Observable Behaviour

- **Observable behaviour can be defined in terms of experimentation.**
- **Consider a coffee machine. We don't need to understand and don't what to understand how the coffee machine works. All we want to understand is what it does when I interact with it. We can experiment with it a record the observable behaviour against the different interactions (conversations) that we have with it.**

The "what" is all about specifying the behaviour of a system in terms of its external observable behaviour. By external I really mean the appropriate scope from the observers point of view.
Observable behaviour can be thought of in terms of an experiment. It takes us all back to the days of the science lab at school. We would have to observe what happened and write down our observations and from this deduce what was really going on.
Consider a coffee machine. We all understand what a coffee machine is and most of us can work a coffee machine but it certainly helps having the instructions on the side. The instructions should be in terms of the observable behaviour of that machine.
Being able to specify the observable behaviour is a fundamental part of what the pi-calculus provides. This doesn't mean we should program in it nor does it mean that we all have to be experts in it. Rather it suggests using it as a formal model that is the basis for expressing observable behaviour.

Encoding observable behaviour

- **Formal semantics**
  - **Understanding the essential nature of concurrency**
  - **Reasoning about behaviour of concurrent systems**
  - **Developing tools to that produce correct systems**

- **Pi-calculus of Professor Robin Milner**
  - **Provides an algebra**
  - **Provides formal semantics for concurrent communicating mobile processes**

What is it this pi-calculus and what is it all about?
It is all about providing an understanding of the essential nature of concurrency that enables us to reason about the behaviour of concurrent systems. It is about developing tools that produce correct systems in the first place – in the same way that many breathed a sigh of relief when we moved away from pointers in C++ to references in Java we may achieve the same with respect to concurrency and communication.
The pi-calculus was invented by Professor Robin Milner in the early part of the 1990's. It is an algebra with formal semantics for concurrent communication and for the mobility of processes.
Now that is a easy to say but what on earth does it mean to have mobility of processes. In essence what is meant is that processes can change the links through which they communicate. You can think of it as changing the topology of a network of processes in which links represent potential to communicate.

Mobility can be used to implement Frank McCabe's doctors surgery/waiting room example. Mobility can also be used to implement Martin Chapman's callback example. In both cases what happens is that a channel (a port) is passed from one process to another to enable the receiver to communicate with some process at the other end of the passed channel.
In essence the pi-calculus is a programming language that supports these features in a formal way such that you can reason over one or more programs written in the calculus.

The pi-calculus in brief

Operation	Notation	Meaning
Prefix	π.P	Sequence
Action	a(y), ā(y)	Communication
Summation	a.P + b.Q $\Sigma \pi_i P_i$	Choice
Recursion	P={…..}.P	Repetition
Replication	!P	Repetition
Composition	P | Q	Concurrency
Restriction	(*vx*)P	Encapsulation

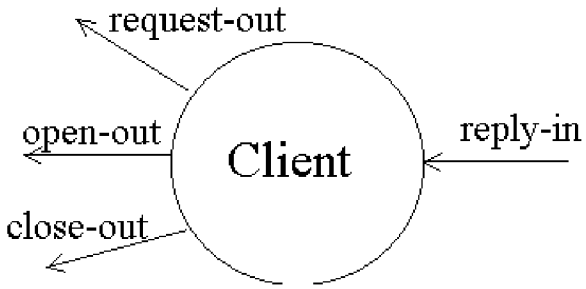
This is the pi-calculus. It's pretty tight in terms of the operators.
Message passing is based on sending names along channels.
The Prefix part means that when π is done the process then behaves like P, whatever P is defined as.
The Actions are communications along channels. In this case a(y) takes a message called y as an input and _(y). The use of a complement (the overbar symbol) signifies an output channel. Channels are connected (in to out and out to in) based on name matching of the channels.
Summation is no more than a choice. So in the example if an a is passed then the process behaves like a P and if a b is passed the process behaves like a Q. The convenience function, the summation, is shorthand for some number I of processes P with **π** actions.
Recursion is achieved by recursive definition of a process.
Replication allows us to define an arbitrary number of processes P all running in parallel.
Composition is achieved by using the bar symbol (|). This means that P and Q run concurrently. If P has an output port and Q an input port of the same name then their exists the possibility of them communicating along the channel denoted by the channel name.
Of course channels can have elaborate names to include the types that they might accept too.Channels can themselves be passed from one process to another. When this happens the process passing the channel looses control of that channel and the receiver then has control. In this way we can pass a printService from one process to another or more elaborately we could pass a creditCheckService in the same way.
Encapsulation of the sort (yx)P means that y passed along channel x to process P becomes local to P. That is nothing can then see what it does with y on x and no one can communicate along the name or see what y is.

