THE FOUNDATIONS OF THE π-CALCULUS

Scientific Seminar Paper

by Simon Heinzel

12.01.2012

Seminar: Concurrency Theory

presented by Dr. Martin Neuhäuser

Chair for Dependable Systems & Software
Saarland University

tutored by

Andrea Turrini

Content

Foundation of the π-Calculus

1 Introduction

The π-Calculus – A model of interleaving concurrency

„Communication is a fundamental and integral part of computing, whether between different computers on a network, or between components within a single computer"(1). Therefore it is important to be able to describe communication between different entities formally. There exist a lot of different calculi to model this communication. You can coarsely divide them into two sections: calculi that model „interleaving concurrency" and calculi that model „true concurrency". Interleaving concurrency means that the different actions of agents and their communication are put in a strong order. At one specific time there can be only one action. When using „true concurrency", we are confronted with several actions at one specific time. An action is not instantaneous and one cannot differentiate the single steps.
The π-Calculus models the first idea of concurrency, the interleaving concurrency. In the π-Calculus the communication is not an extra action that a system can do. The system consists only of different communicating parts. This also includes internal computations. In this paper the syntax of the π-Calculus will be presented and explained, using examples. There will also be given the definition of bisimulation for expressions of the π-Calculus.

2 The π-Calculus

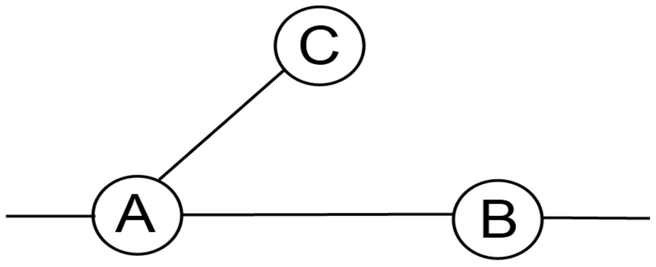
2.1 Development of the π-Calculus from CCS

The Calculus of Communicating Systems (CCS) was developed by the English computer scientist Robin Milner around 1980. It is a process calculus, used to formally describe concurrent systems. CCS is a good process calculus if you have to model a static system of communicating processes, but what do you do if you want to introduce some mobility? Robin Milner found an answer to this problem in developing the π-Calculus out of the existing CCS. To get a better understanding we first have to look closer at the word „mobility" and define what we mean with it.

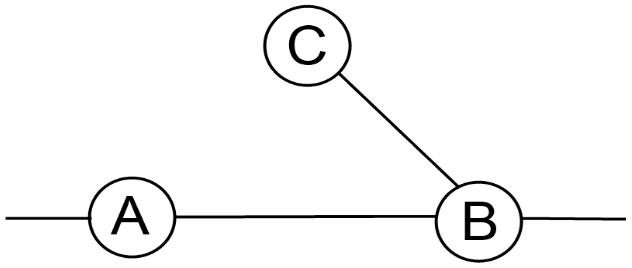
2.2 Mobility

Mobility is a word that is often used by computer scientists in different ways. That is why we have to precisely define what we mean by mobility. In general „mobility means that *something* moves from one place to another. Thereby it is of significance what kind of entity moves, and in what space does it move." (2)
In case you want to model a network of communicating processes, where processes can send messages to one another, you can model this with CCS because it is static. But as soon as messages contain links to other processes about which, the receiving process has no knowledge, you need to change the connections and this we call mobility.

To show that we now look at an abstract system as shown below. Process A is connected to process B and C, but B and C are not connected at all.



Now we want the two processes C and B to form a connection and the connection between A and C to vanish. The changed system would look like shown below:

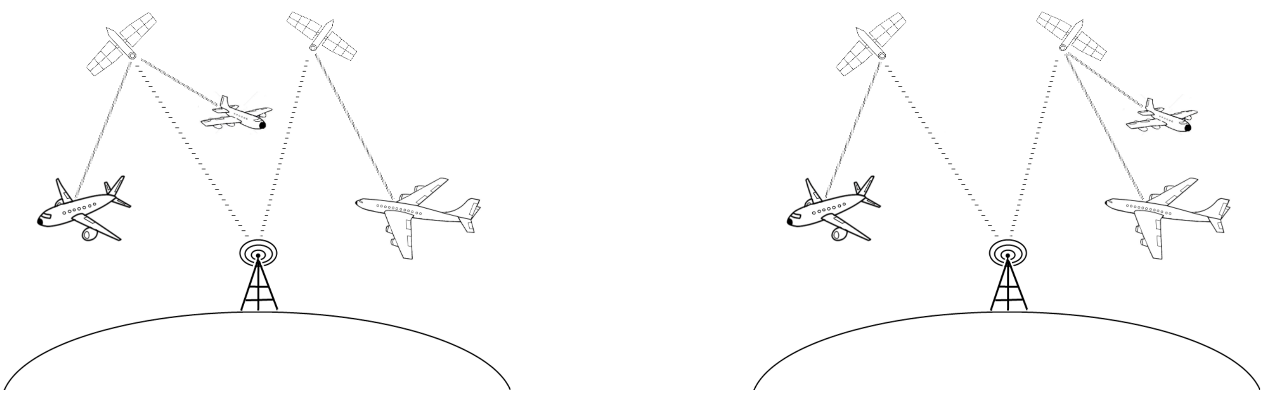


This can be achieved in two different ways. Either the whole process C moves to B or just a link to C moves to B. In both cases the link between A and B is obviously needed to transfer the information.
In one case the entity that moves is a whole process, in the other case it is simply a link to that process.
So we have to decide whether the processes or the links should be moved. If we want the process to move in a spacial way, we have to rearrange every link that points to the process. If we move a link we don't have this problem. However, how do we define the „position“ of a process? If we define it by its links we also change the position of the process and thereby move both when moving one of its links.

2.3 Example for Mobility in Practise

Let us explain this by an example. Consider airplanes that are flying around the earth and must be tracked somehow to know where the airplane is at a specific time. To make it simple imagine that there is a satellite that tracks planes and sends this information to a central station on the ground. Because the earth is round, there must be more than just one satellite to track the planes. Consider that every satellite has a fixed connection to the central station and a connection to every plane it tracks. Now it can happen, that a plane moves too far around the earth to be tracked by its satellite, because every satellite has its fixed range. Now, the connection to the plane has to be transmitted from one satellite to the other.

