Systemtheorie 1

Formale Systeme 1

#342234

http://fmv.jku.at/fs1

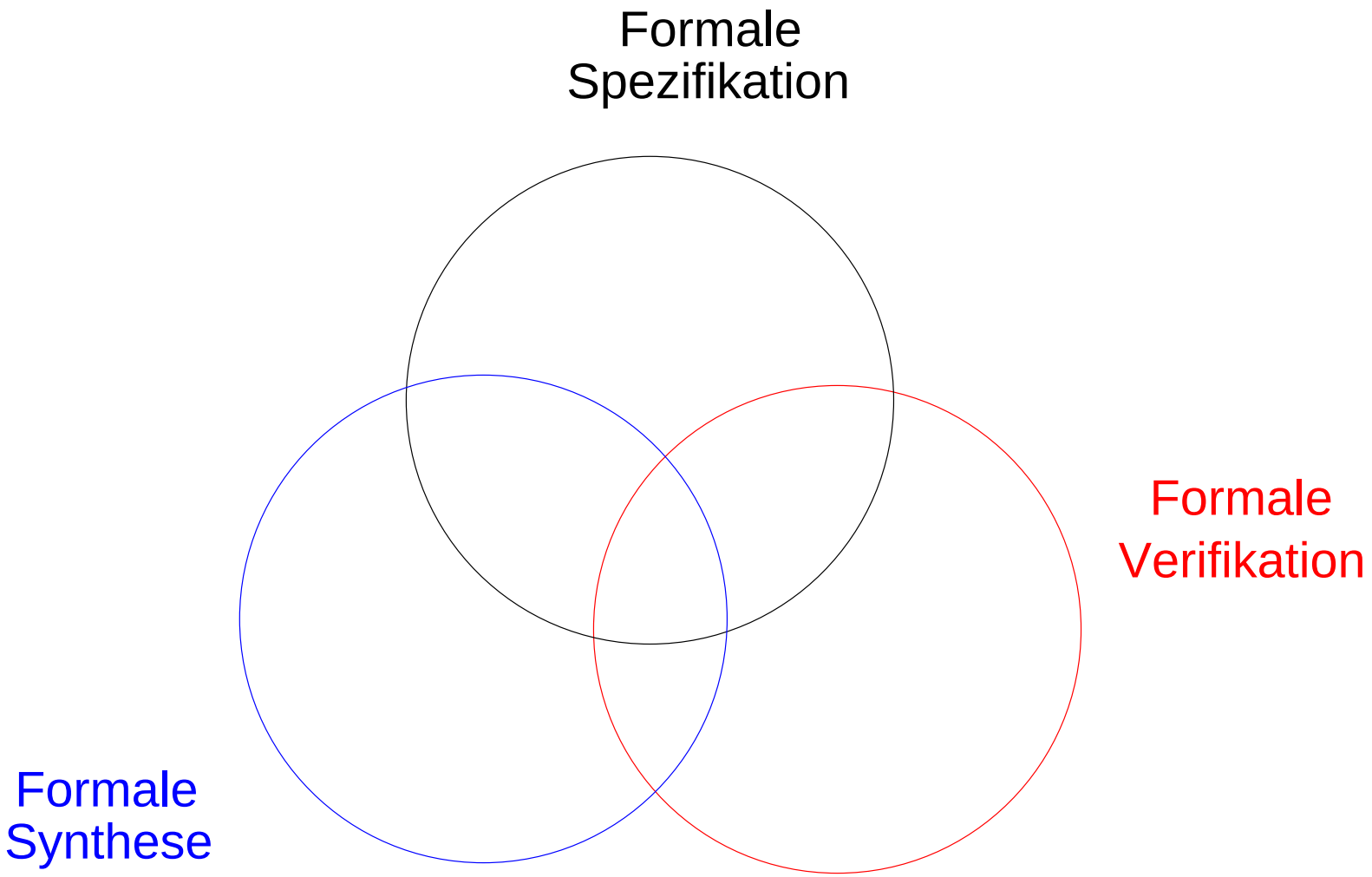
WS 2006/2007

Johannes Kepler Universität

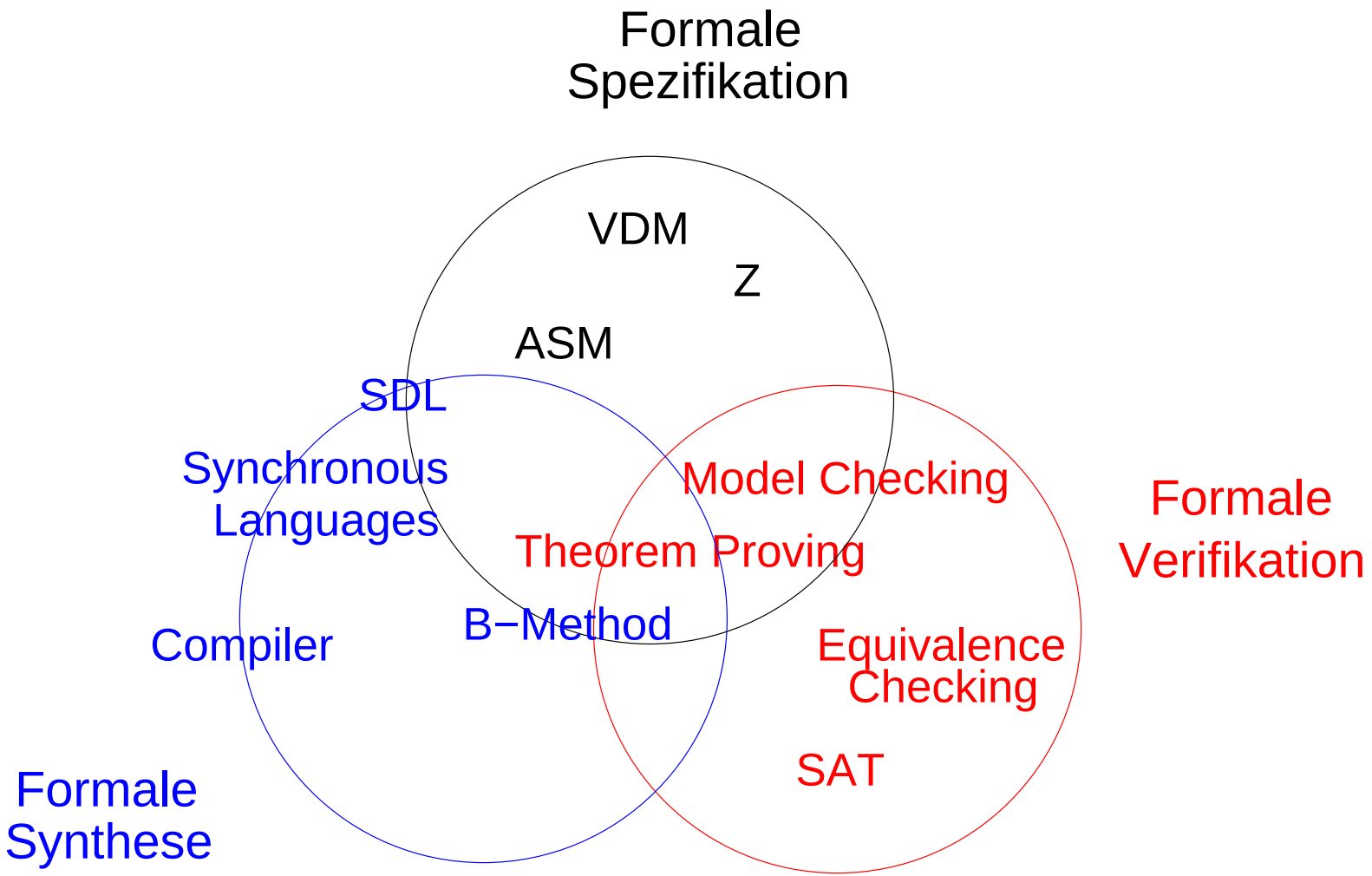
Linz, Österreich

Univ. Prof. Dr. Armin Biere
Institut für Formale Modelle und Verifikation
http://fmv.jku.at

Grobklassifizierung Formaler Methoden in der Informatik



Spezifische Formale Methoden



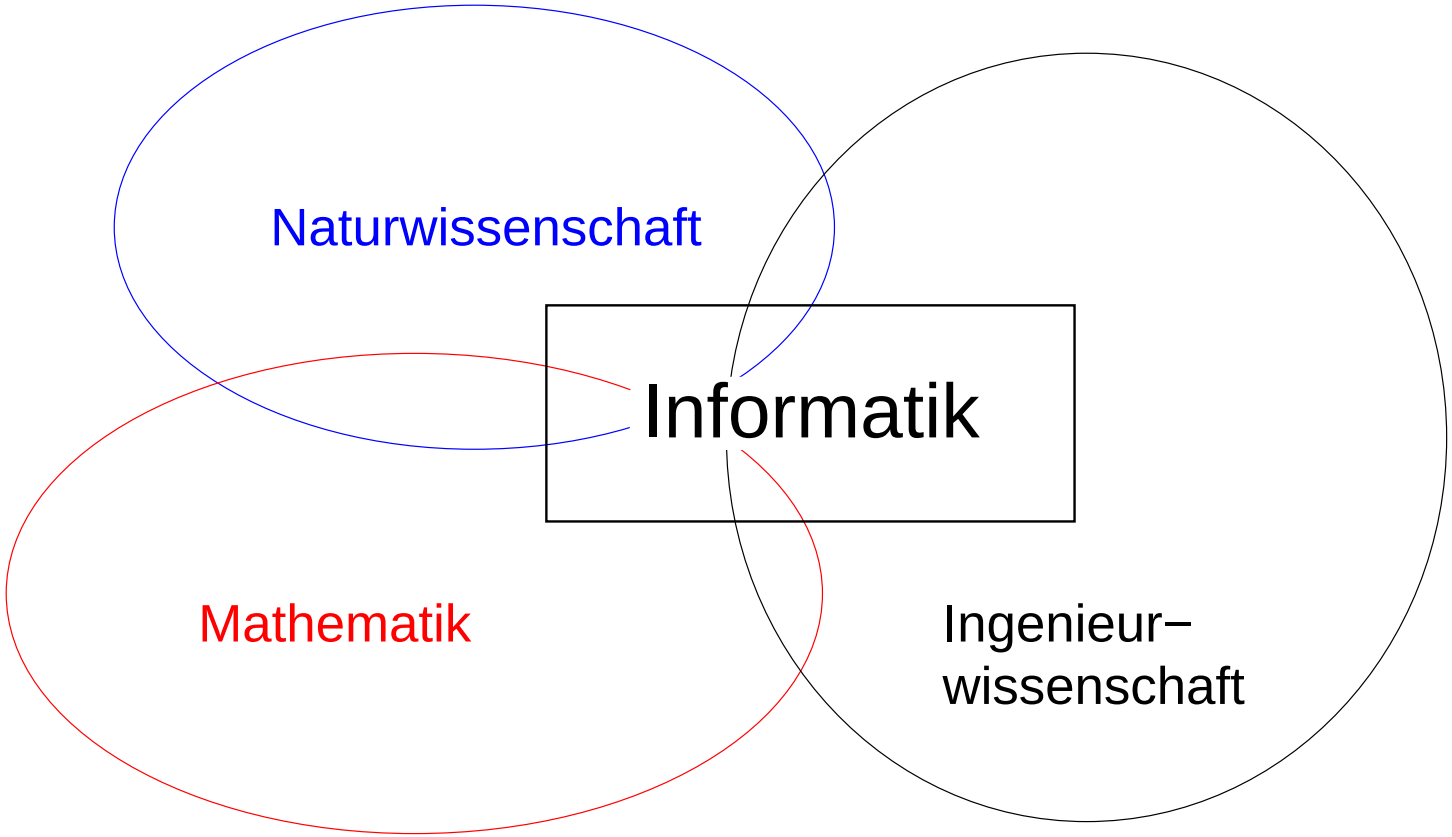
Einordnung der Vorlesung

- Formale Grundlagen III (4. Semester)
  - Formale Modellierung von Informatik-Systemen (meist diskret)
- Systemtheorie I (Formale Systeme I, 5. Semester)
  - Algorithmische Aspekte der Verifikation inkl. Simulation im Allgemeinen
- Systemtheorie II (Formale Systeme II, Model Checking, $>$ 5. Semester)
  - Spezifische Algorithmische Aspekte des Model Checking mit SAT

Motivation zur Beschäftigung mit Formalen Methoden

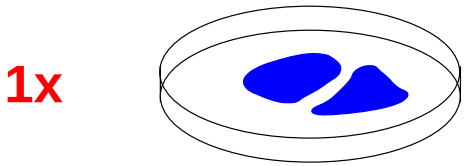
- Formale Methoden als Rückrat der Informatik
  - Abstraktion ist das Kern-Geschäft eines Informatikers
  - Informatik-Systeme **sind** mathematische Gebilde
  - Abstraktionen stellen sich als formale Modelle dar
- Formale Technologien breiten sich in der Praxis aus
  - Simulation abstrakter Modelle zur Validierung oder Optimierung
  - Beispiel: tagtäglicher Gebrauch von Equivalence Checking beim Chip-Design

Einordnung/Herkunft der Informatik



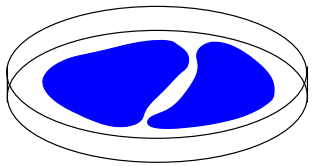
Modelle in der Naturwissenschaft

Größenwachstum einer Bakterienpopulation



1x

18h



2x

Modellierung

$$\frac{\Delta P(t)}{\Delta t} = \alpha P(t)$$

ergibt Differentialgleichung

$$\frac{d P(t)}{d t} = \alpha P(t)$$

mit Lösung

$$P(t) = C e^{\alpha t}$$

Validierung

Realität
(Natur)

Modell
(Mathematik)

Computational Science

- Modellierung und Auswertung von Computer-Modellen
  - von Phänomenen der Natur- aber auch Ingenieurwissenschaften
  - erlaubt Projektion in die Zukunft ...
    z.B. Wettervorhersage
  - ... und Optimierung, wenn der Mensch eingreifen kann
    z.B. Umkehr der Klima-Erwärmung durch weniger Kohlendioxid-Ausstoß
- Beschreibung der Zusammenhänge wiederum durch Gleichungen
  - Gleichungen können i.Allg. **nicht geschlossen** gelöst werden
  - Validierung durch Simulation mit numerischen Methoden
- **Computational Science als Teil der Informatik (?!)**

Empirische Modelliermethodik in der Informatik

am Beispiel Cocomo nach [Boehm81]

Wieviel kostet die Herstellung eines Programmes einer bestimmter Größe?

Applikations-Programme:	PM	=	$2.4 \cdot (\text{KDSI})^{1.05}$
Hilfs-Programme:	PM	=	$3.0 \cdot (\text{KDSI})^{1.12}$
System-Programme:	PM	=	$3.6 \cdot (\text{KDSI})^{1.20}$

PM = Personen-Monate (Kosten)

KDSI = Kilo Delivered Source Instructions (Größe)

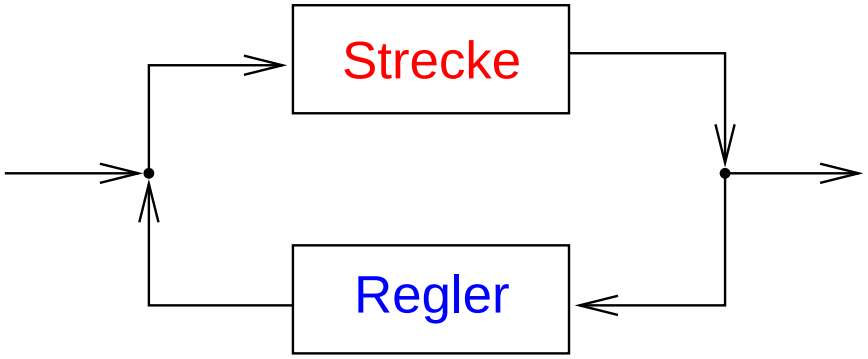
Typische Verwendung **empirischer Methoden** gerade im Software Engineering

Modelle in der Mathematik

- Gegenstand der Mathematik sind eigentlich keine Modelle
  - die Natürlichen Zahlen sind “einfach da”
    Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk [Kronecker]
  - keine Interpretation notwendig: Untersuchungsgegenstand = Realität
- Modellbegriff in der Mathematischen Logik
  - Lässt sich Mathematik durch Mathematik exakt modellieren?
    Nein, nicht im Allgemeinen [Gödel]
  - gewisse Formalisierung sind dennoch möglich:
    eine Vielzahl von Sätzen läßt sich **formal herleiten**
    (d.h. durch symbolisches Schließen in einem formalen Kalkül)

- Programme und Digitale Systeme sind formale Objekte
  - besitzen exakte mathematische Modelle (denotational oder operational)
  - Realität = Modell
    (modulo komplexer Semantik, Compilerbugs, Hardwareversagen, ...)
  - **Eigenschaften der Modelle gelten auch für die Realität**
- Eigenschaften von Modellen zu beweisen ist aufwendig
  - für Software i.Allg. **unentscheidbar**
  - für Hardware i.Allg. in **NP** oder **PSPACE**
- gilt nur für **Funktionale Eigenschaften**, nicht für **Quantitative Aspekte**
  - Verfügbarkeit, Durchsatz, Latenzzeit etc. sind schwer exakt zu modellieren

Modellierung eines Reglers im Einsatz



- der Eingriff des Mensches bzw. das technische System wird mitmodelliert
- Modellierung ist Approximation der Realität
  - reales System verhält sich nicht exakt wie das Modell
- Ziel ist Konstruktion und Optimierung des Reglers

Ingenieursansatz bei der Modellierung für die Informatik

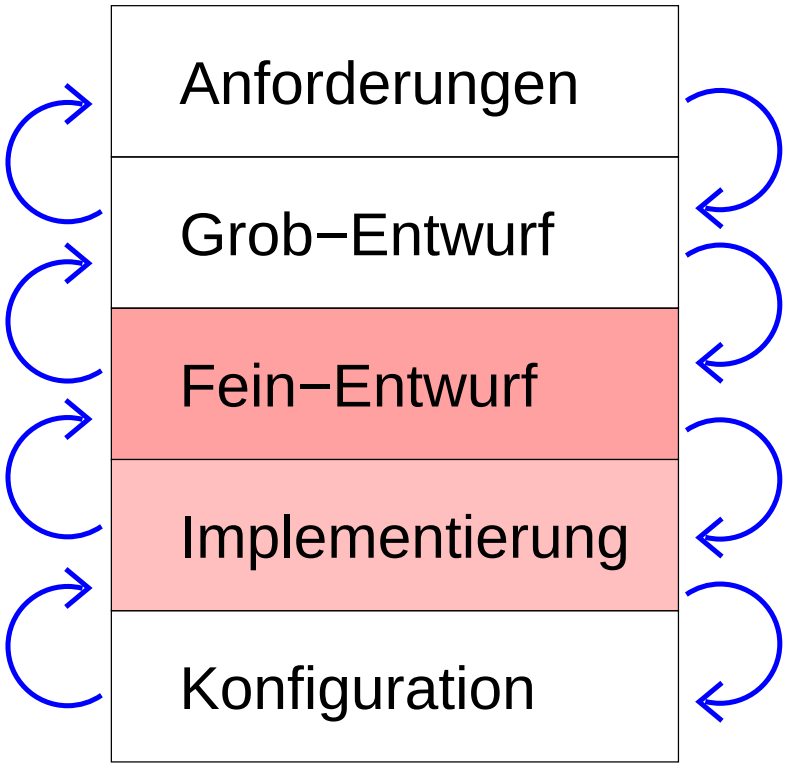
- Ziel ist Konstruktion und Optimierung von **Informatiksystemen**
- Modelle zur **quantitativen** Analyse/Optimierung
  - Warteschlangensysteme, Markov-Chains etc.
  - Simulation von Szenarien
- High-Level Modelle
  - schrittweise Verfeinerung/Synthese (z.B. Code-Erzeugung, Compiler)
  - Beispiel: Model Driven Architecture (MDA) = ausführbare UML Modelle
  - Beispiel: Behavioural Models und Synthesis für Digitale Systeme

Modellbegriff in der Informatik

- Modelle im Sinne der Naturwissenschaft
  - Computational Science, Modellierung mit dem Computer
- Weitere Empirische Modelle
  - empirische Methoden im Software Engineering
- Mathematische Modelliermethodik
  - Logik als Basis, SW/HW als Formel, Realität = Modell
- High-Level Modelle
  - qualitativ (funktional) oder quantitativ, Verfeinerung/Synthese

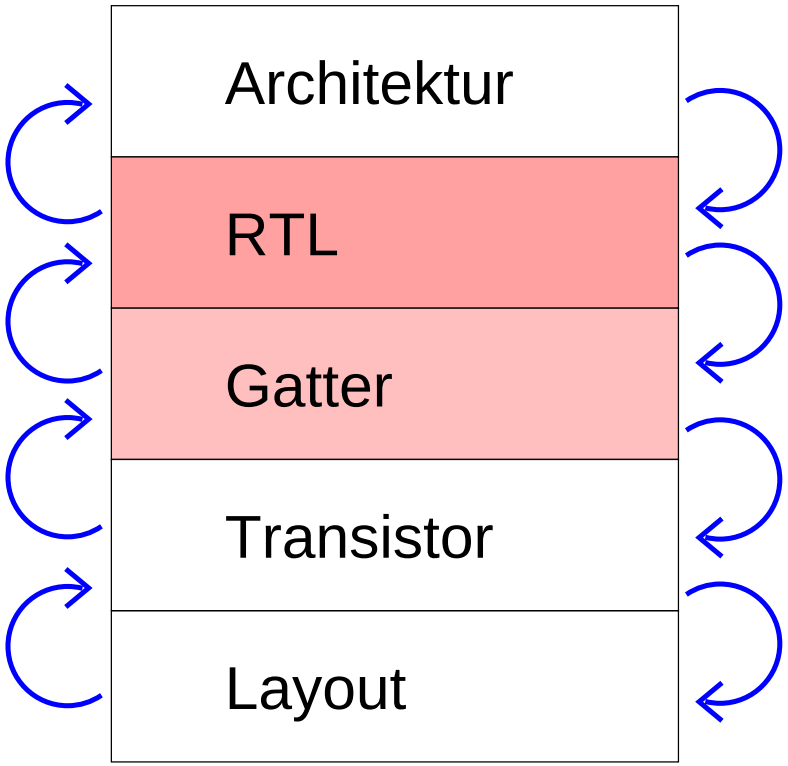
Verfeinerung SW

am Beispiel Software-Entwurf



Verfeinerung HW

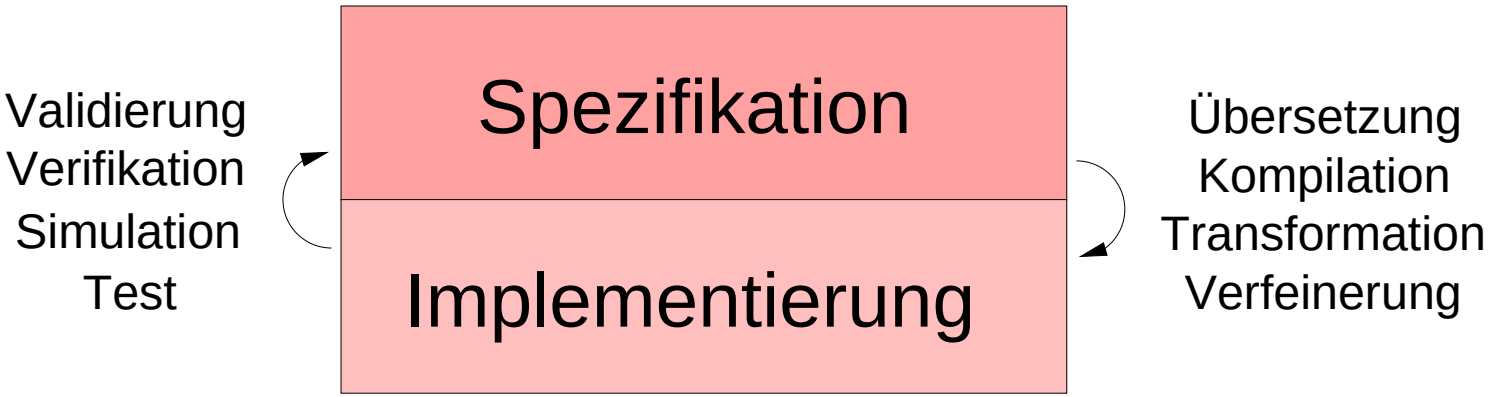
am Beispiel Hardware-Entwurf



Spezifikation vs Implementierung

Verifikation

Synthese



formal oder adhoc

Endlicher Automat (EA)

siehe auch Formale Grundlagen 3

Motivation: Automaten für die Modellierung, Spezifikation und Verifikation verwenden!

