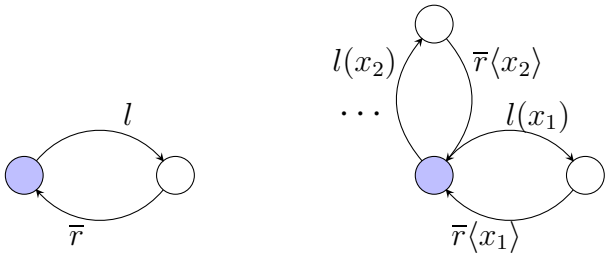
# Einführung in den $\pi$-Kalkül - Teil II

Der Fundamentale Unterschied zwischen CCS und dem $\pi$-Kalkül besteht in der Möglichkeit, Namen zu versenden und zu empfangen. In CCS haben wir Prozesse oft durch Transitionssysteme beschrieben. Im $\pi$-Kalkül ist es im Allgemeinen nicht mehr möglich, einen Prozess $P$ durch ein statisches Transitionssystem zu beschreiben. Das Problem liegt in der potentiell unendlichen Zahl von Namen, welche wiederum zu unendlich vielen Transitionsregeln für einzelne Zustände führen kann.




Der einfache Puffer $Buf = l.\overline{r}.Buf$ aus CCS lässt sich im $\pi$-Kalkül detaillierter modellieren. Wir können nun tatsächlich den Pufferinhalt modellieren: $Buf_\pi = l(x).\overline{r}\langle x\rangle.Buf_\pi$. Wollten wir diesen Puffer als Transitionssystem in einem Prozesskontext darstellen, so müssten wir alle Namen die über $l$ gesendet werden in Betracht ziehen. Das bedeutet konkret, dass wir mit struktureller Kongruenz und $\alpha$-Umbenennung den mit new erzeugten Scope jedes betroffenen Namens auf den Puffer ausdehnen müssten.

Betrachten wir zum Beispiel die Situation

$$!(\text{new } a\ (\overline{l}\langle x\rangle.0)) \mid Buf_\pi.$$

Um ein vollständiges Transitionssystem zu erhalten, müssten wir die abzählbar unendlich vielen Namen $\alpha$-umbenennen und würden so ein Transitionssystem mit unendlich vielen vom Startzustand ausgehenden Kanten erhalten. (Natürlich würden wir zusätzlich noch unendlich viele Knoten mit jeweils genau einem Übergang $\overline{l}\langle x_i\rangle$ zum Nullprozess erhalten. Diese sind allerdings nicht so pro-

blematisch wie die unendlich vielen Übergangsmöglichkeiten für einen einzelnen Knoten.)

Ein statisches Transitionssystem kommt also offensichtlich nicht in Frage. Nun stellt sich die Frage, ob wir Prozesse des $\pi$-Kalküls durch ein dynamisches Transitionssystem, also eines dessen Transitionen seine Struktur verändern können, beschreiben können.

Selbst wenn dies für allgemeine Prozesse des $\pi$-Kalküls nicht möglich sein sollte, kann man immer noch Einfache Systeme dahingehend untersuchen.

Im Folgenden werden wir die Kapitel 9.4-10.2 aus **communicating and mobile systems: the $\pi$-calculus** von Robin Milner diskutieren.

## 9.4 Der polyadische $\pi$-Kalkül

Oft möchte man Nachrichten die aus mehreren Namen bestehen senden und empfangen. Um dies nun auch im $\pi$-Kalkül zu ermöglichen wird die Syntax entsprechend erweitert durch

$$\overline{x}\langle z_1 \dots z_n \rangle.P \text{ und } x(y_1 \dots y_n).Q$$

Logischerweise müssen die $y_i$ paarweise verschieden sein, denn sonst gibt es ein Definitionsproblem:

$$\overline{a}\langle bc \rangle.0 \mid a(xx).P \quad \rightarrow \quad 0 \mid \{bc/xx\}P?$$

Die $z_i$ jedoch müssen nicht zwingend verschieden sein.