It is important to differantiate between the physical movement of the airplanes and the virtual movement of the connections between planes and satellites. How to model this problem in π-Calculus Syntax will be shown in section 4 „Example“.



3 Syntax & Reaction Rules

3.1 Syntax of the π-Calculus

We now proceed with defining the syntax of the π-Calculus. Let *N* be an infinite set of names and let small letters a,b,c…. range over names. Let P,Q,R,... range over process expressions. Robin Milner defines process expressions as a combination of six different elements:

$$P := \sum P_i \mid \alpha . P' \mid P_1 | P_2 \mid (x)P' \mid [x=y]P' \mid !P'$$

a) S*ummation* $\sum_{i \in I} P_i$ (3)

This process behaves like one of the P_i. This means the process can transform in one of the different P_i and make the transitions the P_i can do. After deciding for one of the P_i the transitions of the others are not possible anymore. The index set I is finite. If the index set I is empty we will write **0** and name it *inaction.* The inaction can't do anything. If the cardinality of I is two we can also write $P_1 + P_2$ as a binary sum.

b) P*refix* $\alpha . P$ (4)

The prefix can have three different forms: *1)* $\bar{y}\,x.P$ *, 2)* $y(x).P$ or 3) $\tau . P$

1) $'\bar{y}\,x.P\,'$ is called *negative* prefix. A negative prefix means that the process $\bar{y}\,x.P$ wants to output the name x over the *output port* $\bar{y}$. Afterwards the process behaves like P.

2) $'y(x)'$ is called *positive* prefix. The port y is an *input port* of the process. The process $y(x).P$ is able to receive an arbitrary name z at the port y . It then behaves like P {z/x} . This means that every free occurrence of x in P is substituted by z. The exact definition of substitution is given below. A *positive* prefix $y(x)$ binds the name x in the process $y(x).P$.

3) $'\tau'$ is called *silent* prefix. The τ-action means that the process $\tau.P$ does some internal action in which it is not possible to look into for external observers. This could be an internal computation or other internal action. The communication between two processes which act in parallel, forms also a nonvisible τ-transition. This means that from outside we cannot distinguish between internal actions or communication between processes.

c) C*omposition* $P_1|P_2$ (5)

This process behaves like P_1 and P_2 acting in parallel. There are two possiblities for the two processes to react. First they may act independently. That means P_1 as well as P_2 react without affecting the other. This covers our intuition that they are running in parallel. Second they may synchronize their actions. Precisely, if P_1 is able to execute an output action $\bar{x}$ and P_2 an input action at the same port x , they may synchronize and create the silent action τ , what means that for external observers the synchronisation between processes is not visible in detail.

d) R*estriction* $(x)P$ (6)

This process behaves like P except that the actions x and $\bar{x}$ are not possible for this process. However synchronization between parallel processes in P over the ports x and $\bar{x}$ is still possible because this results in a $\tau-action$. This gives us a perfect possibility to force processes to synchronize. The restriction $(x)P$ binds the name x in the process $(x)P$.

e) M*atch* $[x=y]P$ (7)

This process behaves like P if the names x and y are identical. Otherwise it is not possible to form any further transition, what means it behaves like the process **0**. With this operator we can ensure that some process evolves only further if the structure of the process fulfills specific conditions.

f) *Replication* $!P$ (8)

This operator says that the Process $!P$ stands for an infinite number of copies of P able to run concurrently. $!P$ is able to „spawn“ a new copy of P running in parallel every time it is needed. This operator is very important, because it gives us the possibility to model a process that is able to do the same transitions for an infinite number of times. With this one could model a server which can answer to a call every time, or the call of a function.

Further Definitions:

a) For all names of a process P we will write the expression n(P). We define the *free names* of a process P by all the names which exist in P but are not bound either by a restriction or by a positive prefix. For the *free names* of a process P we will write fn(P) . The *bound names* of a process are all names that occur in P and are not free. So bn(P) := n(P) \ fn(P).

b) Substituiton is defined as follows. We write $P\{z_1/x_{1,...}, z_n/x_n\}$ for the simultaneous substituiton of all free occurences of $x_i \ (for \ 1 \leq i \leq n)$ in P. In case there are *bound names* which are equal to one of the x_i the *bound names* must be renamed in order for them to be distinguished from each other.

Notes

a) We will not differantiate processes which only differ in a change of their *bound names*. For example $a(x).0$ is equal to $a(y).0$.

b) In the simple π-Calculus there is no construction that expresses parametric definitions. Therefore recursive definitions like $A(x) := \bar{x} \ .a(y).A(y)$ are not possible. But we will see, that with the *replication* operator we are able to define such parametric processes.

c) The paranthesis of a formula of the π-Calculus is as follows:

Summation < Composition < Restriction, Prefix, Match
Therefore holds:

$$P|Q+R=(P|Q)+R$$
$$!P|R=(!P)|R$$
$$(x)P|Q=((x)P)|Q$$

d) We will sometimes call bound names „private links“ and names which are free „public names“.

3.2 Reaction Rules

We now proceed to infer transitions. First we have some simple transition rules which are similar to those in CCS:

Tau : $\dfrac{-}{\tau.P \rightarrow^{\tau} P}$ (9)

Sum_l : $\dfrac{P \rightarrow^{\alpha} P'}{P+Q \rightarrow^{\alpha} P'}$ (10)

Sum_r : $\dfrac{Q \rightarrow^{\alpha} Q'}{P+Q \rightarrow^{\alpha} Q'}$

Restriction : $\dfrac{P \rightarrow^{\alpha} P'}{(y)P \rightarrow^{\alpha} (y)P'} \quad if \ \alpha \neq y(x) \wedge \alpha \neq \bar{y}x$ (11)

Further there is another simple rule for the match-operator:

Match : $\dfrac{P \rightarrow^{\alpha} P'}{[x=x]P \rightarrow^{\alpha} P'}$ (12)

For simple prefixes holds:

Output : $$\frac{-}{\bar{x}\,y.P \rightarrow^{\bar{x}y} P} \tag{13}$$

Input : $$\frac{z \notin fn((y)P)}{x(y).P \rightarrow^{x(z)} P\{z/y\}} \tag{14}$$

This models a simple input. We have to demand the y to be unequal to the *free names* in P to prevent confusion with other names, but it actually can be the previously *bound name* from the input prefix.

