

NP Zusammenfassung

Lukas Schaller

June 27, 2019

Contents

1 A1	2
1.1 Aktionen	2
1.2 Trainingsblatt	3
1.3 Nichtdeterminismus	3
1.3.1 Annahmen nebenläufiger Prozesse	3
1.4 Labeled Transitionssystem	4
2 B1 - CSS	5
2.1 CSS_0	5
2.1.1 Syntax	5
2.1.2 Semantik	5
2.2 CCS_0^ω	6
2.2.1 Semantik	6
2.2.2 Geschützte Ausdrücke	6
2.3 Sequentielles CSS_0^ω	6
2.3.1 Semantik	6
2.3.2 Synchronisation	7
2.3.3 Restriktions-Operator	7
2.4 Volle CCS-Power	7
2.4.1 Syntax	7
2.4.2 Semantik	8
2.4.3 Regulärer Ausdruck	8
3 C1	9
3.1 Verklemmung	9
3.2 Gleichheit	10
3.3 (Starke) Bisimilarität	10
3.4 Minimale Repräsentanten eines LTS	11
3.5 Definitionen	12
3.6 CCS_{vp}	13
3.7 CCS_{vp}^Z	13
3.7.1 Ersetzung formell	14
3.8 Nützliches und Vorgehensweisen	14
4 G: Weil Praxis was für Loser ist	15
4.1 Kompatibilität und Programmausführung	15
4.2 Sequentielle Konsistenz	15
4.3 Schwache Konsistenz	16
4.4 Schwache Konsistenz mit Fences	17
4.5 Happens-Before-Konsistenz	18

Chapter 1

A1

1.1 Aktionen

Actionen Seien Kom eine Menge von Kommunikationsaktionen und Int eine davon disjunkte Menge von internen Aktionen. Dann ist

$$Act = Kom \cup Int$$

die Menge aller Aktionen. Dabei gelten folgende Konventionen:

$$\alpha, \beta, \gamma, \dots \in Act \quad a, b, c, \dots \in Int \quad [a], [b], [c], \dots \in Kom$$

Transitionssystem (LTS) Ein beschrittes Transitionssystem TS ist ein Tripel (S, \rightarrow, s_0) , wobei

- S die Zustandsmenge
- $\rightarrow \subseteq S \times Act \times S$ die Transitionsrelation,
- $s_0 \in S$ der initiale Zustand ist

Nachfolger Sei ein LTS $TS = (S, \rightarrow, s_0)$ gegeben. Sei $s \in S$, $C \subseteq S$, $\alpha \in Act$, und $A \subseteq Act$.

$$Post(s, \alpha) = \{s' \in S \mid s \xrightarrow{\alpha} s'\},$$

$$Post(s, A) = \bigcup_{\alpha \in A} Post(s, \alpha)$$

$$Post(C, \alpha) = \bigcup_{s \in C} Post(s, \alpha),$$

$$Post(C, A) = \bigcup_{\alpha \in A} Post(C, \alpha)$$

Aktionen die in Zusand s als nächstes möglich sind:

$$Act(s) = \{\alpha \in Act \mid \exists s' : s \xrightarrow{\alpha} s'\}$$

Aktionen, die in Zustand s als nächstes beobachtet werden können:

$$Kom(s) = \{\alpha \in Kom \mid \exists s' : s \xrightarrow{\alpha} s'\}$$

Erreichbarkeit $Reach(s)$ ist die Menge aller von s erreichbaren Zustände in TS

$$Reach(s) = \bigcup_{n \in \mathbb{N}} Post^n(s)$$

wobei $Post^0(s) = s$ und $Post^{n+1}(s) = Post(Post^n(s), Act)$

1.2 Trainingsblatt

Vorgänger

$$Pre(s, \alpha) = \{s' \in S \mid s' \xrightarrow{\alpha} s\},$$

$$Pre(s, A) = \bigcup_{\alpha \in A} Pre(s, \alpha)$$

$$Pre(C, \alpha) = \bigcup_{s \in C} Pre(s, \alpha),$$

$$Pre(C, A) = \bigcup_{\alpha \in A} Pre(C, \alpha)$$

terminaler Zustand Ein terminaler Zustand ist ein Zustand ohne Nachfolger. Ein Zustand ist genau dann terminal wenn $Act(s) = \emptyset$.

Alphabet des LTS Alphabet eines LTS $TS = \bigcup_{s \in \text{Reach}(TS)} Kom(s)$

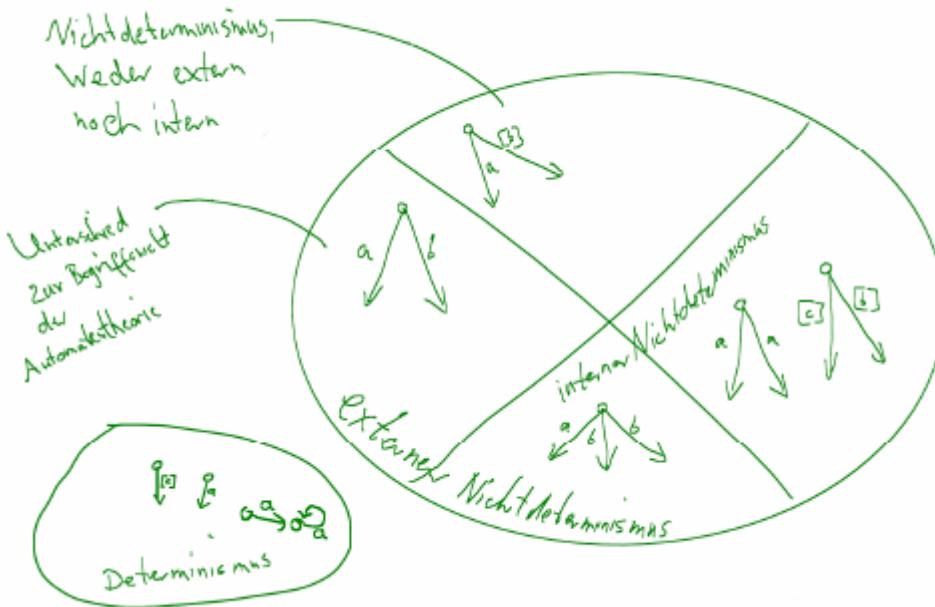
1.3 Nichtdeterminismus

Nichtdeterminismus Sei $Post(s) = Post(s, Act)$ und es sei ein LTS $TS = (S, \rightarrow, s_0)$ gegeben. TS ist deterministisch genau dann wenn für alle $s \in S$,

$$|Post(s)| \leq 1 \text{ und } |Act(s)| \leq 1$$

Andernfalls heißt das TS nichtdeterministisch!