Praktischerweise kann man den polyadischen $\pi$-Kalkül im bisher definierten, monadischen Kalkül kodieren.

Milner gibt uns als "ersten Versuch" die naive Kodierung, die darin besteht, die Elemente des Vektors einzeln nacheinander zu senden bzw. empfangen:

$$\overline{x}\langle z_1 \rangle.\cdots.\overline{x}\langle z_n \rangle.P \text{ und } x(y_1).\cdots.x(y_n).Q$$

Diese funktioniert, wenn es nur einen Sender und einen Empfänger für den Kanal $x$ gibt. Dann werden $z_1$ bis $z_n$ praktisch sequentiell gesendet und empfangen. Das Problem tritt erst auf, wenn mehr als ein Sender oder Empfänger für $x$ existiert. Im Fall mehrerer

Sender können sich die Nachrichten miteinander "verheddern", wie man im Buch sieht. Existiert mehr als ein Empfänger, so kann eine Nachricht auf mehrere Empfänger "zerstreut" werden.

Das Problem liegt in der potenziellen Vielzahl von Verbindungen zwischen verschiedenen Prozessen über den Kanal $x$. Also müssen wir zuerst eine dieser Verbindungen auswählen und dann die Nachricht naiv kodiert explizit über die gewählte Verbindung schicken. Praktisch realisieren wir dies, indem wir einen komplett neuen Namen $w$ über $x$ schicken und danach die Nachricht komponentenweise über $w$ schicken:

$$\begin{aligned} \overline{x}\langle z_1 \dots z_n\rangle.P &\quad\mapsto\quad \text{new } w\ (\overline{x}\langle w\rangle.\overline{w}\langle z_1\rangle.\cdots.\overline{w}\langle z_n\rangle.P) \\ x(y_1 \dots y_n).Q &\quad\mapsto\quad x(m).m(y_1).\cdots.m(y_n).Q \end{aligned}$$

Dadurch, dass wir einen neuen Namen $w$ über $x$ senden, wählen wir eine der Verbindungen die über $x$ bestehen und öffnen einen privaten Kanal $w$ der nur den an dieser Verbindung beteiligten Prozessen bekannt ist.

Nun haben wir eine funktionierende Kodierung, doch zum polyadischen $\pi$-Kalkül fehlt noch ein Schritt. Würden wir den polyadischen Kalkül allein über diese Kodierung definieren, so würden wir noch eine unerwünschte Reaktion zulassen:

$$\begin{aligned} \overline{x}\langle abc\rangle.Q \mid x(yz).P &\quad\mapsto\quad \text{new } w\ (\overline{x}\langle w\rangle.\overline{w}\langle a\rangle.\overline{w}\langle b\rangle.\overline{w}\langle c\rangle.Q) \mid x(m).m(y).m(z).P \\ &\quad\rightarrow\quad \text{new } w\ (\overline{w}\langle a\rangle.\overline{w}\langle b\rangle.\overline{w}\langle c\rangle.Q \mid w(y).w(z).P) \end{aligned}$$

Es handelt sich wie man sieht um die Reaktion zwischen zwei Aktionspräfixen mit unterschiedlicher Anzahl von Parametern. Dies ist von Milner explizit nicht gewünscht, also definiert er für den polyadischen $\pi$-Kalkül eine neue Reaktionsregel REACT durch:

$$\text{REACT}: \quad (x(\vec{y}).P + M) \mid (\overline{x}\langle\vec{z}\rangle.Q + N) \rightarrow \{\vec{z}/\vec{y}\}P|Q$$

wobei $\vec{x}$ und $\vec{y}$ die gleiche Länge haben müssen.

## 9.5 Rekursive Definitionen

Die Prozessausdrücke des $\pi$-Kalküls unterscheiden sich von den in Definition 4.1 definierten CCS-Prozessausdrücken durch die Option, Namen über Kanäle zu verschicken und die im $\pi$-Kalkül zur Verfügung stehende Replikation. Andererseits fehlt dem $\pi$-Kalkül im Gegenzug die Möglichkeit Prozessdefinitionen $A\langle\vec{a}\rangle$ einzusetzen. In CCS waren Prozessdefinitionen die einzige Möglichkeit, Rekursion zu beschreiben. Nun stellt sich die Frage, ob wir mit Replikation dasselbe ausdrücken können wie mit Prozessdefinitionen.

