ReaSyn

Diplomvortrag zu den Arbeiten

Synthesis of Reactive Systems

und

Distributed Games for Reactive Systems

Jens Regenberg (jens@regenberg.org), Tobias Maurer (mail@tobias-maurer.com)

Gliederung

- Einleitung
- Architektur- und Spezifikationstransformation
- Verteilte Spiele
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse

Gliederung

- Einleitung
  - Problemstellung
  - **Rea***Syn* User Interface
  - Verteilte Synthese
  - Automaten und Spiele
- Architektur- und Spezifikationstransformation
- Verteilte Spiele
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse

Problemstellung

Verteilte Synthese

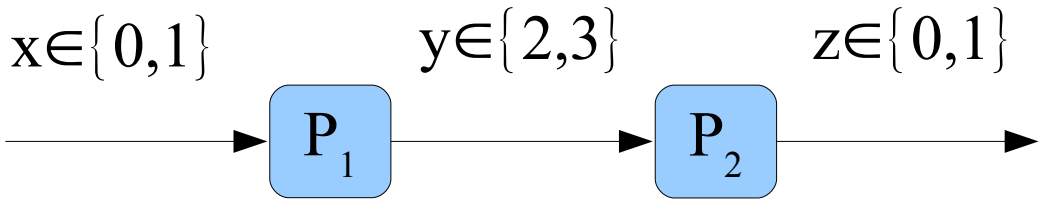
Gegeben eine Architektur und Spezifikation.
Gibt es eine Implementierung für die Prozesse der Architektur,
so dass das Gesamtverhalten der Spezifikation genügt?

Unser Modell

- LTL Spezifikation
- zwischen den Prozessen gibt es keine Delays
- die Umgebung besitzt volle Information

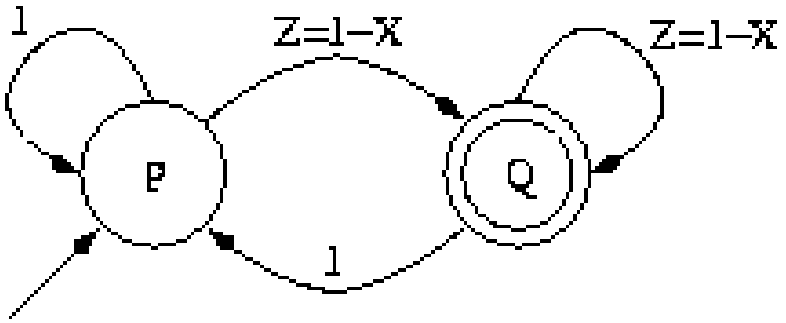
Problemstellung

Architektur:



Spezifikation:

$$\Diamond(Z = 1 - X)$$

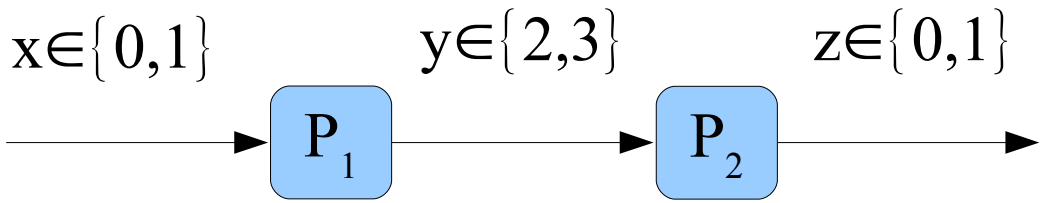


Gliederung

- Einleitung
  - Problemstellung
  - **Rea***Syn* User Interface
  - Verteilte Synthese
  - Automaten und Spiele
- Architektur- und Spezifikationstransformation
- Verteilte Spiele
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse

Architektursyntax:

```
Process P1;
Process P2;

Signal X 0 1;
Signal Y 2 3;
Signal Z 0 1;

Input  P1 X;
Output P1 Y;
Input  P2 Y;
Output P2 Z;
```



Spezifikation:

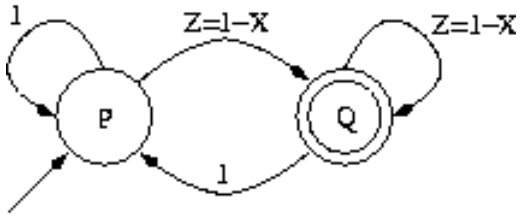
LTL Formel:

```
[]<>("Z=1-X")
```

$\Diamond(Z = 1 - X)$

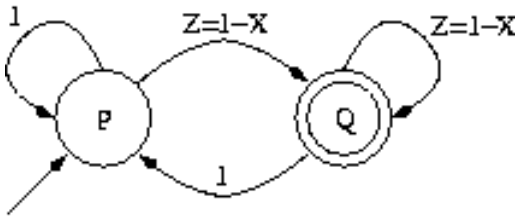
Büchi Automat:

```
State  {P,Q};
Sigma  {Z=1-X};
Init   P;
Accept Q;
Edge   P {} P;
Edge   P {Z=1-X} Q;
Edge   Q {Z=1-X} Q;
Edge   Q {} P;
```

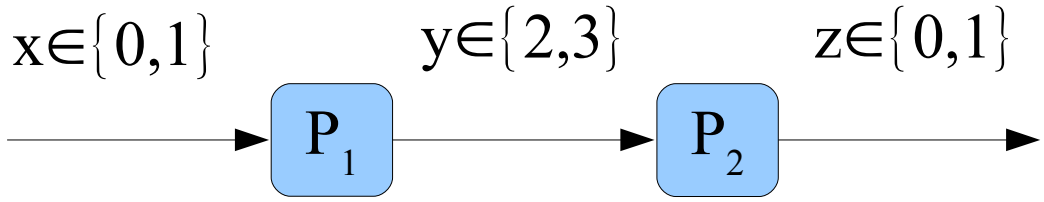


```
chan xChan = [1] of { int };
chan yChan = [1] of { int };
chan zChan = [1] of { int };

int x = 0;
int y = 0;
int z = 0;
bit check = 0;

active proctype P1() {
byte state = 0;
bit xRead = 0;
do
:: (state == 0) ->
        xRead = 0;
        do
        :: xChan?x -> xRead = 1;
        :: (xRead) -> break;
        od;
        if
        :: ((x == 0)) ->
                yChan!2;
                state = 0;
        :: ((x == 1)) ->
                yChan!3;
                state = 0;
        fi;
od
}
```

```
active proctype P2() {
byte state = 0;
bit yRead = 0;
do
:: (state == 0) ->
        yRead = 0;
        do
        :: yChan?y -> yRead = 1;
        :: (yRead) -> break;
        od;
        if
        :: ((y == 2)) ->
                zChan!1;
                state = 0;
        :: ((y == 3)) ->
                zChan!0;
                state = 0;
        fi;
od
}

active proctype env() {
do
:: xChan!1; zChan?z; check = 1; check = 0;
:: xChan!0; zChan?z; check = 1; check = 0;
od
}
```



$\Diamond(Z = 1 - X)$



0	:	2	:	1
1	:	3	:	0

Gliederung

- Einleitung
  - Problemstellung
  - **Rea***Syn* User Interface
  - Verteilte Synthese
  - Automaten und Spiele
- Architektur- und Spezifikationstransformation
- Verteilte Spiele
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse

Wie funktioniert Synthese?

- Transformation der Spezifikation
- Umwandlung der Architektur

- Erzeugung des Verteilten Spiels
- Lösen des Verteilten Spiels
- Generierung einer Strategie

- Erzeugung eines minimierten Programmes

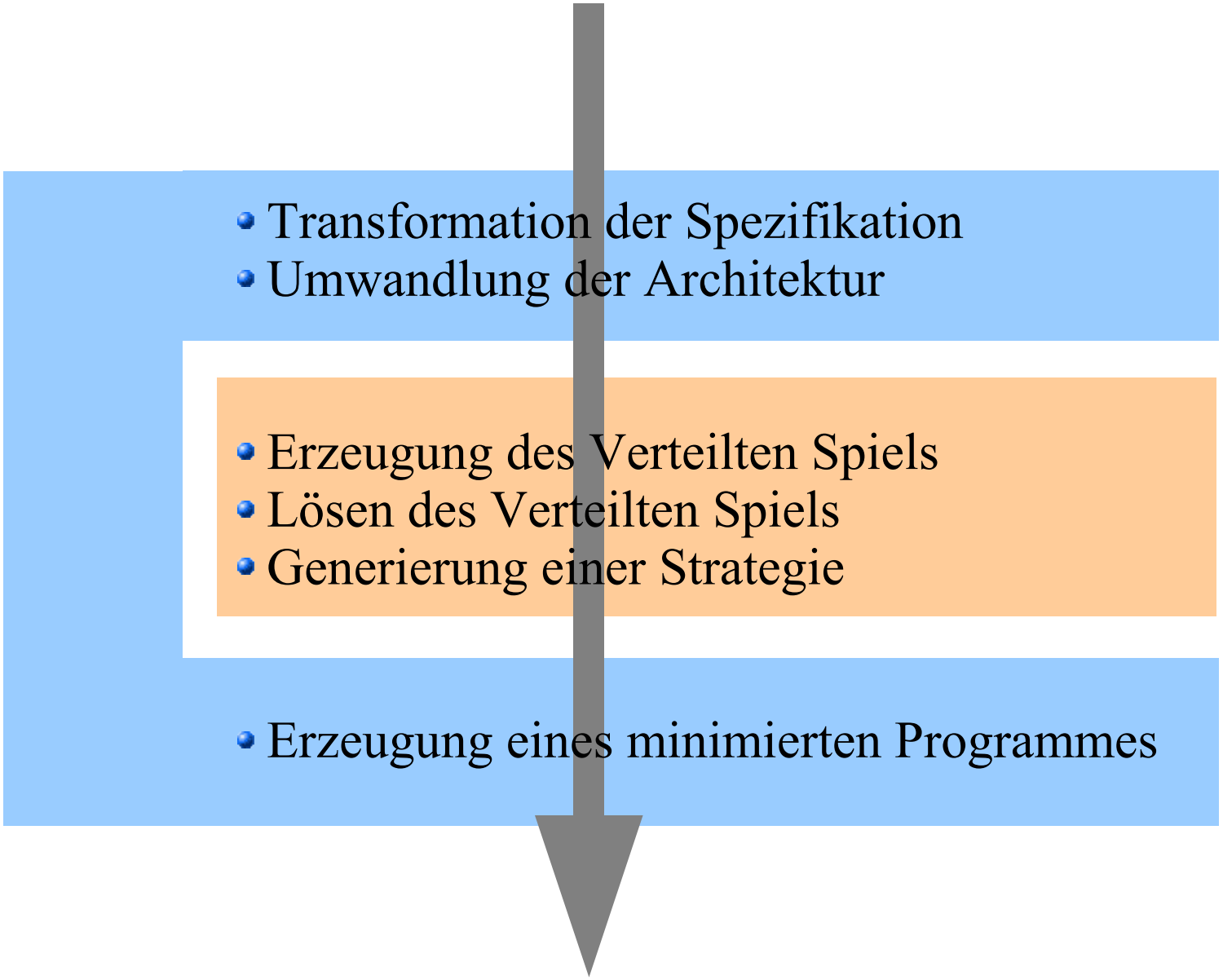
Aufteilung der Arbeiten

- Transformation der Spezifikation
- Umwandlung der Architektur

- Erzeugung des Verteilten Spiels
- Lösen des Verteilten Spiels
- Generierung einer Strategie

- Erzeugung eines minimierten Programmes

Aufteilung der Arbeiten



Synthesis of Reactive Systems Distributed Games for Reactive Systems

Gliederung

- Einleitung
  - Problemstellung
  - **Rea***Syn* User Interface
  - Verteilte Synthese
  - Automaten und Spiele
- Architektur- und Spezifikationstransformation
- Verteilte Spiele
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse

Ein nichtdeterministischer Wortautomat ist ein 5-Tupel:

$$A = (\Sigma, V, \delta, v_0, ACC)$$

- Σ ist das Eingabealphabet
- V ist die Menge der Zustände
- $\delta \in V \times \Sigma \rightarrow 2^V$ die Transitionsfuntion
- v_0 der Startzustand
- ACC ist die Akzeptanzbedingung

A ist deterministisch, wenn für alle $v \in V$, $\sigma \in \Sigma$ gilt:

$$|\delta(v,\sigma)| \leq 1$$

Ein **2-Spieler Spiel *G*** ist ein 5-Tupel:

$G = (P_0, P_1, T, v_0, WIN)$

- P_0, P_1 sind die Knotenmengen der Spieler
- $T = (P_0 \times P_1) \cup (P_1 \times P_0)$ ist die Übergangsfunktion
- $v_0 \in (P_0 \times P_1)$ der Startknoten
- WIN ist eine Gewinnbedingung

Ein **Play π** in einem Spiel ***G*** ist eine Sequenz von Knoten:

$\pi : q_0 q_1 q_2 \ldots \in V^{\omega} \cup V^{*} V_0$

- V_0 ist die Menge aller Knoten die keine Nachfolger besitzen
- Für alle $q_i q_{i+1}$ gibt es ein Tupel $(q_i, q_{i+1}) \in T$.

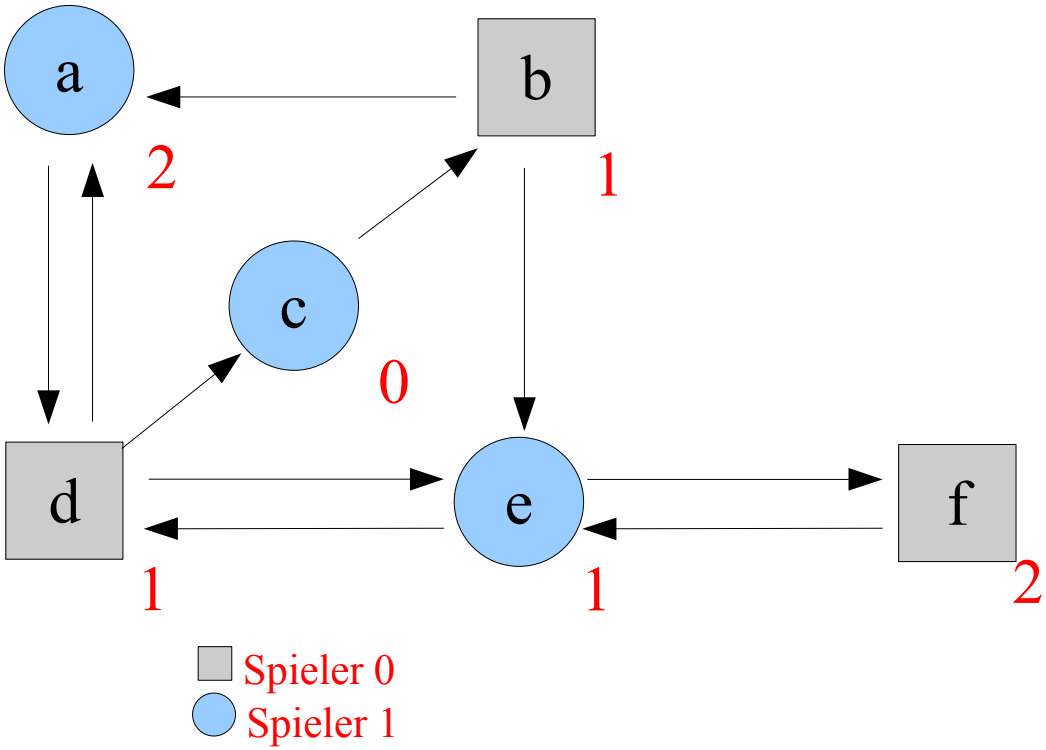
Das **Infinity Set** $\mathrm{Inf}(\pi)$ ist die Menge aller Knoten, die im Play π unendlich oft vorkommen.

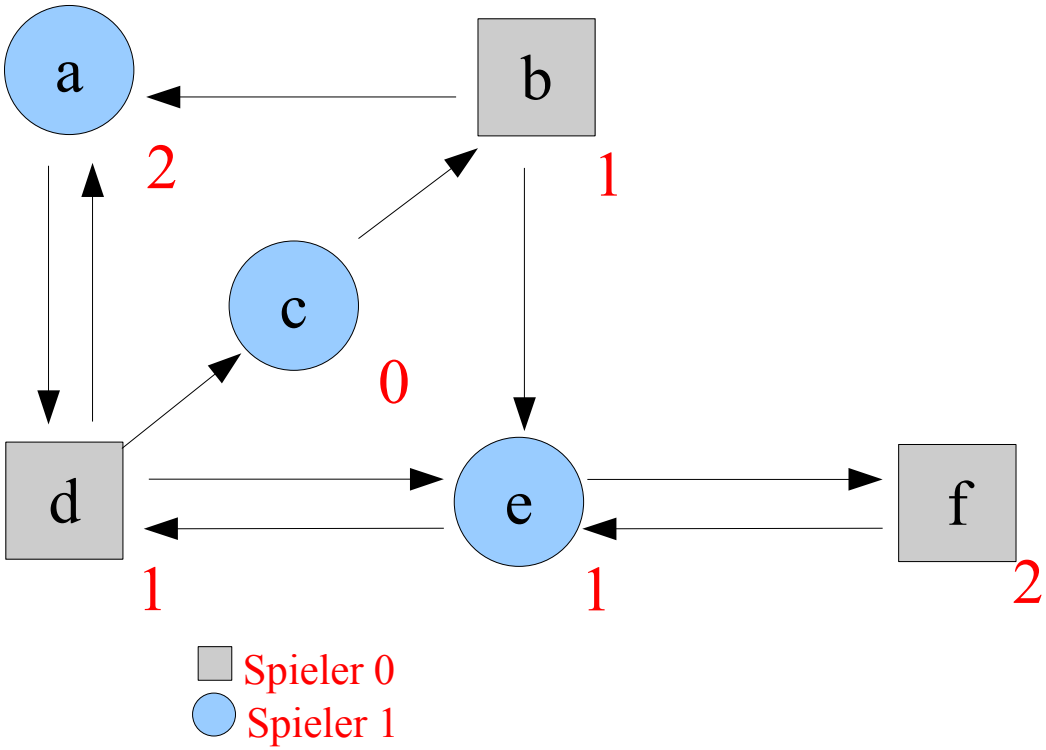
Eine **Gedächtnislose Strategie** für einen Spieler $\sigma \in \{0,1\}$ ist eine Funktion:

$$f_\sigma : \mathrm{P}_\sigma \rightarrow \mathrm{P}_{(1-\sigma)}$$

Ein **Attractor set** $\mathrm{Attr}_\sigma(\mathrm{G},\mathrm{U})$ für den Spieler σ und die Menge U ist die Menge von Knoten, von denen Spieler σ eine Gedächtnislose Strategie attr_σ besitzt, die nach endlichen Schritten innerhalb U oder in ein Dead End für Spieler $1\text{-}\sigma$ führt.

Eine **Winning Region** W_σ für Spieler σ ist die Menge aller Knoten, für die Spieler σ eine Gedächtnislose Strategie besitzt.


a
b
2
1
c
0
d
e
f
1
1
2
Spieler 0
Spieler 1

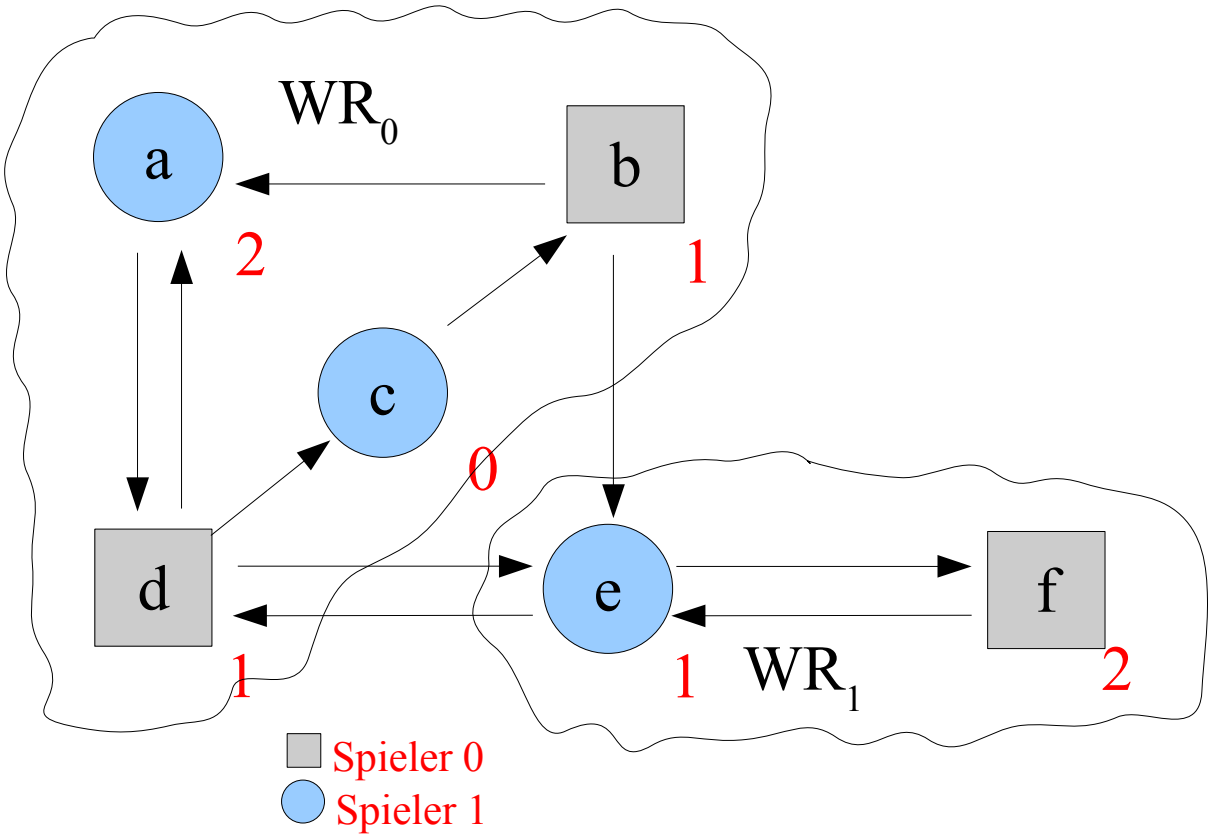


- Attractor:

$Attr_1(G,\{c\}) = \{c\}$

$Attr_0(G,\{c\}) = \{a,b,c,d\}$

Bei-Spiel



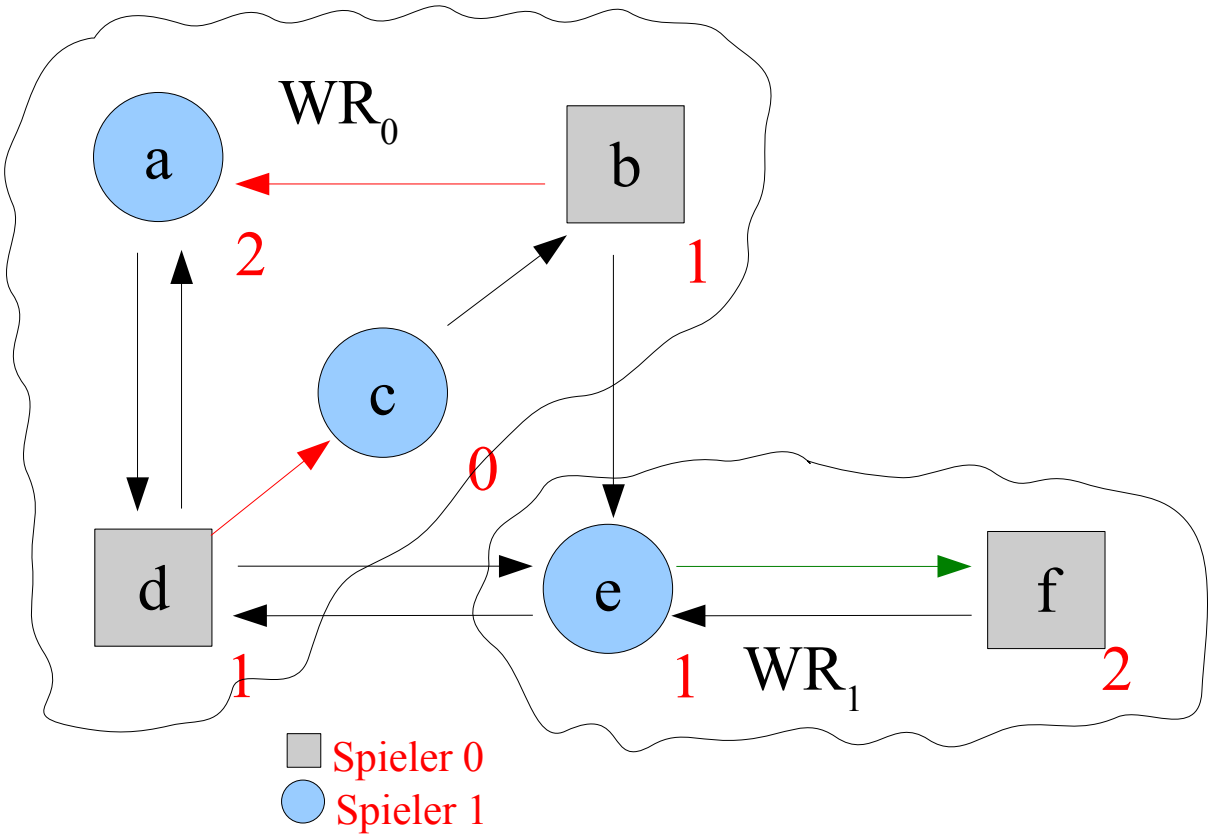
- Attractor:

$Attr_1(G,\{c\}) = \{c\}$

$Attr_0(G,\{c\}) = \{a,b,c,d\}$

- Winning Region
  (Minimale Parität)

Bei-Spiel



- Attractor:

$Attr_1(G,\{c\}) = \{c\}$
$Attr_0(G,\{c\}) = \{a,b,c,d\}$

- Winning Region
  (Minimale Parität)

- Strategie

Gewinnbedingung

Sei *ACC* eine der folgenden Gewinnbedingungen, dann ist W(*ACC*) die Menge aller unendlichen Plays π, so dass von *ACC* akzeptiert wird.