Interner und Externer Nichtdeterminismus

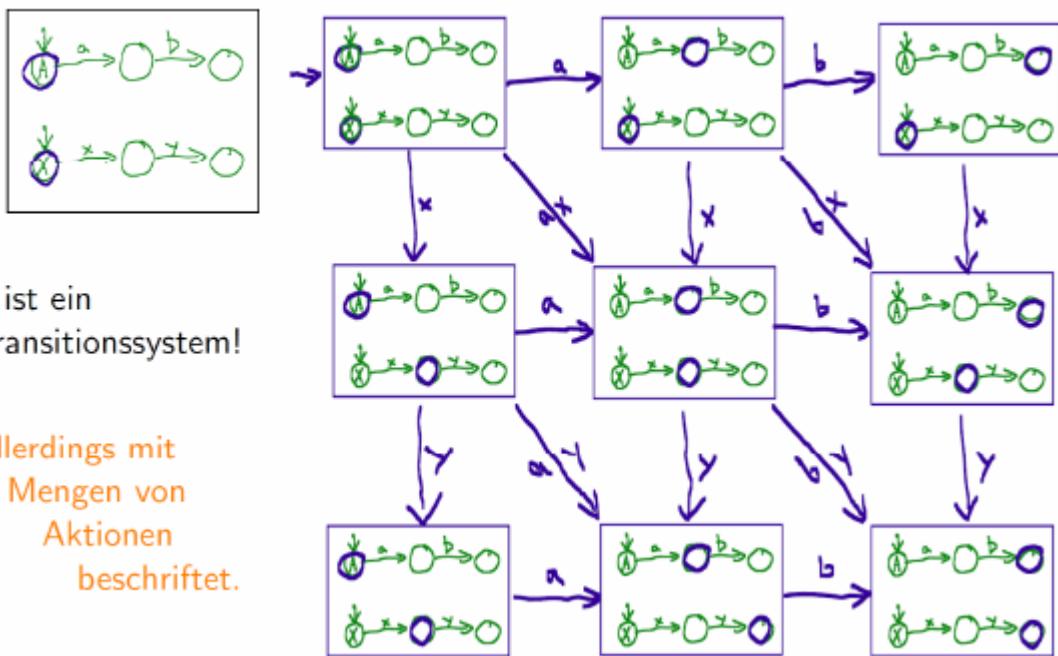


1.3.1 Annahmen nebenläufiger Prozesse

- Zeit wird nur bezüglich relativen Geschwindigkeit der Prozesse untereinander betrachtet, nicht absolut. Jeder Prozess kann gerade beliebig schnell oder langsam voranschreiten.
- Aktionen sind unteilbar und zeitlos.
- Nebenläufige Prozesse agieren komplett voneinander unabhängig und unbeeinflusst, es sei denn, sie kommunizieren explizit miteinander.

1.4 Labeled Transitionssystem

Ein kleines Beispiel:



Das ist ein
Transitionssystem!

Allerdings mit
Mengen von
Aktionen
beschriftet.

Hinweis: Die diagonalen Kanten können wegen der Zeitlosigkeit der Aktionen auch weggelassen werden.

Chapter 2

B1 - CSS

2.1 CSS_0

2.1.1 Syntax

Gegeben sei die Menge aller Aktionen Act . Dann ist die Menge aller Ausdrücke in CSS_0 gegeben durch:

$$P ::= 0 \mid P + P \mid \alpha.P$$

wobei $\alpha \in Act$.

Klammersparregeln

- '+' klammert links: $P + Q + R \rightsquigarrow (P + Q) + R$
- '.' klammert rechts: $\alpha.\beta.P \rightsquigarrow \alpha.(\beta.P)$
- Punkt vor Strich: $\alpha.P + Q \rightsquigarrow (\alpha.P) + Q$

2.1.2 Semantik

Die Semantik einer Sprache beschreibt, welches mathematische Objekt mit einem Ausdruck der Sprache assoziiert werden soll. Die Semantik des Ausdrucks P aus CCS_0 ist ein LTS $\llbracket P \rrbracket = (S, \rightarrow, s_0)$.

- Zustandsmenge S ist die Menge aller CCS_0 -Ausdrücke
- $s_0 = P$
- \rightarrow ist eine Teilmenge von $(CSS_0 \times Act \times CCS_0)$

Inferenzregeln

Hinweis: CCS_0 beinhaltet noch nicht die Inferenzregeln rec

$\text{choice_l} \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$	$\text{prefix} \quad \frac{}{a.P \xrightarrow{\alpha} P}$
$\text{choice_r} \quad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$	$\text{rec} \quad \frac{\Gamma(X)=P \quad P \xrightarrow{\alpha} P'}{X \xrightarrow{\alpha} P'}$

Isomorphie

Zwei Transitionssysteme $TS = (S, \rightarrow, s_0)$ und $TS' = (S', \rightarrow', s'_0)$ sind isomorph ($TS \sim_{iso} TS'$), wenn es eine Bijektion f gibt mit $f : \text{Reach}(TS) \rightarrow \text{Reach}(TS')$, so dass $f(s_0) = s'_0$ und für alle $s_1, s_2 \in \text{Reach}(TS)$ und alle $\alpha \in Act$ gilt: $s_1 \xrightarrow{\alpha} s_2$ genau dann wenn $f(s_1) \xrightarrow{\alpha} f(s_2)$.

Intuition: Zwei LTS sind isomorph solange sie sich ausschließlich im Namen der Zustände unterscheiden. Die Transitionen müssen gleich bleiben!

Satz Für jedes endliche azyklische LTS TS gibt es einen Ausdruck $P \in CCS_0^\omega$, so dass $TS \sim_{iso} \llbracket P \rrbracket$.

