Kommunikation und parallele Prozesse

Mitschrift der Vorlesung von Sergey Goncharov im Wintersemester 2016
von Christian Bay, Frederik Haselmeier und Julian Jakob,
überarbeitet von Sergey Goncharov und Lutz Schröder

Inhaltsverzeichnis

1 Prozessalgebra **3**
1.1 Inhalt der Vorlesung . . . 3
1.2 Literatur . . . 3
1.3 CCS: Motivation . . . 4
1.4 CCS: Konstrukte . . . 5
1.5 Labelled Transition Systems . . . 8
1.6 Syntax von CCS . . . 9
1.7 Semantik von CCS . . . 10
1.8 Value-passing CCS . . . 11
1.9 Ableitungen . . . 12

2 Verhaltensäquivalenz **13**
2.1 Trace-Äquivalenz . . . 14
2.2 Bisimilarität und Bisimulation . . . 15
2.3 Schwache Bisimilarität . . . 21
2.4 Bisimulationsspiele . . . 27
2.5 Schwache Bisimulationsspiele . . . 30
2.6 Bisimulation als Fixpunkt . . . 30
2.7 Der Paige-Tarjan-Algorithmus . . . 34

3 Hennessy-Milner-Logik **41**

4 Der modale μ-Kalkül **45**
4.1 Der (multi-)modale μ-Kalkül . . . 47
4.2 Syntax . . . 47
4.3 Semantik . . . 47
4.4 Temporale Spezifikation im μ-Kalkül . . . 50

5 Erfüllltheitsspiele **52**

6 Der π-Kalkül **55**
6.1 Syntax 56
6.2 Semantik 57
6.3 Beispiele 58

1 Prozessalgebra

Bekannte Prozessalgebren:

- *Calculus of Communicating Systems (CCS)* (Milner)
- *Communicating Sequential Processes (CSP)* (Hoare)
- *Algebra of Communicating Processes (ACP)* (Bergstra/Klop)

Berührungspunkte mit Automatentheorie.

1.1 Inhalt der Vorlesung

- CCS
- Labelled Transition System (LTS): Semantik für *CCS*.
- Hennessy/Millner Logik (HML): Einfache Spezifikationslogik für LTS
- Bisimilarität: Ununterscheidbarkeit von Prozessen
- μ-Kalkül: ausdrucksstarke Erweiterung der HML, Spezifikation temporaler Eigenschaften ('immer', 'letztendlich' etc.)
- π-Kalkül: Erweiterung von *CCS* um dynamische Konfiguration der Prozesskommunikation – Erzeugen und Versenden von Kanalnamen.

1.2 Literatur

- Robin Milner, Communication and Concurrency, Prentice-Hall, 1989.
- Luca Aceto, Kim G. Larsen, An Introduction to Milner's CCS.
- Luca Aceto, Anna Ingólfsdottir, Kim Guldstrand Laresun and Jiri Srba: Reactive Systems, Cambridge University Press, 2007
- Davide Sangiorgi and David Walker, The Pi-Calculus - A Theory of Mobile Processes, Cambridge University Press, 2001.
- Joachim Parrow, An introduction to the π-calculus, in J. Bergstra, A. Ponse and SṠmolka, eds., Handbook of Process Algebra, Elsevier 2001, pp. 479-543.

Die Entwicklung des Materials in der Vorlesung folgt größtenteils Aceto et al. (2007); von dort stammen insbesondere das Getränkeautomatenbeispiel und viele der Übungsaufgaben. Das Kapitel über den π-Kalkül folgt Parrow (2001), insbesondere auch hinsichtlich des Beispielmaterials.

1.3 CCS: Motivation

Kommunikationsmodelle involvieren typischerweise ein Medium, das die Kommunikation trägt:

$$\text{Sender} \longrightarrow \text{Medium} \longrightarrow \text{Receiver}$$

Verbreitete Medientypen sind etwa

1. Medium = Ether: Dann gilt Sender kann immer schicken, Receiver kann immer empfangen, Reihenfolge ist nicht festgelegt.

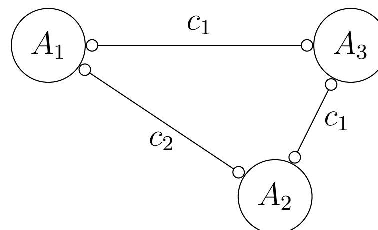
2. Medium = Puffer: Wie in Ether, aber mit definierter Reihenfolge.

3. Medium = Shared Memory: Gemeinsam genutzter Speicher, in dem Reihenfolge und Häufigkeit des Nachrichtenempfangs nicht spezifiziert ist.

4. Weitere Varianten: Begrenzter Speicher, spontaner Verlust von Nachrichten

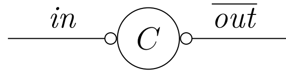
Milners Idee ist nun, dass man nur eine primitive Art von Kommunikation braucht, den sogenannten *Handshake*. Kompliziertere Medien wie Puffer sind dann eigene *Prozesse* oder *Agenten*.

Wir stellen die Kommunikationsstrukture gelegentlich durch *Interfacediagrammme* dar; z.B, besagt



dass Agent A_3 über den Kanal c_1 mit den Agenten A_1 und A_3 kommuniziert, und A_1 und A_2 untereinander über Kanal c_2.

Beispiel 1. Im Diagramm



wird demnach ein einzelner Prozess dargestellt, der über zwei Kanäle $\overline{out}$ und *in* mit einer unspezifizierten Umgebung kommuniziert. Ein mögliches Verhalten dieses Prozesses wäre, abwechselnd auf beiden Kanälen zu kommunizieren, was man in CCS folgendermaßen notieren würde:

$$\begin{aligned} C &= \mathrm{in}(x).C'(x) \\ C'(x) &= \overline{\mathrm{out}}(x).C; \end{aligned}$$

äquivalent, als einzelne Gleichung:

$$C = \mathrm{in}(x).\,\overline{\mathrm{out}}(x).C$$

1.4 CCS: Konstrukte

Wir führen die syntaktischen Konstrukte von CCS ein, mit zunächst nur informeller Semantik (die formale Semantik folgt später).

1. Deadlock: $\emptyset$ (Keine Aktion ist möglich).

2. Aktionen: Ports bzw. Kanäle, über die Prozesse miteinander kommunzieren können. Formal parametrisieren wir die Syntax über eine Menge $\mathcal{A}$ von *Aktionen*, z.B.

   $$\mathcal{A} \ni \{\mathrm{pushButton}, \mathrm{tick}, \dots\}$$

   Aktionen werden syntaktisch als Präfixe verwendet:

   $$a.P \qquad (a \in \mathcal{A}, P \text{ Prozess})$$

   Informell: "Führe a durch, verhalte dich dann wie P."

   Beispiel 2. Modell eines Streichholzes:

   $$\begin{aligned} \mathrm{BadMatch} &= \mathrm{Strike}.\,\emptyset \\ \mathrm{GoodMatch} &= \mathrm{Strike}.\,\mathrm{Burn}.\,\emptyset \end{aligned}$$

3. Rekursion/Prozessdefinition: Wir unterstellen weiterhin eine Menge K von *Prozessnamen.* Eine *(rekursive) Definiton* hat dann die Form

   $$A = P \qquad (A \in K, P \text{ Prozess})$$

Beispiele haben wir oben bereits gesehen, hier noch eins:

$$Clock = tick.\,Clock$$

Intuitiv impliziert dies übrigens

$$Clock = \underbrace{tick.\,tick.\,\ldots.\,tick}_{n-mal}.\,Clock$$

– wir werden das später formal fassen.

4. (Nichtdeterministische) Auswahl

$$P + Q \qquad\qquad (P, Q \text{ Prozesse})$$

Informelle Semantik:

"Wähle einen möglichen Schritt in P oder in Q, und verhalte dich dann weiter wie P bzw Q."

Beispiel 3.

$$\begin{aligned} B_0 &= in(x).B_1(x) \\ B_1(x) &= in(y).B_2(y,x) + \overline{out}(x).B_0 \\ B_2(x,y) &= \overline{out}(y).B_1(x) \end{aligned}$$

Beispiel 4. Ein eventuell nicht funktionierendes Streichholz lässt sich naiv auf eine der folgenden Arten modellieren:

$$\begin{aligned} Match &= strike.\,\emptyset + strike.\,burn.\,\emptyset \\ Match' &= strike.\,(\emptyset + burn.\,\emptyset) \end{aligned}$$

Dies wirft die Frage auf, ob diese beiden Prozesse verhaltensgleich sind. Die Antwort ist **Nein**. Die oben notierte informelle Semantik besagt, dass in $\emptyset + burn.\,\emptyset$ ein möglicher Schritt aus entweder $\emptyset$) oder $burn.\,\emptyset$ ausgewählt wird. Der Prozess $\emptyset$ hat aber keine Schritte, daher wird immer $burn.\,\emptyset$ ausgewählt. Damit gilt in Wirklichkeit also

$$Match' = strike.\,(\emptyset + burn.\,\emptyset) = strike.\,burn.\,\emptyset = GoodMatch$$

Dagegen wird in *Match* zwischen zwei nicht-blockierten Prozessen ausgewählt, beide Zweige tragen also zum Verhalten bei, so dass ein Fehlschlagen tatsächlich möglich ist.

Beispiel 5. Eine Uhr, die stehen bleiben kann, wäre demnach wie folgt zu formulieren:

$$\mathit{Clock} = \mathit{tick}.\,\mathit{Clock} + \mathit{tick}.\,\emptyset$$

5. Parallelkomposition:

$$P \mid Q \qquad\qquad (P, Q \in \text{Prozess})$$

Informelle Semantik: "Führe P und Q nebenläufig aus per Interleaving, oder synchronisiert".

- Interleaving:

  Beispiel 6.

  $$\mathit{Clock1} = \mathit{tick1}.\,\mathit{Clock1}$$
  $$\mathit{Clock2} = \mathit{tick2}.\,\mathit{Clock2}$$

  Welche Aktionsfolgen hat $Clock1 \mid Clock2$?

- Synchronisation:

  Zu jedem $a \in \mathcal{A}$ gibt es eine *duale Aktion* (oder *Co-Aktion*) $\overline{a} \in \mathcal{A}$, also $\overline{\cdot} : \mathcal{A} \to \mathcal{A}$. Diese Abbildung ist eine *Involution*, d.h. es gilt $\overline{\overline{a}} = a$. (Es folgt sofort, dass $\overline{\cdot}$ injektiv ist; wie?)

  Die übliche Lesart ist, dass a als Input und $\overline{a}$ als entsprechender Output verstanden wird. Über gleichzeitige Ausführung von a und $\overline{a}$ können parallele Prozesse *synchronisieren*; nach außen sind die Aktionen dann nicht sichtbar. Hierzu wird formal eine gesonderte *stumme Aktion* (*silent action*) τ eingeführt.

6. Restriktion:

$$P \setminus L \qquad (L \subseteq \mathcal{A})$$

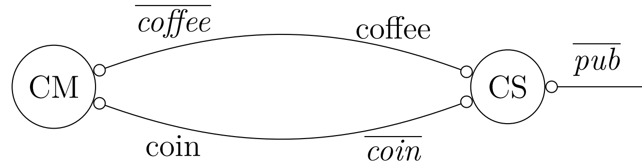
Informelle Semantik: "Verhalte dich wie P, aber ohne sichtbare Aktionen aus L." Der Effekt hiervon ist, dass die Aktionen in L nur für interne Synchronisation in P verwendet werden können.

Beispiel 7. Modellierung eines kaffeetrinkenden Informatikers:

$$\mathit{CoffeeMachine} = \mathit{coin}.\,\overline{\mathit{coffee}}.\,\mathit{CoffeeMachine}$$
$$\mathit{ComputerScientist} = \overline{\mathit{pub}}.\,\overline{\mathit{coin}}.\,\mathit{coffee}.\,\mathit{ComputerScientist}$$

$$\mathit{SmUni} = (\mathit{CM} \mid \mathit{CS}) \setminus \mathit{coin} \setminus \mathit{coffee}$$

Dann stehen *coin* und *coffee* als Kanäle nicht mehr für die Kommunikation mit der Außenwelt zur Verfügung, wir erhalten also folgendes Kommunikationsdiagramm:



7. Umbenennung: Sei $f : Act \to Act$ eine Funktion. Dann ist

$$P[f]$$

der Prozess, den man aus P gewinnt, indem man Aktionen gemäß f umbenennt.

Beispiel 8. Seien $P = tick.P$ und $P' = P[tick \mapsto tack]$. Dann gilt effektiv (wiederum in einem später zu präzisierenden Sinn)

$$P' = tack.P'.$$

1.5 Labelled Transition Systems

Die Semantik von CCS wird in Begriffen von *Transitionssystemen* (*labelled transition systems, LTS*) beschrieben, die wir als nächstes definieren.

Definition 9. Ein LTS ist ein Tupel $(Q, \mathcal{A}, \{\xrightarrow{\alpha} | \alpha \in \mathcal{A}\}$, bestehend aus

- einer Menge $\mathcal{Q}$ von *Zuständen*;
- einer Menge $\mathcal{A}$ von *Aktionen* oder *Labeln* (dies wird bei uns meist die gleiche Menge wie in der CCS-Syntax sein, so dass wir der Einfachheit halber gleich denselben Bezeichner wählen);
- einer *Transitions-* oder *Zustandsübergangsrelation*

$$\to \subseteq \mathcal{Q} \times \mathcal{A} \times \mathcal{Q}.$$

Man schreibt

$$\begin{aligned} s \xrightarrow{\alpha} s', &\text{ wenn } (s, \alpha, s') \in \to; \\ s \xrightarrow{\alpha}, &\text{ wenn } s \xrightarrow{\alpha} s' \text{ für ein } s' \in \mathcal{Q}; \\ s \to, &\text{ wenn } s \xrightarrow{\alpha} \text{ für ein } \alpha \in \mathcal{A}, \end{aligned}$$

und entsprechend negiert ($s \not\to$ etc.). Ein LTS ist *endlich*, wenn $\mathcal{Q}$ endlich ist. Ein LTS ist *endlich verzweigend*, wenn für alle $s \in \mathcal{Q}$ die Menge $\{s' \in \mathcal{Q} \mid s \xrightarrow{\alpha} s', \alpha \in \mathcal{A}\}$ endlich ist.

Aus einem LTS $T = (\mathcal{Q}, \mathcal{A}, \{\xrightarrow{\alpha} | \alpha \in \mathcal{A}\})$ bilden wir ein LTS $T^* = (\mathcal{Q}, \mathcal{A}^*, \{\xrightarrow{w} | w \in \mathcal{A}^*\})$, in dem also die Aktionen Wörter über den ursprünglichen Aktionen sind, wie folgt induktiv:

- $s \xrightarrow{\epsilon} s$ stets ($\epsilon \in \mathcal{A}^*$ ist das leere Wort);
- $s \xrightarrow{aw} t$, wenn es ein $s' \in \mathcal{Q}$ gibt, sodass $s \xrightarrow{a} s'$ und $s' \xrightarrow{w} t$.

In Begriffen von Relationskomposition bedeutet die letzte Klausel, dass

$$\xrightarrow{aw} = \xrightarrow{w} \circ \xrightarrow{a} = \xrightarrow{a}; \xrightarrow{w},$$

wobei wir mit $\circ$ applikative Komposition und mit ; diagrammatische Komposition bezeichnen (die sich nur durch die Reihenfolge ihrer Argumente unterscheiden).

1.6 Syntax von CCS

Wir legen jetzt noch einmal die Syntax von CCS übersichtlich und vollständig formal fest. Wir fixieren also (erneut) eine abzählbare Menge $\mathcal{A}$ von *Aktionen* mit einer *stummen Aktion* $\tau \in \mathcal{A}$ und einer involutiven Operation $\bar{\cdot}$ (*Komplement*) auf $\mathcal{A} \setminus \{\tau\}$ sowie eine ebenfalls abzählbar unendlichen Menge $\mathcal{K}$ von *Prozessnamen*.

Die Menge $\mathsf{Proc}(\mathcal{K}_0)$ der *Prozesse* P, Q mit Namen aus $\mathcal{K}_0 \subseteq \mathcal{K}$ ist dann durch die Grammatik

$$P, Q ::= K \;\Big|\; \alpha.P \;\Big|\; \sum_{i \in I} P_i \;\Big|\; P \mid Q \;\Big|\; P[f] \;\Big|\; P \setminus L.$$

definiert; hierbei sind

- $K \in \mathcal{K}_0$ ein Prozessname;
- $\alpha \in \mathcal{A}$ eine Aktion, und $L \subseteq \mathcal{A} \setminus \{\tau\}$;
- I eine beliebige Indexmenge;
- $f : \mathcal{A} \setminus \{\tau\} \to \mathcal{A} \setminus \{\tau\}$ eine Umbenennungsfunktion mit $f(\bar{a}) = \overline{f(a)}$.

Eine *Spezifikation* besteht dann aus einer Teilmenge $\mathcal{K}_0 \subseteq \mathcal{K}$, einem CCS-Prozess $P \in \mathsf{Proc}(\mathcal{K}_0)$und für jedes $K \in \mathcal{K}_0$ einer rekursiven Gleichung $K = Q$ mit $Q \in \mathcal{Q}(\mathcal{K}_0)$.

Die Operatorpräzedenz lautet, von höherer zu niedrigerer Priorität,

- Restriktion $\setminus$ und Umbenennung [ ];

- Aktionspräfix;
- Parallelkomposition $|$;
- Auswahl $+$.

1.7 Semantik von CCS

Sei $\mathcal{S} = (P, K_1 = P_1, \ldots, K_n = P_n)$ eine CCS-Spezifikation (die Angabe der vorkommenden Prozessnamen lassen wir meist weg).

Die Semantik von $\mathcal{S}$ ist ein Zustand ein einem wie folgt induktiv definierten LTS:

- Zustände sind zunächst alle CCS-Prozesse;
- der ausgezeichnete Zustand hierin ist P;
- Menge der Aktionen ist $\mathcal{A}$;
- die Transitionen sind induktiv durch folgende Regeln definiert (die das Format der *Structural Operational Semantics (SOS)* einhalten).

$$(ACT)\,\frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad (COM3)\,\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$(SUM_j)\,\frac{P_j \xrightarrow{\alpha_j} Q}{\sum_{i \in I} P_i \xrightarrow{\alpha_j} Q} \qquad (RES)\,\frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L}\,(\alpha, \overline{\alpha} \notin L)$$

$$(COM1)\,\frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q} \qquad (REL)\,\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$$

$$(COM2)\,\frac{Q \xrightarrow{a} Q'}{P \mid Q \xrightarrow{a} P \mid Q'} \qquad (CON)\,\frac{P_i \xrightarrow{\alpha} P'}{K_i \xrightarrow{\alpha} P'}$$

Um nicht immer mit einem unendlich großen LTS zu hantieren, schränkt man dann abschließend noch auf den vom ausgezeichneten Prozess P aus erreichbaren Teil ein, d.h. auf diejenigen Zustände (Prozesse) Q, so dass $P \xrightarrow{w} Q$ für ein Wort $w \in \mathcal{A}^*$.

