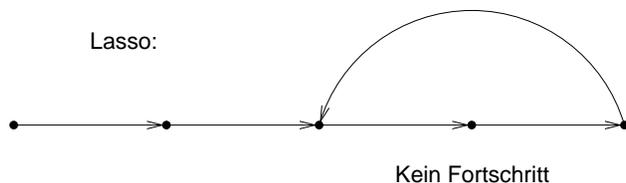


- Liveness – Lebendigkeit
 - im Gegensatz zu Safety- bzw. Sicherheitseigenschaften
 - beschreibt **unausweichbares** Verhalten
 - macht wiederum nur Sinn bei Abstraktion vom konkretem Zeitverhalten:
Konzentration auf mögliche Ereignisfolgen ohne “Timing”
- **Deadlock** ist noch eine Sicherheitseigenschaft:
 - “keine Zustand ohne Nachfolger ist erreichbar”
- **Livelock** als prototypische Lebendigkeitseigenschaft:
 - “System verfängt sich nie in einer Endlosschleife ohne *echten* Fortschritt”

Gegenbeispiele zu Lebendigkeitseigenschaften

- Endliche Systeme:



- Unendliche Systeme:
 - “Divergenz” möglich: Gegenbeispiel muss kein Lasso sein
 - z.B. Zähler über natürlichen Zahlen hochzählend erreicht nie wieder Anfangswert

Weitere Beispiele zu Liveness

- Terminierung von Programmen/Prozessen/Protokollen:
 - z.B. Quicksort terminiert
 - z.B. IEEE Firewire: Initialisierungsphase terminiert mit einer korrekten Topologie
- Erwartete Ereignisse treten auch tatsächlich ein:
 - z.B. Befehle in superskalaren Prozessoren werden “Retired”
 - z.B. der Aufzug kommt wenn er gerufen wird
- in allen Beispielen werden **keine konkreten** Zeitschranken angegeben

Fairness

- Abstraktion vom konkreten “Scheduler”:
 - Reihenfolge der Ausführung von Prozessen ist im Modell beliebig
 - z.B. bei Interleaving oder voll asynchroner Parallelkomposition
- dadurch Artefakte bzw. unrealistische Gegenbeispiele zu Lebendigkeitseigenschaften:
 - Livelock durch permanentes Ignorieren eines Prozesses
- **Fairness** im Prinzip:
 - Scheduling **vernachlässigt** keinen (ausführbaren) Prozess für immer ...
 - ... ohne einen konkreten Scheduler anzugeben
(letzteres oftmals nicht möglich, da relative Geschwindigkeit nicht bekannt)

Definition ein *fares* LTS $A = (S, I, \Sigma, T, F)$ ist ein gewöhnliches LTS (S, I, Σ, T) , mit $F \subseteq T$ die Menge der *fairen* Übergänge von A .

(ein Übergang ist einfach eine Trippel (s, a, s'))

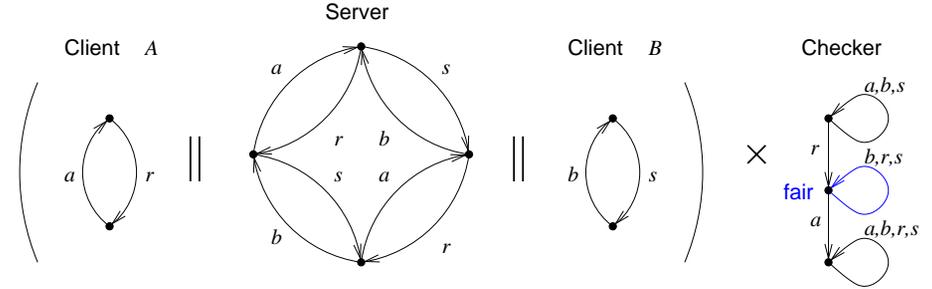
Definition ein unendlicher Pfad $\pi = s_0 \xrightarrow{a_0} s_1 \rightarrow \dots$ in einem fairen LTS ist *fair* gdw.

π unendlich viele faire Übergänge enthält: $|\{i \mid (s_i, a_i, s_{i+1}) \in F\}| = \infty$.

Beispiel wähle für F die Menge der Übergänge in denen eine deterministische Komponente A_j einen lokalen oder globalen Übergang macht, oder aber "disabled" ist ($s \not\rightarrow_a$ für alle $a \in \Sigma_j$). Ein dadurch gegebener fairer Pfad in $A_1 \parallel \dots \parallel A_n$ muss A_j immer wieder "ausführen".