Gegeben ist also eine Prozessdefinition $A(\vec{x}) = Q_A$ wobei in $Q_A$, dem "Rumpf" von $A$, Aufrufe $A\langle\vec{z}\rangle$ von $A$ vorkommen dürfen. Weiter gegeben ist ein Prozess $P$, der Aufrufe von $A$ enthält.

- Wir erfinden einen neuen Namen $a$. Senden von $\vec{x}$ entlang $a$ soll den Aufruf $A\langle\vec{x}\rangle$ repräsentieren.
- Die Prozessdefinition $A(\vec{x}) = Q_A$ übersetzen wir in $!a(\vec{x}).Q_A[A := \overline{a}]$, das Empfangen der Parameter $\vec{x}$ für $A$ und die anschliessende Ausführung des Rumpfes von $A$. Im Rumpf $Q_A$ sind dementsprechend alle Aufrufe von $A$ durch das Senden über $a$ ersetzt.
- Den Prozess $P$ mit der gegebenen Prozessdefinition übersetzen wir zu $\hat{P} \stackrel{def}{=} \text{new } a\ (P[A := \overline{a}] \mid !a(\vec{x}).Q_A[A := \overline{a}])$

Die Replikation stellt uns beliebig viele Aufrufe von $A$ zur Verfügung.

Im Gegensatz zum ursprünglichen über Prozessdefinitionen gegebenen Prozess $P$ haben wir nun einen Prozess $\hat{P}$, der an den Stellen wo eine Prozessdefinition $A$ aufgerufen wird, einen zusätzlichen $\tau$-Übergang begeht, sich ansonsten aber identisch zu $P$ verhält. Dass es so ist, ist nicht zwingend auf den ersten Blick klar. Im Gegensatz zu einer sequentiellen Ausführung des Aufrufs $A\langle\vec{x}\rangle$ durch Einsetzung haben wir nun einen nebenläufigen Ablauf. Was wäre zum Beispiel mit dem Prozess $R = A\langle\vec{x}\rangle.A\langle\vec{y}\rangle$?

Kurzes Nachschlagen in Definition 4.1 zeigt uns, dass $R$ kein gültiger Prozessausdruck ist. Der Aufruf einer Prozessdefinition bildet immer das Ende eines gültigen Prozessausdrucks. Somit ist

klar, dass die nebenläufige Ausführung der Aufrufe von $A$ kein Problem darstellt.

**Beispiel**

Die Prozessdefinition $Buf(l,r) = l(x).\overline{r}\langle x\rangle.Buf(l,r)$ des einfachen Puffers wird übersetzt in

$$!b(l,r).l(x).\overline{r}\langle x\rangle.\overline{b}\langle l,r\rangle.$$

Der Prozess $P = Buf(l,r) \mid \overline{l}\langle x\rangle$ mit der Prozessdefinition $Buf$ wird übersetzt in

$$\text{new } b\ (\overline{b}\langle l,r\rangle \mid \overline{l}\langle x\rangle \mid !b(l,r).l(x).\overline{r}\langle x\rangle.\overline{b}\langle l,r\rangle).$$

Wir haben gesehen, wie über Prozessdefinitionen definierte Prozessausdrücke mit Replikation ausgedrückt werden können. Dies erlaubt es uns, im $\pi$-Kalkül Prozessdefinitionen zu benutzen, denn wir wissen nun wie wir auf diese Weise erzeugte Ausdrücke in den $\pi$-Kalkül übersetzen können. Oft ist es angenehmer und erscheint natürlicher, rekursive Prozeduren über Prozessdefinitionen auszudrücken.