Definition Ein *Endlicher Automat* $A = (S, I, \Sigma, T, F)$ besteht aus

- Menge von Zuständen S (normalerweise endlich)
- Menge von Initialzuständen $I \subseteq S$
- Eingabe-Alphabet Σ (normalerweise endlich)
- Übergangsrelation $T \subseteq S \times \Sigma \times S$
  schreibe $s \xrightarrow{a} s'$ gdw. $(s, a, s') \in T$ gdw. $T(s, a, s')$ "gilt"
- Menge von Finalzuständen $F \subseteq S$

Sprache eines EA

Definition Ein EA A akzeptiert ein Wort $w \in \Sigma^*$ gdw. es s_i und a_i gibt mit

$$s_0 \stackrel{a_1}{\rightarrow} s_1 \stackrel{a_2}{\rightarrow} s_2 \stackrel{a_3}{\rightarrow} \ldots \stackrel{a_{n-1}}{\rightarrow} s_{n-1} \stackrel{a_n}{\rightarrow} s_n,$$

wobei $n \geq 0$, $s_0 \in I$, $s_n \in F$ und $w = a_1 \cdots a_n \quad (n = 0 \Rightarrow w = \varepsilon)$.

Definition Die Sprache $L(A)$ von A ist die Menge der Wörter die er akzeptiert.

- Benutze Automaten oder reguläre Sprachen zur Beschreibung von Ereignisströmen!
- “Konformität” der Ereignisströme der Implementierung zu denen der Spezifikation!
- Konformität kann präzisiert werden als Teilmengenbeziehung der Sprachen

Produkt-Automat

Definition Der Produkt Automat $A = A_1 \times A_2$ von zwei EA A_1 und A_2 mit gemeinsamen Eingabealphabet $\Sigma_1 = \Sigma_2$ hat folgende Komponenten:

$$S = S_1 \times S_2 \qquad I = I_1 \times I_2$$

$$\Sigma = \Sigma_1 = \Sigma_2 \qquad F = F_1 \times F_2$$

$$T((s_1, s_2), a, (s_1', s_2')) \quad \text{gdw.} \quad T_1(s_1, a, s_1') \text{ und } T_2(s_2, a, s_2')$$

Satz Seien A, A_1, und A_2 wie oben, dann $L(A) = L(A_1) \cap L(A_2)$

Beispiel: Konstruktion eines Automaten Wörter mit Prefix ab und Suffix ba akzeptiert.

(als regulärer Ausdruck: $a \cdot b \cdot \mathbf{1}^* \cap \mathbf{1}^* \cdot b \cdot a$, wobei $\mathbf{1}$ für alle Buchstaben steht)

Vollständigkeit und Determinismus

Definition Zu $s \in S$, $a \in \Sigma$ bezeichne $s \xrightarrow{a}$ die Menge der Nachfolger von s definiert als

$$s \xrightarrow{a} = \{s' \in S \mid T(s,a,s')\}$$

Definition Ein EA ist *vollständig* gdw. $|I| > 0$ und $|s \xrightarrow{a}| > 0$ für alle $s \in S$ und $a \in \Sigma$.

Definition ... *deterministisch* gdw. $|I| \leq 1$ und $|s \xrightarrow{a}| \leq 1$ für alle $s \in S$ und $a \in \Sigma$.

Fakt ... deterministisch und vollständig gdw. $|I| = 1$ und $|s \xrightarrow{a}| = 1$ für alle $s \in S$, $a \in \Sigma$.

Teilmengenkonstruktion

Definition Der Power-Automat $A = P(A_1)$ eines EA A_1 hat folgende Komponenten

$$S = P(S_1) \quad (P = \text{Potenzmenge}) \qquad I = \{I_1\}$$

$$\Sigma = \Sigma_1 \qquad F = \{F' \subseteq S \mid F' \cap F_1 \neq \emptyset\}$$

$$T(S', a, S'') \quad \text{gdw.} \quad S'' = \{s'' \mid \exists s' \in S' \text{ mit } T(s', a, s'')\}$$

Satz A, A_1 wie oben, dann $L(A) = L(A_1)$ und A ist deterministisch und vollständig.

Beispiel: Spam-Filter basierend auf der White-List “abb”, “abba”, und “abacus”!

(Regulärer Ausdruck: “abb” | “abba” | “abacus”)

Komplement eines Automaten

Definition Der Komplementär-Automat $A = K(A_1)$ eines endlichen Automaten A_1 hat dieselben Komponenten wie A_1, bis auf $F = S \backslash F_1$.

Satz Der Komplementär-Automat $A = K(A_1)$ eines deterministischen und vollständigen Automaten A_1 akzeptiert die komplementäre Sprache $L(A) = \overline{L(A_1)} = \Sigma^* \backslash L(A_1)$.

Beispiel: Spam-Filter basierend auf der Black-List “abb”, “abba”, und “abacus”!

(Regulärer Ausdruck: $\overline{\text{“abb”} \mid \text{“abba”} \mid \text{“abacus”}}$)

Konformität von Ereignisströmen

- Modellierung und Spezifikation mit Automaten:
  - Ereignisströme einer Implementierung modelliert durch EA A_1
  - Partielle Spezifikation der Ereignisströme als EA A_2
- Konformitäts-Test:
  - $L(A_1) \subseteq L(A_2)$
  - gdw. $L(A_1) \cap \overline{L(A_2)} = \emptyset$
  - gdw. $A_1 \times K(P(A_2))$ keinen erreichbaren Finalzustand hat
- **Beispiel:** Spezifikation $S = (cs \,|\, sc \,|\, ss)^*$, Implementierung $I = \left((s \,|\, c)^2\right)^*$

Weitere Beispiele zur Konformität

- Temporale Eigenschaften: ($\mathbf{1}$ steht für ein beliebiges Zeichen)
  - jeden dritten Schritt gilt a: $(\mathbf{1} \cdot \mathbf{1} \cdot a)^*$
  - genau jeden dritten Schritt gilt a: $(\overline{a} \cdot \overline{a} \cdot a)^*$
  - einem a (acknowledge) muss ein r (request) vorangehen: $\overline{(\overline{r})^* \cdot a}$
  - jedem a muss ein r vorangehen: $\overline{(\mathbf{1}^* \cdot a)^* \cdot (\overline{r})^* \cdot a}$

- Verfeinerung: (am Beispiel Scheduling dreier Prozesse a, b und c)
  - abstrakter Round-Robin Scheduler: $(abc \mid acb \mid bac \mid bca \mid cab \mid cba)^*$
  - Round-Robin Scheduler mit Vorrang a vor b: $(abc \mid acb \mid cab)^*$
  - Round-Robin Scheduler mit Vorrang a vor b und c vor b: $(acb \mid cab)^*$
  - deterministischer Round-Robin Scheduler der Implementierung: $(cab)^*$

Konformität bei Aussagenlogischen Spezifikationen

- ähnliche Vorgehensweise:
  - geg. aussagenlogische Formel f über den booleschen Variablen $V = \{x_1, \ldots, x_n\}$
  - Expansion $E(f) \subseteq 2^n$ ist die Menge der erfüllenden Belegungen von f

    $$(a_1, \ldots, a_n) \in E(f) \quad \text{gdw.} \quad f[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n] = 1$$

  - z.B. $E(f) \neq \emptyset$ gdw. f erfüllbar (engl. satisfiable)

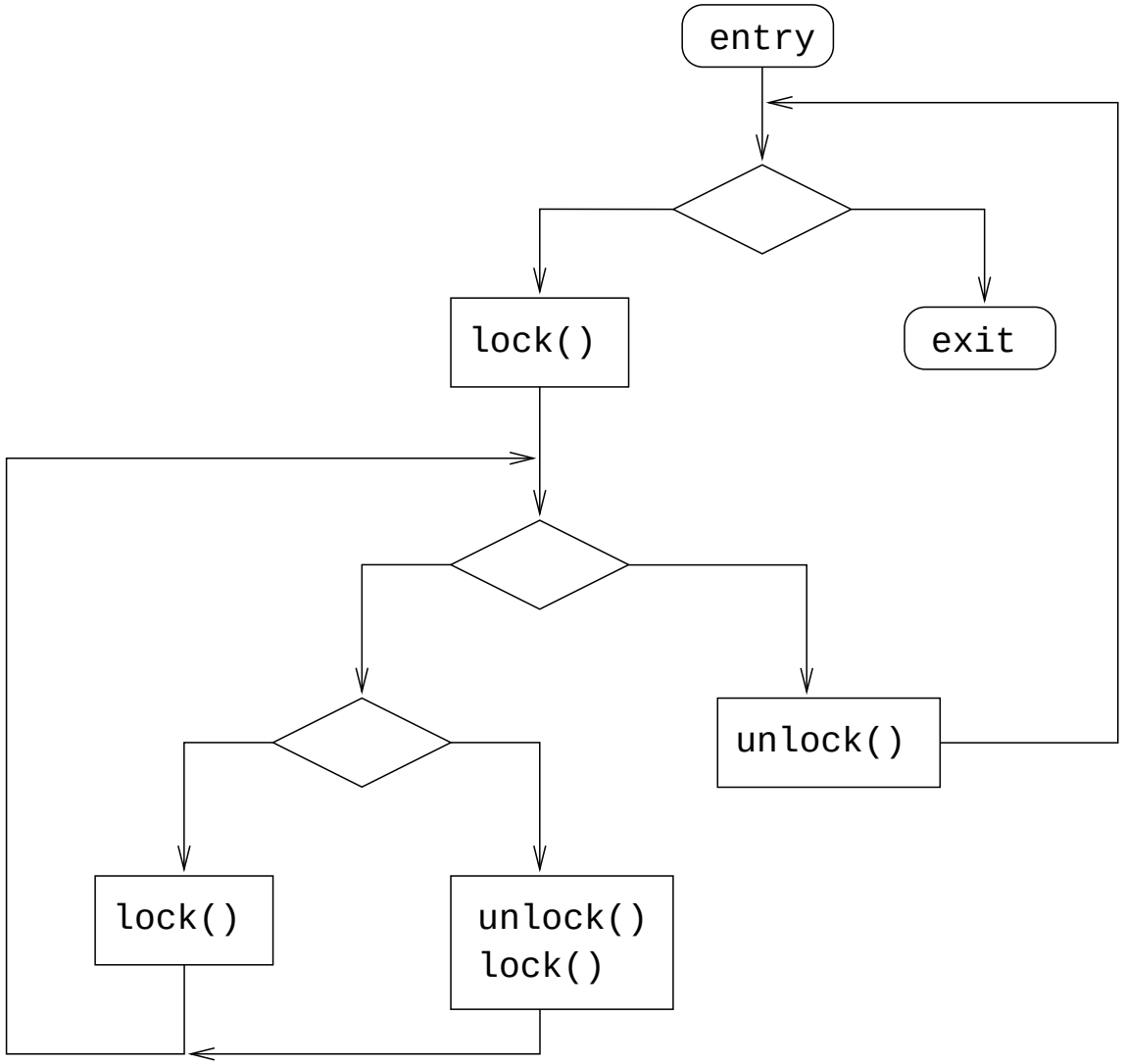
- Modellierung und Spezifikation:
  - f_1 charakterisiert die für die Implementierung möglichen Konfigurationen
  - f_2 beschreibt eine partielle Spezifikation der gültigen Konfigurationen

- Konformitäts-Test: $f_1 \Rightarrow f_2$ gdw. $E(f_1) \subseteq E(f_2)$ gdw. $E(f_1) \cap \overline{E(f_2)} = \emptyset$

  (oder in der Praxis: ... gdw. $f_1 \wedge \neg f_2$ unerfüllbar)

Vom C-Programm zum EA durch Abstraktion


```
while (...) {
  lock ();
  ...
  while (...) {
    if (...) {
      lock ();
      ...
    } else {
      unlock ();
      ...
      lock ();
    }
    ...
  }
  ...
  unlock();
}
```



$$(l \cdot (l \mid u \cdot l)^* \cdot u)^*$$

Probleme bei Abstraktion

```
assert (i < n);
lock ();
do {
  ...
  i++;
  if (i >= n)
    unlock ();
  ...
} while (i < n);
```

$l \cdot (\varepsilon | u)^*$ verletzt partielle Spezifikation $\overline{\mathbf{1}^* \cdot l \cdot \overline{u}^*}$

("`...`" führt weder zu einem `lock` noch `unlock` und lässt `i` und `n` unberührt)

Predicate Abstraction

verfeinerte Abstraktion zu EA durch Einführung einer Prädikats-Variable: $\texttt{b} == (i < n)$

```
assert (i < n);
lock ();
do {
  ...
  i++;
  if (i >= n)
    unlock ();
  ...
} while (i < n);
```

abstrahiert zu

```
assert (b);
lock ();
do {
  ...
  if (b) b = *;
  if (!b)
    unlock ();
  ...
} while (b);
```

$l \cdot \varepsilon^* \cdot u$ erfüllt partielle Spezifikation $\overline{\mathbf{1}^* \cdot l \cdot \overline{u}^*}$

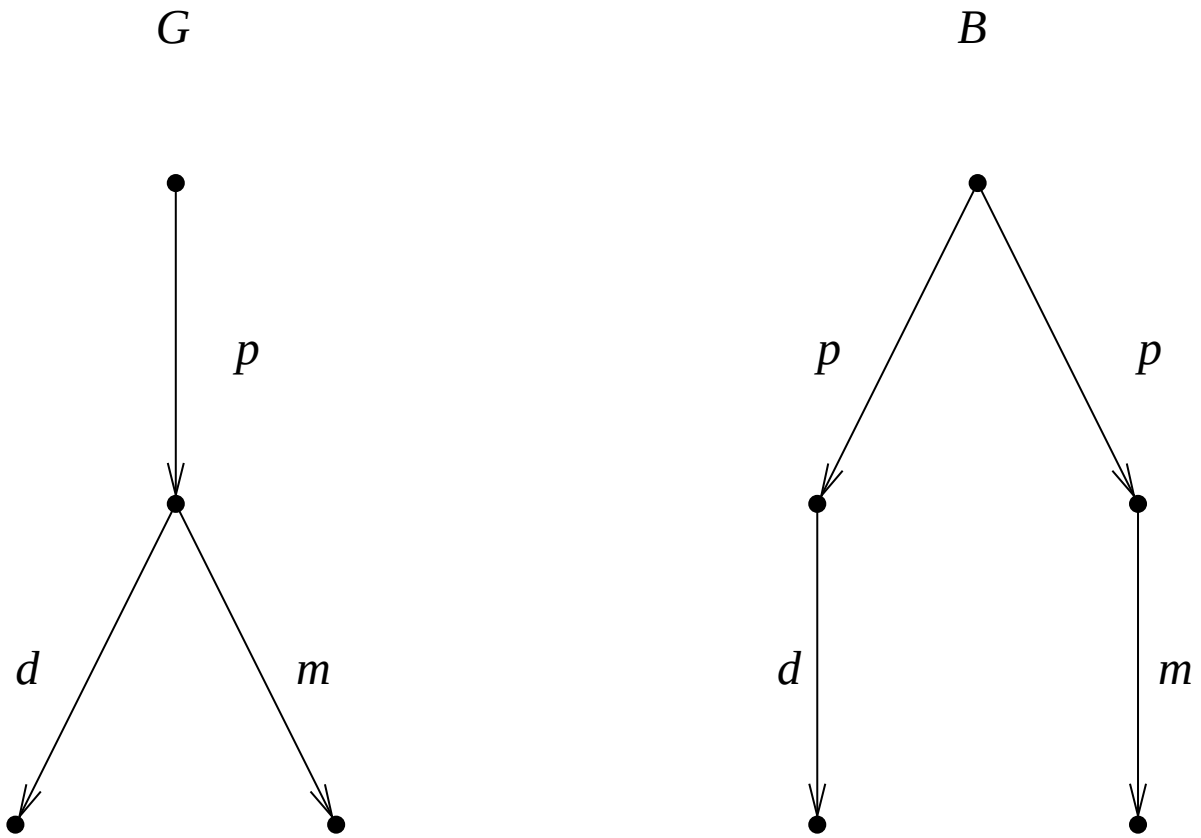
("..." führt weder zu einem `lock` noch `unlock` und lässt `i` und `n` unberührt)

Labelled Transition Systems (LTS)

- semantisches Modell im Kontext von **Process Algebra**
  - Fokus auf **Reaktive Systeme** oder **Offene Systeme**
  - dabei Konzept der **Umgebung** mit **externen** Ereignissen
  - Implementierung (auf einer Abstraktionsebene) bestimmt die **internen** Ereignisse

- ein LTS $A = (S, I, \Sigma, T)$ ist im wesentlichen ein EA:
  - das Verhalten, die Möglichkeit von Übergängen zählt
  - ohne Final-Menge, d.h. auch ohne “explizite” Sprache
  - “implizite” Sprache $L(A)$ durch $F = S$

Schokolade-Automat von Milner



p = Einwurf des Geldbetrages (pay)

d = Auswahl und Ausgabe der dunklen Schokolade

m = Auswahl und Ausgabe der Milch-Schokolade

Sprachäquivalenz Inadequat

- sicherlich sollte die Semantik der beiden LTS “unterschiedlich” sein
  - G erlaubt nach dem Geldeinwurf Wahl der Schokoladenart
  - B setzt nicht-deterministisch beim Geldeinwurf die Schokoladenart fest
- aber B und G sind **sprachäquivalent**:
  - $L(B) = p \cdot (d \mid m) = L(G)$
- Problem überträgt sich auf Konformitäts-Test:
  - Sprach-basierter Konformitäts-Test identifiziert B und G
  - Sprach-Konformität ignoriert das “Branching-Verhalten”

Intuition des Simulationsbegriffes

- Verhalten der Implementierung A_1 sollte gültiges Verhalten der Spezifikation A_2 sein
  - jeder **Übergang** in A_1 hat eine Entsprechung in A_2
  - A_2 **simuliert** A_1
  - A_2 kann möglicherweise mehr

- Vereinfachung der formalen Notation durch Vereinigung zu einem LTS A
  - gemeinsames Alphabet Σ
  - (disjunkte) Vereinigung der anderen Komponenten:

    $S = S_1 \dot\cup S_2, I = I_1 \dot\cup I_2, T = T_1 \dot\cup T_2$
  - Schreibweise: $A = A_1 \dot\cup A_2$

Definition Simulations-Relation

Definition eine Relation $\lesssim \subseteq S \times S$ über LTS A ist eine **Simulation** gdw.

(man ließt $s \lesssim t$ als t simuliert s)

$$s \lesssim t \quad \text{dann} \quad \forall a \in \Sigma,\, s' \in S\,[s \xrightarrow{a} s' \;\Rightarrow\; \exists\, t' \in S\,[\,t \xrightarrow{a} t' \;\wedge\; s' \lesssim t']]$$

Fakt es gibt genau eine maximale Simulation über jedem LTS A

Beweisskizze (S endlich)

- die Vereinigung zweier Simulationen ist wiederum eine Simulation
- die Menge der Simulationen über A ist nicht leer (enthält die Identität)

Konstruktive Bestimmung der Maximalen Simulation

- Ausgangspunkt: $\lesssim_0 = S \times S$ (normalerweise keine Simulation)

- verfeinere $\lesssim_i$ zu $\lesssim_{i+1}$ wie folgt

  $s \lesssim_{i+1} t$ gdw. $s \lesssim_i t$ und $\forall a \in \Sigma,\, s' \in S\,[s \xrightarrow{a} s' \Rightarrow \exists\, t' \in S\,[\, t \xrightarrow{a} t' \wedge s' \lesssim_i t']]$

- bei endlichem S gibt es ein n mit $\lesssim_n = \lesssim_{n+1}$

  - $\lesssim_n$ ist offensichtlich eine Simulation
  - Maximalität schwerer einzusehen

- kann als Fixpunkt-Prozess reformuliert werden

Beweis Konstruktion liefert Maximale Simulation

Sei $\lesssim$ eine Simulation. Zeige $\lesssim \subseteq \lesssim_i$ durch Induktion über i.

