Model-Checking

(SS 2008)

Stefan Schwoon

Institut für Informatik (I7)
Technische Universität München

Organisatorisches

Vorlesung: Mittwoch, 10:45–12:15, MI HS 3
Donnerstag, 14:45–16:15, MI HS 2
Dozent: Stefan Schwoon, schwoon@in.tum.de
Sprechstunde nach Vereinbarung, MI 03.011.053

Übungen: jede Woche, nach Vereinbarung
Übungsleiter: Christian Schallhart, schallha@in.tum.de
MI 03.011.042

Stunden: 4V+2Ü bzw. 8 ECTS-Credits
Prüfung am Ende des Semesters (mündlich oder schriftlich)
Voraussetzung: Teilnahme an den Übungen

Ankündigungen

Webseite:

`http://www7.in.tum.de/um/courses/mc/ss2008/`

dort auch Folien, Übungsblätter etc.

E-Mails:

für kurzfristige Ankündigungen (Terminänderungen etc.)

in die Liste eintragen

Sonstiges

Voraussetzungen:

Grundkenntnisse in Logik, Diskrete Strukturen, Graphen, . . .

Literatur

Clarke, Grumberg, Peled: Model Checking, MIT Press, 1999

Emerson: Temporal and Modal Logic, Kapitel 16 im Handbook of Theoretical Computer Science, vol. B, Elsevier, 1991

Vardi: An Automata-Theoretic Approach to Linear Temporal Logic, LNCS 1043, 1996

Holzmann: The SPIN Model Checker, Addison-Wesley, 2003

Teil 1: Einführung

Motivation

Computersysteme dringen in immer mehr Bereichen unseres Lebens vor:

- Heim-PC
- Mobiltelefon
- Steuersysteme in Autos, Flugzeugen, ...
- Bankautomaten

Je mehr Computer eingesetzt werden, desto größer die Verletzlichkeit durch fehlerhafte Hardware oder Software;

je komplexer die Systeme werden, desto schwieriger sind sie gegen Fehler zu schützen.

Fehler können erhebliche wirtschaftliche Folgen haben oder gar Menschenleben gefährden (geschätzte Kosten durch Computer-Fehler in den USA: 60 Mrd. Dollar pro Jahr (Quelle: Der Spiegel)).

Pentium-Fehler (1994)

Der Pentium-Prozessor lieferte bei Fließkommadivisionen auf bestimmten Zahlenpaaren falsche Ergebnisse, z.B.

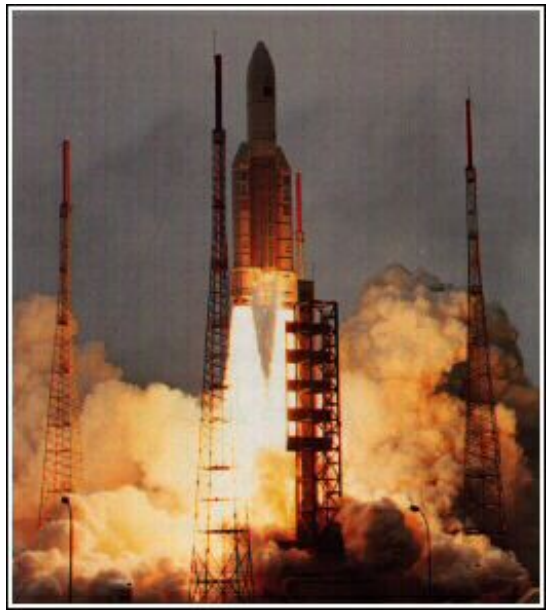
$$4195835 - (4195835/3145727) \times 3145727 = 256$$

Ursache: Aus Effizienzgründen benutzte der Prozessor zur Division eine Tabelle mit 1066 Einträgen, von denen fünf falsche waren.

Geschätzte Kosten der Umtauschaktion: 500 Mio. Dollar

(siehe auch `http://citeseer.nj.nec.com/pratt95anatomy.html`)

Absturz der Ariane 5 (1996)



Die Rakete brach 39 Sekunden nach dem Start infolge zu hohen Luftwiderstands auseinander und wurde gesprengt.

Die Ursache

Ein Sensor, der die horizontale Geschwindigkeit der Rakete maß, lieferte korrekte, jedoch unerwartet hohe Daten.

Die Meßdaten wurden von einer 64-Fließkommazahl in eine 16-bit-Integerzahl konvertiert. Weil die Zahl höher als erwartet war, entstand ein Überlauf.

Der Überlauf erzeugte eine Ausnahme, die nicht abgefangen wurde und das Programm zum Absturz brachte, die die Position der Rakete bestimmen sollte.

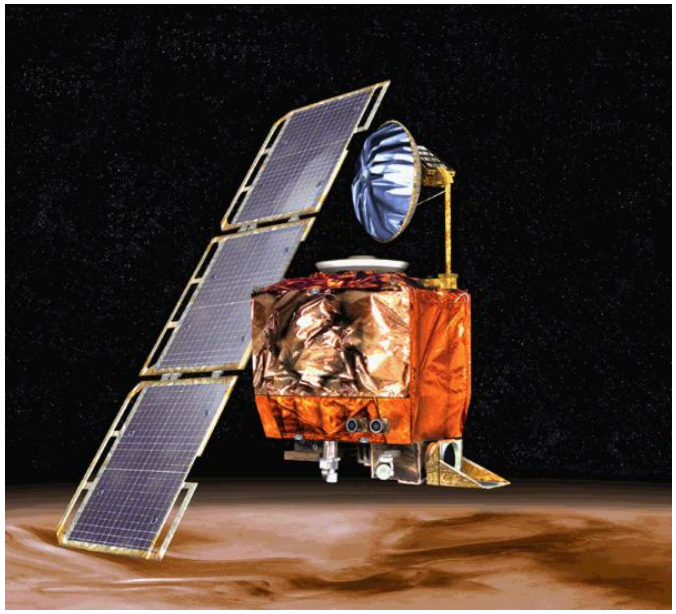
Dadurch erhielt der Steuerungscomputer falsche Höhendaten und leitete eine “Kurskorrektur” ein, die die Rakete von ihrer Laufbahn abbrachte.

Kosten: 500 Mio. – 2 Mrd. Euro

Genauere Informationen (Bericht der Untersuchungskommission):

`http://www.mssl.ucl.ac.uk/www_plasma/missions/cluster/about_cluster/cluster1/ariane5rep.html`

Verlust des Mars Climate Orbiter (1999)



Verursacht durch Verwechslung zwischen metrischen und britischen physikalischen Einheiten.

Abschaltung der USS Yorktown

Ein Seemann gab an einer Stelle versehentlich eine 0 ein, wo dies nicht vorgesehen war. Eine nachfolgende Division durch 0 erzeugte eine Ausnahme, die nicht abgefangen wurde und sich im Netz des Schiffs ausbreitete. Letzten Endes wurde der gesamte Antrieb des Schiffs abgeschaltet.

Auswege

Vermeidung von Fehlern:

- Entwicklung neuer Programmiersprachen
- Methoden des Software-Engineering

Aufspüren von Fehlern:

- Simulation, Testen

Beweis der Korrektheit:

- Deduktive Verifikation (Hoare)
- Automatische Verifikation (Model-Checking)

Simulation und Testen

Können Fehler in der Designphase (bei Simulation) bzw. im fertigen Produkt (beim Testen) nachweisen.

Methoden: Blackbox-/Whitebox-Testing, Abdeckungskriterien etc.

Vorteil: Viele Fehler können relativ schnell und kostengünstig gefunden werden.

Nachteil: Methode ist unvollständig:

> Es gibt weder ein Kriterium, das selbst bei 100% Abdeckung Fehlerfreiheit garantieren könnte, noch gibt es eine verläßliche Möglichkeit, die Anzahl verbliebener Fehler zu schätzen.
>
> Bei wachsender Komplexität wird es schwer, alle Fälle abzudecken.

Testen und Verifikation

Simulation und Testen können Fehler aufdecken, aber nicht deren Abwesenheit beweisen. (Es wird eine Untermenge der möglichen Systemabläufe betrachtet.)

$\longrightarrow$ Nicht ausreichend, insbesondere für sicherheitsrelevante Aspekte

Verifikation betrachtet alle Abläufe eines Systems

$\longrightarrow$ Entdeckung von Fehlern und Beweis der Korrektheit möglich

Deduktive Verifikation

Beweisführung über die formale Semantik (Dijkstra, Hoare et al.)

Beispiel: Hoare-Logik:

$$\{P\}\ S\ \{Q\}$$

Bedeutung: Wenn vor der Ausführung von S das Prädikat P gilt, gilt hinterher Q.

Beweisregeln, z.B.

$$\{P\}\ \mathbf{skip}\ \{P\} \qquad \{P[x/e]\}\ x := e\ \{P\} \qquad \frac{\{P\}\ S_1\ \{Q\} \wedge \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$$

Beweisregel für Schleifen

$$\{P\}\ \mathbf{while}\ \beta\ \mathbf{do}\ C\ \{Q\}$$

Man zeige, dass es eine Invariante I mit folgenden Eigenschaften gibt:

$$P \Rightarrow I \qquad \{I \wedge \beta\}\ C\ \{I\} \qquad I \wedge \neg\beta \Rightarrow Q$$

Terminierung: Man zeige, dass es eine Funktion $f(x, y, \ldots)$ der Programmvariablen gibt, so dass

$$\{\beta \wedge f(x, y, \ldots) = k\}\ C\ \{f(x, y, \ldots) < k\} \qquad f(x, y, \ldots) \leq 0 \Rightarrow \neg\beta$$

Programm C korrekt, wenn $\{true\}\ C\ \{P\}$ gilt, wobei P die zu berechnende Funktion ausdrückt

Eigenschaften der deduktiven Verifikation

Vorteil:

vollständige Methode, deren Leistungsfähigkeit nur durch die mathematischen Fähigkeiten des Benutzers begrenzt ist

Nachteile:

s.o.

Beweis von Hand mühsam (Hilfe durch Theorembeweiser)

o.g. Schema nur geeignet für sequentielle Systeme (keine Nebenläufigkeit)

nur geeignet für Ein-/Ausgabeprogramme, nicht für reaktive Systeme

Reaktive Systeme

Beispiele: Server, Geldautomat, Telefonvermittlung

verteilte Systeme, berechnen keine Funktion, Terminierung i.A. unerwünscht

Von Interesse ist das Verhalten über die gesamte Laufzeit gesehen, etwa:

> Es gibt keine Verklemmungen.
>
> Es werden niemals zwei Prozesse zugleich in einem kritischen Abschnitt sein.
>
> Wenn ein Prozess in einen kritischen Abschnitt eintreten will, wird er den kritischen Abschnitt auch irgendwann erreichen.

$\Rightarrow$ Formalisierung mit Hilfe temporaler Logiken

Model-Checking

Unter Model-Checking versteht man Techniken, die

verifizieren, ob ein gegebenes System eine gegebene Spezifikation erfüllt;

automatisch arbeiten;

entweder die Korrektheit des Systems bzgl. der Spezifikation feststellen;

oder ein Gegenbeispiel liefern, d.h. eine Ausführung, die die Spezifikation verletzt.

Vor- und Nachteile des Model-Checking

Vorteile:

automatisch(!)

geeignet für reaktive, nebenläufige, verteilte Systeme

kann allgemeine temporallogische Eigenschaften testen

Vor- und Nachteile des Model-Checking

Vorteile:

automatisch(!)

geeignet für reaktive, nebenläufige, verteilte Systeme

kann allgemeine temporallogische Eigenschaften testen

Nachteile:

Programme sind i.A. Turing-mächtig $\longrightarrow$ Unentscheidbarkeit

Vor- und Nachteile des Model-Checking

Vorteile:

automatisch(!)

geeignet für reaktive, nebenläufige, verteilte Systeme

kann allgemeine temporallogische Eigenschaften testen

Nachteile:

Programme sind i.A. Turing-mächtig $\longrightarrow$ Unentscheidbarkeit

Ansatz: entscheidbare Unterklassen, hier: endliche Automaten
Weiterführende Vorlesung: Model-Checking II, nächstes Semester

Vor- und Nachteile des Model-Checking

Vorteile:

- automatisch(!)
- geeignet für reaktive, nebenläufige, verteilte Systeme
- kann allgemeine temporallogische Eigenschaften testen

Nachteile:

- Programme sind i.A. Turing-mächtig $\longrightarrow$ Unentscheidbarkeit
  - Ansatz: entscheidbare Unterklassen, hier: endliche Automaten
  - Weiterführende Vorlesung: Model-Checking II, nächstes Semester
- Zustandsraum i.A. sehr groß $\longrightarrow$ rechenaufwändig

Vor- und Nachteile des Model-Checking

Vorteile:

automatisch(!)

geeignet für reaktive, nebenläufige, verteilte Systeme

kann allgemeine temporallogische Eigenschaften testen

Nachteile:

Programme sind i.A. Turing-mächtig $\longrightarrow$ Unentscheidbarkeit

Ansatz: entscheidbare Unterklassen, hier: endliche Automaten
Weiterführende Vorlesung: Model-Checking II, nächstes Semester

Zustandsraum i.A. sehr groß $\longrightarrow$ rechenaufwändig

Ansatz: effiziente Algorithmen und Datenstrukturen

Probleme bei Model-Checking

Aus den zuvor genannten Gründen können wir nicht hoffen, beliebige Eigenschaften beliebiger Programme automatisch zu verifizieren!

Gegebenenfalls muss ein Modell des Systems erstellt werden, das “unwichtige” Aspekte des Systems ignoriert.

Erstellung des Modells und der Spezifikation sowie der Verifikationsschritt selbst verursachen Aufwand und Kosten.

$\Rightarrow$ ökonomischer Nutzen bei kritischen Systemen, etwa bei Prozessoren, Kommunikationsprotokollen, ...

Erfolge des Model-Checking

Seit Ende der 1970er: Erforschung der theoretischen Grundlagen

Seit den 1990er Jahren: Industrielle Anwendungen

Zunächst Hardware-, später Software-Verifikation:

- Verifikation des Cache-Protokolls im IEEE-Futurebus+ (1992)

  Mit SMV konnten verschiedene Fehler gefunden werden, nachdem vier Jahre lang versucht wurde, das Protokoll mit anderen Mitteln zu validieren.

- Verifikation der Fließkomma-Einheit des Pentium4 (2001)
- Static Driver Verifier (Microsoft, 2000–2004) (Windows-Gerätetreiber)

Forschungsgruppen in Firmen: IBM, Intel, Microsoft, OneSpin Solutions, ...

Turing-Award 2007 an die “Paten” der Technik: Clarke, Emerson, Sifakis

Ziel der Vorlesung

Die Vorlesung vermittelte umfassende Grundkenntnisse zu Theorie und Anwendung von Model-Checking, speziell der Modellierung von Systemen, der Erstellung von Spezifikationen und ihrer Verifikation.

Modellierung: Transitionssysteme, Kripke-Strukturen, Tools (Spin, SMV)

Spezifikation: temporale Logik (LTL, CTL)

Verifikation: grundlegende Techniken und Erweiterungen (Halbordnungsreduktion, BDDs, Abstraktion, Bounded Model-Checking, . . .)

Was bedeutet eigentlich “Model-Checking”?

Was bedeutet eigentlich “Model-Checking”?

Was bedeutet eigentlich “Model-Checking”?

Was bedeutet eigentlich “Model-Checking”?

Begriff aus der Logik

temporale Logik: Erweiterung der Aussagenlogik

deutsch: “Modellprüfung” (siehe auch Def. in Wikipedia)

Aussagenlogik (Syntax)

Formeln der Aussagenlogik bestehen aus Grundaussagen, z.B.

$A \quad \widehat{=} \quad$ “Anna ist Archtektin”

$B \quad \widehat{=} \quad$ “Bruno ist Jurist”

und Verknüpfungen, z.B. $\wedge$ (“und”), $\vee$ (“oder”), $\neg$ (“nicht”), $\rightarrow$ (“impliziert”).

Beispiele

Beispiele für Formeln der AL:

$A \wedge B$ (“Anna ist Architektin und Bruno ist Jurist”)

$\neg B$ (“Bruno ist kein Jurist”)

Sind diese Aussagen wahr?

Beispiele

Beispiele für Formeln der AL:

$A \wedge B$ (“Anna ist Architektin und Bruno ist Jurist”)

$\neg B$ (“Bruno ist kein Jurist”)

Sind diese Aussagen wahr?

Antwort: Kommt drauf an.

Manche Formeln sind immer wahr ($A \vee \neg A$) oder immer falsch ($B \wedge \neg B$).

Aber im Allgemeinen wird eine Aussage bzgl. einer Belegung (oder: “Welt”) ausgewertet.

Semantik der Aussagenlogik

Eine Belegung $\mathcal{B}$ ist eine Funktion, die jeder Grundaussage einen Wahrheitswert (1 oder 0) zuweist.

Die Semantik einer Formel F sei eine Menge $[\![F]\!]$ von Belegungen, z.B.

falls $F = A$, dann $[\![F]\!] = \{ \mathcal{B} \mid \mathcal{B}(A) = 1 \}$;

falls $F = F_1 \wedge F_2$, dann $[\![F]\!] = [\![F_1]\!] \cap [\![F_2]\!]$; ...

Andere Schreibweise: $\mathcal{B} \models F$ gdw. $\mathcal{B} \in [\![F]\!]$.
Sprechweise: "$\mathcal{B}$ erfüllt F" bzw. "$\mathcal{B}$ ist Modell von F".

Das Model-Checking-Problem der AL

Problem: Gegeben eine Belegung $\mathcal{B}$ und eine aussagenlogische Formel F,
ist $\mathcal{B}$ Modell von F?

Das Model-Checking-Problem der AL

Problem: Gegeben eine Belegung $\mathcal{B}$ und eine aussagenlogische Formel F,
ist $\mathcal{B}$ Modell von F?

Algorithmus zur Lösung des Problems:

Werte der Grundaussagen aus $\mathcal{B}$ ablesen und in F einsetzen, dann Wahrheitstafel benutzen.

Das Model-Checking-Problem der Aussagenlogik

Problem: Gegeben eine Belegung $\mathcal{B}$ und eine aussagenlogische Formel F,
ist $\mathcal{B}$ Modell von F?

Algorithmus zur Lösung des Problems:

Werte der Grundaussagen aus $\mathcal{B}$ ablesen und in F einsetzen, dann Wahrheitstafel benutzen.

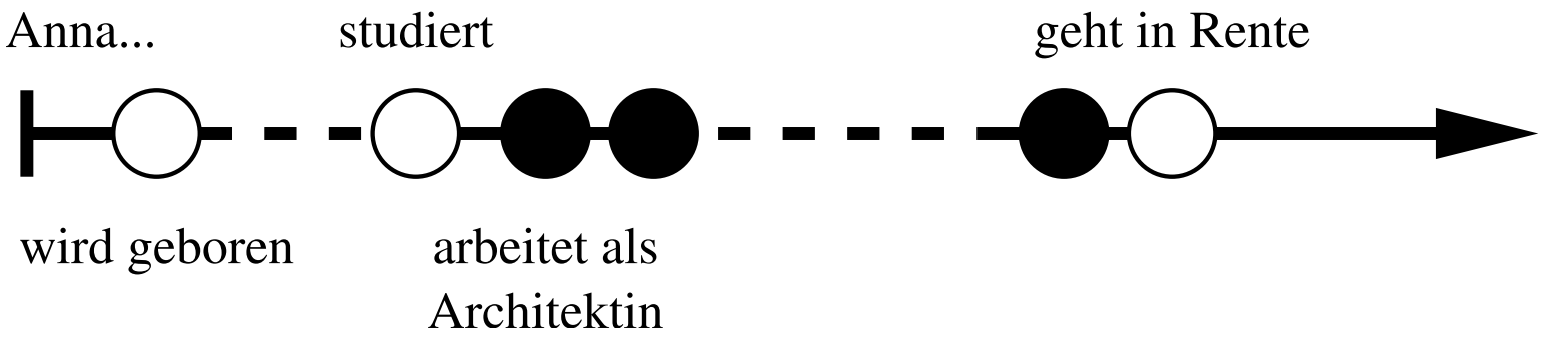
Beispiele: Sei $\mathcal{B}_1(A) = 1$ und $\mathcal{B}_1(B) = 0$. Dann gilt $\mathcal{B}_1 \not\models A \wedge B$ und $\mathcal{B}_1 \models \neg B$.

Sei $\mathcal{B}_2(A) = 1$ und $\mathcal{B}_2(B) = 1$. Dann gilt $\mathcal{B}_2 \models A \wedge B$ und $\mathcal{B}_2 \not\models \neg B$.

Temporale Logik (Zeitlogik)

Berücksichtigt, dass sich die Wahrheitswerte von Grundaussagen mit der Zeit ändern können (die “Welt” ändert sich).

Wahrheitswert von *A* im Laufe von Annas Leben:



Mögliche Aussagen:

Anna wird irgendwann in der Zukunft Architektin sein.

Anna ist solange Architektin, bis sie in Rente geht.

$\Longrightarrow$ Erweiterung der AL mit zeitlichen Modalitäten (irgendwann, solange . . . bis)

Vorschau

Linear-Zeit-Logik (LTL)

Betrachtet Formel mit zeitlichen Modalitäten

statt einzelner Belegungen betrachtet man (unendliche) Sequenzen von Belegungen

Model-Checking-Problem für LTL: Gegeben eine LTL-Formel und eine Sequenz von Belegungen, ist die Sequenz Modell der Formel?

Baum-Zeit-Logik (CTL)

Betrachtet (unendliche) *Bäume* von Sequenzen.

Interpretation: Nicht-Determinismus, es kann mehrere mögliche Entwicklungen geben.

Zusammenhang mit Verifikation

Zustandsraum eines Programms:

- Programmzeiger
- Variablen
- Stack, Heap, . . .

Mögliche Grundaussagen:

- “Variable x hat einen positiven Wert”
- “Der Programmzeiger befindet sich an der Sprungmarke ℓ.”

Gegeben eine Menge von Grundaussagen, liefert jeder Programmzustand eine Belegung!

Programme und Temporallogik

Linear-Zeit-Logik:

Jeder Programmablauf liefert eine Sequenz von Belegungen.

Interpretation des Programms: Menge der möglichen Sequenzen

Fragestellung: Erfüllen alle Sequenzen eine gegebene LTL-Formel?

Baum-Zeit-Logik:

Die Menge aller Programmabläufe lässt sich als Baum von Belegungen darstellen.

Interpretation des Programms: Baum mit Wurzel am Startzustand

Fragestellung: Erfüllt der Baum eine gegebene CTL-Formel?

Ergo: Verifikationsproblem $\widehat{=}$ Modellprüfungsproblem

Teil 2: Kripke-Strukturen

Überleitung

Das Model-Checking-Problem im Allgemeinen:

> Gegeben ein System S und eine temporallogische Eigenschaft ϕ,
> gilt $S \models \phi$ (d.h. "S ist ein Modell von ϕ" oder "S erfüllt ϕ")?