Büchi:

- $ACC = F \subseteq V$: $\pi \in W(ACC)$ gdw. $Inf(\pi) \cap F = \emptyset$

Rabin:

- $ACC = \{(G_0,R_0),(G_1,R_1),\ldots,(G_n,R_n)\}$:

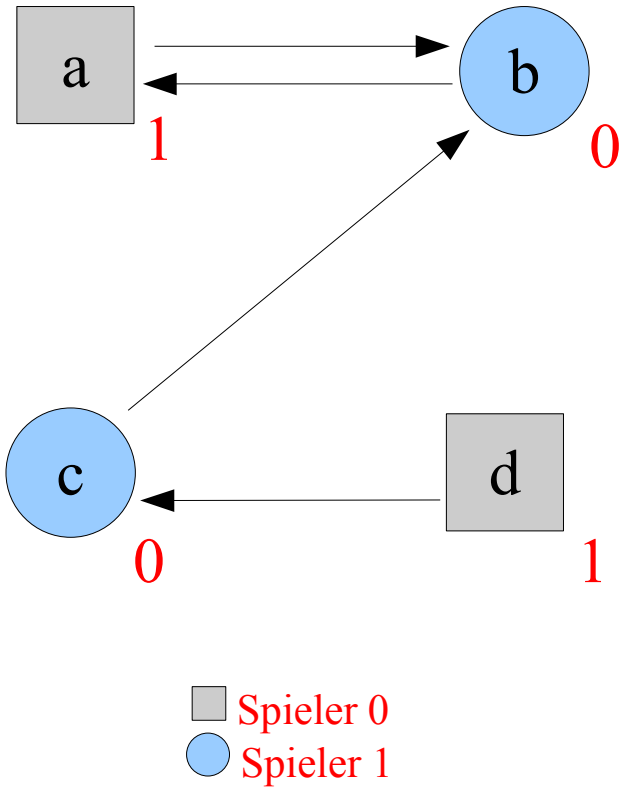
$\pi \in W(ACC)$ gdw. $\exists 0 \leq k \leq n$ sodass $Inf(\pi) \cap G_k = \emptyset$ und $Inf(\pi) \cap R_k \neq \emptyset$

Minimale Parität:

- $(\chi: V \to \mathbb{N})$: $\pi \in W(ACC) \Leftrightarrow \min(Inf(\chi(\pi)))$ gerade

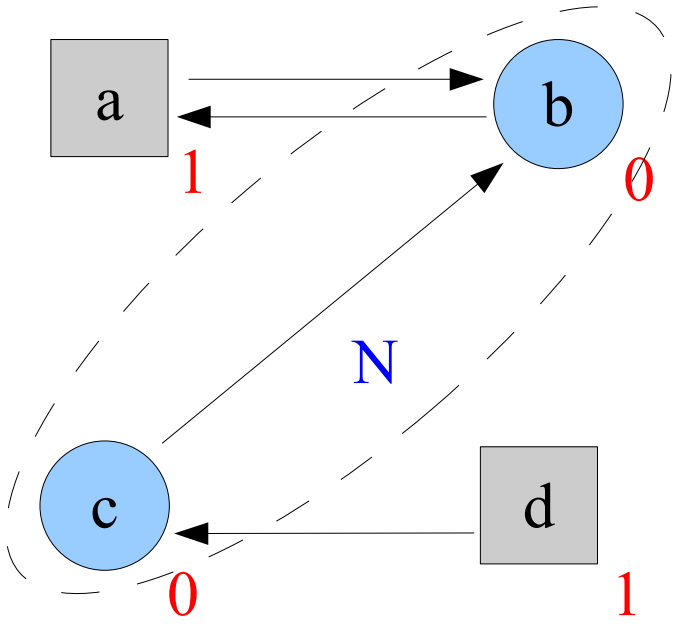
Theorem 1: Parity Spiele sind determiniert

Lösen von Parity Spielen



Spieler 0 gewinnt, wenn die kleinste unendlich oft vorkommende Zahl gerade ist.

Lösen von Parity Spielen

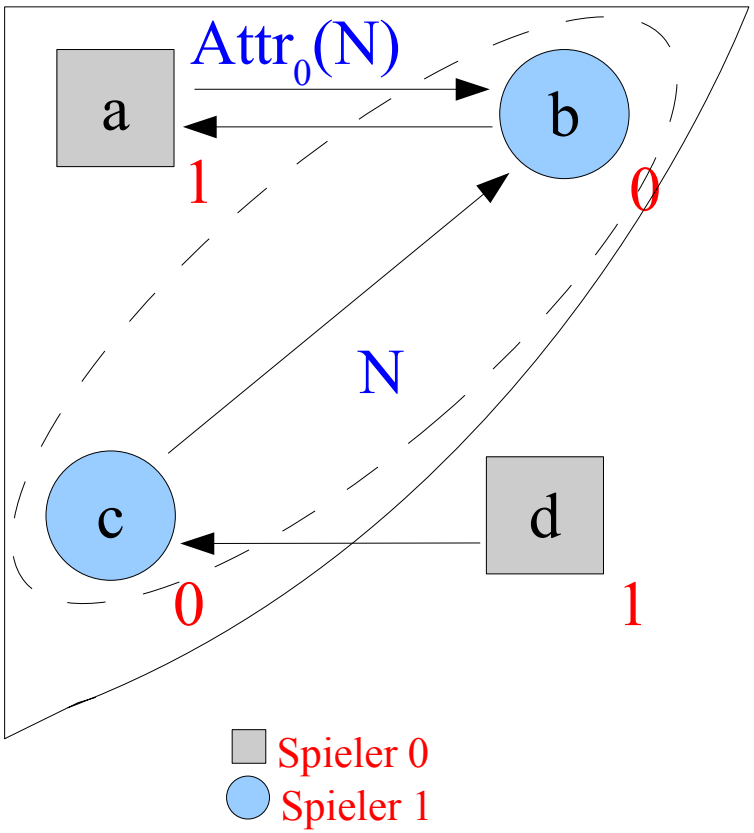


Spieler 0

Spieler 1

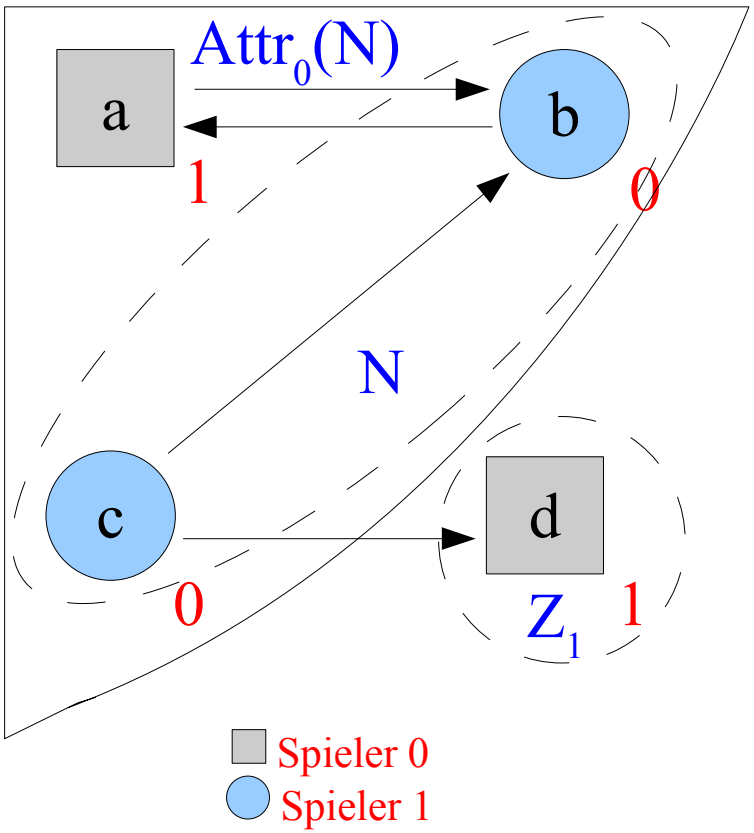
- N = {b,c}

Lösen von Parity Spielen



- N = {b,c}
- $Attr_0(G,N) = \{a,b,c\}$

Lösen von Parity Spielen

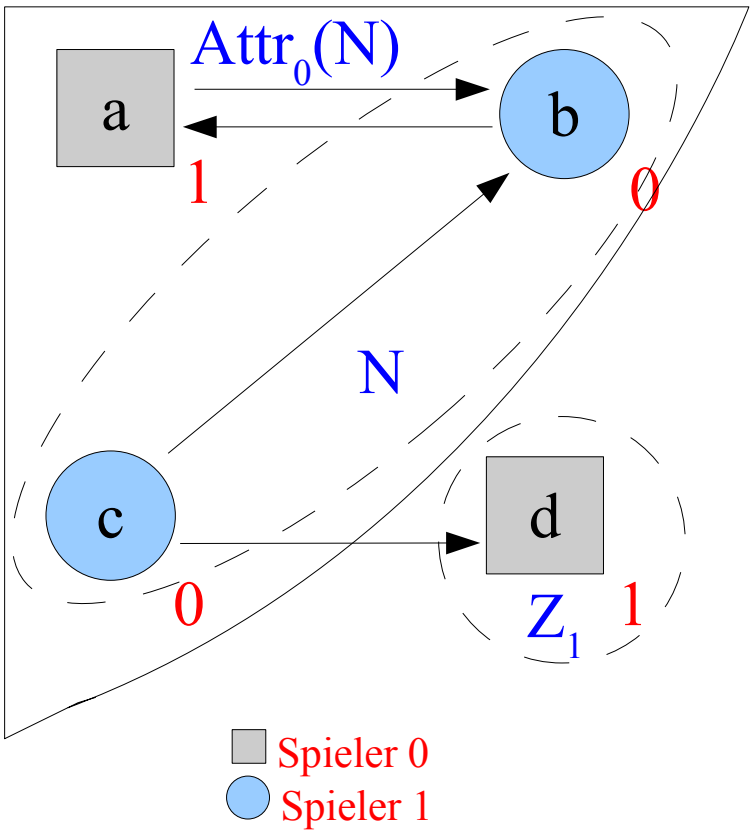


- $N = \{b,c\}$
- $Attr_0(G,N) = \{a,b,c\}$
- $Z_1 = G \setminus \{a,b,c\} = \{d\}$

  Dead End für 1 → gewinnend für 0

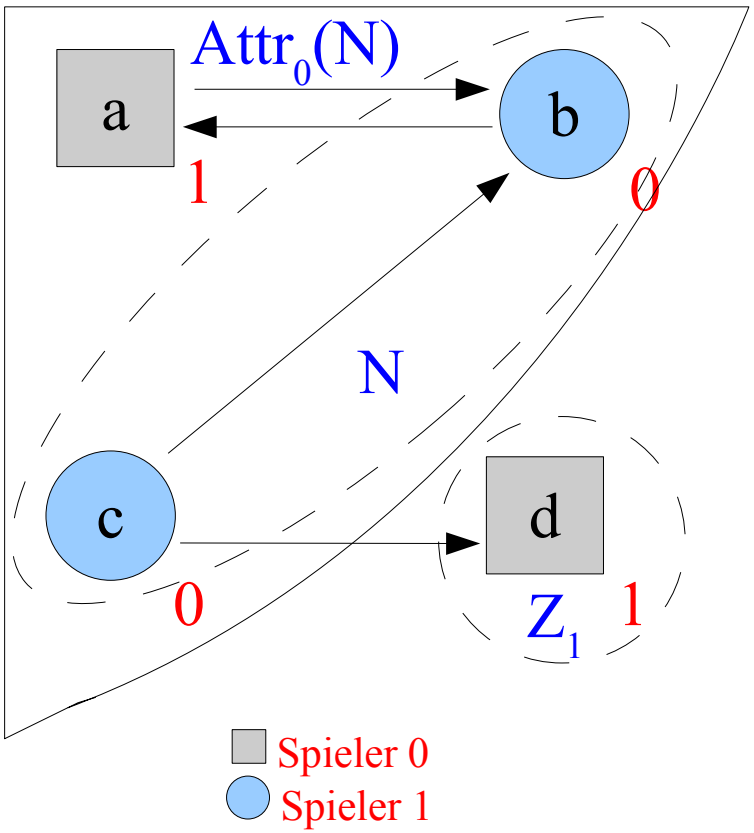
  $Attr_1(G,Z_1) = \{c,d\}$

Lösen von Parity Spielen



- $N = \{b,c\}$
- $Attr_0(G,N) = \{a,b,c\}$
- $Z_1 = G \setminus \{a,b,c\} = \{d\}$
  Dead End für 1 → gewinnend für 0
  $Attr_1(G,Z_1) = \{c,d\}$
- $Attr_0(G,N) \setminus Attr_1(Z_1) = \{a,b\}$
- Algorithmus rekursiv anwenden.
  $N = \{b\}$
  $Attr_0(\{a,b\},\{b\}) = \{a,b\}$

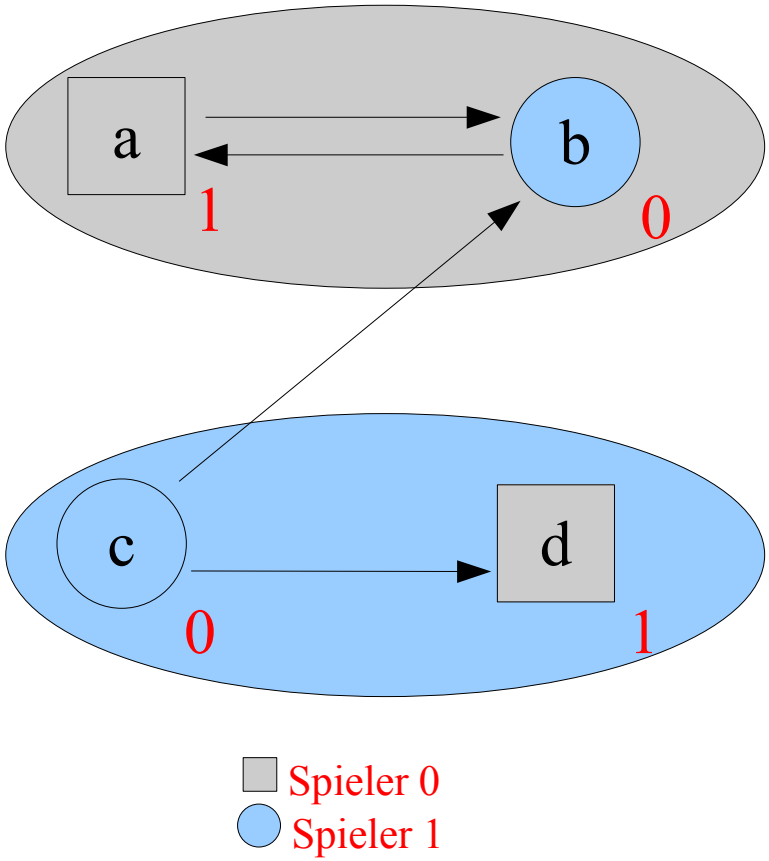
Lösen von Parity Spielen



- $N = \{b,c\}$
- $Attr_0(G,N) = \{a,b,c\}$
- $Z_1 = G \setminus \{a,b,c\} = \{d\}$
  Dead End für 1 → gewinnend für 0
  $Attr_1(G,Z_1) = \{c,d\}$
- $Attr_0(G,N) \setminus Attr_1(Z_1) = \{a,b\}$
- Algorithmus rekursiv anwenden.
  $N = \{b\}$
  $Attr_0(\{a,b\},\{b\}) = \{a,b\}$
- Z_2 ist leer. Algorithmus stoppt.

$WR_0 = Attr_0(\{a,b\},\{b\}) = \{a,b\}$ $WR_1 = Attr_1(Z_1) = \{c,d\}$

Lösen von Parity Spielen



Winning Region für Spieler 0

Winning Region für Spieler 1

$WR_0 = Attr_0(\{a,b\},\{b\}) = \{a,b\}$ $WR_1 = Attr_1(Z_1) = \{c,d\}$

Gliederung

- Einleitung
- Architektur- und Spezifikationstransformation
  - Transformation von Architekturen
  - Transformation von Automaten
  - Transformation der Spezifikation
  - Optimierungen
- Verteilte Spiele
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse

Architektur und Spezifikation

- Architektur ohne Information Fork wird in Pipeline Architektur übersetzt
- Pipeline Architektur wird in Prozess Spiele transformiert
- Prozess Spiele werden benutzt um ein Verhalten des verteilten Systems auszuspielen

- Spezifikation wird in deterministischen Automaten übersetzt.
- Anschliessend in ein Spiel das den Automaten simuliert.
- Spezifikationsspiel verifiziert das von den Architektur Spielen ausgespielte Verhalten

Eine Architektur modelliert die Kommunikation zwischen einzelnen Prozessen. Formal ist eine Architektur ein Tupel

$$\boldsymbol{A} = (P, S, E_S, p_{env}, I, O)$$

- P ist eine Menge von Prozessen
- S ist eine Menge von Signalen
- $E_S \subseteq (P \times S \times P)$ Kanten mit Signalen makiert
- $p_{env} \in P$ ist der Umgebungsprozess
- $I: P \rightarrow 2^S$ Eingangsfunktion
- $O: P \rightarrow 2^S$ Ausgangsfunktion

Eigenschaften einer Architektur $\boldsymbol{A} = (P, S, E_S, p_{env}, I, O)$

- $(p_1, s, p_2) \in E_S \Leftrightarrow s \in O(p_1) \wedge s \in I(p_2)$
- $S = \cup_{p \in P} O(p)$
- $p_1 \neq p_2 \Rightarrow O(p_1) \cap O(p_2) = \emptyset$

Weitere Definitionen:

- $WRITE(s) := \{p \in P | s \in O(p)\}$
- $READ(s) := \{p \in P | s \in I(p)\}$
- $P^- := \{p \in P | O(p) \neq \emptyset\}$
- dom(s): Menge der Werte die s annehmen kann

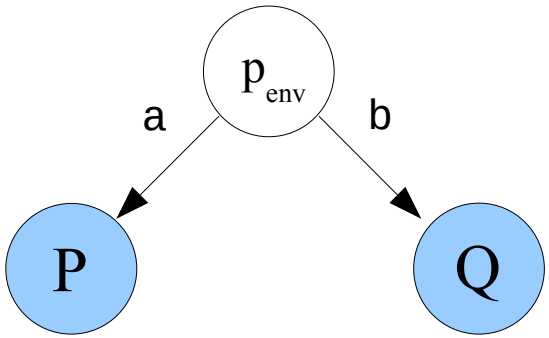
Information Forks

Information Fork [Finkbeiner, Schewe]:

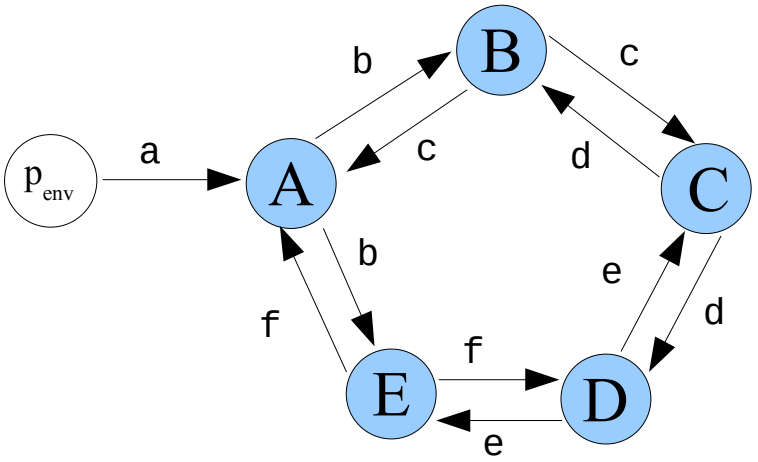
Zwei Prozess erhalten unvergleichbare Eingaben, die nicht von den Prozessen abgelitten werden können.

Alle Architekturen ohne Information Fork werden von **Rea***Syn* in eine Pipeline Architektur übersetzt werden.

Information Fork: Beispiele



Architektur A_0

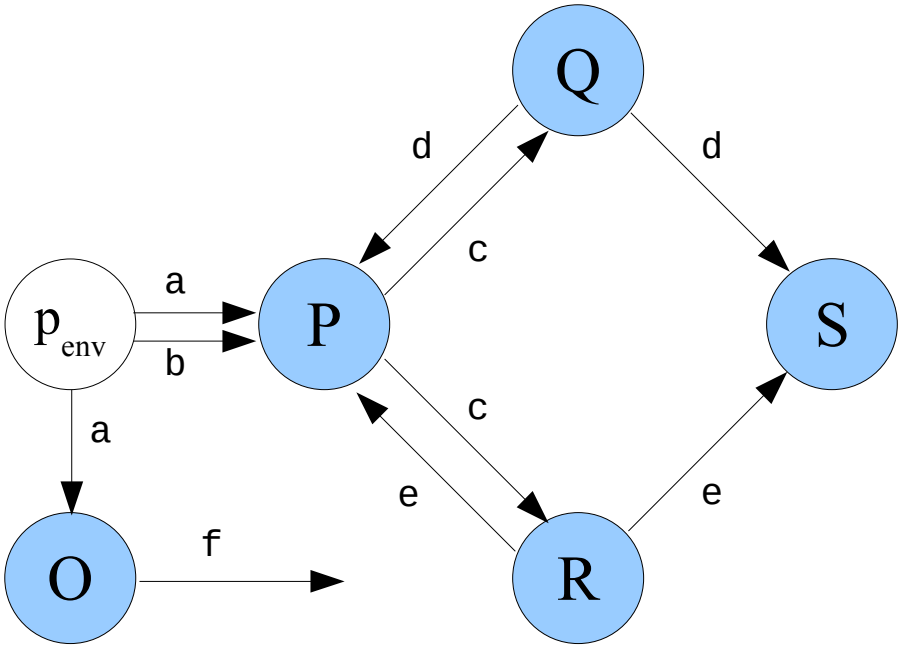


5 Prozess bidirektionaler Ring

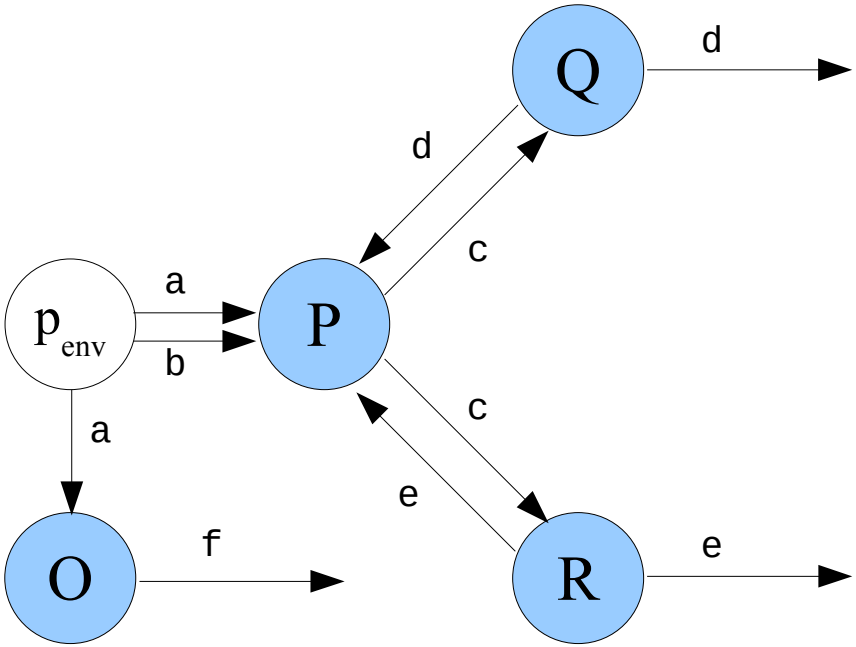
idlefree(A) [Finkbeiner,Schewe]

- $P' = P^{-} \cup p_{env}$
- $S' = S$
- $E'_S = E_S \cap (P' \times S \times P')$
- $I' = I|_{P'}$
- $O' = O|_{P'}$