Furthermore we have the communication rule, for parallel communicating processes:

Communication : $$\frac{P \rightarrow^{\bar{x}y} P',\, Q \rightarrow^{x(z)} Q'}{P|Q \rightarrow^{\tau} P'|Q'\{y/z\}} \tag{15}$$

The rule also exists in the other direction, with P inputting and Q outputting a name.

There is one more rule for parallel composition :

Par_r : $$\frac{P \rightarrow^{\alpha} P'}{P|Q \rightarrow^{\alpha} P'|Q}$$

Par_l : $$\frac{Q \rightarrow^{\alpha} Q'}{P|Q \rightarrow^{\alpha} P|Q'}$$

Here we have to be careful. We have to add the further requirement that

$$\textit{if } \alpha = x(y) \textit{ then } y \notin fn(Q) \textit{ for } \text{Par\_r and } \textit{if } \alpha = x(y) \textit{ then } y \notin fn(P) \textit{ for } \text{Par\_l}$$

This is important, because otherwise we would be able to do the following:

$$(x(y).P|Q)|\bar{x}\,z.R$$

and with the Par_l and Communication Rule we would be able to infer the following transition :

$$(x(y).P|Q)|\bar{x}\,z.R \rightarrow^{\tau} (P|Q)\{z/y\}|R$$

which is obviously wrong in case y appears free in Q, since we only want the y in P to be substituted. To solve this problem we have to rename the *bound name* y first. We need a new name which is not free neither in P nor in Q and can form the following correct transition:

$$(x(y).P|Q)|\bar{x}\,z.R \rightarrow^{\tau} (P\{w/y\}|Q)\{z/w\}|R$$

There is no explicit rule for the *Replication*. But we don't need a special rule for that, as you will see in section 3.6 „Structural Congruence“.

3.3 Syntactic Sugar

3.5.1 Polyadic π-Calculus

Now we want to send more than just one name per message. So we want something that looks like this:

$x(y_{1,}y_{2,}y_{3,\ldots,},y_n)$ and $\bar{x}\,y_1 y_2 y_3 \ldots . y_n$

If we try to model this in a naive and simple way we get a construct that looks like this:

$x(y_{1,}y_{2,}y_{3,\ldots,},y_n).P:=x(y_1).x(y_2)\ldots x(y_n).P$ and similarily

$\bar{x}\,y_1 y_2 y_3 \ldots . y_n . P:=\bar{x}\,y_1.\bar{x}\,y_2 \ldots \bar{x}\,y_n.P$

But now consider the following example:

First we define three processes

$$P:=x(z_1,z_2).P'$$
$$R:=x(w_1,w_2).R'$$
$$Q:=\bar{x}\,y_1\,y_2.Q'$$

Now let's put them in parallel

$$P|Q|R$$

We expect the process Q to send the two names y_1 and y_2 to one of the two processes P or R.
But if we write this down to our naive implementation we get something that looks like this:

$$x(z_1).x(z_2).P'|\bar{x}\,w_1.\bar{x}\,w_2.Q'|x(y_1).x(y_2).R'$$

We can see that Q can send the two names to two different processes. This is not what we want, because we want these operations to be instantaneous. To achieve this property we have to invent something else. We have to ensure that there is no interference between different processes.
To transfer the whole message we first send a new fresh name p (one which was not used so far). Then we send all the other names over the restricted port p. The example from above would now look like this

$$P:=x(p).p(z_1).p(z_2).P'$$
$$R:=x(p).p(w_1).p(w_2).R'$$
$$Q:=(p)(\bar{x}\,p.\bar{p}\,y_1.\bar{p}\,y_2.Q')$$

You can see that both the names y_1 and y_2 are either sent to P or to R because only one of them possesses the link p. This obviously also works when there is an arbitrary number of names to be sent.

3.5.2 Recursive Calls

It may be useful to define processes in a parametric way, as agents are defined, so they have the form:

$$A(x_1, x_{2,} \ldots, x_n) := Q_A$$

In the basic π-Calculus there is no explicit syntax for this. But we can express parametric-defined processes and their recursive calls with help of the replication.

Let $A(x_1, x_{2,} \ldots, x_n) := Q_A$

and P some process in which A is „called".
Three steps are necessary to convert this into the basic π-Calculus.

a) Create a new fresh name a for A

b) For every process R , define $\widetilde{R}$ as the result if you replace every call $A(x_1, x_{2,} \ldots, x_n)$ in R with $\bar{a}\, x_1 x_2 x_3 \ldots . x_n$

c) Replace P with : $\hat{P} := (a)(\widetilde{P} \,|\, !\, a(x_1 x_2 x_3 \ldots . x_n) . \widetilde{Q_A})$

Let's consider the following simple example:

$$A(x) := \bar{x} z . A(x) = Q_A$$
$$B(x) := x(p) . p . B(x) = Q_B$$

$$P := (x) A(x) \,|\, B(x)$$

That means A is able to send the name z over the port x every time it wants to, and B is able to receive a name over port x and then call this name. You can see the process B as an executer which receives a name and then calls that name.
This system is able to make a τ-action first, then B should make the action z and then the initial system should be reached again. This means the system looks like this:

$$(x) A(x) | B(x) \;\; \to^{\tau} \;\; (x) A(x) | z.B(x) \;\; \to^{z} \;\; (x) A(x) | B(x)$$

Now we want to construct something that is as powerful as the system above.
We first invent two new names for A and B , let's say a and b . And now find $\widetilde{Q_A} = \bar{x} z .\, \bar{a} x . 0$ $\widetilde{Q_B} = x z . \bar{b}\, x . 0$. This leads us to

$$\hat{P} = (abx)(\bar{a}\, x.0 | \bar{b}\, x.0 \,|\, !(a(x) . \bar{x}\, z . \bar{a}\, x.0) | !(b(x) . x(p) . p . \bar{b}\, x.0))$$

Let's have a look as to whether this system is able to act like the system above with the recursive calls. We will need more τ-actions but in the end it will be as powerful as the system above.