2.2 CCS_0^ω

2.2.1 Semantik

Gegeben sei die Menge aller Aktionen Act und eine Menge von Rekursionsvariablen Var . Dann ist die Menge der Ausdrücke in CCS_0^ω wie folgt:

$$P ::= o \mid X \mid P + P \mid \alpha.P , \text{ wobei } \alpha \in Act \text{ und } X \in Var$$

Beispiele

$$\begin{aligned} X &:= a.b.Y \\ Y &:= b.Z + a.Y \\ Z &:= a.Y \end{aligned}$$

Eine Menge solcher Gleichungen definiert offensichtlich eine partielle Funktion: $\Gamma \in Var \rightarrow CCS_0^\omega$. In diesem Beispiel ergibt sich demnach: $\Gamma = \{(X, a.b.Y), (Y, b.Z + a.Y), (Z, a.Y)\}$

Es soll gelten:

- $(\alpha.P, \alpha, P) \in \rightarrow$;
- $(P + Q, \alpha, P') \in \rightarrow$, wann immer $(P, \alpha, P') \in \rightarrow$;
- $(P + Q, \alpha, Q') \in \rightarrow$, wann immer $(Q, \alpha, Q') \in \rightarrow$;
- $(X, \alpha, P') \in \rightarrow$, wann immer $\Gamma(x) = P$ und $(P, \alpha, P') \in \rightarrow$;
- nichts sonst ist Element von \rightarrow_Γ

2.2.2 Geschützte Ausdrücke

Eine Variable X ist geschützt in einem Ausdruck P , wenn jedes Auftreten von X in P in einem Teilausdruck von P der Form $\alpha.Q$ enthalten ist. Andernfalls heißt X ungeschützt.

Ein Ausdruck P heißt geschützt, wenn alle darin vorkommenden Variablen in P geschützt sind. Andernfalls heißt P ungeschützt.

Beispiele

ungeschützte Ausdrücke: $X, \tau.X + Y, (a.X) + X$ geschützte Ausdrücke: $\alpha.X, \tau.(X + Y), \alpha.(X + b.X)$

2.3 Sequentielles CCS_0^ω

Für eine Bindung $\Gamma : Var \rightarrow CCS_0^\omega$ sei $\rightarrow_\Gamma \subseteq CCS_0^\Gamma \times Act \times CCS_0^\Gamma$ die kleinste Relation \longrightarrow , die den Inferenzregeln genügen.

2.3.1 Semantik

Sei $LTS_0^\omega = \{(CCS_0^\omega, T, s) | T \subseteq CCS_0^\omega \times Act \times CCS_0^\omega, s \in CCS_0^\omega\}$. Die Semantik von CCS_0^ω ist eine (kaskadierte) Funktion:

$$\begin{aligned} \llbracket _ \rrbracket : (Var \rightarrow CCS_0^\omega) &\rightarrow CCS_0^\omega \rightarrow LTS_0^\omega \\ \llbracket _ \rrbracket \Gamma P &= (CCS_0^\omega, \rightarrow_\Gamma, P) \end{aligned}$$

Wir schreiben $\llbracket P \rrbracket_\Gamma$ für $\llbracket _ \rrbracket \Gamma P$, oder auch nur $\llbracket P \rrbracket$, sofern Γ aus dem Kontext heraus klar ist.

Satz Für jedes endliche LTS TS gibt es einen Ausdruck $P \in CCS_0^\omega$ und eine endliche Bindung Γ , sodass $TS \sim_{iso} \llbracket P \rrbracket_\Gamma$

Zusätzliche Inferenzregeln

TODO: Hier Bild von par.l und par.r einfügen

2.3.2 Synchronisation

Synchronisation bietet die Grundlage um unterschiedliche Prozesse auf Aktionen reagieren zu lassen. So kann zum Beispiel die 'light'-Aktion des Feuerzeug-Prozesses in einem Prozess eines Chinaböllers die passive Aktion 'light' hervorrufen. In CSS machen wir dazu die Kommunikationsaktionen entweder 'aktiv' oder 'passiv'. Statt einer beliebigen Menge Kom von Markierungen verwenden wir eine Menge, die mehr Struktur aufweist: $Kom = A! \cup A?$. Aktionen mit '!' als Output-Aktionen (aktiv) und Aktionen mit '?' als Input-Aktionen interpretiert werden.

Komplementarität Input- und Output-Aktionen treten als Paare auf. Das Komplement von $a \in A! \cup A?$ ist \bar{a} . Auch für interne Aktionen τ gilt $\tau = \bar{\tau}$. Das doppelte Komplement hebt die Wirkung auf: $\alpha \in Act : \bar{\bar{a}} = \alpha$

Hinweis Interne Aktionen können NICHT synchronisieren.

Zusätzliche Inferenzregel

$$\text{sync} \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

2.3.3 Restriktions-Operator

Der Restriktions-Operator 'unterbindet bestimmte (Paare von) Aktionen eines Prozesses. Intuitiv gesprochen erzwingt er eine Synchronisation.

$P \setminus H$ ist ein zweistelliger Operator, mit P als CCS-Ausdruck und H als menge von Kommunikationsaktionen, die unterbunden werden sollen.

Inferenzregel

$$\text{res} \quad \frac{P \xrightarrow{\alpha} P' \quad \alpha \notin H}{P \setminus H \xrightarrow{\alpha} P' \setminus H}$$

Annahmen über zulässige Menge H:

- Die interne Aktion kann nicht unterbunden werden: $\tau \notin H$
- Aktionen treten in H paarweise auf: $a \in H \iff \bar{a} \in H$

2.4 Volle CCS-Power

2.4.1 Syntax

Gegeben sei die Menge aller Kommunikationsaktionen $Kom = A! \cup A?$, $Act = Kom \cup \{\tau\}$ und eine Menge von Rekursionsvariablen Var . Dann ist die Menge aller Ausdrücke in CCS wie folgt gegeben:

$$P ::= 0 \mid X \mid P + P \mid \alpha.P \mid P|P \mid P \setminus H$$

wobei $\alpha \in Act$, $X \in Var$ und $H \subseteq Kom$.

2.4.2 Semantik

Die Semantik der Ausdrücke con CCS ist gegeben durch:

$$\llbracket \cdot \rrbracket : (Var \rightarrow CCS) \rightarrow CCS \rightarrow LTS^{CCS}$$

$$\llbracket \cdot \rrbracket \Gamma P = (CCS, \rightarrow_\Gamma, P)$$

wobei $LTS^{CCS} = \{(CCS, T, s) | T \subseteq CCS \times Act \times CCS, s \in CCS\}$ uns \rightarrow_τ die kleinste Relation \rightarrow ist, die den folgenden Regeln genügt.