Ein Model-Checking-Prozess besteht daher aus drei Komponenten:

1. Modellierung (formale Darstellung des Systems)
2. Spezifikation (Formalisierung der Eigenschaft)
3. Verifikation (Anwendung eines Algorithmus)

Inhalt dieses Abschnitts: Modellierung

Modellierung

Als allgemeinstes Modell benutzen wir den Begriff des Transitionssystems:

$$\mathcal{T} = (S, \rightarrow, r)$$

S	$\widehat{=}$	Zustände (“states”), die das System annehmen kann (endliche oder unendliche Menge)
$\rightarrow \subseteq S \times S$	$\widehat{=}$	Transitionsrelation; beschreibt, welche Aktionen bzw. Zustandsübergänge möglich sind
$r \in S$	$\widehat{=}$	Anfangszustand (“root”)

Beispiel 1: Produzent/Konsument

(Pseudocode-)Programm mit Variablen und Nebenläufigkeit:

var *turn* $\{0,1\}$ **init** 0;
cobegin $\{P \parallel K\}$ **coend**

$P =$

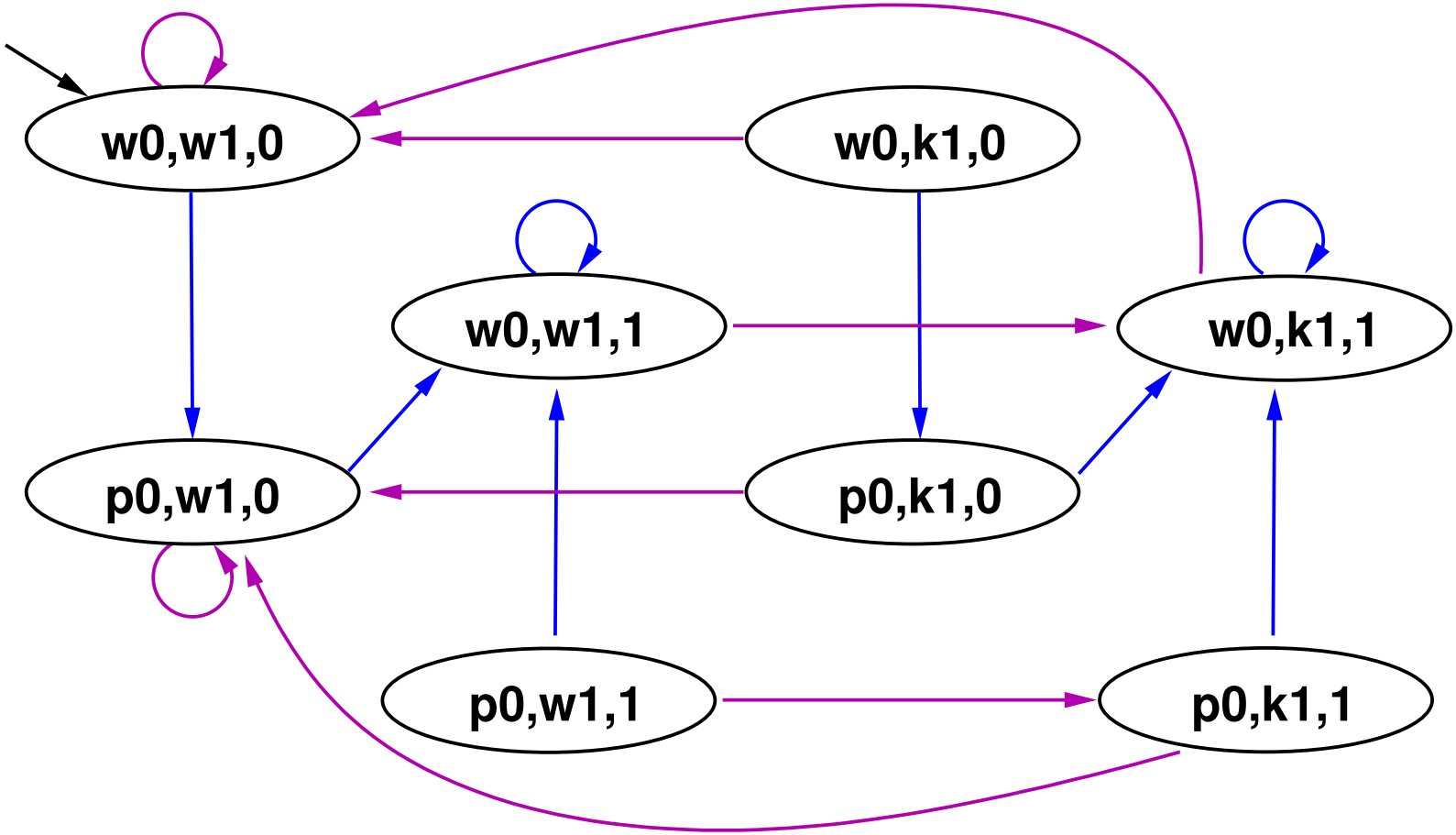
start;
while true do
  w_0: **wait** $(turn = 0)$;
  p_0: /* Produziere */
    $turn := 1$;
od;
end

$K =$

start;
while true do
  w_1: **wait** $(turn = 1)$;
  k_1: /* Konsumiere */
    $turn := 0$;
od;
end

Beispiel 1: Zugehöriges Transitionssystem

$S = \{w_0, p_0\} \times \{w_1, k_1\} \times \{0, 1\}$; Anfangszustand $(w_0, w_1, 0)$



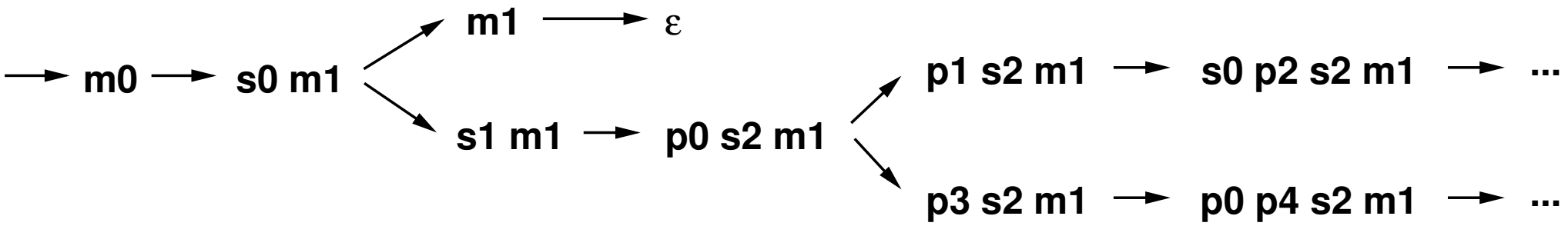
Beispiel 2: Rekursives Programm

procedure p;
p_0: **if** ? **then**
p_1: **call** s;
p_2: **if** ? **then call** p; **end if**;
else
p_3: **call** p;
end if
p_4: **return**

procedure s;
s_0: **if** ? **then return**; **end if**;
s_1: **call** p;
s_2: **return**;

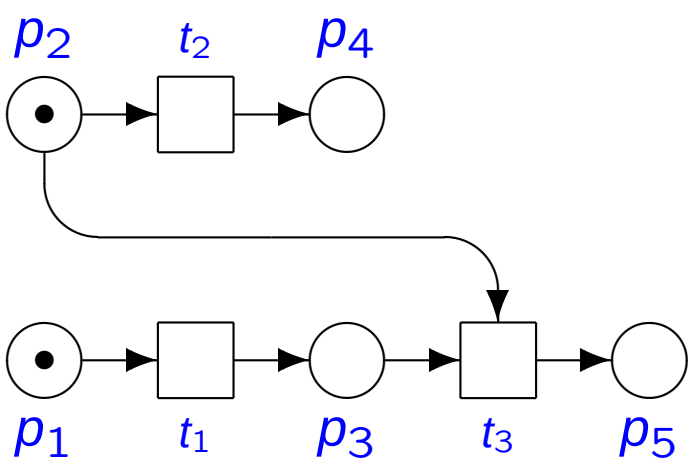
procedure $main$;
m_0: **call** s;
m_1: **return**;

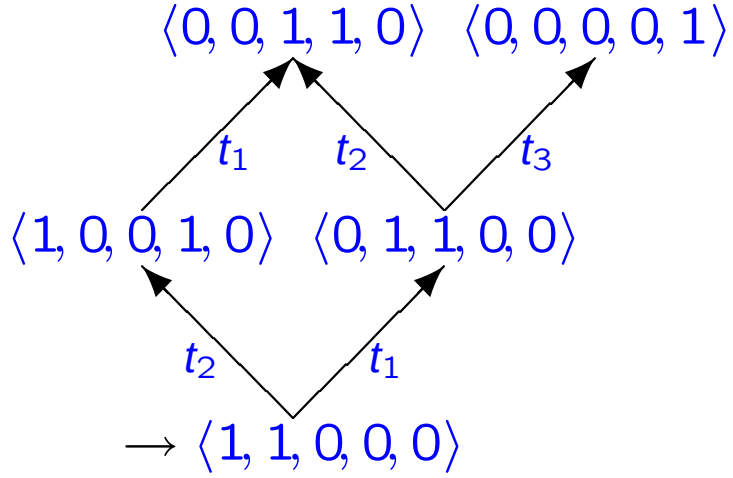
$S = \{p_0, \ldots, p_4, s_0, \ldots, s_2, m_0, m_1\}^*$, Anfangszustand m_0



Beispiel 3: Petri-Netz

Zustandsraum = Menge der Markierungen





Kripke-Strukturen

Idee: Transitionssysteme mit Belegungen anreichern:

$$\mathcal{K} = (S, \rightarrow, r, AP, \nu)$$

$(S, \rightarrow, r)$	$\widehat{=}$	zugrundeliegendes Transitionssystem
AP	$\widehat{=}$	Menge von Grundaussagen (“atomic propositions”)
$\nu : S \rightarrow 2^{AP}$	$\widehat{=}$	Interpretation der Grundaussagen (“valuation”)

Bemerkungen:

- 2^{AP} bezeichnet die *Potenzmenge* von AP.
- ν ordnet jedem Zustand eine *Belegung* zu; $p \in \nu(s)$ gdw. p in s gilt.

Beispiel für Kripke-Struktur

Transitionssystem $(S, \rightarrow, r)$ wie in Beispiel 1.

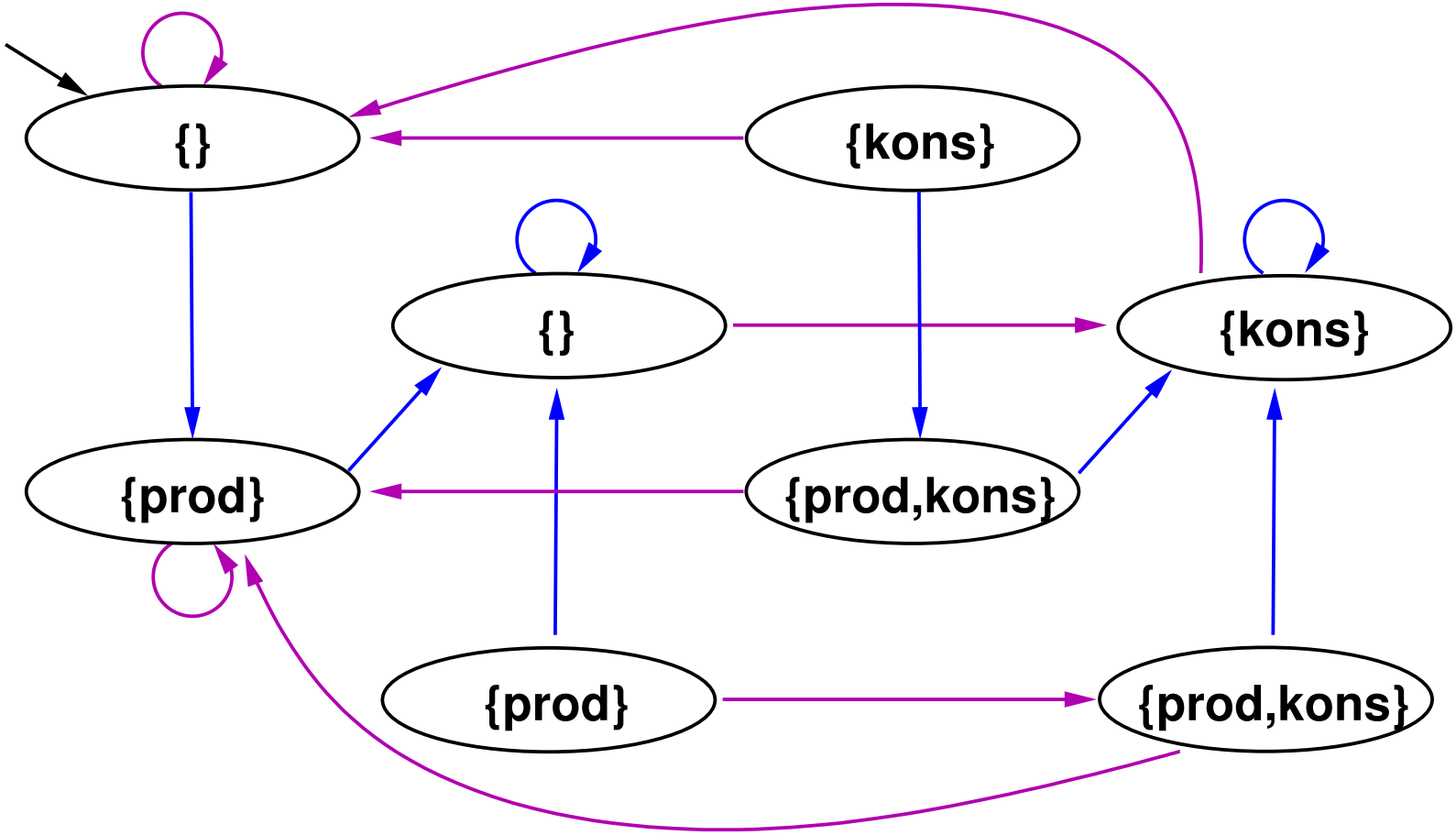
Von Interesse ist z.B., ob etwas konsumiert oder produziert wird.

Sei $AP = \{prod, kons\}$;

$\nu^{-1}(prod) = \{p_0\} \times \{w_1, k_1\} \times \{0, 1\}$;

$\nu^{-1}(kons) = \{w_0, p_0\} \times \{k_1\} \times \{0, 1\}$.

Die von Beispiel 1 induzierten Belegungen:

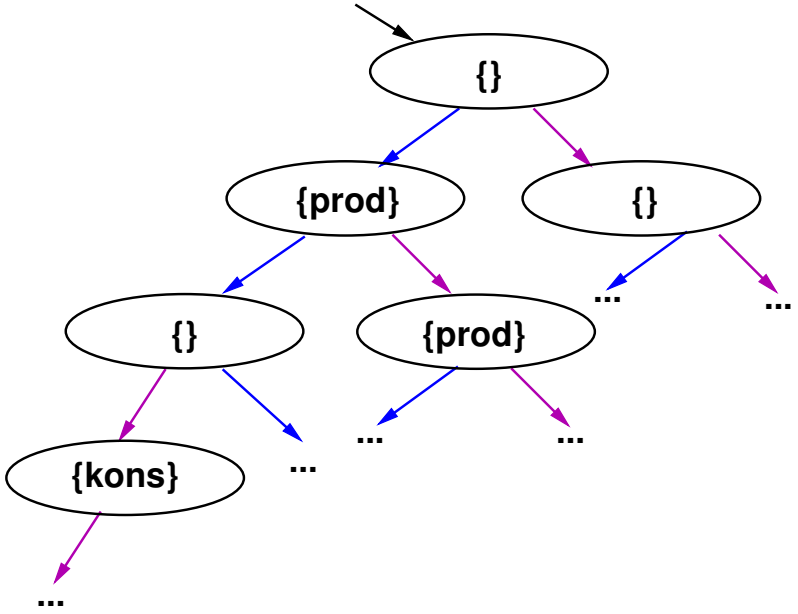


Belegungssequenzen bzw. -bäume

In Linear-Zeit-Logik werden wir die Menge der möglichen Belegungssequenzen betrachten,

z.B. $\emptyset\ \emptyset\ \{prod\}\ \emptyset\ \{kons\}\ \ldots$ oder $\emptyset\ \{prod\}\ \{prod\}\ \{prod\}\ \ldots$

In Baum-Zeit-Logik werden wir die “Entfaltung” der Kripkestruktur betrachten:



Beispiele für temporallogische Eigenschaften

“Es ist niemals möglich, dass *prod* und *kons* gleichzeitig gelten.”

Diese Eigenschaft kann in Beispiel 1 als erfüllt gelten. Es gibt zwar Zustände, in denen *prod* und *kons* zugleich gelten, diese sind aber vom Startzustand nicht erreichbar. Dies kann man durch Inspektion der Sequenzen bzw. des Baums überprüfen.

Eine Eigenschaft dieser Art nennt man auch Invariante.

“Immer, wenn etwas produziert wird, kann es danach konsumiert werden.”

Diese Eigenschaft kann als nicht erfüllt gelten, denn es gibt folgende Belegungssequenz: $\emptyset\ \{prod\}\ \emptyset\ \emptyset\ \emptyset\ldots$ Es wird also produziert, aber danach ist eine Endlosschleife möglich, so dass das System nicht in den “Konsum”-Zustand gelangt.

Eine Eigenschaft dieser Art nennt man auch Reaktionseigenschaft.

Fairness-Bedingungen

Die zweite Eigenschaft schlägt deshalb fehl, weil unsere Modellierung zu einfach war.

Die Modellierung lässt einen Ablauf zu, in dem nur noch der erste Prozess Schritte durchführt (in diesem Fall sogar “leere” Schritte).

Dieses Verhalten kann in nebenläufigen Systemen als “unrealistisch” gelten: Auch wenn ein einzelner Prozess mehrere Schritte hintereinander macht, wird ein “fairer” Scheduler beiden Prozessen immer wieder Rechenzeit geben.

Wir werden daher Abläufe ausschließen wollen, in denen einem Prozess auf Dauer Rechenzeit entzogen wird. Dies nennt man eine Fairness-Annahme.

Unter dieser Fairness-Annahme ist die zweite Bedingung erfüllt.

Implizite Graphen

Bemerkung: Ein Transitionssystem mit Zustandsraum S lässt sich auch als impliziter Graph darstellen:

$$(r, \mathrm{succ})$$

Dabei ist $r \in S$ wie bisher der Anfangszustand und $\mathrm{succ}\colon S \to 2^S$ die Nachfolgerfunktion ($\mathrm{succ}(s)$ ist die Menge der Nachfolger von s).

Mögliche Interpretation:

> Programmcode + Funktion, die den Code interpretiert (d.h. succ implementiert)

Vorteile:

Natürliche Darstellung für Tiefen- oder Breitensuche.

Für große Systeme ist schon die vollständige Konstruktion des Transitionssystems / der Kripke-Struktur sehr teuer (und enthält womöglich unerreichbare Zustände).

Darstellung als impliziter Graph oft kompakter.

Übergänge werden nicht explizit gespeichert.

Notation für Transitionssysteme

Wir schreiben $s \to t$, falls $(s, t) \in \to$ gilt.

Falls $s \to t$, ist s direkter Vorgänger von t und t direkter Nachfolger von s.

S^* bezeichnet die *endlichen*, S^ω die *unendlichen* Sequenzen (Wörter) über S.

$w = s_0 \dots s_n$ ist ein Pfad der Länge n, falls $s_i \to s_{i+1}$ für alle $0 \leq i < n$.

$\rho = s_0 s_1 \dots$ ist ein unendlicher Pfad, falls $s_i \to s_{i+1}$ für alle $i \geq 0$.

Notation für Transitionssysteme II

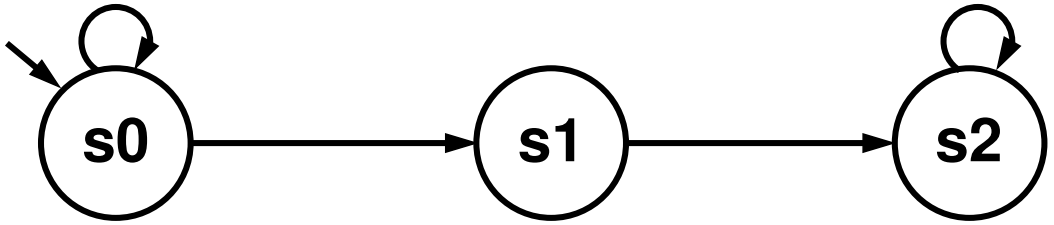
$\rho(i)$ sei das i-te Element von ρ und ρ^i der bei $\rho(i)$ beginnende Suffix.

$s \rightarrow^* t$, falls ein Pfad von s nach t existiert.

$s \rightarrow^+ t$, falls ein Pfad von s nach t mit positiver Länge existiert.

Falls $s \rightarrow^* t$, ist s Vorgänger von t und t Nachfolger von s.

Beispiel



$S = \{s_0, s_1, s_2\}$; Anfangszustand s_0

$s_0 \to s_0$ $s_0 \to s_1$ $s_1 \to s_2$ $s_2 \to s_2$

$s_0 s_1 s_2$ ist ein Pfad der Länge 2, d.h. $s_0 \to^* s_2$ und $s_0 \to^+ s_2$

$s_1 \to^* s_1$, aber $s_1 \not\to^+ s_1$

$\rho = s_0 s_0 s_1 s_2 s_2 s_2 \ldots$ ist ein unendlicher Pfad.

$\rho(2) = s_1$ $\rho^1 = s_0 s_1 s_2 s_2 s_2 \ldots$

Endliche und unendliche Kripke-Strukturen

Prinzipiell kann ein Transitionssystem/eine Kripke-Strukur unendlich viele Zustände haben. Einige Gründe (bei Softwaresystemen) sind:

- Daten: ganze Zahlen, Listen, Bäume, Zeigerstrukturen, ...
- Kontrolle: Prozeduren, dynamische Prozesserzeugung, ...
- Asynchrone Kommunikation: unbeschränkte FIFO-Kanäle
- Unbekannte Parameter: Zahl von Protokoll-Teilnehmern, ...
- Echtzeit: diskrete oder stetige Zeit

Viele (nicht alle!) dieser Merkmale führen zu Turing-mächtigen Berechnungsmodellen (und daher unentscheidbaren Verifikationsproblemen).

Beispiel: Rekursive Programme (Wiederholung)

procedure p;
p_0: **if** ? **then**
p_1: **call** s;
p_2: **if** ? **then call** p; **end if**;
else
p_3: **call** p;
end if
p_4: **return**

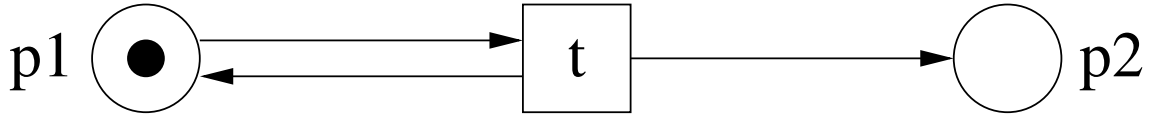
procedure s;
s_0: **if** ? **then return**; **end if**;
s_1: **call** p;
s_2: **return**;

procedure $main$;
m_0: **call** s;
m_1: **return**;

Der Zustandsraum dieses Beispiels ist unendlich (Stack!), CTL- und LTL-Modelchecking trotzdem entscheidbar.

Beispiel: Petri-Netze

Auch der Zustandsraum eines Petri-Netzes kann unendlich sein:



Die erreichbaren Zustände sind: $\langle 1,0\rangle,\ \langle 1,1\rangle,\ \langle 1,2\rangle,\ \ldots$

Erreichbarkeit und LTL entscheidbar, CTL unentscheidbar.

Endliche Kripke-Strukturen

In diesem Kurs beschränken wir uns auf die Betrachtung von Kripke-Strukturen mit endlichem Zustandsraum.

(Behandlung unendlicher Systeme $\Rightarrow$ Model-Checking II)

Endliche Systeme: z.B. Hardware-Systeme, Programme mit endlichen Datentypen (Boolesche Programme), bestimmte Kommunikationsprotokolle, ...

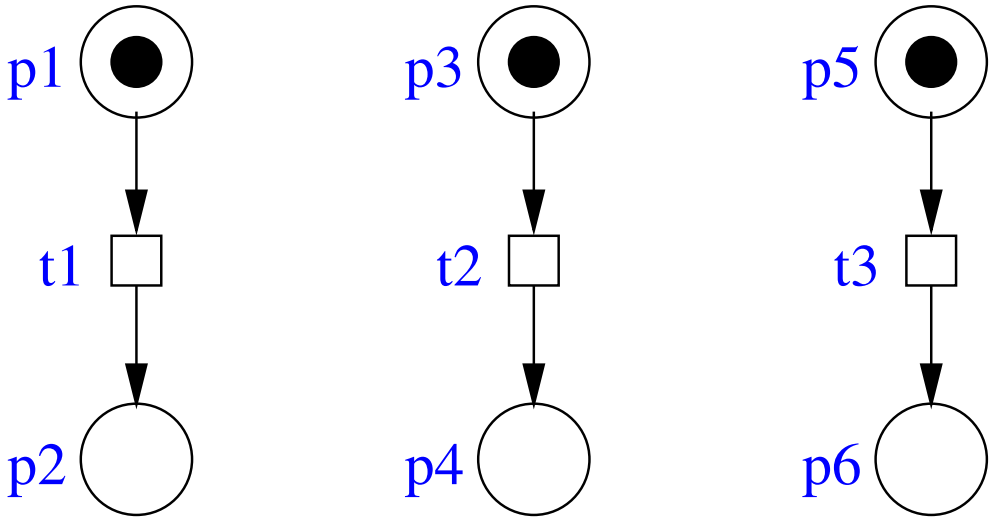
Endliche Modelle können ggfs. durch Abstraktion eines unendlichen Systems gewonnen werden.

Verbleibendes Problem: Zustandsexplosion, Modelle sind evtl. endlich, aber SEHR groß.

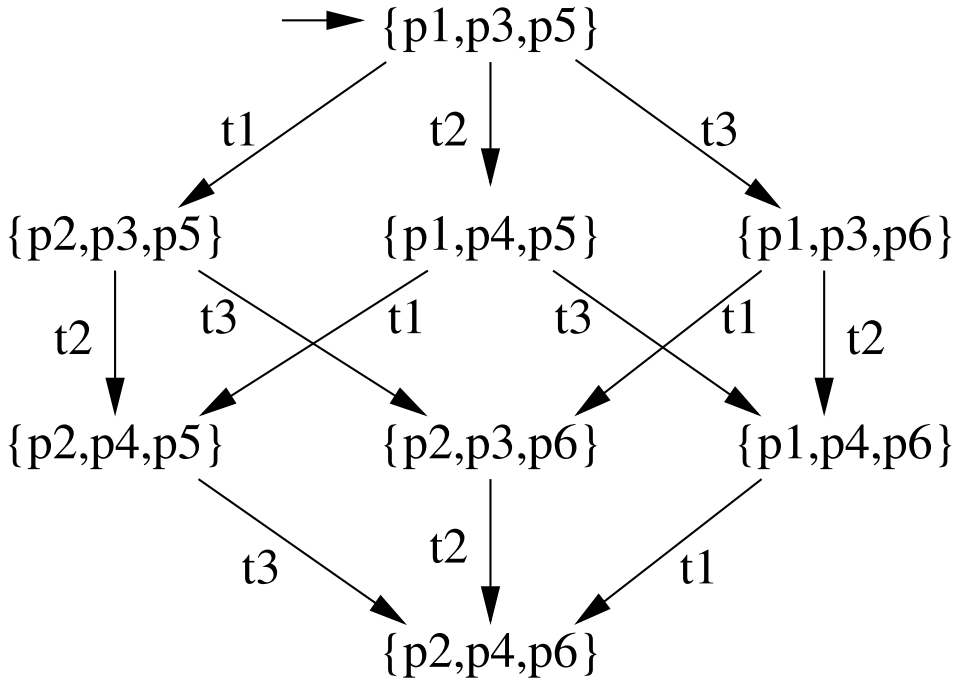
Gründe für Zustandsexplosion (1)

Ein häufiger Grund für Zustandsexplosion ist Nebenläufigkeit.