For the rest of this usecase I shall use the suffixes "-in" and "-out" to denote the port and complemented port names.

A simple client process

A sequential process

Client(open,close,request,reply) =
open-out.request-out.reply-in.request-out.reply-in.close-out.0

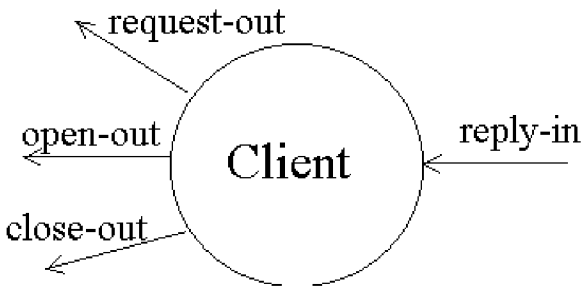


Here we have a process called Client. It has three output channels and one input channel. Alas diagrams, while revealing the structure, do nothing to describe the dynamic behaviour, what we call the observable behaviour.
The definition, in pi terms, is above and really says that:
Client has four ports (yes we call them ports too), namely open, close, request and reply.
The client sends a message on the open port and then on the request port and waits for a response on the reply port. When it receives a message on the reply port it sends another message on the request port again and waits for a reply. Then it sends a message on the close port and terminates (this is the meaning of the 0 term at the end).

A simple client process

A repetitive process

Client(open,close,request,reply) =
open-out.request-out.reply-in.request-out.reply-in.close-out.Client(open,close,request,reply)

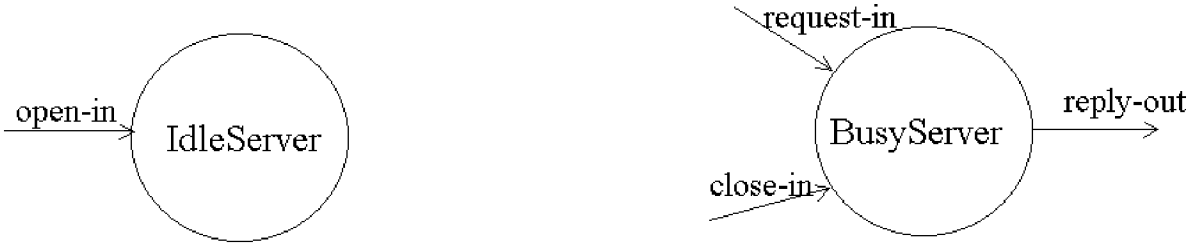


Same as before except it doesn't terminate. I've used a recursive definition to enable this to be expressed. Notice that the diagrammatic form doesn't change.

A simple server process

A process with choices to make

```
IdleServer(o,req,rep,c) =   o-in.BusyServer(o,req,rep,close)
BusyServer(o,req,rep,c) =           req-in.rep-out.BusyServer(o,req,rep,c) +
                                    c-in.IdleServer(o,req,rep,c)
```