Induktionsanfang ist trivial, Induktionsschritt folgt.

Annahme (indirekter Beweis): $\lesssim \not\subseteq \lesssim_{i+1}$.

Dann gibt es s und t mit $s \lesssim t$ aber $s \not\lesssim_{i+1} t$.

Somit muss es s' und a geben mit $s \xrightarrow{a} s'$, aber $t \not\xrightarrow{a} t'$ oder $t \not\lesssim_i t'$ für alle t'.

Mit der Induktionshypothese $\lesssim \subseteq \lesssim_i$ folgt: $t \not\xrightarrow{a} t'$ oder $t \not\lesssim t'$ für alle t'.

Widerspruch zur Voraussetzung $\lesssim$ Simulation.

Weitere Fakten über Simulationen

Fakt maximale Simulation ist eine Halb-Ordnung (insbesondere transitiv)

Beweisskizze

- Reflexivität siehe vorige Beweisskizze
- Transitivität folgt aus unterem Lemma

Lemma transitive Hülle einer Simulation ist wieder eine Simulation

Beweisskizze folgender Operator erhält die Simulationseigenschaft

$$\Psi: P(S \times S) \to P(S \times S) \qquad \Psi(\lesssim)(r,t) \quad \text{gdw.} \quad r \lesssim t \quad \text{oder} \quad \exists s[r \lesssim s \wedge s \lesssim t]$$

Simulation von LTS

Definition LTS A_2 simuliert LTS A_1 gdw. es eine Simulation $\lesssim$ über $A_1 \cup A_2$ gibt, so dass für jeden Anfangszustand $s_1 \in S_1$ von A_1, es einen Anfangszustand $s_2 \in S_2$ von A_2 gibt, mit $s_1 \lesssim s_2$. Man schreibt dann auch $A_1 \lesssim A_2$.

Fakt Simulation von LTS ist eine Halb-Ordnung (insbesondere transitiv)

Beweisskizze

- bilde maximale Simulationsrelation über alle drei LTS
- zeige Existenz von simulierenden Anfangszuständen
- Projektion auf äussere LTS liefert gewünschte Simulation

Simulation von Traces

Definition Ein *Trace* eines LTS A ist ein Wort $w = a_1 \cdots a_n \in \Sigma^*$ mit

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s_n,$$

wobei $s_0 \in I$ und $n \geq 0$.

Fakt $L(A) = \{w \mid w \text{ Trace von } A\}$

Satz (Simulation ist eine konservative Abstraktion)

LTS A_2 simuliere A_1 via Simulation $\lesssim$ (also $A_1 \lesssim A_2$), dann gilt $L(A_1) \subseteq L(A_2)$.

Anwendung $P \lesssim A \leq S \quad \Rightarrow \quad L(P) \subseteq L(S)$

(P = Programm, A Abstraktion, S Spezifikation)

Wenn man nur an der Sprache bzw. den Traces interessiert ist, kann man dennoch die Abstraktion immer so konstruieren, dass das Programm simuliert wird.

Schwache Simulation

- $\tau \in \Sigma$ bezeichnet ein nicht beobachtbares *internes Ereignis*
- vorherige Definition der Simulation ist dann eine **starke Simulation**

$$s \lesssim t \quad \text{dann} \quad \forall a \in \Sigma,\, s' \in S\,[s \xrightarrow{a} s' \;\Rightarrow\; \exists\, t' \in S\,[\,t \xrightarrow{a} t' \;\wedge\; s' \lesssim t']]$$

- man schreibe $s \xrightarrow{\tau^* a} t$ falls es $s_0, \cdots, s_n$ gibt mit

$$s = s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_{n-1} \xrightarrow{a} s_n = t$$

- eine Relation $\lesssim$ ist eine **schwache Simulation** gdw.

$$s \lesssim t \quad \text{dann} \quad \forall a \in \Sigma\backslash\{\tau\},\, s' \in S\,[s \xrightarrow{\tau^* a} s' \;\Rightarrow\; \exists\, t' \in S\,[\,t \xrightarrow{\tau^* a} t' \;\wedge\; s' \lesssim t']]$$

Zur Schwachen Simulation

- Man verwende τ für *abstrahierte* Ereignisse
  - z.B. nebensächliche Berechnungen/Datenfluss im Programm

- τ-bereinigtes LTS A für ein LTS A_1 mit τ: $\Sigma = \Sigma_1 \setminus \{\tau\}$, $T(s,t)$ gdw. $s \xrightarrow{\tau^* a} t$ in A_1.
  - τ-Bereinigung macht aus schwacher Simulation eine starke (und umgekehrt)
  - damit lassen sich die vorherigen Algorithmen auch hier anwenden

- Transitivität und Anwendungen wie im starken Fall

- **Divergenz** $s \xrightarrow{\tau^+} s$ wird noch unzulänglich behandelt
  - $A_1 \lesssim A_2$ erlaubt A_1 divergent und A_2 nicht

Bisimulation

Idee: Implementierung des spezifizierten Verhaltens und nicht mehr !

Definition eine Relation $\approx$ ist eine **starke Bisimulation** gdw.

$$s \approx t \quad \text{dann} \quad \forall a \in \Sigma,\, s' \in S\,[s \xrightarrow{a} s' \Rightarrow\, \exists\, t' \in S\,[\, t \xrightarrow{a} t' \,\wedge\, s' \approx t']] \text{ und}$$
$$\forall a \in \Sigma,\, t' \in S\,[t \xrightarrow{a} t' \,\Rightarrow\, \exists\, s' \in S\,[\, s \xrightarrow{a} s' \,\wedge\, s' \approx t']]$$

Definition eine Relation $\approx$ ist eine **schwache Bisimulation** gdw.

$$s \approx t \quad \text{dann} \quad \forall a \in \Sigma\backslash\{\tau\},\, s' \in S\,[s \xrightarrow{\tau^* a} s' \Rightarrow\, \exists\, t' \in S\,[\, t \xrightarrow{\tau^* a} t' \,\wedge\, s' \approx t']] \text{ und}$$
$$\forall a \in \Sigma\backslash\{\tau\},\, t' \in S\,[t \xrightarrow{\tau^* a} t' \,\Rightarrow\, \exists\, s' \in S\,[\, s \xrightarrow{\tau^* a} s' \,\wedge\, s' \approx t']]$$

Insbesondere die schwache Bisimulation bei Abstraktion von internen Ereignissen der Implementierung durch τ ist in der Praxis sehr nützlich!

Theorie-Anwendung: bisimulations-äquivalente LTS haben die gleiche Eigenschaften

Exkurs über Minimierung von EA

Geg. deterministischer und vollständiger EA $A = (S, I, \Sigma, T, F)$

- Ausgangspunkt: $\sim_0 = (F \times F) \cup (\overline{F} \times \overline{F})$
  - Partitionierung bezüglich “Endzustands-Flag”
  - Äquivalenzrelation
- verfeinere $\sim_i$ zu $\sim_{i+1}$

  $s \sim_{i+1} t$ gdw. $s \sim_i t$ und

  $$\forall a \in \Sigma,\ s' \in S\ [s \xrightarrow{a} s' \Rightarrow\ \exists\, t' \in S\ [\, t \xrightarrow{a} t' \ \wedge\ s' \sim_i t']] \quad \text{und}$$

  $$\forall a \in \Sigma,\ t' \in S\ [t \xrightarrow{a} t' \Rightarrow\ \exists\, s' \in S\ [\, s \xrightarrow{a} s' \ \wedge\ s' \sim_i t']]$$

- Terminierung $\sim_{n+1} = \sim_n$ spätestens für $n = |S|$
- Äquivalenzrelation $\sim = \sim_n$ erzeugt minimalen Automaten $A/\sim$

Erreichbarkeitsanalyse

Definition Unter *Erreichbarkeitsanalyse* versteht man

- Berechnung aller erreichbaren Zustände
  - Resultat kann explizit oder symbolisch repräsentiert werden
  - wird für weitere Algorithmen oder Optimierung der Systeme benötigt
- Überprüfung, ob gewisse Zustände erreichbar sind
  - entspricht Model Checking von Sicherheitseigenschaften (Safety)

Symbolisch vs Explizit

explizit = jeder Zustand/Übergang wird einzeln repräsentiert

symbolisch = Mengen-Repräsentation von Zuständen/Übergängen durch Formeln

	Explizites Modell	Symbolisches Modell
Explizite Analyse	Graphensuche	Explicit Model Checking
Symbolische Analyse	—	Symbolic Model Checking

klassisch: Symbolic MC für HW, explicit MC für SW

(Praxis heute: symbolic und explicit für SW+HW)

Symbolisches Modell

- für die Theorie reicht explizite Matrix-Darstellung von T

Anfangszustand: 0

Übergangsmatrix	0	1	2	3	4	5	6	7
0	0	0	1	0	1	0	0	0
1	0	0	0	1	0	1	0	0
2	0	0	0	0	1	0	1	0
3	0	0	0	0	0	1	0	1
4	1	0	0	0	0	0	1	0
5	0	1	0	0	0	0	0	1
6	1	0	0	0	0	0	0	0
7	0	1	0	1	0	0	0	0

- in der Praxis ist T in einer Modellierungs- oder Programmiersprache beschrieben

```
initial_state = 0;

next_state = (current_state + 2 * (bit + 1)) % 8;
```

- symbolische Repräsentation kann exponentiell kompakter sein

Symbolische Erreichbarkeitsanalyse

- Symbolische Travesierung des Zustandraumes
  - von einer Menge von Zuständen werden in einem Schritt die Nachfolger bestimmt
  - Nachfolger werden symbolisch berechnet und repräsentiert
  - legt Breitensuche nahe
  - im allgemeinen schon nicht berrechenbar
- Resultat ist eine symbolische Repräsentation aller erreichbaren Zustände
  - z.B. `0 <= state && state < 8 && even (state)`

Parallele Komposition

- symbolische Repräsentation hat oft einen "Parallelоperator"
  - z.B. Produkt von Automaten, der einzelne Automat wird explizit repräsentiert

$$G = K_1 \times K_2$$

  - Komponenten des Systems werden seperat programmiert

```
process A begin P1 end || process B begin P2 end;
```

- Additives Eingehen der Größe der symbolischen Komponenten
  - Programmgrösse des Systems: $|K_1| + |K_2|$, bzw. $|\texttt{P1}| + |\texttt{P2}|$

Zustandsexplosion

- Multiplaktives Eingehen der Anzahl Komponentenzustände in Systemzustandszahl:

$$|S_G| = |S_{K_1}| \cdot |S_{K_2}|$$

- offensichtlich bei sequentieller Hardware:
  - n-Bit Zähler läßt sich mit $O(n)$ Gattern implementieren, hat aber 2^n Zustände
- Hierarchische Beschreibungen führen zu einem weiteren exponentiellen Faktor
- bei Software noch komplexer:
  - in der Theorie nicht primitiv rekursiv (siehe Ackerman-Funktion)
  - in der Praxis ist der Heap (dynamisch allozierte Objekte) das Problem

Explizite vs Symbolische Modellierung

- explizite Repräsentation reduziert sich auf Graphensuche
  - suche erreichbare Zustände von den Anfangszuständen
  - linear in der Grösse **aller** Zustände
- symbolische Modellierung
  - Berechnung der Nachfolger eines Zustandes ist das Hauptproblem
  - **on-the-fly** Expansion von T für einen konkreten erreichten Zustand
  - damit Vermeidung der Berechnung von T für unerreichbare Zustände
  - alternativ symbolische Berechnung/Repräsentation der Nachfolgezustände

Explizite Analyse durch Tiefensuche

- Markiere besuchte Zustände
  - Implementierung abhängig davon, ob Zustände **On-the-fly** generiert werden
- Unmarkierte Nachfolger besuchter Zustände kommen auf den Suchstack
- Nächster zu bearbeitender Zustand wird oben vom Stack genommen
- Falls “Fehlerzustand” erreicht wird hat man einen Weg dorthin
  - der Pfad bzw. Fehlertrace findet sich auf dem Stack
  - der Einfachheit wegen breche die Suche ab

Tiefensuche – Depth First Search (DFS)

```
recursive_dfs_aux (Stack stack, State current)
{
  if (marked (current))
    return;

  mark (current);
  stack.push (current);

  if (is_target (current))
    stack.dump_and_exit ();   /* target reachable */

  forall successors next of current
    recursive_dfs_aux (stack, next);

  stack.pop ();
}
```

Erreichbarkeits-Analyse mit DFS

```
recursive_dfs ()
{
  Stack stack;

  forall initial states state
    recursive_dfs_aux (stack, state);

  /* target not reachable */
}
```

- Abbruch beim Erreichen des Zieles ist schlechter Programmierstil!
- Effizienz verlangt nicht-rekursive Variante
- Markieren sollte durch Hashen ersetzt werden
  - damit Zustände on-the-fly *erzeugt* werden können

Stackoverflow bei Rekursion

```
#include <stdio.h>

void
f (int i)
{
  printf ("%d\n", i);
  f (i + 1);
}

int
main (void)
{
  f (0);
}
```

dieses C-Programm stürzt nach “wenigen” Rekursionsschritten ab

Nicht-Rekursive DFS – Korrekte Version

```
non_recursive_dfs_aux (Stack stack)
{
  while (!stack.empty ())
    {
      current = stack.pop ();
      if (is_target (current))
        dump_family_line_and_exit (current);
      forall successors next of current
        {
          if (cached (next)) continue;
          cache (next);
          stack.push (next);
          next.set_parent (current);
        }
    }
}
```

Vorbereitung Nicht-Rekursive DFS

```
non_recursive_dfs ()
{
  Stack stack;

  forall initial states state
    stack.push (state);

  non_recursive_dfs_aux (stack);

  /* target not reachable */
}
```

DFS – 1. Fehlerhafte Version

```
non_recursive_buggy_dfs_aux (Stack stack)
{
  while (!stack.empty ())
    {
      current = stack.pop ();
      if (is_target (current))
        dump_family_line_and_exit (current);
      if (cached (current)) continue;
      cache (current);
      forall successors next of current
        {
          stack.push (next);
          next.set_parent (current);
        }
    }
}
```

DFS – 2. Fehlerhafte Version

```
non_recursive_but_also_buggy_aux (Stack stack)
{
  while (!stack.empty ())
    {
      current = stack.pop ();
      forall successors next of current
        {
          if (cached (next)) continue;
          cache (next);
          stack.push (next);
          next.set_parent (current);
          if (is_target (next))
            dump_family_line_and_exit (next);
        }
    }
}
```

Breitensuche – Breadth First Search (BFS)

```
bfs_aux (Queue queue)
{
  while (!queue.empty ())
    {
      current = queue.dequeue ();
      if (is_target (current))
        dump_family_line_and_exit (current);
      forall successors next of current
        {
          if (cached (next)) continue;
          cache (next);
          queue.enqueue (next);
          next.set_parent (current);
        }
    }
}
```

Vorbereitung Nicht-Rekursive BFS

```
bfs ()
{
  Queue queue;

  forall initial states state {
    cache (state);
    queue.enqueue (state);
  }

  bfs_aux (queue);

  /* target not reachable */
}
```

DFS versus BFS

- Tiefensuche
  - einfach rekursiv zu implementieren

    (was man aber sowieso nicht sollte)
  - erlaubt Erweiterung auf Zyklenerkennung $\Rightarrow$ Liveness
- Breitensuche
  - erfordert nicht-rekursive Formulierung
  - kürzeste Gegenbeispiele
    - findet immer kürzesten Pfad zum Ziel

Vorwärts- versus Rückwärtstraversierung

- Vorwärtstraversierung bzw. Vorwärtsanalyse
  - Nachfolger sind die als nächstes zu betrachtenden Zustände
  - analysierte Zustände sind alle erreichbar
- Rückwärtstraversierung bzw. Rückwärtsanalyse
  - Vorgänger sind die als nächstes zu betrachtenden Zustände
  - manche untersuchten Zustände können unerreichbar sein
  - meist nur in Kombination mit symbolischen Repräsentationen von Zuständen
  - symbolische Darstellungen bei Rückwärtsanalyse meist “zufällig” und komplex
  - **manchmal terminiert Rückwärtsanalyse in wenigen Schritten**

Globale versus Lokale/On-The-Fly Analyse

- Globale Analyse
  - anwendbar auf Rückwärts- oder Vorwärtstraversierung
  - arbeitet mit *allen* globalen Zuständen (inklusive nicht erreichbaren)
  - Kombination mit expliziter Analyse skaliert sehr schlecht
- Lokale oder On-The-Fly Analyse
  - arbeitet nur auf den erreichbaren Zuständen
  - generiert und analysiert diese on-the-fly ausgehend von Anfangszuständen
- gemischte Vorgehensweise möglich:
  - bestimme vorwärts erreichbare Zustände, darauf dann globale Analysen

Beispiel zu Global versus Lokal

```
process A {
  int count;
  run() {
    while (true)
      if (b.count != this.count)
        count = (count + 1) % (1 << 20);
  }
}

process B {
  int count;
  run() {
    while (true)
      if (a.count == this.count)
        count = (count + 1) % (1 << 20);
  }
}

A a;
B b;
```

Speicherverbrauch

- bei globaler Analyse wird jedem Zustand ein *Markierungsbit* hinzugefügt

```
bool cached (State state) { return state.mark; }

void cache (State state) { state.mark = true; }
```

- bei globaler Analyse **ohne** On-the-fly Zustandsgenerierung
  - $2^{64} = 2^{32} \cdot 2^{32}$ **mögliche** Zustände können nicht vorab alloziert werden
- bei lokaler Analyse **mit** On-the-fly Zustandsgenerierung
  - $2 \cdot 2^{20} = 2097152$ **erreichbare** Zustände
  - ergeben 8-fache Menge an Bytes $\approx$ 16 MB

    (plus Overhead diese zu Speichern, $<$ Faktor 2)

Zustandsspeicherung bei On-The-Fly Expliziter Analyse

- Anforderungen an Datenstruktur zur Speicherung von Zuständen:
  - besuchte Zustände müssen markiert/gespeichert werden (`cache`)

    $\Rightarrow$ pro neuem Zustand eine Einfügeoperation
  - neue Nachfolger werden auf "schon besucht" getestest (`cached`)

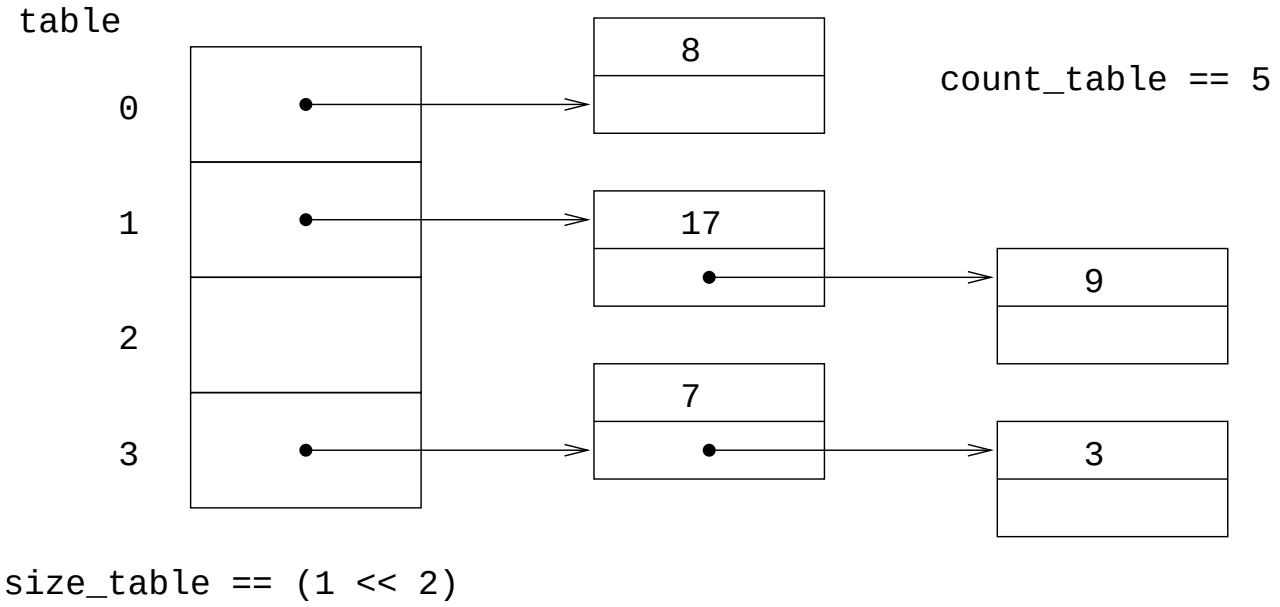
    $\Rightarrow$ pro Nachfolger eine Suche auf Enhaltensein

- Alternative Implementierungen:
  - Bit-Set: pro *möglichem* Zustand ein Bit (im Prinzip wie bei globaler Analyse)
  - Suchbäume: Suche/Einfügen logarithmisch in Zustandsanzahl
  - **Hashtabelle**: Suche/Einfügen konstant in Zustandsanzahl

Hashtabellen zur Zustandsspeicherung

- Laufzeit kann als konstant angesehen werden
  - in der Theorie weit komplexere Analyse notwendig
- “gute” Hashfunktion ist das Wesentliche
  - Randomisierung bzw. Verteilung der Eingabebits auf Hashindex
- Anpassung der Grösse der Hashtabelle:
  - entweder durch dynamische (exponentielle) Vergrösserung
  - oder Maximierung und Anpassung an den verfügbaren Speicher