Beispiel: Betrachten wir das folgende Petri-Netz:

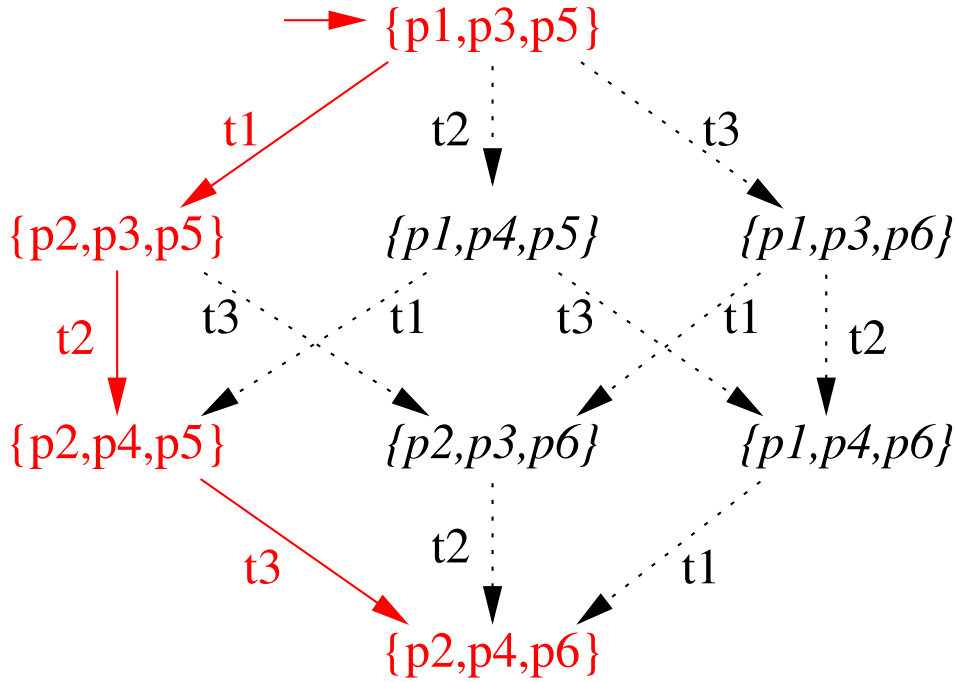


Der Erreichbarkeitsgraph hat $8 = 2^3$ Zustände und $6 = 3!$ mögliche Pfade.



Bei n Komponenten haben wir 2^n Zustände und $n!$ Pfade.

Alle Pfade führen zu $\{p_2, p_4, p_6\}$.



Idee: System reduzieren, indem nur ein Pfad betrachtet wird (aber Vorsicht!).

Gründe für Zustandsexplosion (2)

Ein zweiter häufiger Grund für Zustandsexplosion sind Daten.

z.B. Programme mit wenigen großen oder vielen kleinen Variablen.

Größe des Zustandsraums: 2 hoch Anzahl der Bits

Gegenmaßnahmen:

Abstraktion: “unwichtige” Daten ignorieren

Kompression: mit *Mengen* von Zuständen arbeiten, effiziente Datenstrukturen zur Darstellung und Manipulation von Mengen

Approximation: Zustandsraum über- oder unterapproximieren

Vorlesungsstoff (Approximation)

Temporale Logik: LTL, Büchi-Automaten, CTL, Fairness, ...

Verifikation:

- Grundlegende Algorithmen
- Halbordnungs-Reduktion
- Binary Decision Diagrams
- Abstraktion/Verfeinerung
- Bounded Model Checking
- ...

Tools: Spin, SMV, SAT-Solver, ...

Teil 3: Linear-Zeit-Logik

Vorbemerkungen

Linear-Zeit-Logik allgemein:

- jede Logik, die auf Sequenzen von Belegungen arbeitet
- Modell: Zeit schreitet diskret und linear voran, nur eine mögliche Zukunft
- Ursprung in Philosophie/Logik

Prominenter Vertreter: LTL

- seit Ende der 1970er: Anwendung in formaler Verifikation
- Spezifikation von Korrektheits-Eigenschaften

Belegungen

Seit AP eine Menge von Grundaussagen.

Belegung:

> Funktion, die jedem Element von AP wahr oder falsch zuordnet
>
> Teilmenge von AP, die genau die wahren Elemente enthält

Aus technischen Gründen werden wir die Teilmengen-Darstellung bevorzugen.

Beispiel:

> Sei $AP = \{p, q, r\}$. Die Teilmenge $\{p, q\} \subseteq AP$ entspricht der Belegung, die p und q wahr macht und r falsch macht.

Zur Erinnerung:

2^{AP} ist die Potenzmenge von AP, d.h. die Menge der Teilmengen von AP.

Sei Σ eine Menge, dann bezeichnet Σ^ω die Menge der *unendlichen* Sequenzen von Elementen aus Σ.

$(2^{AP})^\omega$ bezeichnet die Menge (unendlicher) Belegungssequenzen (von AP).

Sei $\sigma \in (2^{AP})^\omega$ eine solche Belegungssequenz.

Dann bezeichnet $\sigma(i)$ das i-te Element der Sequenz, d.h. $\sigma = \sigma(0)\sigma(1)\sigma(2)\ldots$.

σ^i bezeichnet den Suffix, der bei $\sigma(i)$ startet, d.h. $\sigma^i = \sigma(i)\sigma(i+1)\ldots$.

Syntax von LTL

Sei AP eine Menge von Grundaussagen.
Die Menge der LTL-Formeln über AP ist wie folgt:

Ist $p \in AP$, so ist p eine Formel.

Sind ϕ_1, ϕ_2 Formeln, so auch

$$\neg\phi_1, \qquad \phi_1 \vee \phi_2, \qquad \mathbf{X}\,\phi_1, \qquad \phi_1\,\mathbf{U}\,\phi_2$$

Bedeutung: $\mathbf{X} \mathrel{\widehat{=}}$ “next”, $\mathbf{U} \mathrel{\widehat{=}}$ “until”.

Bemerkungen

Dies ist eine minimale Syntax, die wir für Beweise o.ä. betrachten werden.

Weitere nützliche Operatoren lassen sich mithilfe der minimalen Syntax ausdrücken (kommt noch).

Vergleich Aussagenlogik (AL) mit LTL:

	AL	LTL
Syntax	Grundaussagen, logische Operatoren	+ temporale Operatoren
Auswertung auf. . .	Belegungen	Sequenzen von Belegungen
Semantik	Menge von Belegungen	Menge von Belegungssequenzen

Semantik von LTL

Sei ϕ eine LTL-Formel und σ eine Belegungssequenz.
Wir schreiben $\sigma \models \phi$ für "σ erfüllt ϕ."

$\sigma \models$	p	falls $p \in AP$ und $p \in \sigma(0)$
$\sigma \models$	$\neg\phi$	falls $\sigma \not\models \phi$
$\sigma \models$	$\phi_1 \vee \phi_2$	falls $\sigma \models \phi_1$ oder $\sigma \models \phi_2$
$\sigma \models$	$\mathbf{X}\,\phi$	falls $\sigma^1 \models \phi$
$\sigma \models$	$\phi_1\,\mathbf{U}\,\phi_2$	falls $\exists i : \left(\sigma^i \models \phi_2 \ \wedge \ \forall k < i : \sigma^k \models \phi_1\right)$

Semantik von ϕ: $\quad [\![\phi]\!] = \{\, \sigma \mid \sigma \models \phi \,\}$

Beispiele

Sei $AP = \{p, q, r\}$. Überlegen Sie, ob die Sequenz

$$\sigma = \{p\}\ \{q\}\ \{p\}^{\omega}$$

folgende Formeln erfüllt (d.h. die $\models$-Beziehung gilt):

$$p$$

$$q$$

$$\mathbf{X}\, q$$

$$\mathbf{X}\, \neg p$$

$$p\ \mathbf{U}\ q$$

$$q\ \mathbf{U}\ p$$

$$(p \vee q)\ \mathbf{U}\ r$$

Erweiterte Syntax

Folgende Abkürzungen werden uns oft nützlich sein:

$$\begin{aligned} \phi_1 \wedge \phi_2 &\equiv \neg(\neg\phi_1 \vee \neg\phi_2) \\ \phi_1 \rightarrow \phi_2 &\equiv \neg\phi_1 \vee \phi_2 \\ \mathbf{true} &\equiv a \vee \neg a \\ \mathbf{false} &\equiv \neg\mathbf{true} \end{aligned} \qquad \begin{aligned} \mathbf{F}\,\phi &\equiv \mathbf{true}\,\mathbf{U}\,\phi \\ \mathbf{G}\,\phi &\equiv \neg\,\mathbf{F}\,\neg\phi \\ \phi_1\,\mathbf{W}\,\phi_2 &\equiv (\phi_1\,\mathbf{U}\,\phi_2) \vee \mathbf{G}\,\phi_1 \\ \phi_1\,\mathbf{R}\,\phi_2 &\equiv \neg(\neg\phi_1\,\mathbf{U}\,\neg\phi_2) \end{aligned}$$

Bedeutung: $\mathbf{F} \mathrel{\widehat{=}}$ “finally” (irgendwann), $\mathbf{G} \mathrel{\widehat{=}}$ “globally” (immer),
$\mathbf{W} \mathrel{\widehat{=}}$ “weak until”, $\mathbf{R} \mathrel{\widehat{=}}$ “release”.

Einige Beispiel-Formeln

Erreichbarkeit: $\mathbf{G}\,\neg(cs_1 \wedge cs_2)$

cs_1 and cs_2 treten niemals gemeinsam auf.

Bemerkung: Eine entsprechende Belegung vorausgesetzt, drückt dies eine Mutex-Eigenschaft aus (“mutual exclusion”, gegenseitiger Ausschluss).

Sicherheit: $(\neg x)\,\mathbf{W}\,y$

x passiert nicht, bevor y einmal passiert ist.

Bemerkung: Es kann sein, dass y niemals auftritt, dann tritt auch x niemals auf.

Lebendigkeit: $(\neg x)\,\mathbf{U}\,y$

x passiert nicht, bevor y einmal passiert ist, und y passiert irgendwann.

Mehr Beispiele

$\mathbf{G}\,\mathbf{F}\,p$

p tritt unendlich oft auf.

$\mathbf{F}\,\mathbf{G}\,p$

Ab irgendeinem Zeitpunkt gilt *p* immer.

$\mathbf{G}(try_1 \rightarrow \mathbf{F}\,cs_1)$

Bei Mutex-Algorithmen: Wenn Prozess 1 versucht, in seinen kritischen Abschnitt einzutreten, wird ihm das auch irgendwann gelingen.

Tautologie, Äquivalenz

Aus der AL kennen wir Begriffe wie Tautologie, Äquivalenz etc, die sich analog auf LTL übertragen lassen.

Tautologie: Jede Formel ϕ mit $[\![\phi]\!] = (2^{AP})^\omega$ heißt Tautologie.

Unerfüllbarkeit: Jede Formel ϕ mit $[\![\phi]\!] = \emptyset$ heißt unerfüllbar.

Äquivalenz: Zwei Formeln ϕ_1, ϕ_2 heißen äquivalent, gdw. $[\![\phi_1]\!] = [\![\phi_2]\!]$.
Schreibweise: $\phi_1 \equiv \phi_2$

Äquivalenzen: Beziehungen zwischen Operatoren

$$
\begin{aligned}
\mathbf{X}(\phi_1 \vee \phi_2) &\equiv \mathbf{X}\,\phi_1 \vee \mathbf{X}\,\phi_2 \\
\mathbf{X}(\phi_1 \wedge \phi_2) &\equiv \mathbf{X}\,\phi_1 \wedge \mathbf{X}\,\phi_2 \\
\mathbf{X}\,\neg\phi &\equiv \neg\,\mathbf{X}\,\phi \\
\\
\mathbf{F}(\phi_1 \vee \phi_2) &\equiv \mathbf{F}\,\phi_1 \vee \mathbf{F}\,\phi_2 \\
\neg\,\mathbf{F}\,\phi &\equiv \mathbf{G}\,\neg\phi \\
\\
\mathbf{G}(\phi_1 \wedge \phi_2) &\equiv \mathbf{G}\,\phi_1 \wedge \mathbf{G}\,\phi_2 \\
\neg\,\mathbf{G}\,\phi &\equiv \mathbf{F}\,\neg\phi \\
\\
(\phi_1 \wedge \phi_2)\,\mathbf{U}\,\psi &\equiv (\phi_1\,\mathbf{U}\,\psi) \wedge (\phi_2\,\mathbf{U}\,\psi) \\
\phi\,\mathbf{U}\,(\psi_1 \vee \psi_2) &\equiv (\phi\,\mathbf{U}\,\psi_1) \vee (\phi_1\,\mathbf{U}\,\psi_2)
\end{aligned}
$$

Äquivalenzen: Idempotenz- und Rekursionsgesetze

$$\mathbf{F}\,\phi \equiv \mathbf{F}\,\mathbf{F}\,\phi$$

$$\mathbf{G}\,\phi \equiv \mathbf{G}\,\mathbf{G}\,\phi$$

$$\phi\,\mathbf{U}\,\psi \equiv \phi\,\mathbf{U}\,(\phi\,\mathbf{U}\,\psi)$$

$$\mathbf{F}\,\phi \equiv \phi \vee \mathbf{X}\,\mathbf{F}\,\phi$$

$$\mathbf{G}\,\phi \equiv \phi \wedge \mathbf{X}\,\mathbf{G}\,\phi$$

$$\phi\,\mathbf{U}\,\psi \equiv \psi \vee (\phi \wedge \mathbf{X}(\phi\,\mathbf{U}\,\psi))$$

$$\phi\,\mathbf{W}\,\psi \equiv \psi \vee (\phi \wedge \mathbf{X}(\phi\,\mathbf{W}\,\psi))$$

Interpretation von LTL auf Kripke-Strukturen

Sei $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ eine Kripke-Struktur.
Uns interessieren die Sequenzen von Belegungen, die $\mathcal{K}$ erzeugt.

Sei ρ in S^ω eine Ausführung von $\mathcal{K}$ (d.h. eine Sequenz von Zuständen, die bei r beginnt und $\rightarrow$ respektiert).

Wir ordnen ρ ein Bild $\nu(\rho)$ in $(2^{AP})^\omega$ zu; für alle $i \geq 0$ sei

$$\nu(\rho)(i) = \nu(\rho(i))$$

d.h. $\nu(\rho)$ enthält die Grundaussagen, die in den Zuständen von ρ gelten.

Bezeichnen wir mit $[\![\mathcal{K}]\!]$ die Menge sämtlicher solcher Sequenzen:

$$[\![\mathcal{K}]\!] = \{\, \nu(\rho) \mid \rho \text{ ist eine Ausführung von } \mathcal{K} \,\}$$

Das LTL-Model-Checking-Problem

Problem: Gegeben eine Kripke-Struktur $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ und eine LTL-Formel ϕ über AP, gilt $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$?

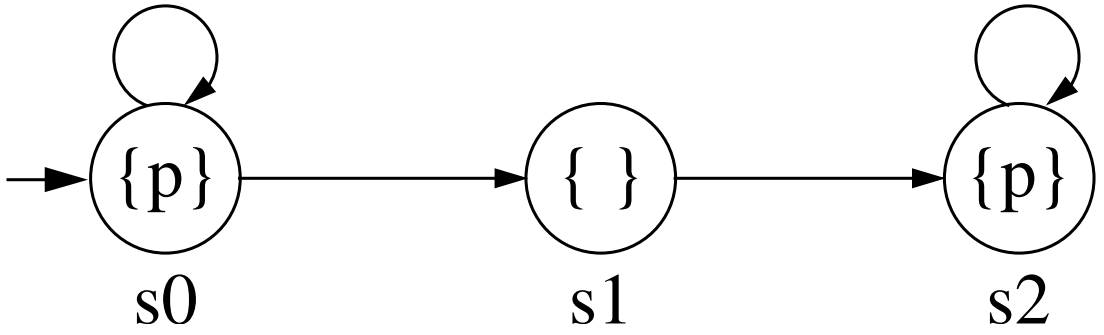
Definition: Falls $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$, schreiben wir $\mathcal{K} \models \phi$.

Interpretation: Jede Ausführung von $\mathcal{K}$ muss ϕ erfüllen, wenn $\mathcal{K} \models \phi$ gelten soll.

Bemerkung: Es kann $\mathcal{K} \not\models \phi$ und $\mathcal{K} \not\models \neg\phi$ gelten!

Beispiel

Betrachten wir die folgende Kripke Struktur $\mathcal{K}$ mit $AP = \{p\}$:



Es gibt zwei Klassen von Ausführungen in $\mathcal{K}$:

(i) Entweder bleibt das System auf ewig in s_0,

(ii) oder es geht irgendwann über s_1 nach s_2 und verbleibt dort.

Es gilt:

$\mathcal{K} \models \mathbf{F}\,\mathbf{G}\,p$, denn alle Abläufe bleiben irgendwann in einem p-Zustand.

$\mathcal{K} \not\models \mathbf{G}\,p$, denn Abläufe vom Typ (ii) enthalten einen Nicht-p-Zustand.

Endliche Sequenzen

Die Definition des Model-Checking-Problems betrachtet nur die *unendlichen* Ausführungen von $\mathcal{K}$.

Wenn $\mathcal{K}$ eine Verklemmung (deadlock) enthält, d.h. einen Zustand ohne Nachfolger, so werden alle (endlichen) Abläufe, die die Verklemmung erreichen, ignoriert.

Dies kann unerwartete Konsequenzen haben:

> Angenommen, ein Fehler führt dazu, dass $\mathcal{K}$ immer einen Deadlock erreicht.
>
> Dann ist $[\![\mathcal{K}]\!] = \emptyset$. Damit erfüllt $\mathcal{K}$ *jede* Formel!

Möglicher Umgang mit Verklemmungen

Deadlocks “per Design” abschaffen:

- Jeden Deadlock-Zustand mit einer Transition zu sich selbst versehen.
- Interpretation: System bleibt unendlich lange im Deadlock-Zustand

Gesonderte Behandlung von Deadlocks:

- Beim LTL-Model-Checking prüfen, ob $\mathcal{K}$ Deadlocks enthält.
- Falls ein Deadlock-Zustand einen Fehler darstellt, diesen eliminieren und mit modifiziertem Modell weitermachen.

Tool-Demonstration: Spin

Demonstration von Spin

Spin ist ein vielseitiger Model-Checker von Gerard Holzmann von Bell Labs.

Erhielt 2002 den ACM Software System Award

Web-Adresse: http://spinroot.com

Buch: Holzmann, The Spin Model Checker (in der Handbibliothek Esparza)

Modellierung mit Spin

Programmbeschreibung mit Promela (Protocol Meta Language)

Geeignet zur Beschreibung endlicher Systeme

Nebenläufige Prozesse, synchrone/asynchrone Kommunikation, Variablen, Datentypen

LTL-Model-Checking (mit Fairness und Reduktionstechniken)

Beispiel: Dekkers Mutex-Algorithmus

Modellierung eines Protokolls für wechselseitigen Ausschluss:

```
bit turn;
bool flag0, flag1;
bool crit0, crit1;

active proctype p0() {
...
}

active proctype p1() {
...
}
```

Dekker: Inhalt von Prozess p0

```
active proctype p0() {
again:  flag0 = true;
        do
        :: flag1 ->
                if
                :: turn == 1 ->
                        flag0 = false;
                        (turn != 1) -> flag0 = true;
                :: else -> skip;
                fi
        :: else -> break;
        od;

        crit0 = true; /* critical section */ crit0 = false;

        turn = 1; flag0 = false;
        goto again;
}
```

Prozess p1: wie p0, nur 0 mit 1 vertauscht

Was geht hier vor? (Promela-Syntax I)

Variablendeklaration:

```
bit turn;
bool flag0, flag1;
bool crit0, crit1;
```

`turn` kann die Werte `0` oder `1` annehmen.

`flag1` etc. können die Werte `true` oder `false` annehmen.

Anfangswerte: immer `0` oder `false`

Weitere Datentypen: `byte` oder selbstdefinierte

Promela-Syntax II

Prozessdeklaration:

```
active proctype p0() {
...
}
```

`proctype` definiert einen Prozess*typ*

`active` bedeutet, dass eine Instanz dieses Prozesstyps am Anfang aktiviert sein soll.

Auch möglich: Mehrere Prozessinstanzen aktivieren, z.B.

```
active [2] proctype mein_prozess() {
...
}
```

Promela-Syntax III

Sprungmarken / Zuweisungen / Sprünge

```
again:  flag0 = true;
...
        goto again;
```

leere Anweisung:

```
skip
```

Promela-Syntax IV

Schleife:

```
do
:: flag1 -> ...
:: else -> break;
od;
```

`flag1` und `else` sind “Wächter”

Ausführung verzweigt nicht-deterministisch in einen Zweig, dessen Wächter erfüllt ist.

`else`-Zweig kann nur genommen werden, wenn kein anderer Wächter erfüllt ist

`break` verlässt die `do`-Anweisung.

Promela-Syntax V

Verzweigung:

```
if
:: turn == 1 -> ...
:: else -> ...;
fi
```

Syntax und Semantik wie `do`, nur dass die Ausführung nicht wiederholt wird.

```
(turn != 1) -> ...
```

Bewachte Anweisung: Blockiert so lange, bis `turn` ungleich `1` ist.

Model-Checking mit Spin (Beispiel 1)

Im Dekker-Algorithmus sollten sich niemals beide Prozesse zugleich in ihren kritischen Abschnitten befinden. Dies lässt sich ausdrücken als:

$$\mathbf{G}\ \neg(\texttt{crit0} \wedge \texttt{crit1})$$

(wobei die Grundaussagen `crit0` und `crit1` bedeuten, dass die entsprechenden booleschen Variablen wahr sind.

(Dies ist wiederum eine Invariante bzw. eine Erreichbarkeits-Eigenschaft.)

Syntax in Spin: `[] !(crit0 && crit1)`

Model-Checking mit Spin (Beispiel 1)

Im Dekker-Algorithmus sollten sich niemals beide Prozesse zugleich in ihren kritischen Abschnitten befinden. Dies lässt sich ausdrücken als:

$$\mathbf{G}\ \neg(\texttt{crit0} \wedge \texttt{crit1})$$

(wobei die Grundaussagen `crit0` und `crit1` bedeuten, dass die entsprechenden booleschen Variablen wahr sind.

(Dies ist wiederum eine Invariante bzw. eine Erreichbarkeits-Eigenschaft.)

Syntax in Spin: `[] !(crit0 && crit1)`

Überprüfung mit Spin (Skript spinLTL):

Eigenschaft erfüllt!

Model-Checking mit Spin (Beispiel 2)

Im Dekker-Algorithmus sollte ein Prozess, der in einen kritischen Abschnitt eintreten will, dies auch irgendwann erreichen.

$$\mathbf{G}(\texttt{flag0} \rightarrow \mathbf{F}\,\texttt{crit0})$$

(“Reaktions-Eigenschaft”, analog für den anderen Prozess)

Syntax in Spin: `[] (flag0 -> <> crit0)`

Model-Checking mit Spin (Beispiel 2)

Im Dekker-Algorithmus sollte ein Prozess, der in einen kritischen Abschnitt eintreten will, dies auch irgendwann erreichen.

$$\mathbf{G}(\texttt{flag0} \rightarrow \mathbf{F}\,\texttt{crit0})$$

(“Reaktions-Eigenschaft”, analog für den anderen Prozess)

Syntax in Spin: `[] (flag0 -> <> crit0)`

Überprüfung mit Spin (Skript spinLTL):

Eigenschaft nicht erfüllt!

Fairness in Spin

Prozess 0 kann dann nicht in seinen kritischen Abschnitt eintreten, wenn Prozess 1 keine Rechenzeit mehr erhält, um `flag1` wieder auf `false` zu setzen.

Eine solche Ausführung ist “unfair”.

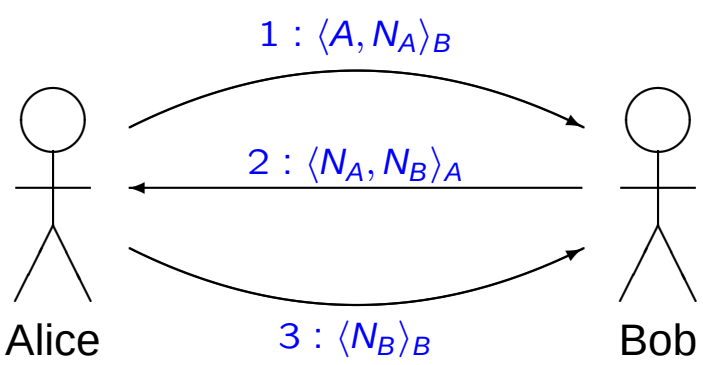
Fairness-Annahme: Betrachte nur diejenigen Ausführungen, in denen beide Prozesse unendlich oft (d.h. immer wieder) Rechenzeit bekommen.