Umgekehrt können wir auch Replikation über Prozessdefinitionen ausdrücken. Gegeben eine Replikation $!P$. Wir wissen: $!P \equiv P|!P$. Als Prozessdefinition wählen wir einfach $P_{rep} \stackrel{def}{=} P|P_{rep}$, das rekursive "Auffalten" der Replikation. In einem Prozess $Q$ den wir übersetzen möchten ersetzen wir dann einfach die Replikation $!P$ durch den Prozessaufruf $P_{rep}$.

## 9.6 Abstraktionen

Abstraktionen spielen vor allem in den späteren Kapiteln eine interessante Rolle. Milners Motivation, Abstraktionen bereits jetzt einzuführen besteht im wesentlichen darin, die Variablenbindung im $\pi$-Kalkül zu vereinheitlichen. Im grundlegenden $\pi$-Kalkül gibt es bereits zwei Möglichkeiten zur Variablenbindung:

$$\pi ::= \tau \mid x(y) \mid \overline{x}\langle y\rangle$$
$$P ::= \sum \pi_i.P_i \mid P|P' \mid \text{new } x\ P \mid !P$$

Einerseits haben wir die Bindung durch den Empfang von Namen über einen Kanal und andererseits die Bindung durch new. Erlaubt man weiterhin die Benutzung von Prozessdefinitionen $A(x) = P$, so haben wir durch den Aufruf $A\langle\vec{z}\rangle \stackrel{def}{=} \{\vec{z}/\vec{x}\}P$ eine weitere Methode zur Variablenbindung geschaffen. Diesen Weg der Variablenbindung können wir natürlich vermeiden, in dem wir die Prozessdefinitionen und die betroffenen Prozesse wie im letzten Teilkapitel gesehen übersetzen. Der Einfachheit und Vollständigkeit halber wollen wir aber trotzdem untersuchen, wie wir die Bindung über Prozessdefinitionen mit Abstraktionen ausdrücken können.

**Definition**

Für einen Prozess $P$ und einen (sinnvollerweise in $P$ frei vorkommenden) Namen $x$ definieren wir die Abstraktion $(x).P$. Der Grad dieser Abstraktion ist 1.

Allgemein definieren wir eine Abstraktion vom Grad $n$ durch $(x_1).\cdots.(x_n).P$. Diese Abstraktion stellt $n$ Binder zur Verfügung.

$((x).P)\langle y\rangle \stackrel{def}{=} \{y/x\}P$, die Applikation der Abstraktion $(x).P$ auf einen Namen $y$, bindet alle freien Vorkommen von $x$ in $P$ durch $y$. Für höhergradige Abstraktionen wird das ganze analog und wie im Buch zu sehen definiert.

Als abkürzende Notation verwenden wir:

- new $A \stackrel{def}{=}$ new $\vec{x}$ $(A\langle\vec{x}\rangle)$
- $x\ A \stackrel{def}{=} x(\vec{y}).(A\langle\vec{y}\rangle)$

Ausserdem benutzen wir statt der Prozessdefinition $A(\vec{x}) \stackrel{def}{=} Q$ die definierende Gleichung der Abstraktion $A \stackrel{def}{=} (\vec{x}).Q$.

Somit haben wir alle drei bisherigen Wege der Variablenbindung durch Abstraktionen ausgedrückt.

Wie man an dieser Stelle im Buch sieht kann man sehr schön Operationen auf Abstraktionen definieren. Abstraktionen werden uns in dieser Ausarbeitung nicht weiter beschäftigen. In Kapitel 11.5 jedoch spielen Abstraktionen eine nützliche Rolle.

# 10 Anwendungen des $\pi$-Kalküls

Uns beschäftigt jetzt die Frage, wie wir das Verhalten eines Prozesses $P$ durch syntaktische Vorraussetzungen einschränken können. Im speziellen betrachten wir das in der Praxis häufig auftretende Critical Section Problem. Gegeben ist ein System bestehend aus mehreren nebenläufigen Prozessen und einer Ressource. Nun möchte man garantieren, dass sich zu jedem Zeitpunkt immer nur höchstens ein Prozess in einer Critical Section bezüglich der Ressource befindet.