1.8 Value-passing CCS

Wir führen noch eine syntaktische Erweiterung von CCS ein, in der wir mittels der Aktionen noch Datenwerte transportieren können; wir nennen diese Erweiterung *value-passing CCS*. Man kann dies direkt auch in CCS kodieren, indem man für jeden möglichen Eingabewert eigene Aktionen und Prozessnamen einführt. So wird zum Beispiel

$$\begin{aligned} C &= in(x).\, C'(x) \\ C'(x) &= \overline{out}(x).\, C \end{aligned}$$

kodiert zu:

$$\begin{aligned} C &= in_n.\, C_n \\ C_n &= \overline{out}_n.\, C \qquad (n \in \mathbb{N}) \end{aligned}$$

Aus Gründen des Komforts definiert man dennoch gesonderte semantische Regeln für value-passing CCS. Man führt dabei wie oben angedeutet Parameter für Aktionen ein, die dann die Form $\alpha(x_1, \dots, x_n)$ haben und mit Werten aus einem vorgegebenen Wertebereich, z.B. den natürlichen Zahlen, belegt werden können (allgemeiner kann man noch Ausdrücke über einer algebraischen Signatur zulassen; dies lassen wir hier der Einfachheit halber weg). Prozess-Gleichungen haben dann die Form $K(x_1, \dots, x_n) = Q$ mit $FV(Q) \subseteq \{x_1, ..., x_n\}$ Hierbei bezeichnet $FV(Q)$ die Menge freien Variablen in Q, wobei man für ausgezeichnete *Eingabeaktionen* (angedeutet durch Abwesenheit von Querstrichen) Präfixe $a(x_1, \dots, x_n)$ als Bindungsoperationen auffasst. Formal wird $FV(Q)$ induktiv wie folgt definiert:

$$\begin{aligned} FV(K(x_1, ..., x_n)) &= \{x_1, \dots, x_n\} \\ FV(\tau.P) &= FV(P) \\ FV(a(x_1, ..., x_n).P) &= FV(P) \backslash \{x_1, ..., x_n\} \\ FV(\overline{a}(x_1, ..., x_n)) &= FV(P) \cup \{(x_1, \dots, x_n)\} \\ FV(\textstyle\sum_i P_i) &= \textstyle\bigcup_i FV(P_i) \\ FV(P \mid Q) &= FV(P) \cup FV(Q) \\ FV(P[f]) &= FV(P) \\ FV(P \backslash L) &= FV(P) \end{aligned}$$

(Man beachte die unterschiedliche Behandlung von Input-Aktionen a und Output-Aktionen $\bar{a}$.) In der Semantik sind die Zustände des relevanten LTS nun *Instanzen* der Prozessterme; eine Instanz entsteht aus einem Prozessterm, indem seine *freien* Variablen durch Werte substitutiert werden. Wir verlangen

in Prozessspezifikationen, dass der ausgezeicheten Prozess *geschlossen* ist, d.h. keine freien Variablen enthält (er ist dann selbst eine Instanz, also ein Zustand des LTS). Die Regeln für Präfix werden wie folgt modifiziert:

$$(ACT_1)\ \frac{}{\overline{a}(n_1,\ldots,n_k).P \xrightarrow{\overline{a}(n_1,\ldots,n_k)} P}$$

$$(ACT_2)\ \frac{}{a(x_1,\ldots,x_k).P \xrightarrow{a(n_1,\ldots,n_k)} P[n_1/x_1,\ldots,n_k/x_k]}.$$

Hierbei stehen $n_1, \ldots, n_k$ für Werte aus dem vorgegeben Bereich. Man beachte wiederum den Unterschied zwischen Ein- und Ausgaben: Eingaben werden gewissermaßen im Prozess gespeichert, d.h. an die entsprechenden Variablen gebunden, Ausgaben (natürlich) nicht.

1.9 Ableitungen

Definition 10 (α-Ableitung)**.** Sei α eine Aktion und P ein CCS-Prozess. Die α-Ableitung von P ist die Menge

$$D_\alpha(P) = \{P' \mid P \xrightarrow{\alpha} P'\}.$$

Beispiel 11.

$$D_{up}(up.(C \mid \emptyset)) = \{C \mid \emptyset\}$$

Satz 12. *Sei $(P, K_1 = P_1, \ldots, K_n = P_n)$ eine CCS-Spezifikation. Dann gelten folgende Gleichungen:*

$$\begin{aligned}
D_\alpha(K_i) &= D_\alpha(P_i) \\
D_\alpha(\alpha.P) &= \{P\}, \quad D_\alpha(\beta.P) = \{\,\} \quad \textit{für } \alpha \neq \beta \\
D_\alpha(\textstyle\sum_i Q_i) &= \textstyle\bigcup_i D_\alpha(Q_i) \\
D_\alpha(P \mid Q) &= \left\{P' \mid Q \;\middle|\; P' \in D_\alpha(P)\right\} \cup \left\{P \mid Q' \;\middle|\; Q' \in D_\alpha(Q)\right\} \quad \textit{für } \alpha \neq \tau \\
D_\tau(P \mid Q) &= \left\{P' \mid Q \;\middle|\; P' \in D_\tau(P)\right\} \cup \left\{P \mid Q' \;\middle|\; Q' \in D_\tau(Q)\right\} \cup \\
&\quad \left\{P' \mid Q' \;\middle|\; \exists \alpha \in \mathscr{L} \cup \overline{\mathscr{L}}.P' \in D_\alpha(P), Q' \in D_{\bar{\alpha}}(Q)\right\} \\
D_\alpha(P \setminus L) &= \begin{cases} \{P' \setminus L \mid P' \in D_\alpha(P)\} & \textit{für } \alpha \notin \mathscr{L} \cup \overline{\mathscr{L}} \\ \emptyset & \textit{sonst} \end{cases} \\
D_\alpha(P[f]) &= \textstyle\bigcup_{f(\beta)=\alpha} D_\beta(P)[f]
\end{aligned}$$

Der Beweis der Gleichungen ergibt sich jeweils unmittelbar aus der Struktur der Semantikregeln.

Die Gleichung $P = a.P + P$ hat mehrere (sogar unendlich viele) Lösungen:

1. $P = a.a.a.\dots$

2. $P = a.a.a.\dots + b.b.b.\dots$

Definition 13 (Bewachte Terme (Guarded terms))**.** Eine Spezifikation $(P, K_1 = P_1, \dots, K_n = P_n)$ heißt *guarded* bzw. *bewacht*, wenn alle Vorkommnisse von K_i in P_j ausschließlich in Untertermen der Form $\alpha.Q$ vorkommen.

Satz 14. *Für jede bewachte Spezifikation* $(P, K_1 = P_1, \dots, K_n = P_n)$ *terminieren die rekursive Gleichungen für die* $D_\alpha(P)$*; insbesondere ist ihre Lösung eindeutig.*

2 Verhaltensäquivalenz

Wir suchen nunmehr nach geeigneten Begriffen von *Verhaltensäquivalenz* von Prozessen. Solche Begriffe sollen zum einen unsere Intuition davon widerspiegeln, was es heißt, zwei Prozesse durch Beobachtung oder Interaktion unterscheiden zu können (oder eben nicht), und zum anderen gewisse Abschlusseigenschaften einhalten. Insbesondere sollte z.B. eine Verhaltensäquivalenz tatsächlich eine Äquivalenzrelation sein (also reflexiv, symmetrisch und transitiv), und zum anderen sollte sie nach Möglichkeit eine *Kongruenz* bezüglich der Prozessalgebraoperationen sein, d.h. ein Austauschen von Prozessen gegen verhaltensäquivalente Prozesse innehalb eines Prozessterms sollte wiederum verhaltensäquivalente Prozesse liefern. Wir führen im folgenden verschiedene Begriffe von Verhaltensäquivalenz ein und beurteilen sie vor diesem Hintergrund.

Wir erinnern vorab kurz an einige Standardbegriffe hinsichtlich Eigenschaften von Relationen:

Definition 15 (Äquivalenz, Präordnung, partielle Ordnung)**.** Eine Relation $R \subseteq X \times X$ heißt

- *reflexiv*, wenn $\forall x \in X. (x, x) \in R$;
- *symmetrisch*, wenn $\forall x, y \in X. (x, y) \in R \Rightarrow (y, x) \in R$;
- *transitiv*, wenn $\forall x, y, z \in X. (x, y) \in R \wedge (y, z) \in R \Rightarrow (x, z) \in R$;
- *antisymmetrisch*, wenn $\forall x, y \in X. (x, y) \in R \wedge (y, x) \in R \Rightarrow x = y$;
- eine *Präordnung*, wenn R reflexiv und transitiv ist;
- eine *Äquivalenz*, wenn R reflexiv, transitiv und symmetrisch ist;

- eine *(partielle) Ordnung*, wenn R reflexiv, transitiv und antisymmetrisch ist.

Definition 16 (Kongruenz)**.** Eine Äquivalenzrelation $R \subseteq X \times X$ ist eine *Kongruenz* bezüglich einer Operation $f : X^n \to X$, wenn mit $x_1 R x'_1, \ldots, x_n R x'_n$ stets auch $f(x_1, \ldots, x_n) R f(x'_1, \ldots, x'_n)$ gilt. Ferner ist R ein *Kongruenz* für eine gegebene Sprache, wie z.B. CCS, wenn R eine Kongruenz bezüglich aller Operationen der Sprache ist, äquivalenterweise wenn $\forall x, y \in X.(x, y) \in R \Rightarrow (C(x), C(y)) \in R$ für alle Kontexte $C(\cdot)$. (Wir erinnern daran, dass ein Kontext ein Term mit genau einem Freiplatz $(\cdot)$ ist).

2.1 Trace-Äquivalenz

Unser erster Begriff von Verhaltensäquivalenz ist rein beobachtend, d.h. orientiert sich nur daran, welche Aktionsfolgen ein Prozess ausführen kann:

Definition 17 (Spuräquivalenz (Trace Equivalence))**.** Sei $T = s(\mathcal{Q}, \mathcal{A}, \{\xrightarrow{\alpha} \mid \alpha \in \mathcal{A}\}$ ein LTS. Für $x \in \mathcal{Q}$ definieren wir einen *Trace* als die Sequenz von Transitionen $\alpha_1, \ldots, \alpha_k \in \mathcal{A}$ $(k \geq 0)$, so dass eine Folge

$$x = x_0 \xrightarrow{\alpha_1} x_1 \ldots \xrightarrow{\alpha_k} x_k$$

von Transitionen in T existiert. Mit $\mathit{Traces}(x)$ bezeichnen wir die Menge aller Traces von x; in früher eingeführter Notation:

$$\mathit{Traces}(x) = \{w \in \mathcal{A}^* \mid x \xrightarrow{w}\}.$$

Zustände x, y in (womöglich verschiedenen) LTS heißen *spuräquivalent* oder *trace-äquivalent*, wenn $\mathit{Traces}(x) = \mathit{Traces}(y)$.

Beispiel 18 (Trace-Äquivalenz)**.** Wir haben

$$\mathit{Traces}(a.\,(b.\,\emptyset + c.\,\emptyset)) = \{\epsilon, a, ab, ac\}$$
$$\mathit{Traces}(a.b.\,\emptyset + a.c.\,\emptyset) = \{\epsilon, a, ab, ac\},$$

die Prozesse $a.\,(b.\,\emptyset + c.\,\emptyset)$ und $a.b.\,\emptyset + a.c.\,\emptyset$ sind also trace-äquivalent. Das ist aber durchaus problematisch; z.B. betrachte man folgende sehr ähnliche Prozesse:

$$CM = \mathit{coin}.\,(\overline{\mathit{coffee}}.CM + \overline{\mathit{tea}}.CM)$$
$$CM' = \mathit{coin}.\,\overline{\mathit{coffee}}.CM' + \mathit{coin}.\,\overline{\mathit{tea}}.CM'$$
$$CA = \overline{\mathit{coin}}.\,\mathit{coffee}.CA$$

und bilde dann folgende zusammengesetzten Prozesse:

$$P_1 = (CA \mid CM) \setminus \{coin, coffee, tea\}$$
$$P_2 = (CA \mid CM') \setminus \{coin, coffee, tea\}$$

Nach Spuräquivalenz sind diese Prozesse gleich. Man denkt zunächst, dass es sich hier um ein Kongruenzproblem handelt, das durch die Interaktion mit CA entsteht; wir werden aber gleich sehen, dass das gerade nicht der Fall ist. Vielmehr ist das unterliegende Problem, dass Trace-Äquivalenz offenbar die Gleichung

$$a.\,(P + Q) \neq a.P + a.Q$$

erfüllt, die offenbar nicht immer wünschenswert ist.

Satz 19. *Trace-Äquivalenz ist eine Kongruenz für CCS.*

Beweis. Wir können eine Berechnungsvorschrift für $\mathit{Traces}(P)$ durch strukturelle Rekursion über Terme angeben; Details erarbeiten wir in der Veranstaltung gemeinsam. Warum folgt daraus die Kongruenzeigenschaft? □

2.2 Bisimilarität und Bisimulation

Wir führen nun eine *feinergranulare* (oder einfach *feinere*) Form von Verhaltensäquivalenz ein, die also *mehr* Prozesse unterscheidet und auf diesem Wege das im vorigen Abschnitt angedeutete Problem der Trace-Äquivalenz eliminiert. Dies ist der Begriff der *starken Bisimilarität.* (Es gibt zahlreiche weitere Äquivalenzen zwischen den beiden Begriffen; in den Übungen lernen wir noch *completed-Trace-Äquivalenz* oder *starke Trace-Äquivalenz* kennen.)

Definition 20 (Bisimulation/Bisimilarität)**.** Seien T_1 und T_2 LTS über demselben Alphabet $\mathcal{A}$ von Aktionen mit Zustandsmengen $\mathcal{Q}_1$ bzw. $\mathcal{Q}_2$. Eine binäre Relation $R \subseteq \mathcal{Q}_1 \times \mathcal{Q}_2$ heißt eine *(starke) Simulation*, wenn für alle $x \in \mathcal{Q}_1$, $y \in \mathcal{Q}_2$ mit xRy gilt:

> *Forth:* Für $x \xrightarrow{\alpha} x'$ existiert stets y' mit $y \xrightarrow{\alpha} y'$ und $x'Ry'$.

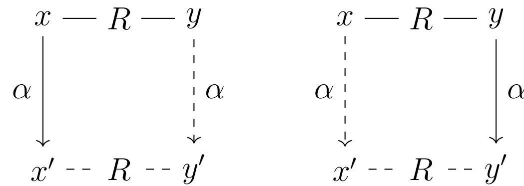
Ferner heißt R eine *(starke) Bismulation*, wenn sowohl R als auch seine Umkehrrelation

$$R^- = \{(y, x) \mid (x, y) \in R\} \subseteq Q_2 \times Q_1\}$$

Bisimulationen sind; die entsprechende Bedingung für R^- übersetzt sich dann in Begriffen von R statt R^-, d.h. für xRy, als

Back: Für $y \xrightarrow{\alpha} y'$ existiert stets x' mit $x \xrightarrow{\alpha} x'$ und $x' R y'$.

In diagrammatischer Darstellung:



(Starke) Bisimilarität ist die durch

$$x \sim y \iff \text{es gibt eine Bisimulation } R \text{ mit } x\,R\,y$$

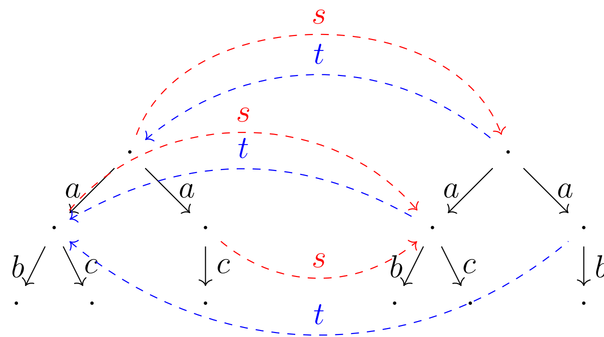
definierte Relation $\sim$ zwischen Zuständen von LTS (mit gemeinsamer Aktionsmenge).

Zusammenfassend haben wir

$$\frac{\text{Beweis}}{\text{Gültigkeit}} = \frac{\text{Bisimulation}}{\text{Bisimilarität}},$$

d.h. die Angabe einer Bisimulation zwischen zwei Zuständen beweist ihre Bisimilarität.

Beispiel 21. Bisimilarität ist stärker als gegenseitige Simulation:



Diese beiden LTS simulieren sich gegenseitig, sind aber nicht bisimilar.