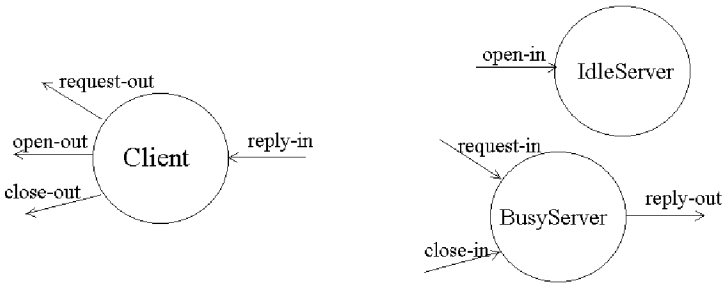


Here we have a server. We want to model the server so that we map state into allowable observable behaviour. It's a richer way of describing the interaction of a system so that we understand the context in which things happen. It helps to clarify and in the end allows us to express such a system unambiguously.
In this example the server has two states, IdleServer and BusyServer. It can do different things in each of these states.
When it is behaving like an IdleServer all it can do is receive a message on it's open port. When it does it then behaves like a BusyServer.
A Busy server accepts a message on it's request port and send a message on it's reply port and then behaves like a BusyServer. Alternatively a Busy server can accept a message on it's close port and then behave like an IdleServer again.
Notice the use of process states as behavioral definition of a process. This the naming of the processes as distinct patterns of behaviour allows us to capture observable behaviour over state.
In this way we can obviate the need to have conditional logic by elaborating the states that this logic would normally yield and declare a process for each one and define the behaviour of each state. We can use a choice (the + operator) to manage which gets fired when as they bind to the guard (the operation if you like and the following process definition if there is one).

Composition

- **We've seen composition through recursive definition in the simple client. We have seen choice (state mappings) in the simple server. What we have yet to do is combine behaviours (web services through their external descriptions).**

- **How do we combine the client and server processes into a single expression that completely expresses the combined (and if appropriate synchronized) behaviour?**



So far all we have done is look at how to describe a single process and how it behaves. We have seen how we can model communications in and out through ports. We have seen how we can deal with sequences, choices and states.
But we have not looked at concurrency, communication that is implied by concurrency and replication. This is what we shall look at now.
What happens when we want to combine our notion of a Client with our notion of a server in its Idle and Busy state?
We can see that we can connect the Client to a server in a idle state and intuitively we can see that only one communication is valid.

We can see that a client can connect to a server in it's busy state and from the pi-program that represents the client and server we can understand the message sequences involved and the subsequent state changes made on the server to go from being busy back to idle.
Once again the diagrams alas are not sufficient to do this. But the pi-programs are completely sufficient to do this.

Composition

SYSTEM = (!Client | IdleServer)
$Client_i$ | IdleServer
$Client_i$ | BusyServer
$Client_j$ | IdleServer
$Client_j$ | BusyServer
…..

- **When Clienti has started an exchange with IdleServer**
- **No other Client can then communicate with the server**
- **Until Clienti has finished and the server is once again Idle**

If we look at a SYSTEM that is composed of many clients (by using the replication operator) and a single server we would write such a system as we can see here.
When the ith Client starts an exchange with the IdleServer the IdleServer transitions into behaving like a BusyServer. It accepts the ith Clients requests until the close message is communicated to it. The ith client accepts the responses accordingly.
When the server transitions, after a close, to an IdleServer it is once again available to other clients so that the jth Client can then issue an open have the request accepted and so on.
Now I'm not suggesting this is a good pi-program. We might want to have another process that acts as a mediator and load balances in some way so that we have multiple servers. But the point is that we can write down clear and unambiguous pi-programs for each process in terms of it's observable behaviour, not in terms of what it may do internally, and then compose them into a set of processes that communicate at the correct points and in the correct way. In effect we can enforce types safe communication based on data types and behavioural types.
Keeping the example simple might allow us to look at more complex interactions and ones that are more meaningful in a business context. However I would contend that the Client/Server example serves as a simple starting point.

Note: Things get very interesting when you try to timeouts and exceptions. One approach is to model these from the observers point of view. So in the case of the client not receiving a response the client will receive a proxy for non-receipt of a message. This represents the timeout but modeled as the receipt of a message that is a timeout message. The same can be done for any exception at the server end because it results in observable behaviour that is indistinguishable from a timeout. It is more a case of what the client wishes to do about the non-receipt of a message that we model but only in terms of what we can observe.