Eine Critical Section bezüglich einer Ressource ist ein auf die Ressource zugreifender eine Abfolge von Anweisungen, die bei nebenläufiger Ausführung durch andere Prozesse (die ebenfalls auf diese Ressource zugreifen) beeinträchtigt werden kann.

Betrachten wir zum Beispiel ein Datenbanksystem bestehend aus dem Datenbankserver, der die Ressource darstellt, und mehreren Clients. Die Clients können Datensätze aus der Datenbank lesen und schreiben. Angenommen, zwei Clients möchten Änderungen am selben Datensatz vornehmen. Beide Clients lesen in beliebiger Reihenfolge den Datensatz aus der Datenbank. Beide Clients nehmen nun ihre Änderungen vor. Der erste Client schreibt seinen geänderten Datensatz in die Datenbank, der zweite Client schreibt seinen Datensatz. In der Datenbank sind nun effektiv nur die Änderungen des zweiten Clients vorgenommen worden. Die Critical Section ist also in diesem Fall der Anweisungsblock "Lesen; Ändern; Schreiben". Wenn sich beide Clients in der Critical Section befinden, kann es ein Problem geben. Die nebenläufige Ausführung von atomaren Anweisungen der Clients verursacht das Problem. Dieses spezielle Critical Section Problem kann man lösen indem man Zugriffsrechte auf die Ressource exklusiv vergibt.

Im $\pi$-Kalkül modellieren wir den Zugriff auf die Ressource über einen Kanal $x$. Wir betrachten ein System $P$ bestehend aus mehreren nebenläufigen Prozessen. Die Ressource lässt sich konkret durch einen Prozess $R$ modellieren der mit $P$ parallel komponiert wird und nur über den Kanal $x$ erreichbar ist. Da uns die Ressource selbst nicht interessiert, betrachten wir nur das System $P$. Wir wollen verhindern, dass zwei nebenläufige Komponenten von $P$ "gleichzeitig" versuchen, auf die Ressource zuzugreifen (also über

$x$ senden):

$$P \not\equiv \text{new } \vec{z}(\ldots \mid \overline{x}\langle a\rangle.Q_1 \mid \overline{x}\langle b\rangle.Q_2 \mid \ldots)$$

Für allgemeine Systeme $P$ aus dem $\pi$-Kalkül ist diese strukturelle Kongruenz mit Proposition 9.13 entscheidbar. Allgemein ist strukturelle Kongruenz von zwei Prozessen ohne weiteres nicht zwingend entscheidbar. Weiterhin wäre es schwierig, eine Menge von syntaktischen Eigenschaften zu finden, die uns die gewünschte Eigenschaft unter Reduktion erhalten, wenn wir beliebige Prozesse zulassen. Daher werden wir uns auf eine unter Reduktion abgeschlossene Teilmenge des $\pi$-Kalküls beschränken. Es handelt sich um Einfache Systeme.

## 10.1 Einfache Systeme

**Definition**

Ein Prozess $P$ heisst *Einfaches System*, wenn er zu einer Normalform

$$(*)\ P \equiv \text{ new } \vec{z}\ (M_1| \ldots |M_m|!N_1| \ldots |!N_n)$$

strukturell kongruent ist, wobei die $M_i$ und $N_j$ Summen sind, die weder eine Replikation noch eine parallele Komposition enthalten.

Eine schöne Eigenschaft von Einfachen Systemen ist die Abgeschlossenheit unter Reduktion: Ist $P$ ein Einfaches System und gilt $P \to P'$, so ist $P'$ ebenfalls ein Einfaches System.

**Beweis**

**Fall 1**: $\tau$-Übergang in einer der Komponenten von $P$

Wir können o.B.d.A. (durch Umsortieren und die strukturelle Kongruenz $!R \equiv R|!R$) annehmen, dass $M_1 \xrightarrow{\tau} M_1'$ gilt. $M_1'$ enthält weder Replikation noch parallele Komposition, da $M_1$ keines der beiden enthielt.

