
```

However, all of the actions in both client processes are shared, so the result is equivalent to CLIENT itself.



FSP—Process labelling

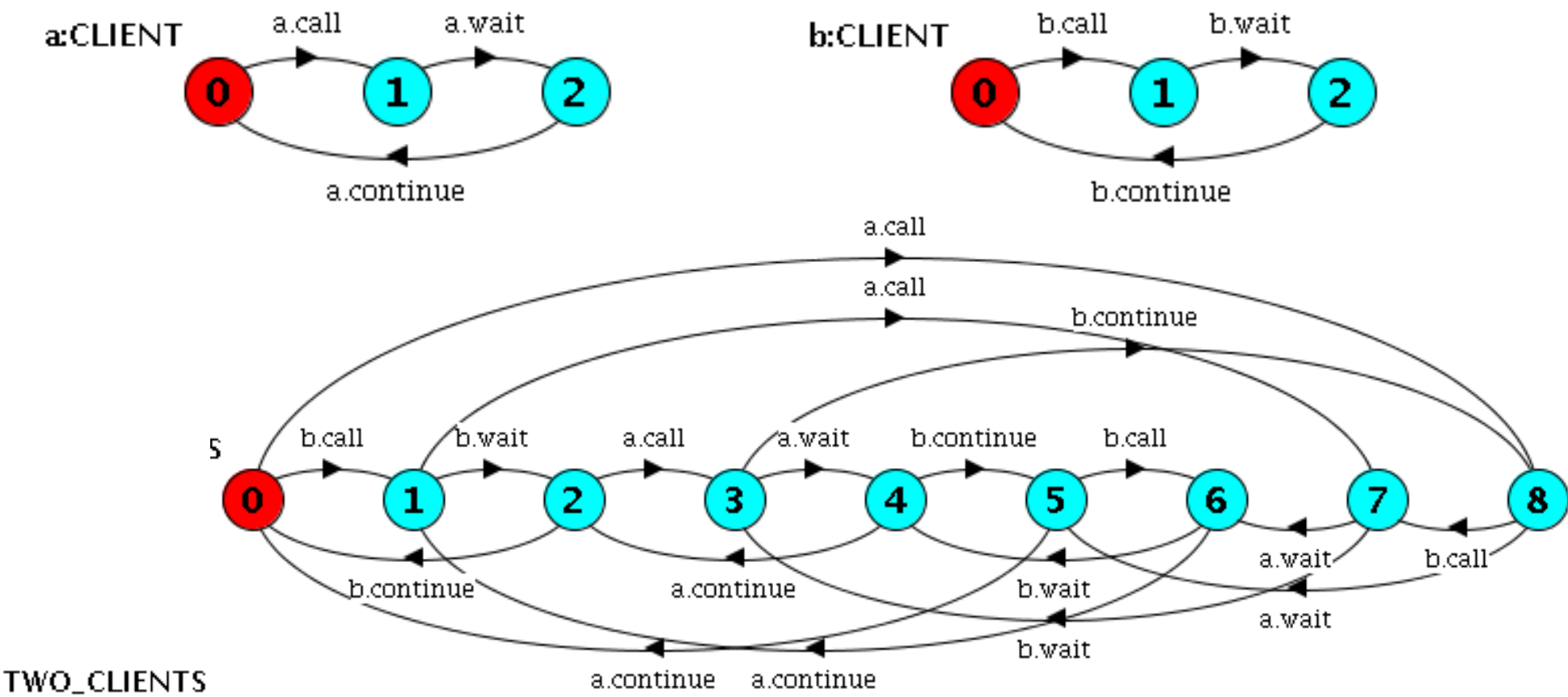
We must instead have different action labels in both clients. Writing a number of different definitions for multiple similar processes would be cumbersome and error prone, so instead, we use process labels.

> "`a:P` *prefixes each action label in* `P` *with* `a`"

To describe two clients executing concurrently, use:

```
||TWO_CLIENTS = (a:CLIENT || b:CLIENT).
```

FSP—Process labelling



The actions in both processes have been prefixed, so are not shared between the two processes.

FSP—Parameterised processes and labelling

An array of prefix labelled processes can be described using parameterised processes:

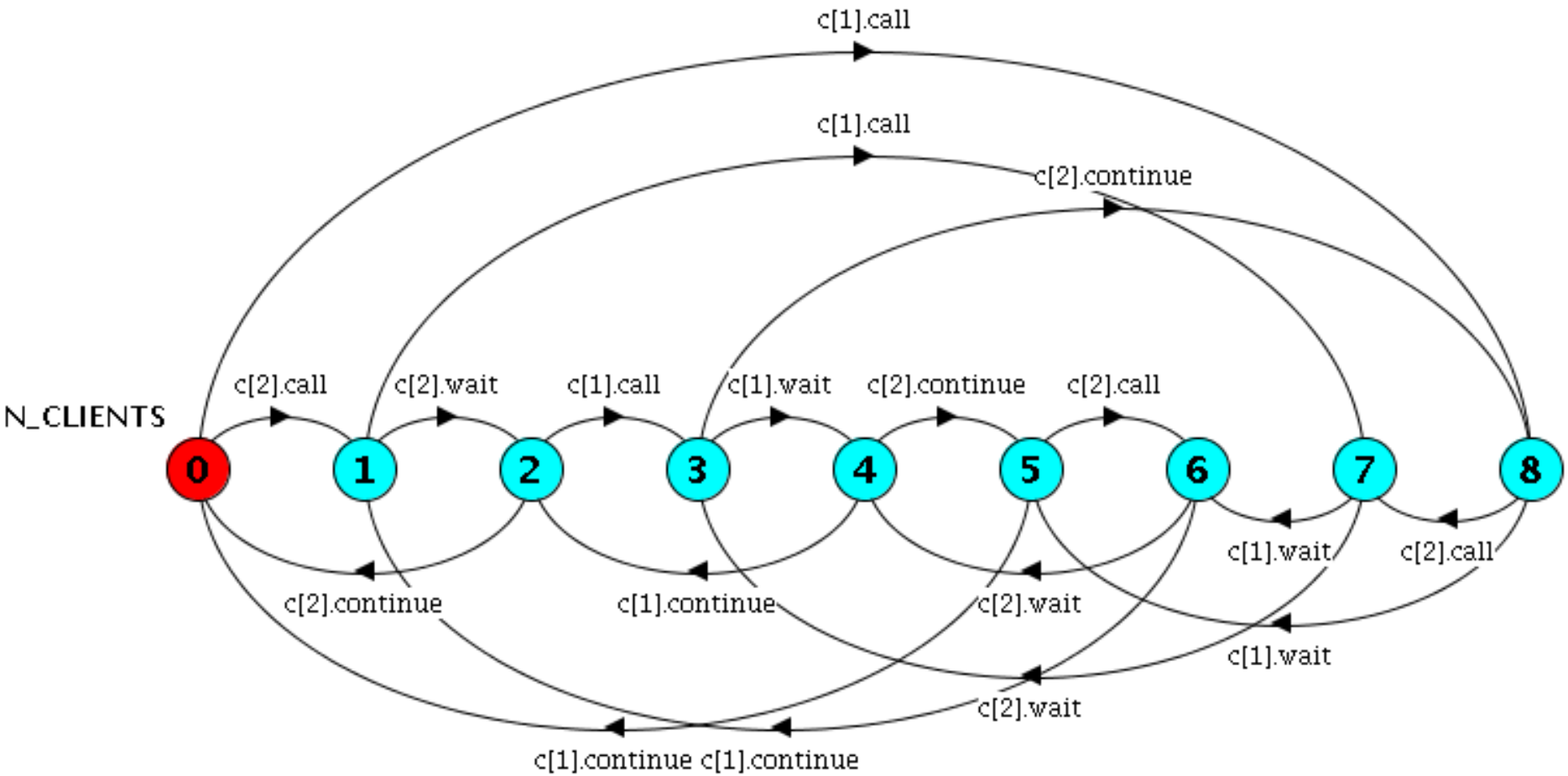
```
||N_CLIENTS(N=3) = (c[i:1..N]:CLIENT).
```

or the equivalent definition:

```
||N_CLIENTS(M=3) = (forall[i:1..M] c[i]:CLIENT).
```

FSP—Parameterised processes and labelling

The LTS for `N_CLIENTS` where `N`=2 is:



FSP—Process labelling

Now, we want to compose our server with multiple clients. However, with the action label in the clients being prefixed, they are no longer the same as the server labels, so will not be shared. To get around this, we can use a set of prefix labels:

> "`{a1,..,ax}::P` *replaces every action label* `n` *in the alphabet of* `P` *with the labels* `a1.n,..,ax.n`.
> *Further, every transition* `n->X` *in the definition of* `P` *is replaced with the transitions* `({a1.n,..,ax.n} -> X)`"

`({a1,..,ax} -> X)` is just shorthand for the set of transitions `(a1 -> X)`, ..., `(ax -> X)`.

FSP—Process labelling

Now we can compose two clients and a server as follows:

```
||TWO_CLIENT_SERVER
  = (a:CLIENT || b:CLIENT ||
     {a,b}::(SERVER/{call/request, wait/reply})).
```

We can compose N clients and one server following this pattern:

```
||N_CLIENT_SERVER(N=2) =
  (( c[i:1..N]:CLIENT )
  ||
  {c[i:1..N]}::(SERVER/{call/request, wait/reply})
  ).
```

FSP—Variable hiding

To reduce complexity, it is often useful to hide variables:

> *'Given a process* `P`*, the process* `P\{a1,..,aN}` *is the same as* `P`*, but with actions names* `a1,..,aN` *removed from* `P`*, making these silent. Silent actions are named* `tau` *and are never shared".*

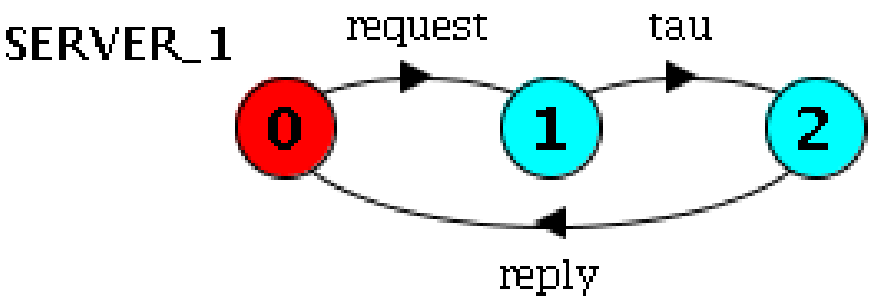
An alternative is to list the variables that are not to be hidden:

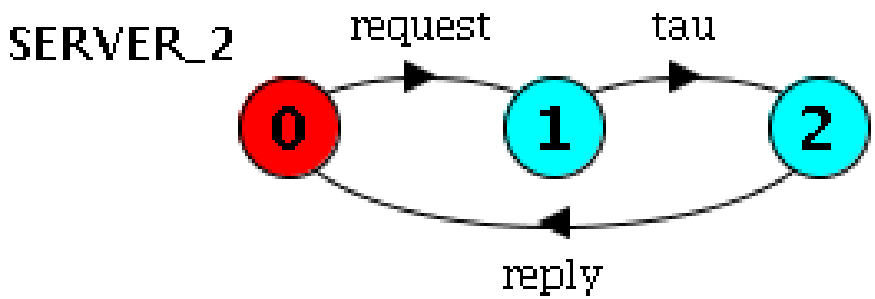
> *'Given a process* `P`*, the process* `P@{a1,..,aN}` *is the same as* `P`*, but with actions names other than* `a1,..,aN` *removed from* `P`*".*

FSP—Variable hiding

Hence the following two definitions result in the same LTS:

```
SERVER_1 = (request -> service -> reply -> SERVER_1)
           @{request, reply}.
SERVER_2 = (request -> service -> reply -> SERVER_2)
           \{service}.
```





Coming to a theatre near you

Deadlock, synchronisation and monitors in FSP.