Hashtabelle mit Collision-Chains




```
unsigned hash (unsigned data) { return data; }

struct Bucket *
find (unsigned data)
{
  unsigned h = hash (data);
  h &= (size_table - 1);
  ...
}
```

Schlechte Hashfunktion für Strings in Compilern

```
unsigned
bad_string_hash (const char * str)
{
  const char * p;
  unsigned res;

  res = 0;

  for (p = str; *p; p++)
    res += *p;

  return res;
}
```

Schlechte Hashfunktion für Strings in Compilern

```
unsigned
very_bad_string_hash (const char * str)
{
  const char * p;
  unsigned res;

  res = 0;

  for (p = str; *p; p++)
    res = (res << 4) + *p;

  return res;
}
```

Klassische Hashfunktion für Strings in Compilern

[Dragonbook]

```
unsigned
classic_string_hash (const char *str)
{
  unsigned res, tmp;
  const char *p;

  res = 0;

  for (p = str; *p; p++)
    {
      tmp = res & 0xf0000000;   /* unsigned 32-bit */
      res <<= 4;
      res += *p;
      if (tmp)
        res ^= tmp >> 28;
    }

  return res;
}
```

Analyse der Klassischen Hashfunktion für Strings

- empirisch sehr gute Randomisierung bei Bezeichnern von Programmiersprachen
  - relative Anzahl Kollisionen als Maßzahl für die Qualität
- **schnell**: maximal 4 logisch/arithmetische Operationen pro Zeichen
- bei längeren Strings von 8 Zeichen und mehr gute Bit-Verteilung
- Überlagerung der 8-Bit Kodierungen der einzelnen Zeichen

  (man beachte aber die ASCII Kodierung)
- Klustering bei vielen kurzen Strings (z.B. in automatisch generiertem Code)

  `n1, ..., n99, n100, ..., n1000`

Bessere Hashfunktion für Strings

```
static unsigned primes [] = { 2000000011, 2000000033, ... };
#define NUM_PRIMES  (sizeof (primes) / sizeof (primes[0]))

unsigned
primes_string_hash (const char * str)
{
  unsigned res, i;
  const char * p;

  i = 0;
  res = 0;
  for (p = str; *p; p++)
    {
      res += *p * primes[i++];
      if (i >= NUM_PRIMES)
        i = 0;
    }

  return res;
}
```

Gute Hashfunktion für Zustände

```
static unsigned primes [] = { 2000000011, 2000000033, ... };
#define NUM_PRIMES (sizeof(primes)/sizeof(primes[0]))

unsigned
hash_state (unsigned * state, unsigned words_per_state)
{
  unsigned res, i, j;

  res = 0;
  i = 0;
  for (j = 0; j < words_per_state; j++)
    {
      res += state[j] * primes [i++];
      if (i >= NUM_PRIMES)
        i = 0;
    }

  return res;
}
```

CRC basierte Hashfunktion für Strings

```
#define CRC_POLYNOMIAL 0x82F63B78
static unsigned
crc_hash_bit_by_bit (const char * str)
{
  unsigned const char * p;
  unsigned res = 0;
  int i, bit;
  for (p = str; *p; p++) {
      res ^= *p;
      for (i = 0; i < 8; i++) {
          bit = res & 1;
          res >>= 1;
          if (bit)
            res ^= CRC_POLYNOMIAL;
        }
    }
  return res;
}
```

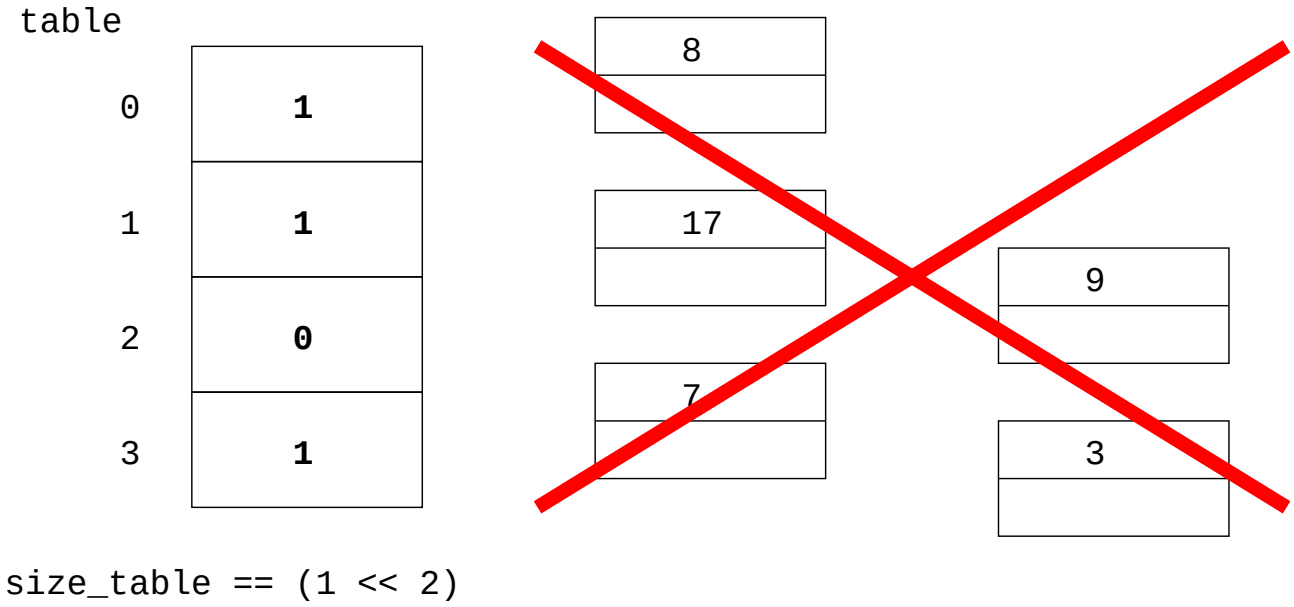
Analyse der Hashfunktion für Zustände

- lässt sich leicht parametrisieren
  - durch verschiedene Primzahlen oder Anfangspositionen
- heutzutage ist Integer-Multiplikation sehr schnell
  - durch mehr Platz für die Multiplizierer auf der ALU
  - und durch Vorhandensein mehrerer Integer-Functional Units
- was noch fehlt ist Anpassung an Tabellengröße
  - bei Zweierpotenzgröße: einfache Maskierung
  - sonst Primzahlgröße und Anpassung durch Modulo-Bildung

Super-Trace Algorithmus

- Probleme mit expliziter **vollständiger** Zustandsraum-Exploration:
  1. wegen Zustandsexplosion oft zu viele Zustände
  2. ein einziger Zustand braucht oft schon viel Platz (dutzende Bytes)
- Ideen des Super-Trace Algorithmus:
  1. behandle Zustände mit demselben Hash-Wert als identisch
  2. speichere nicht die Zustände, sondern Hash-Werte erreichter Zustände
- Super-Trace wird auch **Bit-State-Hashing** genannt

Super-Trace Hashtabelle als Bit-Set



size_table == (1 << 2)

Falls Zustand mit gleichem Hash-Wert früher schon besucht wurde, dann gilt momentaner Zustand als besucht!

Analyse des Super-Trace Algorithmus

- Vorteile:
  - drastische Reduktion des Speicherplatzes auf ein Bit pro Zustand
  - Reduktion um mindestens 8 mal Größe des Zustandes in Bytes
  - Anpassung der Größe der Hash-Tabelle/Bit-Set an vorhandenen Speicher
  - Ausgabe einer unteren Schranke der Anzahl besuchten Zustände
  - Parametrisierung der Hash-Funktion erlaubt unterschiedliche Exploration
- Nachteile:
  - bei nicht-kollisionsfreier Hash-Funktion (der Normalfall) Verlust der Vollständigkeit
  - nur vage Aussage über Abdeckung möglich

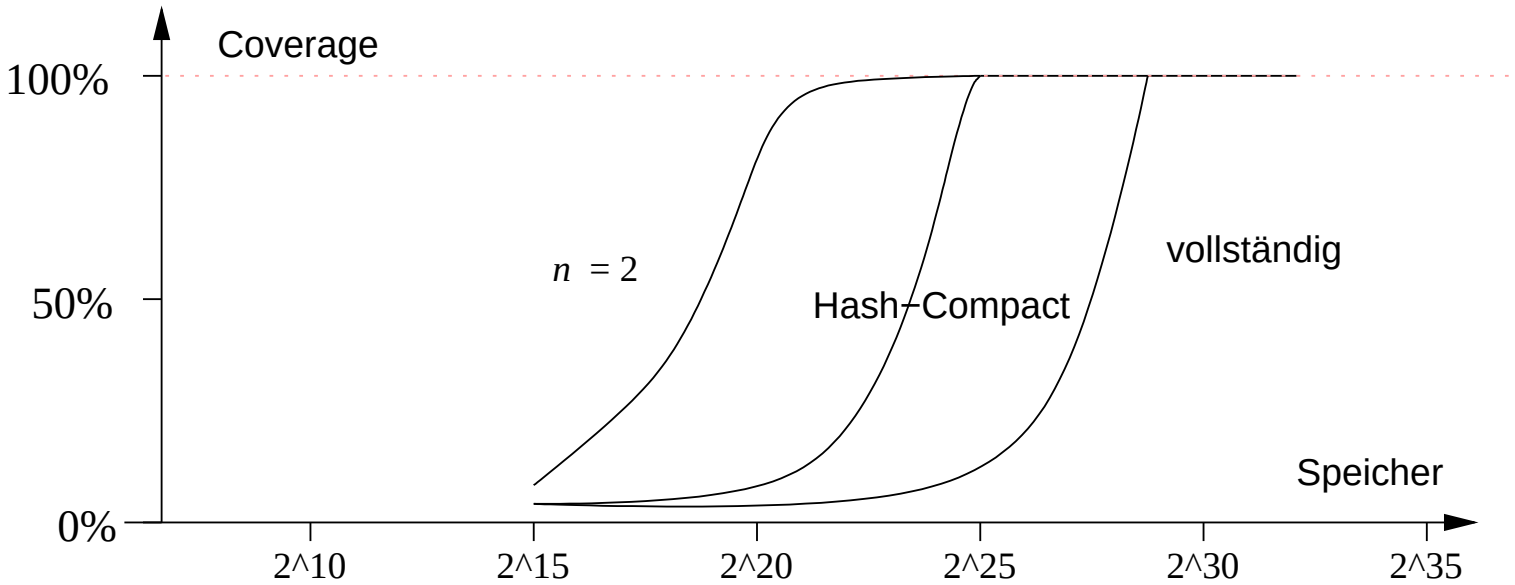
Zwei Hashfunktionen beim Supertrace

- berechne zwei Hashwerte mit unterschiedlichen Hashfunktionen
  - Speichern von Zuständen durch Eintrag jeweils eines Bits in zwei Hashtabellen
  - Zustand wird als schon erreicht angenommen gdw. beide Bits gesetzt
  - z.B. $h_1, h_2 : Keys \rightarrow \{0, \ldots, 2^{32} - 1\}$
  - zwei Hashtabellen mit jeweils 2^{32} Bits = 2^{29} Bytes = 512 MB
- lässt sich auch leicht auf n Hashfunktionen ausbauen
  - $n = 4$ mit 2 GB Speicher für Hashtabellen ist durchaus realistisch
  - vergl. Parameteriesierung der Hashfunktion basierend auf Primzahlmultiplikation

Hash-Compact

- statt n Bits zu setzen kann man auch einfach den n-Bit Hash-Wert speichern
  - bei 256 MB = 2^{28} Bytes Platz für die Hashtabelle bis zu 2^{26} Hashwerte/Zustände
    (32-Bit Hashfunktion $h: Keys \rightarrow \{0, \ldots, 2^{32} - 1\}$, 4 Bytes Hashwert pro Zustand)

- beliebige Varianten ergeben ungefähr folgendes Bild:



(schematisch nach [Holzmann 1998], 427567 erreichbare Zustände, 1376 Bits)

Best-Case-Analyse des Speicher-Verbrauchs

- Idealisierte Best-Case Annahmen:
  - verwendete Hash-Funktion ist möglichst kollisionsfrei
  - Hashtabelle wird solange wie möglich ohne Kollisionen gefüllt

- m Speicherplatz in Bits, s Zustandsgrösse in Bits, r erreichbare Zustände

$$coverage_{n=2}(m) = \frac{m}{r \cdot 2}$$

$$coverage_{\text{hashcompact}}(m) = \frac{m}{r \cdot w} \quad w = \text{Wordgröße in Bits, z.B. 32 Bit}, w \leq \lceil \log_2 r \rceil$$

$$coverage_{\text{complete}}(m) = \frac{m}{r \cdot s}$$

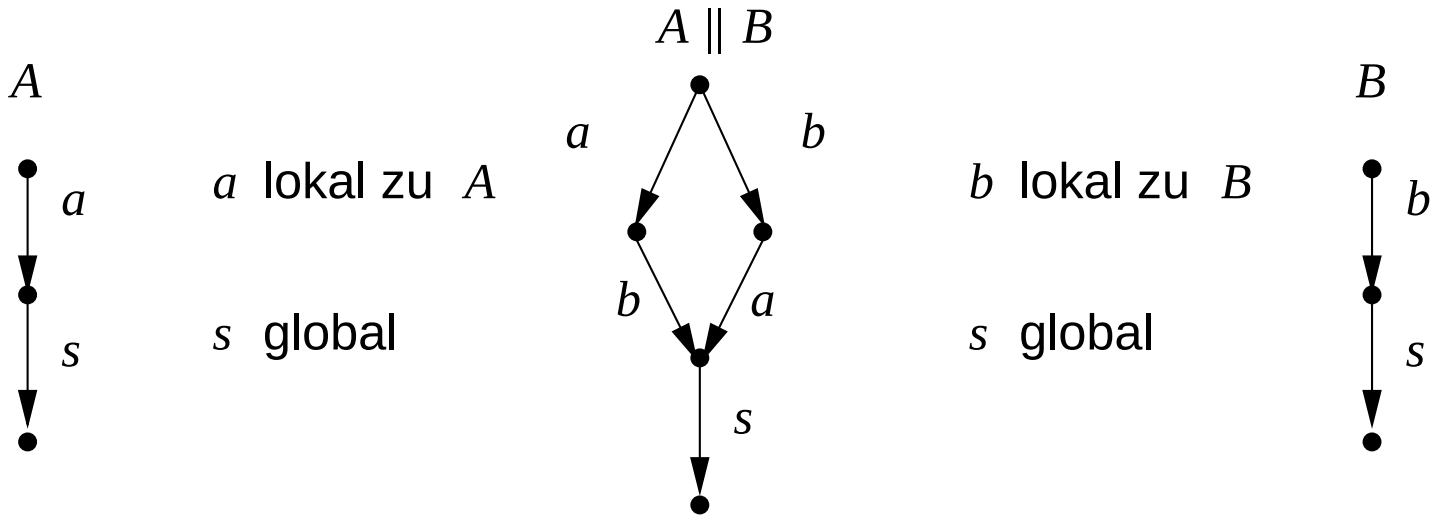
(im vorigen Schaubild ist die x-Achse logarithmisch dargestellt)

- in der Praxis gibt es Kollisionen und die unvollständigen brauchen mehr Speicher

Asynchron versus Synchrone Komposition

- Synchrone Komposition
  - Gleichschritt aller Komponenten zu einem globalen Takt
  - entspricht der Konstruktion des Produkt-Automaten
  - typische Modellierung von Sequentieller **Hardware**
    (obwohl auf Transistor-Ebene eigentlich alles asynchron)
- Asynchrone Komposition
  - Komponenten laufen im Wesentlichen unabhängig nach eigenem Takt
  - typische Modellierung für Kommunikationsprotokolle und Verteilte **Software**
  - Synchronisation: Systemcalls/Interrupts/Signale/Nachrichten/Kanäle/RPC
  - geeignete Vereinfachung: **Interleaving**

Beispiel Interleaving



- jeder Prozess kommt **abwechselnd** dran mit seinen **lokalen** Aktionen
- auf **globale** Aktionen wird per **Rendezvous** synchronisiert
- Standard-Parallel-Komposition in Prozess-Algebra

Definition Zu zwei LTS A_1 und A_2 besteht die parallele Komposition $A = A_1 \,||\, A_2$ aus folgenden Bestandteilen:

$S = S_1 \times S_2$, $\quad \Sigma = \Sigma_1 \cup \Sigma_2$, $\quad I = I_1 \times I_2$, $\quad T$ wird wie folgt definiert:

$$(s,t) \xrightarrow{a} (s',t') \text{ in } A \quad \text{gdw.} \quad \begin{array}{llll} s \xrightarrow{a} s' \text{ in } A_1 & \text{und} & t' = t & \text{falls } a \in \Sigma \backslash \Sigma_2 \\ t \xrightarrow{a} t' \text{ in } A_2 & \text{und} & s' = s & \text{falls } a \in \Sigma \backslash \Sigma_1 \\ s \xrightarrow{a} s' \text{ in } A_1 & \text{und} & t \xrightarrow{a} t' \text{ in } A_2 & \text{falls } a \in \Sigma_1 \cap \Sigma_2 \end{array}$$

Interleaving mit Synchronisation auf gemeinsamen Aktionen

Definition In $A_1 \,||\, A_2$ ist ein Symbol a **lokal** für A_i gdw. $a \in \Sigma_i$ und $a \notin \Sigma_j$ für alle $i \neq j$. Die Menge der lokalen Symbole für A_i wird mit Λ_i bezeichnet.

Definition Ein Symbol heißt **lokal** falls es lokal für ein A_i ist. Allen lokalen Symbole sind in $\Lambda = \bigcup \Lambda_i$ zusammengefasst.

Definition Übergang $(s_1, s_2) \xrightarrow{a} (s_1', s_2')$ in $A_1 \,||\, A_2$ ist lokal (für A_i), falls a lokal (für A_i).

Definition **Globale** Symbole bzw. Übergänge sind solche, die nicht lokal sind. Die Menge der globalen Symbole zu A_i wird mit Γ_i bezeichnet und deren Zusammenfassung als $\Gamma = \bigcup \Gamma_i$.

Falls $i = 1$ so sei $\sigma(i) = 2$ und umgekehrt.

Fakt $(s_1, s_2) \xrightarrow{a} (s_1', s_2')$ in A gdw.

- $s_i \xrightarrow{a} s_i'$ in A_i und $s'_{\sigma(i)} = s_{\sigma(i)}$ falls a lokal für A_i
- $s_j \xrightarrow{a} s_j'$ in A_j für alle j = 1,2 falls a global

Fakt Die asynchrone parallele Komposition $||$ ist assoziativ.

(die Schreibweise $A_1 \,||\, A_2 \,||\, \ldots \,||\, A_n$ ist also wohldefiniert)

Fakt ... und kommutativ modulo Bisimulation: $A_1 \,||\, A_2 \approx A_2 \,||\, A_1$

Fakt Für die Übergangsrelation von $A_1 \,||\, A_2 \,||\, \ldots \,||\, A_n$ gilt:

Sei $\Psi(a) \subseteq \{1,\ldots,n\}$ die Menge der Indizes i mit $a \in \Sigma_i$ und $\overline{\Psi(a)}$ deren Komplement.

$(s_1,\ldots,s_n) \xrightarrow{a} (s'_1,\ldots,s'_n)$ gdw. $s_i \xrightarrow{a} s'_i$ für alle $i \in \Psi(a) \neq \emptyset$ und $s'_j = s_j$ für alle $j \in \overline{\Psi(a)}$.