$M_1' \equiv \text{new } \vec{y}\ L$, wobei $L$ keine Replikation oder parallele Komposition enthalten kann. Damit ist $P'$ ein Einfaches System.

**Fall 2**: Reaktion zwischen zwei Komponenten von $P$

Wir können o.B.d.A. (durch Umsortieren und die strukturelle Kongruenz $!R \equiv R|!R$) annehmen, dass $M_1|M_2 \to Q$ gilt.

D.h. $P' \equiv$ new $\vec{z}(Q|M_3|\ldots|M_m|!N_1|\ldots|!N_n)$. $Q$ enthält keine Replikation da weder $M_1$ noch $M_2$ eine Replikation enthielten.

Man sieht: $Q \equiv$ new $\vec{y}(M_1'|M_2')$, wobei $M_1'$ und $M_2'$ Summen sind, die keine Replikation oder parallele Komposition enthalten. Mit struktureller Kongruenz folgt: $P'$ ist einfach.

**Beispiel**

Puffer sind einfach:

$$\overline{b}\langle l, r\rangle \mid !b(l, r).l(x).\overline{r}\langle x\rangle.\overline{b}\langle l, r\rangle.$$

Das Mobile Phone Beispiel aus Kapitel 8 ist ein Einfaches System.

Ein Einfaches System $P$ ist eine Komposition aus sequentiellen Prozessen und replizierten sequentiellen Prozessen. Die Anzahl der parallelen Komponenten kann sich nur über die Replikation vergrössern.

**CCS-Prozesse als Einfache Systeme**

Um etwas mehr Intuition zu Einfachen Systemen zu entwickeln und die Mächtigkeit Einfacher Systeme zu erkennen, wenden wir die Definition des Einfachen Systems auf CCS-Prozesse an.

Aus Kapitel 4 wissen wir, dass jeder CCS-Prozess $P$ strukturell kongruent ist zu einer Normalform new $\vec{z}$ $(M_1|\ldots|M_m)$. CCS kennt keine Replikation, also bleibt als einzige weitere Bedingung für ein Einfaches CCS-Systyem: $M_i$ enthält keine parallele Komposition. Nun ist es leicht, zu sehen, dass alle Beispiele aus Kapitel 7 (Lottery, Job Shop, Scheduler, Buffer, Counter) Einfache CCS-Systeme sind.

$Zelle = teile.(Zelle \mid Zelle) + stirb.0$ ist kein Einfaches System.

## 10.2 Eindeutige Verwaltung

Jetzt wo wir wissen was ein Einfaches System ist, können wir auf unser eigentliches Anliegen zurückkommen. Wir wollen garantieren, dass nie mehr als eine der parallelen Komponenten eines Einfachen Systems $P$ die Form $\overline{x}\langle a\rangle.Q$ besitzt.

Diese Eigenschaft alleine ist nicht invariant unter Reduktion, denn

$$P = \overline{x}\langle a\rangle \mid \overline{z}\langle b\rangle.\overline{x}\langle c\rangle \mid z(d) \longrightarrow \overline{x}\langle a\rangle \mid \overline{x}\langle c\rangle \mid 0$$

In $P$ hat nur eine der parallelen Komponenten die Form $\overline{x}\langle a\rangle.Q$. Nach dem Reduktionsschritt existiert aber eine weitere Komponente die auf $x$ senden möchte. Wir müssen offenbar unsere Forderung verschärfen. Es liegt nahe, zu verlangen, dass höchstens eine der sequentiellen Komponenten $M_i$ überhaupt die Möglichkeit hat, auf $x$ zu senden (d.h. einen Aktionspräfix der Form $\overline{x}\langle y\rangle.Q$ enthält).