$$(abx)(\bar{a}x.0|\bar{b}x.0|!(a(x).\bar{x}z.\bar{a}x.0)|!(b(x).x(p).p.\bar{b}x.0))$$

$$\rightarrow^{\tau}(abx)(0|\bar{b}x.0|\bar{x}z.\bar{a}x.0|!(a(x).\bar{x}z.\bar{a}x.0)|!(b(x).x(p).p.\bar{b}x.0))$$

$$\rightarrow^{\tau}(abx)(0|0|\bar{x}z.\bar{a}x.0|!(a(x).\bar{x}z.\bar{a}x.0)|x(z).z.\bar{b}x.0|!(b(x).x(p).p.\bar{b}x.0))$$

$$\rightarrow^{\tau}(abx)(0|0|\bar{a}x.0|!(a(x).\bar{x}z.\bar{a}x.0)|z.\bar{b}x.0|!(b(x).x(p).p.\bar{b}x.0))$$

$$\rightarrow^{z}(abx)(0|0|\bar{a}x.0|!(a(x).\bar{x}z.\bar{a}x.0)|\bar{b}x.0|!(b(x).x(p).p.\bar{b}x.0))$$

$$\equiv\ (abx)(\bar{a}x.0|\bar{b}x.0|!(a(x).\bar{x}z.\bar{a}x.0)|!(b(x).x(p).p.\bar{b}x.0))$$

The transformations which happen when you omit the parallel zeros or reorder the terms will be a valid action as shown in the following section „Structural Congruence“.

3.6 Structural Congruence

We now define structuaral congruence over processes. If two processes, P and Q are structually congruent, written $P \equiv Q$, we call them equal. Two processes are structurally congruent if we can transform one into another with the following rules:

a) Summation: The Summation is commutative $P+Q \equiv Q+P$
in our notation, we can reorder every term in a summation.

b) Change of *Bound Names:* as we said the change of *bound names* doesn't change a system

c) Parallel Operator: for the parallel operator hold a few rules:

1. Commutativity : $P|Q \equiv Q|P$
2. Associativity : $(P|Q)|R \equiv P|(Q|R)$
3. Neutral Element : $P|0 \equiv P$

d) Restriction: the rules for restriction are:

1. Commutativity : $(x)(y)P \equiv (y)(x)P$
2. Scope change : $(x)(P|Q) \equiv P|(x)Q$ if $x \notin fn(P)$
3. Vanishing : $(x)0 \equiv 0$

e) Replication: A simple congruence rule applies for this replication, that covers our intuition that !P is able to spawn an arbitrary number of copies of P: $!P \equiv P|!P$

3.7 Scopes

We now get further into so called „scopes". The scope of a name is the area in which the specific name is known. It is important to identify correctly the scope of each name in order to know how a current system looks like at a specific time. In a static system like CCS scopes do not change. But in the π-Calculus we can send names which are channels. Especially we can send names that are private. This means that a scope can be expanded or reduced. It is also possible that ports in different scopes have the same names. But what happens if we expand these scopes into each other? This expansion or reduction is veryfied by structural congruence. With the „Scope change"-rule we can expand a scope to a parallel composed process, or reduce the scope by using the rule in the other direction.
The following examples will explain how these operations change a system. (The figures are taken from Robin Milner, A Calculus of Mobile Processes Part 1, Page 15 to 17).

3.7.1 Scope Extrusion

We first show how scopes can be expanded. If some process passes a private link there are three possibilities, which we will discuss now. Think of three processes P, Q and R. Process P has a private link to R and another link to Q. The system looks like this:

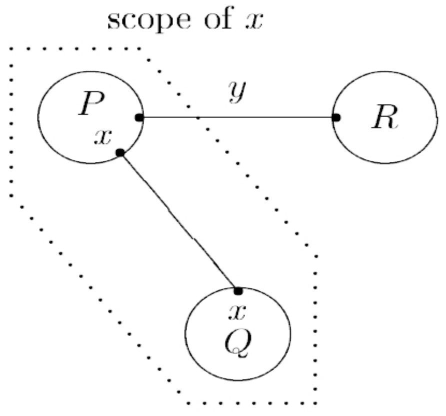
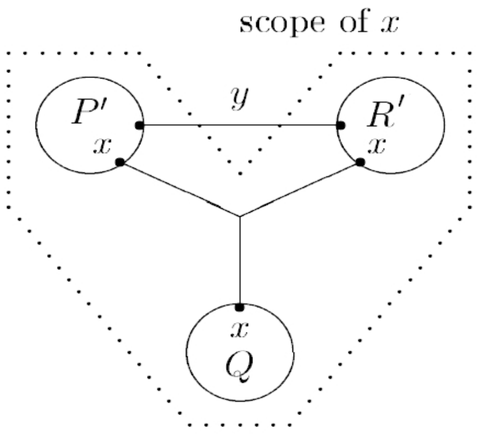


Fig. Sc1

Now P wants so send the private link x over y to Q. A formula could look like this:

$(x)(P|Q)|R$ where $P := \bar{y}x.P'$ and $R := y(z).R'$. And $x \notin fn(R)$

The result of the synchronization of P and Q over the y link looks like this:



$(x)(P|Q)|R \rightarrow^{\tau} (x)(P'|Q'|R'\{x/z\})$
We can transform this transition because of structural congruence law d.2. As you see $x \notin fn(R)$ Therefore we can expand the restriction over R. The system now looks like this if we suppose that $x \in fn(P')$, so P' still possesses the private link x:

Fig Sc2.

There is also the possibility that $x \notin fn(P')$. So P sends the link x to R and then „forgets“ about it. The system transforms after the same τ-action to a system that looks like this:

$$(x)(P|Q)|R \rightarrow^{\tau} P|(x)(Q'|R'\{x/z\})$$

This transition is also verified by the structural congruence law b.2, because we know that $x \notin fn(P')$, so we can reduce the restriction of x to the two parallel processes Q' and R'.

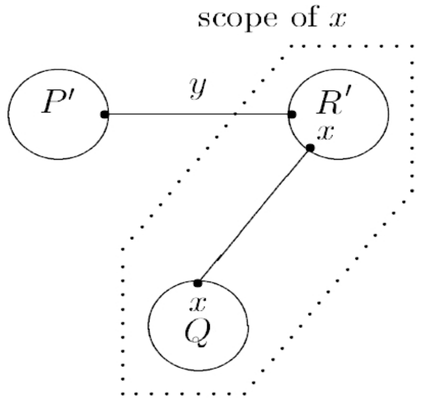


Fig Sc3.