Für den linken Teil der Eigenschaft gibt es einen speziellen Schalter in Spin (Skript spinFairLTL).

Größeres Beispiel: Needham-Schröder-Protokoll

Stephan Merz, Model Checking: A Tutorial Overview, 2001

Ziel des Protokolls: Alice und Bob versuchen, ein “Geheimnis” zu vereinbaren.



- Geheimnis repräsentiert durch Nonces $\langle N_A, N_B \rangle$
- Nachrichten können abgefangen werden
- Annahme: sicheres Public-Key-Verfahren

Ist das Protokoll sicher?

Protokoll-Analyse mit Spin

Darstellung als endliches System

endliche Anzahl von Teilnehmern	Alice, Bob, Intruder
endliche Modelle der Teilnehmer	– Alice und Bob versuchen, nur eine Verbindung aufzubauen – eine (symbolische) Nonce pro Teilnehmer – Eindringling kann eine Nachricht zwischenspeichern
einfaches Netz-Modell	– gemeinsames Kommunikations-Medium – Nachrichten als Tupel $\langle empfaenger, daten \rangle$
Verschlüsselung wird simuliert	Vergleich der Schlüssel statt Berechnung

Einschub: Promela-Syntax

Deklaration eigener Aufzählungstypen:

```
mtype = {red,green,blue};
mtype x;
```

Die erste Zeile erzeugte eine Menge symbolischer Konstanten.

Die zweite Zeile deklariert `x` als eine Variable, die die Werte `0` (uninitialisiert), `red`, `green`, `blue` annehmen kann.

Deklaration eines Records:

```
typedef meintyp { bit b; mtype m; }
```

Einschub: Promela-Syntax

Kanäle:

```
chan c = [3] of mtype;
```

`c` ist der Name des Kanals

Die Zahl in eckigen Klammern gibt die Kapazität an; `[0]` bedeutet synchrone Kommunikation.

Hinter `of` steht der Typ der Daten, der verschickt werden kann.

Schreiben: `c!red;`

Lesen: `mtype color; c?color;`

Promela-Modell: Vereinbarungen

```
#define success          (statusA == ok && statusB == ok)
#define aliceBob         partnerA == bob
#define bobAlice         partnerB == alice
```

Definition eines Aufzählungstyps

```
mtype = {msg1, msg2, msg3, alice, bob, intruder,
         nonceA, nonceB, nonceI, keyA, keyB, keyI, ok};

typedef Crypt { mtype key, d1, d2; }
```

Datentyp (Record) für Nachricht

```
chan network = [0] of {mtype, /* Nachrichten-Nr. */
                       mtype, /* Empfänger */
                       Crypt};
```

Gemeinsamer Kanal

```
mtype partnerA, partnerB;
mtype statusA, statusB;

bool knowNA, knowNB;
```

Status von Alice und Bob

Dem Eindringling bekannte Nonces

Promela-Modell für Alice

```
active proctype Alice() {
  if       Partner auswählen
  :: partnerA = bob; partner_key = keyB;
  :: partnerA = intruder; partner_key = keyI;
  fi;
           Erste Nachricht schicken
  network ! msg1, partnerA, ⟨partner_key, alice, nonceA⟩;

           Auf Antwort (zweite Nachricht) warten
  network ? msg2, alice, data;
           Schlüssel und Nonce überprüfen
  (data.key == keyA) && (data.d1 == nonceA);
  partner_nonce = data.d2;
           Dritte Nachricht schicken
  network ! msg3, partnerA, ⟨partner_key, partner_nonce⟩;
  statusA = ok;
}
```

ähnliches Modell für Bob

Promela-Modell für Eindringling (1)

```
active proctype Intruder() {
  do          Nachricht empfangen/abfangen
  :: network ? msg, _, data ->
     if       Nachricht merken, falls nicht entschlüsselbar
     :: intercepted = data;
     :: skip;
     fi;
     if       Nachricht auswerten, falls für Eindringling bestimmt
     :: (data.key == keyI) ->
        if
        :: (data.d1 == nonceA || data.d2 == nonceA) -> knowNA = true;
        :: else -> skip;
        fi;
        if
        :: (data.d1 == nonceB || data.d2 == nonceB) -> knowNB = true;
        :: else -> skip;
        fi;
     :: else -> skip;
     fi;
  :: ...
```

Promela-Modell für Eindringling (2)

```
  :: ...
  :: if                    Erste Nachricht an Bob schicken
     :: network ! msg1, bob, intercepted;   empfangene Nachricht wiederholen
     :: data.key = keyB;                    Nachricht zusammenstellen
        if   Der Eindringling kann sich als Alice oder sich selbst ausgeben
        :: data.d1 = alice;
        :: data.d1 = intruder;
        fi;
        if   ... und bekannte Nonces verwenden
        :: knowsNA -> data.d2 = nonceA;
        :: knowsNB -> data.d2 = nonceB;
        :: data.d2 = nonceI;
        fi;
        network ! msg1, bob, data;
     fi;
  :: ...      ähnlicher Code für die anderen Nachrichten
  od;
}
```

Protokoll-Analyse mit Spin

Gewünschte Eigenschaften:

$$\mathrm{G}\Big(\mathit{statusA} = \mathit{ok} \wedge \mathit{statusB} = \mathit{ok} \;\Rightarrow\; (\mathit{partnerA} = \mathit{bob} \Leftrightarrow \mathit{partnerB} = \mathit{alice})\Big)$$

$$\mathrm{G}(\mathit{statusA} = \mathit{ok} \wedge \mathit{partnerA} = \mathit{agentB} \;\Rightarrow \neg \mathit{knowsNA})$$

$$\mathrm{G}(\mathit{statusB} = \mathit{ok} \wedge \mathit{partnerB} = \mathit{agentA} \;\Rightarrow \neg \mathit{knowsNB})$$

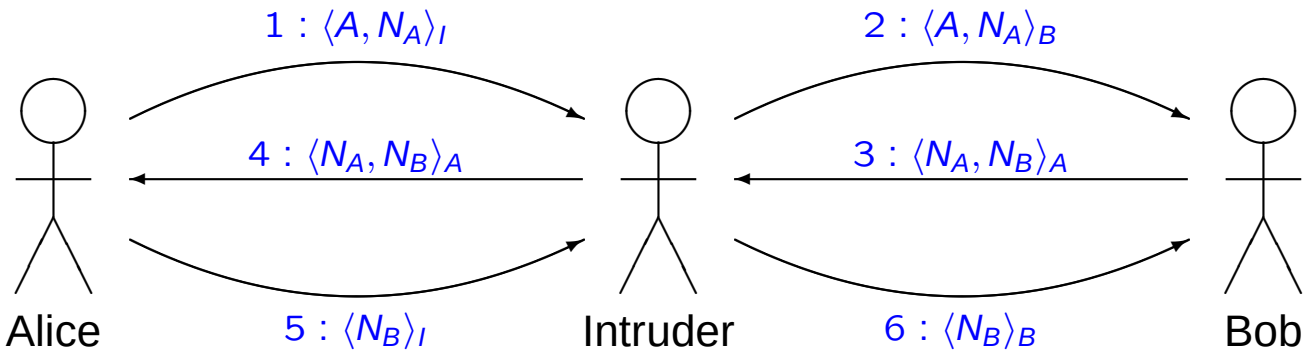
Spin-Ausgabe:

– Eigenschaft verletzt!

Der Fehler im Protokoll

Alice hat eine Verbindung mit Intruder.

Bob glaubt (fälschlicherweise), mit Alice zu reden.



Fehler wurde erst nach 18 Jahren entdeckt!

[Needham, Schröder (1978), Lowe (1996)]

Teil 4: Büchi-Automaten

Vorschau

Model-Checking-Problem: $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$ – wie beantworten wir dies automatisch?

(Historisch) erster Ansatz: Übersetze $\mathcal{K}$ in eine LTL-Formel $\psi_{\mathcal{K}}$, prüfe, ob $\psi_{\mathcal{K}} \rightarrow \phi$ eine Tautologie ist. Problem: Sehr aufwändig.

Sprach-/automatentheoretischer Ansatz: $[\![\mathcal{K}]\!]$ und $[\![\phi]\!]$ sind Sprachen (über unendlichen Wörtern).

> Finde eine geeignete Klasse von Automaten, die diese Sprachen akzeptiert.
>
> Implementiere geeignete Operationen auf diesen Automaten, um das Problem zu lösen.

Dies ist der Ansatz, dem wir folgen werden.

Büchi-Automaten

Ein Büchi-Automat ist ein Tupel

$$\mathcal{B} = (\Sigma, S, s_0, \Delta, F)$$

mit:

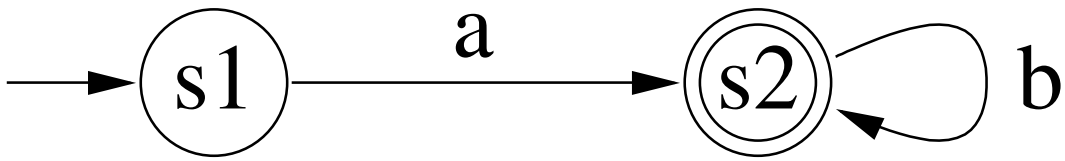
Σ	endliches Alphabet,
S	endliche Menge von Zuständen,
$s_0 \in S$	Anfangszustand,
$\Delta \subseteq S \times \Sigma \times S$	Transitionsrelation,
$F \subseteq S$	Akzeptanzzustände.

Bemerkungen:

- Definition und graphische Darstellung genau wie bei endlichen Automaten.
- Büchi-Automaten arbeiten aber mit *unendlichen Wörtern*, andere Akzeptanzbedingung.

Beispiel

Graphische Darstellung eines Büchi-Automaten:



Die Bestandteile dieses Automaten sind $(\Sigma, S, s_1, \Delta, F)$, wobei:

- $\Sigma = \{a, b\}$ (die Symbole auf den Kanten)
- $S = \{s_1, s_2\}$ (die Kreise)
- s_1 (mit Pfeil markiert)
- $\Delta = \{(s_1, a, s_2), (s_2, b, s_2)\}$ (die Kanten)
- $F = \{s_2\}$ (mit doppelter Umrandung)

Sprache eines Büchi-Automaten

Sei $\mathcal{B} = (\Sigma, S, s_0, \Delta, F)$ ein Büchi-Automat.

Ein Lauf von $\mathcal{B}$ über einem unendlichen Wort $\sigma \in \Sigma^\omega$ ist eine unendliche Folge von Zuständen $\rho \in S^\omega$ mit $\rho(0) = s_0$ und $(\rho(i), \sigma(i), \rho(i+1)) \in \Delta$ für $i \geq 0$.

Ein Lauf ρ ist akzeptierend gdw. für unendlich viele i gilt: $\rho(i) \in F$.

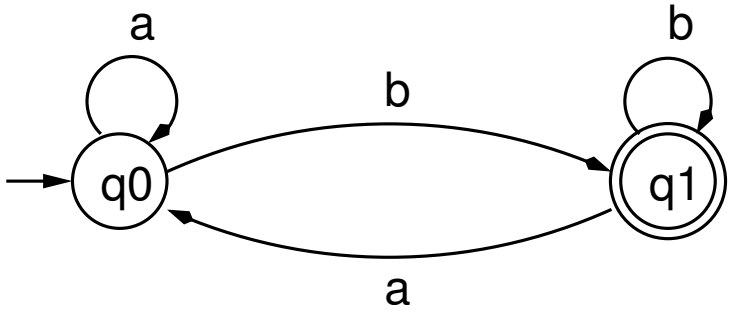
D.h., ρ besucht unendlich oft akzeptierende Zustände.
(Schubfachprinzip: Irgendein akzeptierender Zustand wird unendlich oft besucht.)

$\sigma \in \Sigma^\omega$ wird von $\mathcal{B}$ akzeptiert gdw. ein akzeptierender Lauf über σ in $\mathcal{B}$ existiert.

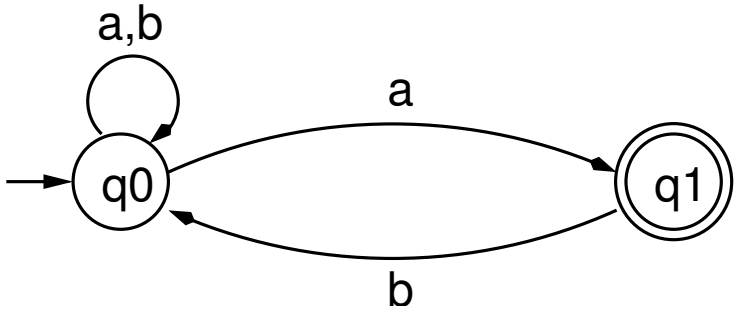
Die Sprache von $\mathcal{B}$, geschrieben $\mathcal{L}(\mathcal{B})$, ist die Menge der von $\mathcal{B}$ akzeptierten Wörter.

Büchi-Automaten: Beispiele

“unendlich oft b”



“unendlich oft ab”



Einschub: Büchi-Automaten und LTL

Sei AP eine Menge von Grundaussagen.

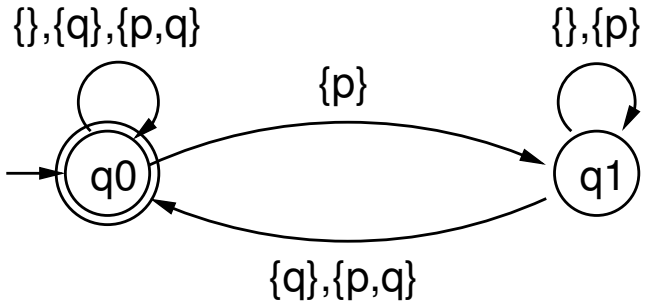
Ein Automat mit Alphabet 2^{AP} akzeptiert eine Menge von Belegungssequenzen.

Behauptung: Für jede LTL-Formel ϕ gibt es einen Büchi-Automaten $\mathcal{B}$, so dass $\mathcal{L}(\mathcal{B}) = [\![\phi]\!]$ gilt.

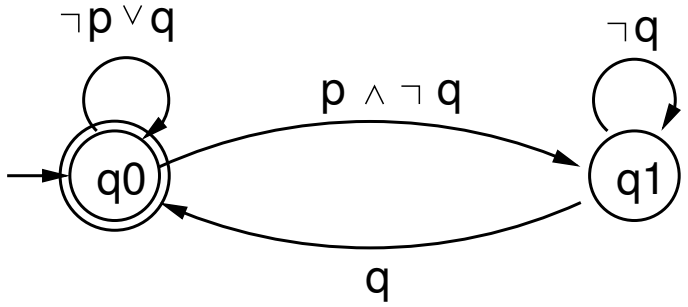
(Diese Behauptung werden wir später noch beweisen.)

Beispiele: $\mathbf{F}\,p$, $\mathbf{G}\,p$, $\mathbf{G}\,\mathbf{F}\,p$, $\mathbf{G}(p \rightarrow \mathbf{F}\,q)$, $\mathbf{F}\,\mathbf{G}\,p$

Beispiel-Automat für $G(p \rightarrow \mathbf{F}\, q)$, mit $AP = \{p, q\}$.



Alternativ können wir Kanten mit Formeln der AL beschriften; eine Formel F steht dann für alle Elemente von $[\![F]\!]$. Hier:



Operationen auf Büchi-Automaten

Die von Büchi-Automaten akzeptierten Sprachen werden auch ω-reguläre Sprachen genannt.

Wie die regulären Sprachen sind auch ω-reguläre Sprachen unter den üblichen boolschen Operationen abgeschlossen.

D.h., sind $\mathcal{L}_1$ und $\mathcal{L}_2$ ω-regulär, so auch

$$\mathcal{L}_1 \cup \mathcal{L}_2, \qquad \mathcal{L}_1 \cap \mathcal{L}_2, \qquad \mathcal{L}_1^c.$$

Im folgenden betrachten wir Operationen, die aus Büchi-Automaten für $\mathcal{L}_1$ und $\mathcal{L}_2$ Automaten für Vereinigung und Schnitt produzieren.

In den folgenden Folien sei $\mathcal{B}_1 = (\Sigma, S, s_0, \Delta_1, F)$ und $\mathcal{B}_2 = (\Sigma, T, t_0, \Delta_2, G)$. (Annahme: $S \cap T = \emptyset$.)

Vereinigung

$\mathcal{B}_1$ und $\mathcal{B}_2$ “nebeneinander stellen” + neuen Anfangszustand.

D.h., der Automat $\mathcal{B} = (\Sigma,\ S \cup T \cup \{u\},\ u,\ \Delta_1 \cup \Delta_2 \cup \Delta_u,\ F \cup G)$
akzeptiert $\mathcal{L}(\mathcal{B}_1) \cup \mathcal{L}(\mathcal{B}_2)$, wobei

u ein “frischer” Zustand ist ($u \notin S \cup T$);

$\Delta_u = \{\, (u, \sigma, s) \mid (s_0, \sigma, s) \in \Delta_1 \,\} \cup \{\, (u, \sigma, t) \mid (t_0, \sigma, t) \in \Delta_2 \,\}$.

Schnitt-Konstruktion I (Spezialfall)

Wir betrachten zunächst den Fall, wo alle Zustände in $\mathcal{B}_2$ akzeptierend sind, d.h. $G = T$.

Idee: Produziere den Kreuzprodukt-Automaten (wie bei endlichen Automaten), prüfe, ob F unendlich oft besucht wird.

Sei $\mathcal{B} = (\Sigma,\ S \times T,\ \langle s_0, t_0 \rangle,\ \Delta,\ F \times T)$, wobei

$$\Delta = \{\, (\langle s, t \rangle, a, \langle s', t' \rangle) \mid a \in \Sigma,\ (s, a, s') \in \Delta_1,\ (t, a, t') \in \Delta_2 \,\}.$$

Dann gilt: $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}_1) \cap \mathcal{L}(\mathcal{B}_2)$.

Schnitt-Konstruktion II (allgemeiner Fall)

Prinzip: Wir konstruieren erneut einen Kreuzprodukt-Automaten $\mathcal{B}$.

Problem: Die Akzeptanzbedingung von $\mathcal{B}$ muss prüfen, ob *beide* Akzeptanzmengen unendlich oft besucht werden.

Idee: Wir erzeugen zwei Kopien des Kreuzprodukts.

- In der ersten Kopie warten wir auf einen Zustand aus F.
- In der zweiten Kopie warten wir auf einen Zustand aus G.
- Nach jedem gesuchten Zustand wechseln wir in die andere Kopie.

Die Akzeptanzbedingung stellt sicher, dass wir unendlich oft zwischen den Kopien hin- und herwechseln.

Sei $\mathcal{B} = (\Sigma, U, u, \Delta, H)$, wobei

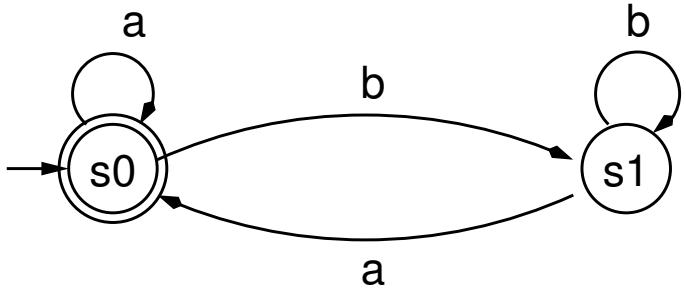
$$U = S \times T \times \{1,2\}, \quad u = \langle s_0, t_0, 1\rangle, \quad H = F \times T \times \{1\}$$

$$(\langle s,t,1\rangle, a, \langle s',t',1\rangle) \in \Delta \text{ gdw. } (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2,\ s \notin F$$
$$(\langle s,t,1\rangle, a, \langle s',t',2\rangle) \in \Delta \text{ gdw. } (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2,\ s \in F$$
$$(\langle s,t,2\rangle, a, \langle s',t',2\rangle) \in \Delta \text{ gdw. } (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2,\ t \notin G$$
$$(\langle s,t,2\rangle, a, \langle s',t',1\rangle) \in \Delta \text{ gdw. } (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2,\ t \in G$$

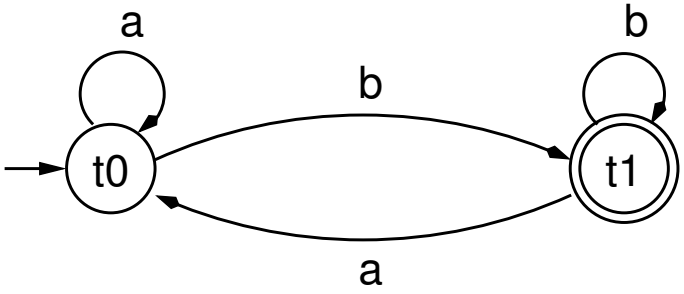
Bemerkungen:

- Der Automat startet in der ersten Kopie.
- Die Wahl der Akzeptanzzustände ist etwas beliebig, $S \times G \times \{2\}$ wäre auch möglich.
- Die Konstruktion kann auf den Schnitt von n Automaten verallgemeinert werden.

Schnitt: Beispiel

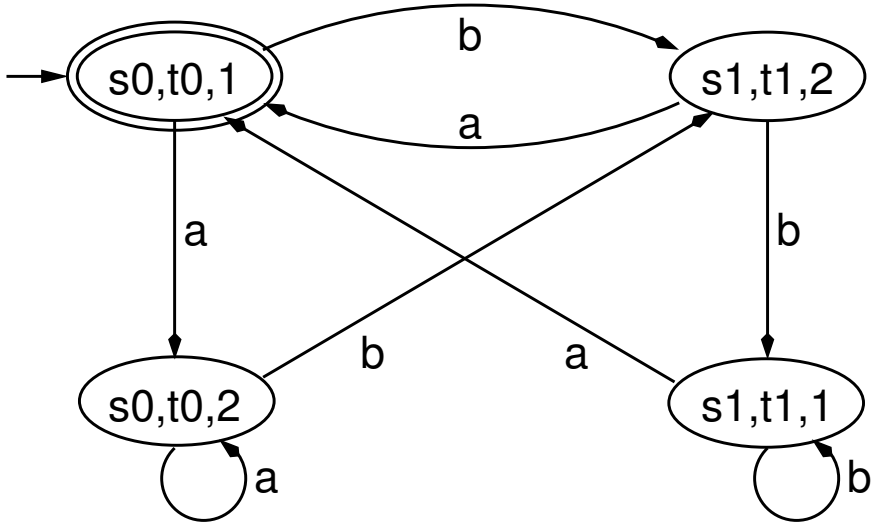


B1



B2

B1 x B2



Komplement

Gegeben $\mathcal{B}_1$, konstruiere $\mathcal{B}$ mit $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}_1)^c$.

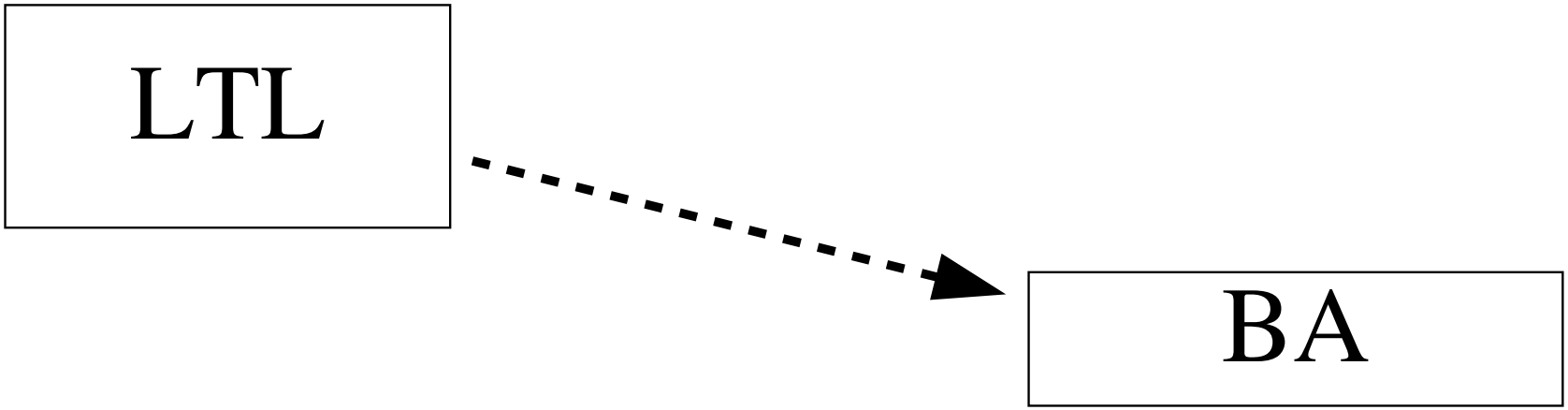
Geht auch, ist aber kompliziert.
(Und wir werden es für diesen Kurs nicht brauchen.)