Inferenzregeln

$\text{prefix} \quad \frac{}{a.P \xrightarrow{\alpha} P}$	$\text{choice_l} \quad \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$	$\text{choice_r} \quad \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$	$\text{rec} \quad \frac{\Gamma(X)=P \quad P \xrightarrow{\alpha} P'}{X \xrightarrow{\alpha} P'}$
$\text{par_l} \quad \frac{P \xrightarrow{\alpha} P'}{P Q \xrightarrow{\alpha} P' Q'}$	$\text{sync} \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P Q \xrightarrow{\tau} P' Q'}$	$\text{par_r} \quad \frac{Q \xrightarrow{\alpha} Q'}{P Q \xrightarrow{\alpha} P Q'}$	$\text{res} \quad \frac{P \xrightarrow{\alpha} P' \quad \alpha \notin H}{P \setminus H \xrightarrow{\alpha} P' \setminus H}$

2.4.3 Regulärer Ausdruck

Ein CCS-Ausdruck wird regulär genannt, wenn er durch folgende Grammatik gebildet werden kann:

$$P ::= 0 \mid X \mid P + P \mid \alpha.P \mid R$$

$$R ::= 0 \mid R + R \mid \alpha.R \mid R|R \mid R \setminus H$$

Satz Sofern Γ eine endliche Bindung ist bei der alle rechten Seiten regulär sind, ist $Reach(\llbracket P \rrbracket_\Gamma)$ für jedes $P \subset CCS$ endlich.

Chapter 3

C1

Dieses Kapitel befasst sich mit der beobachtbaren Verhalten von Prozessen. Die Kommunikationsaktionen und die Spuren (mögliche Aktionsfolgen) lassen sich beobachten. Die internen Aktionen und die durchlaufenden Zustände lassen sich offensichtlich nicht beobachten!

Definition: Spuren

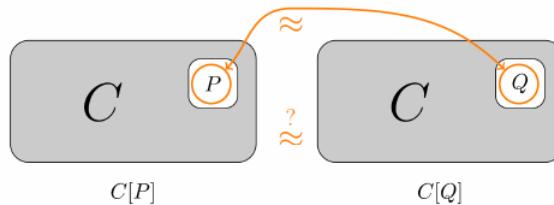
Sei $TS = (S, \rightarrow, s_0)$ ein LTS. Dann definieren wir die Funktion $LTS \rightarrow 2^{Act^*}$ als

$$Traces(TS) := \{\alpha_1\alpha_2\dots\alpha_n \in Act^* | n \geq 0 \exists s_1, \dots, s_n : \forall 0 < i \leq n : s_{i-1} \xrightarrow{\alpha_i} s_i\}$$

als die Menge der endlichen Spuren von TS . Wir schreiben $s_0 \rightsquigarrow^\varrho s'$

Definition: Kongruenzrelation

Eine Äquivalenzrelation \approx auf CCS ist eine Kongruenzrelation, wenn für alle CCS-Ausdrücke P, Q und alle Kontexte $C[.]$, $P \approx Q$ impliziert, dass $C[P] \approx C[Q]$ gilt.



Beispiel: $\approx = \{x, y \in \mathbb{Z}^2 \mid |x| = |y|\}$

Algebraische Gesetze:

$$\begin{array}{ll} P + Q \sim_{tr} Q + P & \text{Kommutativität von } + \\ (P + Q) + R \sim_{tr} P + (Q + R) & \text{Assoziativität von } + \\ P + 0 \sim_{tr} P & 0 \text{ als neutrales Element von } + \end{array}$$

Spuräquivalenz

Sei $\Gamma : Var \rightarrow CCS, P, Q, R \in CCS$ und $H \subseteq Kom$ sowie $\alpha \in Act$. Wenn $P \sim_{tr} Q$, dann auch:

$$\begin{array}{lll} P + R \sim_{tr} Q + R & R + P \sim_{tr} R + Q & P|Q \sim_{tr} Q|R \\ R|P \sim_{tr} R|Q & \alpha.P \sim_{tr} \alpha.Q & P \setminus H \sim_{tr} Q \setminus H \end{array}$$

3.1 Verklemmung

Ein Zustand s ist terminal, falls $Post(s, Act) = \emptyset$. Wir schreiben dann $s \rightsquigarrow$. Falls P in $\llbracket P \rrbracket_\Gamma$ terminal ist, nennen wir P einen verklemmten Prozess.

Terminierende Spuren Die terminierenden Spuren eines Prozesses P sind alle Spuren, die einem terminalen Zustand enden.

$$\begin{aligned} TTraces(P) &= TTraces(\llbracket P \rrbracket_{\Gamma}) \\ TTraces(\llbracket P \rrbracket) &:= \{\varrho \in Traces(\llbracket P \rrbracket_{\Gamma}) \mid \exists P' : P \rightsquigarrow^{\varrho} P' \wedge P' \rightarrow\} \end{aligned}$$

Beispiele:

- $\text{Traces}(a!.b!.0 + a!.0) = \{\epsilon, a!, a! b!\}$
- $\text{Traces}(a!.b!.0) = \{\epsilon, a!, a! b!\}$
- $TTraces(a!.b!.0 + a!.0) = \{a!, a! b!\}$
- $TTraces(a!.b!.0) = \{a! b!\}$

3.2 Gleichheit

Die 'Super'-Gleichheit...

- ist eine Äquivalenzrelation
- ist eine Kongruenzrelation
- erhält Spuren
- ist verklemmungssensitiv (erhält terminierende Spuren)
- ist grob genug (größer als \sim_{iso})

3.3 (Starke) Bisimilarität

Zwei Prozesse sind äquivalent, wenn sie zustandsweise gleich sind. 'Gleiche' Zustände erlauben Transitionen mit den gleichen Aktionen, die wieder in 'gleichen' Zuständen landen.

Eine Relation $R \subseteq S \times S$ über den Zuständen eines LTS $TS = (S, \rightarrow, s_0)$ ist eine (starke) Bisimulation, wenn für alle $P, Q \in S$ mit $(P, Q) \in R$ und für alle $\alpha \in Act$ gilt:

- Für jedes $P' \in S$ mit $P \xrightarrow{\alpha} P'$ gilt: Es gibt ein $Q' \in S$ mit $Q \xrightarrow{\alpha} Q'$ und $(P', Q') \in R$.
- Für jedes $Q' \in S$ mit $Q \xrightarrow{\alpha} Q'$ gilt: Es gibt ein $P' \in S$ mit $P \xrightarrow{\alpha} P'$ und $(P', Q') \in R$.