Let's consider a case in which it is a bit more difficult. Again we consider the system $(x)(P|Q)|R$. But now R possesses a public link with name x. This is possible because we do not demand every link to have a different name, as long as they appear in different scopes. But now after the transition the scopes will overlap. To differentiate between the two links one of them has to be renamed. We will choose the private name x, because we know the whole scope of this link. The public name cannot be changed in a simple way because the destination of the link is unknown. The system now looks like this:

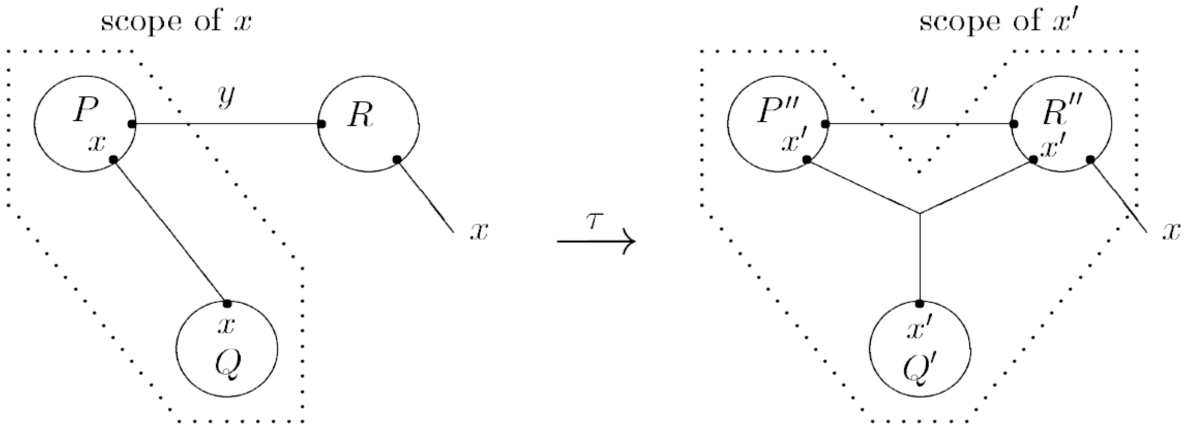


Fig. Sc4

3.7.2 Scope Intrusion

We have discussed the expansion of a scope by sending a private link, but what happens if we send a public name into a scope of a private link with the same name? Consider this example:

Process P owns two public links x and y to the processes Q and R. R owns a private link x to the process S. P now wants to send its link x over y to R.

$P|Q|(x)(R|S)$ where $P := \bar{y}x.P'$ and $R := y(x).Q'$

The diagram of the system would look like this:

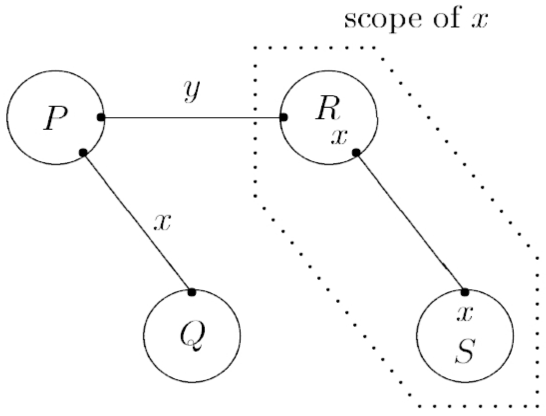
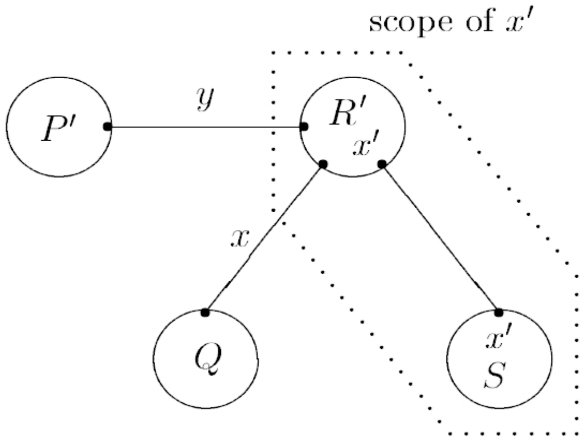


Fig Sc5

If the processes P and R now communicate over their y-link and P sends the public link x to Q, the system changes as follows:

$P|Q|(x)(R|S) \rightarrow^{\tau} P'|R|(x')(R'|S)$ {x'/x}

The diagram of the system looks like this:



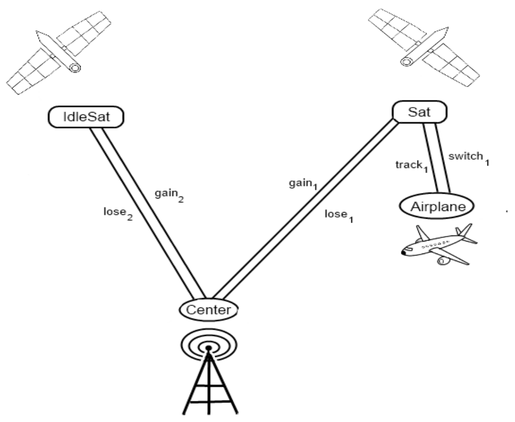
The private name x of the processes R and S has to be changed, to avoid collision with the public name x.

Fig. Sc6

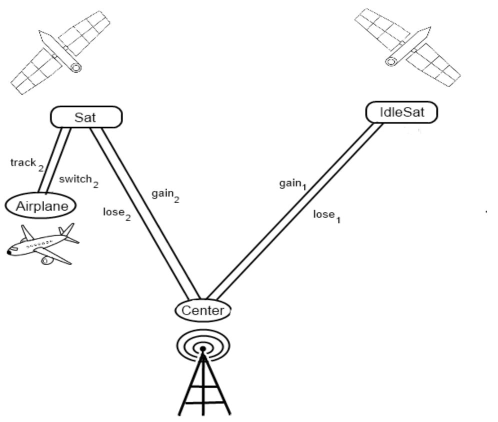
4 Example

In the introductory section we mentioned the example of tracked airplanes. We will now delve into this idea in greater detail and express it in the π-Calculus. First, we simplify the idea a little bit, and reduce the problem to one airplane and two switching satellites. The satellites are connected with a central station, which we call „Center“. This station handles the switching of the two different satellites. That means it can tell one satellite to stop the tracking of the airplane and the other to take over the tracking (gain, lose). The satellite itself can have a connection to the airplane. It can track the plane, and tell it to switch to the other satellite (track, switch). The satellite can be in two states, one state in which it tracks a plane and a second idle state in which it waits for instruction.