Beispiel 22. Wir definieren ein LTS $(\mathcal{Q}, \mathcal{A}, \{\xrightarrow{\alpha} \mid \alpha \in \mathcal{A}\})$ durch

$$\begin{aligned}
\mathcal{Q} &= \{s, s_1, s_2, t_0, t_1\} \\
\mathcal{A} &= \{a, b\} \\
\xrightarrow{\alpha} &= \{(s, s_1), (s, s_2), (t, t_1)\} \\
\xrightarrow{\beta} &= \{(s_1, s_2), (s_2, s_2), (t_1, t_1)\} \\
R &= \{(s, t), (s_1, t_1), (s_2, t_1))\}
\end{aligned}$$

Wir haben dann die folgende Bisimulation R:

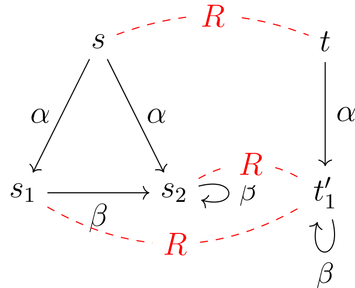


Abbildung 1: Bisimulation

Beispiel 23. Wir definieren ein LTS $(\mathcal{Q}, \mathcal{A}, \{\xrightarrow{\alpha} | \alpha \in \mathcal{A}\})$ durch

$$\begin{aligned} \mathcal{Q} &= \{s_i | i \geq 1\} \cup \{t\} \\ \mathcal{A} &= \{a\} \\ \xrightarrow{\alpha} &= \{(s_i, s_{i+1}) \mid i \geq 1\} \cup \{(t, t)\}. \end{aligned}$$

Dann haben wir $s \sim t$, weil $R = \{(s_i, t) \mid i \geq 1\}$ eine Bisimulation ist:

$$s_1 \xrightarrow{a} s_2 \xrightarrow{a} s_3 \xrightarrow{a} \dots$$

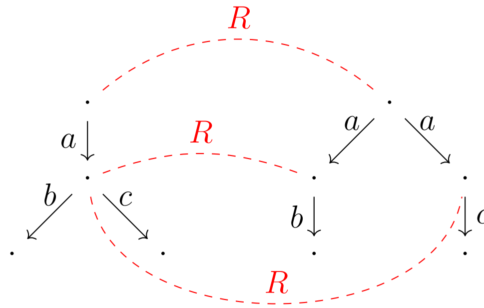
$$t \circlearrowleft a$$

Abbildung 2: Beispiel 2: Bisimulation

Beispiel 24. Gegenbeispiel:

$$a.\,(b.0 + c.0) \not\sim a.b.0 + a.c.0$$

Der Versuch, eine Bisimulation zwischen den beiden Prozessen zu konstruieren, scheitert wie folgt:



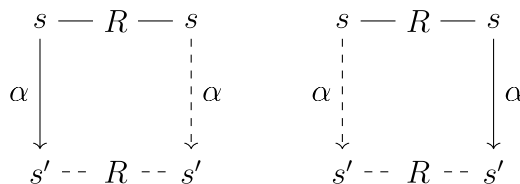
Satz 25. *Es gilt Folgendes:*

1. *Bisimilarität ist eine Äquivalenzrelation*
2. *Bisimilarität ist die größte Bisimulation*
3. *Für Zustände s, t gilt*

$$s \sim t \iff \forall s \xrightarrow{\alpha} s'.\, \exists t \xrightarrow{\alpha} t'.\, s' \sim t' \text{ und } \forall t \xrightarrow{\alpha} t'.\, \exists s \xrightarrow{\alpha} s'.\, s' \sim t'.$$

Beweis. 1. $\sim$ ist eine Äquivalenz:

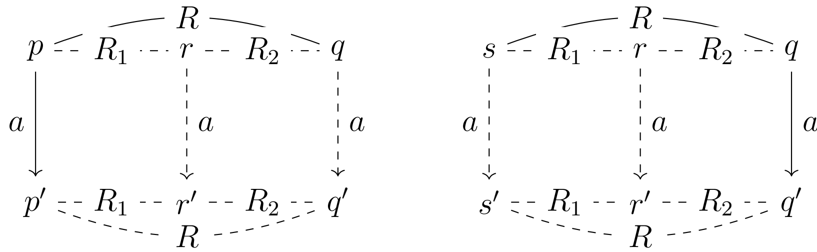
- Reflexivität: für alle s gilt $s \sim s$, weil $R = \{(s, s) \mid s \in Q\}$ eine Bisimulation ist:



- Symmetrie: Sei $s \sim t$; zu zeigen ist $t \sim s$. Nach Voraussetzung gibt es eine Bisimulation R mit $s\,R\,t$. Dann ist R^- wieder eine Bisimulation, denn weil R eine Bisimulation ist, sind sowohl R^- als auch $(R^-)^- = R$ Simulationen. Natürlich gilt tR^-s, und damit $t \sim s$.
- Transitivität: Sei $s \sim t$ und $t \sim u$; zu zeigen ist $s \sim u$. Nach Voraussetzung existieren Bisimulationen R_1 und R_2 mit sR_1t und tR_2u. Wir setzen

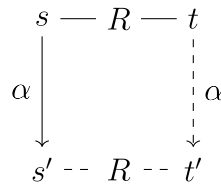
  $$R = R_2 \circ R_1 = \{(p, q) \mid \exists r.\ (p, r) \in R_1, (r, q) \in R_2\}.$$

  Dann gilt nach Voraussetzung $s\,R\,u$. Ferner ist R eine Bisimulation:



Das zeigt $s \sim t$.

2. Da Bisimilarität $\sim$ per Definition alle Bisimulationen enthält, ist nur noch zu zeigen, dass $\sim$ selbst eine Bisimulation ist. Seien also $s \sim t$ und $s \xrightarrow{\alpha} s'$. Nach Voraussetzung existiert dann eine Bisimulation R mit sRt. Dann haben wir



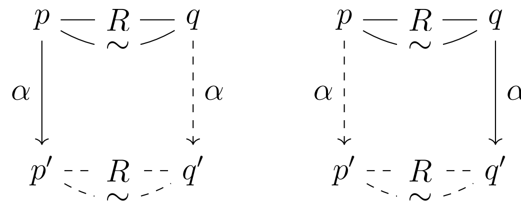
und somit auch $s' \sim t'$. Der Beweis Back-Bedingung ist symmetrisch.

3. Die Implikation $\Rightarrow$ gilt nach dem vorherigen Punkt; wir zeigen $\Leftarrow$. Dazu zeigen wir, dass unter den gegebenen Annahmen

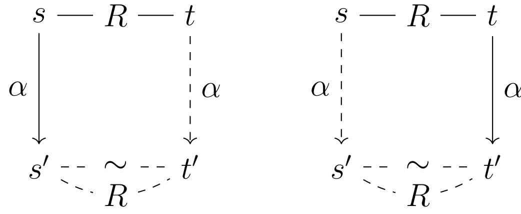
$$R = \sim \cup \{(s, t)\}$$

eine Bisimulation ist. Wir unterscheiden für pRq die beiden Fälle in der Definition von R:

1. Fall: $(p, q) \in \sim$, dann:



2. Fall: $(p, q) = (s, t)$; dann haben wir per Annahme



□

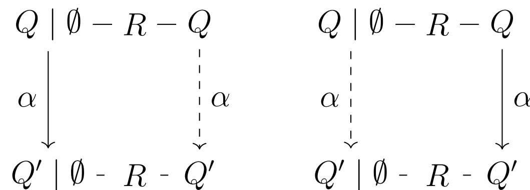
Satz 26. *Es gilt:*

$$\begin{aligned}
P + Q &\sim Q + P \\
P + P &\sim P \\
P + 0 &\sim P \\
P + (Q + R) &\sim (P + Q) + R \\
P \mid Q &\sim Q \mid P \\
P \mid \emptyset &\sim P \\
(P \mid Q) \mid R &\sim P \mid (Q \mid R)
\end{aligned}$$

Beweis. Wir greifen zwei Beispiele heraus:

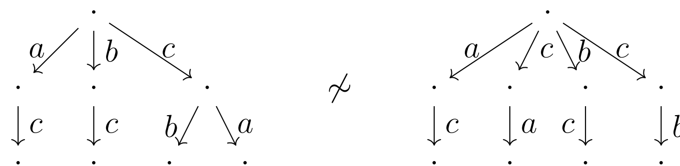
$P + Q = Q + P$: Dies ist eine leichte Anwendung von Theorem 25.3: Nach der Semantik von CCS haben $P + Q$ und $Q + P$ unter allen Aktionen die gleichen Nachfolger (nämlich die von P und die von Q), so dass wir in der Bedingung in Theorem 25.3 sogar $=$ statt nur $\sim$ haben. Die anderen Aussagen über $+$ zeigt man genauso.

$P \mid \emptyset = P$: Wir definieren $R = \{(P \mid \emptyset, P) \mid P \text{ ein CCS-Prozess}\}$ und zeigen, dass R eine Bisimulation ist:



□

Beispiel 27. Im allgemeinen gilt *nicht* $(P + Q) \mid S \sim (P \mid S) + (Q \mid S)$; z.B. $(a.\emptyset + b.\emptyset) \mid c.\emptyset \not\sim (a.\emptyset \mid c.\emptyset) + (b.\emptyset \mid c.\emptyset)$:



Satz 28. $\sim$ *ist eine Kongruenz für CCS*

Beweis. Wir beweisen den interessantesten Fall, die Parallelkomposition $|$: Sei

$$R = \{(P \mid S, Q \mid S) \mid \; P, Q, S \text{ CCS-Prozesse }, P \sim Q\};$$

Es reicht zu zeigen, dass R eine Bisimulation ist. Da R symmetrisch ist, genügt die Forth-Bedingung. Wir unterscheiden Fälle über die Nachfolger von $P \mid S$ gemäß der Semantikregeln:

(COM1): Sei $P \xrightarrow{\alpha} P'$; da $P \sim Q$, existiert Q' mit $Q \xrightarrow{\alpha} Q'$ und $P' \sim Q'$. Dann haben wir

$$\begin{array}{ccccc}
P \mid S & \text{——} & R & \text{——} & Q \mid S \\
\downarrow \alpha & & & & \alpha \downarrow \\
P' \mid S & \text{----} & R & \text{----} & Q' \mid S
\end{array}$$

(COM2): Sei $S \xrightarrow{\alpha} S'$; dann haben wir

$$\begin{array}{ccccc}
P \mid S & \text{——} & R & \text{——} & Q \mid S \\
\downarrow \alpha & & & & \alpha \downarrow \\
P \mid S' & \text{----} & R & \text{----} & Q \mid S'
\end{array}$$

(COM3): Sei $P \xrightarrow{\alpha} P'$ und $S \xrightarrow{\overline{\alpha}} S'$. Da $P \sim Q$, existiert Q' mit $Q \xrightarrow{a} Q'$ und $P' \sim Q'$; damit haben wir

$$\begin{array}{ccccc}
P \mid S & \text{——} & R & \text{——} & Q \mid S \\
\downarrow \tau & & & & \tau \downarrow \\
P \mid S' & \text{----} & R & \text{----} & Q' \mid S'
\end{array}$$

□

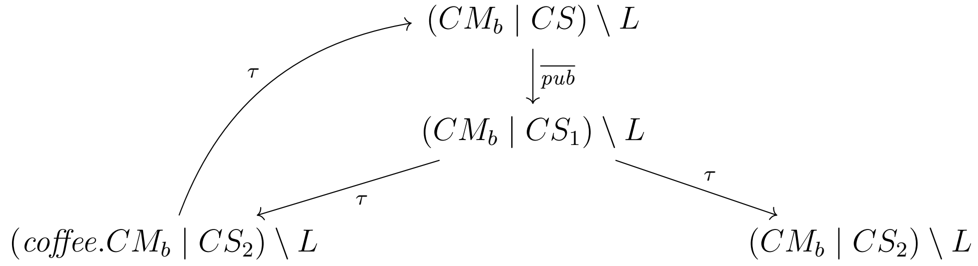
2.3 Schwache Bisimilarität

Die bisherigen Begriffe von Prozessäquivalenz berücksichtigen die besondere Rolle der stummen Aktion nicht; z.B. gilt natürlich

$$a.\varnothing \nsim a.\tau.\varnothing,$$

in der Tat sind die beiden Prozesse noch nicht einmal trace-äquivalent. Intuitiv sollte die Äquivalenz aber letztlich gelten, weil ja die zusätzliche τ-Aktion auf der rechten Seite gerade nicht beobachtbar sein soll. Ähnliches gilt unter den gehabten Definitionen von CS und CM für $(CM \mid CS) \setminus L$ mit $L = \{\mathit{coffee}, \mathit{coin}\}$ einerseits und $SmUni = \overline{pub}.SmUni$ andererseits. Die folgende alternative Kaffeemaschine CM_b wollen wir dagegen in diesem Kontext sicher weiterhin unterscheiden:

$$CM_b = coin.\,\overline{coffee}.CM_b + coin.CM_b$$



D.h. eine womögliche Idee, durch τ-Aktionen verbundene Zustände einfach zu identifizieren, tut offenbar nicht das Richtige. Stattdessen modifizieren wir die Transititionsrelation:

Definition 29 (Double-Arrow Konstruktion)**.** Sei $(\mathcal{Q}, \mathcal{A}, \{\xrightarrow{\alpha} \mid \alpha \in \mathcal{A}\})$ ein LTS und $\tau \in \mathcal{A}$. Die *schwache Transitionsrelation* $\stackrel{\alpha}{\Rightarrow} \subseteq \mathcal{Q} \times \mathcal{Q}$ ist wie folgt definiert: $P \stackrel{\alpha}{\Rightarrow} Q$ bedeutet, dass

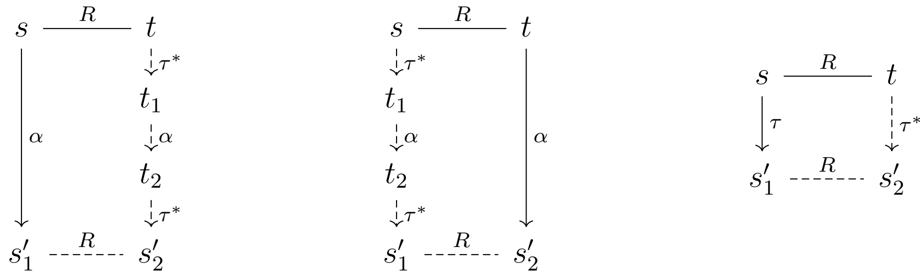
- $P \xrightarrow{\tau}^* P' \xrightarrow{\alpha} Q' \xrightarrow{\tau}^* Q$ mit geeignetem P', Q' wenn $\alpha \neq \tau$, und
- $P \xrightarrow{\tau}^* Q$, wenn $\alpha = \tau$.

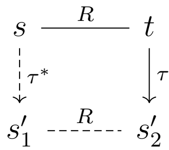
(dabei bezeichnet $\xrightarrow{\tau}^*$ den reflexiv-transitiven Abschluss von $\xrightarrow{\tau}$, d.h. $P \xrightarrow{\tau}^* Q$ genau dann, wenn es $w = \tau \dots \tau$ gibt, so dass $P \xrightarrow{w} Q$).

Definition 30 (Schwache Bisimulation/Bisimilarität)**.** Eine binäre Relation $R \in \mathcal{Q} \times \mathcal{Q}$ ist eine *schwache Bisimulation*, wenn aus sRt stets folgt, dass

- $s \xrightarrow{\alpha} s' \implies \exists t'.\, t \stackrel{\alpha}{\Rightarrow} t' \wedge s'Rt'$ (*forth*)
- $t \xrightarrow{\alpha} t' \implies \exists s'.\, s \stackrel{\alpha}{\Rightarrow} s' \wedge s'Rt'$ (*back*).

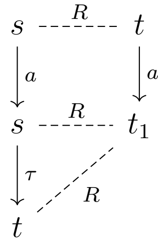
Graphisch:



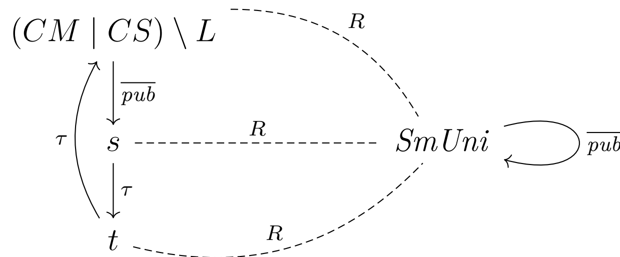


Wir schreiben $P \approx Q$, wenn es eine schwache Bisimulation R gibt, so dass $P\,R\,Q$. Die Relation $\approx$ heißt *schwache Bisimilarität*.

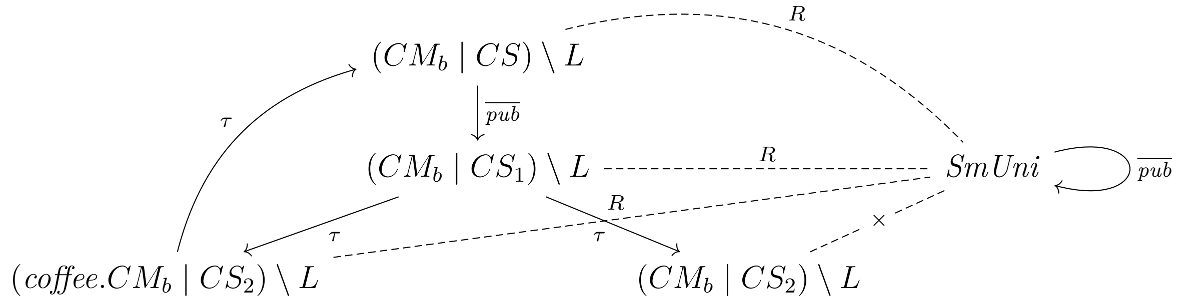
Beispiel 31. $a.\varnothing \not\sim a.\tau.\varnothing$, aber $a.\varnothing \approx a.\tau.\varnothing$:



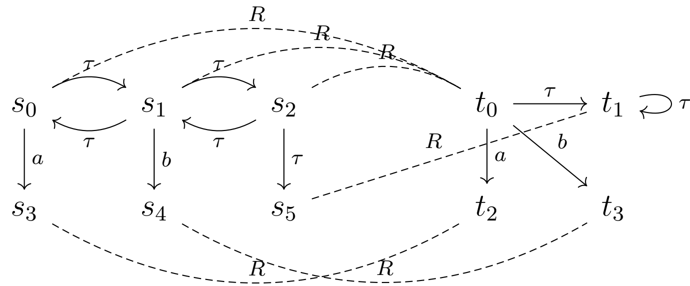
Beispiel 32. $(CM \mid CS) \setminus L \approx SmUni$ mit $SmUni = \overline{pub}.SmUni$:



Beispiel 33. $(CM_b \mid CS) \setminus L \not\approx SmUni$ mit $SmUni = \overline{pub}.SmUni$:

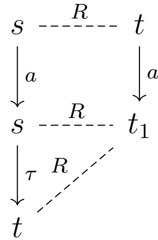


Beispiel 34. $s_0 \approx t_0$:

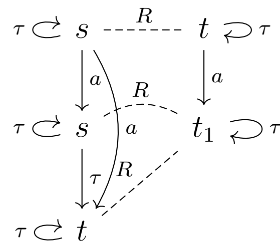


Satz 35. *Sei $T = (\mathcal{Q}, \mathcal{A}, \{\xrightarrow{\alpha} \mid \alpha \in \mathcal{A}\})$ ein LTS mit $\tau \in \mathcal{A}$. Sei $T' = (\mathcal{Q}, \mathcal{A}, \{\xRightarrow{\alpha} \mid \alpha \in \mathcal{A}\})$. Dann ist R eine schwache Bisimulation für T gdw. R eine starke Bisimulation für T' ist.*

Beispiel 36. T:



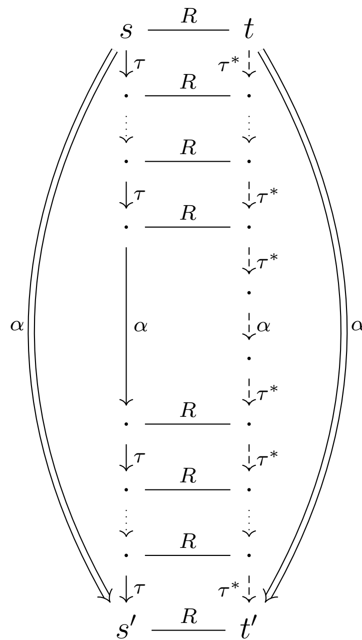
T':



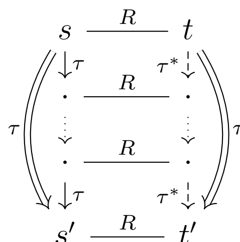
Satz 35. "$\Leftarrow$": Klar durch Vergleich der Definitionen, da aus $s \overset{\alpha}{\to} s'$ stets $s \overset{\alpha}{\Rightarrow} s'$ folgt.

"$\Rightarrow$": Sei sRt und $s \overset{\alpha}{\Rightarrow} s'$. Wir unterscheiden Fälle über α:

$\alpha \neq \tau$:



$\alpha = \tau$:

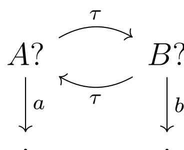


□

Beispiel 37. Man betrachte die folgende ‘Polling-Schleife’:

$$A? = a.\varnothing + \tau.B?$$
$$B? = b.\varnothing + \tau.A?$$

Als LTS:



Für

$$P = a.\varnothing + b.\varnothing$$

haben wir dann

$$A? \approx P,$$

obwohl $A?$ hat einen Livelock hat, P dagegen nicht. Diese Phänomen ist unter dem Namen *fair abstraction from divergence* bekannt.