Dies ist leider immer noch nicht hinreichend, denn

$$P = \overline{x}\langle b\rangle \mid \overline{y}\langle x\rangle \mid y(b).\overline{b}\langle c\rangle \rightarrow \overline{x}\langle b\rangle \mid \overline{x}\langle c\rangle$$

$P$ erfüllt offenbar die syntaktische Forderung, dass nur eine parallele Komponente einen Aktionspräfix der Form $\overline{x}\langle a\rangle$ enthält. Allerdings existiert ein weiterer Prozess, der $x$ über $y$ versendet. Es gibt also zwei Prozesse die "Umgang" mit $x$ haben. Es leuchtet ein, dass nur eine der parallelen Komponenten von $P$ ein "Recht" auf $x$ haben sollte.

**Definition**

Ein Prozess $P$ *verwaltet* einen Kanal $x$, wenn er einen Aktionspräfix der Form $\overline{x}\langle a\rangle$ oder $\overline{y}\langle x\rangle$ enthält.

**Definition: Eindeutige Verwaltung**

Ein Einfaches System $P \equiv \text{new } \vec{z}\ (M_1|\ldots|M_m|N_1|\ldots|N_n)$ *verwaltet* einen Kanal $x$ *eindeutig*, wenn höchstens eines der $M_i$ und keines der $N_j$ den Kanal $x$ verwaltet.

Obwohl unsere Forderungen immer stärker geworden sind, haben wir immer noch keine Invarianz unter Reduktion erreicht:

$$P_1 = \overline{y}\langle x\rangle.\overline{x}\langle b\rangle \mid y(a).\overline{a}\langle c\rangle \rightarrow \overline{x}\langle b\rangle \mid \overline{x}\langle c\rangle$$

$P_1$ verwaltet $x$ eindeutig. Die erste parallele Komponente von $P_1$ versendet den Namen $x$ und ermöglicht so der zweiten Komponente, $x$ zu verwalten. Jedoch verwaltet die erste Komponente $x$ weiterhin und so entsteht ein System in dem $x$ nicht mehr eindeutig verwaltet wird. Also müssen wir sicherstellen, dass eine Komponente die $x$ versendet den Kanal $x$ danach nicht mehr verwaltet.

**Definition: $x$-Vergesslichkeit**

Ein Prozess $P$ heisst *$x$-vergesslich*, wenn in jedem Subterm der Form $\overline{a}\langle x\rangle.Q$ der Prozess $Q$ den Kanal $x$ nicht verwaltet.

Anschaulich gesehen garantiert uns die $x$-Vergesslichkeit zusammen mit der eindeutigen Verwaltung von $x$, dass $P$ nach einem Reduktionsschritt $x$ immer noch eindeutig verwaltet.

Es wäre allerdings zu schön gewesen wenn $x$-Vergesslichkeit unter Reduktion erhalten bliebe:

$P_1 = \overline{y}\langle x\rangle.\overline{x}\langle b\rangle \mid y(a).\overline{a}\langle c\rangle$ ist nicht $x$-vergesslich.

$P_2 = \overline{u}\langle x\rangle \mid u(a).\overline{y}\langle a\rangle.\overline{a}\langle b\rangle \mid y(a).\overline{a}\langle c\rangle$ ist $x$-vergesslich, aber...

$P_2 \to P_1$

Offenbar wird $x$ von einem Prozess der Form $u(a).Q$ empfangen, wobei $Q$ nicht $a$-vergesslich ist. Daher erhalten wir einen Prozess $\{x/a\}Q$ der nicht $x$-vergesslich ist.

Wenn wir nun allgemein von jedem Subterm der Form $u(a).Q$ von $P$ die $a$-Vergesslichkeit fordern, ist es anschaulich klar, dass wir uns eine vorhandene $x$-Vergesslichkeit unter Reduktion erhalten. Die $x$-Vergesslichkeit erhält uns im Gegenzug die eindeutige Verwaltung welche unsere ursprüngliche Forderung einschliesst.

**Satz**

Sei $P$ ein Einfaches System mit den folgenden Eigenschaften:

$P$ verwaltet $x$ eindeutig

$P$ ist $x$-vergesslich

für jeden Subterm der Form $z(y).Q$ von $P$ ist $Q$ $y$-vergesslich

Es gelte $P \to P'$. Dann gilt: $P'$ ist einfach und erfüllt ebenfalls die drei Bedingungen.