This system may look like the one in Fig.5



We now want the system to change, in a way that the plane is now tracked by the second satellite. See Fig. 6



It is necessary to describe the different agents and show that these agents behave like the system we want. We define four processes which we call:

IdleSat, Sat, Center and *Airplane.*

We are now able to construct parametric agents which can be written down. Their parameters show what their current connections are.

Sat is able to track the airplane as long as it possesses the link to the airplane. If it is told to lose the airplane it sends this information and becomes idle. Note that the old satellite needs to send the „connection to the new satellite“ also to the airplane, because the airplane needs this connection. This also means that the connection *lose* has to carry the connection to the new satellite. The process might look like this:

$$Sat(track, switch, gain, lose) := \overline{track}.Sat(track, switch, gain, lose)$$
$$+\ lose(t,s).\overline{switch}\ \ t\,s.IdleSat(gain, lose)$$

We now need to define the idle satellite. This process does nothing but waiting the center to tell it to gain an airplane.

$$IdleSat(gain, lose) := gain(t,s).Sat(t, s, gain, lose)$$

Airplane needs to answer the connection track, and has to be able to change the connection it answers to :

$$Airplane(track, switch) := track.Airplane(track, switch) + switch(t,s).Airplane(t,s)$$

And now to the *center*. *Center* possesses both links to the two satellites and knows which satellite momentarily tracks the airplane. In our initial configuration the airplane is tracked from satellite 1, that means *Center* is able to tell satellite 1 to lose its connection to the plane. Remember that the connection to satellite 2 has to be sent with the *Lose* signal. Secondly it has to tell satellite 2 to gain the connection. Then *Center* is able to act vice versa. This is modelled by the two recursive definitions below:

$$Center_1 := \overline{lose_1}\, track_2\, switch_2 . \overline{gain_2}\, track_2\, switch_2 . Center_2$$

$$Center_2 := \overline{lose_2}\, track_1\, switch_1 . \overline{gain_1}\, track_1\, switch_1 . Center_2$$

Now we put all these processes in parallel, so that the system looks like this:

$$(track_1, track_2, switch_1, switch_2, gain_1, gain_2, lose_1, lose_2)[\,Airplane(track_1, switch_1)$$
$$|\,Sat(track_1, switch_1, gain_1, lose_1)\,|\,IdleSat(gain_2, lose_2)\,|\,Center_1]$$

We restricted all channels to force communication between the different entities. We now have to show that this system is able to perform transitions we want. So the system looks afterwards like this:

$$(track_1, track_2, switch_1, switch_2, gain_1, gain_2, lose_1, lose_2)[\,Airplane(track_2, switch_2)$$
$$|\,IdleSat(gain_1, lose_1)\,|\,Sat(track_2, switch_2, gain_2, lose_2)\,|\,Center_2]$$

Let's take a look at how this works. First *center* performs the action $\overline{lose_1}\, track_2 switch_2$ and synchronizes with the input action $lose(t,s)$ of *Sat* to a τ-transition. Then *Sat* makes the output action $\overline{lose_1}\, track_2 switch_2$ and synchronizes with the input action of *Airplane*. Both these transitions are derived from the Communication and Restriction Reaction Rule. The system then looks like this :

$$(track_1, track_2, switch_1, switch_2, gain_1, gain_2, lose_1, lose_2)[\,Airplane(track_2, switch_2)$$
$$|\,IdleSat(gain_1, lose_1)\,|\,IdleSat(gain_2, lose_2)\,|\,\overline{gain_2}\, track_2 switch_2 .\ Center_2]$$

Now the System performs the action $\overline{gain_2}\, track_2\, switch_2$ which synchronizes with

$$IdleSat(gain_2, lose_2) = gain(t,s).Sat(t, s, gain_2, lose_2)$$

After these three τ-transitions the system looks like we want:

$$(track_1, track_2, switch_1, switch_2, gain_1, gain_2, lose_1, lose_2)[\,Airplane(track_2, switch_2)$$
$$|\,IdleSat(gain_1, lose_1)\,|\,Sat(track_2, switch_2, gain_2, lose_2)\,|\,Center_2]$$

5 Strong Bisimulation – Equity of Processes

We now proceed to show an idea for an equity in the relation between processes. We could simply use the identity or the structural congruence, but both these ideas of equity are „too strong“. Consider the following example:

$P := a(x).0$ and $Q := a(x).0 + a(x).0$

These two processes are neither identical nor structurally congruent, but we want them to be equivalent, because obviously both processes can only perform the same transitions. That is why we will use „bisimulation“.
First, here is the definition of bisimulation developed for CCS:

Let P and Q be processes. A binary relation S is a bisimulation if $P\,S\,Q$ implies the two following attributes:

a) If $P \rightarrow^{\alpha} P'$ then for some Q' holds that $Q \rightarrow^{\alpha} Q'$ and $P'\,S\,Q'$

b) If $Q \rightarrow^{\alpha} Q'$ then for some P' it holds that $P \rightarrow^{\alpha} P'$ and $P'\,S\,Q'$

This means that either P and Q have to be able to simulate every transition the other process can perform, and the resulting processes must be again in the bisimulation.
For the π-Calculus we have to be a bit more careful because in an input action a process can receive a name, that it uses later on. That means it is not sufficient that the processes P' and Q' continue to simulate. We have to claim the simulation for all instantiations of the names that can be received. So we have to slightly alter the definition of bisimulation for the π-calculus :

Let P and Q be processes. A binary relation S is a bisimulation if $P\,S\,Q$ implies following :

a) If $P \rightarrow^{\alpha} P'$ and α is a τ or output action, then for some Q' it holds that $Q \rightarrow^{\alpha} Q'$ and $P'\,S\,Q'$

b) If $Q \rightarrow^{\alpha} Q'$ and α is a τ or output action, then for some P' it holds that $P \rightarrow^{\alpha} P'$ and $P'\,S\,Q'$

c) If $P \rightarrow^{x(y)} P'$ and $y \notin n(P) \cup n(Q)$, then for some Q' holds that $Q \rightarrow^{x(y)} Q'$ and $\forall w\, P'$ {w/y} $S\,Q'$ {w/y}

d) If $Q \rightarrow^{x(y)} Q'$ and $y \notin n(P) \cup n(Q)$, then for some P' holds that $P \rightarrow^{x(y)} P'$ and $\forall w{:}\ P'$ {w/y} $S\,Q'$ {w/y}

This definition is not explicit, since many different relations preserve these conditions, for example, the identity preserves them. Thus we search the largest bisimulation. The relation $\sim$, strong bisimilarity, is defined as follows:

For two processes P and Q, $P \sim Q \Leftrightarrow \exists S\ with\, P\,S\,Q$ and $S\ is\ a\ bisimulation$

But first we show that Structural Congruence is a bisimulation.
Therefore we have to show that the five different laws are valid.