Definition Zu zwei LTS A_1 und A_2 besteht die *voll asynchrone* parallele Komposition $A = A_1 \,|||\, A_2$ aus folgenden Bestandteilen:

$S = S_1 \times S_2$, $\quad \Sigma = P(\Sigma_1 \cup \Sigma_2)$, $\quad I = I_1 \times I_2$, $\quad T$ wird wie folgt definiert:

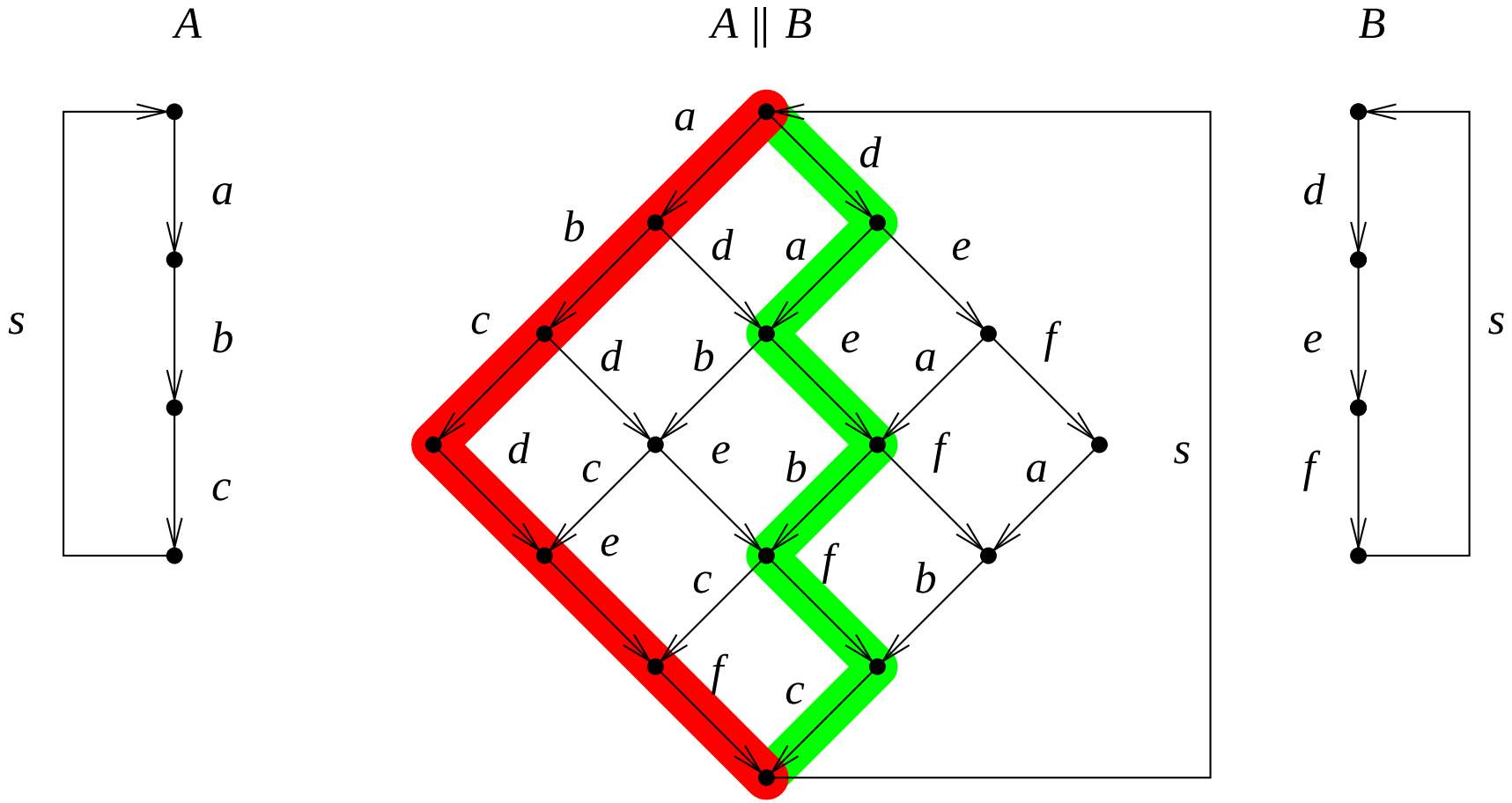
$$(s,t) \xrightarrow{M} (s',t') \text{ in } A \quad \text{gdw.} \quad \begin{cases} s \xrightarrow{a} s' \text{ in } A_1 \text{ und } t' = t & \text{falls } M = \{a\} \subseteq \Sigma_1 \setminus \Sigma_2 \\ t \xrightarrow{b} t' \text{ in } A_2 \text{ und } s' = s & \text{falls } M = \{b\} \subseteq \Sigma_2 \setminus \Sigma_1 \\ s \xrightarrow{a} s' \text{ in } A_1 \text{ und } t \xrightarrow{a} t' \text{ in } A_2 & \text{falls } M = \{a\} \subseteq \Sigma_1 \cap \Sigma_2 \\ s \xrightarrow{a} s' \text{ in } A_1 \text{ und } t \xrightarrow{b} t' \text{ in } A_2 & \text{falls } a \in \Sigma_1 \setminus \Sigma_2 \\ & \text{und } b \in \Sigma_2 \setminus \Sigma_1 \\ & \text{und } M = \{a,b\} \end{cases}$$

Vergleich Interleaving mit Voller Asynchroner Komposition

- Erweiterung der Vollen Asynchronen Komposition auf n LTS:
  - $\Sigma = P(\Sigma_1 \cup \cdots \cup \Sigma_n), \quad T = \ldots$
    (Möglichkeit der gleichzeitigen Synchronisation auf mehrere globale Symbole)
  - insgesamt also exponentielles Aufblasen des Alphabetes!

- Interleaving ist eine Vereinfachung
  - **Fakt** gleiche Menge erreichbarer Zustände
  - Länge von Pfaden zwischen zwei Zuständen kann sich unterscheiden, aber ...
  - ... Modellierung nicht exakt bez. relativer Geschwindigkeit von Komponenten

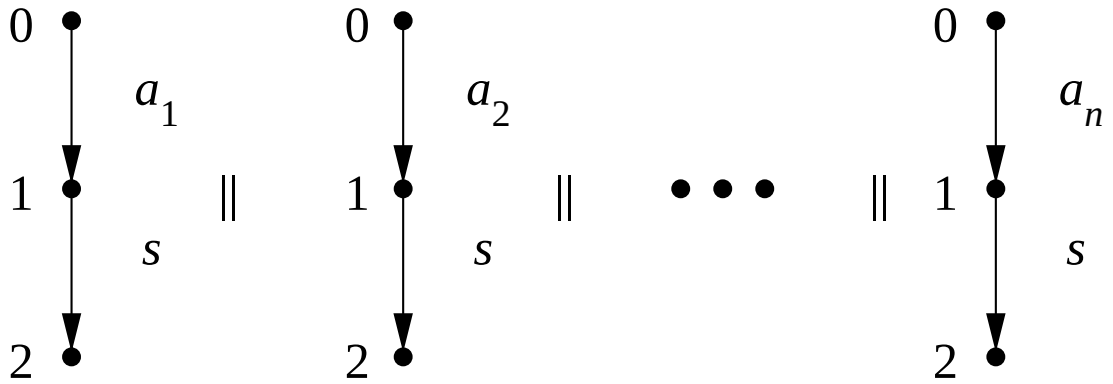
Zustandsexplosion durch Interleaving



Idee Verfolge nur einen der 8 möglichen Pfade, z.B. den roten oder den grünen

Exponentielle Reduktion

$$\Sigma_i = \{a_i, s\}, \quad \Sigma = \{a_1, \ldots, a_n, s\}, \quad \Lambda_i = \{a_i\}, \quad \Gamma = \Gamma_i = \{s\}$$

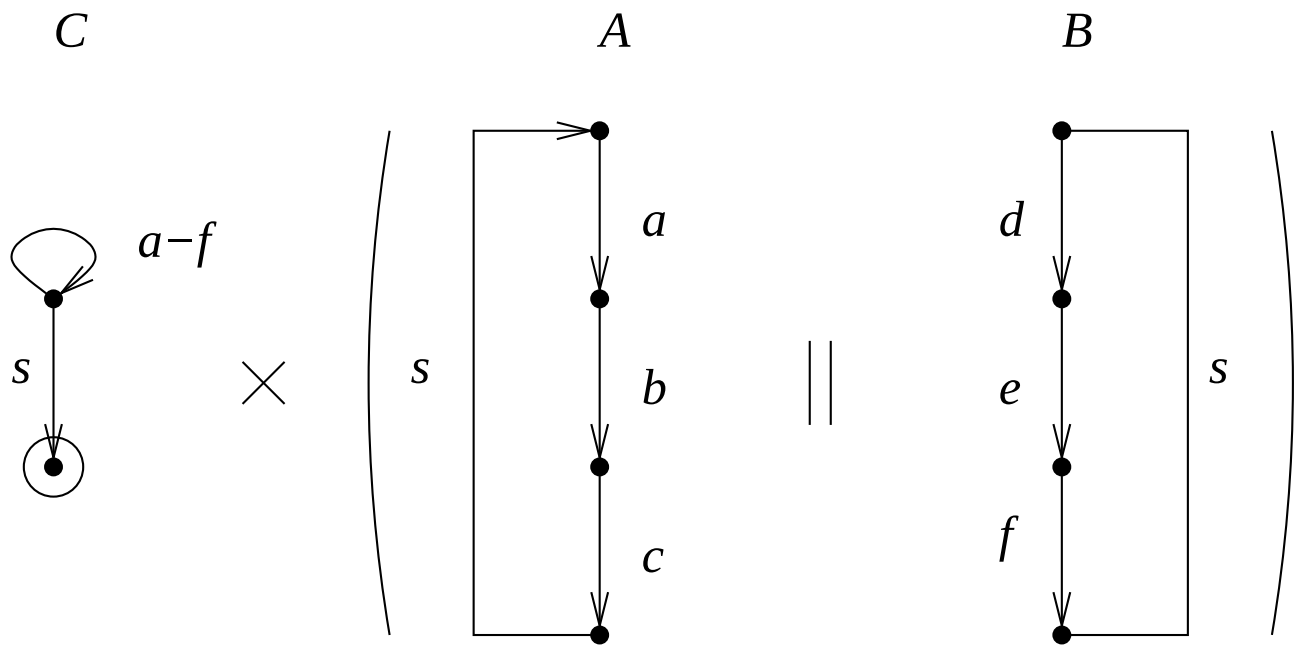


Anzahl Zustände: $|S| = |\{0,1,2\}^n| = 3^n.$

Anzahl erreichbarer Zustände: $|\{0,1\}^n \cup \{2\}^n| = 2^n + 1$

Anzahl notwendiger Zustände: $|(1^*0^* \cap \{0,1\}^n) \cup \{2\}^n| = (n+1)+1$

Synchrone Composition mit Checker-Automaten



$$C \times (A \,||\, B)$$

Einschränkungen

- Checker muss “invariant” gegenüber weggelassenen Übergängen sein
  - Auslassen von Übergängen darf Erreichen von Finalzuständen nicht verhindern
  - Reduktion ist abhängig vom Checker!
- Maximal viele Übergänge wegzulassen geht nicht:
  - nur möglich wenn man schon die Erreichbarkeit von Finalzuständen kennt
  - damit wäre kein Gewinn möglich!
- Ziel ist möglichst einfaches Kriterium zum Weglassen von Übergängen
  - welches im besten Fall statisch überprüft werden kann
  - oder effizient dynamisch während der Suche

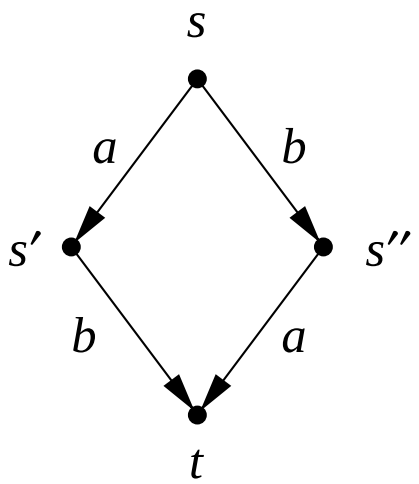
Lokaler Zustand

Definition Ein Zustand $s = (s_1, \ldots, s_n)$ in $A_1 \,||\, \cdots \,||\, A_n$ heißt **lokal** zu A_i gdw.

alle Übergänge in A_i ausgehend von s_i lokal für A_i sind und ein solcher existiert.

Definition Ein Symbol a kommutiert mit dem Symbol b im Zustand s gdw.

es für alle s', s'' mit $s \xrightarrow{a} s'$ und $s \xrightarrow{b} s''$ einen Zustand t gibt, mit $s' \xrightarrow{b} t$ und $s'' \xrightarrow{a} t$.



Fakt Sei s lokal zu A_i, dann kommutieren alle Λ_i mit allen andere $\Sigma \backslash \Lambda_i$.

Lokale Partielle Ordnungs-Reduktion

Partial Order Reduction = Partielle Ordnungs-Reduktion

Definition Eine *Expansion* eines Zustandes ist eine Teil-Menge seiner Nachfolger.

BFS/DFS iteriert nur noch über die Expansion von `current` in der inneren Schleife

```
partial_order_recursive_dfs_aux (Stack stack, State current)
{
  ...

  forall next in expansion (current)
    partial_order_recursive_dfs_aux (stack, next);

  ...
}
```

Definition Eine *partielle* Expansion ist eine **echte** Teil-Menge der Nachfolger.

Expansion in der Lokalen Partiellen Ordnungs-Reduktion

Definition Die *Lokale Partielle Ordnungs-Reduktion* wählt als möglicherweise partielle Expansion eines Zustandes s eine Teil-Menge seiner Nachfolger wie folgt aus:

- genau die lokalen Übergänge von A_i, falls s lokal zu A_i ist

  (in diesem Fall hat man eine partielle Expansion)

- bei mehreren solchen A_i wird ein beliebiges i gewählt

- falls kein solches A_i existiert, besteht die Expansion aus allen Nachfolgern

Lokal Äquivalente Traces

auch “stutter equivalent”

Definition zwei Traces w und w' sind **lokal äquivalent**, geschrieben $w \approx_l w'$, gdw. wenn sie nach Herausstreichen aller lokalen Symbole identisch sind.

Fakt die lokale Äquivalenz $\approx_l$ ist tatsächlich eine Äquivalenzrelation

Beweis

- Reflexivität, Symmetrie folgen unmittelbar
- Schreibweise $w|_{\Sigma'}$ für den Trace w eingeschränkt auf Symbole aus Σ'
- Transitivität: aus $w_1 \approx_l w_2$, $w_2 \approx_l w_3$ folgt $w_1|_\Gamma = w_2|_\Gamma = w_3|_\Gamma$ und somit $w_1 \approx_l w_3$

Unsichtbare Symbole

Definition Checker C *ignoriert lokale Symbole* gdw.

$s \xrightarrow{a} = \{s\}$ in C für alle lokalen $a \in \Lambda$

(auch "a ist unsichtbar für C")

Fakt C ignoriere lokale Symbole und $w \approx_l w'$, dann $w \in L(C) \Leftrightarrow w' \in L(C)$

Beweis Sei $w \in L(C)$. Zustandssequenz, die w akzeptiert, akzeptiert auch w' und umgekehrt.

Suche mit Partieller Ordnungs-Reduktion muss also nur für jede Äquivalenzklasse von $\approx_l$ mindestens einen Repräsentanten ablaufen.

Volle Expansion bei Zyklen

Problem

- eine partielle Expansion “verzögert” Ausführung Übergänge anderer A_j
- diese könnten für immer verzögert werden

Lösung

- Zyklus mit nur partiellen Expansionen muss an einer Stelle voll expandiert werden
- lässt sich leicht in Tiefensuche (DFS) einbauen:
  - jeder Zyklus wird durch “Backedge” zu einem Knoten auf Suchstack geschlossen
  - bei einer “Backedge” wird `current` einfach voll expandiert
- Approximation bei BFS: volle Expansion bei Kanten zu Knoten früherer Generation

Partielle Ordnung bei Allgemeineren Modellen

- statt Synchronisation durch Aktionen:
  - Synchronisation durch gemeinsame globale Variable
  - und/oder Synchronisation durch Monitore/Semaphoren
  - und/oder Synchronisation durch Nachrichten/Kanäle
- Partielle Ordnungs-Reduktion lässt sich mit folgenden Konzepten anwenden:
  - Unabhängigkeit bzw. Kommutieren von Befehlen
    - Read/Write und Write/Read abängig, Read/Read unabhängig
    - analog für Nachrichten (Read = Receive, Write = Send)
  - Unsichtbarkeit von lokalen Befehlen

Partielle Ordnung bei Message Passing Modellen

- Message Passing
  - Kommunikation durch Nachrichtenaustausch über Kanäle/Puffer/FIFO
  - beachte: endliche Modelle haben immer Kanäle endlicher Kapazität
- Unabhängige oder lokale Operationen (2 Prozesse, 1 Kanal):
  - Lesen von einem **nicht vollem** Kanal
  - Schreiben in einen **nicht leeren** Kanal
- Abhängige oder globale Operationen (2 Prozesse, 1 Kanal):
  - Lesen von einem **vollem** Kanal
  - Schreiben in einen **leeren** Kanal

Liveness und Fairness

- Liveness – Lebendigkeit
  - im Gegensatz zu Safety- bzw. Sicherheitseigenschaften
  - beschreibt **unausweichbares** Verhalten
  - macht wiederum nur Sinn bei Abstraktion vom konkretem Zeitverhalten:

    Konzentration auf mögliche Ereigisfolgen 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”

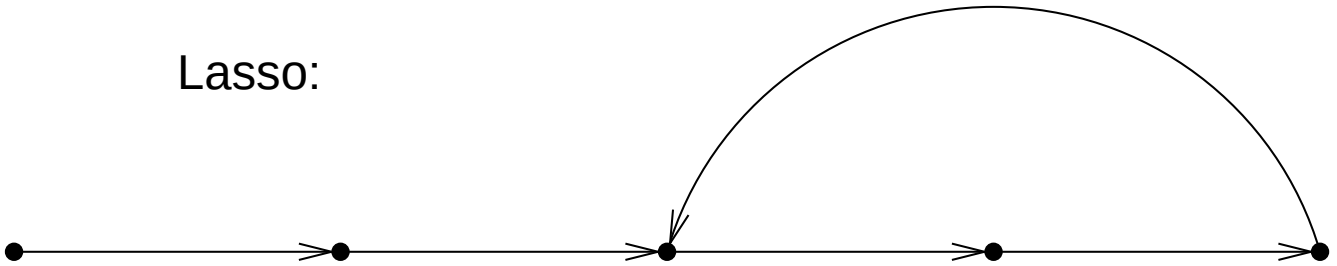
Weitere Beispiele zu Liveness

- 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

Gegenbeispiele zu Lebendigkeitseigenschaften

- Endliche Systeme:

Lasso:



Kein Fortschritt

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

Fairness

- Abstraktion vom konkreten “Scheduler”:
  - Reihenfolge der Ausführung von Prozessen ist im Modell beliebig
  - z.B. bei Interleaving oder voll asynchroner Parallelkomposition
- dadurch Artifakte 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

    (letzeres oftmals nicht möglich, da relative Geschwindigkeit nicht bekannt)

Definition ein *faires 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.

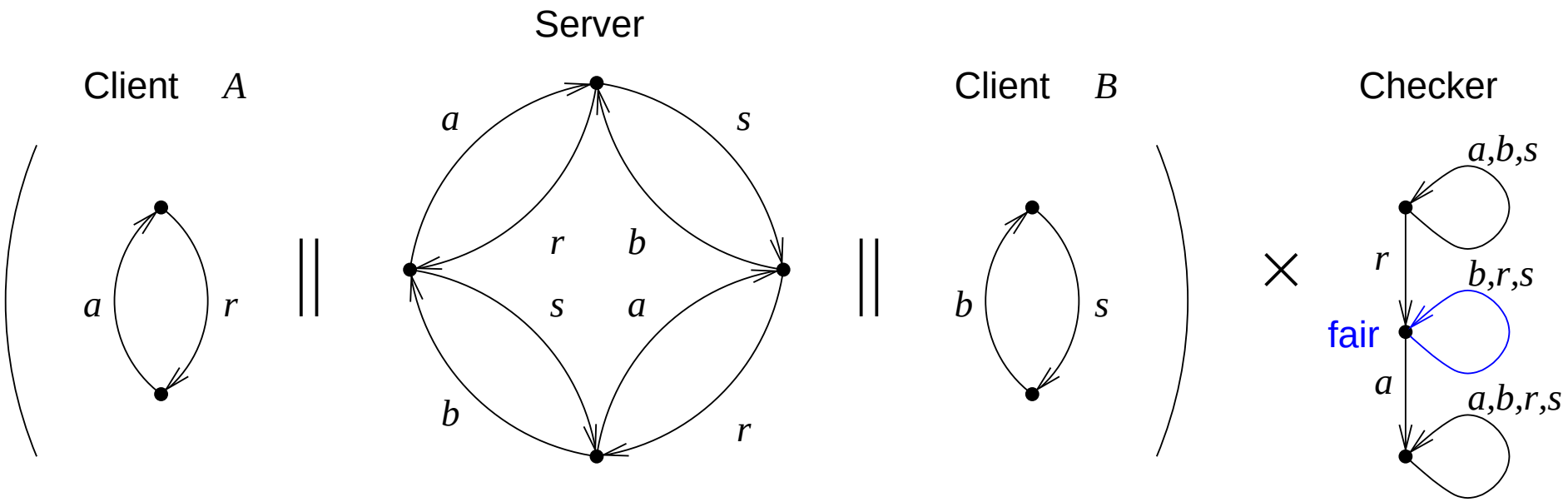
(ein Übergang ist einfach eine Trippel (s, a, s'))

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

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

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 ($s \not\xrightarrow{a}$ für alle $a \in \Sigma_j$). Ein dadurch gegebener fairer Pfad in $A_1 \mid\mid \cdots \mid\mid A_n$ muss A_j immer wieder "ausführen".

Fairness im Checker



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 $r(bs)^{\omega} = rbsbsbs\ldots$

(bei dem A genau einmal ausgeführt wird)

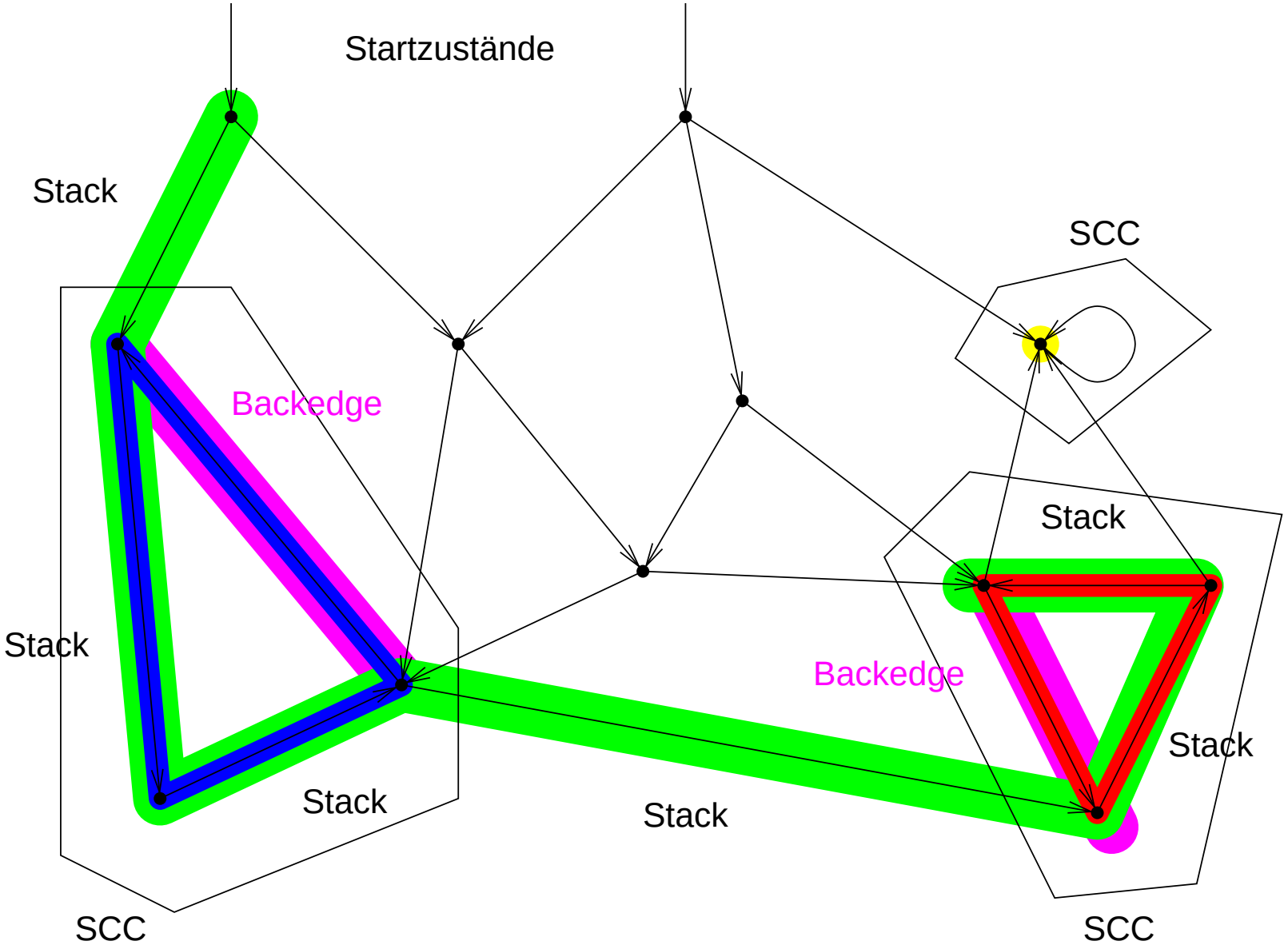
Suche nach Fairen Pfaden

- 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 (= Zustandraum),

    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

SCC


Startzustände
Stack
SCC
Backedge
Stack
Backedge
Stack
Stack
Stack
Stack
SCC
SCC

SCC Zerlegung nach Tarjan

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

Verallgemeinerte Fairness

Problem im Beispiel ist der Scheduler nicht *fair*

Definition ein *verallgemeinertes faires LTS* $A = (S, I, \Sigma, T, F_1, \ldots, F_n)$ 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 $\pi = s_0 \stackrel{a_0}{\rightarrow} s_1 \rightarrow \ldots$ 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.