Satz 38.

1. *$\approx$ ist eine Äquivalenz.*

2. *$\approx$ ist die größte schwache Bisimulation.*

3. *Für Zustände s, t gilt $s \approx t$ genau dann, wenn für alle $\alpha \in \mathcal{A}$ gilt:*

   - $s \xrightarrow{\alpha} s_1' \Rightarrow \exists s_2'.\, s_1' \approx s_2' \wedge t \stackrel{\hat{\alpha}}{\Rightarrow} s_2'$
   - $t \xrightarrow{\alpha} s_2' \Rightarrow \exists s_1'.\, s_1' \approx s_2' \wedge s \stackrel{\hat{\alpha}}{\Rightarrow} s_1'$

Der Beweis des obigen Satzes verläuft völlig analog wie für starke Bisimilarität. Anders als starke Bisimulation ist aber schwache Bisimulation, wie das folgende Beispiel zeigt, keine Kongruenz bezüglich +!

Beispiel 39.

$$\varnothing \approx \tau.\varnothing, \quad \text{aber}$$
$$a.\varnothing + \varnothing \not\approx a.\varnothing + \tau.\varnothing$$

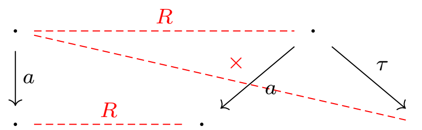


Abbildung 3:

Satz 40. $\approx$ *ist bezüglich aller Operationen außer + eine Kongruenz.*

Satz 41.

2.4 Bisimulationsspiele

Wir formalisieren als nächstes ein spielorientierte Sichtweise auf Bisimulation, d.h. wir geben ein Spiel an, in ein Spieler versucht, die Bisimilarität zweier Zustände zu zeigen, und einer, sie zu widerlegen; das Spiel ist so gebaut, dass der richtige Spieler gewinnt.

Die Spieler nennen wir

- S für *Spoiler* (auch *Attacker*, *Opponent*)
- D für *Duplicator* (auch *Defender*, *Proponent*, *Prover*)

Wie man anhand der Namen vielleicht errät, spielt D für und S gegen Bisimilarität.

Das Spiel wird in Runden gespielt; jede Runde besteht aus einem Zug von S und einem anschließenden Zug von D. Das Spiel dauert so lange, bis

einer Spieler nicht mehr ziehen kann, und kann insbesondere auch unendlich lange dauern.

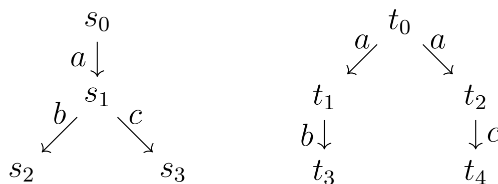
Seien nun $T_1 = (Q_1, \mathcal{A}, \to)$ und $T_2 = (Q_2, \mathcal{A}, \to)$ LTS, mit Anfangszuständen $s_0 \in A_1$, $t_0 \in Q_2$. Wir beschreiben das *Bisimulationsspiel* $\mathcal{G}(s_0, t_0)$. Die Spielpositionen nennen wir *Konfigurationen*; eine Konfiguration ist ein Paar (s, t) von Zuständen $s \in Q_1$, $t \in Q_2$. Die Spielregeln sind dann wie folgt beschrieben:

- *Anfangskonfiguration:* (s_0, t_0)
- *Züge* bei aktueller Konfiguration (s, t):
  - S wählt eines der beiden LTS, z.B. T_2, und darin eine Transition vom aktuellen Zustand, hier $t \xrightarrow{\alpha} t'$.
  - D wählt daraufhin im anderen LTS, im Beispiel also T_1, eine Transition $t \xrightarrow{\alpha} t'$.

  Die nach der Runde erreichte neue Konfiguration ist dann (s', t').
- *Gewinnbedingung:* Wer am Zug ist und keinen Zug hat (d.h. von eine Transition von einem Deadlock aus wählen müsste), verliert. Unendlich lange Spielverläufe gewinnt D.

Satz 42. *Gegeben Daten wie oben gilt $s_0 \sim t_0$ genau dann, wenn D in $\mathcal{G}(s_0, t_0)$ eine Gewinnstrategie besitzt.*

Beispiel 43. In der folgenden Situation gewinnt S:

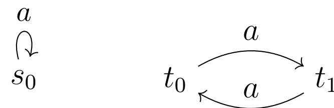


Eine Gewinnstrategie (von mehreren) für S ist wie folgt:

- Ziehe in der ersten Runde $t_0 \xrightarrow{a} t_1$. Die einzige Antwort von D ist dann $s_0 \xrightarrow{a} s_1$, so dass die Konfiguration (s_1, t_1) erreicht wird. (Es ist in der Tat egal, was S in der ersten Runde tut, durch diese Wahl vermeiden wir nur Fallunterscheidungen in der Spielanalyse.)
- Ziehe $s_1 \xrightarrow{c} s_3$. Spieler D kann nicht antworten und verliert.

Laut obigem Satz sind also s_0, t_0 nicht bisimilar, was wir ja auch schon wussten.

Beispiel 44. Im folgenden Fall gewinnt offenbar D:



– S zieht in jeder Runde eine a-Transition auf einer der beiden Seiten, die D dann mit der (jeweils einzigen) verfügbaren a-Transition auf der anderen Seite beantwortet. Da das Spiel damit nie abbricht, gewinnt D. Nach obigem Satz sind die Prozesse also bismilar.

Bemerkung 45. Es ist bei unendlichen Spielen im allgemeinen nicht klar, dass sie *determiniert* sind, d.h. in jedem Fall einer der Spieler eine Gewinnstrategie hat. Bisimulationsspiele gehören aber zu einer Klasse von Spielen, wo dies der Fall ist; ein Beweis führt für unsere gegenwärtigen Zwecke zu weit ab vom Thema. Wir werden diese Tatsache im Beweis von Satz 42 auch nicht verwenden.

Beweis (Satz 42). '$\Rightarrow$': Sei R eine Bisimulation mit $s_0 R t_0$. Wir verwenden R als Gewinnstrategie, genauer gesagt als *Invariante*, deren Durchsetzung D den Gewinn garantiert. Allgemein werden Gewinnstrategien in Spielen gerne durch Invarianten beschrieben. Eine Invariante I ist dabei eine Eigenschaft von Konfigurationen; damit sie tatsächlich eine Gewinnstrategie liefert, muss sie folgende Eigenschaften erfüllen:

- I muss für die *Anfangskonfiguration* gelten;
- I muss tatsächlich *durchsetzbar* sein, d.h. wenn I vor einer Runde erfüllt ist, muss D erzwingen können, dass I auch nach der Runde erfüllt ist;
- Spielverläufe, in denen durchgängig die Invariante gilt, müssen von D *gewonnen* werden.

Wir prüfen der Vollständigkeit halber alle Bedingungen:

- Für die Anfangskonfiguration gilt $s_0 R t_0$ per Annahme.
- Wenn vor einer Runde sRt gilt und S o.E. eine Transition $t \xrightarrow{a} t'$ in T_2 spielt, dann existiert per *Back*-Bedingung eine Transition $s \xrightarrow{a} s'$ mit $s'Rt'$. Spieler D spielt diese Transition und hat die Invariante damit durchgesetzt.

- Gegeben sei nun ein Spielverlauf, in dem für alle Konfigurationen sRt gilt. Wir unterscheiden zwei Fälle:
    - Das Spiel bricht ab, weil einer der Spieler nicht ziehen kann. Wir haben gesehen, dass D immer ziehen kann, wenn S ziehen kann; in diesem Fall kann also S nicht ziehen, und D gewinnt.
    - Das Spiel bricht nicht ab; dann gewinnt D.

'$\Leftarrow$': Wir zeigen, dass

$$R = \{(s,t) \in Q_1 \times Q_2 \mid D \text{ gewinnt } \mathcal{G}(s,t)\}$$

eine Bisimulation ist. Per Symmetrie der Spielregeln genügt es, die Forth-Bedingung zu zeigen. Sei dazu sRt und $s \xrightarrow{a} s'$. Dann kann S in der ersten Runde von $\mathcal{G}(s,t)$ eben diese Transition $s \xrightarrow{a} s'$ ziehen; sei $t \xrightarrow{a} t'$ die Antwort von D gemäß seiner Gewinnstrategie. Da dies eben eine Gewinnstrategie ist, gewinnt D das verbleibende Spiel $\mathcal{G}(s',t')$, d.h. es gilt wie verlangt $s'Rt'$. □

2.5 Schwache Bisimulationsspiele

Völlig analog erhält man ein Spiel für schwache Bisimilarität, das sich von dem für starke Bisimilarität nur dadurch unterscheidet, dass D auch mit einer schwachen Transition $\overset{a}{\Rightarrow}$ antworten kann. Man zeigt wie oben:

Satz 46. *Für Zustände s', t' in LTS gilt $s \approx t$ genau dann, wenn D eine Gewinnstrategie in $\mathcal{G}(s,t)$ hat.*

2.6 Bisimulation als Fixpunkt

Eine weitere besonders wichtige Perspektive auf Bisimilarität liefert die Fixpunkttheorie; insbesondere erhält man durch eine Fixpunktcharakterisierung von Bisimilarität mehr oder weniger sofort einen Algorithmus zur Berechnung der Bisimilaritätsrelation auf einem gegebenen LTS. Wir führen zunächste in die benötigten Aspekte der Fixpunkttheorie auf geordneten Mengen ein.

Definition 47. Sei $(X, \sqsubseteq)$ eine partiell geordnete Menge (Poset). Eine Funktion $f : X \to X$ ist *monoton*, wenn $\forall x, y \in X.\, x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$. Ein Element $x \in X$ ist *obere Schranke* einer Teilmenge $S \subseteq X$, wenn $\forall y \in S.\, y \sqsubseteq x$, und *Supremum* von S ($x = \bigsqcup S$), wenn x *kleinste* obere Schranke von S; Letzteres bedeutet im einzelnen, dass x erstens selbst eine obere Schranke von S ist, und zweitens kleiner oder gleich jeder oberen Schranke von S (insgesamt also $\forall y \in S.\, y \sqsubseteq z$ gdw. $x \sqsubseteq z$).

Die Begriffe *untere Schranke* und *Infimum* werden in der offensichtlichen Weise dual hierzu definiert; insbesondere ist x Infinum von S ($x = \bigsqcap S$), wenn x die größte untere Schranke von S ist.

Das Poset $(X, \sqsubseteq)$ ist ein *vollständiger Verband*, wenn jede Teilmenge von X ein Supremum besitzt.

In einem vollständigen Verband $(X, \sqsubseteq)$ ist $\bot = \bigsqcup \emptyset$ das kleinste Element und $\top = \bigsqcup X$ das größte Element.

Satz 48. *In einem vollständigen Verband hat jede Teilmenge ein Infimum.*

(Natürlich gilt auch die duale Aussage, dass die Existenz aller Infima die Existenz aller Suprema existiert, so dass also vollständige Verbände äquivalenterweise auch durch die Existenz aller Infima charakterisiert sind.) Insbesondere gilt also in einem vollständigen Verband auch $\bot = \bigsqcap X$ und $\top = \bigsqcup \emptyset$.

Beweis. Sei $(X, \sqsubseteq)$ ein vollständiger Verband und $S \subseteq X$. Setze

$$\bar{S} = \{s \in X \mid s \text{ ist untere Schranke für } S\}.$$

Wir behaupten, dass $\bigsqcup \bar{S}$ Infimum von S ist. Nach Konstruktion ist $\bigsqcup \bar{S}$ größer gleich allen unteren Schranken von S; zu zeigen ist also nur noch, dass $\bigsqcup \bar{S}$ selbst untere Schranke von S ist. Sei also $x \in S$. Dann ist x obere Schranke von $\bar{S}$: Wenn $y \in \bar{S}$ ist, dann ist y untere Schranke von S, insbesondere also $y \sqsubseteq x$. Da aber $\bigsqcup \bar{S}$ die kleinste obere Schranke von $\bar{S}$ ist, folgt wie verlangt $\bigsqcup \bar{S} \sqsubseteq x$. □

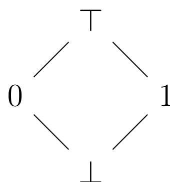
Beispiel 49. Beispiele von vollständigen Verbänden:

1. $([0, 1], \leq)$ ist ein vollständiger Verband.

2. $(\mathcal{P}X, \subseteq)$ ist ein vollständiger Verband, wobei $\mathcal{P}X = \{Y \mid Y \subseteq X\}$ die Potenzmenge von X ist.

3. $(\mathbb{N}, \leq)$ ist *kein* vollständiger Verband, da $\bigsqcup \mathbb{N}$ nicht existert. Dagegen ist $(\mathbb{N} \cup \{\infty\}, \leq)$ ein vollständiger Verband

4. Das durch das Hasse-Diagramm



beschriebene Poset ist ein vollständiger Verband.

Beispiel 50. Sei $(X, \sqsubseteq)$ ein vollständiger Verband; dann ist $(X, \sqsubseteq)^{op} = (X, \sqsubseteq^{-1})$ ebenfalls ein vollständiger Verband.

Definition 51. Sei $(D, \sqsubseteq)$ ein vollständiger Verband und $f : D \mapsto D$ eine monotone Funktion. Wir definieren die folgenden Mengen:

$$\begin{aligned} \mathit{Fix}(f) &= \{x \in D \mid fx = x\} && \text{(Fixpunkte von } f) \\ \mathit{Fix}^{\mu}(f) &= \{x \in D \mid fx \sqsubseteq x\} && \text{(Präfixpunkte von } f) \\ \mathit{Fix}^{\nu}(f) &= \{x \in D \mid x \sqsubseteq fx\} && \text{(Postfixpunkte von } f) \end{aligned}$$

Seien $\mu f = \bigsqcap \mathit{Fix}^{\mu}(f)$, $\nu f = \bigsqcup \mathit{Fix}^{\nu}(f)$.

Satz 52. *(Knaster-Tarski) Sei $(X, \sqsubseteq)$ ein vollständiger Verband und sei $f : X \mapsto X$ monoton. Dann gilt:*

1. *$\mu f \in \mathit{Fix}(f) \subseteq \mathit{Fix}^{\mu}(f)$,*
2. *$\nu f \in \mathit{Fix}(f) \subseteq \mathit{Fix}^{\nu}(f)$,*
3. *μf ist das kleinste Element von $\mathit{Fix}^{\mu}(f)$ (also kleinster (Prä-)Fixpunkt von f),*
4. *νf ist das größte Element von $\mathit{Fix}^{\nu}(f)$ (also größter (Post-)Fixpunkt von f).*

Beweis. Die Behauptungen 2. und 4. folgen per Dualität aus 1. und 3. Behauptung 3. folgt per Definition von μf sofort aus 1. Zu zeigen bleibt $\mu f \in \mathit{Fix}(f)$, d.h. $f(\mu f) = \mu f$. Wir zerlegen diese Gleichung in zwei Ungleichungen:

$f(\mu f) \sqsubseteq \mu f$: Es reicht zu zeigen, dass $f(\mu f)$ untere Schranke von $\mathit{Fix}^{\mu}(f)$ ist. Sei also $x \in \mathit{Fix}^{\mu}(f)$. Per Definition von $\mu(f)$ gilt dann $\mu(f) \sqsubseteq x$; per Monotonie von f folgt wie verlangt $f(\mu(f)) \sqsubseteq f(x) \sqsubseteq x$, wobei die letzte Ungleichung einfach die definierende Eigenschaft von x als Präfixpunkt von f ist.

$f(\mu f) \sqsubseteq \mu f$: Nach der schon bewiesenen umgekehrten Ungleichung und Monotonie von f gilt $f(f(\mu f)) \sqsubseteq f(\mu f)$, d.h. $f(\mu f) \in \mathit{Fix}^{\mu}(f)$. Nach Definition von $\mu(f)$ folgt $\mu f \sqsubseteq f(\mu f)$. □

Satz 53 (Fixpunktsatz von Kleene für endliche Verbände)**.** *Sei $(X, \sqsubseteq)$ ein endlicher (vollständiger) Verband und $f : X \mapsto X$ monoton. Dann existiert $n \in \mathbb{N}$, so dass $\mu f = f^n(\bot)$; dual existiert $m \in \mathbb{N}$, so dass $\nu f = f^m(\top)$. Genauer gesagt bilden die $f^i(\bot)$ eine aufsteigende Folge, die ab $i = n$ bei $f^n(\bot) = \mu(f)$ stationär wird (d.h. $f^k(\bot) = f^n(\bot)$ für alle $k \geq n$); dual bilden die $f^i(\top)$ eine absteigende Folge, die ab $i = m$ bei $f^m(\top) = \nu(f)$ stationär wird.*

Dabei definieren wir f^n wie üblich induktiv durch

$$f^0(x) = x$$
$$f^{i+1}(x) = f(f^i(x)).$$

(Im Satz steht das Wort „vollständig“ in Klammern, weil endliche Verbände immer vollständig sind; ein Verband ist dabei definiert als eine partielle Ordnung mit endlichen Infima und Suprema.)

Beweis. Per Dualität reicht es, die Behauptung für $\mu(f)$ zu beweisen. Man zeigt induktiv, dass die $f^i(\bot)$ eine aufsteigende Folge bilden: Der Induktionsanfang $f^0(\bot) \sqsubseteq f^1(\bot)$ folgt sofort aus der Eigenschaft von $f^0(\bot) = \bot$ als kleinstem Element von D. Im Induktionsschritt folgt $f^{i+1}(\bot) = f(f^i(\bot)) \sqsubseteq f(f^{i+1}(\bot)) = f^{i+2}(\bot)$ per Monotonie von f aus der Induktionsannahme $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$. Da X endlich ist, muss es daher ein n mit $f^n(\bot) = f^{n+1}(\bot)$ geben, und dann bleibt die Folge ab dieser Stelle stationär im Sinne des Satzes. Da $f^n(\bot)$ somit ein Fixpunkt von f ist, insbesondere also $\mu(f) \sqsubseteq f^n(\bot)$, reicht es, zu zeigen, dass $f^i(\bot) \sqsubseteq \mu(f)$ für alle i. Wir zeigen dies per Induktion über i. Der Induktionsanfang ist klar, da $f^0(\bot) = \bot$. Im Induktionsschritt erhalten wir aus der Induktionsannahme $f^i(\bot) \sqsubseteq \mu(f)$ per Monotonie von f

$$f^{i+1}(\bot) = f(f^i(\bot)) \sqsubseteq f(\mu(f)) = \mu(f).$$

□

Wir charakterisieren nun Bisimulationen als Postfixpunkte und (damit) Bisimilärität als größten (Post-)Fixpunkt eines monotonen Operators auf Relationen. Sei $T = (\mathcal{Q}, \mathcal{A}, \{\xrightarrow{\alpha} \mid \alpha \in \mathcal{A}\})$ ein LTS und $R \subseteq \mathcal{Q} \times \mathcal{Q}$ eine binäre Relation auf $\mathcal{Q}$. Wir definieren $\mathcal{F}(R) \subseteq \mathcal{Q} \times \mathcal{Q}$ wie folgt:
$(s, t) \in \mathcal{F}(R)$ und $s, t \in \mathcal{Q}$ genau dann, wenn

- $s \xrightarrow{\alpha} s'$ impliziert, dass t' existiert, so dass $t \xrightarrow{\alpha} t'$ und $(s', t') \in R$ (*Forth*);
- $t \xrightarrow{\alpha} t'$ impliziert, dass s' existiert, so dass $s \xrightarrow{\alpha} s'$ und $(s', t') \in R$ (*Back*).