Zwei Zustände heißen bisimilar wenn es eine Bisimulation R gibt mit $(P, Q) \in R$.

$$(P, Q) \in \sim \Leftrightarrow \exists \text{ Bisimulation } R : (P, Q) \in R \Leftrightarrow P \text{ und } Q \text{ sind bisimilar}$$

\sim ist die Relation aller zueinander bisimilaren Zustände. Sie heißt **Bisimilarität**. Gleichzeitig ist \sim eine Bisimulation, und zwar die größtmögliche.

Algebra der Bisimilarität

Zwei Prozesse $P, Q \in CCS_0$ sind genau dann bisimilar ($P \sim Q$), wenn sie sich durch Anwenden der folgenden Axiome syntaktisch ineinander umgeformt werden können:

- $P + Q \sim Q + P$
- $(P + Q) + R \sim P + (Q + R)$
- $(P + 0 \sim P$
- $P + P \sim P$

3.4 Minimale Repräsentanten eines LTS

Hier wird das *kleinste* LTS, das bisimilar zu dem gegebenen LTS ist, gesucht. Der gefundene Repräsentant ist bis auf Isomorphie eindeutig!

Algorithmus

1. Gehe davon aus, dass alle Zustände gleich (im Sinne von Bisimilarität) sind
2. Suche solange Argumente, warum Zustände noch verschieden sind, bis die Zustände in ihre Äquivalenzklassen getrennt sind
3. Fasse Zustände einer Äquivalenzklasse Zusammenfassung
4. Lösche doppelte Transitionen

Kapitel D

3.5 Definitionen

Spur: $P \xrightarrow{\sigma} Q$

Für $\sigma = \alpha_1 \dots \alpha_n \in Act^*$ schreiben wir $P \xrightarrow{\sigma} Q$, falls $\sigma = \varepsilon$ und $P = Q$ oder es einen Zustand s_0, \dots, s_n gibt, mit $s_{i-1} \xrightarrow{\alpha_i} s_i$ für $i = 1, \dots, n$ und $P = s_0$ und $Q = s_n$

Schwache Bisimulation $s \approx t$

- wenn $s \xrightarrow{\alpha} s'$, dann gibt es eine Transition $t \xrightarrow{\alpha} t'$, so dass $s' \approx t'$
- wenn $t \xrightarrow{\alpha} t'$, dann gibt es eine Transition $s \xrightarrow{\alpha} s'$, so dass $s' \approx t'$

Auswahl-Kongruenz: \approx^+

Zwei Ausdrücke $P, Q \in CCS_0$ heißen genau dann Auswahl-Kongruent, geschrieben $P \approx^+ Q$, wenn für alle $R \in CCS_0$ gilt:

$$P + R \approx Q + R$$

Beobachtungskongruenz: $\hat{\equiv}$

- wenn $P \xrightarrow{\alpha} P'$, dann $\exists (n, m) \in \mathbb{N}^2 : Q \xrightarrow{\tau^n \alpha \tau^m} Q' \wedge P' \approx Q'$
- wenn $Q \xrightarrow{\alpha} Q'$, dann $\exists (n, m) \in \mathbb{N}^2 : P \xrightarrow{\tau^n \alpha \tau^m} P' \wedge P' \approx Q'$

Es gilt außerdem: $P \hat{\equiv} Q \Leftrightarrow P =^+ Q$

Schwache Transition: \Longrightarrow

Sei ein LTS $TS = (S, \rightarrow, s_0)$ gegeben. Wir definieren die Transitionsrelation $\Longrightarrow \subset S \times Act \times S$ wie folgt:

- $s \xrightarrow{\tau} s' \Leftrightarrow \exists n \geq 0 : s \xrightarrow{\tau^n} s'$
- $s \xrightarrow{a} s' \Leftrightarrow \exists s'', s''' \in S : s \xrightarrow{\tau} s'' \xrightarrow{a} s''' \xrightarrow{\tau} s'$

mit $a \in Kom$. Beachte dass auch $s'' = s$ und $s''' = s'$ möglich ist.

Weak Traces: \sim_{wtr}

$$WTraces(P) = \{a_1 a_2 \dots a_n | \tau^* a_1 \tau^* a_2 \tau^* \dots \tau^* a_n \tau^* \in Traces(P)\}$$

Schwache Spuräquivalenz: $P \sim_{wtr} Q$

P und Q sind schwach Spuräquivalent $P \sim_{wtr} Q$, genau dann wenn $WTraces(\llbracket P \rrbracket_\Gamma) = WTraces(\llbracket Q \rrbracket_\Gamma)$ gilt

3.6 CCS_{vp}

Syntax

Die Menge aller Ausdrücke in $CCS_{vp}^{\mathbb{Z}}$ ist durch die folgende Grammatik gegeben:

$$P ::= 0 \mid X[u_1, \dots, u_n] \mid P + P \mid \alpha.P \mid P|P \mid P\setminus H \mid \text{when}(b)P,$$

wobei

$$\alpha ::= \tau \mid a! \mid a? \mid a!v \mid a?v \mid a!x \mid a?x$$

und $a \in \mathbb{K}$, $X \in Var$, $H \subseteq Kom$, $v \in \mathbb{V}$, $x \in D$ sowie $r_i \in D \cup \mathbb{V} \cup \mathbb{K}$.

Semantik

Die Regeln aus CCS werden um folgende Regeln erweitert bzw prefix wird abgeändert:

$$\begin{array}{c} \text{prefix} \quad \frac{\alpha \in \text{Act}}{\alpha.P \xrightarrow{\alpha} P} \\ \text{rec} \quad \frac{T(X[u_1, \dots, u_n]) = P \quad P\{e_1/u_1, \dots, e_n/u_n\} \xrightarrow{\alpha} P'}{X[e_1, \dots, e_n] \xrightarrow{\alpha} P'} \\ \text{input} \quad \frac{\sqrt{\alpha} \in \mathbb{V}}{\alpha?x.P \xrightarrow{\alpha?v} P\{v/x\}} \end{array}$$

3.7 CCS_{vp}^Z

Syntax

Die Menge aller Ausdrücke in $CCS_{vp}^{\mathbb{Z}}$ ist durch die folgende Grammatik gegeben:

$$P ::= 0 \mid X[u_1, \dots, u_n] \mid P + P \mid \alpha.P \mid P|P \mid P\setminus H \mid \text{when}(b)P,$$

wobei

$$\alpha ::= \tau \mid a! \mid a? \mid a!e \mid a?x$$

und $X \in Var$, $H \subseteq Kom$, $a \in \mathbb{K} \cup D$, $e \in AExp$, $b \in BExp$, $x \in D$ sowie $u_i \in AExp \cup \mathbb{K}$.