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. $F_1 < \ldots < F_n$
  - 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

Repräsentation von Kombinatorischer Logik

- Abstrakter Datentyp Boolesche Logik:

  Konstruktoren: Erzeugung von boolesche Konstanten und Variablen

  Operationen: Konjunktion, Disjunktion, Negation, ...

  Abfragen: Test auf Erfüllbarkeit, Tautologie, ...

- Basis-Datentyp in EDA-Werkzeugen:

  Simulatoren, Optimierer, Compiler

- Trade-Off zwischen effizienten Operationen und Platzverbrauch

Implementierung als Funktionstabelle

0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

f

0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

g

0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$f \vee g$

0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

$\neg f$

Operationen einfach Punktweise ausführen

Ad: Funktionstabelle

- Funktionstabelle ist immer 2^n Zeilen gross bei n Variablen
- Operationen sind alle linear in der Grösse der Argumente:

  z.B. Konjunktion ergibt Funktionstabelle gleicher Grösse
- Darstellung ist kanonisch:

  zwei äquivalente boolesche Formeln haben dieselbe Funktionstabelle
- Abfragen sind auch linear:

  Tautologie: überprüfe Zeilen auf Gleichheit

  Erfüllbarkeit: suche 1-Zeile

DNF

- potentiell kompakter als Funktionstabelle

  nur Anzahl Kernprimimplikanten bestimmt Grösse

- unmittelbare Implementierung als minimale 2-stufige Schaltung (PLA)

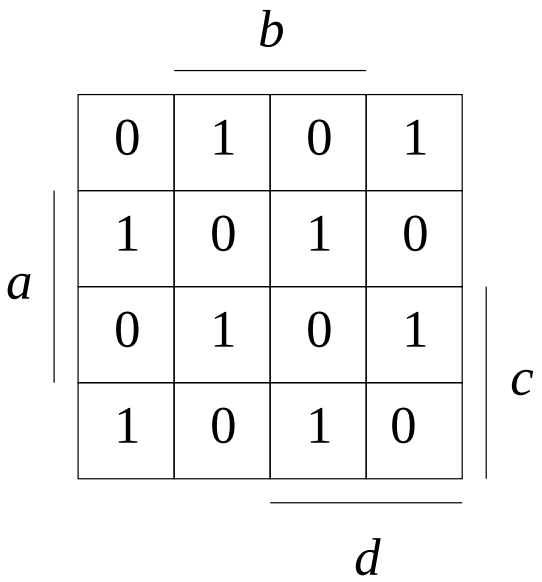
- Disjunktion linear (ohne Minimierung)

- Konjunktion linear und Negation exponentiell (auch ohne Minimierung)

- Erfüllbarkeit mit konstantem Aufwand:

  DNF erfüllbar gdw. mind. ein Monom vorhanden

Exponentielle DNF für Parity



$$a \oplus b \oplus c \oplus d$$

- keine Zusammenfassung von Min-Termen im KV-Diagramm möglich
- nur *volle* Min-Terme als Primimplikanten (bestehend aus maximaler Anzahl von Literalen)
- DNF für Parity von n Variablen hat 2^{n-1} Monome

Konjunktion von DNF

$$\underbrace{(a \cdot \overline{b} \quad \vee \quad \overline{a} \cdot b \cdot \overline{c})}_{\text{1. Operand}} \quad \wedge \quad \underbrace{(a \cdot \overline{b} \quad \vee \quad \overline{b} \cdot \overline{c})}_{\text{2. Operand}}$$

Ausmultiplizieren

$$a \cdot \overline{b} \cdot a \cdot \overline{b} \quad \vee \quad a \cdot \overline{b} \cdot \overline{b} \cdot \overline{c} \quad \vee \quad \overline{a} \cdot b \cdot \overline{c} \cdot a \cdot \overline{b} \quad \vee \quad \overline{a} \cdot b \cdot \overline{c} \cdot \overline{b} \cdot \overline{c}$$

Vereinfachen der einzelnen Min-Terme

$$a \cdot \overline{b} \quad \vee \quad a \cdot \overline{b} \cdot \overline{c}$$

Minimierung (z.B. mit Quine-McCluskey)

$$a \cdot \overline{b}$$

Konjunktion von DNF

$$\underbrace{(a \vee b \vee c)}_{\text{1. Operand}} \wedge \underbrace{(d \vee e \vee f)}_{\text{2. Operand}}$$

Ausmultiplizieren

$$a \cdot d \vee a \cdot e \vee a \cdot f \vee b \cdot d \vee b \cdot e \vee b \cdot f \vee c \cdot d \vee c \cdot e \vee c \cdot f$$

Keine weitere Vereinfachung möglich!

Beispiel lässt sich leicht generalisieren:

Konjunktion zweier DNF mit n und mit m Monomen hat $O(n \cdot m)$ Monome

Ad: Konjunktion von DNF

- maximal linear grösseres Ergebnis (in beiden Argumenten)
- man verliert Minimalität:

  Ausmultiplizieren min. DNF ergibt nicht notwendigerweise min. DNF
- anschliessende Minimierung hätte wiederum exponentiellen Aufwand
- auch implizite Generierung der Monome der Konjunktion möglich

Syntaktische Darstellungen

- z.B. Netzliste der kombinatorischen Schaltung:

  Hypergraph aus Gattern (Knoten) und Signalen (Hyper-Edges)

  (Hyper-Edge: Menge von mit der Kante verbundenen Knoten)

- z.B. Parse-Tree einer booleschen Formel

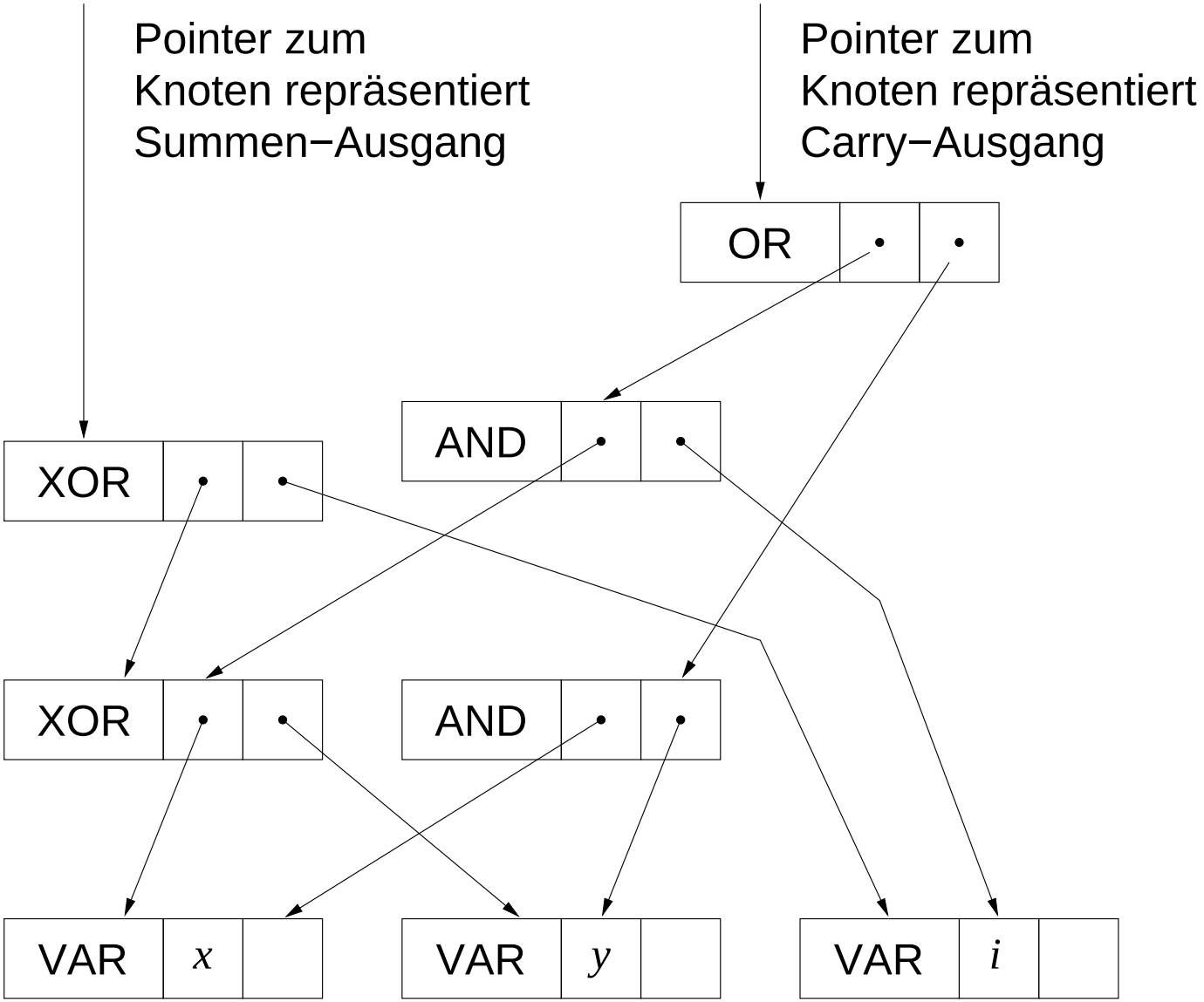
- Sharen gemeinsamer Teil-Formeln ist kompakter:

  Carry-Out eines Ripple-Adders als Baum ist exponentiell gross

  Carry-Out eines Ripple-Adders als Netzliste ist nur linear gross

- z.B. Parse-DAG (Directed Acyclic Graph) für kombinatorische Logik

Parse-DAG eines Voll-Addierers



Knoten-Darstellung in C

```
enum Tag
{
  OR, AND, XOR, NOT, VAR, CONST
};

typedef struct Node Node;
typedef union NodeData NodeData;

union NodeData
{
  Node *child[2];
  int idx;
};

struct Node
{
  enum Tag tag;
  NodeData data;
  int mark;                    /* traversal */
};
```

Ad: Knoten-Darstellung in C

- Ähnliche Darstellung wie von Symbolen in einem Compiler
- Variablen werden durch Integer-Indizes dargestellt
- Boolesche Konstanten werden durch den 0 bzw. 1 Index dargestellt
- Operations-Knoten haben Pointer auf Kinder
- Knoten können mehrfach referenziert werden

  (Speicherverwaltung: Referenz-Counting oder Garbage Collection)
- Keine Zyklen (DAG)!

Erzeugung von Konstanten

```
Node *
new_node_val (int constant_value)
{
  Node *res;

  res = (Node *) malloc (sizeof (Node));
  memset (res, 0, sizeof (Node));
  res->tag = CONST;
  res->data.idx = constant_value;

  return res;
}
```

üblicherweise nur 0 und 1 als Werte für `constant_value`

Erzeugung von Variablen

```
Node *
new_node_var (int variable_index)
{
  Node *res;

  res = (Node *) malloc (sizeof (Node));
  memset (res, 0, sizeof (Node));
  res->tag = VAR;
  res->data.idx = variable_index;

  return res;
}
```

Variable werden an Hand ihres Index unterschieden

Erzeugung von Operations-Knoten

```
Node *
new_node_op (enum Tag tag, Node * child0, Node * child1)
{
  Node *res;

  res = (Node *) malloc (sizeof (Node));
  memset (res, 0, sizeof (Node));
  res->tag = tag;
  res->data.child[0] = child0;
  res->data.child[1] = child1;

  return res;
}
```

Operations-Typ wird als erstes Argument mitübergeben

(Annahme: `child1` ist `0` für Negation)

Erzeugung DAG für Voll-Addierer

```
Node *x, *y, *i, *o, *s, *t[3];

x = new_node_var (0);
y = new_node_var (1);
i = new_node_var (2);
t[0] = new_node_op (XOR, x, y);
t[1] = new_node_op (AND, x, y);
t[2] = new_node_op (AND, t[0], i);
s = new_node_op (XOR, t[0], i);
o = new_node_op (OR, t[1], t[2]);
```

Explizites Sharen durch temporäre Pointer `t[0]`, `t[1]` und `t[2]`

Beispiel-Algorithmus: Input-Cone Variablen

```
void
input_cone_node (Node * node)
{
  if (node->mark)
    return;
  node->mark = 1;
  switch (node->tag)
    {
    case CONST:
      break;
    case VAR:
      printf ("variable %d in input cone\n", node->data.idx);
      break;
    case NOT:
      input_cone_node (node->data.child[0]);
      break;
    default:                    /* assume binary operator */
      input_cone_node (node->data.child[0]);
      input_cone_node (node->data.child[1]);
      break;
    }
}
```

Wiederherstellen des Mark-Flag

```
void
mark_node (Node * node, int new_value)
{
  if (node->mark == new_value)
    return;

  node->mark = new_value;
  switch (node->tag)
    {
    case VAR:
    case CONST:
      return;
    case NOT:
      mark_node (node->data.child[0], new_value);
      break;
    default:
      mark_node (node->data.child[0], new_value);
      mark_node (node->data.child[1], new_value);
      break;
    }
}
```

Ad: Parse-DAG

- Algorithmen basieren auf Depth-First-Search
- Vermeidung von Mehrfach-Traversierung mit `mark`-Flag
- Meist zwei Phasen: Traversierung, Wiederherstellung `mark`-Flag
- Konjunktion, Negation sind schnell (verwende `op`)
- Tautologie und Erfüllbarkeit schwierig
- Nur explizites Sharen: keine kanonische Darstellung

And-Inverter-Graphs (AIG)

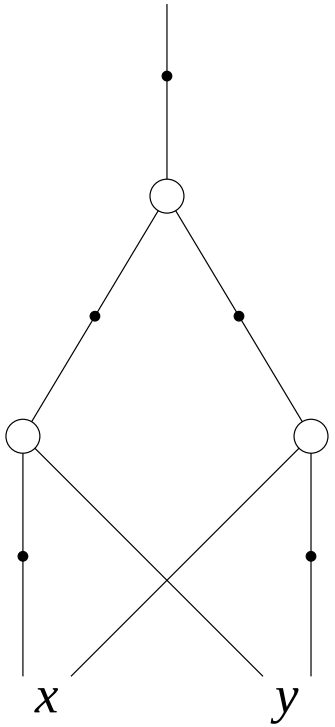
- logische Basis-Operationen: Konjunktion und Negation
- DAG-Darstellung:

  Operations-Knoten sind alles Konjunktionen

  Negation/Vorzeichen als Kanten-Attribut (daher *signed*)

  (platzsparend als LSB im Pointer)
- automatisches Sharen syntaktisch isomorpher Teilgraphen
- Vereinfachungs-Regeln mit konstantem Look-Ahead

XOR als AIG



Negationen/Vorzeichen sind Kantenattribute

(gehören nicht zu den Knoten)

```
typedef struct AIG AIG;

struct AIG
{
  enum Tag tag;                 /* AND, VAR */
  void *data[2];
  int mark, level;              /* traversal */
  AIG *next;                    /* hash collision chain */
};

#define sign_aig(aig) (1 & (unsigned) aig)
#define not_aig(aig) ((AIG*)(1 ^ (unsigned) aig))
#define strip_aig(aig) ((AIG*)(~1 & (unsigned) aig))
#define false_aig ((AIG*) 0)
#define true_aig ((AIG*) 1)
```

Annahme für Korrektheit:

`sizeof(unsigned) == sizeof(void*)`

Vorteile von Signs

- Alignment moderner Prozessoren “verschwendet” sowieso mehrere LSBs

  Alignment ist typischerweise 4 oder 8 Bytes $\Rightarrow$ 2 oder 3 LSBs übrig
  `malloc` liefert *aligned blocks* (z.B. 8 Byte aligned auf Sparc)

- negierte und unnegierte Formel enspricht einem Knoten

  (potentiell Halbierung des Speicherplatzes)

- maximale Reduktion auf einen einzigen Operations-Typ (AND)

- Negation extrem effizient (Bit im Pointer umsetzen)

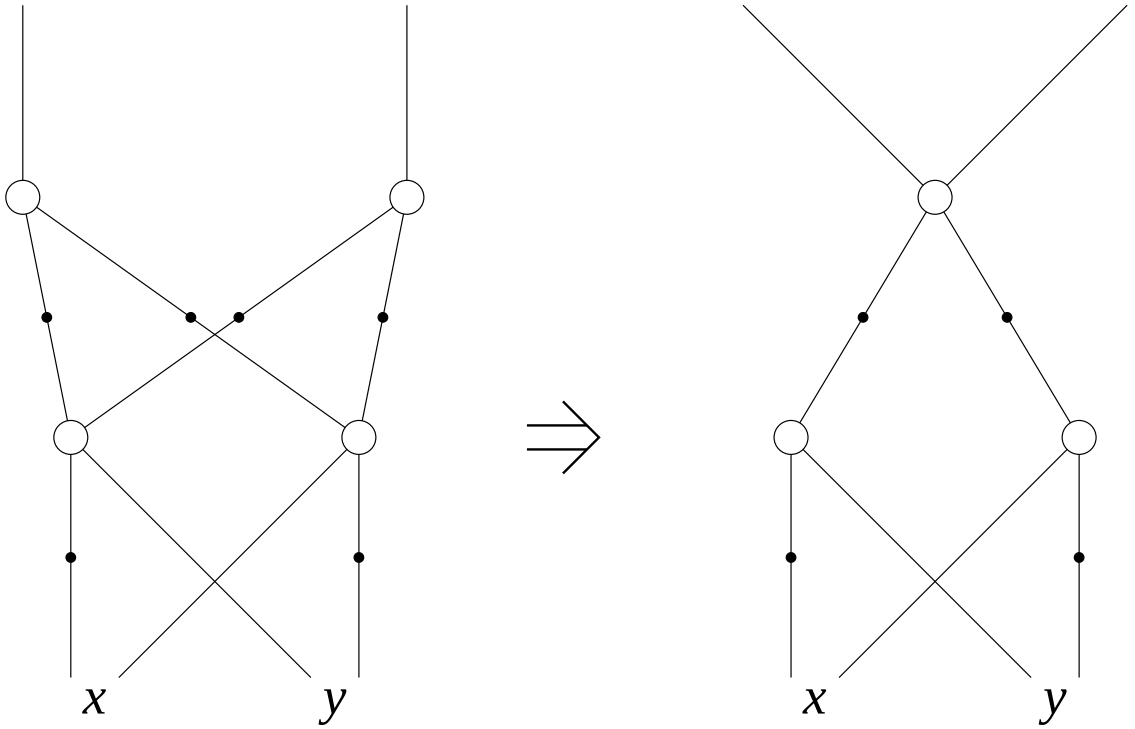
- erlaubt zusätzliche Vereinfachungsregeln

  direkte Erkennung von Argumenten unterschiedlichem Vorzeichens

Vereinfachungs-Regeln

```
int
simp_aig (enum Tag tag, void *d0, void *d1, AIG ** res_ptr)
{
  if (tag == AND)
    {
      if (d0 == false_aig || d1 == false_aig || d0 == not_aig (d1))
        { *res_ptr = false_aig; return 1; }

      if (d0 == true_aig || d0 == d1)
        { *res_ptr = d1; return 1; }

      if (d1 == true_aig)
        { *res_ptr = d0; return 1; }
    }

  return 0;
}
```

Syntaktisches Sharen



Verschmelzen von Knoten mit gleichen Kindern

Ad: Syntaktisches Sharen

- wird auch *Algebraische Reduktion* genannt
- Hauptvorteil ist automatische Kompaktifizierung
- Implementierung:

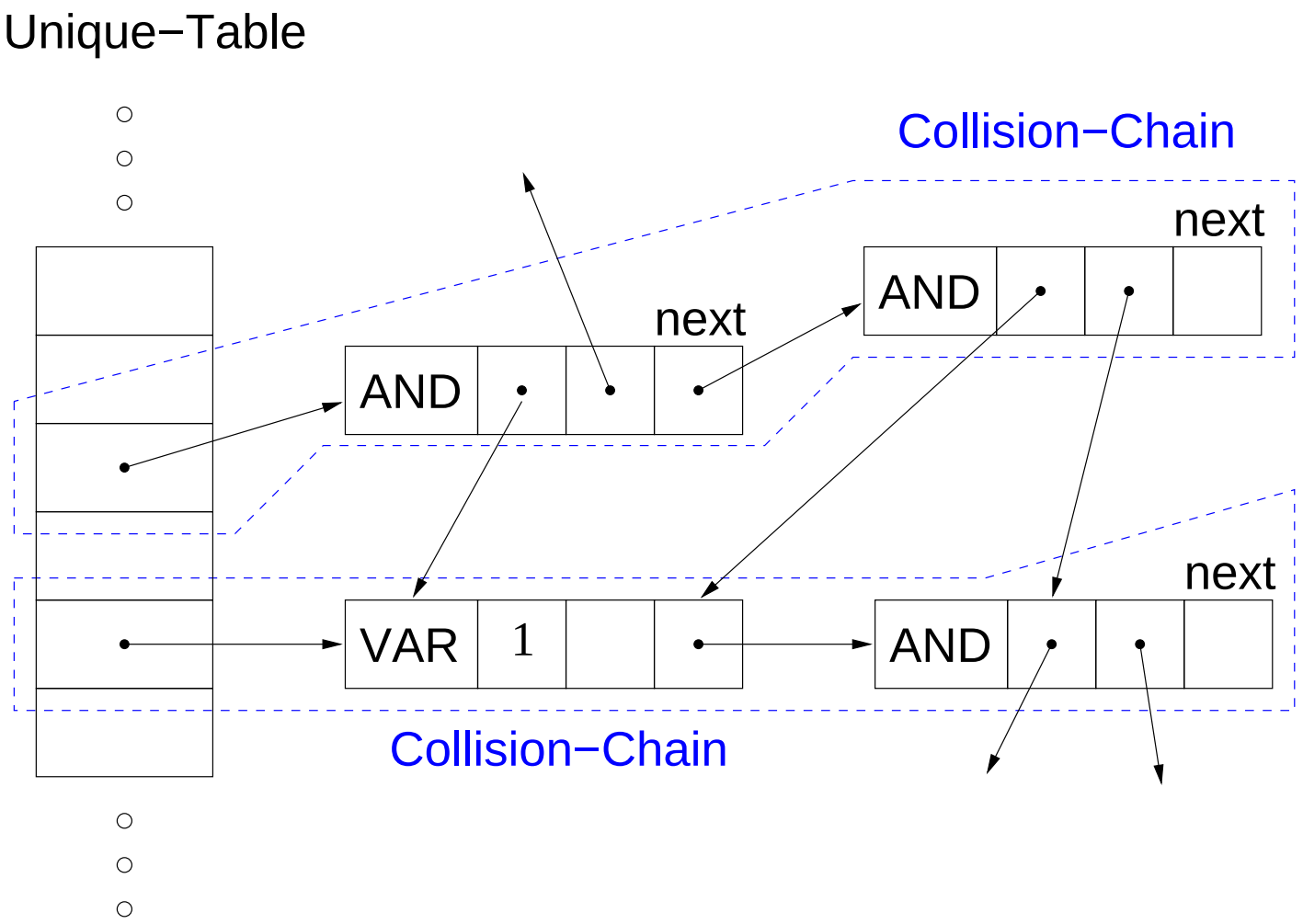
  Knoten in einer Hash-Tabelle (Unique-Table) gespeichert
- On-The-Fly Reduktion:

  Invariante: zwei Knoten haben immer unterschiedliche Kinder

  Erzeugung neuer Knoten: zunächst Suche nach äquivalentem Knoten

  Suche erfolgreich liefert äquivalenten Knoten als Resultat

Unique-Table



Suche nach Äquivalenten Knoten in C

```
#define UNIQUE_SIZE (1 << 20)
AIG *unique[UNIQUE_SIZE];

AIG **
find_aig (enum Tag tag, void *d0, void *d1)
{
  AIG *r, **p;
  unsigned h = (tag + ((int) d0) * 65537 + 13 * (int) d1);
  h = h & (UNIQUE_SIZE - 1);    /* modulo UNIQUE_SIZE */
  for (p = unique + h; (r = *p); p = &r->next)
    if (r->tag == tag && r->data[0] == d0 && r->data[1] == d1)
      break;

  return p;
}
```

Ad: Suche nach Äquivalenten Knoten in C

1. Bilde Hashwert als Kombination des Tags und der Pointer-Werte

   (bzw. der Variablen-Indizes bei Konstanten)

2. Normalisiere Hashwert auf Grösse der Unique-Table

3. Durchsuche Collision-Chain an der Hashwert-Position in der Unique-Table

4. Vergleiche Knoten mit neu zu erzeugendem Knoten

5. Falls gleich gib Pointer auf Link zu diesem Knoten zurück

6. Ansonsten gib Pointer auf letztes leeres Link-Feld in Chain zurück

Speichern von Knoten in Unique-Table

```
void
insert_aig (AIG * aig)
{
  AIG **p;
  int l[2];

  p = find_aig (aig->tag, aig->data[0], aig->data[1]);
  assert (!*p);
  aig->next = *p;
  *p = aig;

  if (aig->tag == AND)
    {
      l[0] = strip_aig (aig->data[0])->level;
      l[1] = strip_aig (aig->data[1])->level;
      aig->level = 1 + ((l[0] < l[1]) ? l[1] : l[0]);
    }
  else
    aig->level = 0;
}
```

`find_aig` gibt (neue) Position des gehashten Knotens zurück

```
AIG *
new_aig (enum Tag tag, void *data0, void *data1)
{
  AIG *res;
  if (tag == AND && data0 > data1)
    SWAP (data0, data1);
  if (tag == AND && (simp_aig (tag, data0, data1, &res)))
    return res;
  if ((res = *find_aig (tag, data0, data1)))
    return res;
  res = (AIG *) malloc (sizeof (AIG));
  memset (res, 0, sizeof (AIG));
  res->tag = tag;
  res->data[0] = data0;
  res->data[1] = data1;
  insert_aig (res);

  return res;
}
```

Abgeleitete Konstruktoren

```
AIG *
var_aig (int variable_index)
{
  return new_aig (VAR, (void *) variable_index, 0);
}

AIG *
and_aig (AIG * a, AIG * b)
{
  return new_aig (AND, a, b);
}

AIG *
or_aig (AIG * a, AIG * b)
{
  return not_aig (and_aig (not_aig (a), not_aig (b)));
}

AIG *
xor_aig (AIG * a, AIG * b)
{
  return or_aig (and_aig (a, not_aig (b)), and_aig (not_aig (a), b));
}
```

Zählen von AIG Knoten

```
int
count_aig (AIG * aig)
{
  if (sign_aig (aig))
    aig = not_aig (aig);
  if (aig->mark)
    return 0;
  aig->mark = 1;
  if (aig->tag == AND)
    return count_aig (aig->data[0]) + count_aig (aig->data[1]) + 1;
  else
    return 1;
}
```

Man muss explizit die Vorzeichen behandlen

aber sonst genauso DFS-orientiert wie bei DAG-Darstellung

Markieren von AIG Knoten

```
void
mark_aig (AIG * aig, int new_value)
{
  if (sign_aig (aig))
    aig = not_aig (aig);
  if (aig->mark == new_value)
    return;
  aig->mark = new_value;
  if (aig->tag == AND)
    {
      mark_aig (aig->data[0], new_value);
      mark_aig (aig->data[1], new_value);
    }
}
```

Weniger Fälle und weniger Code als bei DAG Darstellung!

Übersetzung von DAG nach AIG

```
AIG *
node2aig (Node * node)
{
  switch (node->tag)
    {
    case VAR:
      return new_aig (VAR, (void *) node->data.idx, 0);
    case CONST:
      return node->data.idx ? true_aig : false_aig;
    case AND:
      return and_aig (node2aig (node->data.child[0]),
                      node2aig (node->data.child[1]));
    case OR:
      return or_aig (node2aig (node->data.child[0]),
                     node2aig (node->data.child[1]));
    case XOR:
      return xor_aig (node2aig (node->data.child[0]),
                      node2aig (node->data.child[1]));
    default:
      assert (node->tag == NOT);
      return not_aig (node2aig (node->data.child[0]));
    }
}
```

Visualisierung

```
void
vis_aig_aux (AIG * aig, FILE * file, int max_level)
{
  assert (!sign_aig (aig));
  if (aig->mark)
    return;
  aig->mark = 1;
  switch (aig->tag)
    {
    case VAR:
      fprintf (file, "%p@%d:c'%d\n", aig, 2 * max_level, (int) aig->data[0]);
      break;
    default:
      assert (aig->tag == AND); /* TODO: constants */
      vis_aig_edge (aig, aig->data[0], file, max_level);
      vis_aig_edge (aig, aig->data[1], file, max_level);
      vis_aig_aux (strip_aig (aig->data[0]), file, max_level);
      vis_aig_aux (strip_aig (aig->data[1]), file, max_level);
      break;
    }
}
```

Simple DFS: Graph wird rekursiv in Datei geschrieben

AIG Missing Features

- robusterer C-Code notwendig

  (z.B. 64 Bit-Anpassung)

- Speicher-Management fehlt ganz

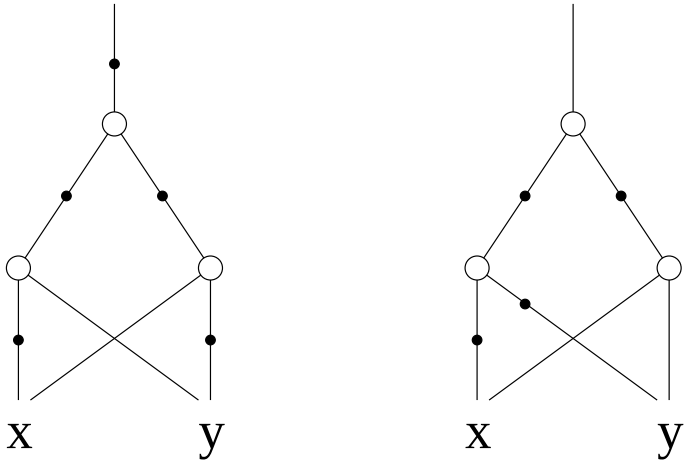
  (z.B. Reference-Counting oder GC)

- Eingabe-Format und Parser fehlen

- Vereinfachungsregeln für Enkel fehlen

  (z.B. Distributiv-Gesetz)

Nicht-Kanonizität von AIGs



$$\overline{x} \cdot y \vee x \cdot \overline{y} \quad \equiv \quad (\overline{x} \vee \overline{y}) \cdot (x \vee y)$$

beide Formeln und AIGs beschreiben XOR von x und y

(Ausmultiplizieren der rechten Formel ergibt linke Formel)

i.Allg. gibt es mehrere AIGs für dieselbe boolesche Funktion

Binary Decision Diagrams

- neue dreistellige Basis-Operation ITE (if-then-else):

  Bedingung ist immer eine Variable

- gehen zurück auf Shannon (deshalb auch Shannon-Graphs)

- meist verwendete Version sind die ROBDDs

  Reduced Ordered Binary Decision Diagrams

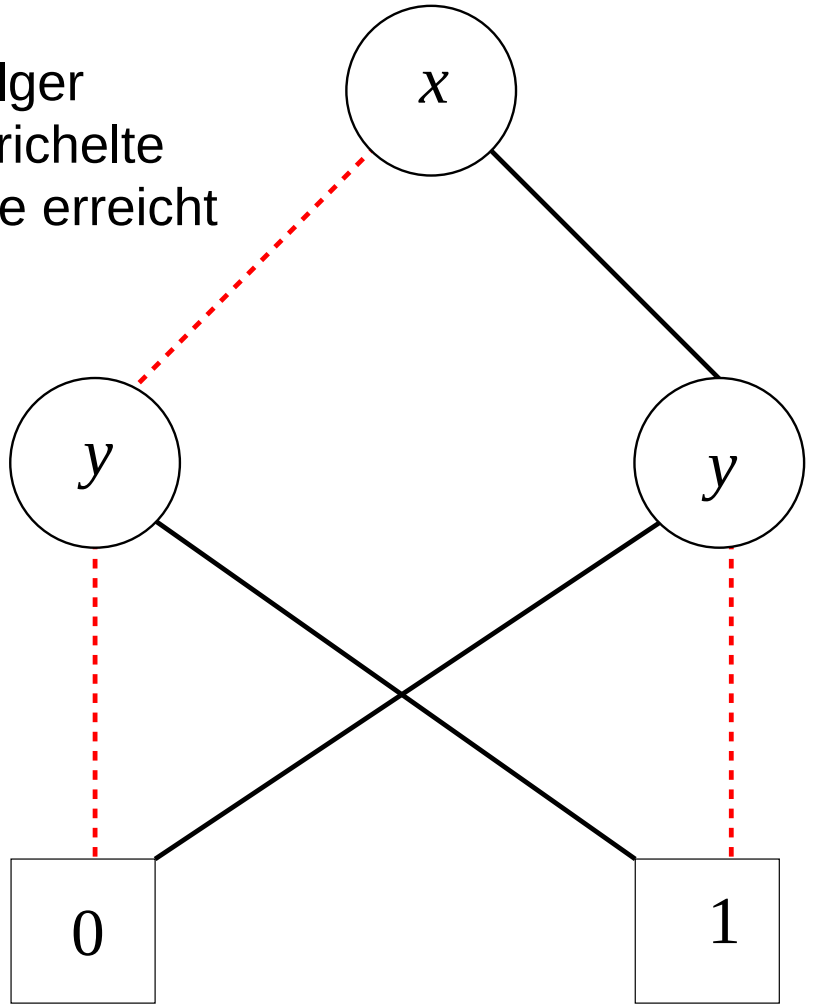
- [Bryant86] hat Kanonizität von ROBDDs gezeigt:

  Jede Boolesche Funktion hat genau einen ROBDD

Graphische Darstellung

Knoten ist
markiert mit
einer Variablen

ELSE−Nachfolger
wird über gestrichelte
oder rote Kante erreicht



THEN−Nachfolger
wird über durchgezogene
Kante erreicht

Boolesche
Konstante
FALSE

Boolesche
Konstante
TRUE

Semantik

- innere Knoten sind ITE, Blätter sind boolesche Konstante
- Schreibweise $ite(x, f_1, f_0)$ bedeutet *wenn* x *dann* f_1 *ansonsten* f_0

  (beachte: ELSE-Argument f_0 kommt hinter f_1 trotz umgekehrter Indizierung)
- Semantik $eval$ produziert booleschen Ausdruck aus einem BDD

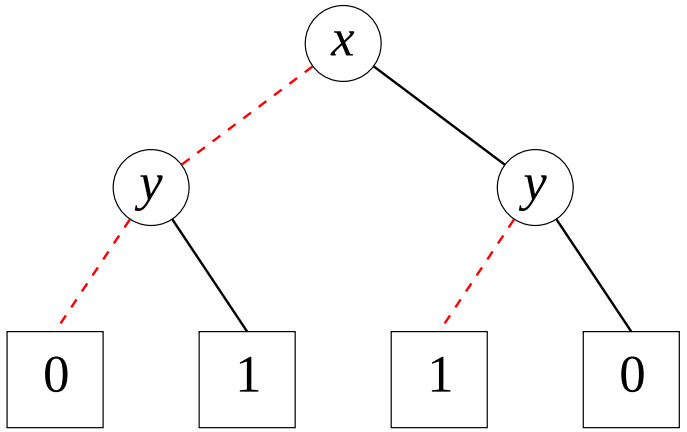
$$\begin{aligned} eval(0) &\equiv 0 \\ eval(1) &\equiv 1 \\ eval((ite(x, f_1, f_0)) &\equiv x \cdot eval(f_1) \vee \overline{x} \cdot eval(f_0) \end{aligned}$$

- BDDs sind auch wieder algebraisch reduzierte DAG's

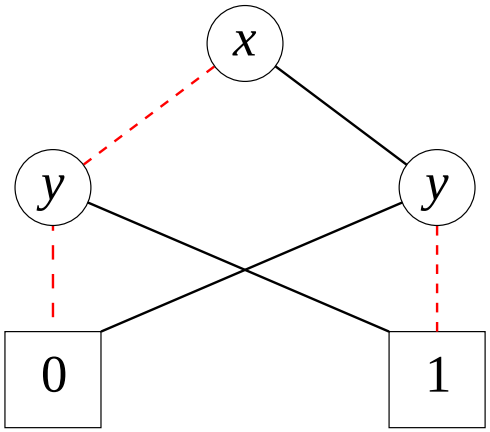
  (max. Sharen von isomorphen Teil-Graphen wie bei AIGs)
- Negations-Kanten wie bei AIGs möglich

Von der Funktionstabelle zum BDD

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

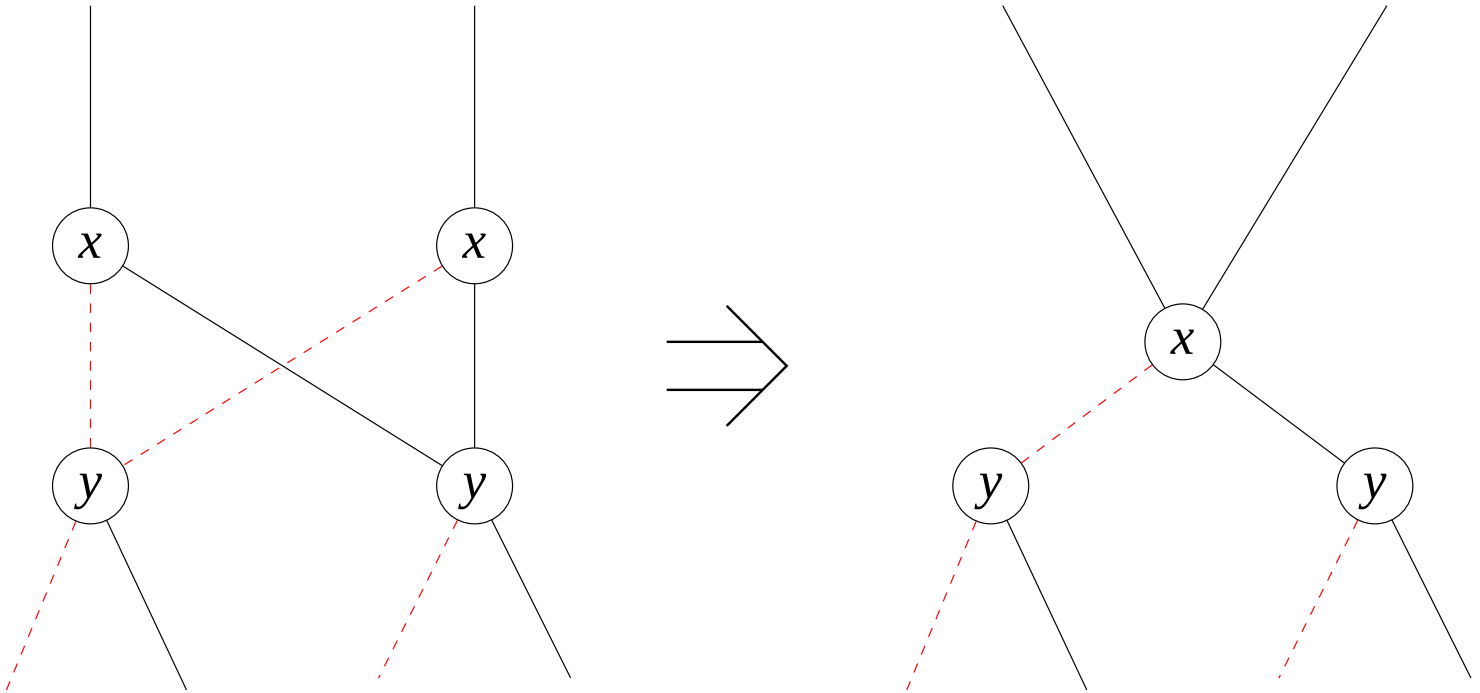


Entscheidungs-Baum



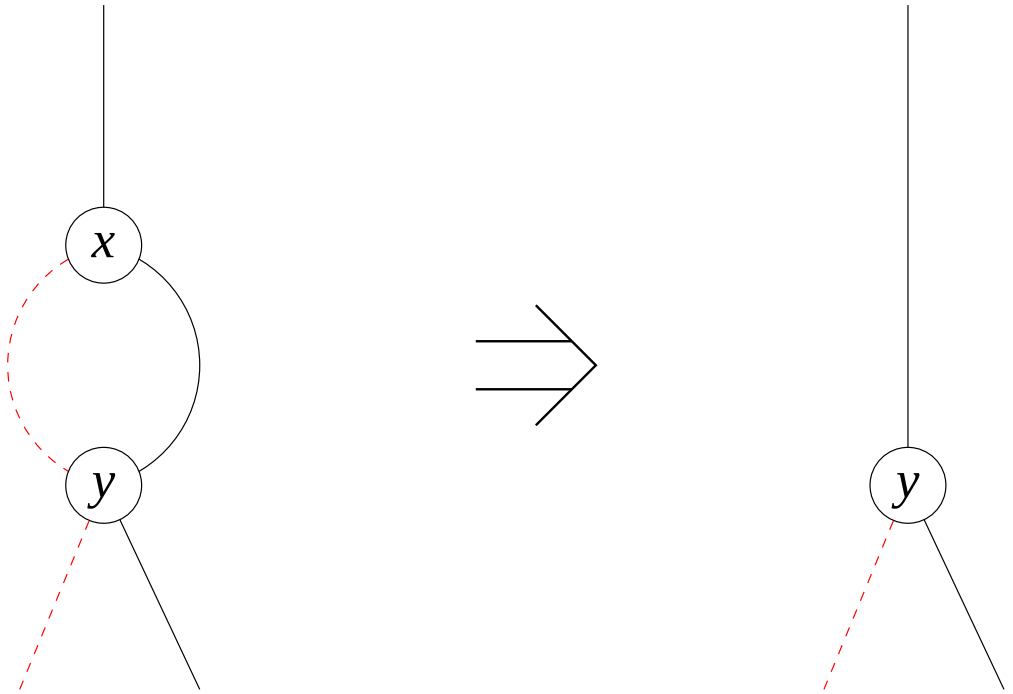
Entscheidungs-Diagramm (DAG)