Dann gilt:

- Eine Relation $R \subseteq \mathcal{Q} \times \mathcal{Q}$ ist genau dann eine Bisimulation, wenn $R \subseteq \mathcal{F}(R)$, d.h. wenn R Postfixpunkt von $\mathcal{F}$ ist; und somit
- $\sim = \bigsqcup\{R \in \mathcal{Q} \times \mathcal{Q} \mid R \subseteq \mathcal{F}(R)\} = \nu\mathcal{F}$

Aus dem Satz von Kleene ergibt sich nun unmittelbar ein Algorithmus zur Berechnung der Bisimilaritätsrelation auf einem gegebenen LTS $T = (\mathcal{Q}, \mathcal{A}, \rightarrow))$:

1. Intitialisiere R auf $R = \mathcal{Q} \times \mathcal{Q}$
2. Wenn $R = \mathcal{F}(R)$, terminiere und gib R zurück.
3. Setze $R := \mathcal{F}(R)$.
4. Gehe zu Schritt 2.

Was ist die Laufzeit dieses Algorithmus?

Wichtig bei der Analyse ist unter anderem folgende Feststellung:

Lemma 54. *Im obigen Algorithmus ist R zu jedem Zeitpunkt eine Äquivalenzrelation.*

Beweis. Wir müssen zeigen, dass das Funktional $\mathcal{F}$ Äquivalenzrelationen bewahrt. Sei also R eine Äquivalenzrelation; wir zeigen, dass $\mathcal{F}(R)$ eine Äquivalenzrelation ist.

Reflexivität: Sei $s \in \mathcal{Q}$. Zu zeigen ist $(s, s) \in \mathcal{F}(R)$. Sei also $s \xrightarrow{a} s'$; wir wählen auf der „anderen" Seite ebenfalls die Transition $s \xrightarrow{a} s'$, und da R eine Äquivalenzrelation ist, gilt wie verlangt $s'Rs'$. Die Back-Bedingung zeigt man symmetrisch.

Symmetrie: Sei $(s, t) \in \mathcal{F}(R)$; zu zeigen ist $(t, s) \in \mathcal{F}(R)$. Wir beweisen o.E. nur die Forth-Bedingung. Sei also $t \xrightarrow{a} t'$. Dann existiert gemäß der Back-Bedingung für $(s, t) \in \mathcal{F}(R)$ ein s' mit $s \xrightarrow{a} s'$ und $s'Rt'$; da R symmetrisch ist, gilt wie verlangt $t'Rs'$.

Transitivität: Seien $(s, t), (t, u) \in \mathcal{F}(R)$; zu zeigen ist $(s, u) \in \mathcal{F}(R)$. Wir zeigen o.E. nur die Forth-Bedingung. Sei also $s \xrightarrow{a} s'$. Da $(s, t) \in \mathcal{F}(R)$, existiert $t \xrightarrow{a} t'$, so dass $s'Rt'$. Da $(t, u) \in \mathcal{F}(R)$, existiert $u \xrightarrow{a} u'$, so dass $t'Ru'$. Da R transitiv ist, folgt wie verlangt $s'Ru'$. □

Dementsprechend heißen Algorithmen des obigen Typs *Partition-Refinement-Algorithmen.* Wie viele Iterationen benötigt also der obige Algorithmus schlimmstenfalls?

2.7 Der Paige-Tarjan-Algorithmus

Allgemein arbeitet man bei Partition-Refinement-Algorithmen statt mit Äquivalenzrelationen äquivalenterweise mit *Partitionen* $P = \{B_1, \dots, B_b\}$, d.h. disjunkten Zerlegungen $\mathcal{Q} = B_1 \cup \dots \cup B_n$ der Zustandsmenge in Äquivalenzklassen B_i, die man hier kurz als *Blöcke* bezeichnet. Der Einfachheit halber nehmen wir an, es gebe nur eine Aktion, die wir dann nicht mehr erwähnen. Wir schreiben für $S \subseteq \mathcal{Q}$ im folgenden

$$\to [S] = \{s' \mid \exists s \in S.\, s \to s'\}$$

und

$$\leftarrow [S] = \{s \mid \exists s' \in S.\, s \to s'\}$$

für die Menge der Nachfolger bzw. Vorgänger von Elementen von S. Wir nennen eine Partition P wie oben *stabil* bezüglich $S \subseteq \mathcal{Q}$, wenn für alle $B \in P$ entweder

$$B \subseteq \leftarrow[S] \quad \text{oder} \quad B \cap \leftarrow[S] = \emptyset$$

gilt, wenn also entweder alle oder keine Zustände in B Transitionen nach S haben. Eine Partition P heißt *stabil* bezüglich einer Partition R, wenn P stabil bezüglich aller Blöcke von R ist, und *stabil* schlechthin, wenn P stabil bezüglich P ist. Man überzeugt sich leicht, dass eine Partition genau dann stabil ist, wenn die zugehörige Äquivalenzrelation eine Bisimulation ist. Man arbeitet bei Partition-Refinement-Algorithmen typischerweise mit einer *Initialpartition* P_0 und sucht dann die *gröbste stabile Verfeinerung* der Initialpartition, wobei eine Partition P *feiner* ist als eine Partition R, wenn jeder Block von R Vereinigung von Blöcken von P ist (was in Begriffen von Äquivalenzrelationen bedeutet, dass die von P repräsentierte Äquivalenzrelation in der von R repräsentierten enthalten ist). Wir interessieren uns hier nur für die Berechnung von Bisimilarität und unterstellen deswegen, dass P_0 stets aus (maximal) zwei Blöcken besteht, den Deadlocks und allen anderen Zuständen (einer dieser Blöcke kann natürlich leer sein und entfällt dann).

Die Definition von Stabilität motiviert eine Funktion *split*: Gegeben eine Partition P und $S \subseteq \mathcal{Q}$ ist $\mathit{split}(P, S)$ die Partition, die man aus P erhält, indem man jeden Block $B \in P$ in $B' = B \cap \leftarrow[S]$ und $B'' = B \setminus B'$ aufspaltet. Wir nennen S einen *Spalter* von P, wenn $\mathit{split}(P, S) \neq P$, wenn also P nicht stabil bezüglich S ist. In diesen Begriffen besteht eine Berechnung von $\mathcal{F}(R)$ im wesentlichen in einer sukzessiven Anwendung von *split* auf alle Blöcke von R.

Man kann den Algorithmus dann liberalisieren, indem man *split* wiederholt auf beliebige Splitter anwendet. Wenn man als Splitter stets Blöcke der aktuellen Partition verwendet, erreicht man so bei geschickter Datenhaltung und guter Implementierung von *split* eine Laufzeit von $\mathcal{O}(m)$ pro Anwendung von *split* und damit eine Gesamtlaufzeit von $\mathcal{O}(mn)$, wobei m die Anzahl Transitionen und n die Anzahl Zustände des Systems ist. Der Paige-Tarjan-Algorithmus geht noch etwas subtiler vor und erreicht damit eine Gesamtlaufzeit $\mathcal{O}(m \log n)$.

Der Algorithmus arbeitet mit *zwei* Partitionen P und R, und hält die folgende Invariante aufrecht:

P ist feiner als R und stabil bezüglich allen Blöcken von R.

Ein Block $B \in R$ heißt *einfach*, wenn $B \in P$, und sonst *zusammengesetzt*. Der Algorithmus verläuft im Groben wie folgt:

1. Initialisiere P auf P_0 und R auf $\{\mathcal{Q}\}$.
2. Wiederhole folgenden Schritt *Refine* so lange, bis $P = R$:
   (a) Wähle einen zusammengesetzten Block $S \in R$.
   (b) Wähle $B \in P$ mit $B \subseteq S$ und $|B| \leq |S|/2$.
   (c) Ersetze in R S durch B und $S \setminus B$.
   (d) Setze $P := \mathit{split}(S \setminus B, \mathit{split}(B, P))$.

Im letzten Schritt von *Refine* werden effektiv alle Blöcke $D \in P$ in drei Teile (sofern jeweils nichtleer) unterteilt, nämlich

$$D \cap \leftarrow[B] \cap \leftarrow[S \setminus B] \qquad D \cap \leftarrow[B] \setminus \leftarrow[S \setminus B] \qquad D \setminus \leftarrow[B]$$

(der dritte Teil würde nach Definition von *split* in die Mengen $(D \setminus \leftarrow[B]) \cap \leftarrow[S \setminus B]$ und $(D \setminus \leftarrow[B]) \setminus \leftarrow[S \setminus B]$ zerfallen, von denen aber, weil P stabil bezüglich S ist, entweder erste leer ist, nämlich wenn $D \cap \leftarrow[S] = \emptyset$, oder die zweite, nämlich wenn $D \subseteq \leftarrow[S]$). Man beachte, dass die mittlere der drei Mengen gleich

$$D \cap (\leftarrow[B] \setminus \leftarrow[S \setminus B])$$

ist.

Wir zeigen zunächst, dass der skizzierte Algorithmus korrekt ist:

Lemma 55. *Der Paige-Tarjan-Algorithmus berechnet die gröbste stabile Verfeinerung von P_0.*

Beweis. Wir zeigen, dass

P und R vergröbern Bisimilarität

eine (weitere) Invariante des Algorithmus ist. Dies gilt nach Initialisierung für R trivialerweise und für P nach unserer festen Wahl von P_0; wir müssen zeigen, dass *Refine* die Invariante erhält. Für R ist dies klar, da bei Aktualisierung von R stets nur Zustände getrennt werden, die in P schon getrennt sind. Für P rechnen wir wie folgt: Sei $s \sim t$, und sei $D \in P$ der Block mit $s, t \in D$; wir müssen zeigen, dass s und t nach Aufspaltung von D im gleichen Block verbleiben. Dazu reicht es zu zeigen, dass für jeden Block $B \in P$ gilt

$$s \in \leftarrow[B] \iff t \in \leftarrow[B].$$

Per Symmetrie reicht eine Implikation. Sei also $s \in \leftarrow[B]$, d.h. wir haben $s \to s'$ mit $s' \in B$. Wir erhalten t' mit

$$\begin{array}{ccc} s & \overset{\sim}{\text{———}} & t \\ \downarrow & & \downarrow \\ s' & \overset{\sim}{\text{———}} & t' \end{array}$$

Da P Bisimilarität vergröbert, folgt aus $s' \in B$, dass auch $t' \in B$ und damit wie verlangt $t \in \leftarrow[B]$.

Ferner ist klar, dass der Algorithmus tatsächlich die schon behauptete Invariante

P ist stabil bezüglich R

einhält: Nach Initialisierung gilt die Invariante nach unserer festen Wahl von P_0, und die beiden *split*-Anwendungen am Ende eines *Refine*-Schritts stellen gerade sicher, dass P stabil bezüglich des neuen R wird.

Da nach Terminierung P stabil ist (nämlich stabil bezüglich R, was jetzt aber gleich P ist), induziert P eine Bisimulation, verfeinert also Bisimilarität. Nach der ersten Invarianten vergröbert P außerdem Bisimilarität, fällt also insgesamt gerade mit Bisimilarität zusammen. □

Die Herausforderung liegt dann in den Implementierungsdetails von *Refine*. Der Algorithmus verwendet die folgenden Datenstrukturen:

- *Zustände*
- *Transitionen*
- *Blöcke* in R
- *Blöcke* in P
- *Zusammengesetzte Blöcke*
- *Zähler* $\mathit{count}(s, S) = |{\to}[\{s\}] \cap S|$ für $s \in \mathcal{Q}$, $S \in R$ (d.h. $\mathit{count}(s, S)$ gibt die Anzahl in S liegender Nachfolger von s an).

Diese Daten sind wie folgt verlinkt:

- Transitionen $s \to s'$ auf s;
- Zustände s' auf $\leftarrow[\{s'\}]$ als Liste;
- Block $B \in P$ hat Integer $|B|$ und zeigt auf seine Elemente als doppelt verkettete Liste (daher Entfernung von Elementen in $\mathcal{O}(1)$));

- Zustand s auf Block $B \in P$ mit $s \in B$;
- Block $S \in R$ auf Blöcke $B \in P$ mit $B \subseteq S$ als doppelt verkettete Liste;
- Block $B \in P$ auf Block $S \in R$ mit $B \subseteq S$;
- Transition $s \to s'$ auf $count(s, S)$ für $s' \in S \in R$;
- die zusammengesetzten Blöcke bilden eine *Menge* C.

Die Implementierung von *Refine* verfährt dann im einzelnen wie folgt:

1. *(Spalter auswählen)* Entferne einen Block S aus C. Vergleiche die ersten beiden P-Blöcke in S und wähle den kleineren B.

2. *(R aktualisieren)* Entferne B aus S und erzeuge einen neuen Block S' in R, der nur aus B besteht. Wenn S noch zusammengesetzt ist, füge es wieder zu C hinzu.

3. *(Berechne $\leftarrow[B]$)* Kopiere die Elemente von B in einen temporären Block B'. Initialisiere $\leftarrow[B]$ auf leer. Gehe durch $s' \in B$ und eingehende Kanten $s \to s'$ (verlinkt von s' aus!) und füge die aufgefundenen s zu $\leftarrow[B]$ hinzu und markiere und verlinke sie zur Vermeidung von Duplikaten. Berechne gleichzeitig einen neuen Zähler $count(s, B)$ für jedes s, und verlinke diesen mit s.

4. *(Berechne $P' = split(B, P)$)* Spalte jeden Block $D \in P$ mit $D \cap \leftarrow[B] \neq \emptyset$ in $D_1 = D \cap \leftarrow[B]$ und $D_2 = D \setminus D_1$. Gehe dazu durch die Elemente $s \in \leftarrow[B]$. Finde für jedes s das $D \in P$ mit $s \in D$, erzeuge zu D einen neuen mit D verlinkten Block D' in S (wenn nicht schon vorher geschehen), und verschiebe s von D nach D'.

   Erzeuge währenddessen eine Liste der aufgespaltenen Blöcke D. Verarbeite diese Liste anschließend wie folgt:

   (a) Hebe die Verlinkung von D nach D' auf.

   (b) Entferne D, wenn jetzt $D = \emptyset$.

   (c) Sonst: Füge den Block $S \in R$ mit $D \subseteq S$ zu C hinzu, wenn S jetzt genau zwei Blöcke enthält (nämlich D und D').

5. *(Berechne $\leftarrow[B] \setminus \leftarrow[S \setminus B]$)* Gehe wie in Schritt 3 durch die Transitionen $s \to s'$ mit $s' \in B'$. Wenn $count(s, B) = count(s, S)$ (verlinkt mit s bzw. $s \to s'$!), füge s zu $\leftarrow[B] \setminus \leftarrow[S \setminus B]$ hinzu (wenn nicht schon drin, s.o.).

6. *(Berechne split$(S \setminus B, P')$)* Wie für B, aber mit $\leftarrow[B] \setminus \leftarrow[S \setminus B]$ statt $\leftarrow[B]$.

7. *(Aktualisiere Zähler)* Gehe wie gehabt durch die Transitionen $s \to s'$ mit $s' \in B'$. Dekrementiere $count(s, S)$ (verlinkt mit $s \to s'$). Wenn dies 0 wird, lösche den Zähler $count(s, S)$. Ersetze den an $s \to s'$ hängenden Zähler $count(s, S)$ (in jedem Fall) durch $count(s, B)$ (verlinkt mit s). Lösche abschließend B', hebe alle Markierungen auf.

(Das eventuelle Löschen von $count(s, S)$ im letzten Schritt dient gewissermaßen nur dem Aufräumen; wenn $count(s, S)$ nämlich 0 wird, ist der Zähler $count(s, S)$ von keiner Transition $s \to s'$ mehr verlinkt, da alle Nachfolger von s im Block B liegen, der ja zu diesem Zeitpunkt bereits aus S entfernt ist. Die Endwerte der Zähler $count(s, S)$ sind natürlich vorhersagbar, sie betragen am Ende des Gesamtschritts jeweils $count(s, B)$ weniger als vorher, aber man müsste bei direkter Subtraktion zur Vermeidung von wiederholter Subtraktion die schon abgehandelten s wieder aufwändig markieren. Die schrittweise Dekrementierung bedeutet dagegen insofern keinen Effizienzverlust, als ja ohnehin alle Transitionen $s \to s'$ mit $s' \in B'$ einzeln durchgegangen werden müssen.)

Die Laufzeitanalyse gestaltet sich dann wie folgt:

Lemma 56. *Ein* Refine*-Schritt mit Spalter B benötigt Zeit*

$$\mathcal{O}(|B| + \sum_{s' \in B} \leftarrow[\{s'\}]).$$

(Die Summe in der obigen Formel ist gerade die Anzahl Transitionen nach B.)

Beweis. Wir schätzen die Zeit für die einzelnen Phasen von *Refine* ab; es reicht natürlich, dass jede Phase die angegebene asymptotische Schranke einhält.

1. *(Spalter auswählen)* $\mathcal{O}(1)$.

2. *(R aktualisieren)* $\mathcal{O}(1)$.

3. *(Berechne $\leftarrow[B]$)* Diese Phase enthält eine Schleife über B, mit Schleifenkörper in $\mathcal{O}(1)$, und eine Schleife über $s' \in B$ und $s \to s'$, mit Schleifenkörper in $\mathcal{O}(1)$, was gerade Zeit $\mathcal{O}(|B| + \sum_{s' \in B} \leftarrow[\{s'\}])$ benötigt.

4. *(Berechne $P' = split(B, P)$)* Diese Phase enthält eine Schleife über $s \in \leftarrow[B]$ mit Schleifenkörper in $\mathcal{O}(1)$, was also Zeit $\mathcal{O}(|\leftarrow[B]|) = \mathcal{O}(\sum_{s' \in B} \leftarrow[\{s'\}])$ benötigt. Die zweite Schleife über die Liste der aufgespaltenen Blöcke hat höchstens so viele Durchläufe wie die erste.

5. *(Berechne $\leftarrow[B] \setminus \leftarrow[S \setminus B]$)* Wie Phase 3.

6. *(Berechne split$(S \setminus B, P')$)* Wie Phase 4.