Idlefree Architektur: Beispiel



Idlefree Architektur: Beispiel



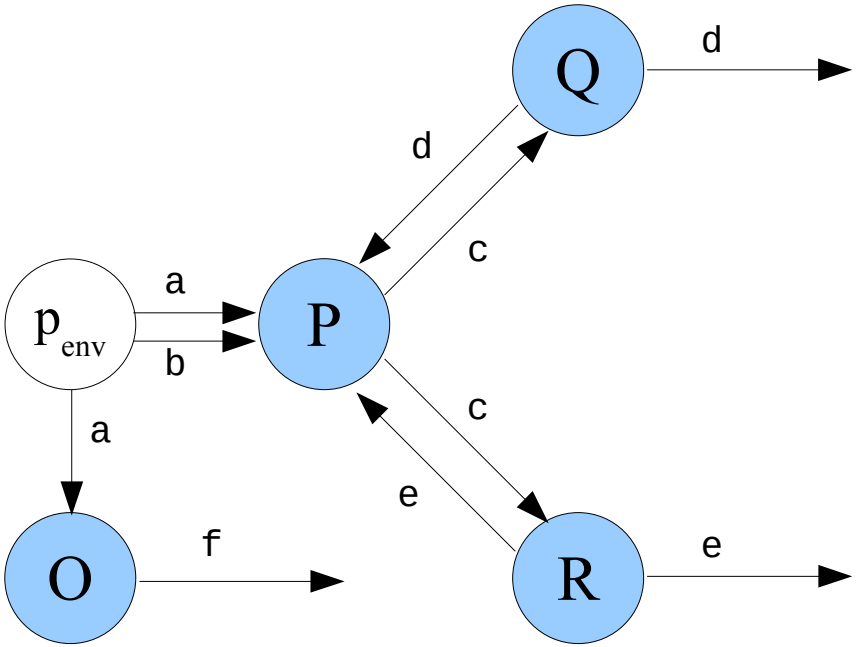
Äquivalenzrelation:

$$p \sim p' \Leftrightarrow p \preccurlyeq p' \wedge p' \preccurlyeq p$$

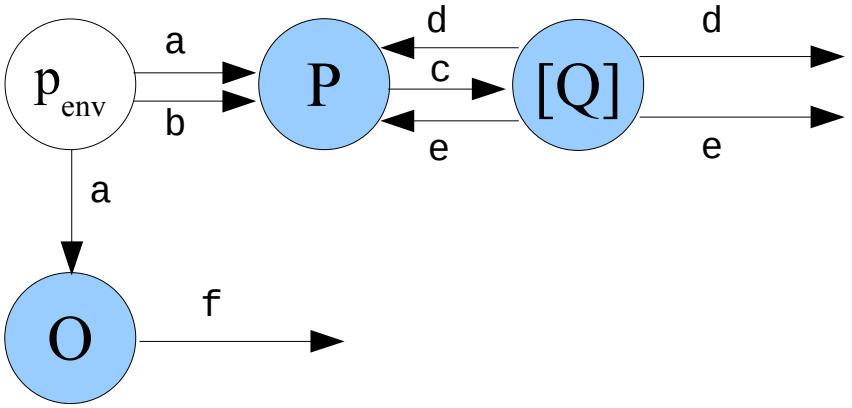
quot(A) [Finkbeiner,Schewe]

- $P' = (P^{-}/\sim) \cup p_{env}$
- $S' = S$
- $E'_s = \{([p], s, [p']) \mid (p, s, p') \in E_s \wedge [p] \neq [p']\}$
- $I'(p) = \{s \in S \mid \exists p' \in P.p \sim p' \wedge s \in I(p')\}$
- $O'(p) = \{s \in S \mid \exists p' \in P.p \sim p' \wedge s \in O(p')\}$

Quotienten Architektur: Beispiel



Quotienten Architektur: Beispiel

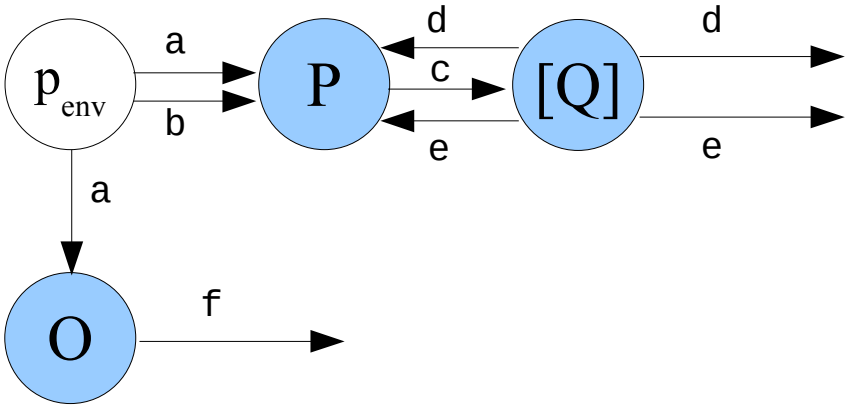


Azyklische Architektur

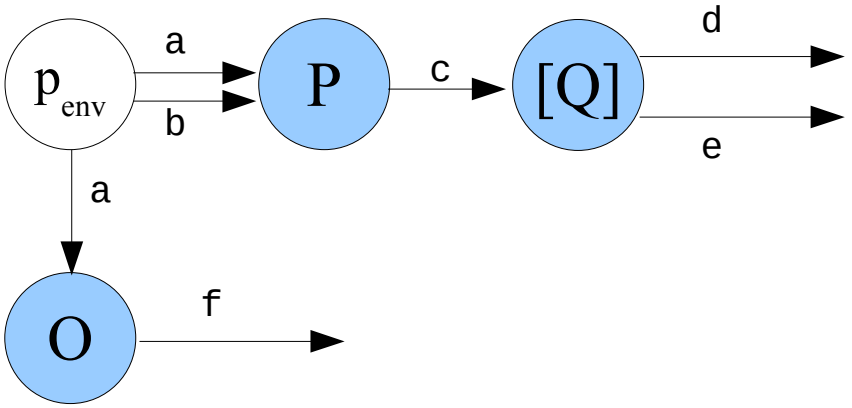
acycle(A) [Finkbeiner,Schewe]

- $P' = P$
- $S' = S$
- $E'_s = \{(p, s, p') \in E_s \mid p \leqslant p'\}$
- $I'(p) = \{s \in I(p) \mid \forall q \in \mathrm{WRITE}(s).\ q \leqslant p\}$
- $O' = O$

Azyklische Architektur: Beispiel



Azyklische Architektur: Beispiel



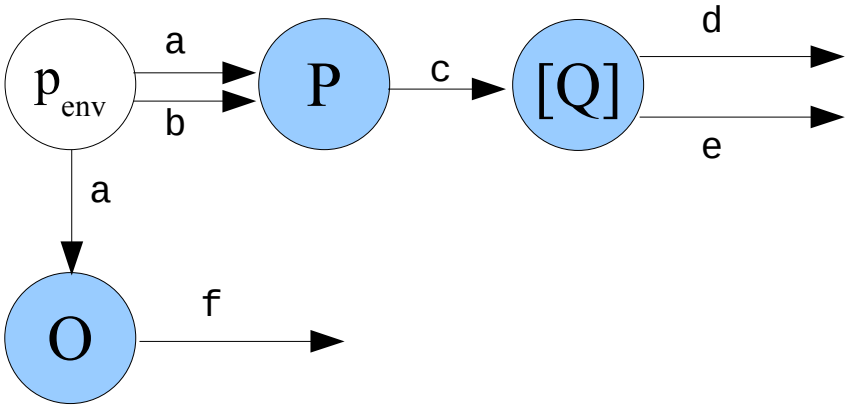
$$n_p := |\{q \in P \mid q \preccurlyeq p\}|$$

$$s_I := \{s_i \mid s \in S \wedge \min\{n_p \mid p \in \mathrm{WRITE}(s)\} < i \leq \max\{n_p \mid p \in \mathrm{READ}(s)\}\}$$

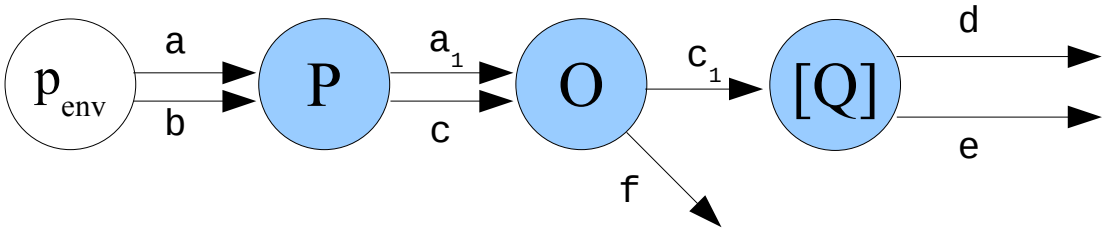
*pipe(**A**)*

- $P' = P$
- $S' = S$
- $E'_s = \{(p, s, p') \in E_s \mid p \preccurlyeq p'\}$
- $I'(p) = \{s \in I(p) \mid \forall q \in \mathrm{WRITE}(s).\ q \preccurlyeq p\}$
- $O' = O$

Pipeline Architektur: Beispiel



Pipeline Architektur: Beispiel



Pipeline nach Parity Spiel

- Ein Spiel pro Prozess in der Pipeline
- P_0 wählt die zu berechnende Funktion des Prozesses
- P_1 wählt den Eingabewert des Prozesses

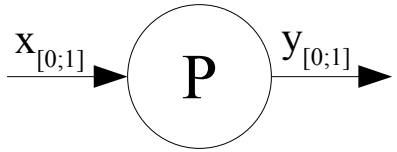
Prozess ***P*** mit Eingabedomain ***I*** und Ausgabedomain ***O***

$$\boldsymbol{G_P} = (P_0, P_1, T, v_{init}, \chi)$$

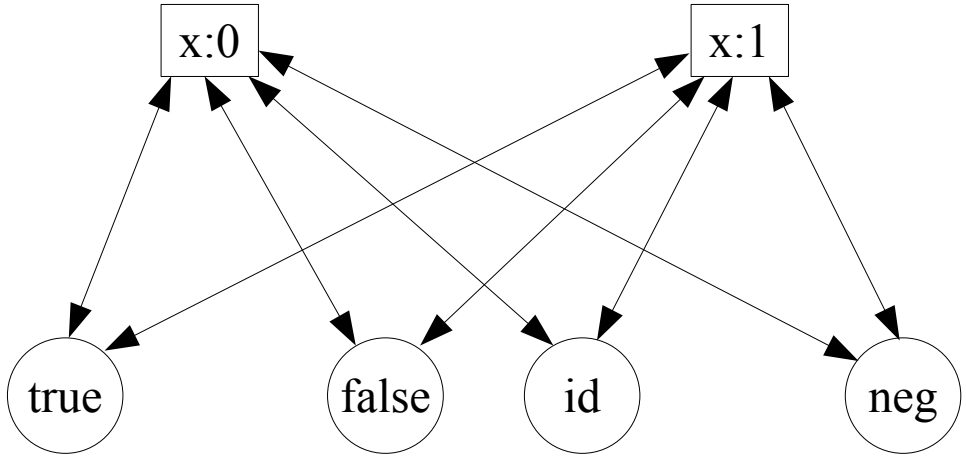
- $P_0 = \boldsymbol{I}$
- $P_1 = \{f \mid f: \boldsymbol{I} \rightarrow \boldsymbol{O}\}$
- $T = P_0 \times P_1$
- $v_{init} = i \in \boldsymbol{I}$

Pipeline nach Parity Spiel: Beispiel

Prozess ***P***



Prozess Spiel G_P



P_0 Knoten

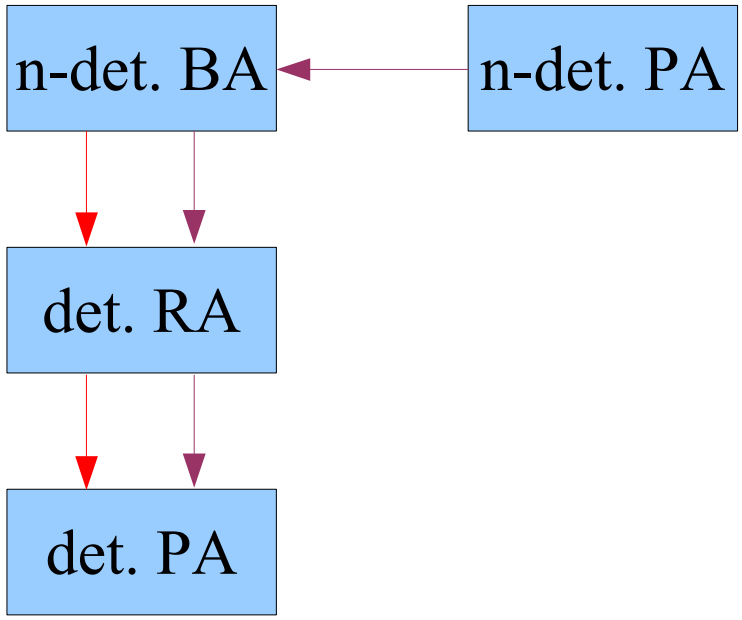
P_1 Knoten

Gliederung

- Einleitung
- Architektur- und Spezifikationstransformation
  - Transformation von Architekturen
  - Transformation von Automaten
  - Transformation der Spezifikation
  - Optimierungen
- Verteilte Spiele
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse

Zwei Transformationsketten:

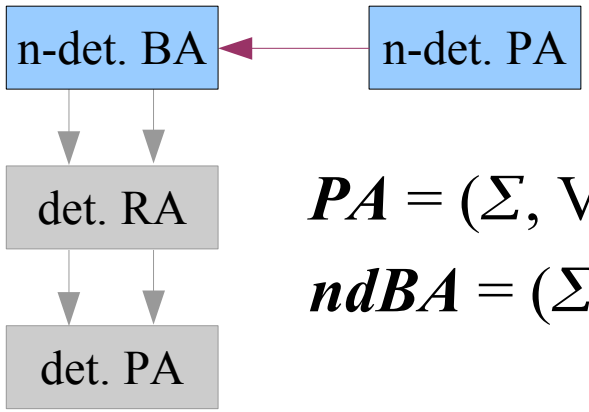
- Transformation der Spezifikation
- Transformation von Gewinnbedingungen



Spezifikation

Gewinnbedingung

min. PA nach n-det. BA

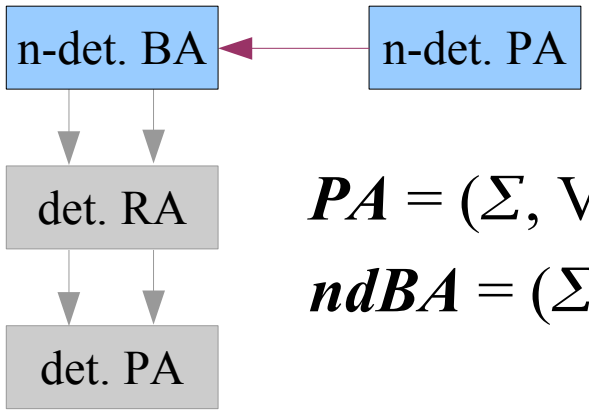


$\boldsymbol{PA} = (\Sigma, V, \delta, v_{init}, \alpha)$

$\boldsymbol{ndBA} = (\Sigma, V', \delta', v_{init}, F)$

- Büchi Automat rät die minimale Parität ***p*** und den Zeitpunkt ab dem ***p*** die minimale auftretende Parität ist.
- Wechsel in eine Kopie des parity Automaten, in dem nur noch Paritäten $\geq \boldsymbol{p}$ vorkommen.
- $S = \{\alpha(v) \mid v \in V\} \cap 2\mathbb{N}$ Menge der in PA vorkommenden geraden Paritäten

min. PA nach n-det. BA

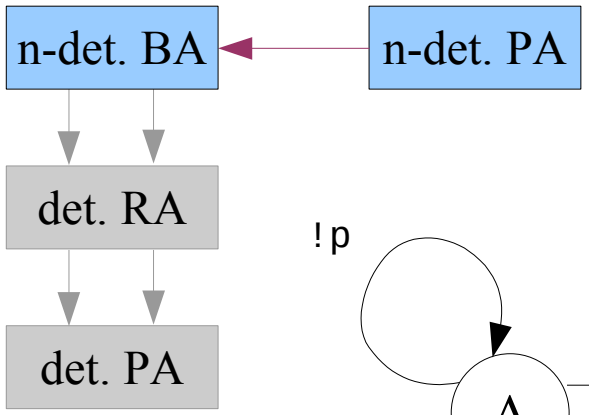


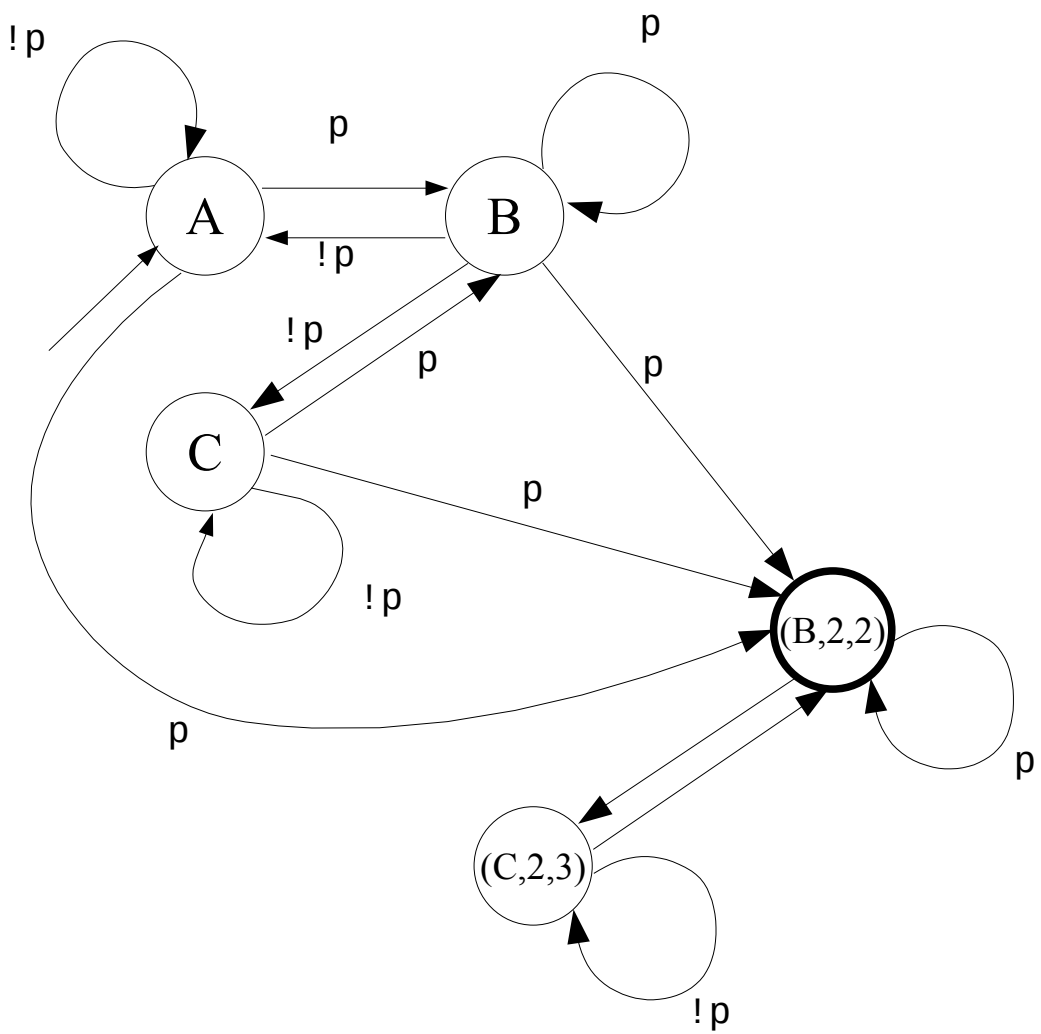
$\boldsymbol{PA} = (\Sigma, V, \delta, v_{init}, \alpha)$

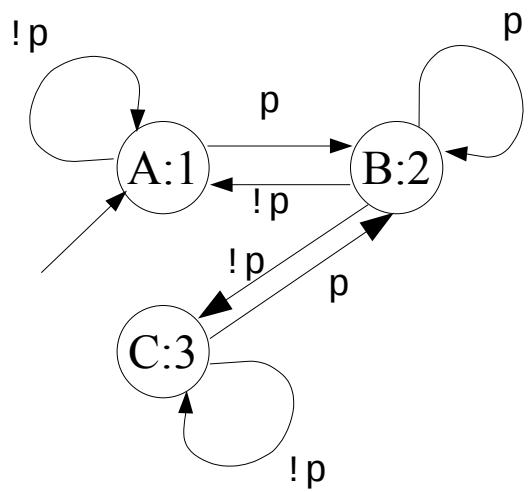
$\boldsymbol{ndBA} = (\Sigma, V', \delta', v_{init}, F)$

- $V' = V \cup \bigcup_{i \in S} \{(v, i, j) \mid v \in V \wedge \alpha(v) = j \wedge i \leq j\}$
- $\delta'(v, \sigma) = \delta(v, \sigma) \cup \{(v', i, i) \mid v' \in \delta(v, \sigma) \wedge \alpha(v') = i \wedge i \bmod 2 = 0\}$
- $\delta'((v, i, j), \sigma) = \{(w, i, k) \mid \alpha(w) = k \wedge i \leq k \wedge w \in \delta(v, \sigma)\}$
- $F = \{(v, i, i) \mid v \in V\}$

min. PA nach n-det. BA: Beispiel

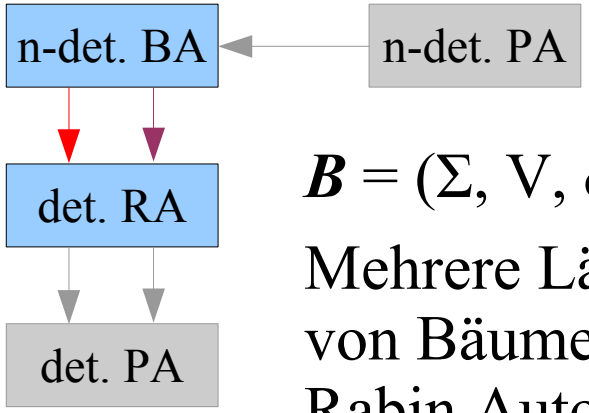






Parity Automat

Safra's Konstruktion

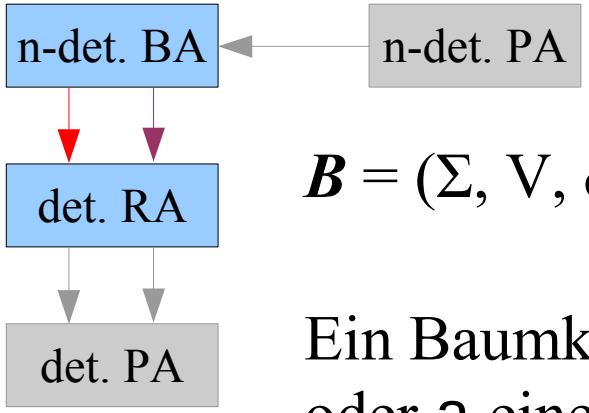


$\boldsymbol{B} = (\Sigma, V, \delta, v_0, F)$ $\boldsymbol{R} = (\Sigma', V', \delta', v_{0'}, R)$

Mehrere Läufe des Büchi Automaten werden mit Hilfe von Bäumen dargestellt. Ein Baum ist ein Knoten des Rabin Automaten mit den Eigenschaften:

- Jeder Baumknoten ist entweder weiß oder grün
- Knoten sind nach ihrem Alter sortiert
- Knoten sind mit $U \subseteq V$ markiert.
  - Die Vereiningung der Markierungen der Nachfolger eines Knotens v sind eine echte Teilmenge der Markierung von v
  - Knoten die nicht auf einem Pfad des Baums sind haben disjunkte Markierungen
- Jeder Knoten hat einen eindeutigen Namen zwischen 0 und $2|V|$.

Safra's Konstruktion



$\boldsymbol{B} = (\Sigma, V, \delta, v_0, F)$ $\boldsymbol{R} = (\Sigma', V', \delta', v'_{0'}, C)$

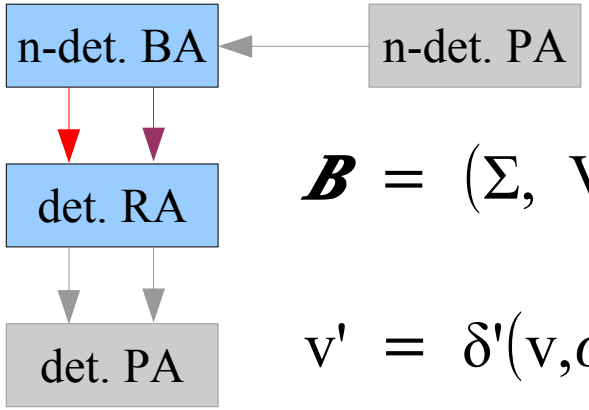
Ein Baumknoten `a` ist links von `b`, wenn `a` älter ist als `b` oder `a` einen Vorgänger hat der älter ist als ein Vorgänger von `b`.

- $\Sigma' = \Sigma$
- V': Menge von Bäumen mit den 4 Eigenschaften
- v'_0: Baum mit einem weißen Knoten mit $\{v_0\}$ markiert
- $C = \{(G_i, R_i) \mid G_i \in V', R_i \in V'\}$

  G_i: Menge der Bäume mit einem grünen Knoten namens `i`.

  R_i: Menge der Bäume ohne einem Knoten namens `i`.

Safra's Konstruktion



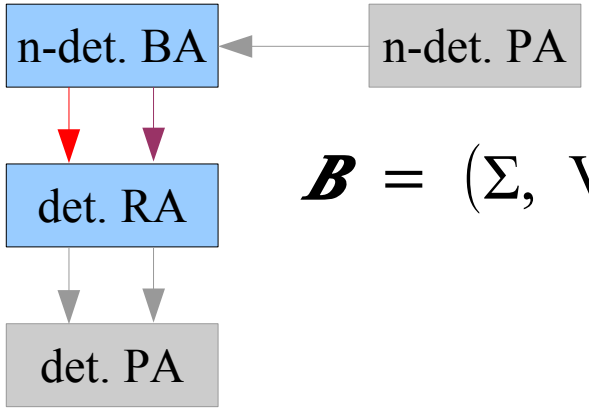
$\boldsymbol{B} = (\Sigma,\ V,\ \delta,\ v_0,\ F)$ $\qquad$ $\boldsymbol{R} = (\Sigma',\ V',\ \delta',\ v'_{0'},\ C)$

$v' = \delta'(v,\sigma)$ wird in 8 Schritten berechnet:

(1)Kopiere den Baum von v nach v'
(2)Färbe alle Baumknoten von v' weiß
(3)Ersetze alle Markierungen U durch $\delta'(U,\sigma)$
(4)Für jeden Baumknoten markiert mit U erzeuge neuen Nachfolger markiert mit $U \cap F$.