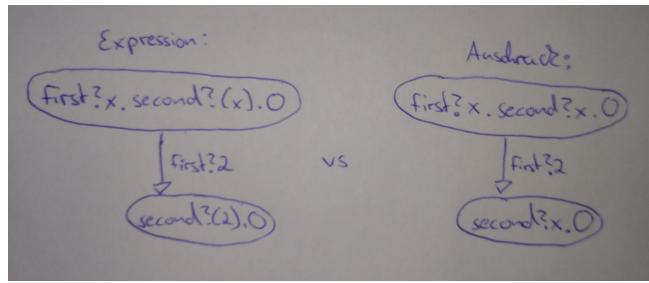
Semantik von CCS_{vp}^Z

$$\begin{array}{c} \text{prefix} \quad \frac{\alpha \in \{a!, a? \mid a \in \mathbb{K}\} \cup \{\tau\}}{\alpha.P \xrightarrow{\alpha} P} \\ \text{output} \quad \frac{e \Downarrow n}{a!e.P \xrightarrow{a!n} P} \quad \text{value} \quad \frac{e \Downarrow n}{a?e.P \xrightarrow{a?n} P} \quad \text{input} \quad \frac{n \in \mathbb{Z}}{a?x.P \xrightarrow{a?n} P \{n/x\}} \\ \text{res} \quad \frac{P \xrightarrow{\alpha} P' \quad \alpha \notin H}{P \setminus H \xrightarrow{\alpha} P' \setminus H} \quad \text{choice_l} \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \quad \text{choice_r} \quad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'} \\ \text{par_l} \quad \frac{P \xrightarrow{\alpha} P'}{P | Q \xrightarrow{\alpha} P' | Q} \quad \text{par_r} \quad \frac{Q \xrightarrow{\alpha} Q'}{P | Q \xrightarrow{\alpha} P | Q'} \quad \text{sync} \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P | Q \xrightarrow{\tau} P' | Q'} \\ \text{cond} \quad \frac{P \xrightarrow{\alpha} P' \quad b \Downarrow \text{true}}{\text{when}(b)P \xrightarrow{\alpha} P'} \quad \text{rec} \quad \frac{\Gamma(X[u_1, \dots, u_n]) = P \quad P\{e_1/u_1, \dots, e_n/u_n\} \xrightarrow{\alpha} P'}{X[e_1, \dots, e_n] \xrightarrow{\alpha} P'} \end{array}$$

3.7.1 Ersetzung formell

$0\{v/x\}$	=	0
$(P + Q)\{v/x\}$	=	$P\{v/x\} + Q\{v/x\}$
$(P Q)\{v/x\}$	=	$P\{v/x\} Q\{v/x\}$
$(\alpha.P)\{v/x\}$	=	$P\{v/x\} \setminus \{\alpha\{v/x\} \mid \alpha \in H\}$
$a?\text{ oder } a!\{v/x\}$	=	$a?\text{ oder } a!, \text{ falls } a \neq x$ $v?\text{ oder } v!, \text{ falls } a = x$
$\tau\{v/x\}$	=	τ
$(a?y.P)\{v/x\}$	=	$a?\{x/v\}y.P\{v/x\}, \text{ falls } x \neq y$ $a?\{x/v\}y.P, \text{ falls } x = y$
$(a!y.P)\{v/x\}$	=	$a!\{x/v\}y.P\{v/x\}, \text{ falls } x \neq y$ $a!\{x/v\}y.P, \text{ falls } x = y$
$X[y_1, \dots]\{x/v\}$	=	$X[y_1, \dots], \text{ falls } x \neq y$ $X[v, \dots], \text{ sonst}$

Variable vs Ausdruck:



3.8 Nützliches und Vorgehensweisen

Relationen im Überblick

	Id(CCS)	\sim_{iso}	\sim	\sim_{tr}	Univ(CCS)
Äquivalenzrelation	✓	✓	✓	✓	✓
Kongruenzrelation	✓	✗	✓	✓	✓
erhält Spuren	✓	✓	✓	✓	✗
verklemmungssensitiv	✓	✓	✓	✗	✗
grob genug	✗	✗	✓	✓	✓

Chapter 4

G: Weil Praxis was für Loser ist

Ihr wollt Praxis? Hier noch ein paar Definitionen für euch, anstatt darüber zu diskutieren wann es sinnvoll ist sein Programm nebenläufig zu gestalten und welche Vorteile es wirklich bringen könnte. Mit anderen Worten: Warum machen wir das hier eigentlich? Keine Ahnung, hier die Definitionen...

4.1 Kompatibilität und Programmausführung

Folgendes Beispiel zeigt einen Programm-Code und dessen Ausführungsdiagramm. Für jedes Programm gibt es genau einen Thread, das heißt einen Strang im Diagramm. Im Ausführungsdiagramm befinden sich lediglich Lese- und Schreiboperationen wieder, wobei $x?v$ eine Leseoperation ist und der Wert v in die globale Variable x eingelesen wird. $x!v$ ist eine Schreiboperation, bei der v in die Variable x geschrieben wird. Es gibt eine Programmordnung - hier mit blauen Pfeilen dargestellt - die angibt, welche Ausdrücke vor anderen Ausdrücken ausgeführt werden müssen. Die roten Pfeile geben die *writeSeen* Funktion an, die für einen Leseoperation angibt, woher der gelesene Wert stammt, also wo er geschrieben wurde. Wichtig: Beide Funktionen unterliegen nicht zwingend einer logischen, zeitlichen Ordnung die der Programm-Code vermuten lässt!

```

1 number = 42;      |   r_1 = ready;
2 ready = true;    |   r_2 = ready;
3           |   r_3 = number;
4           |   r_4 = number;