7. *(Aktualisiere Zähler)* Diese Phase enthält eine Schleife über $\leftarrow[B]$, mit Schleifenkörper in $\mathcal{O}(1)$, was wiederum Zeit $\mathcal{O}(\sum_{s' \in B} \leftarrow[\{s'\}])$ benötigt. Das abschließende Löschen der Markierungen hält dieselbe Zeitschranke ein.

□

Satz 57. *Der Paige/Tarjan-Algorithmus läuft in Zeit $\mathcal{O}((m+n)\log n)$, wobei m die Anzahl Transitionen des Systems bezeichnet und n die Anzahl Zustände.*

Beweis. In einem Lauf des Algorithmus ist ein gegebener Zustand s' in höchstens $\log(n)+1$ als Spalter verwendeten Blöcken auf, da jeder s' enthaltende Spalter immer höchstens halb so groß wie der vorige solche Spalter ist. (Denn: Wenn s' in einem Spalter B vorkommt, wird B anschließend ein Block in R; sobald also s erneut in einem Spalter B' vorkommt, ist der entsprechende zusammengesetzte Block $S \supseteq B'$ eine Teilmenge von B, und B' höchstens halb so groß wie S.) Wenn $B_1, \dots, B_k$ die vom Algorithmus verwendeten Spalter sind, ist die Gesamtlaufzeit nach dem obigen Lemma also, wenn c die Konstante in der Laufzeitabschätzung des *Refine*-Schritts bezeichnet,

$$
\begin{aligned}
&\sum_{i=1}^{k} c \cdot (|B_i| + |\leftarrow[B_i]|) \\
&= c \cdot \sum_{i=1}^{k} \sum_{s' \in B_i} (1 + |\leftarrow[\{s'\}]|) \\
&= c \cdot \sum_{s' \in \mathcal{Q}} \sum_{i \mid s' \in B_i} (1 + |\leftarrow[\{s'\}]|) \\
&\leq c \cdot \sum_{s' \in \mathcal{Q}} (1 + \log n)(1 + |\leftarrow[\{s'\}]|) \\
&= c \cdot (1 + \log n) \cdot \sum_{s' \in \mathcal{Q}} (1 + |\leftarrow[\{s'\}]|) \\
&= c \cdot (1 + \log n) \cdot (n + m) \\
&= \mathcal{O}((m+n)\log n)
\end{aligned}
$$

□

Wir bemerken noch, dass außer in wenig sinnvollen Randfällen tendenziell $m \geq n$ gilt, so dass die Schranke üblicherweise leicht verkürzt als $\mathcal{O}(m \log n)$ angegeben wird.

3 Hennessy-Milner-Logik

Hennessy-Milner-Logik (HML) macht Aussagen über die mögliche zukünftige Entwicklung eines LTS, vom aktuellen Zustand aus gesehen. Sie ermöglicht z.B. die Formalisierung von Aussagen wie

- CS möchte (momentan) keinen Tee trinken;
- CS möchte (momentan) weder Tee noch Kaffee trinken;
- CS trinkt gleich Kaffee und veröffentlicht dann.

Hennessy-Milner-Logik ist eine Form von *Modallogik*, und erweitert als solche die übliche Aussagenlogik um *Modaloperatoren*:

Definition 58 (HML, Syntax)**.** Formeln in HML über der Menge $\mathcal{A}$ von Aktionen sind durch die Grammatik

$$\phi, \psi ::= \bot \mid \neg\phi \mid \phi \wedge \psi \mid [a]\psi \qquad\qquad (a \in \mathcal{A})$$

definiert, und *Negationsnormalformen* durch

$$\phi, \psi ::= \top \mid \bot \mid \phi \wedge \psi \mid \phi \vee \psi \mid \langle a\rangle\phi \mid [a]\psi \qquad\qquad (a \in \mathcal{A}).$$

In Formeln werden $\top$, $\vee$, $\rightarrow$ und $\mathcal{O}$ wie üblich in $\bot$, $\neg$, $\wedge$ dekodiert, und $\langle a\rangle\phi$ als $\neg[a]\neg\phi$ (dagegen sind $\top$, $\vee$ und $\langle a\rangle$ in Negationsnormalformen First-Class Citizens). Für $A = \{a_1, \dots, a_n\}$ vereinbaren wir ferner die Abkürzungen

$$\begin{aligned}\langle A\rangle\phi &= \langle a_1\rangle\phi \vee \dots \vee \langle a_n\rangle\phi \\ [A]\phi &= \langle a_1\rangle\phi \wedge \dots \wedge \langle a_n\rangle\phi\end{aligned}$$

(mit Randfällen $\langle\emptyset\rangle\phi = \bot$, $[\emptyset]\phi = \top$).

Definition 59 (HML, Semantik)**.** Sei $T = (\mathcal{Q}, \mathcal{A}, \{\xrightarrow{a} \mid a \in \mathcal{A}\})$ ein LTS. Wir definieren *Erfülltheit* $\models$ von Formeln ϕ in Zuständen $s \in \mathcal{Q}$ (mit $s \models \phi$ gelesen als 's erfüllt ϕ') rekursiv durch

$$\begin{aligned}&s \not\models \bot \\ &s \models \phi \wedge \psi && \text{gdw. } s \models \phi \text{ und } s \models \psi \\ &s \models [a]\phi && \text{gdw. } s' \models \phi \text{ für alle } s' \text{ mit } s \xrightarrow{a} s'.\end{aligned}$$

Mit der angegebenen Kodierung folgt dann z.B. ferner

$$\begin{aligned}&s \models \top && \text{stets} \\ &s \models \phi \vee \psi && \text{gdw. } s \models \phi \text{ oder } s \models \psi \\ &s \models \langle a\rangle\phi && \text{gdw. } s' \text{ existiert mit } s \xrightarrow{a} s' \text{ und } s' \models \phi.\end{aligned}$$

Wir definieren die *Extension* $[\![\phi]\!]$ einer Formel ϕ in T als

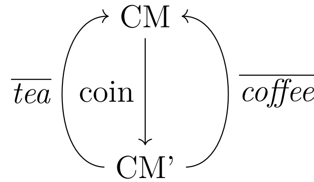
$$[\![\phi]\!] = \{s \in \mathcal{Q} \mid s \models \phi\}.$$

Damit gilt

$$\begin{aligned}
[\![\bot]\!] &= \emptyset \\
[\![\neg\phi]\!] &= \mathcal{Q} \setminus [\![\phi]\!] \\
[\![\phi \land \psi]\!] &= [\![\phi]\!] \cap [\![\psi]\!] \\
[\![\langle a \rangle \phi]\!] &= \{s \in \mathcal{Q} \mid D_a(s) \cap [\![\phi]\!] \neq \emptyset\}
\end{aligned}$$

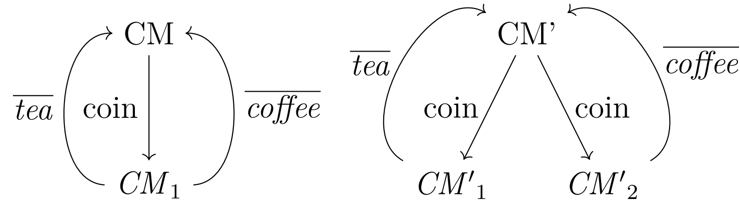
wobei $D_a(s) = \{s' \mid s \xrightarrow{a} s'\}$.

Beispiel 60.



$$\begin{aligned}
&s \models [coin]\top \iff (\forall s'.\ s \xrightarrow{coin} s' \Rightarrow s' \models \top) \iff s \in \{CM, CM'\} \\
&s \models \langle coin \rangle \top \iff (\exists s'.\ s \xrightarrow{coin} s') \iff s \in \{CM\} \\
&s \models [coin]\langle \overline{coffee} \rangle \top \iff (\forall s'.\ s \xrightarrow{coin} s' \Rightarrow s' \models \langle \overline{coffee} \rangle \top) \\
&\quad \iff (\forall s'.\ s \xrightarrow{coin} s' \Rightarrow s' \in \{CM'\}) \iff s \in \{CM, CM'\}
\end{aligned}$$

Beispiel 61.



Es gibt eine Formel ϕ, sodass $CM \models \phi$, aber $CM' \not\models \phi$, und zwar

$$\phi = [coin]\langle \overline{coffee} \rangle \top,$$

weil $[\![\phi]\!] = \{CM, CM_1, CM_1', CM_2'\} \not\ni CM'$.

Wir klären noch kurz, warum wir in der Darstellung von HML via Negationsnormalformen ohne Negation auskommen:

Definition 62. Sei ϕ eine Negationsnormalform in HML. Wir definieren eine Negationsnormalform $\overline{\phi}$ induktiv durch

$$\begin{aligned}\overline{\top} &= \bot \\ \overline{\top} &= \bot \\ \overline{\phi \wedge \psi} &= \overline{\phi} \vee \overline{\psi} \\ \overline{\phi \vee \psi} &= \overline{\phi} \wedge \overline{\psi} \\ \overline{[a]\phi} &= \langle a \rangle \overline{\phi} \\ \overline{\langle a \rangle \phi} &= [a]\overline{\phi}\end{aligned}$$

Lemma 63. *Für jede Negationsnormalform $\overline{\phi}$ in HML sind $\neg\phi$ und $\overline{\phi}$ logisch äquivalent.*

Beweis. Beweis: Induktion über ϕ. □

Wir erinnern daran, dass ein LTS $(\mathcal{Q}, \mathcal{A}, \rightarrow)$ endlich verzweigend ist, wenn $D_\alpha(s) = \{s' \mid s \xrightarrow{\alpha} s'\}$ für alle $s \in \mathcal{Q}$ und alle $\alpha \in \mathcal{A}$ endlich ist.

Beispiel 64 (Unendlich verzweigendes LTS)**.** $P = a.\ \emptyset \mid P$ ist nicht endlich verzweigend:

$$\begin{aligned}&P \xrightarrow{a} \emptyset \mid P \\ &P \xrightarrow{a} a.\ \emptyset \mid \emptyset \mid P \\ &P \xrightarrow{a} a.\ \emptyset \mid a.\ \emptyset \mid \emptyset \mid P \\ &\dots\end{aligned}$$

Definition 65. Zustände s, t sind *(in HML) logisch ununterscheidbar*, wenn s und t dieselben HML-Formeln erfüllen, d.h. wenn die Äquivalenz

$$P \models \phi \iff Q \models \phi$$

für alle HML-Formeln ϕ gilt.

Satz 66 (Bisimulationsinvarianz von HML)**.** *Bisimilare Zustände sind in HML logisch ununterscheidbar.*

Beweis. Seien s, t Zustände in LTS, so dass $s \sim t$, und sei ϕ eine HML-Formel. Wir zeigen per Induktion über ϕ, dass $s \models \phi$ genau dann, wenn $t \models \phi$. Die

Booleschen Fälle $\bot, \neg, \wedge$ sind, wie meistens in Invarianzbeweisen, leicht; zur Illustration führen wir den Schritt für Negation hier explizit durch:

$$\begin{aligned} & s \models \neg\phi \\ & \iff s \not\models \phi \\ & \iff t \not\models \phi && \text{(IV)} \\ & \iff t \models \neg\phi. \end{aligned}$$

Der einzig interessante Fall ist der für die modale Box; per Symmetrie von $\sim$ reicht es, eine der beiden Implikationen zu zeigen. Sei also $s \models [a]\phi$; zu zeigen ist $t \models [a]\phi$. Sei also $t \xrightarrow{a} t'$; dann existiert s' mit $s \xrightarrow{a} s'$ und $s' \sim t'$. Da $s \models [a]\phi$, gilt $s' \models \phi$, nach IV also auch $t' \models \phi$, wie verlangt. □

In endlich verzweigenden Systemen gilt auch die Umkehrung:

Satz 67 (Hennessy/Milner)**.** *In HML logisch ununterscheidbare Zustände in endlich verzweigenden LTS sind bisimilar.*

Beweis. Wir zeigen, dass die Relation

$$R = \{(s,t) \mid s,t \text{ logisch ununterscheidbar}\}$$

eine Bisimulation ist, d.h. dass sich jedes partielle Viereck

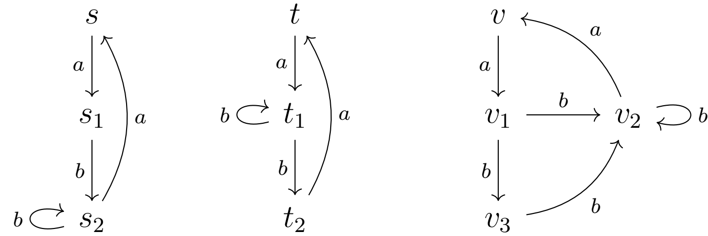
$$\begin{array}{ccc} s & \overset{R}{\text{———}} & t \\ \downarrow{\scriptstyle\alpha} & & \vdots{\scriptstyle\alpha} \\ s' & \overset{R}{\text{-------}} & t' \end{array}$$

wie angedeutet vervollständigen lässt. Wir zeigen die Existenz von t' per Widerspruch. Wenn $t_1, \dots, t_n$ die (endlich vielen!) α-Nachfolger von t sind, nehmen wir also an, dass für jedes $i = 1, \dots, n$ eine Formel ϕ_i existiert, die s' und t_i unterscheidet; da die Logik Negation hat, können wir dann annehmen, dass $s \models \neg\phi_i$ und $t_i \models \phi_i$. Dann gilt $t_i \models \phi_1 \vee \dots \vee \phi_n$ für alle i, aber $s' \not\models \phi_1 \vee \dots \vee \phi_n$, und damit

$$\begin{aligned} t &\models [\alpha](\phi_1 \vee \dots \vee \phi_n) \\ s &\not\models [\alpha](\phi_1 \vee \dots \vee \phi_n), \end{aligned}$$

im Widerspruch zu sRt. □

Beispiel 68.



Gesucht sind paarweise unterscheidende Formeln:

$$(s,t) : [a][b]\langle a\rangle\top$$
$$(t,v) : [a][b]\langle b\rangle\top$$
$$(s,v) : [a][b]\langle a\rangle\top$$

4 Der modale μ-Kalkül

Motivation:

Man betrachte das LTS

$$a \circlearrowleft p$$

$$a \circlearrowleft q \xrightarrow{a} r$$

Die Wurzelzustände p und q sind offenbar nicht bisimilar, und wir finden unmittelbar eine unterscheidende HML-Formel:

$$p \models [a]\langle a\rangle\top$$
$$q \not\models [a]\langle a\rangle\top$$

Das folgende System ist bis auf die Anzahl a-Aktionen, die zum Deadlock führen, sehr ähnlich wie q oben:

$$a \circlearrowleft q_1^n \xrightarrow{a} q_2^n \xrightarrow{a} \dots \xrightarrow{a} q_n^n$$

Wir unterscheiden es in HML wie folgt von p:

$$p \models [a]^n\langle a\rangle\top$$
$$q_1^n \not\models [a]^n\langle a\rangle\top$$

Um Unterschiede wie die zwischen p und q_1^n *allgemein* zufassen, brauchen wir einen Begriff von *Invarianz* bzw. von *Erreichbarkeit* von Eigenschaften: Wir schreiben *Inv* für *invariant* und
Pos für *possible*, und setzen informell

$$\begin{aligned} \mathit{Inv}(\langle a \rangle \top) &= \langle a \rangle \top \wedge [a]\langle a \rangle \top \wedge [a][a]\langle a \rangle \top \wedge \dots \\ \mathit{Pos}([a]\bot) &= [a]\bot \vee \langle a \rangle [a]\bot \vee \langle a \rangle \langle a \rangle [a]\bot \vee \dots \end{aligned}$$

Solche unendlichen Konjunktionen und Disjunktionen sind aber nicht Teil der logischen Syntax, und es wäre zwar theoretisch möglich, aber praktisch eher ungünstig, sie einzuführen. Stattdessen beobachten wir, dass $\mathit{Inv}(\phi)$ die folgende rekursive Gleichung erfüllt:

$$\mathit{Inv}(\phi) = \phi \wedge [\mathcal{A}]\mathit{Inv}(\phi)$$

Ähnlich erfüllt $\mathit{Pos}(\phi)$ die rekursive Gleichung

$$\mathit{Pos}(\phi) = \phi \vee \langle \mathcal{A} \rangle \mathit{Pos}(\phi).$$

Wir können diese Gleichungen als Definitionen ansehen, aber auf sehr unterschiedliche Weise. Beide Gleichungen sind Fixpunktgleichungen; die erste besagt z.B., dass $\mathit{Inv}(\phi)$ grob gesagt Fixpunkt einer Abbildung der Art $X \mapsto (\phi \wedge [\mathcal{A}]X)$ ist (formal lässt sich das im Moment noch nicht fassen, da wir keine Notation für Mengen von Zuständen in Formeln haben), ähnlich in der zweiten Gleichung. Wir können dann die zweite Gleichung im Geiste unserer bereits entwickelten Fixpunkttheorie in partiellen Ordnungen als einen *kleinsten* Fixpunkt begreifen, was am ehesten unserem normalen Begriff von Rekursion entspricht – wenn wir uns *Pos* als rekursiv programmierte Funktion vorstellen, besagt die Gleichung, dass $\mathit{Pos}(\phi)$ in einem Zustand s gilt, also ein ϕ erfüllender Zustand von s aus erreichbar ist, wenn entweder ϕ in s bereits gilt oder s einen Nachfolger hat, von dem aus ein ϕ erfüllender Zustand erreichbar ist; die Antwort „ja“ akzeptieren wir nur bei Terminierung des Verfahrens, nicht abbrechende Suchen nach einem ϕ erfüllenden Nachfolger gelten als fehlgeschlagen, liefern also „nein“ als Antwort.

Dagegen macht eine Sicht auf *Inv* als kleinsten Fixpunkt keinen Sinn; der kleinste Fixpunkt der rekursiven Gleichung im obigen LTS ist z.B. $\emptyset$ (warum?). Korrekt ist hier stattdessen, $\mathit{Inv}(\phi)$ als *größten* Fixpunkt zu lesen: Wir überprüfen gewissermaßen alle von einem Zustand aus erreichbaren Zustände auf Erfülltheit von ϕ; wenn wir immer weiter suchen, ohne eine Verletzung von ϕ zu finden, ist die Antwort „ja“. (Was ist der größte Fixpunkt der Definition von *Pos*?)

4.1 Der (multi-)modale μ-Kalkül

Wie in HML verwenden wir Modalitäten (mehrere, daher „multi“) $[\alpha]$ und $\langle\alpha\rangle$ mit $\alpha \in \mathcal{A}$.