Diese Schritte können die geforderten Eigenschaften des Baumes verletzen. (5) – (8) stellen die Eigenschaften wieder her.

Safra's Konstruktion



$\boldsymbol{B} = (\Sigma,\ V,\ \delta,\ v_0,\ F)$ $\qquad$ $\boldsymbol{R} = (\Sigma',\ V',\ \delta',\ v'_{0'},\ C)$

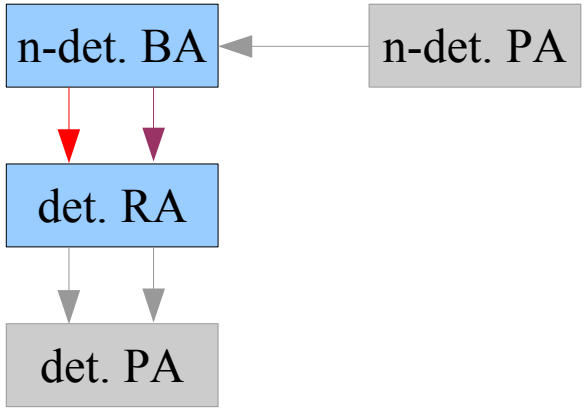
(5) Für jedes $u \in U$ eines Baumknoten b entferne u aus allen Baumknoten links von b.

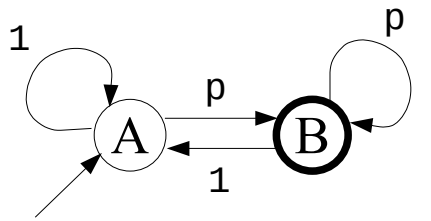
(6) Entferne alle Baumknoten mit leeren Markierungen

(7) Wird die Markierung von den Markierungen der Nachfolger subsumiert, entferne Nachfolger und färbe den Baumknoten grün

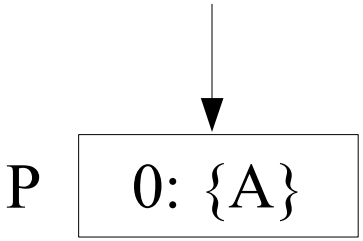
(8) Jeder neue Knoten bekommt einen Namen der noch nicht im Baum vorgekommen ist.

Safra's Konstruktion: Beispiel

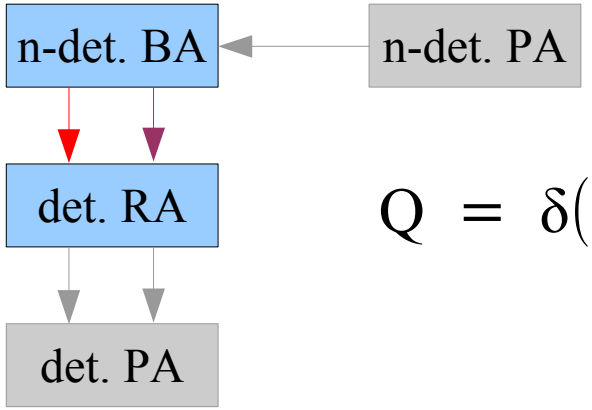




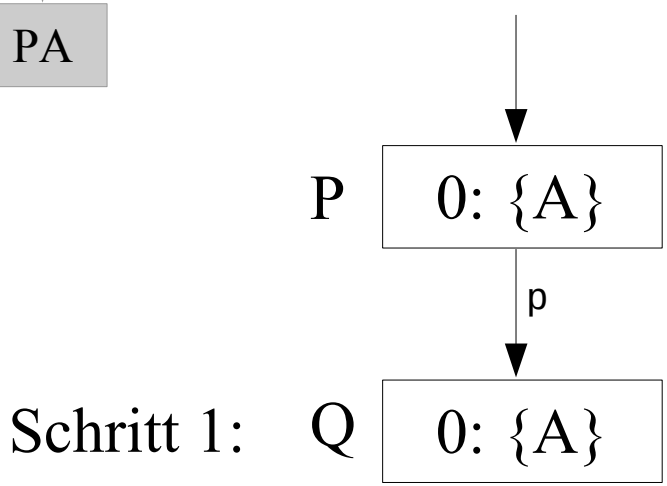
Büchi Automat

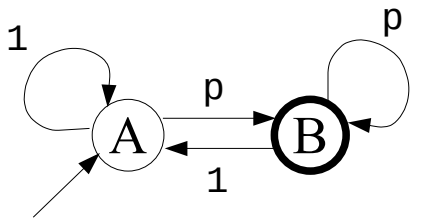


Safra's Konstruktion: Beispiel



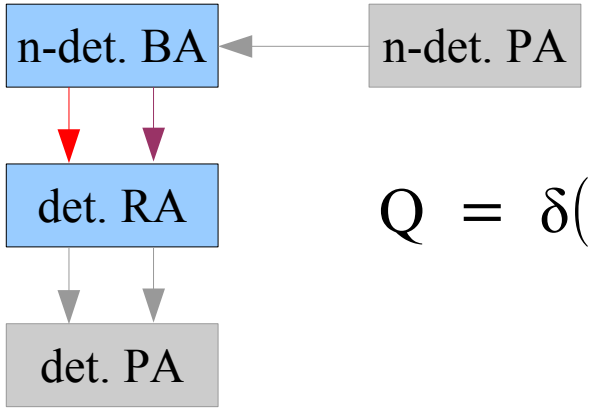
$Q = \delta(P,p)$



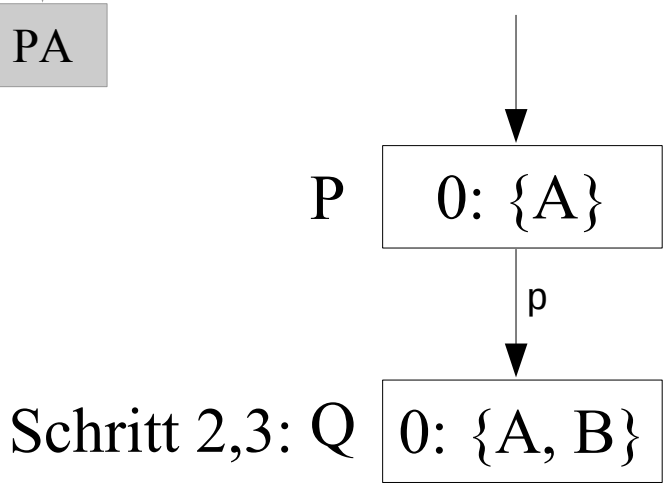


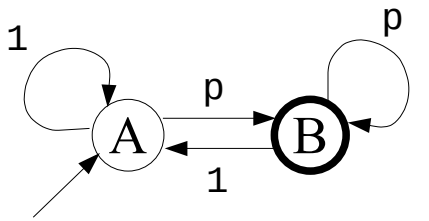
Büchi Automat

Safra's Konstruktion: Beispiel



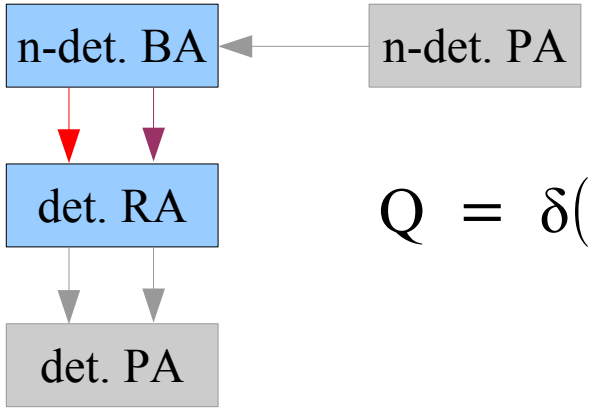
$Q = \delta(P,p)$



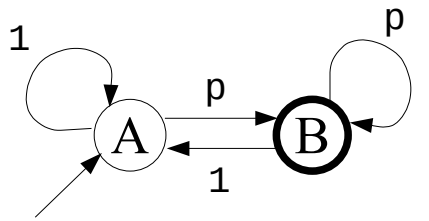


Büchi Automat

Safra's Konstruktion: Beispiel

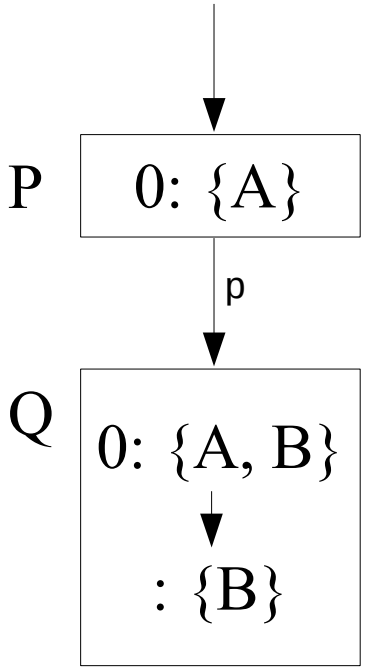


$Q = \delta(P,p)$

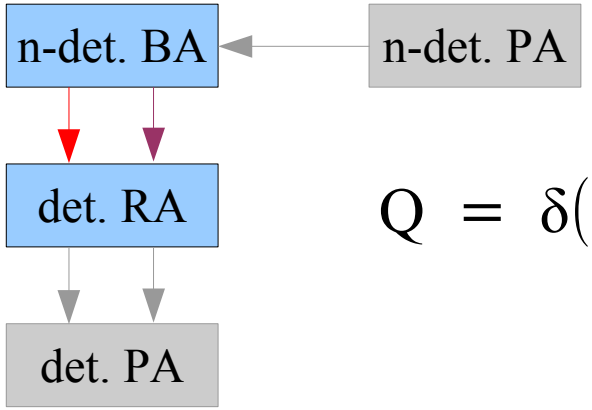


Büchi Automat

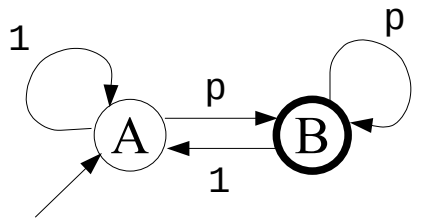
Schritt 4:



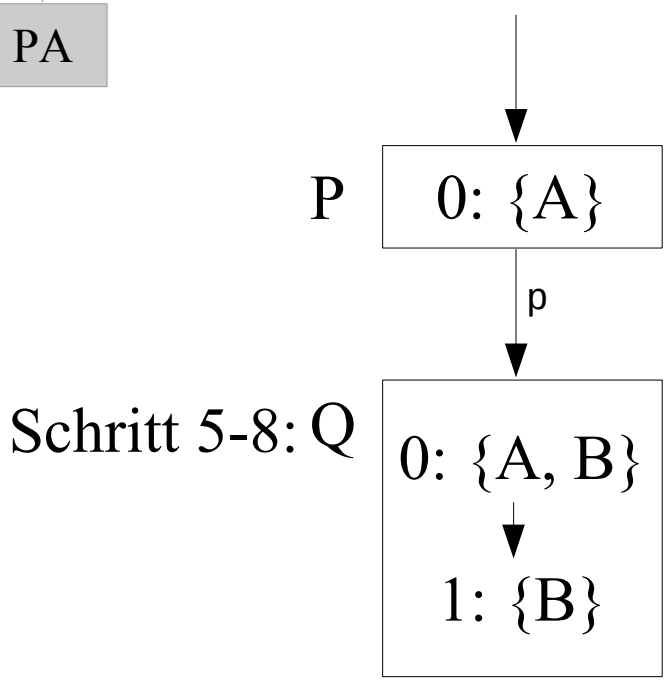
Safra's Konstruktion: Beispiel



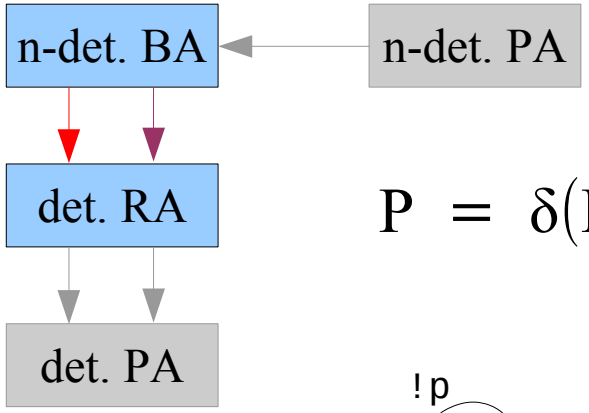
$Q = \delta(P,p)$



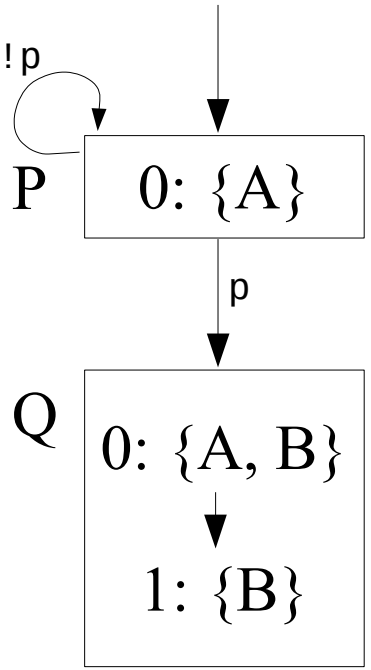
Büchi Automat

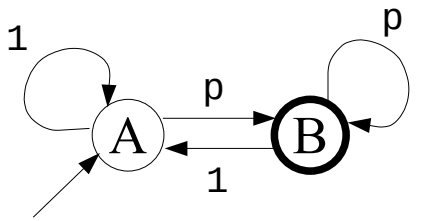


Safra's Konstruktion: Beispiel



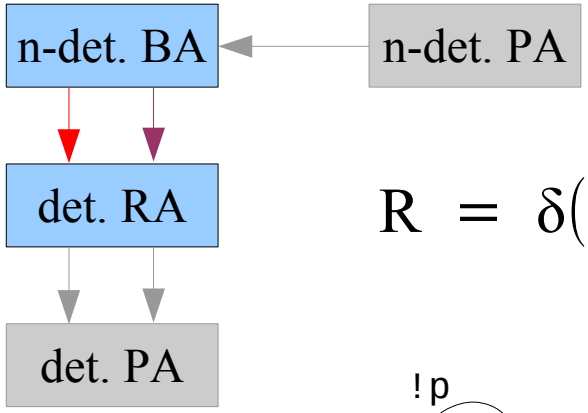
$$P = \delta(P, !p)$$



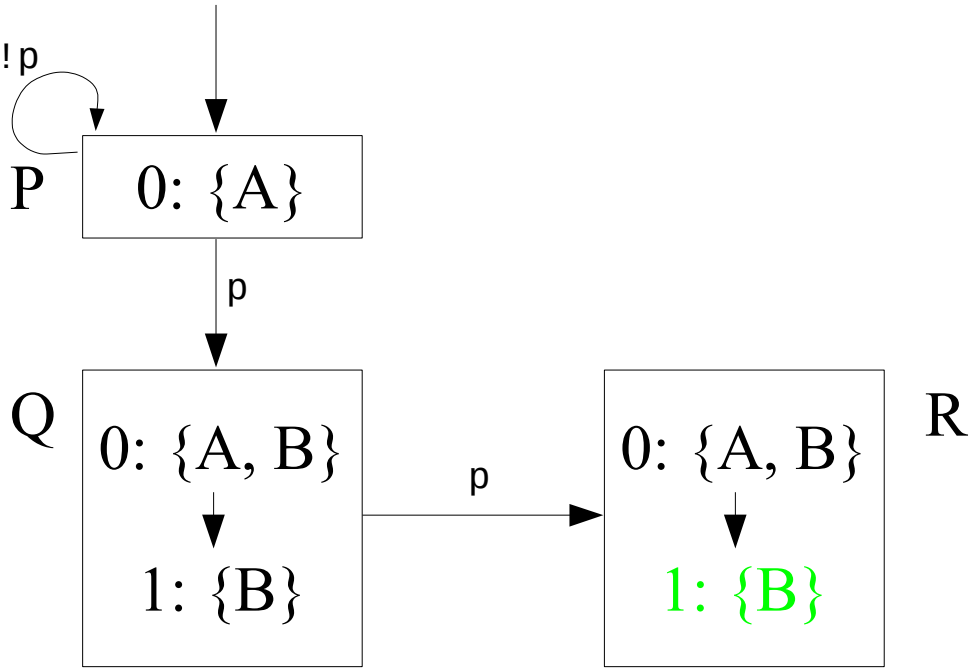


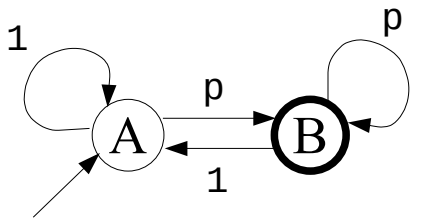
Büchi Automat

Safra's Konstruktion: Beispiel



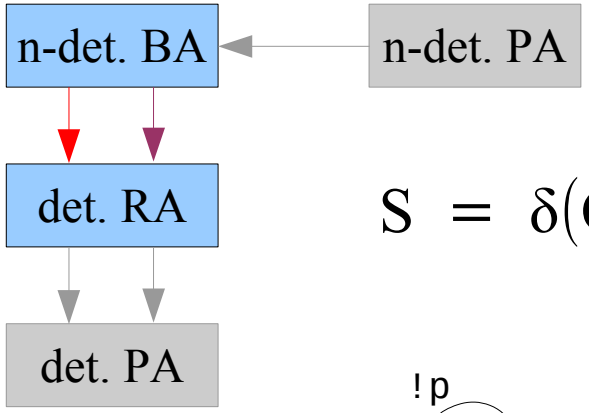
$R = \delta(Q,p)$



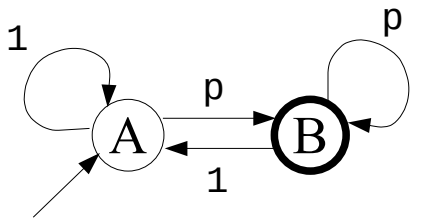


Büchi Automat

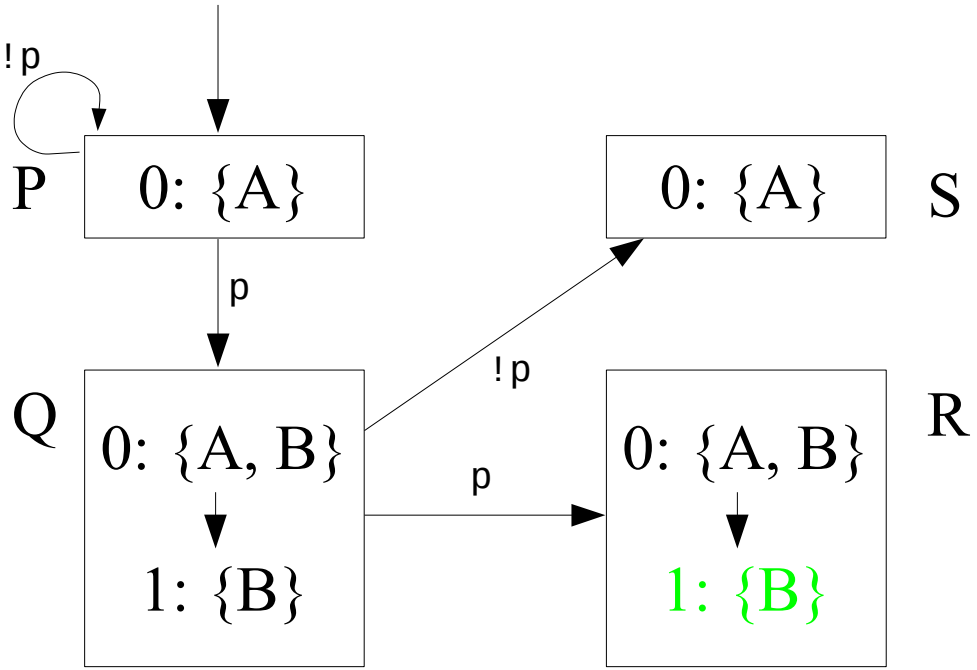
Safra's Konstruktion: Beispiel



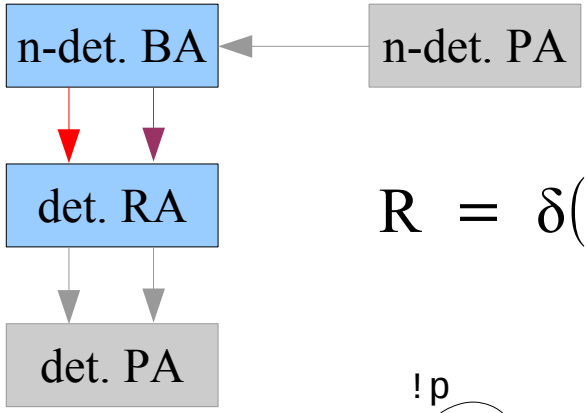
$S = \delta(Q,!p)$



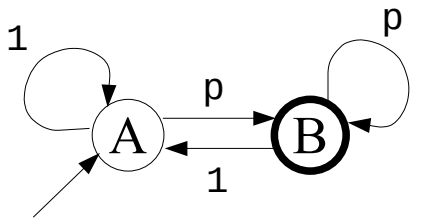
Büchi Automat



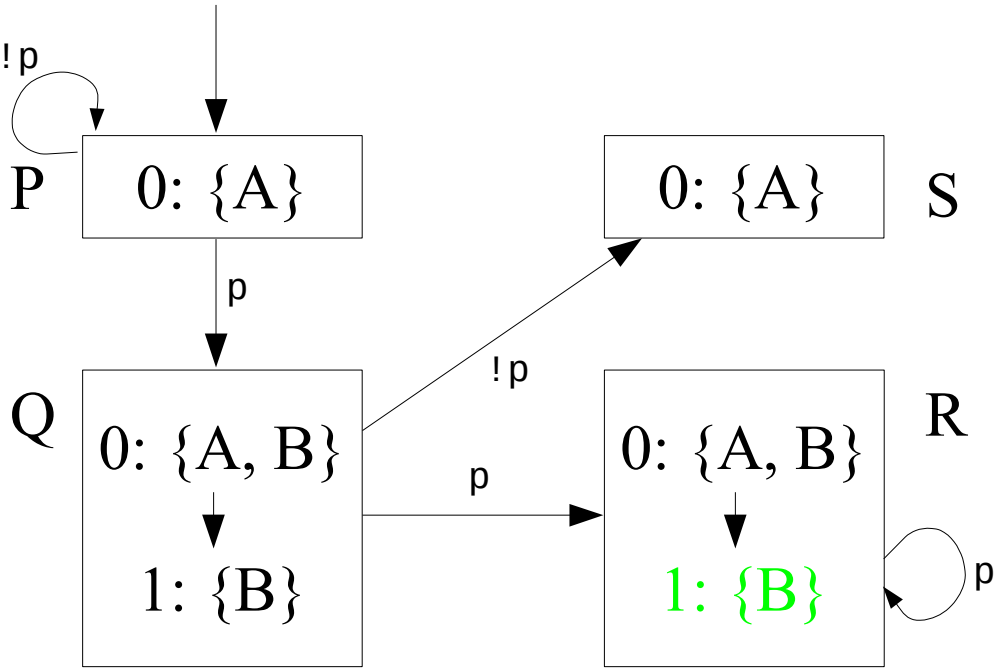
Safra's Konstruktion: Beispiel



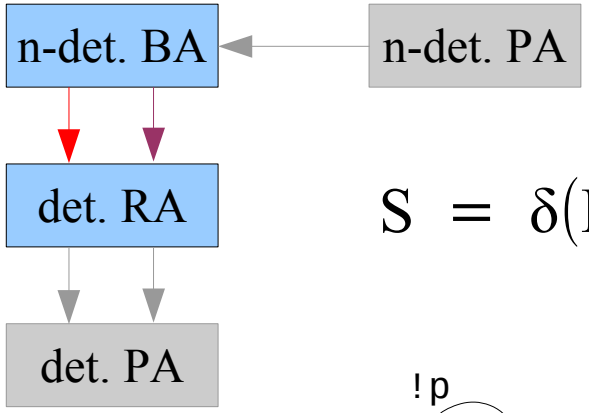
$$R = \delta(R,p)$$



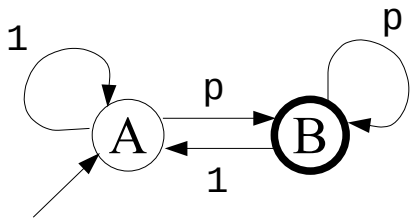
Büchi Automat



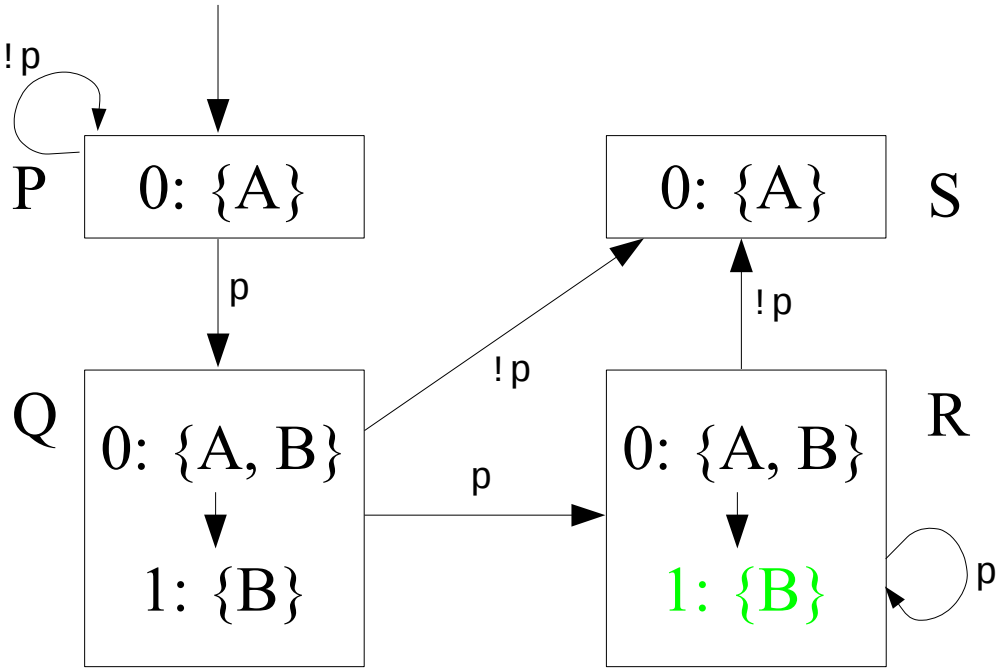
Safra's Konstruktion: Beispiel



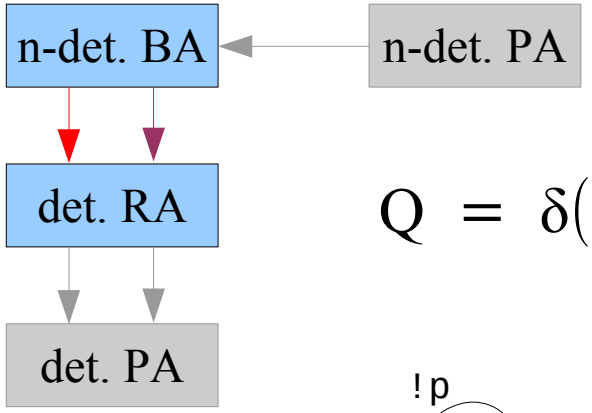
$$S = \delta(R, !p)$$



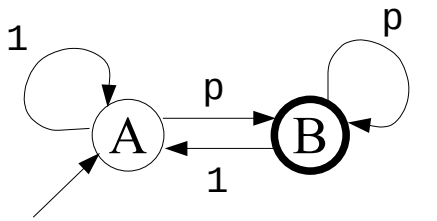
Büchi Automat



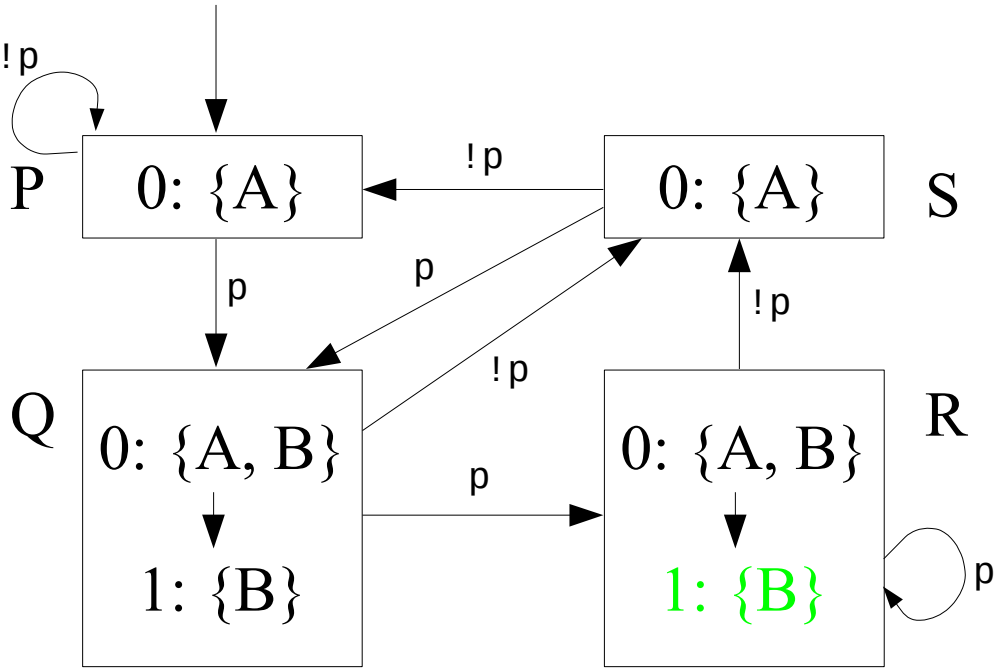
Safra's Konstruktion: Beispiel



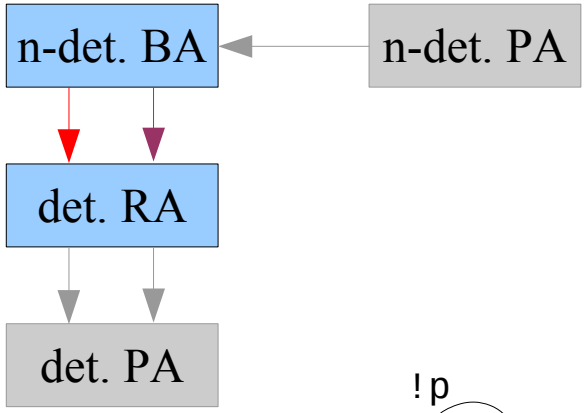
$Q = \delta(S,p)$ $P = \delta(S,!p)$

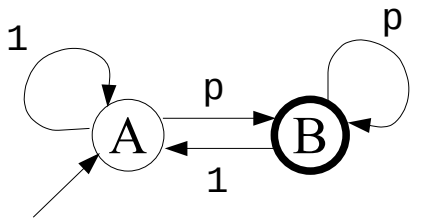


Büchi Automat

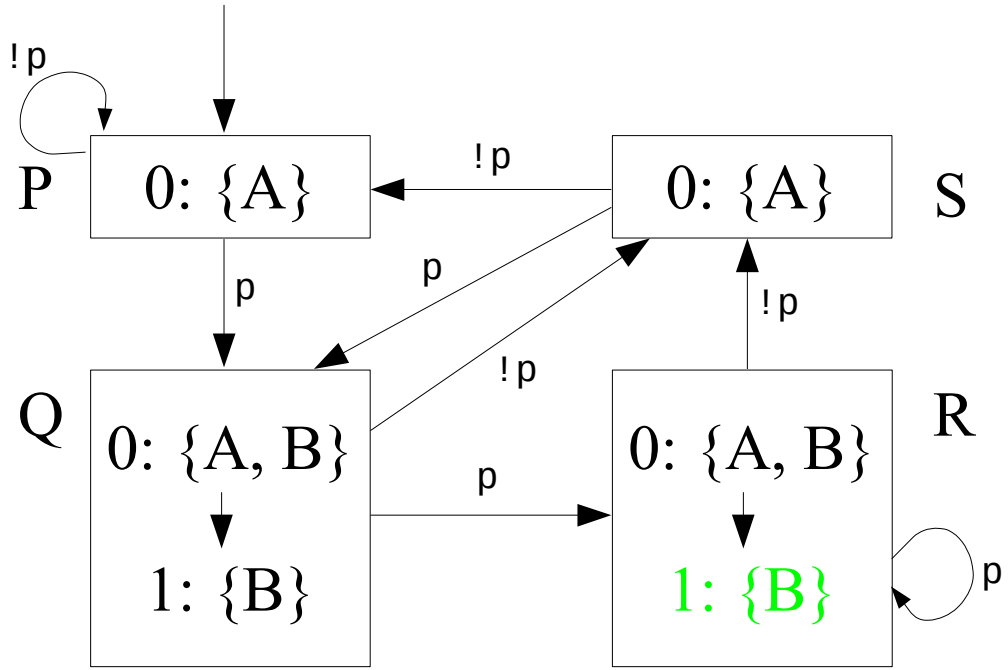


Safra's Konstruktion: Beispiel

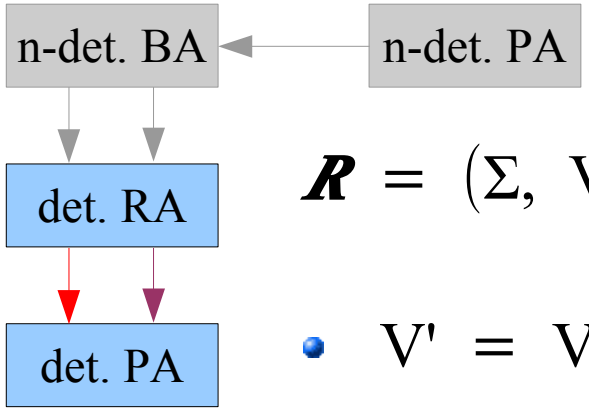




Büchi Automat



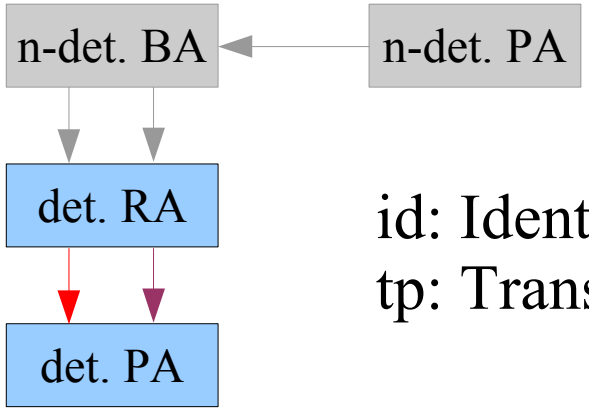
Rabin Paare: $C = \{(\{S\},\emptyset), (\{R\},\{P,S\})\}$



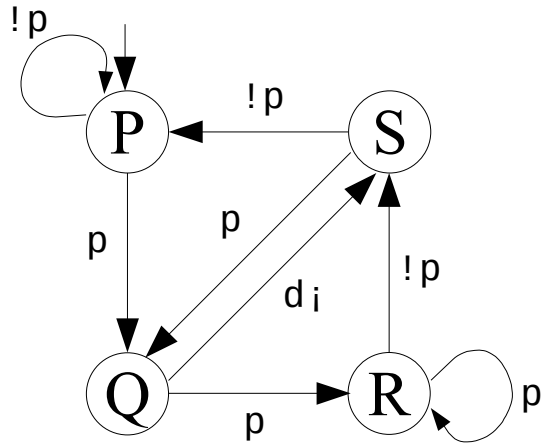
$$\boldsymbol{R} = (\Sigma,\ V,\ \delta,\ v_0,\ (G_i,\ R_i)_{i\in I}) \quad \boldsymbol{P} = (\Sigma,\ V',\ \delta',\ v'_{0'},\ \alpha)$$

- $V' = V \times \mathrm{perm}(I) \times I \times I$
  perm(I) sind die Permutationen von I.
- $(v',\ \pi',\ r',\ g') \in \delta((v,\ \pi,\ r,\ g),\ \sigma) \Leftrightarrow$
  - $v' \in \delta(v,\ \sigma)$
  - $\pi' = (p_1,...,p_k)$ erhält man aus $\pi = (l_1,...,l_k)$ indem man die l_i nach links schiebt mit $v' \in R_{li}$
  - r' ist der größte Index i, so dass $v \in R_{li}$ oder 0 wenn v in keinem R vorkommt.
  - g' ist der größte Index i, so dass $v \in G_{li}$ oder 0 wenn v in keinem R vorkommt
- $\alpha(q,\ \pi,\ r,\ g) = 2g$, iff $g > r$
- $\alpha(q,\ \pi,\ r,\ g) = 2r+1$, iff $g \leq r$

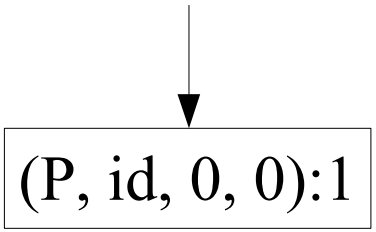
Parity Automat: Beispiel



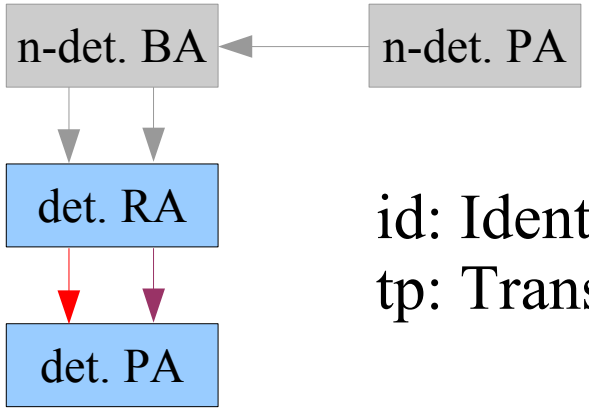
id: Identische Permutation
tp: Transposition



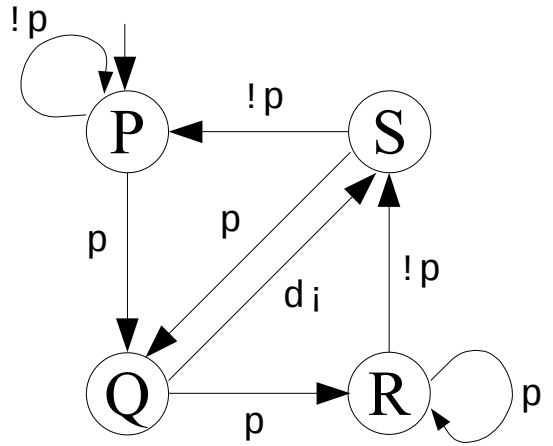
C = { ({S},∅), ({R},{P,S}) }
Rabin Automat



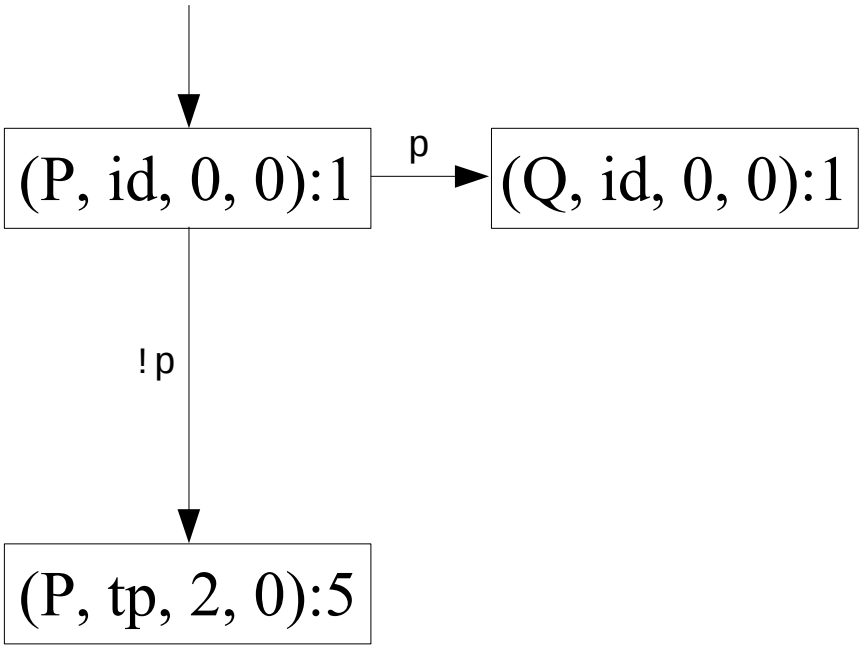
Parity Automat: Beispiel



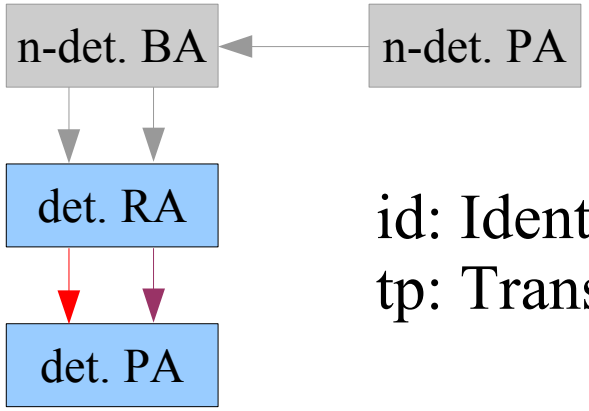
id: Identische Permutation
tp: Transposition



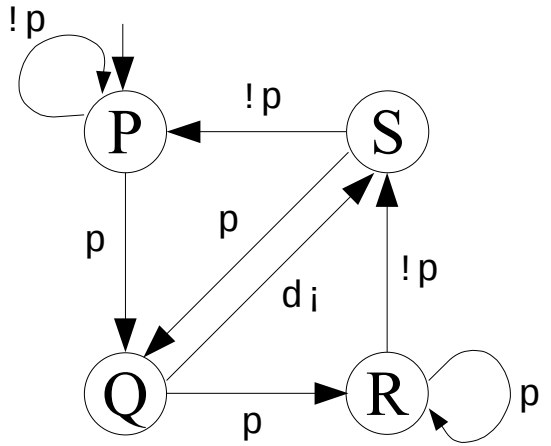
C = { ({S},∅), ({R},{P,S}) }
Rabin Automat



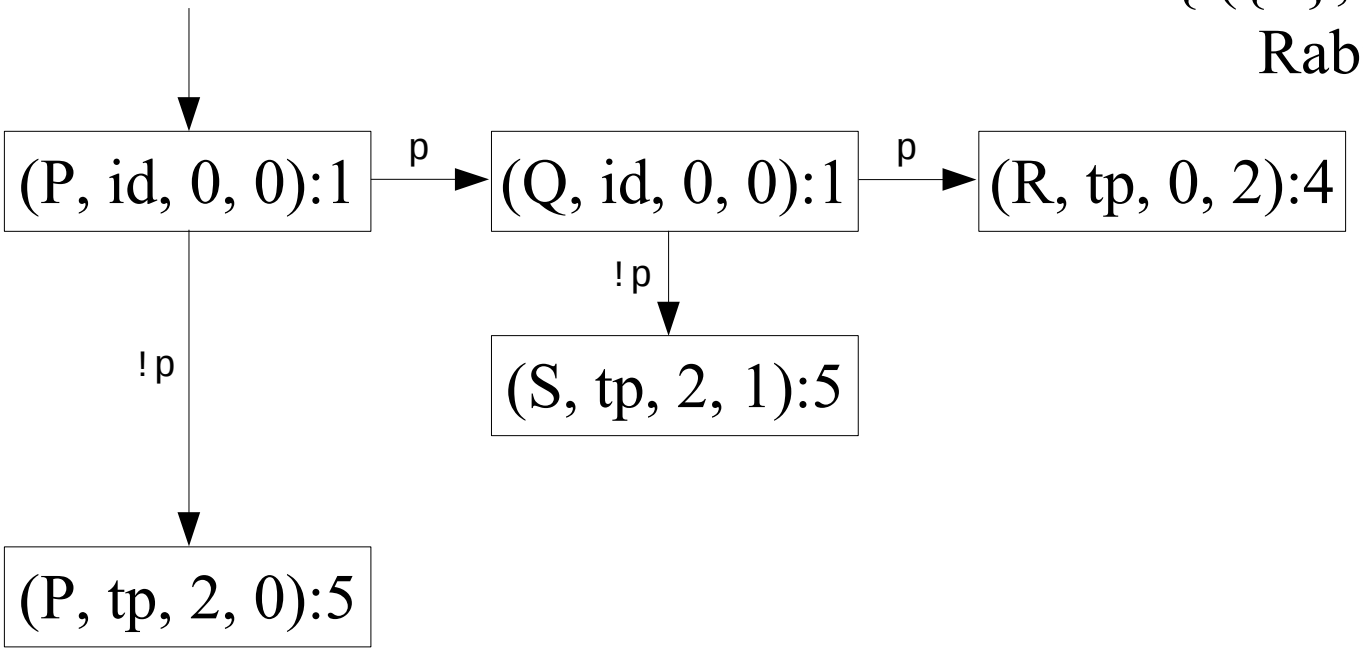
Parity Automat: Beispiel



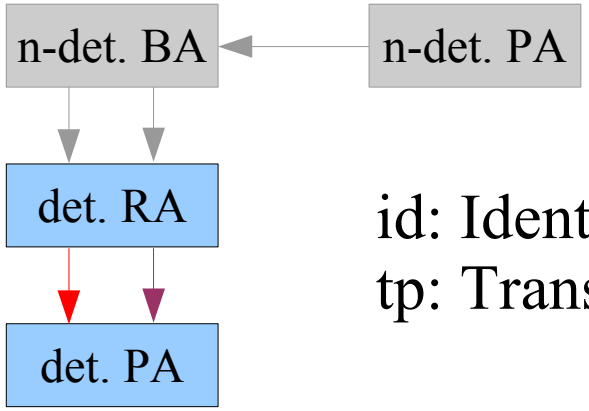
id: Identische Permutation
tp: Transposition



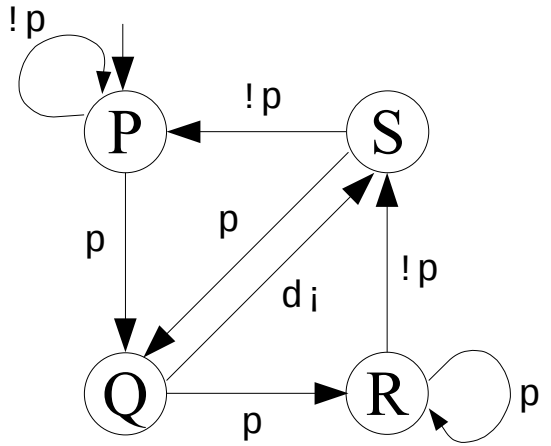
C = { ({S},∅), ({R},{P,S}) }
Rabin Automat



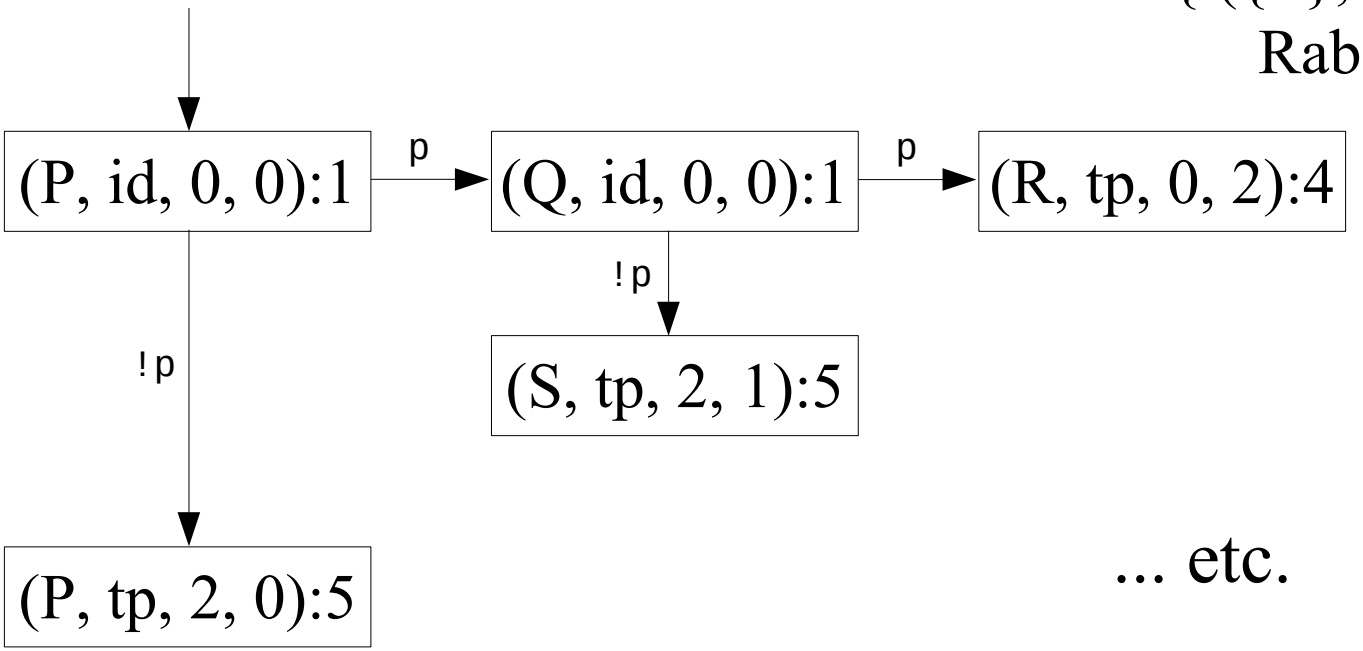
Parity Automat: Beispiel



id: Identische Permutation
tp: Transposition



C = { ({S},∅), ({R},{P,S}) }
Rabin Automat



... etc.