Suche nach Fairen Pfaden

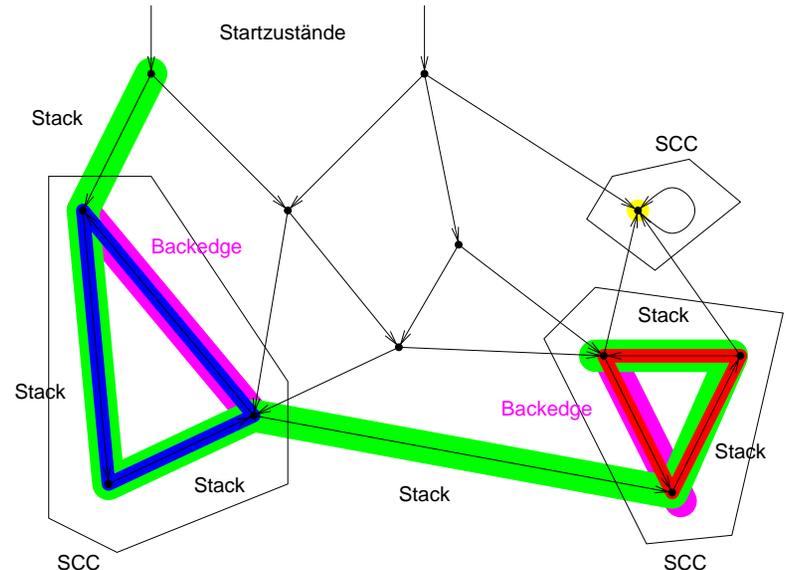
- Im Prinzip (führt aber zu quadratischem Aufwand):
 - Backedge: Kante von `current` zu Knoten auf dem Stack der rekursiven DFS
 - Zyklus geschlossen durch Backedge ist fair gdw. er einen fairen Übergang hat
- SCC = strongly connected component
 - maximale Menge von Unterknoten eines Graphen (= Zustandsraum), in der jeder Knoten von jedem anderen erreichbar ist
- jeder Zyklus (also auch der Zyklus eines Lasso) ist vollständig in einer SCC enthalten
 - SCC's lassen sich leicht in DFS finden: linearer Algorithmus von Tarjan zur Zerlegung eines Graphen in seine SCC



$F =$ alle $\{b, r, s\}$ -Übergänge, bei denen der Checker im unteren Zustand ist (Gegenbeispiel dazu, dass ein a nach einem r irgendwann erfolgen muss)

es gibt einen fairen Pfad mit dem Trace $r(bs)^\omega = rbsbsbs\dots$ (bei dem A genau einmal ausgeführt wird)

SCC



- für jeden Knoten/Zustand merke
 1. *Depth First Search Index* (DFS_I): ordnet Knoten wie sie gefunden werden
 2. minimal zu erreichender DFS_I durch Backedge (MRDFS_I)
(auch über noch nicht besuchte Nachfolger)
- bestimme DFS_I in der Prefix-Phase
(bevor Nachfolger besucht werden)
- minimiere MRDFS_I rekursiv über MRDFS_I der Nachfolger
(in der Suffix-Phase)
- `pop` in rekursiver DFS erst dann wenn und solange MRDFS_I = DFS_I
- all die Knoten mit demselben MRDFS_I bilden eine SCC

Faire Pfade bei Verallgemeinerter Fairness

- analog zum Algorithmus mit nur einem Fairness Constraint:
 - Teste beim Schließen eines Zyklus *alle* Fairness Constraints
 - oder überprüfe ob gefundene SCC alle Fairness Constraints “erlaubt”
- Alternative Konstruktion:
 - reduziere mehrere Fairness Constraints auf eines
 - zunächst lineare Ordnung der Fairness Constraints, z.B. $F_1 < \dots < F_n$
 - Kreuzprodukt mit modulo n Zähler welcher von i nach $(i + 1) \bmod n$ wechselt gdw. der Übergang im LTS das Fairness Constraint F_{i+1} erfüllt
 - einziges neues Fairness Constraint beim Zählerübergang von $n - 1$ nach 0

Problem im Beispiel ist der Scheduler nicht *fair*

Definition ein *verallgemeinertes faires LTS* $A = (S, I, \Sigma, T, F_1, \dots, F_n)$ ist ein LTS (S, I, Σ, T) , mit $F_i \subseteq T$ eine Familie von Fairness Constraints.

Definition Übergang (s, a, s') heißt *fair* für das Fairness Constraint F_i gdw. $(s, a, s') \in F_i$.

Definition ein unendlicher Pfad $\pi = s_0 \xrightarrow{a_0} s_1 \rightarrow \dots$ in einem verallgemeinerten fairen LTS ist *fair* gdw. π für jedes Fairness Constraint F_j unendlich viele faire Übergänge enthält: $|\{i \mid (s_i, a_i, s_{i+1}) \in F_j\}| = \infty$.

Voriges Beispiel wähle als zweites Fairness Constraint alle Übergänge, bei denen A beteiligt ist. Dann gibt es keinen fairen Pfad, also auch kein Gegenbeispiel, dem “Request” r folgt also immer ein “Acknowledge” a .