Algebraische Reduktions-Regel



Maximales Sharen isomorpher Teil-Graphen

Semantische Reduktions-Regel



Elimination redundanter Innerer Knoten

Reduced Ordered BDDs

- maximale algebraische und semantische Reduktion

  (das Reduced in ROBDDs)

- Variablen von der Wurzel zu den Blättern sind gleich geordnet

  (das Ordered in ROBDDs)

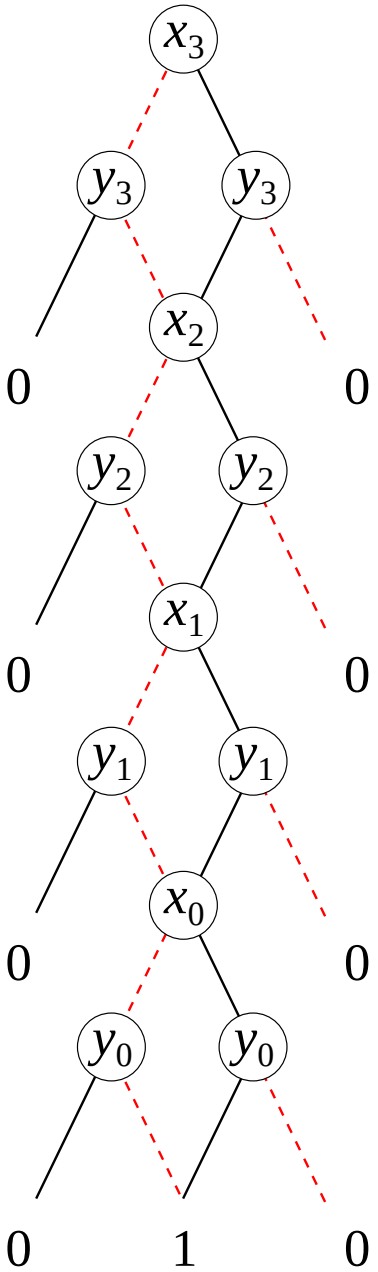
- diesen Annahmen machen BDDs kanonisch modulo Variablenordnung

- unterschiedliche Ordnungen führen i.Allg. zu unterschiedlichen BDDs

- Variablenordnung bestimmt essentiell die Grösse von BDDs

- **im weiteren meinen wir immer ROBDDs, wenn wir BDD sagen**

Linearer BDD für Vektoren-Vergleich



Boolesche Funktion:

$$\Pi_{i=0}^{n-1} x_i = y_i$$

Verschränkte Variablen-Ordnung (interleaved)

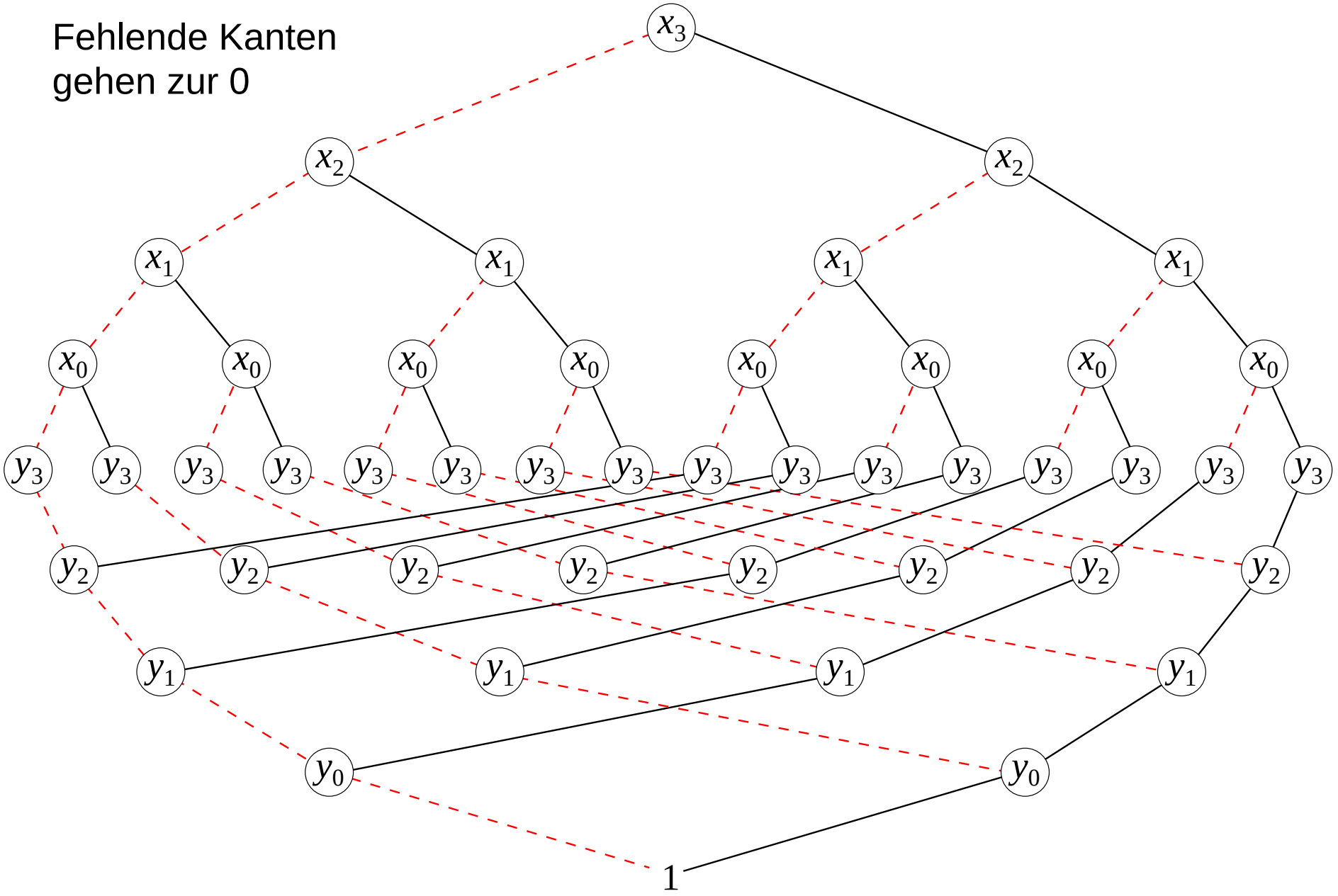
$$x_3 > y_3 > x_2 > y_2 > x_1 > y_1 > x_0 > y_0$$

Allgemein:

Vergleich zweier n-Bit Vektoren braucht bei verschränkter Ordnung $3 \cdot n$ innere Knoten.

Exponentieller BDD für Vektoren-Vergleich

Fehlende Kanten
gehen zur 0



Variablen-Ordnung

- es gibt exponentielle Unterschiede zwischen Variablen-Ordnungen
- es gibt aber leider auch *exponentielle* Funktionen:

  BDD ist immer exponentiell, z.B. mittlere Ausgabe-Bits von Multiplizierer
- es gibt Heuristiken zur statischen Variablen-Ordnung:

  meist einfache Auflistung der Variablen in DFS des DAG/AIG

  (etwa wie in `input_cone_aig`)
- zusätzlich gibt es noch *dynamische Umordnung von Variablen*

  basiert auf *in-place* Austausch von benachbarten Variablen

Tautologie, Erfüllbarkeit, Äquivalenz

Kanonizität der BDDs für boolesche Funktionen ergibt:

- ein BDD ist eine Tautologie, gdw. er nur aus dem 1-Blatt besteht
- ein BDD ist erfüllbar, gdw. er nicht aus dem 0-Blatt besteht
- zwei BDDs sind äquivalent, gdw. die BDDs sind isomorph

  (werden BDDs wie AIGs in der gleichen Unique-Tabelle gespeichert, dann ist der Test auf Äquivalenz ein einfacher Pointer-Vergleich)

Frage: Wo ist die NP-Vollständig der Erfüllbarkeit geblieben?

Antwort: Versteckt sich im Auftwand der Erzeugung eines BDDs.

Shannonsche Expansionsregel

lautet wie folgt:

$$f(x) \quad \equiv \quad x \cdot f(1) \vee \bar{x} \cdot f(0)$$

Sei nun x die oberste Variable zweier BDDs f und g:

$$f \equiv ite(x, f_1, f_0) \qquad g \equiv ite(x, g_1, g_0)$$

mit f_i bzw. g_i die Kinder von f und g für $i = 0, 1$. Dann folgt

$$f(0) = f_0 \qquad g(0) = g_0 \qquad f(1) = f_1 \qquad g(1) = g_1$$

da nach dem **R** in **R**OBDD x nur ganz oben in f und g vorkommt.

$$\begin{aligned}(f@g)(x) &\equiv x \cdot (f@g)(1) \vee \bar{x} \cdot (f@g)(0) \\ &\equiv x \cdot (f(1)@g(1)) \vee \bar{x} \cdot (f(0)@g(0)) \\ &\equiv x \cdot (f_1@g_1) \vee \bar{x} \cdot (f_0@g_0)\end{aligned}$$

wobei $@$ eine beliebige zweistellige boolesche Operation ist (z.B. $\wedge$, $\vee$, $\oplus$, ...)

Rekursives Schema zur Berechnung von Operationen auf BDDs

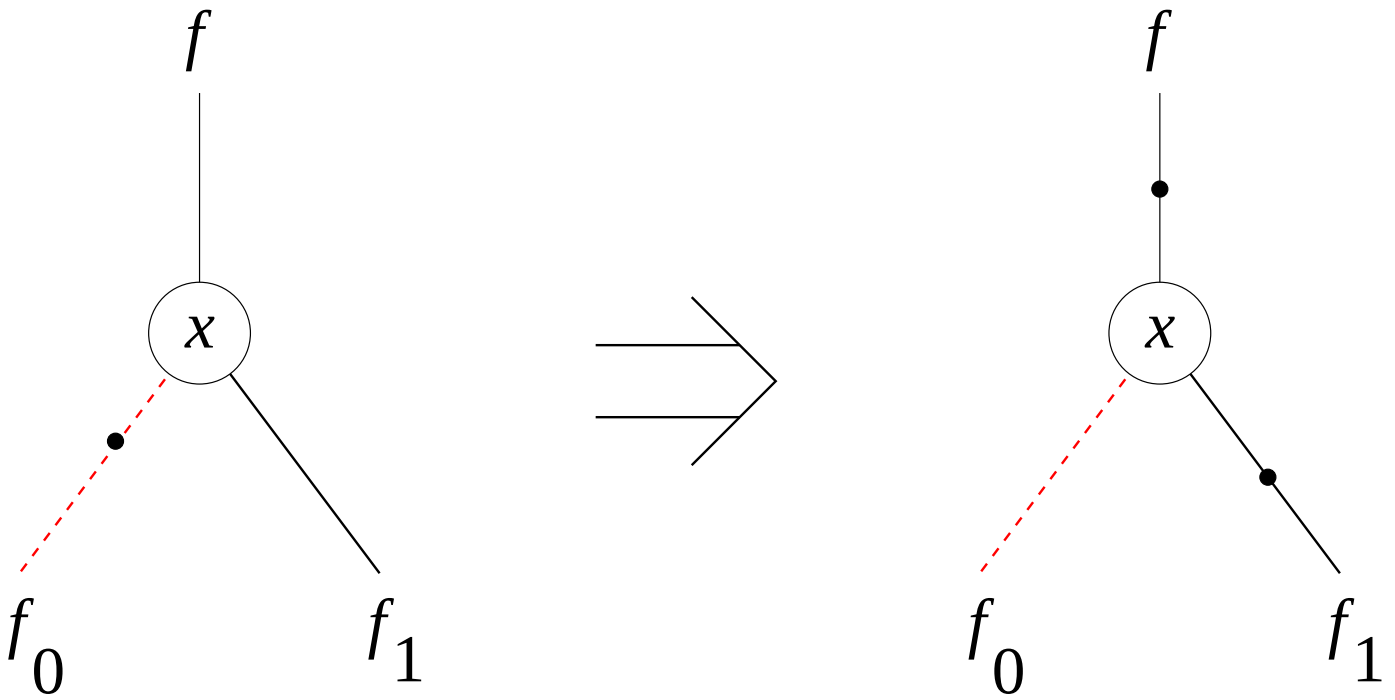
Repräsentation von BDDs in C

```
typedef struct BDD BDD;

struct BDD
{
  int idx, mark;
  BDD *child[2], *next;
};

#define sign_bdd(ptr) (1 & (unsigned) ptr)
#define strip_bdd(ptr) ((BDD*) (~1 & (unsigned) ptr))
#define not_bdd(ptr) ((BDD*) (1 ^ (unsigned) ptr))
#define true_bdd ((BDD*) 1)
#define false_bdd ((BDD*) 0)

#define is_constant_bdd(ptr) \
  ((ptr) == true_bdd || (ptr) == false_bdd)
```

Unique-Tabelle für BDDs

```
#define UNIQUE_SIZE (1 << 20)
BDD *unique[UNIQUE_SIZE];

BDD **
find_bdd (int idx, BDD * c0, BDD * c1)
{
  BDD *r, **p;
  unsigned h = (idx + ((int) c0) * 65537 + 13 * (int) c1);
  h = h & (UNIQUE_SIZE - 1);
  for (p = unique + h; (r = *p); p = &r->next)
    if (r->idx == idx && r->child[0] == c0 && r->child[1] == c1)
      break;

  return p;
}
```

Normalisierung von BDD Knoten



$$
\begin{aligned}
ite(x, f_1, \overline{f_0}) &\equiv x \cdot f_1 \vee \overline{x} \cdot \overline{f_0} \equiv \overline{(\overline{x} \vee \overline{f_1}) \cdot (x \vee f_0)} \\
&\equiv \overline{x \cdot \overline{f_1} \vee \overline{x} \cdot f_0 \vee \overline{f_1} \cdot f_0} \\
&\equiv \overline{x \cdot \overline{f_1} \vee \overline{x} \cdot f_0} \equiv \overline{ite(x, \overline{f_1}, f_0)}
\end{aligned}
$$

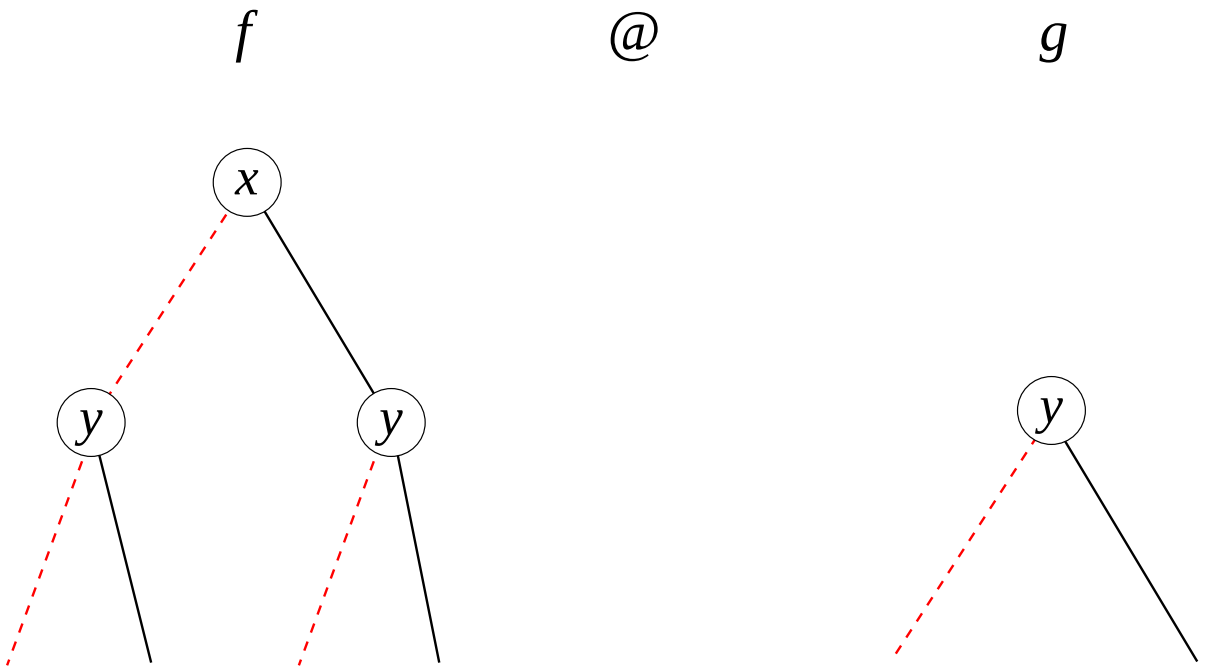
Hilfs-Konstruktor für BDDs

```
BDD *
new_bdd_aux (int idx, BDD * c0, BDD * c1)
{
  BDD *res;

  assert (!sign_bdd (c0));

  if ((res = *find_bdd (idx, c0, c1)))
    return res;

  res = (BDD *) malloc (sizeof (BDD));
  memset (res, 0, sizeof (BDD));
  res->idx = idx;
  res->child[0] = c0;
  res->child[1] = c1;
  *find_bdd (idx, c0, c1) = res;

  return res;
}
```

Konstruktor für BDDs

```
BDD *
new_bdd (int idx, BDD * c0, BDD * c1)
{
  BDD *res;
  int sign;

  if (c0 == c1)
    return c0;

  if ((sign = sign_bdd (c0)))
    {
      c0 = not_bdd (c0);
      c1 = not_bdd (c1);
    }

  res = new_bdd_aux (idx, c0, c1);

  return sign ? not_bdd (res) : res;
}
```

Unterschiedliche Top-Indizes



$$(f \wedge g)(x) \quad \equiv \quad x \cdot (f_1 \wedge g) \vee \bar{x} \cdot (f_0 \wedge g)$$

bei nicht passenden Indizes wird nur ein Knoten zerlegt

Top-Index Berechnung

```
int
top_idx_bdd (BDD * a, BDD * b)
{
  int res[2];
  res[0] = (is_constant_bdd (a)) ? -1 : strip_bdd (a)->idx;
  res[1] = (is_constant_bdd (b)) ? -1 : strip_bdd (b)->idx;
  return res[res[0] < res[1]];
}
```

gibt einfach den Index der Variablen im obersten Knoten zurück

(hier kann einer der beiden Argumente eine Konstante sein)

Berechnung einfacher Kofaktoren

```
BDD *
cofactor (BDD * bdd, int pos, int idx)
{
  BDD *res;
  int sign;
  if (is_constant_bdd (bdd))
    return bdd;
  if ((sign = sign_bdd (bdd)))
    bdd = not_bdd (bdd);
  res = (bdd->idx == idx) ? bdd->child[pos] : bdd;
  return sign ? not_bdd (res) : res;
}
```

`pos`-ter Kofaktor von `bdd` ist `pos`-tes Kind wenn Variablen-Index passt

ansonsten wird der gleiche BDD zurückgegeben

(man beachte Propagierung des Vorzeichens)

Divide

```
void
cofactor2 (BDD * a, BDD * b, BDD * c[2][2], int *idx_ptr)
{
  int idx = *idx_ptr = top_idx_bdd (a, b);
  c[0][0] = cofactor (a, 0, idx);
  c[0][1] = cofactor (a, 1, idx);
  c[1][0] = cofactor (b, 0, idx);
  c[1][1] = cofactor (b, 1, idx);
}
```

bestimme Top-Index der beiden BDDs `a` und `b` und

berechne Kofaktoren bezüglich des Top-Index

Basis-Funktoren für Apply

```
BDD *
basic_and (BDD * a, BDD * b)
{
  assert (is_constant_bdd (a) && is_constant_bdd (b));
  return (BDD *) (((unsigned) a) & (unsigned) b);
}

BDD *
basic_or (BDD * a, BDD * b)
{
  assert (is_constant_bdd (a) && is_constant_bdd (b));
  return (BDD *) (((unsigned) a) | (unsigned) b);
}

BDD *
basic_xor (BDD * a, BDD * b)
{
  assert (is_constant_bdd (a) && is_constant_bdd (b));
  return (BDD *) (((unsigned) a) ^ (unsigned) b);
}
```

Bryant's Apply Algorithmus

```
typedef BDD *(*BasicFunctor) (BDD *, BDD *);

BDD *
apply (BasicFunctor op, BDD * a, BDD * b)
{
  BDD *tmp[2], *c[2][2];
  int idx;
  if (is_constant_bdd (a) && is_constant_bdd (b))
    return op (a, b);
  cofactor2 (a, b, c, &idx);
  tmp[0] = apply (op, c[0][0], c[1][0]);
  tmp[1] = apply (op, c[0][1], c[1][1]);
  return new_bdd (idx, tmp[0], tmp[1]);
}
```

Definition eines Funktions-Pointer-Typ, zur Aufnahme von Operationen

Abkürzungen

```
BDD *
and_bdd (BDD * a, BDD * b)
{
  return apply (basic_and, a, b);
}

BDD *
or_bdd (BDD * a, BDD * b)
{
  return apply (basic_or, a, b);
}

BDD *
xor_bdd (BDD * a, BDD * b)
{
  return apply (basic_xor, a, b);
}

BDD *
var_bdd (int idx)
{
  return new_bdd (idx, false_bdd, true_bdd);
}
```

Zusammenfassung Apply

- es fehlt noch ein **Cache**: dann linear in beiden Argumenten

  im Cache merkt man sich schon mal durchgeführte Berechnungen

  dann können alle Knoten von beiden Argumenten nur einmal auftreten

- **Optimierung**: Spezialalgorithmen für AND und XOR einführen

  auch wenn nur ein Argument konstant ist, kann man früher abbrechen

  ebenso, wenn beide Argumente bis auf das Vorzeichen identisch sind

- weitere BDD-Algorithmen basieren auf gleichem Schema:

  Kofaktoren bestimmen, Rekursion, mit Shannon zusammensetzen

Zusammenfassung BDDs

- Kanonische Datenstruktur für boolesche Funktionen
- kein Allheilmittel aber wesentlich besser als DNF
- häufig kompakt für in der Praxis auftretenden Funktionen
- Trade-Off: *Platz statt Zeit*
- Anwendungen:

  Synthese, Symbolische Simulation, Equivalence Checking, Model-Checking