Gliederung

- Einleitung
- Architektur- und Spezifikationstransformation
  - Transformation von Architekturen
  - Transformation von Automaten
  - Transformation der Spezifikation
  - Optimierungen
- Verteilte Spiele
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse

- Spezifikation als LTL Formel oder Büchi Automat

- LTL2BA [Gastin, Oddoux]:
  - Spezifikation → Büchi Automat in 3 Schritten:

    1) LTL → very weak alternating automaton
    2) VWAA → generalized Büchi automaton
    3) GBA → Büchi Automat

Parity Automat nach Parity Spiel

$\boldsymbol{dPA} = (\Sigma, V, \delta, v_{init}, \alpha)$ liefert das minimal parity Spiel:

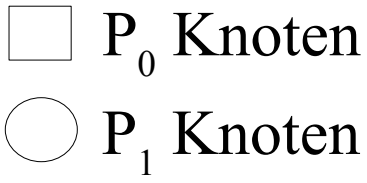
$\boldsymbol{pG} = (P_0, P_1, T, v_{init}, \chi)$

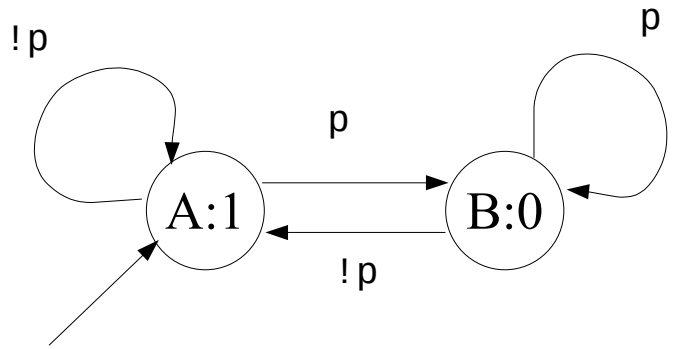
- $P_0 = V \times \Sigma$
- $P_1 = V$
- $T = \{(p_1,p_0), (p_0, p'_1) \mid p_0 = (p_1, \sigma) \wedge \delta(p_1, \sigma) = p'_1\}$
- $\chi(v) = \alpha(v) \wedge \chi((v, \sigma)) = \alpha(v)$

Konstruktion für deterministische Büchi Automaten analog

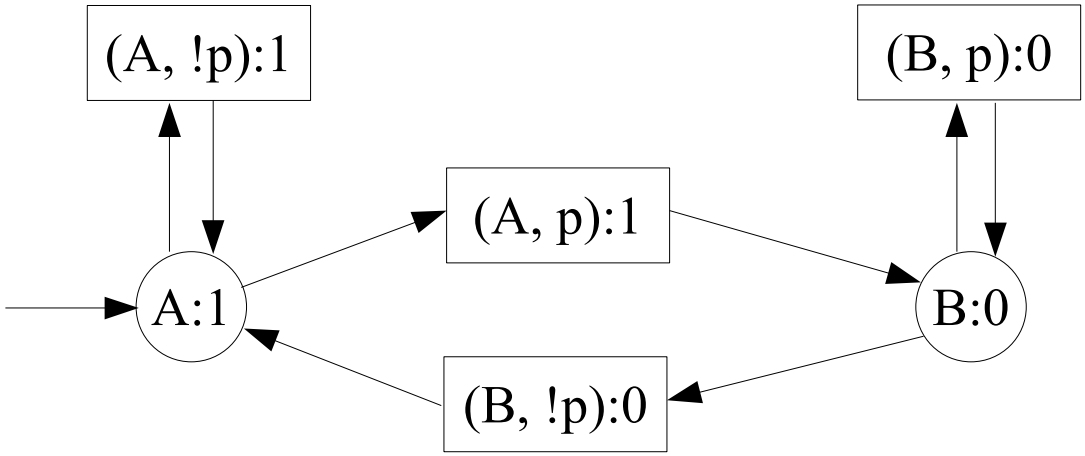
- $\chi(v) = 0 \Leftrightarrow v \in F$, sonst $\chi(v) = 1$

PA nach Parity Spiel: Beispiel





Parity Automat



Parity Spiel

Gliederung

- Einleitung
- Architektur- und Spezifikationstransformation
  - Transformation von Architekturen
  - Transformation von Automaten
  - Transformation der Spezifikation
  - Optimierungen
- Verteilte Spiele
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse

Delayed Simulation [Etessami, Wilke, Schuller 2001]:

- Zwei Spieler Spiel: Duplicator ↔ Spoiler
- Duplicator und Spoiler starten auf je einem Knoten
- Spoiler zieht in einen Nachfolger
- Duplicator kopiert den Zug
- Spoiler zieht ....

Duplicator Startknoten kann den Spoiler Startknoten simulieren, wenn Duplicator das Spiel gewinnt.

$\boldsymbol{BA} = (\Sigma,\ Q,\ \delta,\ q_{init},\ F)$

$\boldsymbol{G_{del}} = (V_D,\ V_S,\ T,\ \chi)$

- $V_D = \{(b,q,q',\sigma) \mid q,q' \in Q \wedge \sigma \in \Sigma \wedge b \in \{0,1\} \wedge \exists q'' \in Q.\ q \in \delta(q'',a)\}$
- $V_S = \{(b,q,q') \mid q,q' \in Q \wedge b \in \{0,1\} \wedge (q' \in F \rightarrow b=0)\}$
- $T = \{((b,q_1,q'_1,\sigma),(b,q_1,q'_2)) \mid q'_2 \in \delta(q'_1,\sigma) \wedge q'_2 \notin F\}$
  $\cup \{((b,q_1,q'_1,\sigma),(0,q_1,q'_2)) \mid q'_2 \in \delta(q'_1,\sigma) \wedge q'_2 \in F\}$
  $\cup \{((b,q1,q'1),(b,q2,q'1,\sigma)) \mid q2 \in \delta(q_1,\sigma) \wedge q'_2 \notin F\}$
  $\cup \{((b,q1,q'1),(1,q2,q'1,\sigma)) \mid q2 \in \delta(q_1,\sigma) \wedge q'_2 \in F\}$
- $\chi((b,q,q')) = b \wedge \chi((b,q,q',\sigma)) = 2$

Quotienten Automat

$$\boldsymbol{BA} = (\Sigma,\ Q,\ \delta,\ q_{init},\ F)$$

Zwei Zustände $q_1,q_2 \in Q$ sind äquivalent $(q_1 \approx q_2)$, wenn (b,q_1,q_2) und (b',q_2,q_1) in der Winning Region von Duplicator sind.

$$\boldsymbol{BA}/\approx \ :=\ (\Sigma,\ Q/\approx,\ \delta_{\approx},\ [q_{init}],\ F/\approx)$$

- $[q]$ ist die Restklasse von q bzgl. $\approx$
- $\delta_{\approx}([q],\sigma) = \{[q'] \mid \exists(q_0 \in [q],\ q'_0 \in [q']).\ q'_0 \in \delta(q_0,\sigma)\}$
- $\mathcal{L}(\boldsymbol{BA}) = \mathcal{L}(\boldsymbol{BA}/\approx)$

Gliederung

- Einleitung
- Architektur- und Spezifikationstransformation
- Verteilte Spiele
  - Konstruktion
  - Reduktion der Spieler
  - Determinisierung
  - Optimierung I
  - Optimierung II
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse

Ein Verteiltes Spiel [Mohalik, Walukiewicz] **G**
ist ein 4-Tupel $\mathbf{G}=(P,E,T,\mathit{ACC})$ mit:

$$P = E_0 \cup P_0 \times \ldots \times E_n \cup P_n \setminus E$$
$$E = E_0 \times \ldots \times E_n$$
$$T = T_E \cup T_P$$

$T_p \ni ((x_0,\ldots,x_n)\rightarrow(x_0',\ldots,x_n'))$, gdw.
$(x_i \rightarrow x_i') \in T_i$ für alle $x_i \in P_i$ und $x_i = x_i'$ für alle $x_i \in E_i$.

$E \times P \subseteq T_e \ni ((x_0,\ldots,x_n)\rightarrow(x_0',\ldots,x_n'))$, wenn
$(x_i \rightarrow x_i') \in T_i$ oder $xi = xi'$ für alle $x_i \in P_i \cup E_i$

Aus der lokalen Sicht ist die globale Position nicht bestimmbar.

Generelle Unentscheidbarkeit

Aus der lokalen Sicht ist die globale Position nicht bestimmbar.

Generelle Unentscheidbarkeit

ABER:

Informationshierarchie:
Prozesse nach dem Grad der Informiertheit geordnet

Daher entscheidbar.

Konstruktion

Die Konstruktion beginnt mit lokalen Spielen
für Prozesse und Spezifikation.

Beginne mit Startknoten des Verteilten Spiels:
Kartesisches Produkt der lokalen Startknoten.

Alle erreichbaren Knoten werden nacheinander abgearbeitet.

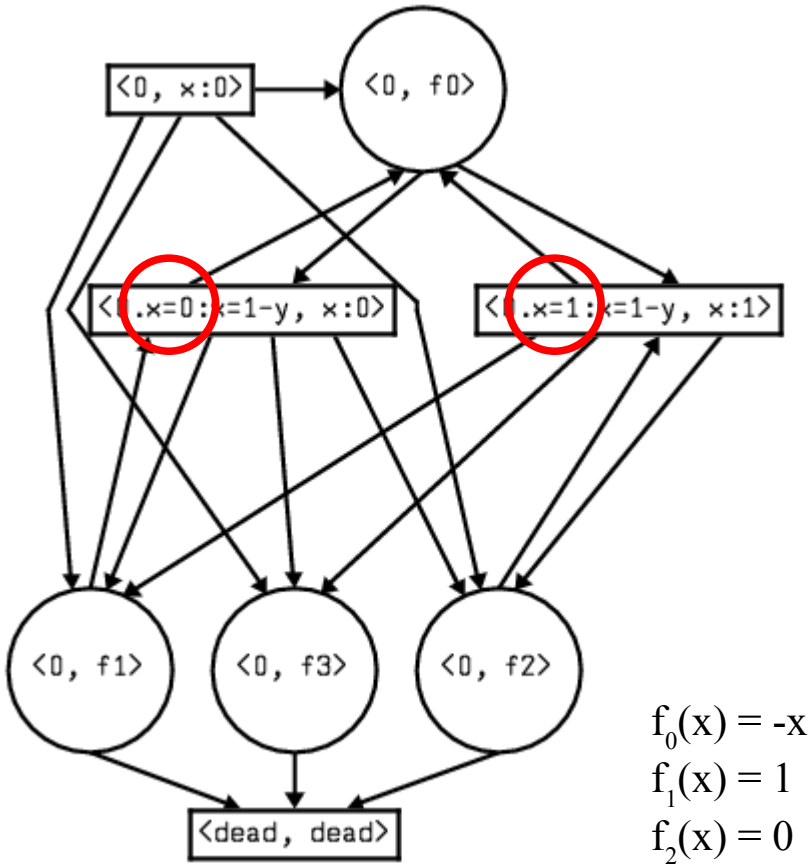
Bilden der Nachfolger für Knoten N wie folgt...

Bilden der Nachfolger für Knoten N wie folgt:

Spieler:
(i) S_i Menge der Nachfolger der i-ten Komponente im i-ten lokalen Spiel.
(ii) Berechne $S=S_0 \times S_1 \times \ldots \times S_{n-1}$
(iii) Füge Kante $\{N\} \times S$ und Knoten S zum Verteilten Spiel hinzu

Umgebung:
(i)-(ii) wie oben
(iii) Werte die Funktionen des Umgebungsknoten aus;
überprüfe, ob der Ausdruck in der 0-ten Komponente mit der Variablenbelegung zu Wahr auswertet.

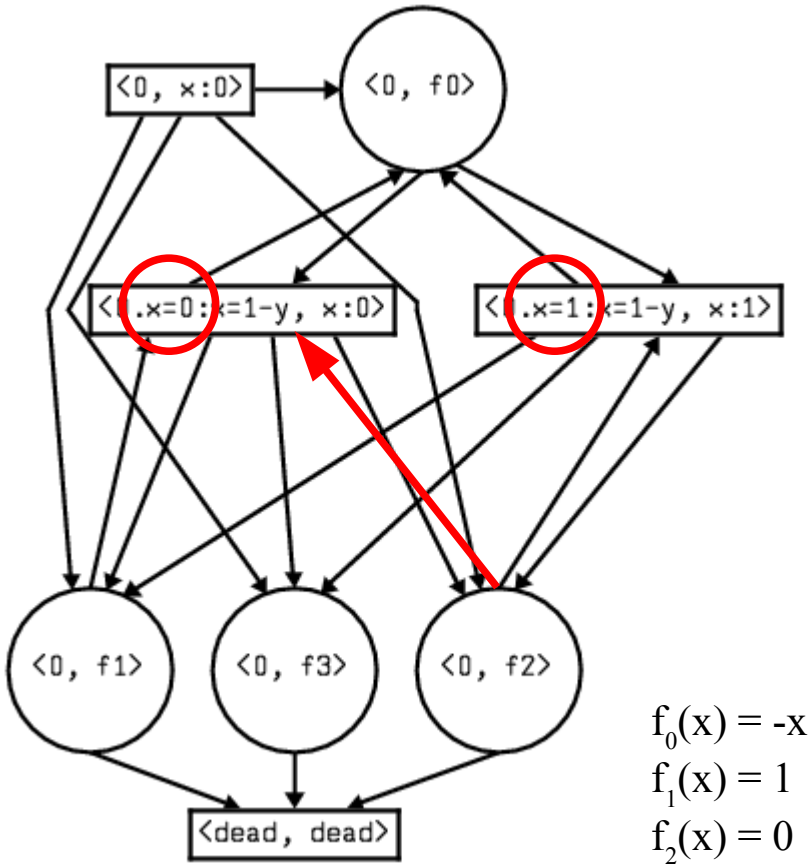
Die Parität der Verteilten Knoten wird auf die des Formelspiels gesetzt.



$f_0(x) = -x$
$f_1(x) = 1$
$f_2(x) = 0$
$f_3(x) = x$

Umgebung

Spieler

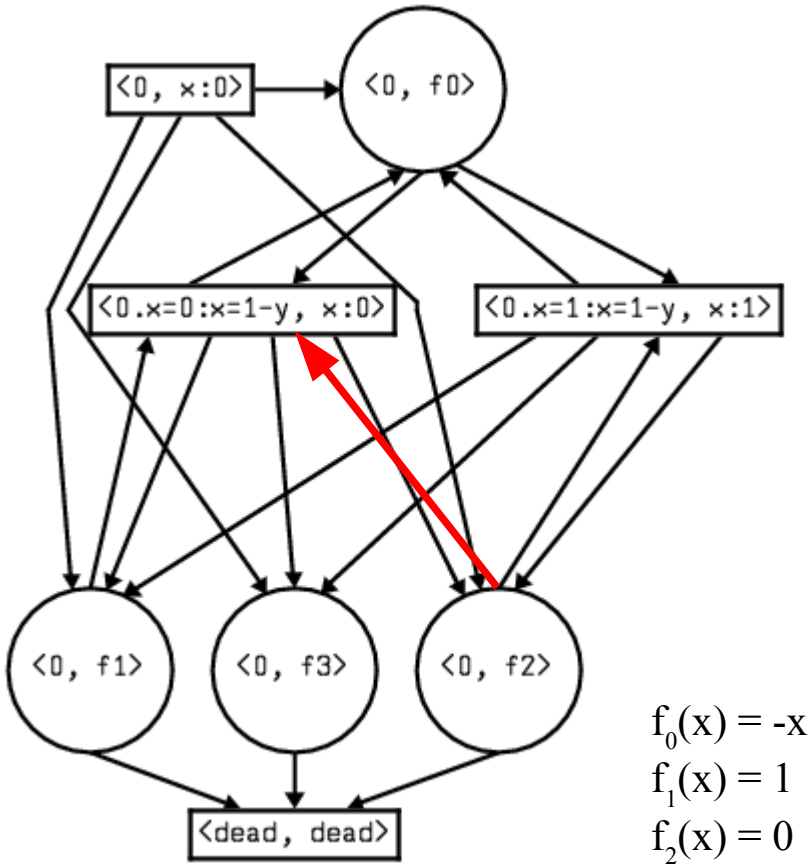


$<0,f_2>$?→ <0.x=0:x=1-y,x:0>

$f_0(x) = -x$
$f_1(x) = 1$
$f_2(x) = 0$
$f_3(x) = x$

○ Umgebung

▭ Spieler



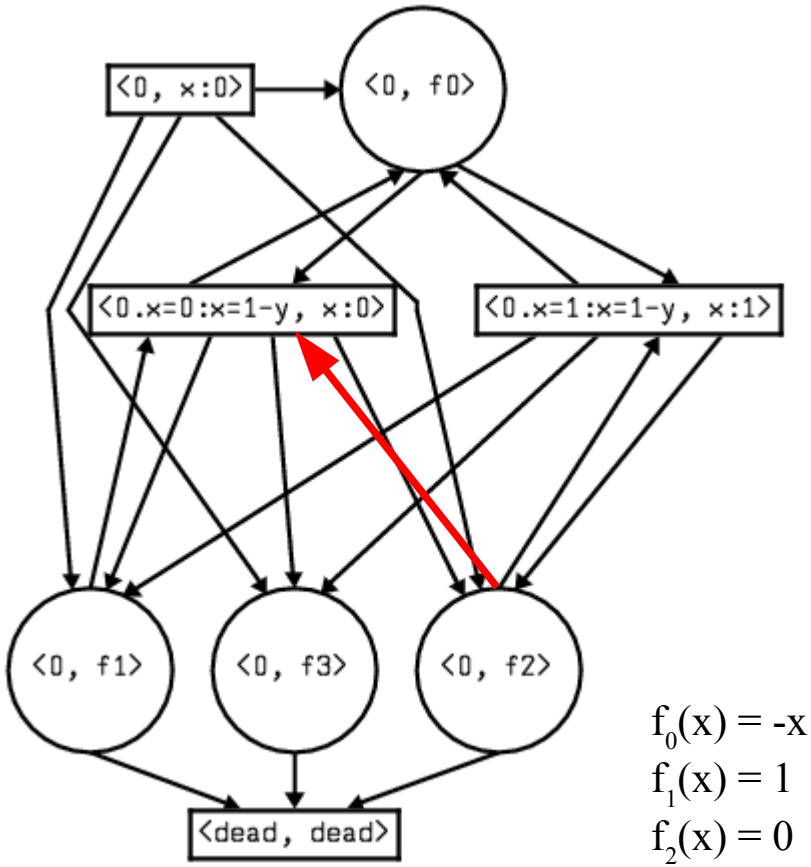
$f_0(x) = -x$
$f_1(x) = 1$
$f_2(x) = 0$
$f_3(x) = x$

Umgebung

Spieler

$\langle 0,f_2 \rangle \; ?\rightarrow \langle 0.x=0:x=1-y,x:0 \rangle$

- x=0 durch Umgebung definiert



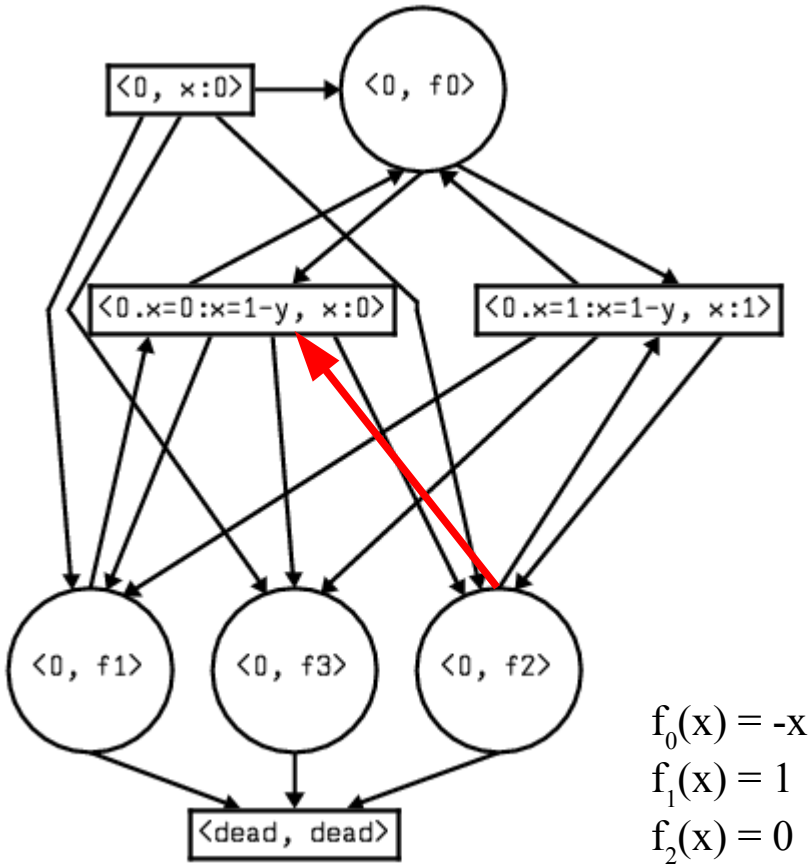
$f_0(x) = -x$
$f_1(x) = 1$
$f_2(x) = 0$
$f_3(x) = x$

Umgebung

Spieler

$\langle 0,f_2 \rangle$?→ $\langle 0.x=0:x=1-y,x:0 \rangle$

- x=0 durch Umgebung definiert
- $f_2(0) = 0$



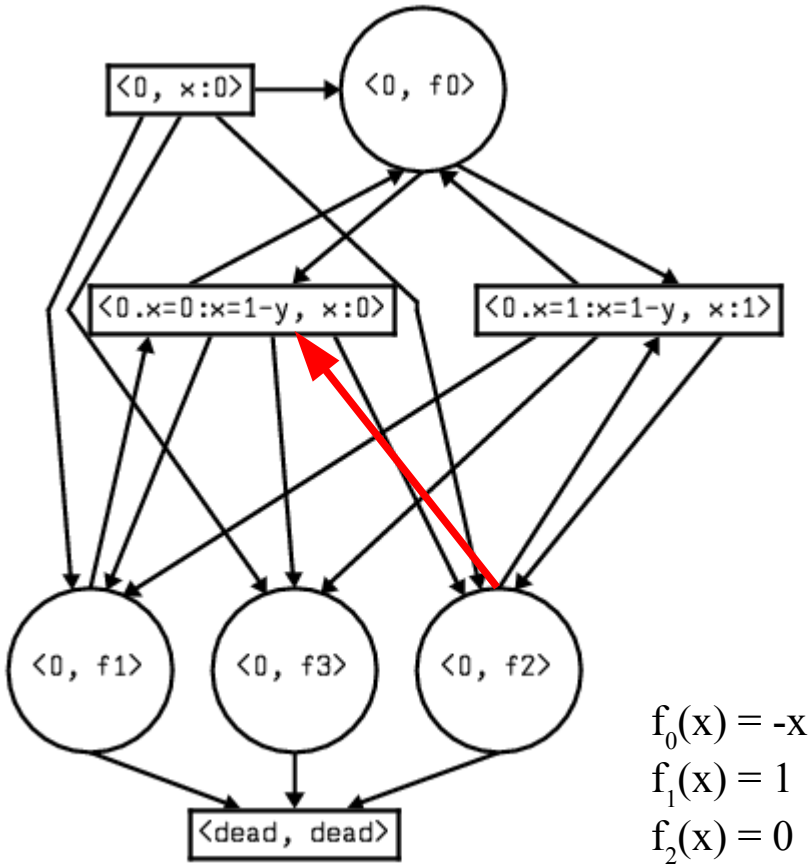
$f_0(x) = -x$
$f_1(x) = 1$
$f_2(x) = 0$
$f_3(x) = x$

Umgebung

Spieler

$\langle 0,f_2\rangle$?→ <0.x=0:x=1-y,x:0>

- x=0 durch Umgebung definiert
- $f_2(0) = 0$
- y=0



$f_0(x) = -x$
$f_1(x) = 1$
$f_2(x) = 0$
$f_3(x) = x$

Umgebung

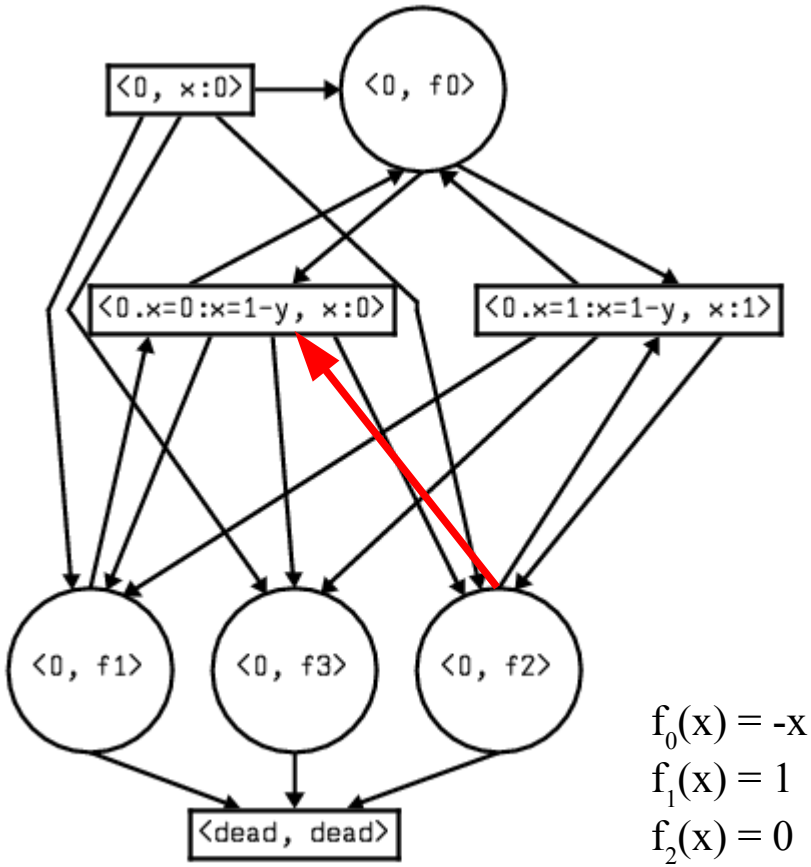
Spieler

$\langle 0,f_2 \rangle$?→ <0.x=0:x=1-y,x:0>

- x=0 durch Umgebung definiert
- $f_2(0) = 0$
- y=0
- auswerten von „x=1-y“:

0=1-0 ist falsch!