4.2 Syntax

Wir definieren Formeln im μ-Kalkül durch die Grammatik

$$\phi, \psi ::= \neg x \mid x \mid \phi \wedge \psi \mid \phi \vee \psi \mid [\alpha]\phi \mid \langle\alpha\rangle\psi \mid \mu x.\, \phi \mid \nu x.\, \phi$$

wobei x aus einer abzählbaren Menge *Vars* von Variablen kommt, $\alpha \in \mathcal{A}$ und $\neg$ nur vor freien Variablen x im unten definierten Sinn vorkommt. Wir verwenden gelegentlich $\eta \in \{\mu, \nu\}$ als Metavariable in Ausdrücken $\eta x.\, \phi$. Wir können $\top$ und $\bot$ definieren per

$$\top = \nu x.\ x$$
$$\bot = \mu x.\ x$$

Beispiel 69. Der folgende Ausdruck ist eine Formel im μ-Kalkül:

$$\nu z.\ \mu y.\ \nu x.\ ([a](\langle b\rangle x \wedge z) \vee [\mathcal{A}]y)$$

Wir sehen μ und ν als *Variablenbinder* an und definieren demgemäß die Menge $FV(\phi)$ der *freien Variablen* in einer Formel ϕ durch

- $FV(x) = x$;
- $FV(\neg x) = x$;
- $FV(\phi \wedge \psi) = FV(\phi \wedge \psi) = FV(\phi) \cup FV(\psi)$;
- $FV([a]\phi) = FV(\langle a\rangle\phi) = FV(\phi)$
- $FV(\mu x.\ \phi) = FV(\nu x.\ \phi) = FV(\phi) \setminus \{x\}$

Eine Formel ϕ ist *geschlossen*, wenn $FV(\phi) = \emptyset$. Nach obiger Einschränkung der Syntax kommt insbesondere in geschlossenen Formeln keine Negation vor.

4.3 Semantik

Allgemein definieren wir die Semantik von Formeln ϕ über einem LTS $T = (\mathcal{Q}, \mathcal{A}, \rightarrow)$ relativ zu einer *Umgebung*

$$\sigma : \mathit{Vars} \rightarrow \mathcal{P}(\mathcal{Q}),$$

die also eine Interpretation der Variablen als Mengen von Zuständen vorgibt. Die *Extension* $[\![\phi]\!]\sigma \subseteq \mathcal{Q}$ einer Formel ϕ ist dann rekursiv definiert durch

Definition 70. Sei σ eine Umgebung. Wir definieren die Semantik $[\![\phi]\!]_\sigma$ von Formeln induktiv wie folgt:

- $[\![x]\!]\sigma = \sigma(x)$;
- $[\![\neg x]\!]\sigma = \mathcal{Q} \setminus \sigma(x)$;
- $[\![\phi \vee \psi]\!]\sigma = [\![\phi]\!]\sigma \cup [\![\psi]\!]\sigma$;
- $[\![\phi \wedge \psi]\!]\sigma = [\![\phi]\!]\sigma \cap [\![\psi]\!]\sigma$;
- $[\![\langle\alpha\rangle\phi]\!]\sigma = \{s \in \mathcal{Q} \mid \exists s'.\ s \xrightarrow{\alpha} s' \wedge s' \in [\![\phi]\!]\sigma\}$;
- $[\![[\alpha]\phi]\!]\sigma = \{s \in \mathcal{Q} \mid \forall s'.\ s \xrightarrow{\alpha} s' \implies s' \in [\![\phi]\!]\sigma\}$;
- $[\![\mu x.\ \phi]\!]\sigma = \bigcap\{S \subseteq \mathcal{Q} \mid [\![\phi]\!]\sigma[x \mapsto S] \subseteq S\}$;
- $[\![\nu x.\ \phi]\!]\sigma = \bigcup\{S \subseteq \mathcal{Q} \mid S \subseteq [\![\phi]\!]\sigma[x \mapsto S]\}$;

wobei $\sigma[x \mapsto S](y) = \sigma(y)$ für $y \neq x$, und $\sigma[x \mapsto S](x) = S$. Wir lassen σ in $[\![\phi]\!]\sigma$ weg, wenn ϕ geschlossen ist. Wir schreiben in diesem Fall $s \models \phi$ für $s \in [\![\phi]\!]$.

Beispiel 71.

$$
\begin{aligned}
[\![\top]\!] = [\![\nu x.\ x]\!] &= \bigcup\{S \subseteq \mathcal{Q} \mid S \subseteq [\![x]\!][x \mapsto S]\} = \mathcal{Q},\\
[\![\bot]\!] = [\![\mu x.\ x]\!] &= \bigcap\{S \subseteq \mathcal{Q} \mid [\![x]\!][x \mapsto S] \subseteq S\} = \emptyset.
\end{aligned}
$$

Wie bereits für HML können wir Negation definieren mittels

$$
\begin{aligned}
\neg(\neg x) &= x,\\
\neg(\mu x.\, \phi) &= \nu x.\, \neg\phi[\neg x/x],\\
\neg(\nu x.\, \phi) &= \mu x.\, \neg\phi[\neg x/x]
\end{aligned}
$$

und weiteren Klauseln wie in HML. (Man mach sich klar, dass die syntaktischen Beschränkungen des μ-Kalküls hier in der Tat eingehalten werden.)

Beispiel 72. Negation einer μ-Kalkül-Formel:

$$
\begin{aligned}
&\neg(\mu z.\, ([\mathit{tick}]\bot \vee [\mathcal{A}]z)\\
=&\nu z.\, (\langle \mathit{tick}\rangle\top \wedge \langle\mathcal{A}\rangle\neg\neg z)\\
=&\nu z.\, (\langle \mathit{tick}\rangle\top \wedge \langle\mathcal{A}\rangle z).
\end{aligned}
$$

Satz 73. *Sei $\eta x.\, \psi$ eine Formel und σ eine Umgebung. Dann ist die Funktion $S \mapsto [\![\psi]\!]\sigma[x \mapsto S]$ von $\mathcal{P}(\mathcal{Q})$ nach $\mathcal{P}(\mathcal{Q})$ monoton.*

Beweis. Induktion über ψ. Man beachte, dass der Fall $\phi = \neg x$ nicht vorkommt. Wir beweisen als weitere Beispiele nur die Fälle für $\psi = \langle a \rangle \phi$ und $\psi = \mu x.\ \phi$:

$\psi = \langle a \rangle \phi$: Sei $S_1 \subseteq S_2 \subseteq \mathcal{Q}$. Zu zeigen ist

$$[\![\langle a \rangle \phi]\!]\sigma[x \mapsto S_1] \subseteq [\![\langle a \rangle \phi]\!]\sigma[x \mapsto S_2].$$

Sei also $s \in [\![\langle a \rangle \phi]\!]\sigma[x \mapsto S_1]$. Dann existiert s', so dass $s \xrightarrow{\alpha} s'$ und $s' \in [\![\phi]\!]\sigma[x \mapsto S_1]$, woraus nach Induktionsvoraussetzung $s' \in [\![\phi]\!]\sigma[x \mapsto S_2]$ folgt. Per Definition der Semantik gilt also wie verlangt $s \in [\![\langle a \rangle \phi]\!]\sigma[x \mapsto S_2]$.

$\psi = \mu y.\, \phi$: Sei $S_1 \subseteq S_2 \subseteq \mathcal{Q}$. Zu zeigen ist

$$[\![\mu y.\ \phi]\!]\sigma[x \mapsto S_1] \subseteq [\![\mu y.\ \phi]\!]\sigma[x \mapsto S_2].$$

Sei $s \in [\![\mu y.\ \phi]\!]\sigma[x \mapsto S_1]$, d.h. für alle $S \subseteq \mathcal{Q}$ gilt

$$[\![\phi]\!]\sigma[x \mapsto S_1][y \mapsto S] \subseteq S \implies s \in S. \tag{$*$}$$

Wenn $x = y$, dann gilt

$$[\![\phi]\!]\sigma[x \mapsto S_1][y \mapsto S] = [\![\phi]\!]\sigma[y \mapsto S] = [\![\phi]\!]\sigma[x \mapsto S_2][y \mapsto S]$$

Wenn $x \neq y$, dann gilt

$$\begin{aligned} [\![\phi]\!]\sigma[x \mapsto S_1][y \mapsto S] &= [\![\phi]\!]\sigma[y \mapsto S][x \mapsto S_1] \\ &\subseteq [\![\phi]\!]\sigma[y \mapsto S][x \mapsto S_2] && \text{(Per I.V.)} \\ &= [\![\phi]\!]\sigma[x \mapsto S_2][y \mapsto S]. \end{aligned}$$

In beiden Fällen gilt also

$$[\![\phi]\!]\sigma[x \mapsto S_1][y \mapsto S] \subseteq [\![\phi]\!]\sigma[x \mapsto S_2][y \mapsto S] \tag{$**$}$$

Für alle $S \subseteq \mathcal{Q}$ haben wir somit Implikationen

$$[\![\phi]\!]\sigma[x \mapsto S_2][y \mapsto S] \subseteq S \overset{(**)}{\implies} [\![\psi]\!]\sigma[x \mapsto S_1][y \mapsto S] \overset{(*)}{\implies} s \in S.$$

Per Definition der Semantik folgt wie verlangt $s \in [\![\mu y.\, \phi]\!]\sigma[x \mapsto S_2]$. □

Wenn wir also

$$\begin{aligned} f : \mathcal{P}(\mathcal{Q}) &\to \mathcal{P}(\mathcal{Q}) \\ S &\mapsto [\![\phi]\!]\sigma[x \mapsto S] \end{aligned}$$

setzen, erhalten wir nach dem Fixpunktsatz von Knaster und Tarski (Satz 52)

$$\begin{aligned} [\![\mu x.\, \phi]\!]\sigma &= \bigcap \{S \subseteq \mathcal{Q} \mid f(S) \subseteq S\} = \mu f \\ [\![\nu x.\, \phi]\!]\sigma &= \bigcup \{S \subseteq \mathcal{Q} \mid S \subseteq f(S)\} = \nu f \end{aligned}$$

Für endliche LTS gilt damit nach dem Fixpunktsatz von Kleene für endliche Verbände (Satz 53):

$$
\begin{aligned}
[\![\mu x.\, \phi]\!]\sigma &= f^n(\emptyset) \quad \text{für } n \text{ mit } f^n(\emptyset) = f^{n+1}(\emptyset) \\
[\![\nu x.\, \phi]\!]\sigma &= f^m(\mathcal{Q}) \quad \text{für } m \text{ mit } f^m(\mathcal{Q}) = f^{n+1}(\mathcal{Q}).
\end{aligned}
$$

Beispiel 74. Man betrachte das LTS

$$
\overset{\overset{\textit{tick}}{\circlearrowright}}{\textit{Clock}} \xrightarrow{\textit{tick}} \textit{tick}.\,\emptyset \xrightarrow{\quad \textit{tick} \quad} \emptyset
$$

Die Formel $\nu z.\, \langle \textit{tick} \rangle z$ besagt, dass unendlich oft die Aktion *tick* durchgeführt werden kann. Wir berechnen $[\![\nu z.\, \langle \textit{tick} \rangle z]\!]$ nach der oben angedeuteten iterativen Methode: Sei $g(S) = [\![\langle \textit{tick} \rangle z]\!][z \mapsto S]$. Dann gilt per Definition

$$
\begin{aligned}
&g^0(\mathcal{Q}) = \mathcal{Q}, \\
&g^1(\mathcal{Q}) = [\![\langle \textit{tick} \rangle z]\!][z \mapsto \mathcal{Q}] = \{\textit{Clock}, \textit{tick}.\,\emptyset\}, \\
&g^2(\mathcal{Q}) = [\![\langle \textit{tick} \rangle z]\!][z \mapsto \{\textit{Clock}, \textit{tick}.\,\emptyset\}] = \{\textit{Clock}\}.
\end{aligned}
$$

Dagegen besagt die Formel $\mu z.\, [\textit{tick}]\bot \vee \langle \mathcal{A} \rangle z$, dass ein Zustand erreicht werden *kann*, der die Aktion *tick* verweigert. Wir berechnen $[\![\mu z.\, [\textit{tick}]\bot \vee \langle \mathcal{A} \rangle z]\!]$ iterativ: Sei $f(S) = [\![[\textit{tick}]\bot \vee \langle \mathcal{A} \rangle z)]\!][S/z]$. Dann

$$
\begin{aligned}
&f^0(\emptyset) = \emptyset \\
&f^1(\emptyset) = [\![[\textit{tick}]\bot \vee \langle \mathcal{A} \rangle z)]\!][z \mapsto \emptyset] = \{\emptyset\} \\
&f^2(\emptyset) = [\![[\textit{tick}]\bot \vee \langle \mathcal{A} \rangle z)]\!][z \mapsto \{\emptyset\}] = \{\emptyset, \textit{tick}.\,\emptyset\} \\
&f^3(\emptyset) = [\![[\textit{tick}]\bot \vee \langle \mathcal{A} \rangle z)]\!][z \mapsto \{\emptyset, \textit{tick}.\,\emptyset\}] = \{\emptyset, \textit{tick}.\,\emptyset, \textit{Clock}\}
\end{aligned}
$$

Was besagt die Formel $\mu z.\, [\textit{tick}]\bot \vee [\mathcal{A}]z$? Was die Formel $\nu z.\, [\textit{tick}]\bot \vee [\mathcal{A}]z$?

4.4 Temporale Spezifikation im μ-Kalkül

Wir haben bereits folgende Formelschemata kennengelernt:

$$
\begin{aligned}
\textit{Pos}(\phi) &= \mu x.\, (\phi \vee \langle \mathcal{A} \rangle x) && \text{– weak liveness} \\
\textit{Inv}(\phi) &= \nu x.\, (\phi \wedge [\mathcal{A}] x) && \text{– strong safety}
\end{aligned}
$$

Die Operatoren *Pos* und *Inv* sind dual, d.h. $\neg \textit{Pos}(\phi) = \textit{Inv}(\neg \phi)$, denn

$$
\begin{aligned}
&\neg Pos(\phi)\\
&= \neg\mu x.\,(\phi \vee \langle\mathcal{A}\rangle x)\\
&= \nu x.\,(\neg\phi \wedge [\mathcal{A}]x)\\
&= Inv(\neg\phi)
\end{aligned}
$$

Ein weiteres oft benötigtes Formelschema ist *strong liveness*:

$$\mu x.(\phi \vee (\langle\mathcal{A}\rangle\top \wedge [\mathcal{A}]x)),$$

und dual dazu weak safety:

$$\nu x.(\phi \wedge ([\mathcal{A}]\bot \vee \langle\mathcal{A}\rangle x))$$

Was bedeuten diese Formeln?

Computation Tree Logic (CTL) ist ein Fragment des μ-Kalküls, allerdings üblicherweise einer Form des Kalküls, in der zum einen nur eine Aktion angenommen wird (die zudem meist als nie blockiert vorausgesetzt wird) und zum anderen die Zustände mit Wahrheitsbelegungen für propositionale Atome ausgestattet sind; mit anderen Worten sitzt in CTL die atomare Information üblicherweise an den Zuständen, nicht an den Transitionen. Wir können die CTL-Operatoren dennoch sinnvoll für unseren Kontext anpassen: Wir schreiben kurz $\Diamond = \langle\mathcal{A}\rangle$, $\Box = [\mathcal{A}]$, und setzen

- $EX\phi = \Diamond\phi$
- $A[\phi\, U\psi] = \mu x.\, \psi \vee (\phi \wedge \Box x)$
- $E[\phi\, U\psi] = \mu x.\, \psi \vee (\phi \wedge \Diamond x)$

Sinn der Notation ist hierbei, dass A und E sogenannte *Pfadquantoren* sind und besagen, dass eine Formel auf allen maximal langen (ggf. unendlichen) Pfaden vom aktuellen Zustand aus gilt bzw. auf mindestens einem Pfad. Die *temporalen Operatoren* X und U dagegen sind als auf einem bereits gegebenen Pfad intepretiert zu verstehen. Dabei besagt $X\phi$, dass ϕ im nächsten Zustand gilt, und $[\phi\, U\psi]$, dass auf dem Pfad irgendwann ein Zustand erreicht wird, in dem ψ gilt, und bis dahin ϕ gilt. Man mache sich klar, dass obige Definitionen tatsächlich diese Lesart implementieren.

In CTL werden A/E und X/U aber nur in Kombination wie oben verwendet, die dann jeweils als ein Operator angesehen werden (bezeichnet als EX, AU bzw. EU). (In der erweiterten Logik CTL* können Pfadquantoren und temporale Operatoren unabhängig verwendet werden; diese Logik ist allerdings wesentlich schwerer in den μ-Kalkül einzubetten als CTL.)

In CTL werden folgende weitere Operatoren verwendet:

	X (next)	F (finally)	G (globally)	U (until)	W (weak until)
A (all)	AX	AF	AG	AU	AW
E (exists)	EX	EF	EG	EU	EW

Hierbei ist $F\phi$ zu lesen als „ϕ gilt irgendwann auf dem Pfad“, $G\phi$ als „ϕ gilt auf dem gesamten Pfad“, und $\phi\, W\, \psi$ als „ϕ gilt auf dem Pfad (mindestens), bis ψ gilt“ (was aber nicht eintreten muss). Die zusätzlichen Operatoren sind aber durch EX, AU und EU. ausdrückbar:

$$
\begin{aligned}
AX\phi &= \neg EX(\neg\phi) = \neg\Diamond\neg\phi = \Box\phi, \\
AF\phi &= A[\top\, U \phi] = \mu x.\phi \vee (\top \wedge \Box x) = \mu x.\, \phi \vee \Box x, \\
EF\phi &= E[\top\, U \phi] = \mu x.\phi \vee (\top \wedge \Diamond x) = \mu x.\, \phi \vee \Diamond x, \\
AG\phi &= \neg EF(\neg\phi) = \neg\mu x.\neg\phi \vee \Diamond x = \nu x.\, \phi \wedge \Box x, \\
EG\phi &= \neg AF(\neg\phi) = \neg\mu x.\neg\phi \vee \Box x = \nu x.\, \phi \wedge \Diamond x, \\
A[\phi\, W \psi] &= \neg E[\neg\psi\, U \neg(\phi \vee \psi)] = \nu x.\, (\psi \vee (\phi \wedge \Box x)), \\
E[\phi\, W \psi] &= \neg A[\neg\psi\, U \neg(\phi \vee \psi)] = \nu x.\, (\psi \vee (\phi \wedge \Diamond x)).
\end{aligned}
$$

5 Erfülltheitsspiele

Man kann die Erfülltheit von Formeln in Zuständen von LTS spieltheoretisch beschreiben. Die Behandlung des vollen μ-Kalküls in dieser Form führt für unsere gegenwärtigen Zwecke zu weit; man kann aber relativ zwanglos die Erfülltheit von Formeln charakterisieren, die Fixpunkte von nur einer Form (μ oder ν) beinhalten.

Wir entwickeln das Spiel zunächst für den einfachen Fall von geschlossenen Formeln der Form $\mu x.\, \phi_0$ mit ϕ_0 fixpunktfrei. Um die Ausdrucksstärke etwas zu erhöhen, lassen wir $\top$ und $\bot$ als First-Class Citizens in ϕ_0 zu, mit der erwarteten Semantik. Erfülltheit einer solchen Formel in einem Zustand s_0 eines LTS $T = (\mathcal{Q}, \mathcal{A}, \to)$ charakterisieren wir durch folgendes Spiel:

- *Spieler:* $\forall$ (*Abaelard*) und $\exists$ (*Eloise*); $\exists$ spielt für Erfüllheit, $\forall$ dagegen.
- *Konfigurationen:* Paare
  $$(s, \psi)$$
  aus einem Zustand $s \in \mathcal{Q}$ und einer Unterformel ψ von ϕ_0 (inklusive ϕ selbst und x).
- *Initialkonfiguration:* (s_0, x).