```

Figure 4.1: Code

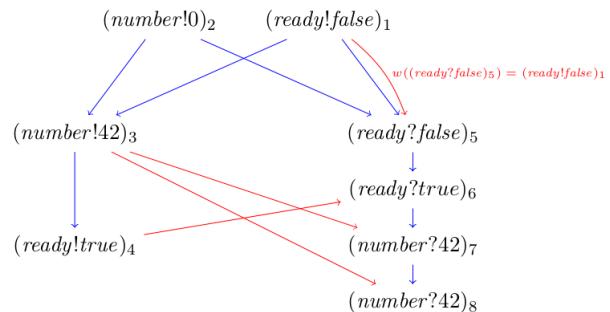


Figure 4.2: Diagramm

4.2 Sequentielle Konsistenz

In diesem Modell nimmt man an, dass jeder Thread für sich sequentiell ausgeführt wird. Leseoperationen beziehen sich hier also implizit auf Schreiboperationen in der Vergangenheit. Die einzelnen Schritte zwischen den Threads kann man jedoch beliebig mischen. Das heißt also, dass alle Ausdrücke eines Programms in der "richtigen" Reihenfolge ausgeführt werden (so wie im Programm notiert von oben nach unten), aber der Thread nach jedem Schritt zu einem anderen Thread wechseln kann und dort weiter macht. Es ist die klassische Vorstellung, dass einfach jeder Thread irgendwann seine Prozessorzeit bekommt um weiter ausgeführt zu werden während die anderen warten... (auch wenn das so nicht ganz stimmt)

```

1 number = 42;      |   r_1 = ready;
2 ready = true;    |   r_2 = ready;
3           |   r_3 = number;
4           |   r_4 = number;

```

Figure 4.3: Code

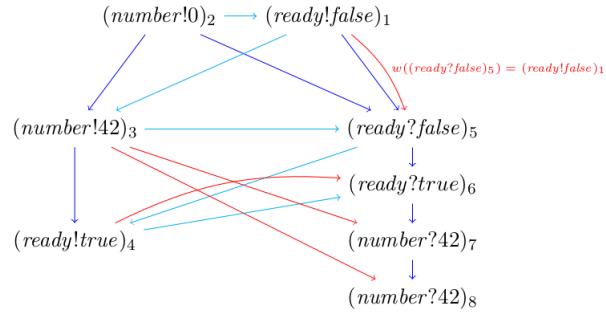


Figure 4.4: Diagramm

Die hellblauen Pfeile ergeben mit den blauen Pfeilen der Programmordnung nun eine totale Ordnung der Operationen...

Sequentielle Konsistenz Eine mit einem Programm kompatible Ausführung (J, P, w) ist *sequentiell konsistent* wenn es eine totale Ordnung \leq auf J gibt, so dass:

- $P \subseteq \leq$
- für alle $x?v \in J$ gilt:
 - $w(x?v) \leq x?v$
 - es gibt kein $x!v' \neq w(x?v)$ so dass $w(x?v) \leq x!v' \leq x?v$

4.3 Schwache Konsistenz

Hier wird es unintuitiv. Mit schwacher Konsistenz müssen Leseoperationen sich nun nicht mehr auf eine Schreiboperation in der Vergangenheit beziehen (Out-Of-Thin-Air). Diese Schreiboperation darf sich jedoch nicht in der Zukunft befinden! Mit anderen Worten: Man darf nicht aus der Zukunft lesen aber beliebig aus der Vergangenheit, solange es einen Pfad gibt bei dem der Wert nicht zwischendrin überschrieben wird. Das ermöglicht z.B. das Lesen längst veralteter Werte. Die totale Ordnung der Operationen wird somit partiell und entspricht noch den einzelnen Programmabläufen (Ein Statement kommt immernoch vor dem darunter stehenden Statement im Programm). Das Ergebnis des Programms lässt sich allerdings daran nicht mehr eindeutig erklären.

Intuition-Regel : Solange man nicht in der eigenen natürlichen Programm-Ordnung aus der Zukunft liest, kann gelesen werden was man will. Heißt also aus anderen partiellen Pfaden kann man quasi immer lesen wo man möchte. In der eigenen natürlichen Programmordnung müsste man den aktuellen Wert lesen!

```

1 number = 42;      |   r_1 = ready;
2 ready = true;    |   r_2 = ready;
3           |   r_3 = number;
4           |   r_4 = number;

```

Figure 4.5: Code

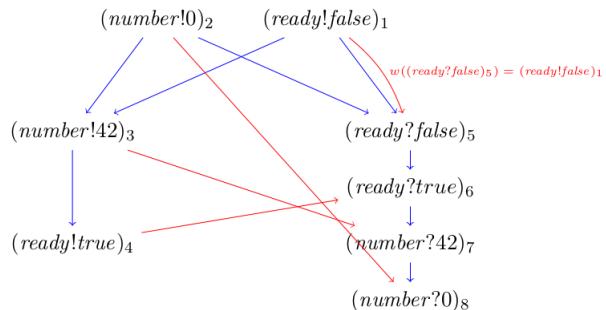


Figure 4.6: Diagramm

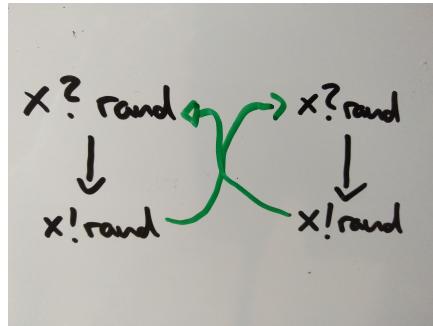
Warum gilt hier $(number?0)_8$? Man nimmt von $(number!0)_2$ z.B. den rechten Pfad, da hier nicht $number$ zwischendrin überschrieben wird.

Schwache Konsistenz Eine mit einem anderen Programm kompatible Ausführung (J, P, w) ist schwach konsistent, wenn es eine partielle Ordnung \leq auf J gibt, so dass:

- $P \subseteq \leq$
- für alle $x?v \in J$ gilt:
 - $\neg(w(x?v) \geq x?v)$
 - es gibt kein $x!v' \neq w(x?v)$ so dass $w(x?v) \leq x!v' \leq x?v$

Out-Of-Thin-Air

Mit schwacher Konsistenz kann man jegliche zufällige Werte lesen, die nirgends irgendwo geschrieben werden. Allerdings ist das nicht an jeder Stelle möglich, denn dieser Wert muss sich zyklisch begründen lassen. Wenn ein Programm einen zufälligen Wert ließt, muss es irgendwo später diesen auch schreiben und sich auf ein anderes Programm beziehen, welches diesen auch ließt und danach schreibt. Dieses Schreiben muss sich dann auf das Lesen des ersten Programms beziehen. Somit haben sich die Programme untereinander begründet obwohl der Wert im eigentlich Programm nie vorgekommen ist.



4.4 Schwache Konsistenz mit Fences

Fences sind ein Low-Level-Konzept um Probleme mit veralteten Daten in Caches besser in den Griff zu bekommen. Werte, die nach einem Fence geschrieben werden können nicht vor dem Fence gelesen werden. Wenn ein Wert nach einer Fence-Anweisung geschrieben wird, sehen wir alles, was davor geschrieben wurde. Dieser Satz im Skript möchte sagen, dass ab diesem Zeitpunkt die aktuellen Werte (auch aller anderen Variablen dieses Programms) wirklich gelten. Die folgenden zwei Beispiele sollen die Funktion verdeutlichen.