Konstruktion



$f_0(x) = -x$
$f_1(x) = 1$
$f_2(x) = 0$
$f_3(x) = x$

Umgebung

Spieler

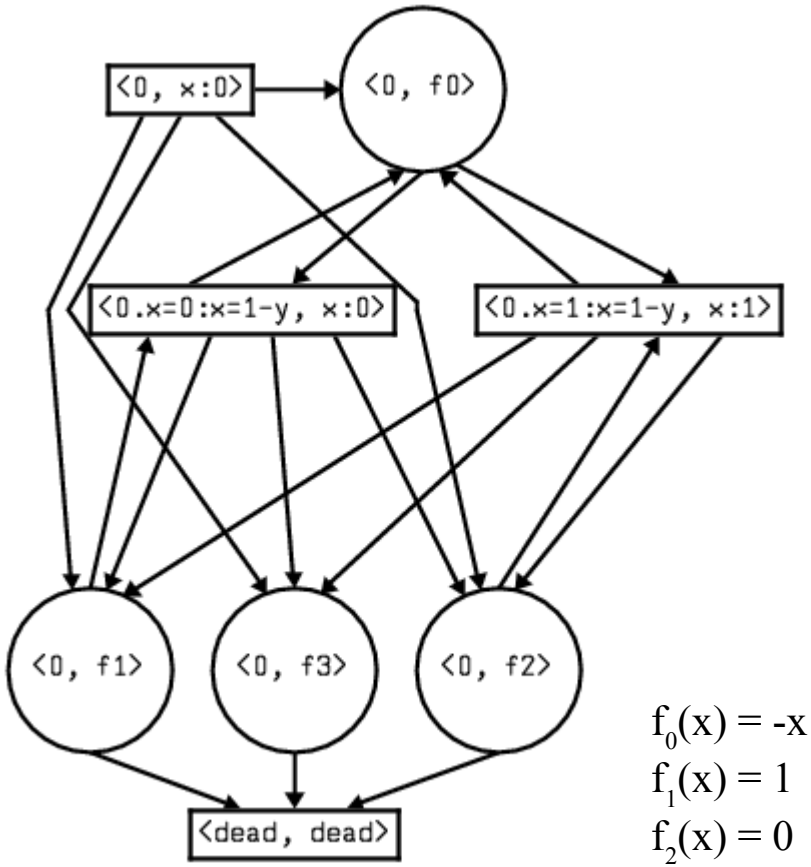
<0,f_2> → <0.x=0:x=1-y,x:0>

NEIN!!!

- x=0 durch Umgebung definiert
- $f_2(0) = 0$
- y=0
- auswerten von „x=1-y“:

0=1-0 ist falsch!

Konstruktion



$f_0(x) = -x$
$f_1(x) = 1$
$f_2(x) = 0$
$f_3(x) = x$

Umgebung

Spieler

<0,f_2> → <0.x=0:x=1-y,x:0>

NEIN!!!

- x=0 durch Umgebung definiert
- $f_2(0) = 0$
- y=0
- auswerten von „x=1-y“:

0=1-0 ist falsch!

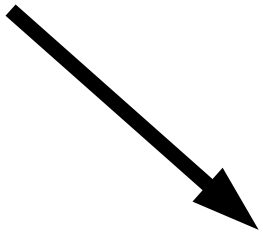
Gliederung

- Einleitung
- Architektur- und Spezifikationstransformation
- Verteilte Spiele
  - Konstruktion
  - Reduktion der Spieler
  - Determinisierung
  - Optimierung I
  - Optimierung II
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse

DIVIDE ( )

GLUE ( )

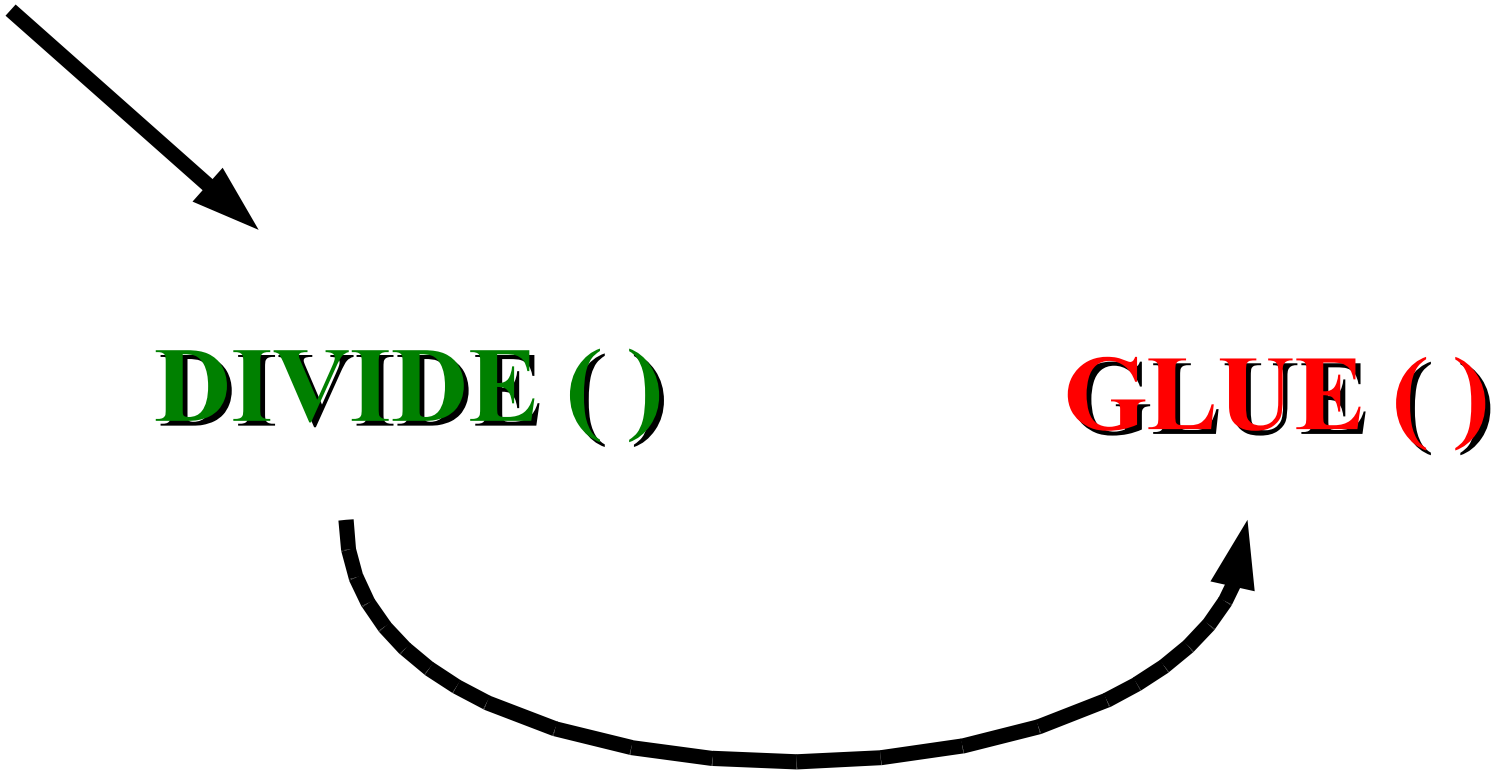
Was passiert jetzt?



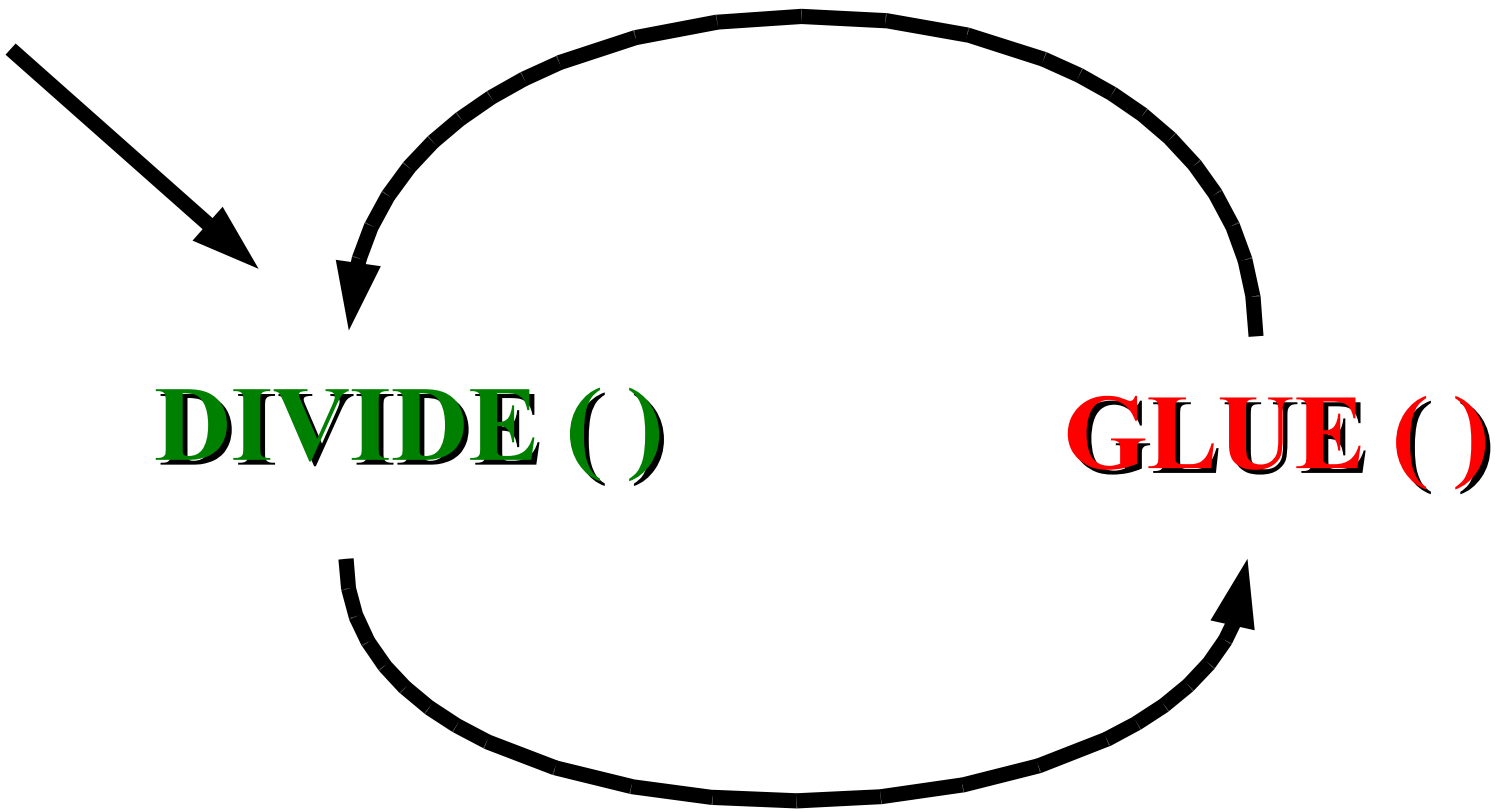
DIVIDE ( )

GLUE ( )

Was passiert jetzt?



Was passiert jetzt?



Gliederung

- Einleitung
- Architektur- und Spezifikationstransformation
- Verteilte Spiele
  - Konstruktion
  - Reduktion der Spieler
  - Determinisierung
  - Optimierung I
  - Optimierung II
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse

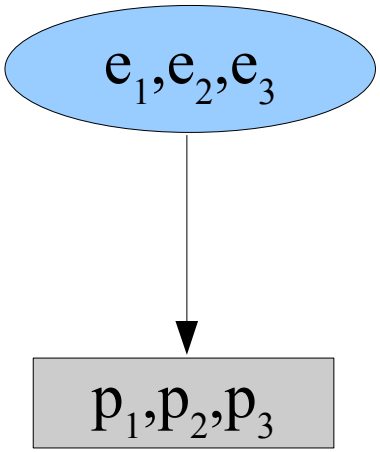
Reduktion der Spieler

DIVIDE( ) transformiert ein Verteiltes Spiel mit n lokalen Spielern $\mathbf{G}=(P,E,T,\mathcal{ACC})$ in ein Verteiltes Spiel $\mathbf{G_{DIVIDED}}=(P',E',T',\mathcal{ACC}')$ mit n-1 lokalen Spielern:

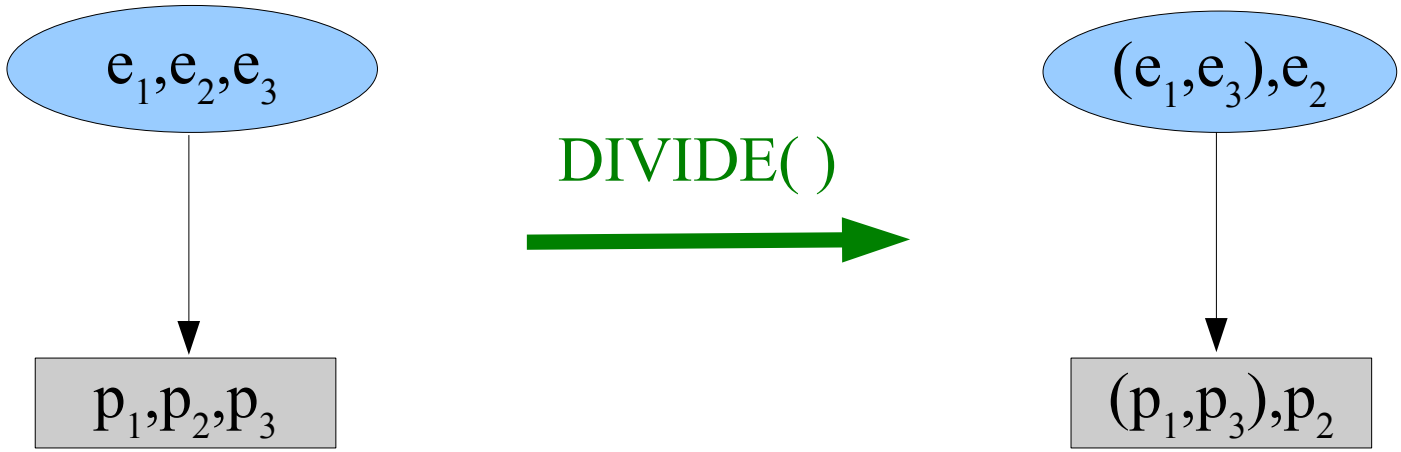
- $P_i' = P_i$, $E_i' = E_i$ und $T_i' = T_i$ für alle $i=1,\ldots,n-1$
- $E_0' = E_0 \times E_n$ und $P_0' = (P_0 \cup E_0) \times (P_n \cup E_n) \setminus E_0'$
- Für $(p_0,e_0) \in T_0$ und $(p_n,e_n) \in T_n$, gibt es die folgenden Transitionen in T_0':

$$(p_0,e_n) \rightarrow (e_0,e_n),$$
$$(e_0,p_n) \rightarrow (e_n,e_0),$$
$$(p_0,p_n) \rightarrow (e_0,e_n).$$

Reduktion der Spieler



Reduktion der Spieler

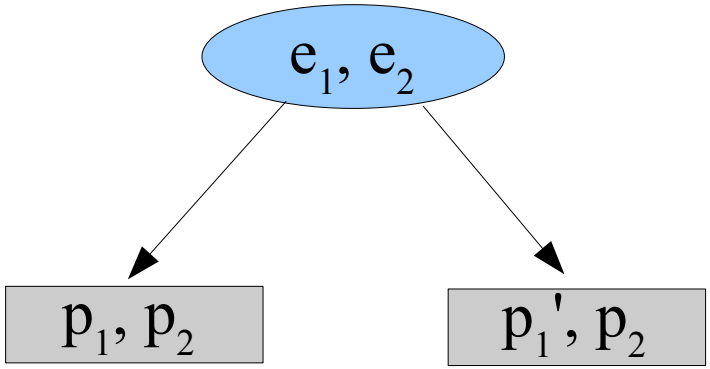


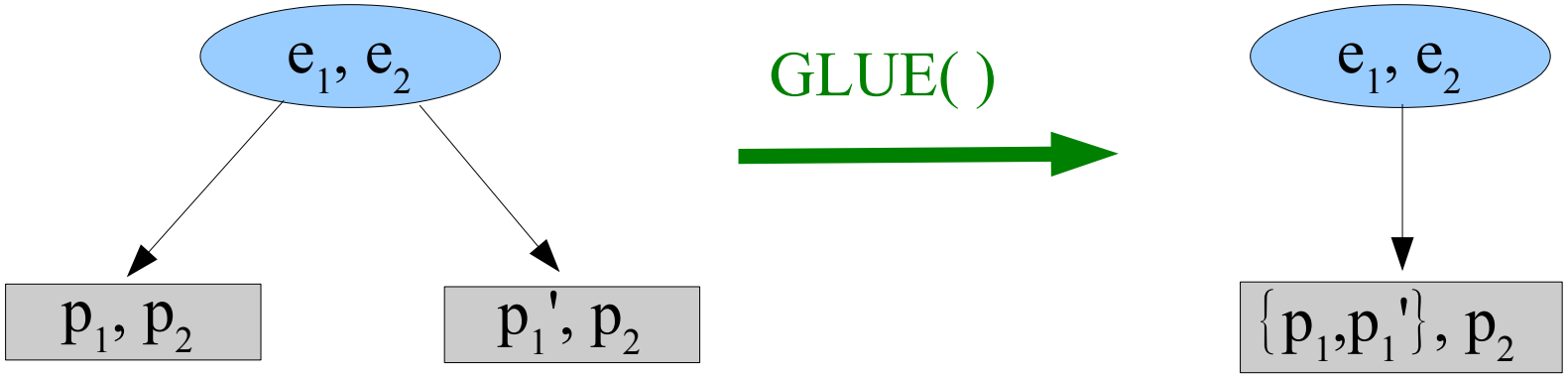
Gliederung

- Einleitung
- Architektur- und Spezifikationstransformation
- Verteilte Spiele
  - Konstruktion
  - Reduktion der Spieler
  - Determinisierung
  - Optimierung I
  - Optimierung II
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse

GLUE( ) transformiert ein Verteiltes Spiel $\mathbf{G}=(P,E,T,ACC)$ in ein Verteiltes Spiel $\mathbf{G_{GLUED}}=(P',E',T',ACC')$ mit:

- $P_i' = P_i$, $E_i' = E_i$ und $T_i' = T_i$ für alle $i=1,\ldots,n$
- $P_0' = 2^{E0\times P0}$ und $E_0'=2^{P0\times E0}$
- $p'\rightarrow_0 e'$, wenn es für jedes $(e,p)\in p'$ ein $(p,e)\in e'\cap T_0$ gibt
- $(e_0',\underline{e})\rightarrow(x_0',\underline{x})\in T'$ mit $x_0\neq\emptyset$ und
  $x_0'=\{(e_0,x_0)\mid\exists(p',e_0)\in e'_0.(e_0,\underline{e})\rightarrow(x_0,\underline{x})\}$


e_1, e_2
p_1, p_2
p_1', p_2



Wachstum exponentiell in der Anzahl der Spielknoten.

GLUE( ) - Gewinnbedingung

- Im Spiel GLUE(G) ist ein Play eine Sequenz von Mengen.
- Ein THREAD ist ein Pfad durch diese Sequenz.
  Für zwei aufeinander folgende Knoten $(v_1,v_2),(v_3,v_4) \in$ THREAD gilt $v_2=v_3$
- (*) „all-Threads“-Gewinnbedingung:
  Ein Play in GLUE(G) ist gewinnend für den Spieler, wenn jeder THREAD(Play) im Lokalen Spiel gewinnend ist.
- Es gibt einen deterministischen Parity Automaten,
  der Sequenzen über $(P_0' \bullet E_0')$ mit der Eigenschaft (*) erkennt.

Konstruktion folgt später!

Vereinfachung der GLUE( )-Gewinnbedingung

Abschätzung I:

Setze Parität des Verteilten Knotens auf die kleinste gerade vorkommende
(größte ungerade, falls nur ungerade Paritäten vorkommen)

Abschätzung II:

Setze Parität auf größte Gerade.
(kleinste ungerade)

Vereinfachung der GLUE( )-Gewinnbedingung

Abschätzung I:

Setze Parität des Verteilten Knotens auf die kleinste gerade vorkommende (größte ungerade, falls nur ungerade Paritäten vorkommen)

Abschätzung II:

Setze Parität auf größte Gerade.
(kleinste ungerade)

✚ lineare Laufzeit, statt exponentieller

Liefert Implementierung mit AI ⇒ korrekte Lösung.

Nicht Lösbar mit AII ⇒ keine Lösung.

Verteiltes Spiel

Durch wiederholtes Anwenden von DIVIDE( ) und GLUE( )

2-Spieler-Spiel

Lösen

Gedächtnislose, nichtdet. Strategie

Winning Regions

Keine Strategie:
Spezifikation auf der Architektur nicht erfüllbar

Gibt es eine Strategie:
Strategie beschreibt ein Verhalten der Architektur, das die Spezifikation erfüllt.

Gliederung

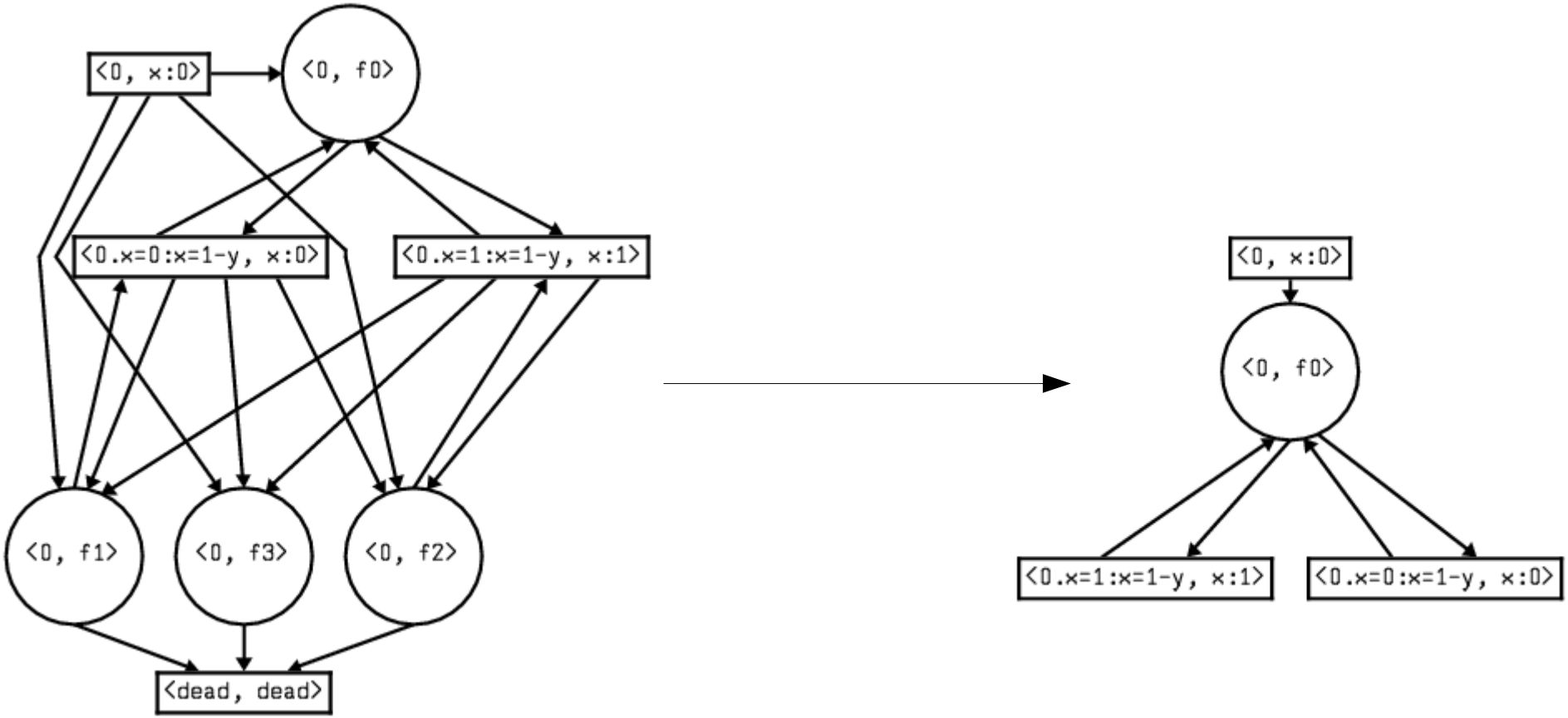
- Einleitung
- Architektur- und Spezifikationstransformation
- Verteilte Spiele
  - Konstruktion
  - Reduktion der Spieler
  - Determinisierung
  - Optimierung I
  - Optimierung II
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse

„Dead End“ - Reduktion:

$\mathrm{DeadEnd}_{\mathrm{Attr}}(G) := \mathrm{Attr}_{\mathrm{Env}}(G, \emptyset)$

- Bilde $\mathrm{DeadEnd}_{\mathrm{Attr}}(G)$
- Entferne alle Knoten in $\mathrm{DeadEnd}_{\mathrm{Attr}}(G)$ und benachbarte Kanten

Korrektheit folgt aus direkt aus der Definition des Attractors.