Weiterführende Literatur:

Wolfgang Thomas, *Automata on Infinite Objects*,
Chapter 4 in *Handbook of Theoretical Computer Science*,

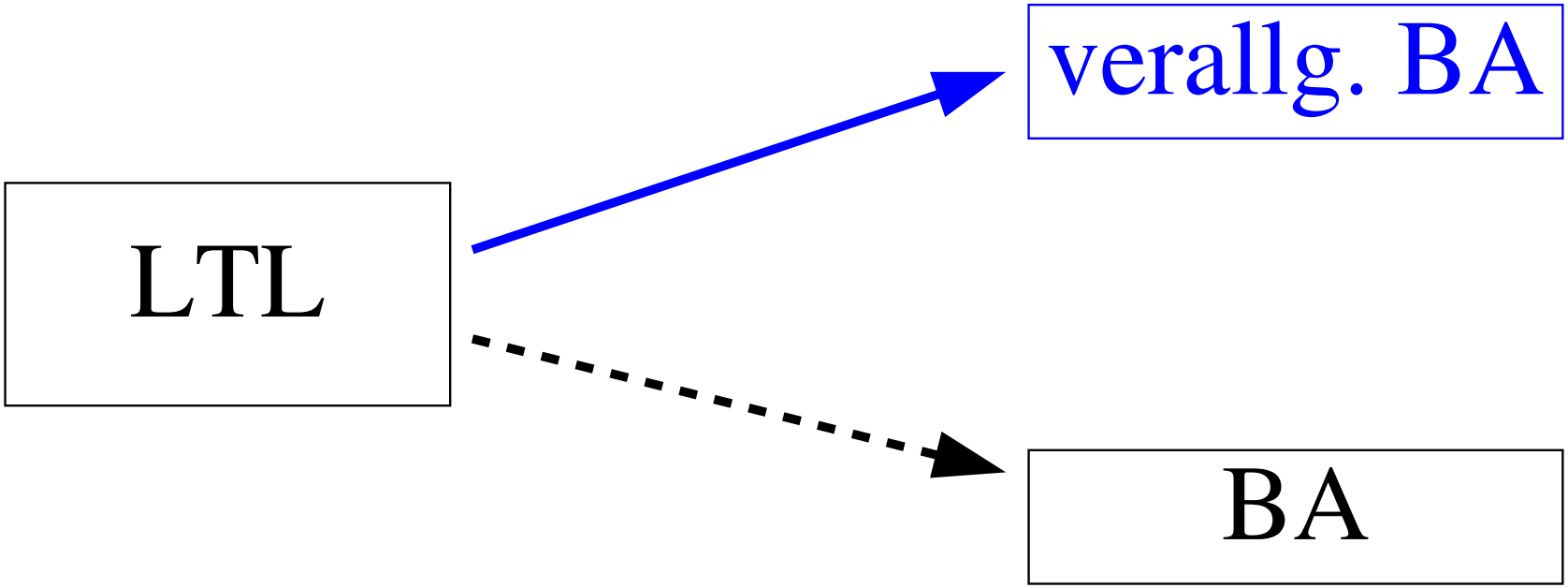
Igor Walukiewicz, Skript zu *Automata and Logic*, Kapitel 3,
`www.labri.fr/Perso/~igw/Papers/igw-eefss01.ps`

Programmvorschau



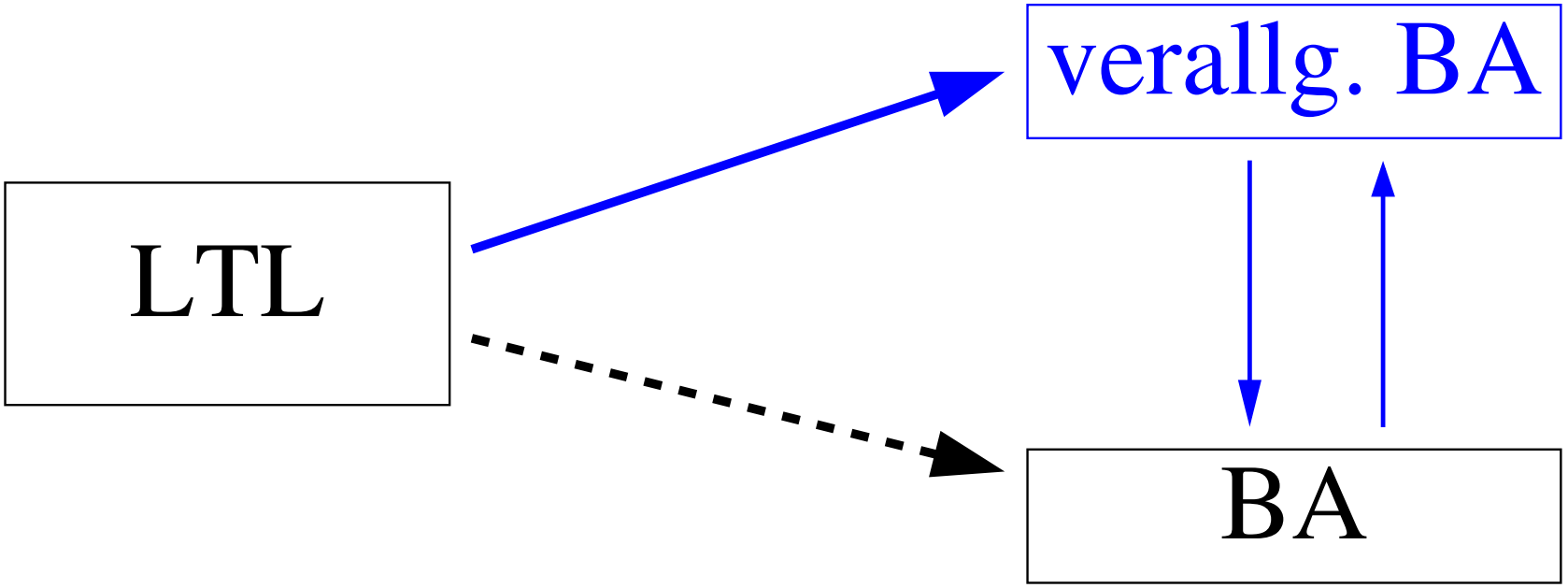
Gewünscht: Übersetzung von LTL-Formeln in Büchi-Automaten.

Programmvorschau



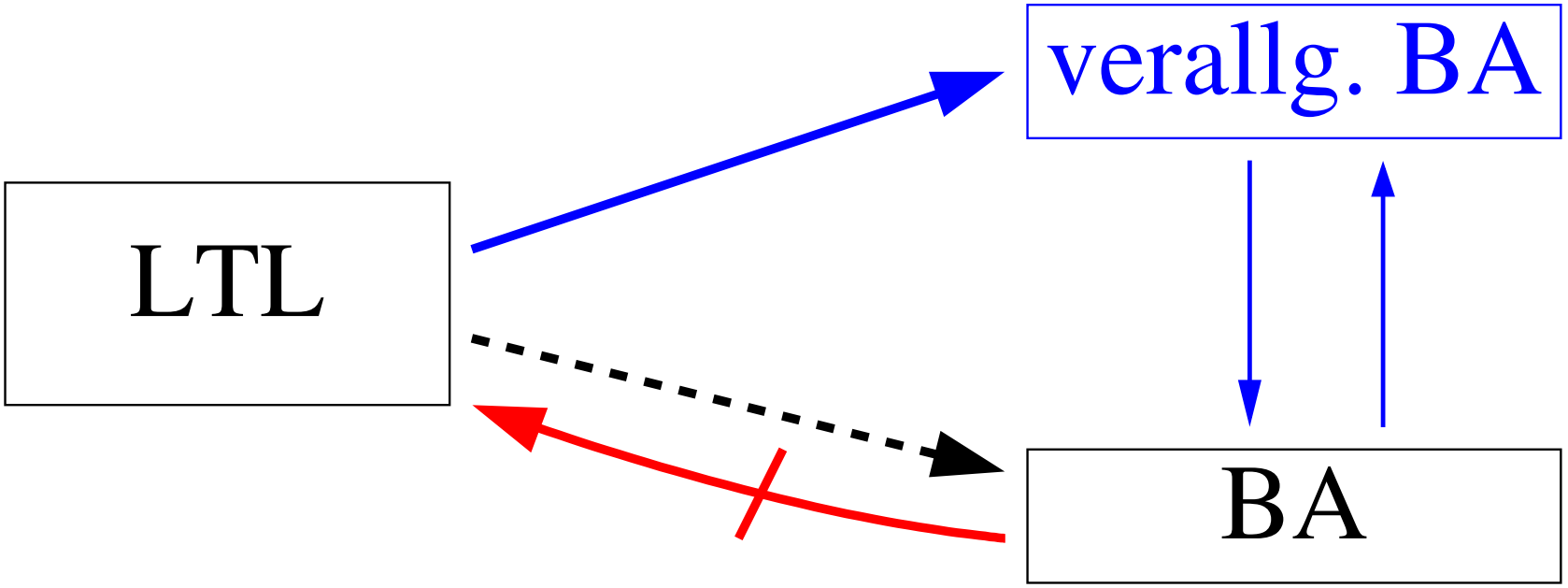
Umweg: Übersetzung in sogenannte verallgemeinerte Büchi-Automaten (VBA).

Programmvorschau



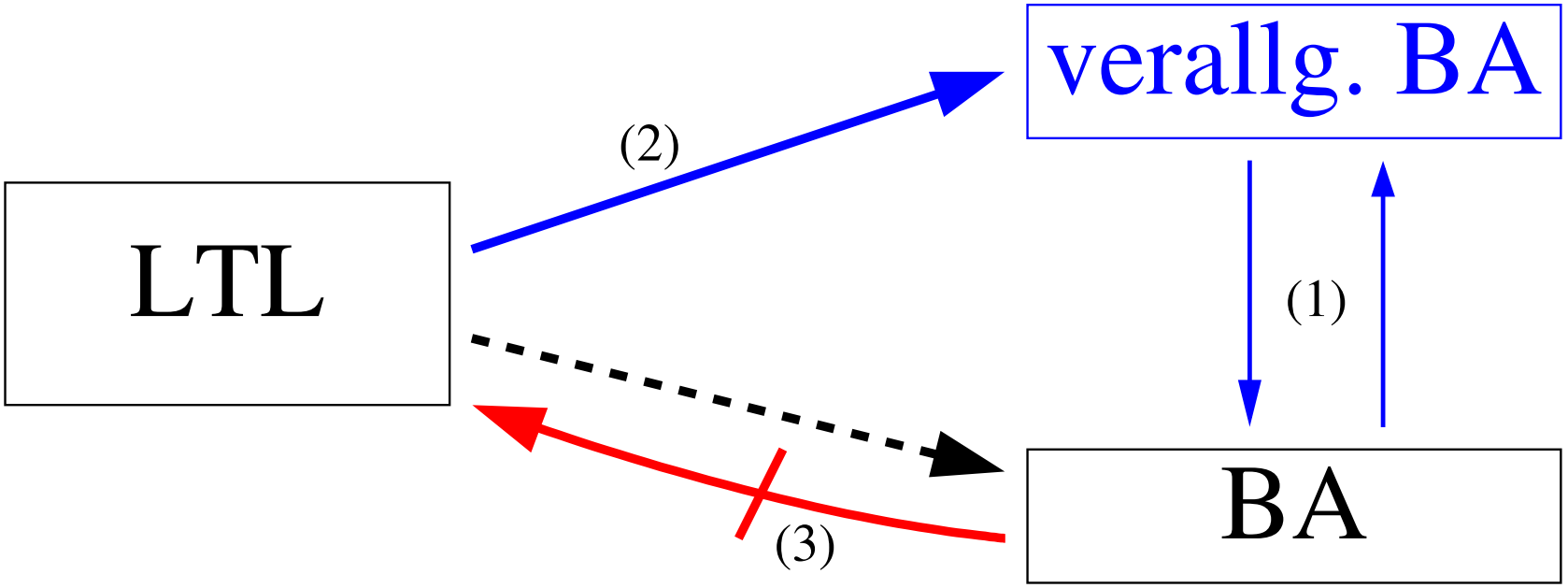
VBAs akzeptieren die gleiche Sprache von Klassen wie BAs.

Programmvorschau



Rückübersetzung BA nach LTL im Allgemeinen nicht möglich.

Programmvorschau



Reihenfolge unseres Vorgehens

Verallgemeinerte Büchi-Automaten

Ein verallgemeinerter Büchi-Automat (VBA) ist ein Tupel $\mathcal{V} = (\Sigma, S, s_0, \Delta, \mathcal{F})$.

Einziger Unterschied zu normalen BA:

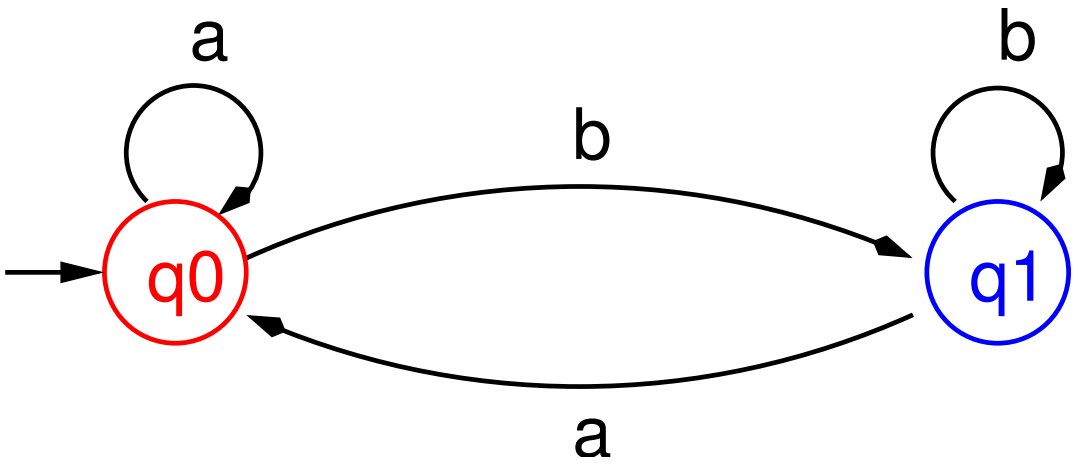
Die Akzeptanzbedingung $\mathcal{F} \subseteq 2^S$ ist eine *Menge von Zustandsmengen*.

Sei z.B. $\mathcal{F} = \{F_1, \ldots, F_n\}$. Ein Lauf ρ in $\mathcal{V}$ ist akzeptierend gdw. für jedes F_i ($i = 1, \ldots, n$) gilt: ρ besucht unendlich oft Zustände aus F_i.

Anders ausgedrückt: mehrere Akzeptanzbedingungen auf einmal.

VBA: Beispiel

Im untenstehenden VBA sei $\mathcal{F} = \{ \{q_0\}, \{q_1\} \}$.



Sprache des Automaten: “unendlich oft *a und* unendlich oft *b*”

Anmerkung: Im Allgemeinen müssen Akzeptanzbedingungen *nicht* disjunkt sein.

Vorteil: VBA können kleiner sein als BA für die gleiche Sprache

Übersetzung BA $\leftrightarrow$ VBA

VBAs akzeptieren die gleiche Klasse von Sprachen wie BAs.

D.h., zu jedem BA gibt es einen VBA, der die gleiche Sprache akzeptiert, und umgekehrt.

Teil 1 der Behauptung (BA $\rightarrow$ VBA):

Sei $\mathcal{B} = (\Sigma, S, s_0, \Delta, F)$ ein (normaler) BA.

Dann ist $\mathcal{V} = (\Sigma, S, s_0, \Delta, \{F\})$ ein VBA mit $\mathcal{L}(\mathcal{V}) = \mathcal{L}(\mathcal{B})$.

Teil 2 der Behauptung (VBA $\longrightarrow$ BA):

Sei $\mathcal{V} = (\Sigma, S, s_0, \Delta, \{F_1, \ldots, F_n\})$ ein VBA.

Wir konstruieren $\mathcal{B} = (\Sigma, S', s_0', \Delta', F)$ wie folgt:

$S' = S \times \{1, \ldots, n\}$

$s_0' = (s_0, 1)$

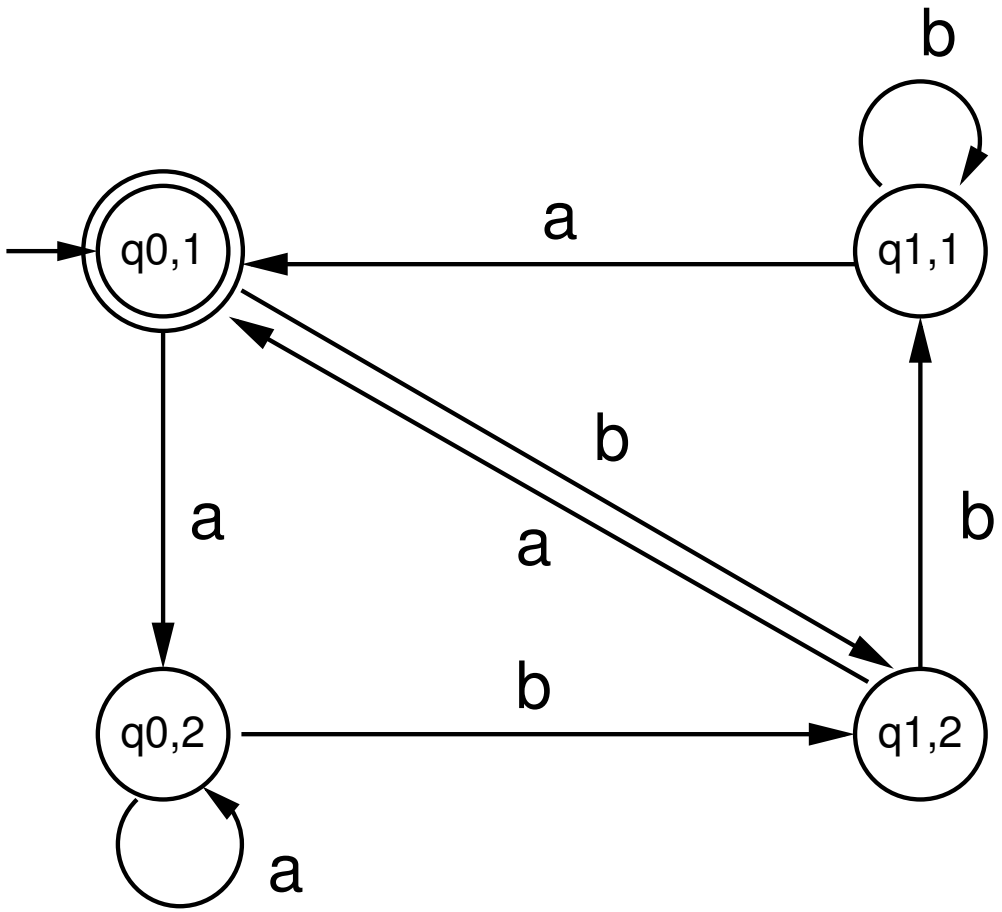
$F = F_1 \times \{1\}$

$((s,i), a, (s',k)) \in \Delta'$ gdw. $1 \leq i \leq n,\ (s, a, s') \in \Delta$
und $k = \begin{cases} i & \text{falls } s \notin F_i \\ (i \bmod n) + 1 & \text{falls } s \in F_i \end{cases}$

Dann gilt $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{V})$. (Idee: n-facher Schnitt)

VBA $\longrightarrow$ BA: Beispiel

Der BA zum vorigen VBA (“unendlich oft *a und* unendlich oft *b*”) ist wie folgt:



Bemerkung: Mehrere Anfangszustände

Unsere Definition von BA bzw. VBA legt fest, dass ein Automat genau einen Anfangszustand hat.

Für die Übersetzung LTL $\rightarrow$ BA wird es nützlich sein, einen VBA mit mehreren Anfangszuständen zu verwenden.

> Bedeutung: Ein Wort wird akzeptiert, wenn es von irgendeinem Startzustand aus akzeptiert wird.

Jeder (V)BA mit mehreren Anfangszuständen kann in einen Automaten mit genau einem Anfangszustand umgewandelt werden (offensichtlich).

Teil 5: LTL und Büchi-Automaten

Überblick

Problemstellung:

Gegeben sei eine LTL-Formel ϕ über AP.

Gesucht: VBA $\mathcal{V}$ (mit mehreren Anfangszuständen) mit $\mathcal{L}(\mathcal{V}) = [\![\phi]\!]$.

($\mathcal{V}$ kann dann in einen normalen BA umgewandelt werden.)

Vorbemerkungen:

Analogie bei regulären Sprachen: reg. Ausdruck $\longrightarrow$ NFA

Unterschied: für LTL $\longrightarrow$ BA ist keine modulare Konstruktion möglich!

Der Automat muss ggfs. mehrere Teilformeln *gleichzeitig* überprüfen (z.B.: $(\mathbf{G}\,\mathbf{F}\,p) \rightarrow (\mathbf{G}(q \rightarrow \mathbf{F}\,r))$ oder $(p\,\mathbf{U}\,q)\,\mathbf{U}\,r$).

Vorbemerkungen:

Im Folgenden lernen wir eine sehr einfache Übersetzungsprozedur kennen.

Diese produziert suboptimale Automaten (exponenziell in $|\phi|$).

Nicht zur manuellen Anwendung gedacht! (nur als Beweis, dass Übersetzung möglich ist, “proof of concept”)

Es gibt bessere Übersetzer (Heuristiken), aber die zugrunde liegende Theorie führt zu weit.

Interessantes, noch nicht abgeschlossenes Forschungsgebiet!

Struktur der Konstruktion

1. Wir überführen ϕ zunächst in eine Normalform mit einfacherer Struktur.

2. Zustände sind jeweils für mehrere Teilformeln “verantwortlich”.

   Bilde die Menge der Teilformeln, jeder Zustand wird mit einer Untermenge beschriftet.

3. Die Übergangsrelation sorgt dafür, dass “einfache” Teilformeln (wie p oder $\mathbf{X}\,p$ erfüllt werden.

4. Die Akzeptanzbedingung sorgt für die Einhaltung der $\mathbf{U}$-Teilformeln.

Negations-Normalform

Definition: Sei AP eine Menge von Grundaussagen. Die Menge der Formeln in *Negations-Normalform* (NNF) über AP ist wie folgt:

Ist $p \in AP$, so sind p und $\neg p$ in NNF.
(Bemerkung: Negationen tauchen *nur* vor Grundaussagen auf.)

Sind ϕ_1 und ϕ_2 in NNF, so auch

$$\phi_1 \vee \phi_2, \quad \phi_1 \wedge \phi_2, \quad \mathbf{X}\,\phi_1, \quad \phi_1\,\mathbf{U}\,\phi_2, \quad \phi_1\,\mathbf{R}\,\phi_2.$$

Negations-Normalform

Definition: Sei AP eine Menge von Grundaussagen. Die Menge der Formeln in *Negations-Normalform* (NNF) über AP ist wie folgt:

Ist $p \in AP$, so sind p und $\neg p$ in NNF.
(Bemerkung: Negationen tauchen *nur* vor Grundaussagen auf.)

Sind ϕ_1 und ϕ_2 in NNF, so auch

$$\phi_1 \vee \phi_2, \quad \phi_1 \wedge \phi_2, \quad \mathbf{X}\,\phi_1, \quad \phi_1\,\mathbf{U}\,\phi_2, \quad \phi_1\,\mathbf{R}\,\phi_2.$$

Behauptung: Jede LTL-Formel ϕ besitzt eine äquivalente Formel in NNF:

$$\begin{array}{rcl@{\qquad}rcl}
\neg(\phi_1\,\mathbf{R}\,\phi_2) & \equiv & \neg\phi_1\,\mathbf{U}\,\neg\phi_2 & \neg(\phi_1\,\mathbf{U}\,\phi_2) & \equiv & \neg\phi_1\,\mathbf{R}\,\neg\phi_2 \\
\neg(\phi_1 \wedge \phi_2) & \equiv & \neg\phi_1 \vee \neg\phi_2 & \neg(\phi_1 \vee \phi_2) & \equiv & \neg\phi_1 \wedge \neg\phi_2 \\
\neg\,\mathbf{X}\,\phi & \equiv & \mathbf{X}\,\neg\phi & \neg\neg\phi & \equiv & \phi
\end{array}$$

NNF: Beispiel

Beispiel einer Umwandlung nach NNF:

$$
\begin{aligned}
\mathbf{G}(p \rightarrow \mathbf{F}\, q) &\equiv \neg\, \mathbf{F}\, \neg(p \rightarrow \mathbf{F}\, q) \\
&\equiv \neg(\mathbf{true}\, \mathbf{U}\, \neg(p \rightarrow \mathbf{F}\, q)) \\
&\equiv \neg \mathbf{true}\, \mathbf{R}\, (p \rightarrow \mathbf{F}\, q) \\
&\equiv \mathbf{false}\, \mathbf{R}\, (\neg p \vee \mathbf{F}\, q) \\
&\equiv \mathbf{false}\, \mathbf{R}\, (\neg p \vee (\mathbf{true}\, \mathbf{U}\, q))
\end{aligned}
$$

Hinweis: Im Folgenden gehen wir davon aus, dass die zu übersetzende Formel ϕ bereits in NNF ist.

Teilformeln

Definition: Sei ϕ eine Formel in NNF. Die Menge $Sub(\phi)$ ist induktiv wie folgt definiert:

- $\phi \in Sub(\phi)$;
- $\mathbf{true} \in Sub(\phi)$;
- falls $\phi_1 \in Sub(\phi)$, dann auch $\neg\phi_1 \in Sub(\phi)$, und umgekehrt;
- falls $\mathbf{X}\,\phi_1 \in Sub(\phi)$, dann auch $\phi_1 \in Sub(\phi)$;
- falls $\phi_1 \vee \phi_2 \in Sub(\phi)$, dann auch $\phi_1, \phi_2 \in Sub(\phi)$;
- falls $\phi_1 \wedge \phi_2 \in Sub(\phi)$, dann auch $\phi_1, \phi_2 \in Sub(\phi)$;
- falls $\phi_1\ \mathbf{U}\ \phi_2 \in Sub(\phi)$, dann auch $\phi_1, \phi_2 \in Sub(\phi)$;
- falls $\phi_1\ \mathbf{R}\ \phi_2 \in Sub(\phi)$, dann auch $\phi_1, \phi_2 \in Sub(\phi)$.