1. Reordering the terms of a sum

It is enough to show that a bisimulation S exists such that $P\,S\,Q$ with $P:=P_1+P_2$ and $Q:=P_2+P_1$. If we can switch the terms in a binary sum, we can part the sum with arity n into different binary sums, and inductively change the terms, and so reorder the n terms in an arbitrary way.
To show that this bisimulation exists we present one and show that it is a bisimulation

$$S:=\{(P_1+P_{2,}P_2+P_1)|P_{1,}P_2\,processes\}\cup Id$$

Obviously, Id is a bisimulation, so we only need to show that the expansion does not change this fact.

$$If\ P_1+P_2\rightarrow^{\alpha}P_1'\ then\ P_1\rightarrow^{\alpha}P_1'(reaction\ rule\ \text{sum\_l})$$
$$\Rightarrow the\ following\ transition\ is\ also\ possible\ P_2+P_1\rightarrow^{\alpha}P_1'\ \text{and}\ P_1'\,S\,P_1'\ because(P_1',P_1')\in Id$$

$$If\ P_1+P_2\rightarrow^{\alpha}P_2'\ then\ P_2\rightarrow^{\alpha}P_2'(reaction\ rule\ \text{sum\_r})$$
$$\Rightarrow the\ following\ transition\ is\ also\ possible\ P_2+P_1\rightarrow^{\alpha}P_2'\ \text{and}\ P_2'\,S\,P_2'\ because(P_2',P_2')\in Id$$

The other direction and the case that α is an input transition are shown the same way, therefore S is a bisimulation. □

2. Change of *bound names*

$P\sim P$ {z/x} $\quad if\ z\notin n(P)$

This means that P is strong bisimilar to a process with renamed *bound names,* if the *bound name* is substituted with a fresh name. The proof is done by structural induction over the depth of P.

3. Parallel Operator

a) Commutativity

The bisimulation for $P_1|P_2\sim P_2|P_1$ is given by

$$S:=\{(P_1|P_2,P_2|P_1)|P_{1,}P_2\,processes\}\cup Id$$

To prove this we have to consider three different cases. From the two parallel processes we can infer three different transitions:

1. $P_1|P_2\rightarrow^{\alpha}P_1'|P_2$
2. $P_1|P_2\rightarrow^{\alpha}P_1|P_2'$
3. $P_1|P_2\rightarrow^{\tau}P_1'|P_2'$

case 1:

$$If\ P_1|P_2\rightarrow^{\alpha}P_1'|P_2,\ then\ P_1\rightarrow^{\alpha}P_1'(reaction\ rule\ \text{Par\_l})$$
$$\Rightarrow\ P_2|P_1\rightarrow^{\alpha}P_2|P_1'(reaction\ rule\ \text{Par\_r})\ \text{and}\ (P_1'|P_{2,}P_2|P_1')\in S$$

case 2:

$$If\ P_1|P_2\rightarrow^{\alpha}P_1|P_2',\ then\ P_2\rightarrow^{\alpha}P_2'(reaction\ rule\ \text{Par\_r})$$
$$\Rightarrow\ P_2|P_1\rightarrow^{\alpha}P_2'|P_1(reaction\ rule\ \text{Par\_l})\ \text{and}\ (P_1'|P_{2,}P_2|P_1')\in S$$

case 3:

$$\mathit{If}\ P_1|P_2 \rightarrow^{\tau} P_1'|P_2',\ \mathit{then}\ P_1 \rightarrow^{x(y)} P_1' \text{ and } Q \rightarrow^{\bar{x}z} Q'$$

$$\text{or } P_1 \rightarrow^{\bar{x}z} P_1' \text{ and } Q \rightarrow^{x(y)} Q' (\mathit{reaction\ rule}\ \text{Communication})$$

$$\Rightarrow\ P_2|P_1 \rightarrow^{\tau} P_2'|P_1' (\mathit{reaction\ rule}\ \text{Communication}) \text{ and } (P_1'|P_2', P_2'|P_1') \in S$$

The two cases that α is an input transition are similar to the two that are already proven.
Therefore S is a bisimulation. □

b) Associativity

$$S := \{((P_1|P_2)|P_3, P_1|(P_2|P_3)), P_1, P_2, P_3\ \mathit{processes}\}$$

To show this we would have to show it for about 30 different cases. The proof is very similar to the one shown above. One of the cases will be shown here, the other cases are very similar.
$\mathit{If}\ (P_1|P_2)|P_3 \rightarrow^{\tau} (P_1'|P_2')|P_3$, and $P_1 \rightarrow^{\bar{x}y} P_1'$ and $P_2 \rightarrow^{x(z)} P_2'$ Therefore holds:

$$(P_1|P_2)|P_3 \rightarrow^{\tau} (P_1'|P_2')|P_3 (\text{Communication})$$

$$(P_1|P_2)|P_3 \rightarrow^{\tau} (P_1'|P_2')|P_3 (\text{Par\_l})$$

Now we have to be able to infer the transition $\mathit{If}\ P_1|(P_2|P_3) \rightarrow^{\tau} P_1'|(P_2'|P_3)$. To do so, we have to use the Par_r rule for the process $(P_2|P_3)$. To be able to infer the transition, we need to ensure that $z \notin \mathit{fn}(P_3)$. To do so we can rename z in P_2 to a fresh name, because z is bound in P_2. Therefore, the transition $(P_2|P_3) \rightarrow^{x(z')} (P_2'|P_3)$ is possible, so we can also infer $P_1|(P_2|P_3) \rightarrow^{\tau} P_1'|(P_2'|P_3)$ and $(((P_1'|P_2')|P_3), P_1'|(P_2'|P_3))$ is again in the bisimulation. □

c) Neutral Element

$$S := \{(P|0, P), P\ \mathit{process}\} \cup \mathit{Id}$$

This is obviously a bisimulation, because Id is a bisimulation and **0** is not able to form any transition. □