<0, x:0>
<0, f0>
<0.x=0:x=1-y, x:0>
<0.x=1:x=1-y, x:1>
<0, f1>
<0, f3>
<0, f2>
<dead, dead>
<0, x:0>
<0, f0>
<0.x=1:x=1-y, x:1>
<0.x=0:x=1-y, x:0>

Gliederung

- Einleitung
- Architektur- und Spezifikationstransformation
- Verteilte Spiele
  - Konstruktion
  - Reduktion der Spieler
  - Determinisierung
  - Optimierung I
  - Optimierung II
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse

Alle Knoten, die für die Umgebung gewinnend sind werden nie für den Spieler gewinnend sein.

Gewinnende Knoten für die Umgebung können entfernt werden.

„Winning Region“-Optimierung:

- Lösen des Verteilten Spiels
- Entfernen der für das Environment gewinnenden Knoten und deren benachbarten Kanten.

Optmierung I vs. Optimierung II

„Dead End“-Optimierung

„Winning Region“-Optimierung

Optmierung I vs. Optimierung II

„Dead End“-Optimierung

✚ viel schneller

„Winning Region“-Optimierung

✚ stark bei unlösbaren DSPs

Optmierung I vs. Optimierung II

„Dead End“-Optimierung

✚ viel schneller

„Winning Region“-Optimierung

Optmierung I vs. Optimierung II

„Dead End“-Optimierung

✚ viel schneller

„Winning Region“-Optimierung

✚ stark bei unlösbaren DSPs

In **Rea**Syn sind beide Vereinfachungen optional anzuwählen.

Die „Dead End“-Optimierung läuft nach der Konstruktion des Verteilten Spiels. Die „Winning Region“-Optimierung danach und nach jeder „GLUE()“-Operation.

Am Ende steht ein zwei Spieler Spiel.

Lösen

Gliederung

- Einleitung
- Architektur- und Spezifikationstransformation
- Verteilte Spiele
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse

"All-Threads" Gewinnbedingung

$\boldsymbol{G} = (P_0, P_1, T, v_0, \alpha)$ parity Spiel

$\boldsymbol{G}_{glue} = (P'_0, P'_1, T', v'_0, WIN)$ "All-Threads" Spiel

Ziel: Transformation von $\boldsymbol{G}_{glue}$ in ein parity Spiel.

"all-Threads" Gewinnbedingung ist eine reguläre Gewinnbedingung. D.h. es existiert ein deterministischer parity Automat, der Sequenzen über $(P'_0 \cup P'_1)$, die akzeptierende Plays von $\boldsymbol{G}$ darstellen, akzeptiert.

Siegbedingungstransformation

$\boldsymbol{G} = (P_0, P_1, T, v_0, \alpha)$ parity Spiel

$\boldsymbol{G}_{glue} = (P'_0, P'_1, T', v'_0, WIN)$ "all-Threads" Spiel

- Nicht deterministischer parity Automat. Akzeptiert Sequenzen mit mindestens einem nicht akzeptierenden Thread.
- Determinisieren des parity Automaten. (2te Transformationskette)
- Parity Spiel als Produkt von $\boldsymbol{G}_{glue}$ und dem det. parity Automaten

Nicht-deterministischer Automat

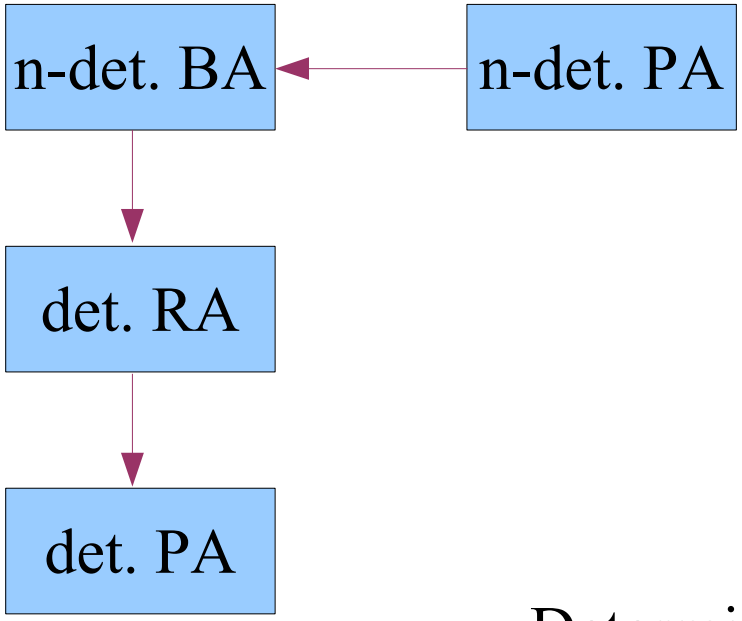
$\boldsymbol{G} = (P_0, P_1, T, v_0, \alpha)$ parity Spiel

$\boldsymbol{G}_{glue} = (P'_0, P'_1, T', v'_0, WIN)$ "all-Threads" Spiel

$\boldsymbol{dPa} = (V, \sum, \delta, v^*_0, \chi)$

- $V = P_0 \cup P_1$
- $\sum = P'_0 \cup P'_1$
- $\delta(v,\sigma) = \{v' \in V \mid (v,v') \in \sigma\}$
- $v^*_0 = v_0$
- $\chi(v) = \alpha(v) + 1$

Deterministischer Automat



Determinisierung in 3 Schritten:

- n-det. parity Automat → n-det. Büchi Automat
- n-det. Büchi Automat → det. Rabin Automat
- det. Rabin Automat → det. parity Automat

$\boldsymbol{G}_{\boldsymbol{glue}} = (P'_0, P'_1, T', v'_0, \mathrm{WIN})$ "all-Threads" Spiel

$\boldsymbol{dPa} = (V, \sum, \delta, v^*_0, \chi)$ deterministischer parity Automat

$\boldsymbol{pG} = (P_0, P_1, T, v_0, \alpha)$

- $P_0 = V \times P'_0$
- $P_1 = V \times P'_1$
- $T = \{((v, n), (v', n')) \mid (n, n') \in T' \wedge \delta(v,n) = v\}$
- $v_0 = (v^*_0, v'_0)$
- $\chi((v, v')) = \alpha(v) + 1$

Gliederung

- Einleitung
- Architektur- und Spezifikationstransformation
- Verteilte Spiele
- Transformation von Siegbedingungen
- Erzeugung von Programmen
  - Programm Erzeugung
  - Optimierungen
- Ergebnisse

Erzeugung der Programme

Problem:

- Zu Winning Region und nicht deterministischer Strategie, generiere ein korrektes Programm

Erzeugung der Programme

Problem:

- Zu Winning Region und nicht deterministischer Strategie, generiere ein korrektes Programm

Lösung:

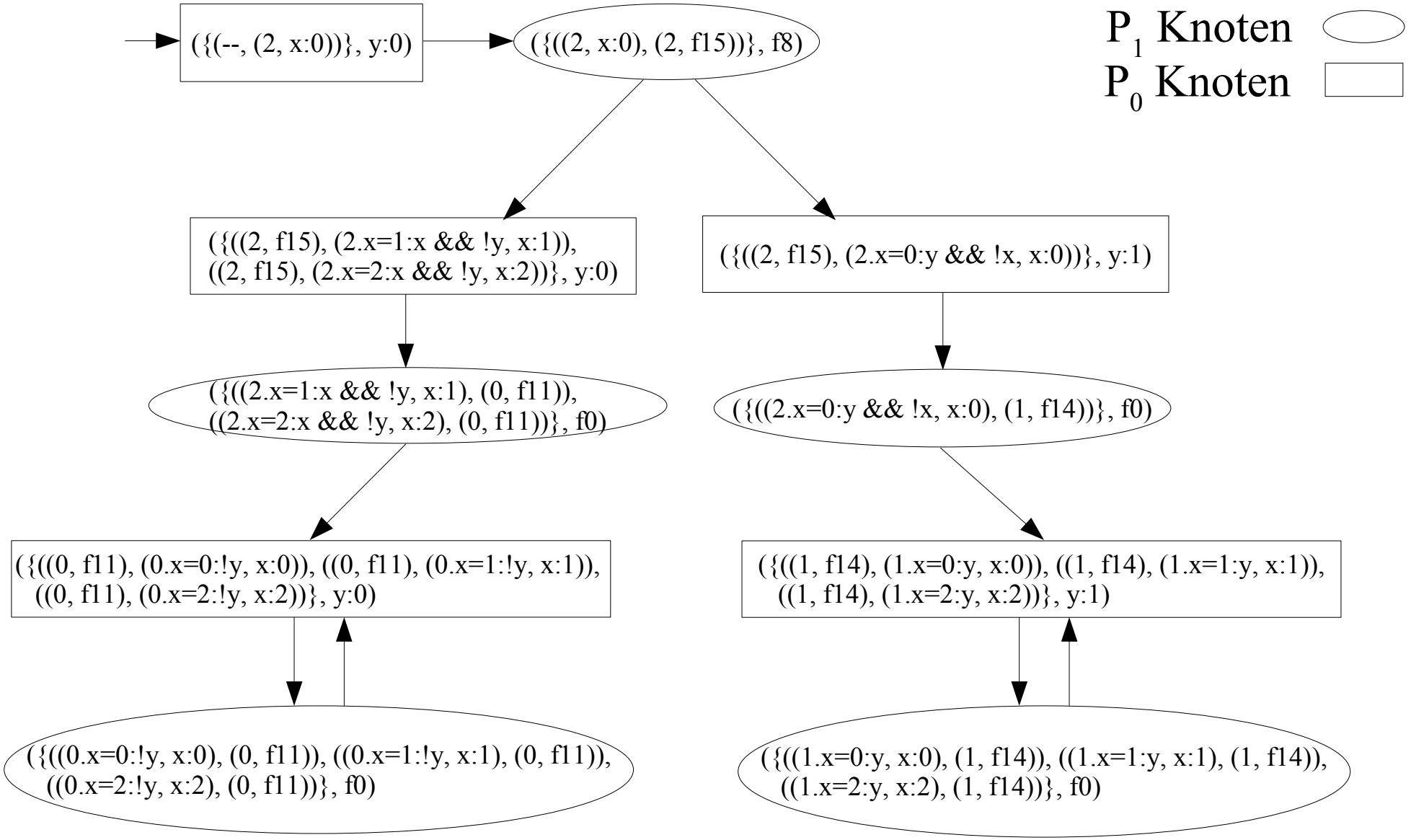
- Determinisiere die Strategie
- Aufspalten der deterministischen Strategie in Strategien für die einzelnen Prozesse der Architektur
- Übersetze Strategie in Programm Code.

Strategie-Determinisierung

- Nicht deterministische Strategie legt für P_0 Knoten der Winning Region eine Menge von gewinnenden Nachfolgern fest.
- Deterministische Strategie wählt zufällig einen Nachfolger aus dieser Menge aus.

- Deterministische Strategie und Winning Region stellen einen unendlichen Baum dar:
  - Winning Region = Knotenmenge
  - deterministische Strategie bestimmt Nachfolger für P_0 Knoten
  - Transitionen des vert. Spiels bestimmen Nachfolger für P_1 Knoten

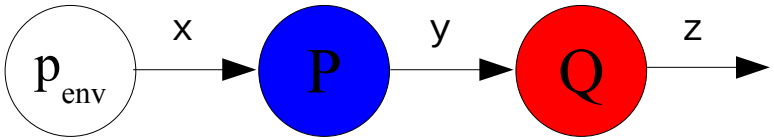
Deterministische Strategie



Aufspalten der Strategie

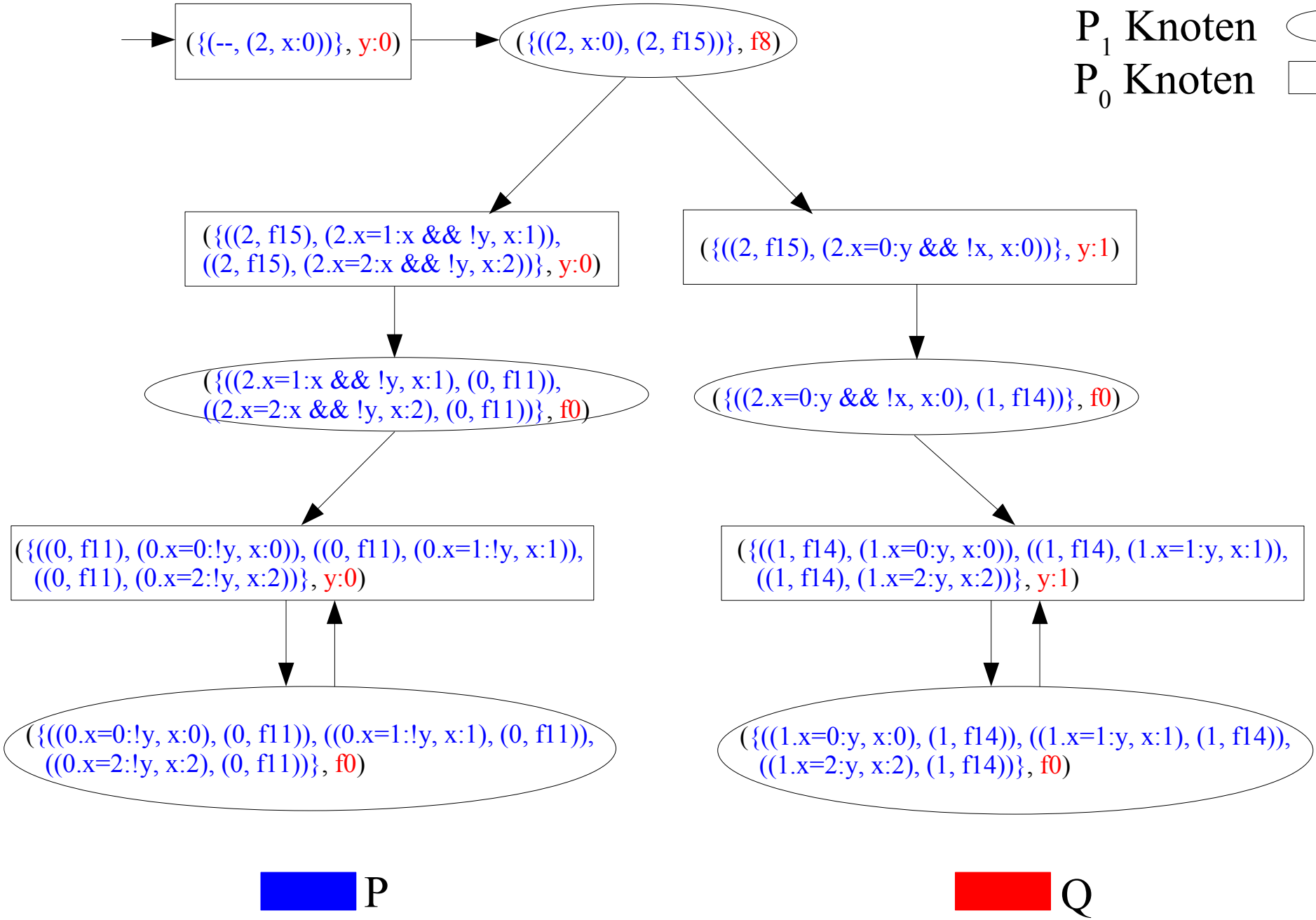
Finde für jeden Prozess der Architektur die passende Komponente in der Winning Region

Architektur:



(x∈[0;2], y∈[0;1], z∈[0;2])

Deterministische Strategie



Erzeugung der Programme

- Jeder P_1 Knoten repräsentiert einen Zustand im erzeugten Programm
- Jeder P_0 Knoten repräsentiert einen Nachfolgezustand im erzeugten Programm (abh. von Eingabewert)

Gliederung

- Einleitung
- Architektur- und Spezifikationstransformation
- Verteilte Spiele
- Transformation von Siegbedingungen
- Erzeugung von Programmen
  - Programm-Erzeugung
  - Optimierungen
- Ergebnisse

Programm Optimierung

Programmgröße ist direkt proportional zu Anzahl der Knoten von P_1 in dem Play:

Zwei Optimierungsansätze:

- Verkleinerung der Winning Region (Quotienten Region)
- Finden einer Strategie die möglichst wenig P_1 Knoten besucht.

Winning Region Optimierung

In dem verteilten Spiel repräsentieren P_0 Knoten Eingabewerte der Umgebung; P_1 Knoten repräsentieren Funktionen die von den Prozessen berechnet werden.

Zwei Optimierungsschritte:

- Reduziere P_0 Knoten (ordinary Simulation [Milner 89])
- Reduziere P_1 Knoten

P_0 Knoten Optimierung

Zwei Spieler Spiel: Duplicator ↔ Spoiler
Spoiler macht einen Zug, Duplicator muss den Zug kopieren.

Winning Region $\boldsymbol{G} = (P_0, P_1, F, \chi)$

$\boldsymbol{eqG} = (V_D, V_S, T, \alpha)$

- $V_D = \{(v, v', e) \mid v, v' \in P_0 \wedge e \in P_1 \wedge (e, v) \in F\}$
- $V_S = \{(v, v') \mid v, v' \in P_0\}$
- $T = \{((v_1, v'_1, e), (v_1, v'_2)) \mid \exists e' \in P_1.$ e und e' repr. die gleichen Funktionen $\wedge (v'_1, e') \in F \wedge (e', v'_2) \in F\} \cup$
  $\{((v_1, v'_1), (v_2, v'_1, e)) \mid (v_1, e) \in F \wedge (e, v_2) \in F\}$
- $\alpha(v) = 0$

P_0 Knoten Optimierung

- Das Spiel ***eqG*** ist sehr einfach zu lösen.
- Zwei Knoten v, v' sind äquivalent wenn (v,v') und (v',v) in der Winning Region des Duplicators enthalten sind.
- Äquivalente Knoten werden in der Winning Region zusammengefasst.

P_1 Knoten Optimierung

- Zwei P_1 Knoten e, e' sind äquivalent, wenn sie die gleichen Funktionen repräsentieren und die gleichen Nachfolger haben.
- Äquivalente Knoten werden in der Winning Region zusammengefasst.
- Vorherige Reduzierung der P_0 Knoten liefert bessere Reduzierung der P_1 Knoten

Strategie Auswahl

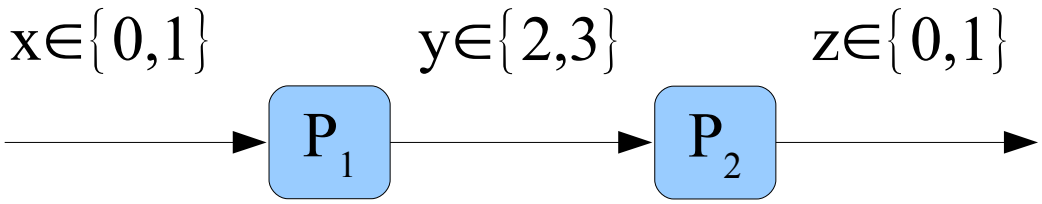
- Anzahl der P1 Knoten entspricht den Zuständen im erzeugten Programm.
- Suche eine gewinnende Strategie, die möglichst wenige P1 Knoten besucht.
- Breitensuche über die Winning Region

Strategie Auswahl

- Alle Plays müssen im Start Knoten des verteilten Spiels beginnen.
- Zähle alle Strategien mit einem P_1 Knoten auf.
- Ist eine gewinnende Strategie f dabei so wähle f.
- Sonst erhöhe die Anzahl der erlaubten P_1 Knoten um 1.

Gliederung

- Einleitung
- Architektur- und Spezifikationstransformation
- Verteilte Spiele
- Transformation von Siegbedingungen
- Erzeugung von Programmen
- Ergebnisse



$\Box(x = z)$

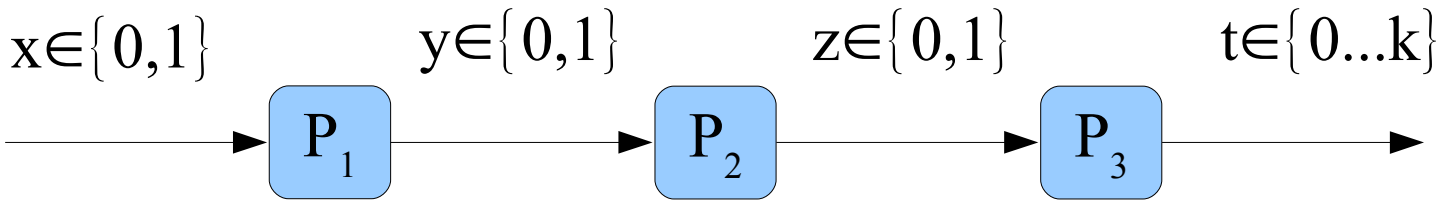
Optimierung	Nein	Ja
Verteiltes Spiel	22	15
Nach Glue( )	-	-
Laufzeit (ms)	110	40

$\Box(x \rightarrow \Diamond z)$

Optimierung	Nein	Ja
Verteiltes Spiel	82	82
Nach Glue( )	443	443
Laufzeit (s)	24,76	24,93

Implementierung: P_1: 0→2, 1→3; P_2: 2→0, 3→1

Beispiel 2



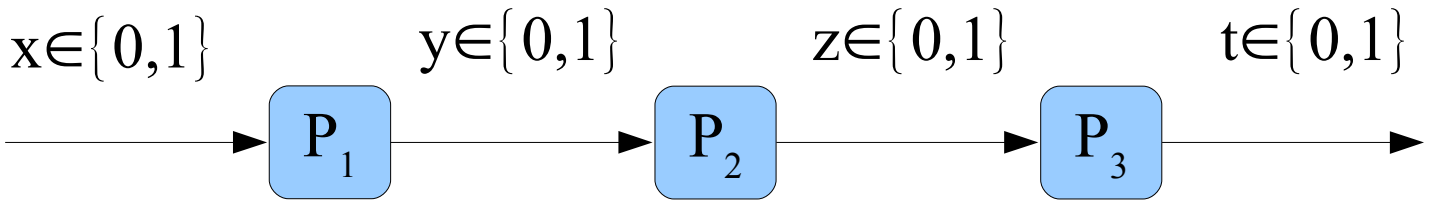
□(y)

Architektur A_i hat i+1 mögliche Werte für t

Größe Prozessspiel $O(|\text{output}|^{|\text{input}|})$

Architektur	A1		A5		A10		A15	
Optimierung	Nein	Ja	Nein	Ja	Nein	Ja	Nein	Ja
Verteiltes Spiel	70	21	582	149	1942	489	7062	1763
Nach Glue( )	467	34	4051	291	13571	971	49411	3528
Laufzeit	1:27.48	0:03.23	34:21.64	1:40.81	5:46:47.11	32:29.16	†	18h

Beispiel 3



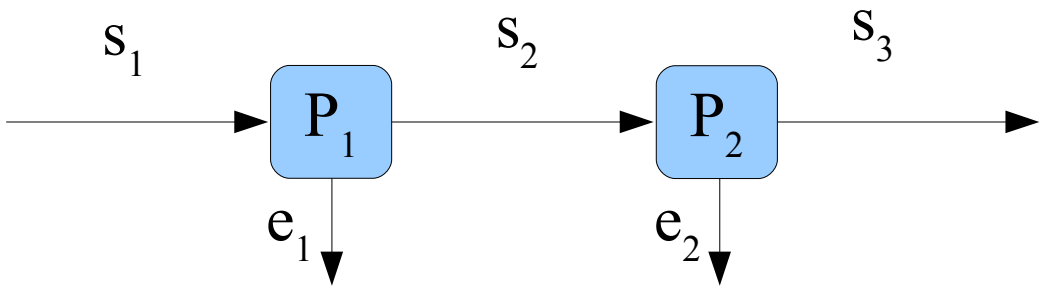
$$\Box(y) \wedge \Diamond(y = 0)$$

	Knoten
Verteiltes Spiel	138
Nach Glue( )	860
Optimierung I	41
Optimierung II	0
Nach Glue( )	-

- Es werden alle Knoten gelöscht.
- 60ms vs. 14 min

Dining Philosophers:

Architektur:

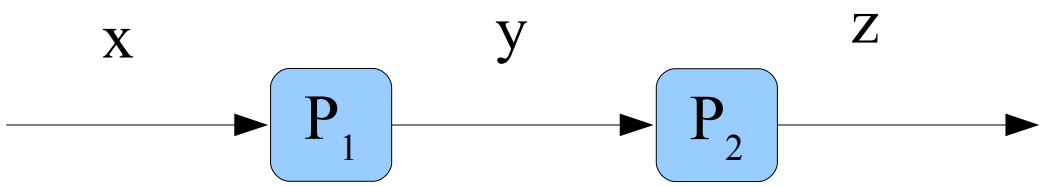


Spezifikation:

$[]<>(!s_1) \rightarrow ([](e_1 \rightarrow (!s_1 \wedge s_2) \wedge []<>(e_1) \wedge [](e_2 \rightarrow (!s_2 \wedge s_3) \wedge []<>(e_2))$

Spezifikationsspiel:	mit BA Opt.	ohne BA Opt.
2 Philosophen	99	1107
3 Philosophen	1683	24288
4 Philosophen	8385	125388

Architektur:



Spezifikation:

$(x \rightarrow ([]y)) \wedge (!x \rightarrow([]!y))$

Erzeugtes Programm mit Optimierung: 3 Zustände
ohne Optimierung: 5 Zustände

So sehen wir betroffen
den Vorhang zu und alle Fragen offen.

Vielen Dank!