Hinweis: Es gibt $|Sub(\phi)| = \mathcal{O}(|\phi|)$.

Konsistente Mengen

Wir erinnern uns an Punkt 2 der Konstruktion:

> Wir werden Zustände mit Teilmengen von $Sub(\phi)$ beschriften.

Idee (1): Ein Zustand, der mit der Menge *M* beschriftet ist, akzeptiert genau die Sequenzen, die *alle* Formeln aus *M* erfüllen.

Idee (2): Ein Zustand, der mit der Menge *M* beschriftet ist, akzeptiert außerdem genau die Sequenzen, die *alle* Formeln aus $Sub(\phi) \setminus M$ nicht erfüllen.

Daher schließen wir von vornherein solche Mengen *M* aus, die (unter obigen Voraussetzungen) die leere Sprache akzeptieren müssten.

Die übrigen Zustände nennen wir konsistent.

Definition: Eine Menge $M \subset Sub(\phi)$ heißt *konsistent*, wenn sie folgende Bedingungen erfüllt:

- $\mathbf{true} \in M$
- falls $\phi_1 \in Sub(\phi)$, dann $\neg\phi_1 \in M$ gdw. $\phi_1 \notin M$;
- falls $\phi_1 \wedge \phi_2 \in Sub(\phi)$, dann $\phi_1 \wedge \phi_2 \in M$ gdw. $\phi_1 \in M$ und $\phi_2 \in M$;
- falls $\phi_1 \vee \phi_2 \in Sub(\phi)$, dann $\phi_1 \vee \phi_2 \in M$ gdw. $\phi_1 \in M$ oder $\phi_2 \in M$.

Mit $CS(\phi)$ bezeichnen wir die Menge aller konsistenten Teilmengen von $Sub(\phi)$.

Übersetzung (1)

Sei ϕ eine Formel in NNF. Es sei $\mathcal{V} = (\Sigma, S, S_0, \Delta, \mathcal{F})$ ein VBA mit:

$\Sigma = 2^{AP}$

(d.h. die Belegungen über AP)

$S = CS(\phi)$

(d.h. jeder Zustand ist eine konsistente Menge von Teilformeln)

$S_0 = \{ M \in S \mid \phi \in M \}$

(d.h. die Anfangszustände akzeptieren Sequenzen, die ϕ erfüllen)

Δ and $\mathcal{F}$: siehe nachstehende Folie

Übersetzung (2)

Übergangsrelation: $(M, \sigma, M') \in \Delta$ gdw. $\sigma = M \cap AP$ und:

- wenn $\mathbf{X}\,\phi_1 \in Sub(\phi)$, dann $\mathbf{X}\,\phi_1 \in M$ gdw. $\phi_1 \in M'$;
- wenn $\phi_1\,\mathbf{U}\,\phi_2 \in Sub(\phi)$, dann $\phi_1\,\mathbf{U}\,\phi_2 \in M$
  gdw. $\phi_2 \in M$ oder ($\phi_1 \in M$ und $\phi_1\,\mathbf{U}\,\phi_2 \in M'$);
- wenn $\phi_1\,\mathbf{R}\,\phi_2 \in Sub(\phi)$, dann $\phi_1\,\mathbf{R}\,\phi_2 \in M$
  gdw. $\phi_1 \wedge \phi_2 \in M$ oder ($\phi_2 \in M$ und $\phi_1\,\mathbf{R}\,\phi_2 \in M'$).

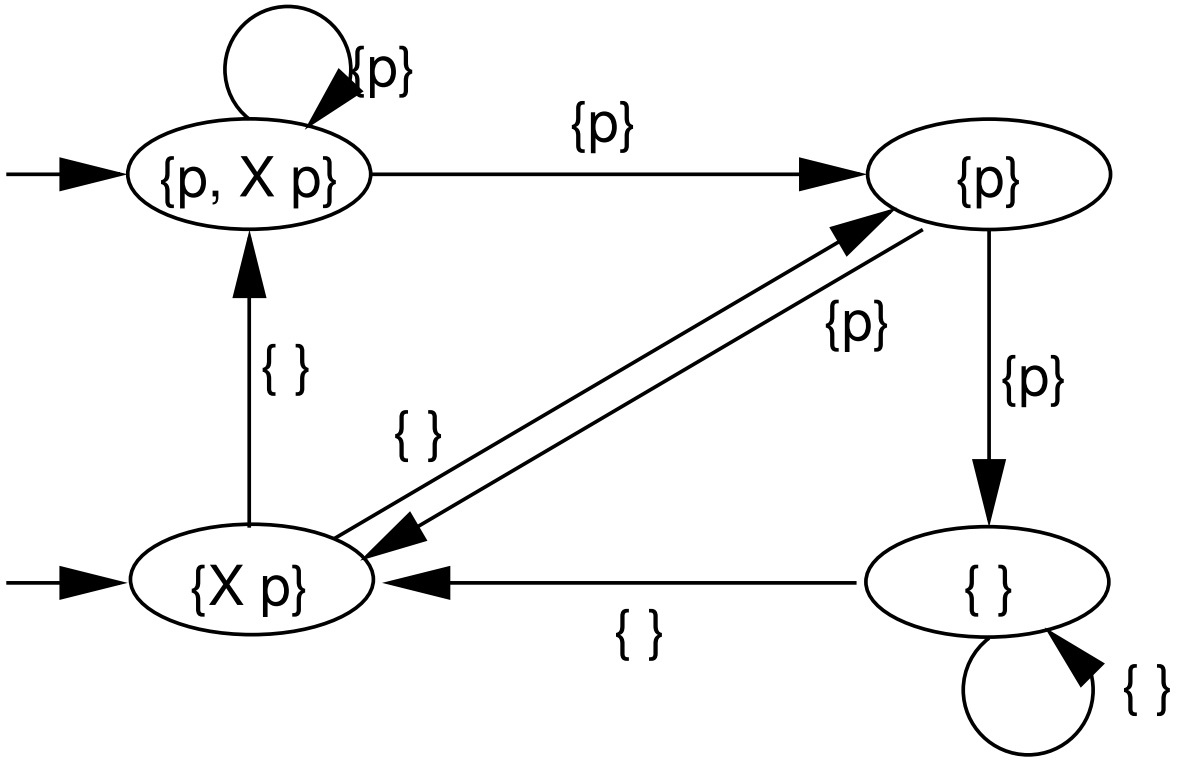
Akzeptanzbedingung:

$\mathcal{F}$ enthält eine Menge F_ψ für jede Teilformel ψ der Form $\phi_1\,\mathbf{U}\,\phi_2$, so dass

$$F_\psi = \{\, M \in CS(\phi) \mid \phi_2 \in M \ \text{ oder } \ \neg(\phi_1\,\mathbf{U}\,\phi_2) \in M \,\}.$$

Übersetzung: Beispiel 1

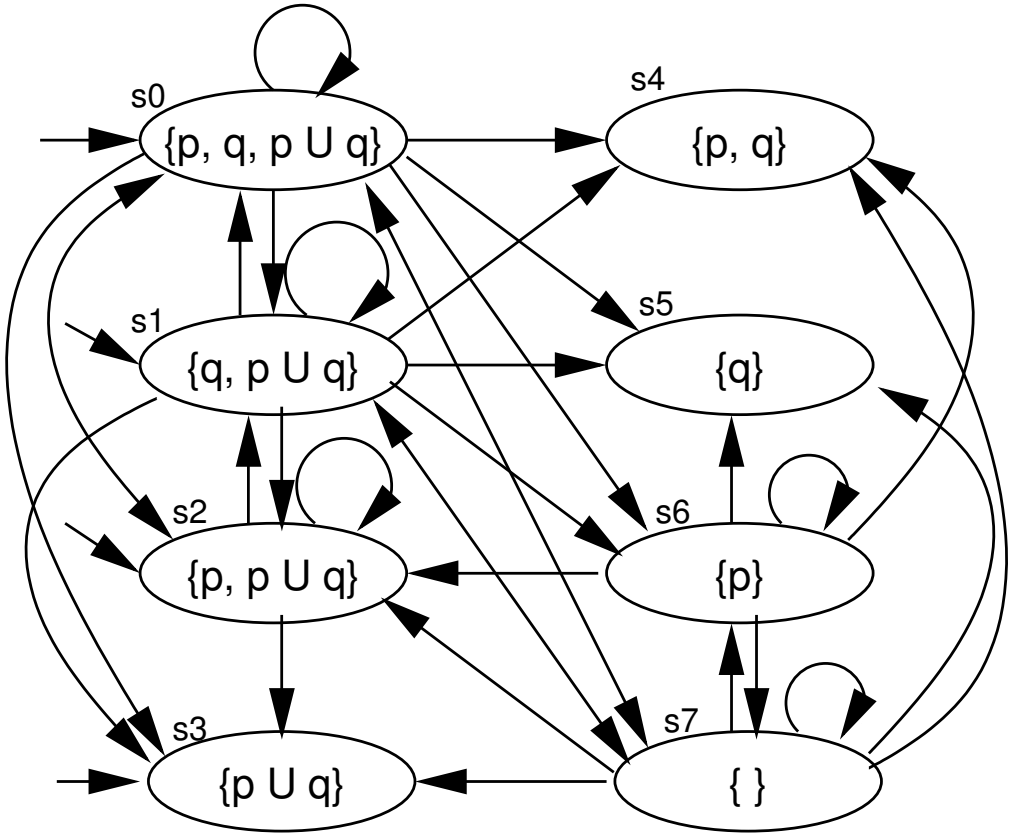
$\phi = \mathbf{X}\, p$



Dieser VBA hat zwei Anfangszustände und Akzeptanzbedingung $\mathcal{F} = \emptyset$, d.h. jeder mögliche unendliche Lauf ist akzeptierend. (Die negierten Formeln wurden aus den Zustandsbeschriftungen weggelassen.

Übersetzung: Beispiel 2

$\phi \equiv p \, \mathbf{U} \, q$



VBA mit $\mathcal{F} = \{\{s_0, s_1, s_4, s_5, s_6, s_7\}\}$, hier auch die Transitionsbeschriftungen weggelassen.

Korrektheitsbeweis

Wir wollen Folgendes beweisen:

$$\sigma \in \mathcal{L}(\mathcal{V}) \quad \text{gdw.} \quad \sigma \in [\![\phi]\!]$$

Dazu beweisen wir folgende allgemeinere Eigenschaft:

> Sei α irgendeine Folge konsistenter Mengen (d.h. Zuständen in $\mathcal{V}$) und sei σ irgendeine Folge von Belegungen über AP.
>
> α ist ein akzeptierender Lauf von $\mathcal{V}$ über σ
> gdw. $\sigma^i \in [\![\psi]\!]$ für alle $i \geq 0$ und $\psi \in \alpha(i)$.
>
> Obige Korrektheit folgt dann aus der Wahl der Anfangszustände.

Korrektheit (2)

Vorbemerkung: Auf beiden Seiten der Äquivalenz gilt nach Konstruktion $\sigma(i) = \alpha(i) \cap AP$ für alle $i \geq 0$.

Beweis durch strukturelle Induktion über ψ:

für $\psi = p$ und $\psi = \neg p$, falls $p \in AP$:
offensichtlich, da $\sigma^i \in [\![p]\!]$ gdw. $p \in \sigma(i)$ gdw. $p \in \alpha(i)$.

für $\psi_1 \vee \psi_2$ und $\psi_1 \wedge \psi_2$: folgt daraus, dass $\alpha(i)$ konsistent ist und aus der IV über ψ_1 bzw. ψ_2.

für $\mathbf{X}\,\psi_1$: folgt aus der Konstruktion von Δ sowie IV über ψ_1.

Korrektheit (3)

für $\psi = \psi_1 \, \mathbf{R} \, \psi_2$:

Folgt aus der Konstruktion von $\triangle$, der Rekursionsformel für $\mathbf{R}$ und der Induktionsvoraussetzung.

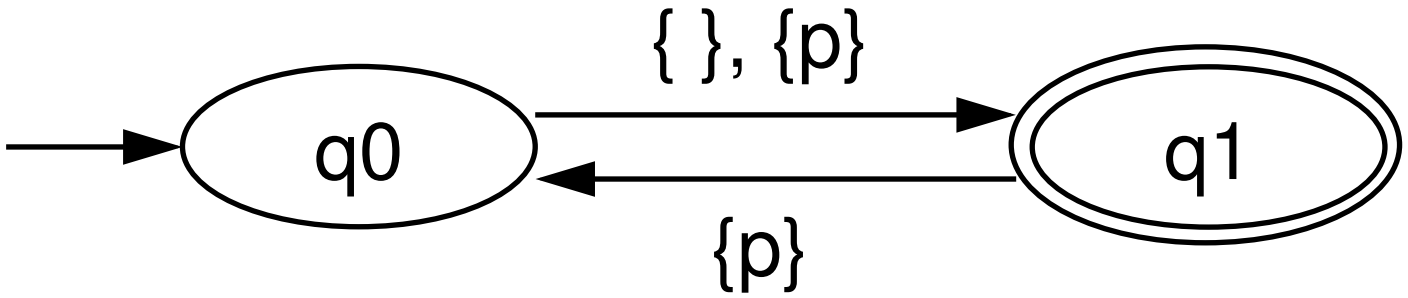
für $\psi = \psi_1 \, \mathbf{U} \, \psi_2$:

Analog zu $\mathbf{R}$, nur müssen wir zusätzlich erzwingen, dass $\psi_2 \in \alpha(k)$ für irgendein $k \geq i$. Angenommen, dies sei nicht der Fall, dann gilt $\psi_1 \, \mathbf{U} \, \psi_2 \in \alpha(k)$ für alle $k \geq i$. Keiner dieser Zustände ist aber in F_ψ, so dass α nicht akzeptierend sein kann. Daher muss die Annahme falsch sein.

Übersetzung BA $\longrightarrow$ LTL

Die umgekehrte Übersetzung (BA $\longrightarrow$ LTL) ist *nicht* immer möglich.

D.h., es gibt Büchi-Automaten $\mathcal{B}$, für die es keine Formel ϕ gibt mit $\mathcal{L}(\mathcal{B}) = [\![\phi]\!]$ (Wolper, 1983).



Die Eigenschaft "*p* gilt in jedem zweiten Schritt" lässt sich nicht in LTL ausdrücken (Beweis: nächste Folie).

Beweis (BA $\not\rightarrow$ LTL)

Wir zeigen zunächst die folgende, allgemeinere Tatsache:

> Sei ϕ eine beliebige LTL-Formel über AP und n die Anzahl der $\mathbf{X}$-Operatoren in ϕ. Wir betrachten die Sequenzen
>
> $$\sigma_i = \{p\}^i \, \emptyset \, \{p\}^\omega$$
>
> für $i \geq 0$. Für alle Paare $i, k > n$ gilt: $\sigma_i \models \phi$ gdw. $\sigma_k \models \phi$.

Beweis durch strukturelle Induktion über ϕ:

> Falls $\phi = p$, für $p \in AP$, dann ist $n = 0$ und $i, k \geq 1$.
> Also gilt $\sigma_i \models p$ und $\sigma_k \models p$.
>
> Für die übrigen Fälle nehmen wir als Induktionsvoraussetzung, dass die Aussage für ϕ_1 und ϕ_2 gilt, d.h. wenn darin n_1- bzw. n_2-mal der $\mathbf{X}$-Operator vorkommt, so gilt für alle $i_1, k_1 > n_1$, dass $\sigma_{i_1} \models \phi_1$ gdw. $\sigma_{k_1} \models \phi_1$, und analog für ϕ_2.

Falls $\phi = \neg\phi_1$, so folgt das Gewünschte direkt aus der IV.

Für $\phi = \phi_1 \vee \phi_2$: analog.

Falls $\phi = \mathbf{X}\,\phi_1$, dann ist $n_1 = n - 1$. Da $i - 1, k - 1 > n - 1 = n_1$ sind, gilt nach IV: $\sigma_i^1 = \sigma_{i-1} \models \phi_1$ gdw. $\sigma_k^1 = \sigma_{k-1} \models \phi_1$, woraus das Gewünschte folgt.

Für $\phi = \phi_1\ \mathbf{U}\ \phi_2$: Sei $m > n$ beliebig. Es gilt:

$$\phi_1\ \mathbf{U}\ \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \mathbf{X}(\phi_1\ \mathbf{U}\ \phi_2))$$

Durch rekursive Anwendung dieses Gesetzes erhalten wir:

$$\sigma_m \models \phi \quad \text{gdw.} \quad \sigma_m \models \phi_2 \vee (\sigma_m \models \phi_1 \wedge (\sigma_{m-1} \models \phi_2 \vee (\ldots$$
$$(\sigma_{n+1} \models \phi_1 \wedge \sigma_n \models \phi_1\ \mathbf{U}\ \phi_2))))$$

Nach IV können wir die Indizes größer n äquivalent durch $n+1$ ersetzen:

$$\sigma_m \models \phi \quad \text{gdw.} \quad \sigma_{n+1} \models \phi_2 \vee (\sigma_{n+1} \models \phi_1 \wedge (\sigma_{n+1} \models \phi_2 \vee (\ldots (\sigma_{n+1} \models \phi_1 \wedge \sigma_n \models \phi_1 \,\mathbf{U}\, \phi_2))))$$

Dies lässt sich äquivalent wie folgt vereinfachen:

$$\sigma_m \models \phi \quad \text{gdw.} \quad \sigma_{n+1} \models \phi_2 \vee (\sigma_{n+1} \models \phi_1 \wedge \sigma_n \models \phi_1 \,\mathbf{U}\, \phi_2)$$

Die Gültigkeit von $\sigma_m \models \phi$ ist also vollkommen unabhängig von m, woraus die gewünschte Aussage für i und k folgt.

Damit ist der Beweis für die allgemeinere Tatsache vollständig.

Nehmen wir nun an, es gäbe eine LTL-Formel ϕ für die Eigenschaft das vorigen BA ("*p* gilt in jedem zweiten Schritt"). Sei *n* die Anzahl der $\mathbf{X}$-Operatoren in ϕ.

Betrachten wir die Sequenzen σ_{n+1} und σ_{n+2}.

Wenn *n* gerade ist, so gilt $\sigma_{n+1} \not\models \phi$ und $\sigma_{n+2} \models \phi$. Wenn *n* ungerade ist, verhält es sich umgekehrt.

Die zuvor bewiesene Tatsache zeigt jedoch, dass dies nicht möglich ist: σ_{n+1} und σ_{n+2} erfüllen ϕ beide, oder beide erfüllen ϕ *nicht*. Also kann es eine solche Formel ϕ nicht geben.

Das LTL-Model-Checking-Problem (Vorschau)

Gegeben: Kripke-Struktur $\mathcal{K} = (S, \to, r, AP, \nu)$, LTL-Formel ϕ über AP.

Gefragt: Gilt $\mathcal{K} \models \phi$?

Lösung: (Ansatz)

Wir betrachten $\mathcal{K}$ als einen Büchi-Automaten $\mathcal{B}_\mathcal{K}$:

$\mathcal{B}_\mathcal{K} = (2^{AP}, S, r, \Delta, S)$, wobei $\Delta = \{ (s, \nu(s), t) \mid s \to t \}$

Offensichtlich gilt $[\![\mathcal{K}]\!] = \mathcal{L}(\mathcal{B}_\mathcal{K})$.

Wir übersetzen $\neg\phi$ in einen Büchi-Automaten $\mathcal{B}_{\neg\phi}$.

Es gilt:

$$
\begin{array}{rrcl}
 & \mathcal{K} & \models & \phi \\
\Longleftrightarrow & [\![\mathcal{K}]\!] & \subseteq & [\![\phi]\!] \\
\Longleftrightarrow & [\![\mathcal{K}]\!] & \cap & [\![\neg\phi]\!] = \emptyset \\
\Longleftrightarrow & \mathcal{L}(\mathcal{B}_{\mathcal{K}}) & \cap & \mathcal{L}(\mathcal{B}_{\neg\phi}) = \emptyset
\end{array}
$$

Ergo:

- Büchi-Automaten $\mathcal{B}_{\mathcal{K}}$ und $\mathcal{B}_{\neg\phi}$ konstruieren.
- Die beiden Automaten schneiden (Spezialfall der Schnitt-Konstruktion!)
- MC-Problem = Leerheitsproblem des Schnitt-Automaten

Komplexität der Übersetzung

Die vorgestellte Übersetzung liefert zu einer Formel ϕ einen BA der Größe $\mathcal{O}(2^{|\phi|})$.

Dadurch ist der vorgestellte Model-Checking-Algorithmus exponenziell in $|\phi|$.

Frage: Gibt es eine bessere LTL-Büchi-Übersetzung?

Antwort 1: Nein (zumindest nicht generell). Es gibt Formeln mit notwendigerweise exponenziell großen Automaten.

Beispiel: Folgende Formel über $\{p_0, \ldots, p_{n-1}\}$ erzwingt, dass die Werte der p_i wie in einem n-Bit-Zähler alternieren.

$$\mathbf{G}(p_0 \not\leftrightarrow \mathbf{X}\, p_0) \wedge \bigwedge_{i=1}^{n-1} \mathbf{G}\left(\left(p_i \not\leftrightarrow \mathbf{X}\, p_i\right) \leftrightarrow \left(p_{i-1} \wedge \neg\, \mathbf{X}\, p_{i-1}\right)\right)$$

Die Formel hat Größe $\mathcal{O}(n)$. Offensichtlich muss ein Automat für diese Formel mindestens 2^n Zustände haben.

Antwort 2: Ja (manchmal). Es gibt Übersetzungsprozeduren, die *in vielen Fällen* kleinere Automaten produzieren.

Einige Tools:

Spin (Aufruf: `spin -f 'p U q'`)

LTL2BA (Web-Applet)

Literatur:

Gerth, Peled, Vardi, Wolper: *Simple On-the-fly Automatic Verification of Linear Temporal Logic*, 1996

Oddoux, Gastin: *Fast LTL to Büchi Automata Translation*, 2001

Teil 6: Leerheitstest

für Büchi-Automaten

Überblick

Wie gesehen reduziert sich das Model-Checking-Problem auf die Feststellung, ob die Sprache eines Büchi-Automaten $\mathcal{B}$ leer ist.

Zur Erinnerung: $\mathcal{B}$ entsteht aus dem Schnitt einer Kripke-Struktur $\mathcal{K}$ mit dem BA für die *Negation* von ϕ.

Wenn $\mathcal{B}$ irgendein Wort akzeptiert, nennen wir ein solches Wort Gegenbeispiel.

$\mathcal{K} \models \phi$ gdw. $\mathcal{B}$ akzeptiert die leere Sprache.

Typische Instanzen:

Größe von $\mathcal{K}$: ein paar Hundert, mehrere Tausend, ggfs. Millionen von Zuständen.

Größe von $\mathcal{B}_{\neg\phi}$: meist nur eine Handvoll Zustände

Typische Vorgehensweise (z.B. in Spin):

$\mathcal{K}$ indirekt gegeben durch ein Programm in irgendeiner Beschreibungssprache (C, Java / bei Spin: Promela); Model-Checking-Programm generiert $\mathcal{K}$ intern.

$\mathcal{B}_{\neg\phi}$ zu Beginn des Tests aus ϕ generiert.

Typische Vorgehensweise:

$\mathcal{B}$ wird zur Laufzeit aus (der Beschreibung von) $\mathcal{K}$ und aus $\mathcal{B}_{\neg\phi}$ generiert und *zugleich* auf Leerheit getestet (Model-Checking “on-the-fly”).

Geeignete Darstellung von $\mathcal{B}$: impliziter Graph (s_0, succ), d.h. Anfangszustand r und Nachfolgerfunktion $\mathsf{succ}\colon S \to 2^S$

Größe von $\mathcal{B}$ zu Beginn der Prozedur nicht bekannt!

Speicherbedarf

Transitionen werden nicht explizit gespeichert, sondern durch Aufrufe von succ nach Bedarf generiert.

Hashtabelle für bereits verarbeitete Zustände

Speicherbedarf pro Zustand:

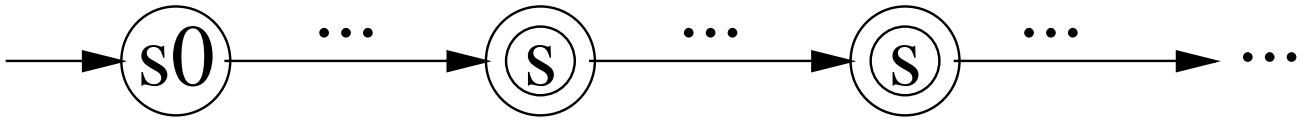
Deskriptor: Programmzähler, Werte von Variablen, aktive Prozesse etc.
(oft 10 bis einige Hundert Bytes)

Verwaltungsinformation: Daten, die vom Leerheitstest gebraucht werden
(ein paar Bytes)

Simpler Algorithmus I: Lasso-Check

Sei $\mathcal{B} = (\Sigma, S, s_0, \delta, F)$ ein Büchi-Automat.