- *Züge:* Abhängig von der Formel in der aktuellen Konfiguration:
  - $(s, \phi \wedge \psi)$: $\forall$ ist am Zug und zieht wahlweise zu (s, ϕ) oder (s, ψ).
  - $(s, \phi \vee \psi)$: $\exists$ ist am Zug und zieht wahlweise zu (s, ϕ) oder (s, ψ).
  - $(s, \bot)$: $\exists$ ist am Zug, hat aber keinen Zug.
  - $(s, \top)$: $\forall$ ist am Zug, hat aber keinen Zug.
  - $(s, \langle a \rangle \phi)$: $\exists$ ist am Zug und zieht zu einem (s', ϕ) mit $s \xrightarrow{w} s'$.
  - $(s, [a]\phi)$: $\forall$ ist am Zug und zieht zu einem (s', ϕ) mit $s \xrightarrow{w} s'$.
  - (s, x): Es wird ein Schiedsrichterzug (der zur Vereinheitlichung beliebig $\forall$ oder $\exists$ zugerechnet werden kann) zu (s, ϕ_0) ausgeführt.
- *Spielende:* Nur dann, wenn der am Zug befindliche Spieler nicht ziehen kann (d.h. bei $(s, \top)$ oder $(s, \bot)$, sowie bei $(s, \langle a \rangle \phi)$ oder $(s, [a]\phi)$, wenn s keine a-Transition hat)
- *Gewinnbedingung:* Wer nicht ziehen kann, verliert. Unendliche Spiele gewinnt $\forall$.

Das Spiel für $\nu x.\, \phi_0$ verläuft genauso, mit dem einzigen Unterschied, dass $\exists$ unendliche Spiele gewinnt.

Satz 75. *Für einen Zustand s_0 in einem LTS $T = (\mathcal{Q}, \mathcal{A}, \to)$ und eine μ-Kalkül-Formel $\mu x.\, \phi$ mit ϕ fixpunktfrei und $FV(\phi) \subseteq \{x\}$ gilt $s \models \mu x.\, \phi$ genau dann, wenn $\exists$ das zugehörige Erfülbarkeitsspiel gewinnt; Entsprechendes gilt für $\nu x.\, \phi$.*

Beweis. Wir beweisen zunächst die Aussage für $\nu x.\, \phi_0$, die wir in zwei Implikationen zerlegen:

„$\implies$“: Sei $s_0 \models \nu x.\, \phi_0$; wir müssen eine Gewinnstrategie für $\exists$ im Erfülltheitsspiel für $\nu x.\, \phi_0$ angeben, die wir wieder durch eine Invariante auf Konfigurationen (s, ϕ) beschreiben, nämlich gerade

$$s \models \phi[\nu x.\, \phi_0/x].$$

Wir prüfen die relevanten Bedingungen:

- Die Invariante gilt nach Annahme in der Anfangskonfiguration (s_0, x).
- $\exists$ kann die Invariante durchsetzen: Wir nehmen an, dass die Invariante in (s, ϕ) gilt, und unterscheiden Fälle nach ϕ.

- $\phi = \chi \vee \psi$: $\exists$ ist am Zug. Per Annahme gilt $s \models \chi[\nu x.\, \phi_0/x]$ oder $s \models \psi[\nu x.\, \phi_0/x]$; o.E. Letzteres. Dann zieht s nach (s, ψ), so dass weiter die Invariante erfüllt ist.
- $\phi = \chi \wedge \psi$: $\forall$ ist am Zug; beide möglichen Nachfolgekonfigurationen erfüllen die Invariante.
- $\phi = \langle a \rangle \chi$: $\exists$ ist am Zug. Per Annahme existiert $s \xrightarrow{a} s'$ mit $s' \models \chi[\nu x.\, \phi_0/x]$, und $\exists$ zieht nach (s', χ).
- $\phi = [a]\chi$: $\forall$ ist am Zug und zieht zu (s', χ) mit $s \xrightarrow{a} s'$; nach Annahme gilt dann weiter die Invariante.
- $\phi = x$: Der Schiedsrichter zieht zu (s, ϕ_0); nach der Fixpunkteigenschaft ist $\phi_0[\nu x.\, \phi_0]$ äquivalent zu $\nu x.\, \phi_0$, so dass weiterhin die Invariante gilt.

- Die Invariante gewinnt: Wenn (s, ϕ) die Invariante erfüllt und $\exists$ am Zug ist, dann kann $\exists$ ziehen – dies steht überhaupt nur dann in Frage, wenn ϕ die Form $\phi = \langle a \rangle \chi$ hat, und ist in diesem Fall aber gerade durch $s \models \langle a \rangle \dots$ garantiert. Da $\exists$ unendliche Spiele gewinnt, reicht dies aus.

„$\Longleftarrow$": Wir schreiben kurz $[\![\phi]\!](S) = [\![\phi]\!]\sigma[x \mapsto S]$; dann ist $\nu x.\, \phi_0$ der größte Postfixpunkt von $[\![\phi]\!]$. Es reicht also zu zeigen, dass

$$W_\exists^\nu = \{s \in \mathcal{Q} \mid \exists \text{ gewinnt } (s, x)\}$$

ein Postfixpunkt von $[\![\phi_0]\!]$ ist, d.h.

$$W_\exists^\nu \subseteq [\![\phi]\!](W_\exists^\nu).$$

Wir zeigen durch Induktion über Unterformeln ϕ von ϕ_0 allgemeiner, dass für $s \in \mathcal{Q}$

$$\exists \text{ gewinnt } (s, \phi) \implies s \in [\![\phi]\!](W_\exists^\nu). \tag{1}$$

- $\phi = x$: Per Definition gilt $s \in W_\exists^\nu = [\![x]\!](W_\exists^\nu)$.
- $\phi = \chi \wedge \psi$: Nach den Regeln des Spiels gewinnt $\exists$ sowohl (s, χ) als auch (s, ψ), per Induktionsvoraussetzung also $s \in [\![\chi]\!](W_\exists^\nu)$ und $s \in [\![\psi]\!](W_\exists^\nu)$, also $s \in [\![\chi \wedge \psi]\!](W_\exists^\nu)$.
- $\phi = \chi \vee \psi$: O.E. gewinnt $\exists$ (s, χ); per Induktionsvorausetzung weiter wie im vorigen Fall.
- $\phi = \langle a \rangle \chi$: Es existiert $s \xrightarrow{a} s'$, so dass $\exists$ (s', χ) gewinnt. Nach Induktionsvoraussetzung folgt $s' \in [\![\chi]\!](W_\exists^\nu)$ und damit per Semantik $s \in [\![\langle a \rangle \chi]\!](W_\exists^\nu)$.

- $\phi = [a]\chi$: Sei $s \xrightarrow{a} s'$. Nach den Regeln des Spiels gewinnt $\exists$ (s', χ); per Induktionsvoraussetzung weiter wie oben.

Wenn nun $\exists$ (s, x) gewinnt, gewinnt nach den Spielregeln $\exists$ auch (s, ϕ_0); per (1) folgt $s \in [\![\phi_0]\!](W_\exists^\nu)$ wie verlangt.

Die Aussage für $\mu x.\, \phi_0$ zeigen wir dann wie folgt.

Für „ $\implies$ “ zeigen wir völlig analog wie in der „$\impliedby$“-Richtung des ν-Falls, dass

$$W_\exists^\mu = \{s \mid \exists \text{ gewinnt } (s, x)\}$$

(hier bezieht sich „gewinnt“ natürlich auf das Spiel für $\mu x.\, \phi_0$!) ein Präfixpunkt von $[\![\phi_0]\!]$ ist.

Für „$\impliedby$“ gehen wir per Kontraposition vor, und verwenden den schon bewiesenen Teil für ν. Sei also $s \not\models \mu x.\, \phi_0$. Dann $s \models \nu x.\, \bar{\phi}_0$, wobei $\bar{\phi}_0 = \neg\phi_0[\neg x/x]$ die Dualisierung von ϕ_0 ist. Nach der Aussage für ν folgt, dass $\exists$ das Spiel für $\nu x.\, \bar{\phi}_0$ gewinnt. Dieses Spiel erhält man aber gerade, indem man das Spiel für $\mu x.\, \phi_0$ nimmt und die Rollen von $\exists$ und $\forall$ vertauscht; d.h. $\forall$ gewinnt das Spiel für $\mu x.\, \phi_0$, woraus unmittelbar folgt, dass $\exists$ dieses Spiel nicht gewinnt, wie verlangt. $\square$

6 Der π-Kalkül

Wir erinnern uns daran, dass CCS-Prozesse sich über gemeinsame Kanäle synchronisieren; wir haben bereits eine rein syntaktische Erweiterung des Kalküls kennengelernt, in der über die Kanäle Werte kommuniziert werden. Der π-Kalkül ist mit dem Ziel entwickelt worden, *Mobilität* von Prozessen in diesem Paradigma zu modellieren. Dazu werden zwei neue Ideen in den Kalkül eingeführt:

- Über die Kanäle können auch (bzw. in der Grundversion ausschließlich) Kanalnamen kommuniziert werden, die dann im empfangenden Prozess für weitere Kommunikation genutzt werden. Z.B. in

  $$\bar{a}b.S \mid a(x).\bar{x}c.R \xrightarrow{\tau} S \mid \bar{b}c.R$$

  wird der Kanalname b über Kanal a vom linken zum rechten Prozess gesendet, der anschließend über b mit weiteren Prozessen kommunizieren kann.

- Dies betrifft auch restringierte Namen, also private Kanäle, die damit ihren Scope vergrößern können. Für die Restriktion $P \setminus \{a\}$ schreibt man im π-Kalkül $(\nu a)(P)$ (nicht zu verwechseln mit dem gleichnamigen

Fixpunktoperator; das ν spricht sich auf Englisch wie „new“). Z.B. klappt obiger Transfer von b auch dann, wenn b restringiert ist:

$$(\nu b)(\bar{a}b.S \mid P) \mid a(x).\bar{x}c.R \xrightarrow{\tau} (\nu b)(S \mid P \mid \bar{b}c.R)$$

Hier war der Name b zunächst privat für die linken beiden Prozesse, wird aber dann über den öffentlichen Kanal a an den rechten Prozess kommuniziert, und ist danach privat für alle drei Prozesse.

Die im zweiten Punkt angesprochene *scope extrusion* ist die Hauptursache für die im Vergleich zu CCS relativ hohe technische Komplexität des π-Kalküls.

6.1 Syntax

Die Syntax des π-Kalküls unterscheidet sich wie schon angedeutet nur marginal von der von value-passing CCS. Explizit: Wir unterstellen abzählbar unendliche Vorräte von (Kanal-)namen $a, b, \dots$, die wir als Eingabekanäle $a, b, \dots$ oder Ausgabekanäle $\bar{a}, \bar{b}, \dots$ verwenden, sowie Prozessnamen $A, B, \dots$. Wir unterscheiden nicht zwischen Namen und Variablen. Prozessnamen haben eine gegebene Arität und sind dann über entsprechend viele Namen parametrisiert, d.h. wenn A Arität k hat, verwenden wir A in der Form $A(x_1, \dots, x_k)$. Kanalnamen sind der Einfachheit halber stets einstellig; wir haben Präfixe

$\bar{a}b$	Ausgabe von b über Kanal a
$a(x)$	Empfang von x über Kanal a
τ	Stumme Aktion

– Achtung: Wie in den obigen Beispielen gesehen, spielt der Name x in $a(x)$ die Rolle einer Eingabevariablen, während in $\bar{a}b$ der Name b selbst über a gesendet wird. Prozesse $P, Q, \dots$ definieren wir über die Grammatik

$$P, Q ::= A(x_1, \dots, x_k) \mid \emptyset \mid \alpha.P \mid P + P \mid P|P \mid (\nu x)P.$$

Wie schon in value-passing CCS *bindet* $a(x)$ seine Variable; Gleiches gilt für die Restriktion ν. Wir schreiben $FN(\cdot)$ für die Menge der freien Namen eines syntaktischen Objekts und $BN(\cdot)$ für die Menge der gebundenen Namen.

Wie bisher besteht eine *Prozessspezifikation* aus einer Auswahl von Prozessnamen und einer definierenden Gleichung $A(x_1, \dots, x_k) = P$ für jeden ausgewählten Prozessnamen.

Strukturelle Kongruenz Die Definition der Semantik des π-Kalküls wird zur Einsparung von Regeln typischerweise über einer vorab deklarierten Gleichheit von Prozesstermen definiert. Diese ist zu unterscheiden von später eingeführten Bisimilaritätsbegriffen; sie identifiziert nur Prozesse, die „offensichtlich" das Gleiche tun. Im einzelnen beinhaltet strukturelle Kongruenz $\equiv$ folgende Gleichungen:

- α-Äquivalenz, d.h. Umbenennung gebundener Namen;
- Assoziativität und Kommutativität sowie Neutralität von $\emptyset$ für $+$ und $|$;
- Auffaltung von Prozessdefinitionen; und
- Scope-Erweiterung:

$$\begin{aligned} (\nu x)\emptyset &= \emptyset \\ (\nu x)(P \mid Q) &= P \mid (\nu x)Q && (x \notin FV(P)) \\ (\nu x)(P + Q) &= P + (\nu x)Q && (x \notin FV(P)) \\ (\nu x)(\nu y)P &= (\nu y)(\nu x)P. \end{aligned}$$

Wie der Name sagt, ist $\equiv$ per Definition eine Kongruenz, d.h. die obigen Gleichungen können in Folge und in umgekehrter Richtung sowie tief in Termen verwendet werden.

Insbesondere können wir eine nicht unterhalb eines Präfixes stehende Restriktion (νx) stets ganz nach vorn ziehen, ggf. unter geeigneter Umbenennung. Ferner können wir für $x \notin FV(P)$ die Gleichung $(\nu x)P = P$ herleiten:

$$P = P \mid \emptyset = P \mid (\nu x)\emptyset = (\nu x)(P \mid \emptyset) = (\nu x)P.$$

6.2 Semantik

Wie schon für CCS geben wir die Semantik des π-Kalküls mittels eines LTS an. Dieses LTS benötigt mehr Aktionen als zunächst vielleicht erwartet: Die Prozesse

$$(\nu x)\bar{a}x.P \quad \text{und} \quad \bar{a}x.P$$

müssen klarerweise verschiedene Interpretationen bekommen: Das Präfix $\bar{a}x$ bewirkt im linken Prozess die Extrusion eines *gebundenen* Namens x, während rechts x als *freier* Name kommuniziert wird. Wir führen zur Definition des LTS daher folgende Aktionen ein:

- τ (stumme Aktion)

- $\bar{a}x$ (*freie* Ausgabe)
- $a(x)$ (Eingabe)
- $\bar{a}\nu x$ mit $a \neq x$ (*gebundene* Ausgabe).

Dabei ist x sin $a(x)$ und in $\bar{a}\nu x$ gebunden, dagegen in $\bar{a}x$ frei. Die Regeln der Semantik lauten dann wie folgt:

$$(\mathit{STRUCT})\ \frac{P' \equiv P \quad P \xrightarrow{\alpha} Q \quad Q \equiv Q'}{P' \xrightarrow{\alpha} Q'}$$

$$(\mathit{ACT})\ \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad (\mathit{SUM})\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

$$(\mathit{COM}_1)\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ (BN(\alpha) \cap FN(Q) = \emptyset)$$

$$(\mathit{COM}_3)\ \frac{P \xrightarrow{a(x)} P' \quad Q \xrightarrow{\bar{a}u} Q'}{P \mid Q \xrightarrow{\tau} P'[u/x] \mid Q'}$$

$$(\mathit{RES})\ \frac{P \xrightarrow{\alpha} P'}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'}\ (x \notin FN(\alpha) \cup BN(\alpha))$$

$$(\mathit{OPEN})\ \frac{P \xrightarrow{\bar{a}x} P'}{(\nu x)P \xrightarrow{\bar{a}\nu x} P'}\ (a \neq x)$$

Scope Extrusion funktioniert in diesem System wie folgt: Für $u \notin FN(P)$ haben wir

$$\begin{array}{r c}
(\mathit{RES}) & \dfrac{a(x).P \mid \bar{a}u.Q \xrightarrow{\tau} P[u/x] \mid Q}{(\nu u)(a(x).P \mid \bar{a}uQ) \xrightarrow{\tau} (\nu u)(P[u/x] \mid Q)} \\
(\mathit{STRUCT}) & \overline{a(x).P \mid (\nu u)\bar{a}u.Q \xrightarrow{\tau} (\nu u)(P[u/x] \mid Q)}
\end{array}$$

6.3 Beispiele

- Wir können auch in den Scope einer Restriktion Namen hineinkommunizieren:

$$(\nu b)(a(x).P) \mid \bar{a}b.Q \xrightarrow{\tau} ((\nu b')P[b'/b][b/x]) \mid Q$$

  – wie wird dies im System hergeleitet?

- *Ressourcenmanagement:* Wir nehmen Prozesse R (*Ressource*, z.B. ein Drucker), S (*Server*) und C (*Client*) an. Der Server kontrolliert den Drucker mittels eines nur ihm zur Verfügung stehenden Triggers e:

  $$(\nu e)(\bar{c}e.S \mid e.R).$$

  Der Client fordert den Trigger vom Server über einen Kanal c an:

  $$c(x).\bar{x}.C.$$

  Bei $e \notin FN(C)$ haben wir dann insgesamt Transitionen

  $$\begin{aligned} & c(x).\bar{x}.C \mid (\nu e)(\bar{c}e.S \mid e.R) \\ & \xrightarrow{\tau} (\nu e)(\bar{e}.C \mid S \mid e.R) \\ & \xrightarrow{\tau} (\nu e)(C \mid S \mid R). \end{aligned}$$

- *Koordination von Messages:* Wir nehmen an, dass es mehrere Clients gibt, und der Server aber nacheinander zwei Namen d, e an *denselben* Client kommunizieren möchte. Dies stellt er durch Einrichten eines gemeinsamen Kanals mit einem Client sicher:

  $$(\nu p)\bar{c}p.\bar{p}d.\bar{p}e.S.$$

  Ein Client

  $$c(z).z(x).z(y).C$$

  empfängt nach der Synchronisierung dann beide Namen nacheinander auf p; die Nachrichten können an keinen zweiten Client gehen, da p privat für S und C ist. Wie reduziert die Parallelkomposition von Client und Server? (Achtung: Was ist, wenn $p \in FN(C)$?)

- Wenn der Server darauf wartet, dass der Client mit einer Aufgabe (z.B. eben drucken) fertig ist, verabredet er ein Signal r mit dem Client, der dieses Signal dann sendet, wenn er fertig ist:

  $$((\nu r)\bar{e}r.r.S) \mid e(x).\bar{p}.\bar{x}.C$$

  (irrelevante Argumente von Präfixen lassen wir hier weg). Wie verhält sich dieser Prozess laut Semantik? (Man nehme zur Vereinfachung $r \notin FN(C)$ an.)