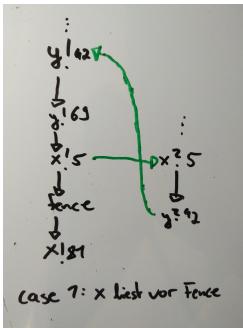


Figure 4.7: x liest vor Fence

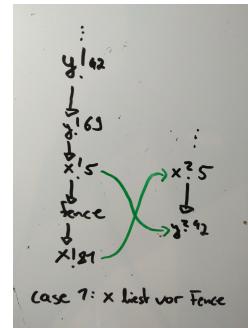


Figure 4.8: x liest nach Fence

Man sieht, dass in dem rechten Teilprogramm noch nicht aus dem neuen Wert von x nach dem “fence” gelesen wird. Solange das nicht passiert, darf y also auch auf einen veralteten Wert, nämlich den ersten Wert zugreifen (Unter der Annahme von schwacher Konsistenz)

Angenommen das x im rechten Programm ließt den Wert nach dem Fence. Würde davor gelesen werden könnten diese Operationen sich noch auf Werte vor dem Fence beziehen. Das y steht in der partiellen Ordnung

aber nun nach dem x (rechte Seite). Dementsprechend kann y nun nur noch den aktuellsten Wert in dem linken Teilprogramm lesen. Veraltete Werte sind jetzt nicht mehr möglich!

number = 0 und ready = false.

```

1 number = 42;      |   r_1 = ready;
2 fence;           |   r_2 = ready;
3 ready = true;    |   r_3 = number;
4                   |   r_4 = number;

```

Figure 4.9: Code

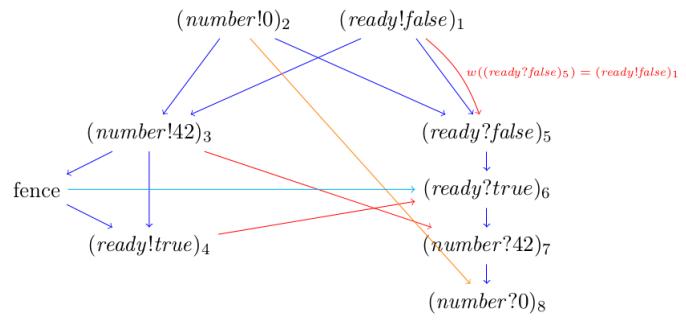


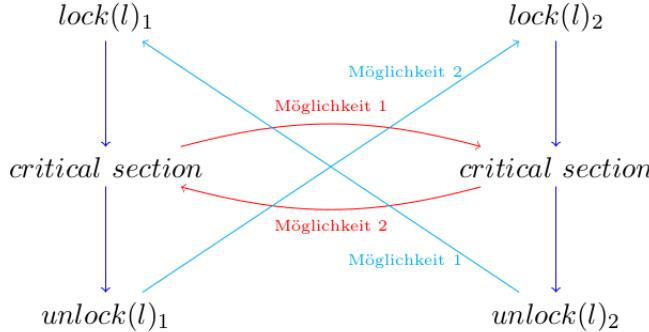
Figure 4.10: Diagramm

Schwache Konsistenz mit Fences Eine mit einem Programm mit fence-Anweisung kompatible Ausführung (J, P, w) ist schwach konsistent wenn es eine partielle Ordnung \leq auf J gibt, so dass:

- $P \subseteq \leq$
- für alle $x?v \in J$ gilt:
 - $\neg(w(x?v) \geq x?v)$
 - es gibt kein $x!v' \neq w(x?v)$ so dass $w(x?v) \leq x!v' \leq x?v$
- für alle fence-Operationen und alle Lese-Operationen $x?v \in J$ gilt: $w(x?v) \geq \text{fence} \Rightarrow x?v \geq \text{fence}$

4.5 Happens-Before-Konsistenz

Der Begriff "Schwache Konsistenz mit Locks" beschreibt diese Konsistenz in unserem Fall vielleicht besser. Eine Ordnung wird zwischen den Ausführungen der kritischen Abschnitte festgelegt um sicherzustellen, dass jeder Thread in einem Solchen eine aktuelle Sicht auf den Speicher hat. Es ergeben sich also mit der **partiellen Ordnung** und den locks nur noch folgende **writeSeen**-Möglichkeiten.



Happens-Before-Konsistenz Eine mit einem Programm mit Locks kompatible Ausführung (J, P, w) ist happens-before-konsistent wenn es eine partielle Ordnung \geq auf J gibt, so dass:

- $P \subseteq \leq$
- für alle $x?v \in J$ gilt:
 - $\neg(w(x?v) \geq x?v)$
 - es gibt kein $x!v' \neq w(x?v)$ so dass $w(x?v) \leq x!v' \leq x?v$
- für zwei verschiedene Lock-Unlock-Paare $lock(l)_1, unlock(l)_1$ und $lock(l)_2, unlock(l)_2$ gilt entweder
 - $unlock(l)_1 \leq lock(l)_2$ oder
 - $unlock(l)_2 \leq lock(l)_1$

Intuition Das Locken erweitert die Ordnung und macht sie total. Sie dafür die folgenden Graphen:

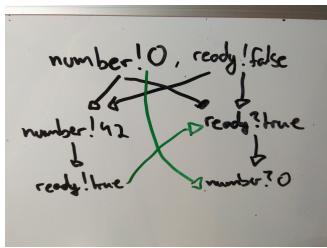


Figure 4.11: Graph ohne locks

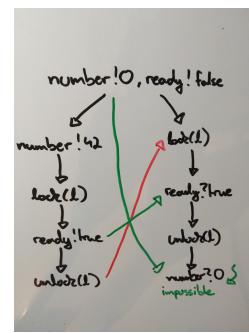


Figure 4.12: Graph mit locks und Problem

Man beachte, wenn man eine lock-unlock-Reihenfolge festgelegt hat - hier mit rotem Pfeil dargestellt - werden die zwei partiellen Programmabläufe jetzt zu einer neuen natürlichen Programmordnung. Deshalb kann in Figur 4.10 nicht 0 in number eingelesen werden, da jetzt "number!42" in der eigenen natürlichen Programmordnung steht und somit ein bereits aktualisierter Wert gelesen wird!