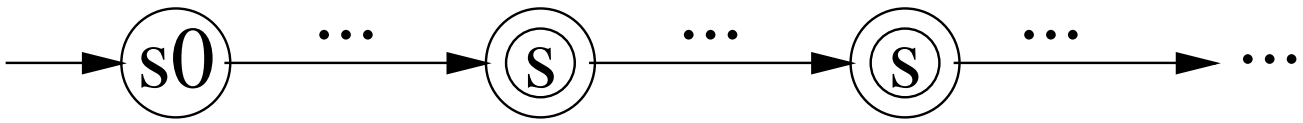
$\mathcal{L}(\mathcal{B}) \neq \emptyset$ gdw. es gibt $s \in F$, so dass $s_0 \rightarrow^* s \rightarrow^+ s$



Simpler Algorithmus I: Lasso-Check

Sei $\mathcal{B} = (\Sigma, S, s_0, \delta, F)$ ein Büchi-Automat.

$\mathcal{L}(\mathcal{B}) \neq \emptyset$ gdw. es gibt $s \in F$, so dass $s_0 \rightarrow^* s \rightarrow^+ s$

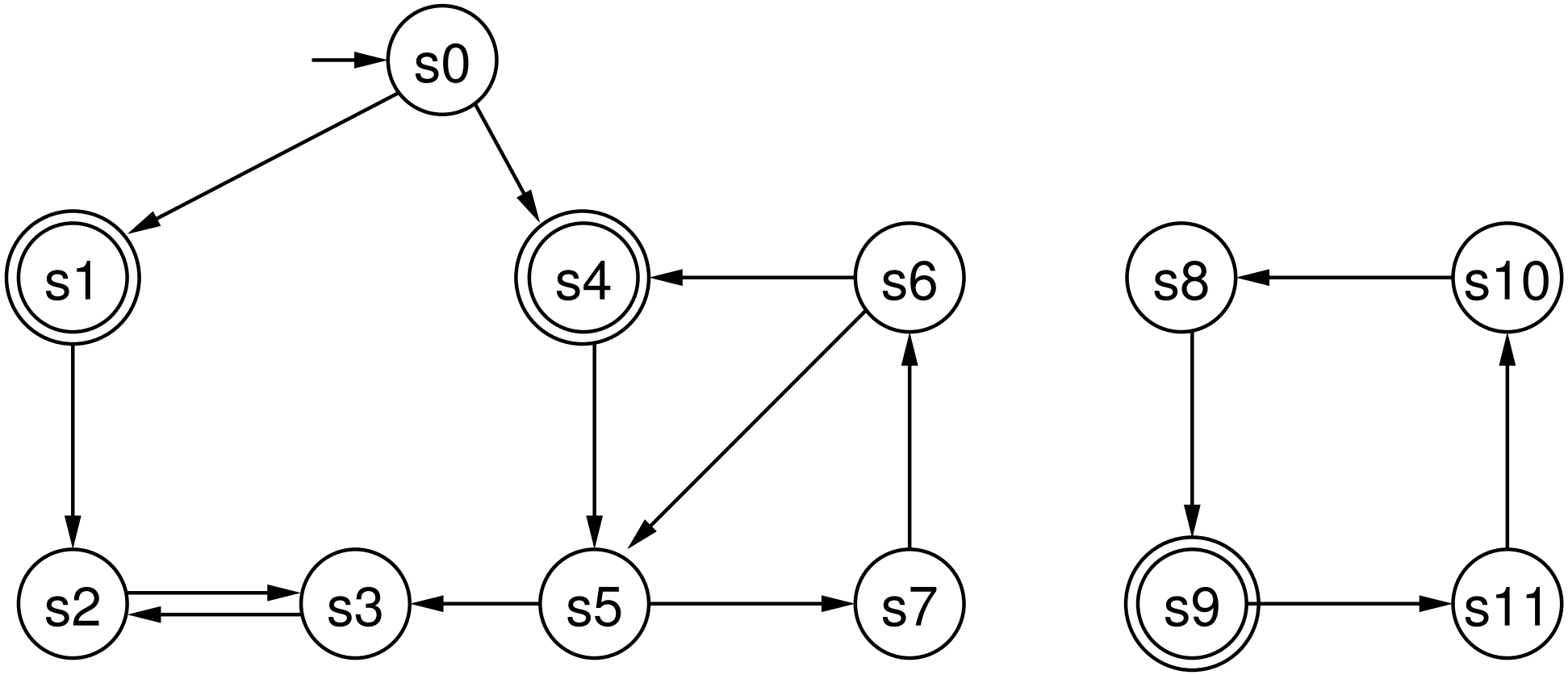


Naive Lösung:

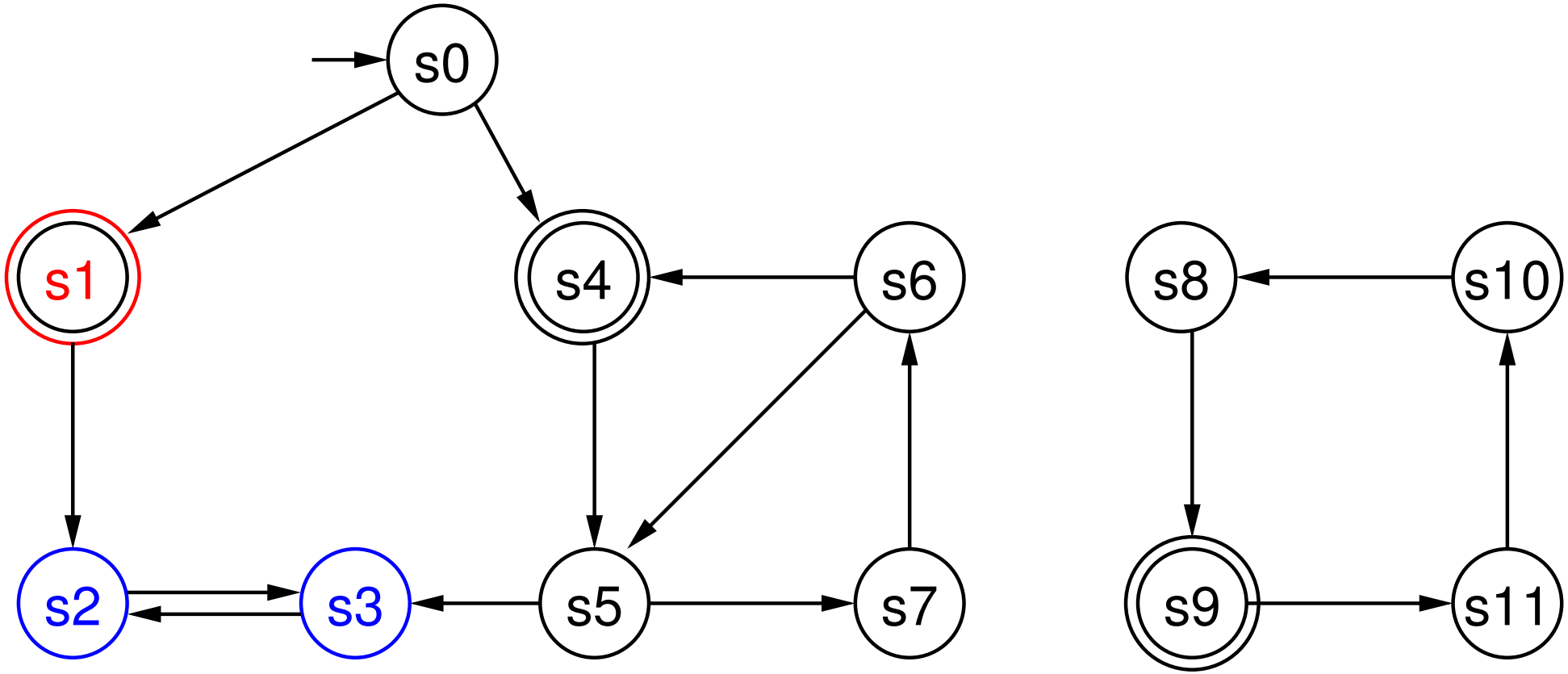
Stelle für jedes $s \in F$ fest, ob von s ein Kreis möglich ist; sei $F_\circ \subseteq F$ die Menge der Zustände, für die das der Fall ist.

Stelle fest, ob aus s_0 ein Zustand aus $F_\circ$ erreichbar ist.

Beispiel

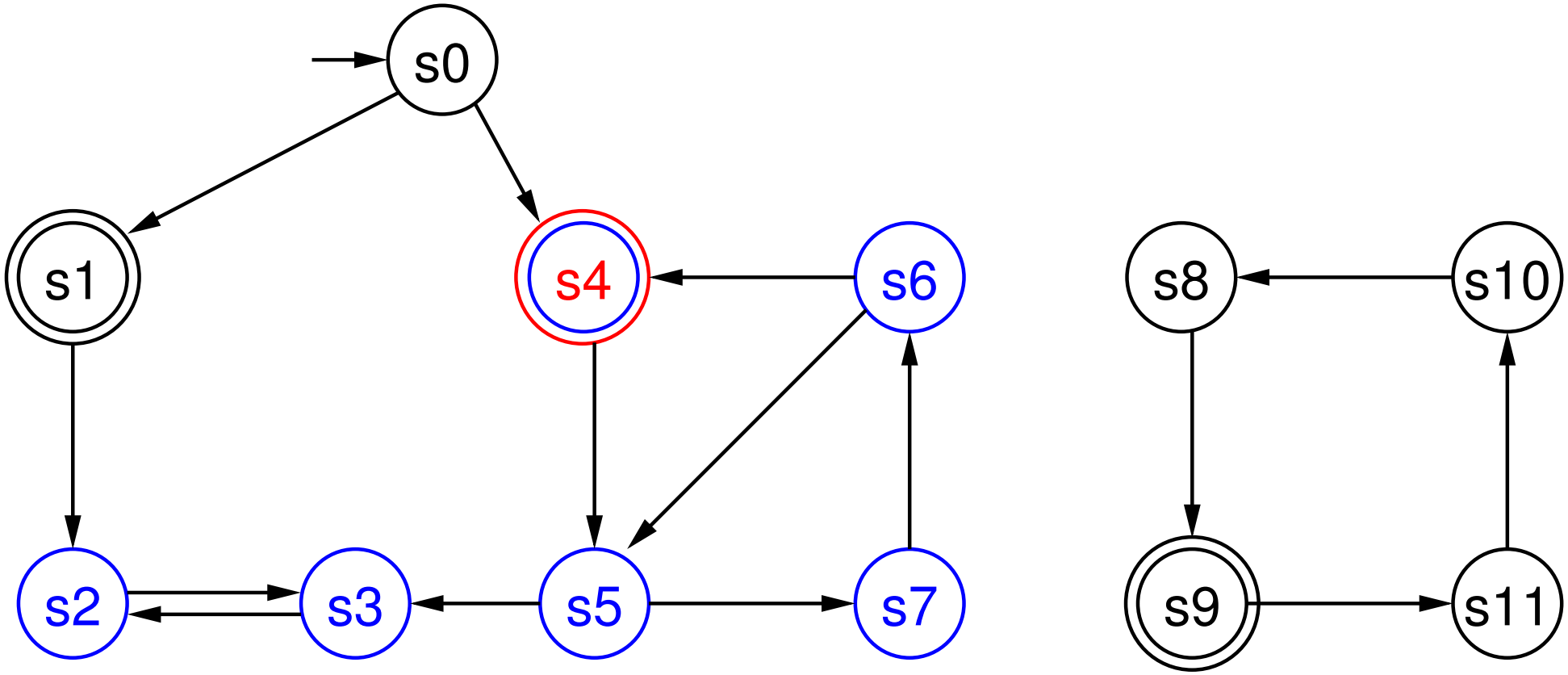


Beispiel: Von s_1 erreichbare Zustände



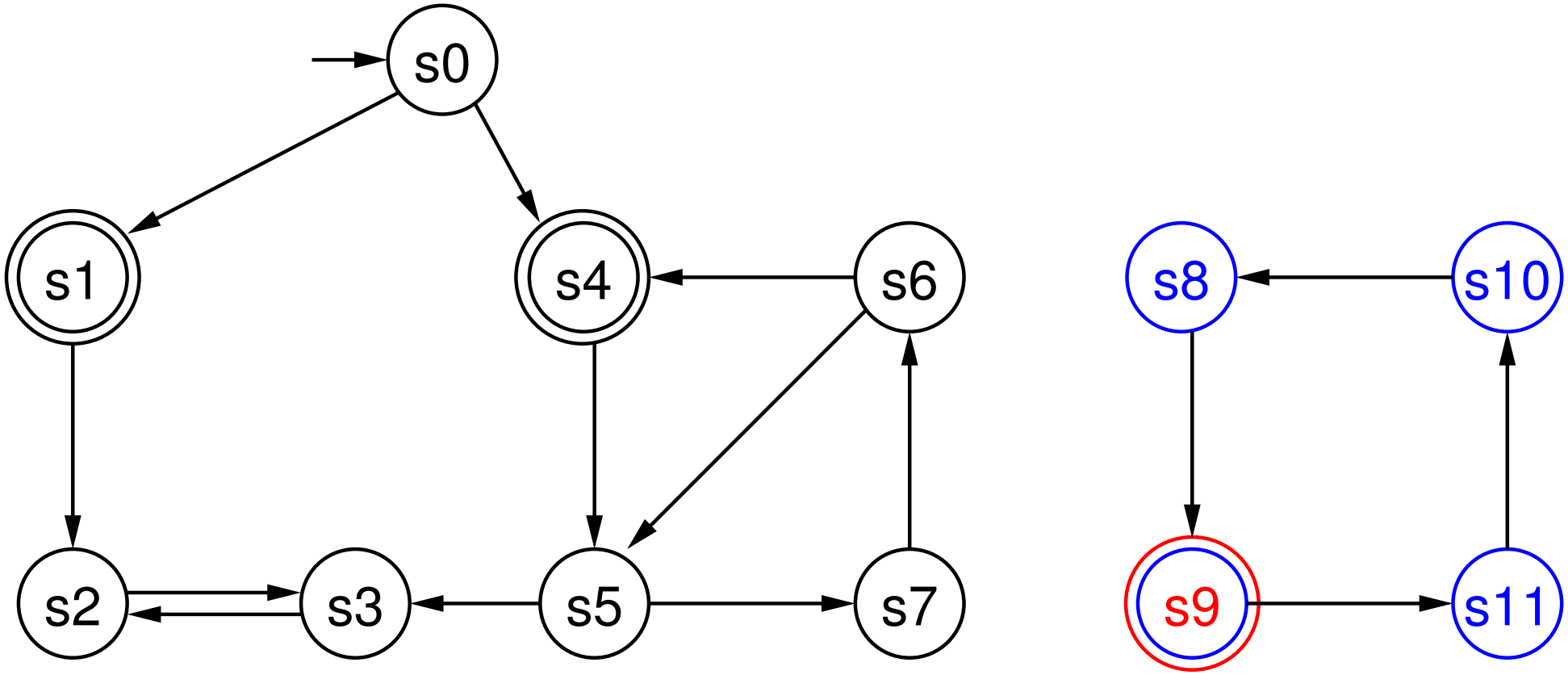
Es gibt keinen Kreis, der s_1 enthält.

Beispiel: Von s_4 erreichbare Zustände



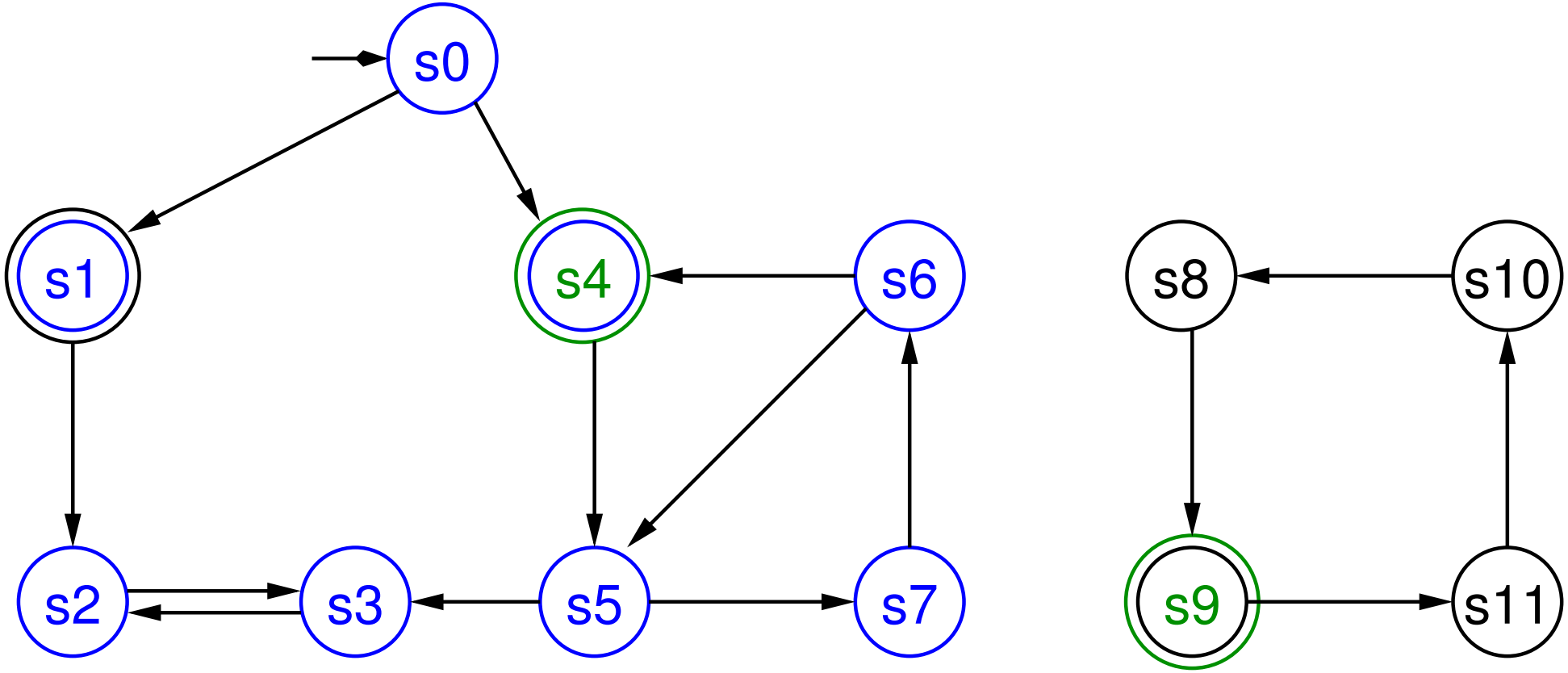
Es gibt einen Kreis: $s_4 \rightarrow s_5 \rightarrow s_7 \rightarrow s_6 \rightarrow s_4$.

Beispiel: Von s_9 erreichbare Zustände



Es gibt einen Kreis: $s_9 \rightarrow s_{10} \rightarrow s_{11} \rightarrow s_8 \rightarrow s_9$.

Beispiel: Von s_0 erreichbare Zustände



Automat nicht-leer, da $s_4 \in F_\circ$ erreichbar ist.

Lasso-Check: Zeitbedarf

Dieser simple Test muss $|F| + 1$ Male den BA durchsuchen.

Zeitbedarf für jede Suche: Anzahl Zustände + Anzahl Transitionen

Gesamtbedarf: $\mathcal{O}(|F| \cdot (|S| + |\delta|))$, d.h. quadratisch in $|\mathcal{B}|$.

Quadratische Laufzeit bei Mio. Zuständen: inakzeptabel

Starke Zusammenhangskomponenten

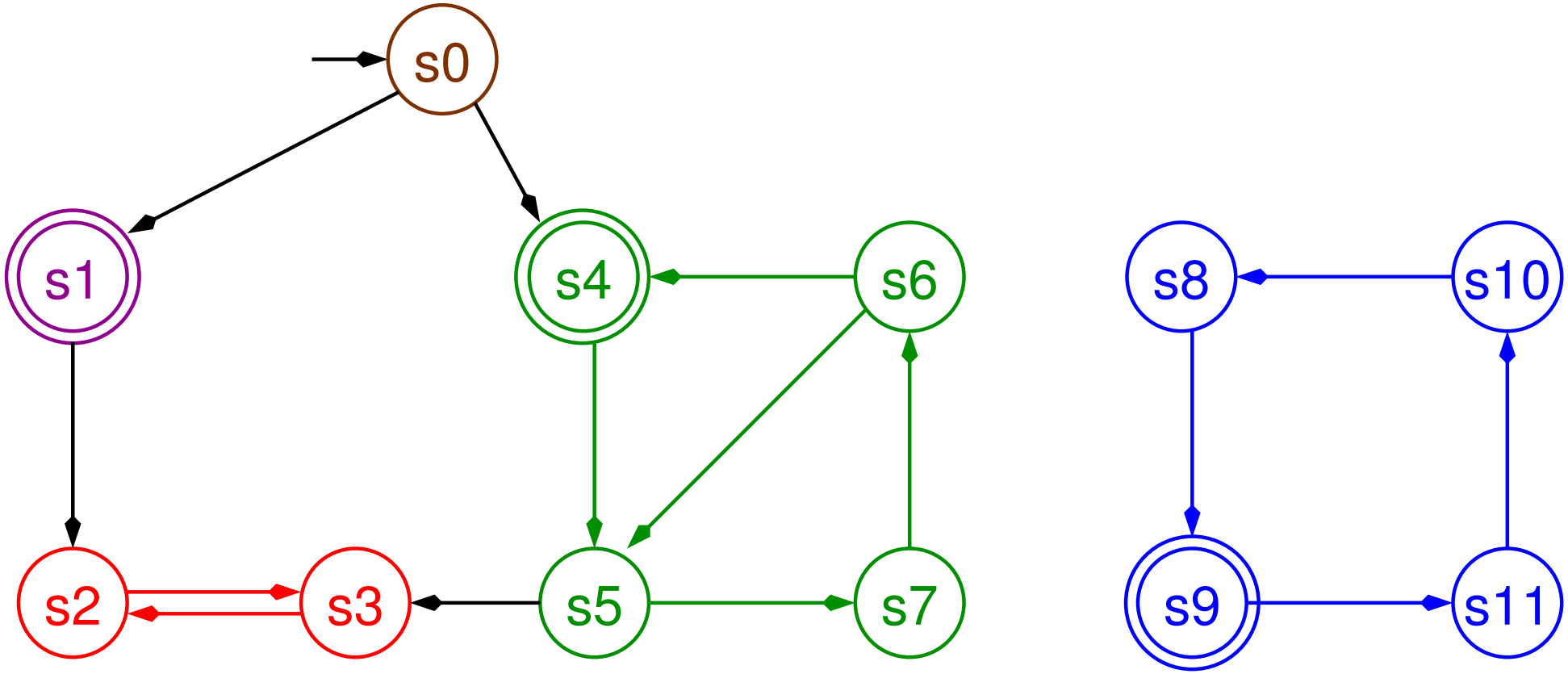
$C \subseteq S$ heisst starke Zusammenhangskomponente (SCC), gdw.

für alle $s, s' \in C$ gilt: $s \rightarrow^* s'$;

es gibt keine echte Obermenge von C, die diese Bedingung erfüllt (d.h. C ist maximal).

Eine starke Zusammenhangskomponente C heisst trivial, gdw. $|C| = 1$ und für $s \in C$ gilt $s \not\rightarrow s$ (einzelner Zustand ohne Schleife).

Beispiel: SCCs



Die SCCs $\{s_0\}$ und $\{s_1\}$ sind trivial.

Simpler Algorithmus II: SCCs

Beobachtung: $\mathcal{L}(\mathcal{B}) \neq \emptyset$, gdw. $\mathcal{B}$ enthält eine von s_0 erreichbare nicht-triviale SCC, die einen akzeptierenden Zustand enthält.

Simpler Algorithmus II: SCCs

Beobachtung: $\mathcal{L}(\mathcal{B}) \neq \emptyset$, gdw. $\mathcal{B}$ enthält eine von s_0 erreichbare nicht-triviale SCC, die einen akzeptierenden Zustand enthält.

Einfacher Algorithmus: für jeden akzeptierenden Zustand s

berechne die Menge V_s der Vorgänger von s;

berechne die Menge N_s der Nachfolger von s;

$V_s \cap N_s$ ist die SCC, die s enthält;

teste, ob $V_s \cap N_s \supset \{s\}$ oder $s \rightarrow s$.

Simpler Algorithmus II: SCCs

Beobachtung: $\mathcal{L}(\mathcal{B}) \neq \emptyset$, gdw. $\mathcal{B}$ enthält eine von s_0 erreichbare nicht-triviale SCC, die einen akzeptierenden Zustand enthält.

Einfacher Algorithmus: für jeden akzeptierenden Zustand s

berechne die Menge V_s der Vorgänger von s;

berechne die Menge N_s der Nachfolger von s;

$V_s \cap N_s$ ist die SCC, die s enthält;

teste, ob $V_s \cap N_s \supset \{s\}$ oder $s \rightarrow s$.

Laufzeit: erneut quadratisch

Effiziente Lösung

Im Folgenden werden wir eine Lösung kennenlernen, die linear in $|\mathcal{B}|$ ist (d.h. proportional zu $|S| + |\delta|$).

Die Lösung basiert auf Tiefensuche (depth-first search, DFS) und partitioniert $\mathcal{B}$ dabei in SCCs.