4. Restriction

a) Commutativity

The commutativity of $(x)(y)P \sim (y)(x)P$ is proved by showing that

$$S = \{((x)(y)P, (y)(x)P), P\ \mathit{process}\}$$

is a bisimulation. □

b) Scope change

$$(x)(P|Q) \sim (x)P|Q \quad if\ x \notin fn(Q)$$

We need to show that the relation $S=\{((x)(P|Q),(x)P|Q)|P,Q\ processes, x \notin fn(Q)\} \cup Id$
There are 16 different cases to be shown. This is done similar to the proofs above. One case will be shown here.

$(x)(P|Q) \rightarrow^{\tau} (x)(P'|Q')$ and $P \rightarrow^{\bar{a}y} P'$ and $Q \rightarrow^{a(y)} Q'$ $where\ a \neq x\ because\ x \notin fn(P)$

$(x)P \rightarrow^{\bar{a}y} P'\ because\ a \neq x \quad \Rightarrow (x)P|Q \rightarrow^{\tau} (x)P'|Q'$

and $((x)(P'|Q'),(x)P'|Q') \in S$

□

c) Vanishing

The statement $(x)0 \sim 0$ is easy to prove, because neither (x)0 nor 0 is able to perform any transition, so the trivial bisimulation is $S=\{(0,(x)0)\}$. □

Strong bisimilarity is not preserved by substituiton, as the following example shows:

$$\bar{x}u.0|y(v).0 \ \sim\ \bar{x}u.y(v).0+y(v).\bar{x}u.0$$

This bisimulation is only preserved if we demand x and y to be unequal. If we substitue y by x we get the following equatation:

$$\bar{x}u.0|x(v).0 \ \sim\ \bar{x}u.x(v).0+x(v).\bar{x}u.0$$

This relation is not true since the left side can form a τ-transition, by synchronizing over the port x, which the right side can not.

Strong bisimilarity is also not preserved by the input prefix. This results from the fact that it is not preserved by substituiton. Because after an input action there will be a substituiton.

$$\bar{x}u.0|y(v).0 \ \sim\ \bar{x}u.y(v).0+y(v).\bar{x}u.0$$

The same example as above. But now consider

$$z(y).(\bar{x}u.0|y(v).0) \ \sim\ z(y).(\bar{x}u.y(v).0+y(v).\bar{x}u.0)$$

The strong bisimilarity claims that for an input action the two processes continue to simulate for every instantiation. But if y is instantiated to x the same problem appears as in the substituiton problem above.

Early Bisimulation

The definition of strong bisimulation from above is only one possibility how bisimulation can be defined. You can switch the quantifiers in the condition:

If $P \rightarrow^{x(y)} P'$ and $y \notin n(P) \cup n(Q)$ then for some Q' holds that $Q \rightarrow^{x(y)} Q'$ and
$\forall w P'$ {w/y} $S Q'$ {w/y}

and it's inverse (the two conditions c) and d) are changed) to

If $P \rightarrow^{x(y)} P'$ and $y \notin n(P) \cup n(Q)$ then for all w there is a Q' and it holds that $Q \rightarrow^{x(y)} Q'$
P' {w/y} $S Q'$ {w/y}

The resulting bisimulation is strictly weaker than the one defined above. For the original strong bisimulation, we want one derivative of the other process to be able to simulate the first one for all instantiations of w. For the second definition, we only claim that for every instantiation of w there exists a derivative that simulates it. We do not demand them to be the same. The simulating transitions may be different for different instantiatons of w.
Thus you can think, that the instantiation of w happens before the transition and the process can afterwards „decide“ which transition to choose. That is why this bisimulation is called „early bisimulation“.
To show an example where two processes can bisimulate each other with early bisimulation but not with strong bisimulation consider this:

$$P := x(w).R + x(w).0$$

$$Q := x(w).R + x(w).0 + x(w).[w=p].R$$

Now $Q \rightarrow^{x(w)} [w=p].R$. P has to simulate this transition. With the early bisimulation, P „knows“ whether w=p and can choose $x(w).R$ or $x(w).0$. But with the strong bisimulation P is not able to make a transition such that P' is able to simulate $[w=p].R$ for every instantiation of w since $[w=p].R$ can act like 0 or like R dependent of w.

6. Conclusion

The presented π-Calculus is a base for many different programming languages or base for more specific process calculi. Because of its simplicity there are a lot of different expansions and usages. For example the spi-calculus, which is a calculus for cryptographic protocols. It expands the syntax of the π-Calculus with syntax for encrypting and decrypting [4]. Another example is the Probabilistic Applied Pi-Calculus which expand the original calculus with operators which can decide nondeterministic and probabilistic for different choices[5]. Even in molecular biology the π-Calculus is base for a calculus, named k-calculus, used for describing cellular communication[6] and modelling cell networks[7].

6 References

[1] „Communicating and Mobile Systems : The π-Calculus“, Robin Milner, © Cambridge University Press 1999

[2] A Calculus of Mobile Processes, Part I, paper, Robin Milner, Joachim Parrow, David Walker, University of Edinburgh 1989

[3] A Calculus of Mobile Processes, Part II, paper, Robin Milner, Joachim Parrow, David Walker, University of Edinburgh 1989

[4] A calculus for cryptographic protocols: The spi calculus, Martín Abadi, Andrew D. Gordon, In Proceeding of the 4th ACM Conference on Computer and Communication Security, 1997

[5] A Probabilistic Applied Pi-Calculus, Jean Goubault-Larrecq, Catuscia Palamidessi, Angelo Troina, LSV-ENS Cachan

[6] Formal molecular biology, Vincent Danos, Cosimo Laneve, Department of Computer Science, University of Bologna, France

[7] Application of a stochastic name-passing calculus to representation and simulation of molecular processes, Corrado Piami, Aviv Regev, Ehud Shapiro, William Silverman, 2001