Literatur: Couvreur 1999, Gabow 2000

Tiefensuche (Basis-Version)

```
nr = 0;
hash = {};
dfs(s0);
exit;

dfs(s) {
   add s to hash;
   nr = nr+1;
   s.num = nr;

   for (t in succ(s)) {
      // verarbeite Transition s -> t
      if (t not yet in hash) { dfs(t); }
   }
}
```

Speicherbedarf

Globale Variablen: Zähler *nr*, Hashtabelle für Zustände

Verwaltungsinformation: “DFS-Nummer” *s.num*

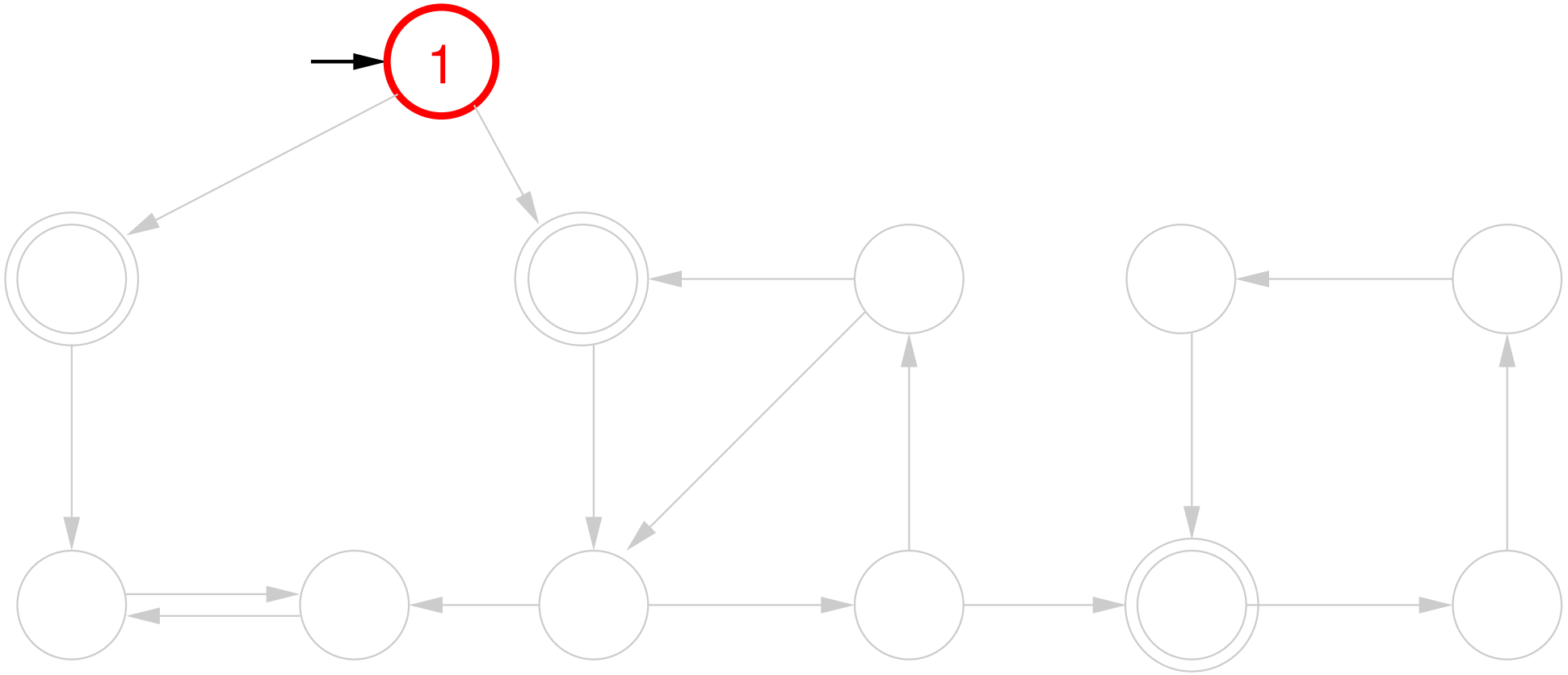
Suchpfad: Keller (Stack) mit den noch nicht abgeschlossenen Aufrufen von *dfs*

Eigenschaften des Suchpfads

(1) Sei $s_0 s_1 \ldots s_n$ der Suchpfad zu irgendeinem Zeitpunkt.
Dann gilt $s_i.num < s_j.num$ gdw. $i < j$.
Außerdem gilt $s_i \rightarrow^* s_j$, falls $i < j$.

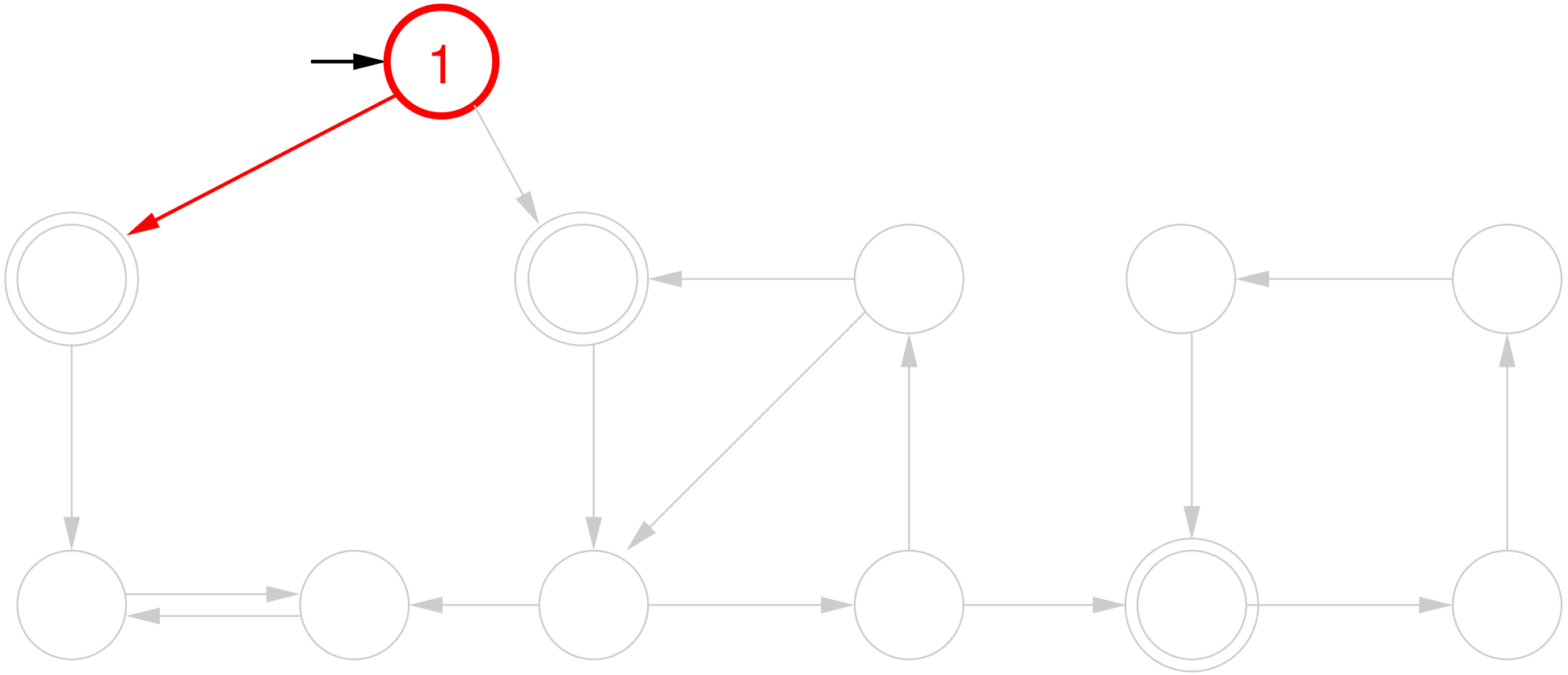
Beweis: Folgt aus der Logik des Programms bzw. aus der Reihenfolge der rekursiven Aufrufe.

Beispiel: Tiefensuche



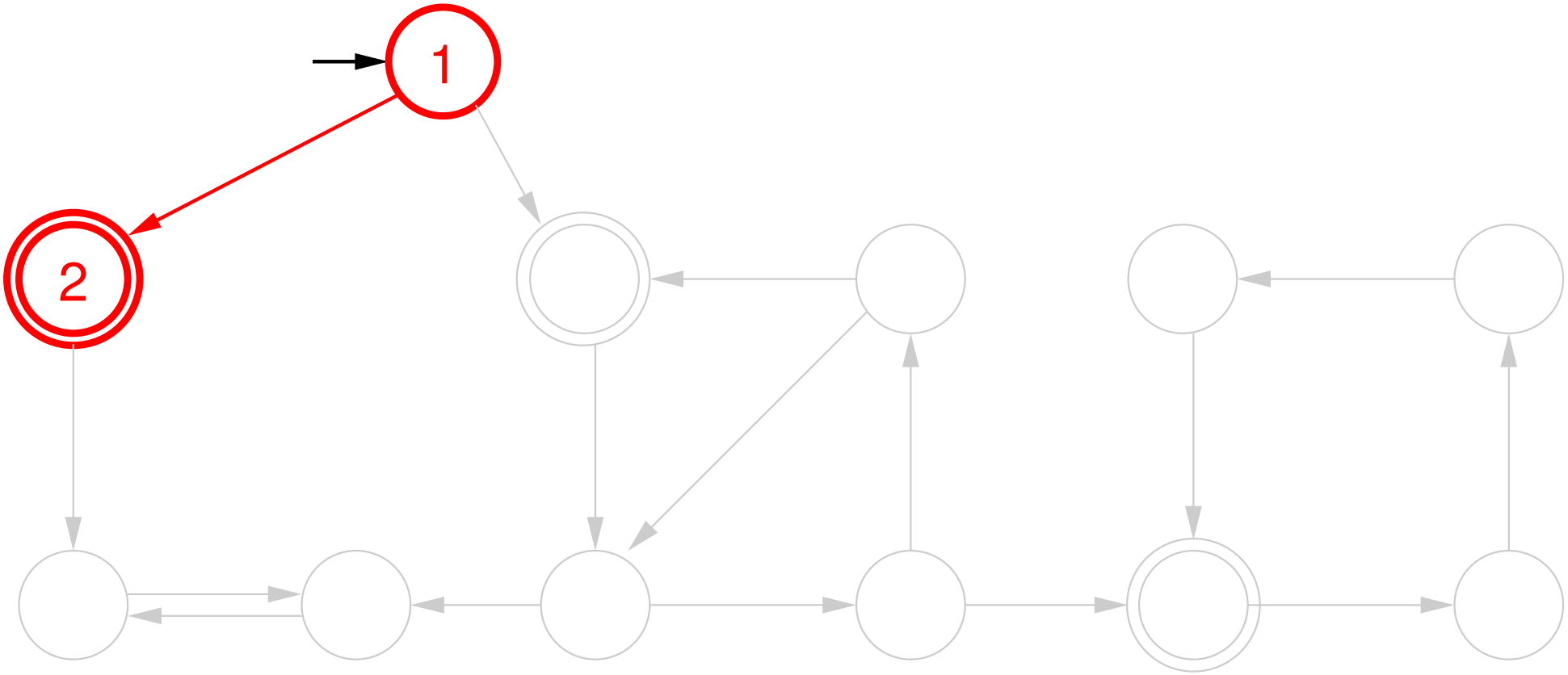
Suchpfad rot dargestellt, andere gesehene Zustände schwarz, sonstige grau.

Beispiel: Tiefensuche



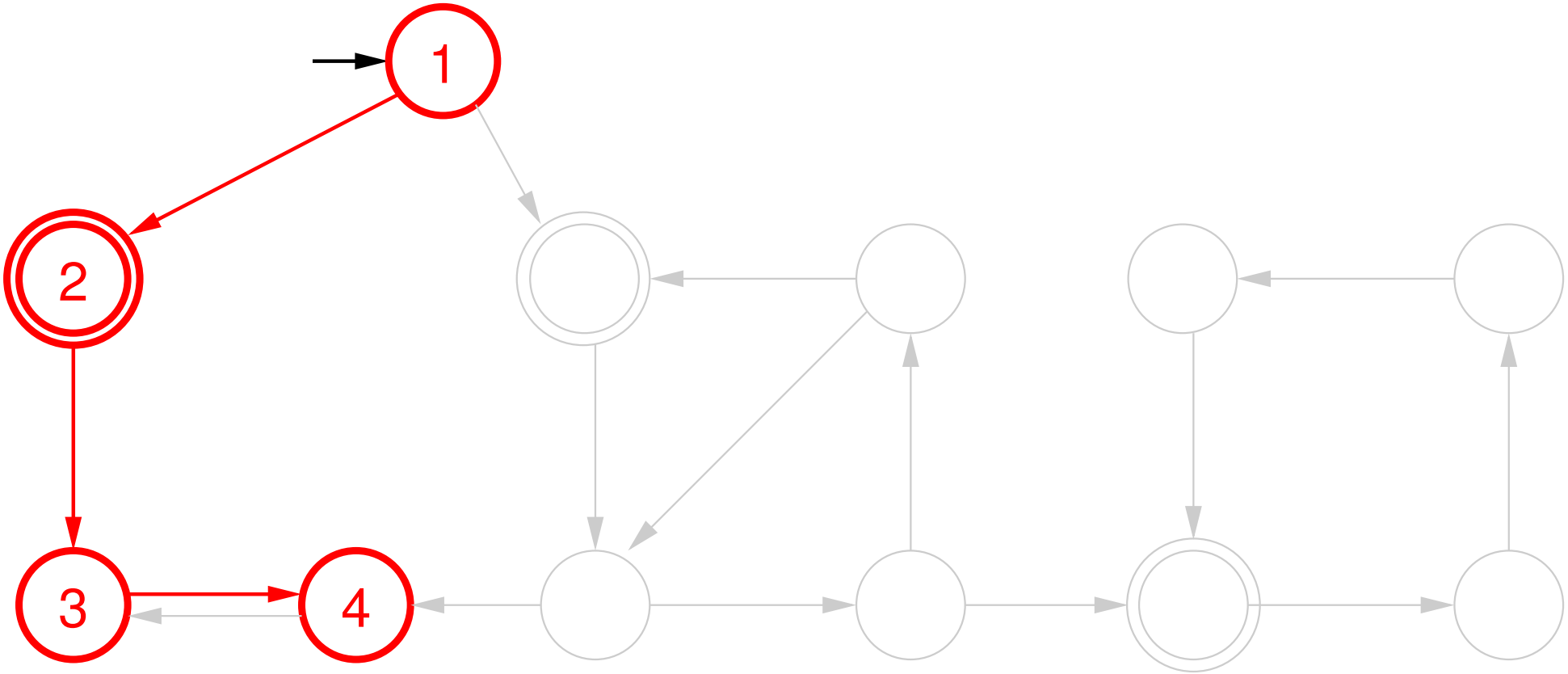
Tiefensuche beginnt beim Anfangszustand und erforscht einen Nachfolger.

Beispiel: Tiefensuche



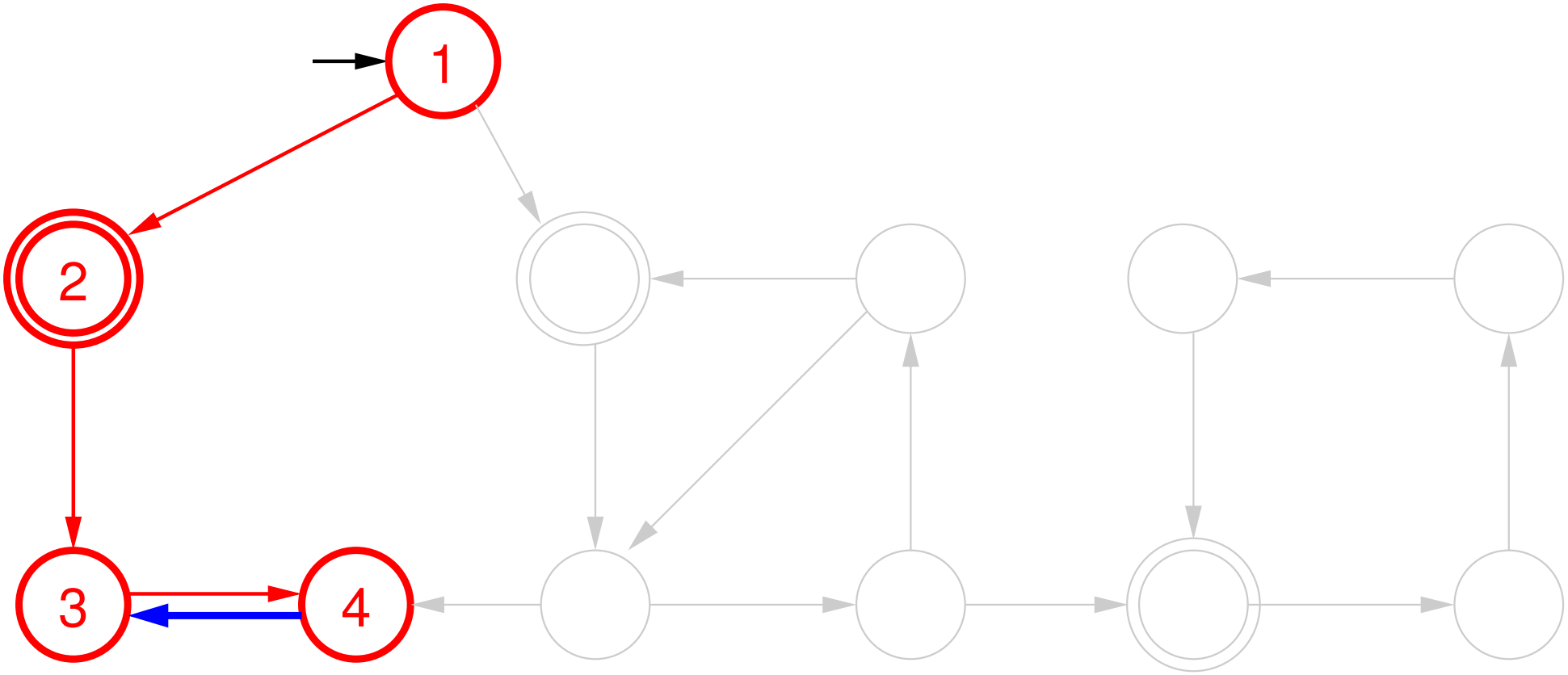
Nachfolgezustand noch nicht bekannt: Rekursiver Aufruf, erhält Nummer 2.

Beispiel: Tiefensuche



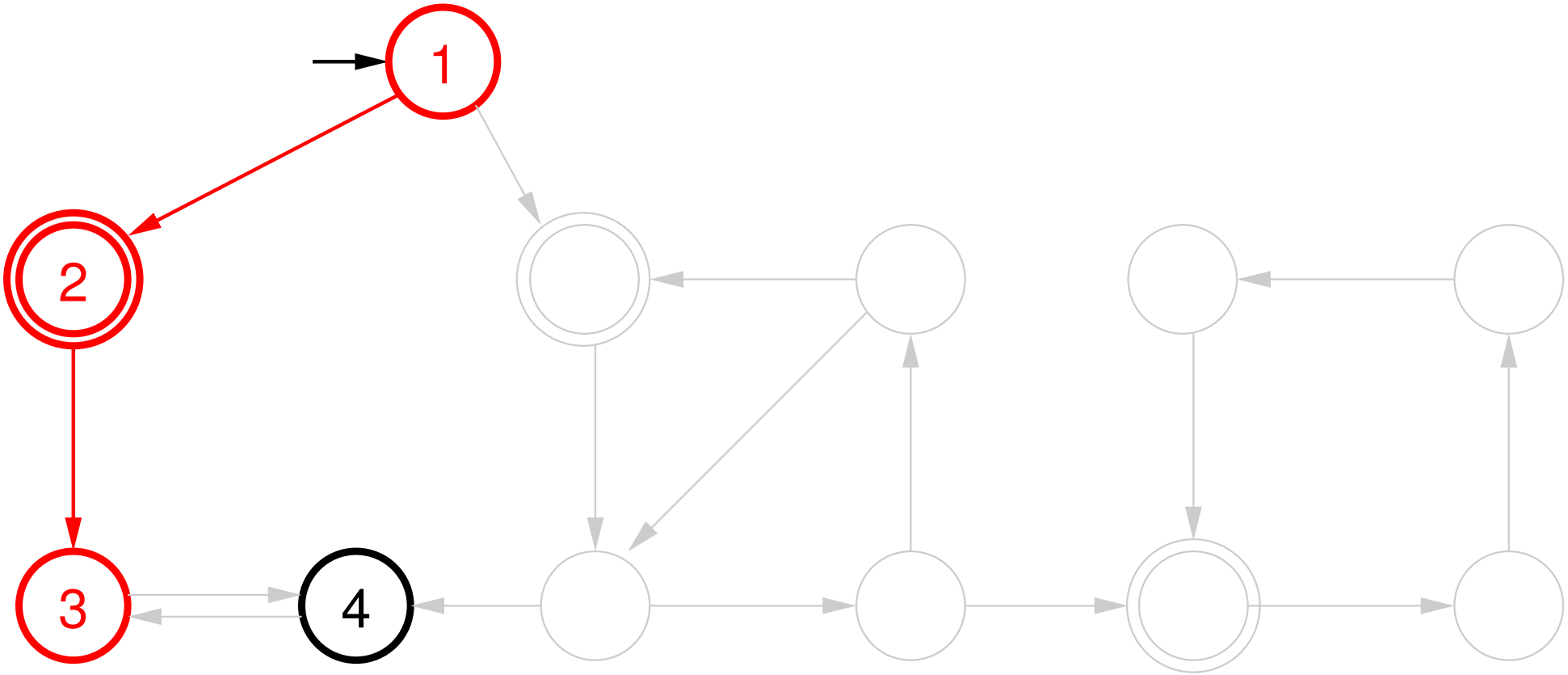
Weitere unbekannte Zustände werden erforscht. . .

Beispiel: Tiefensuche



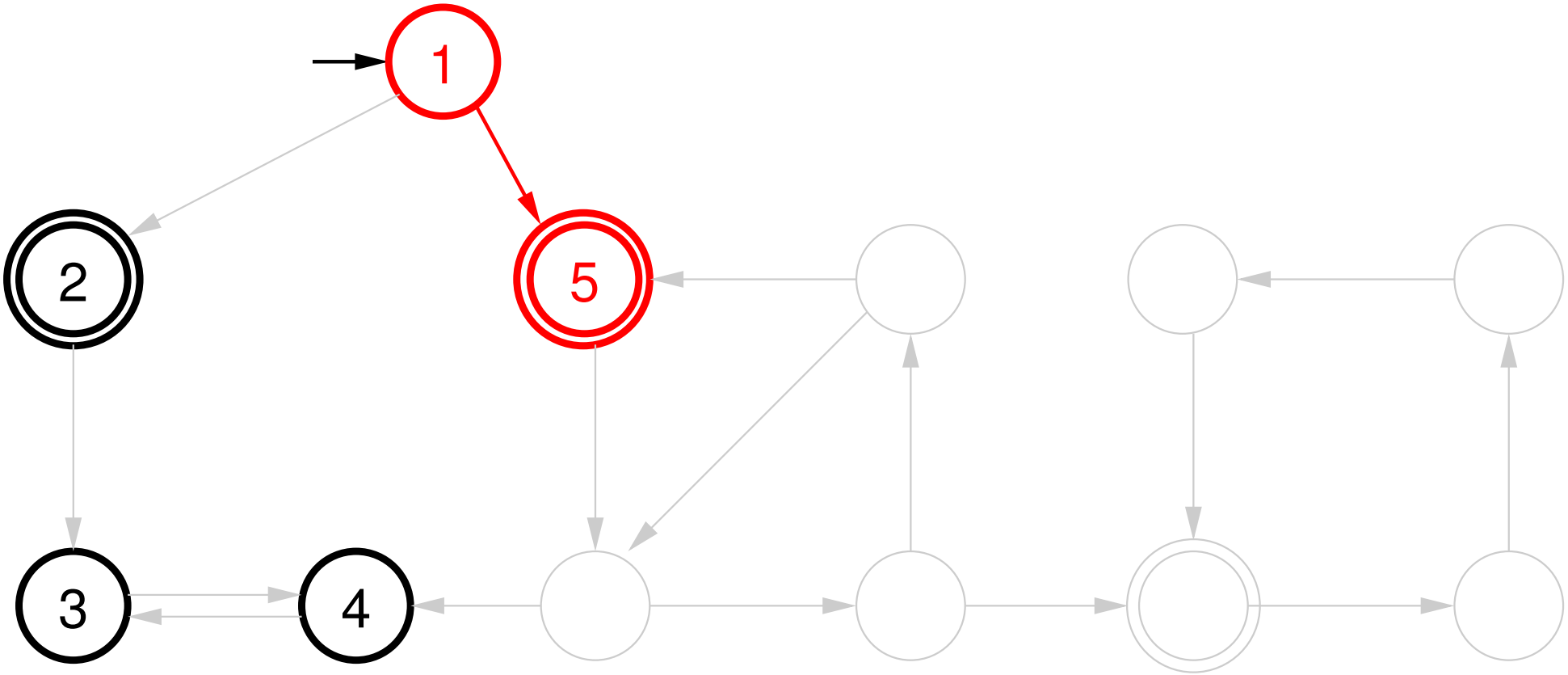
Kante von 4 nach 3: bereits erforschter Zustand, keine erneute Rekursion

Beispiel: Tiefensuche



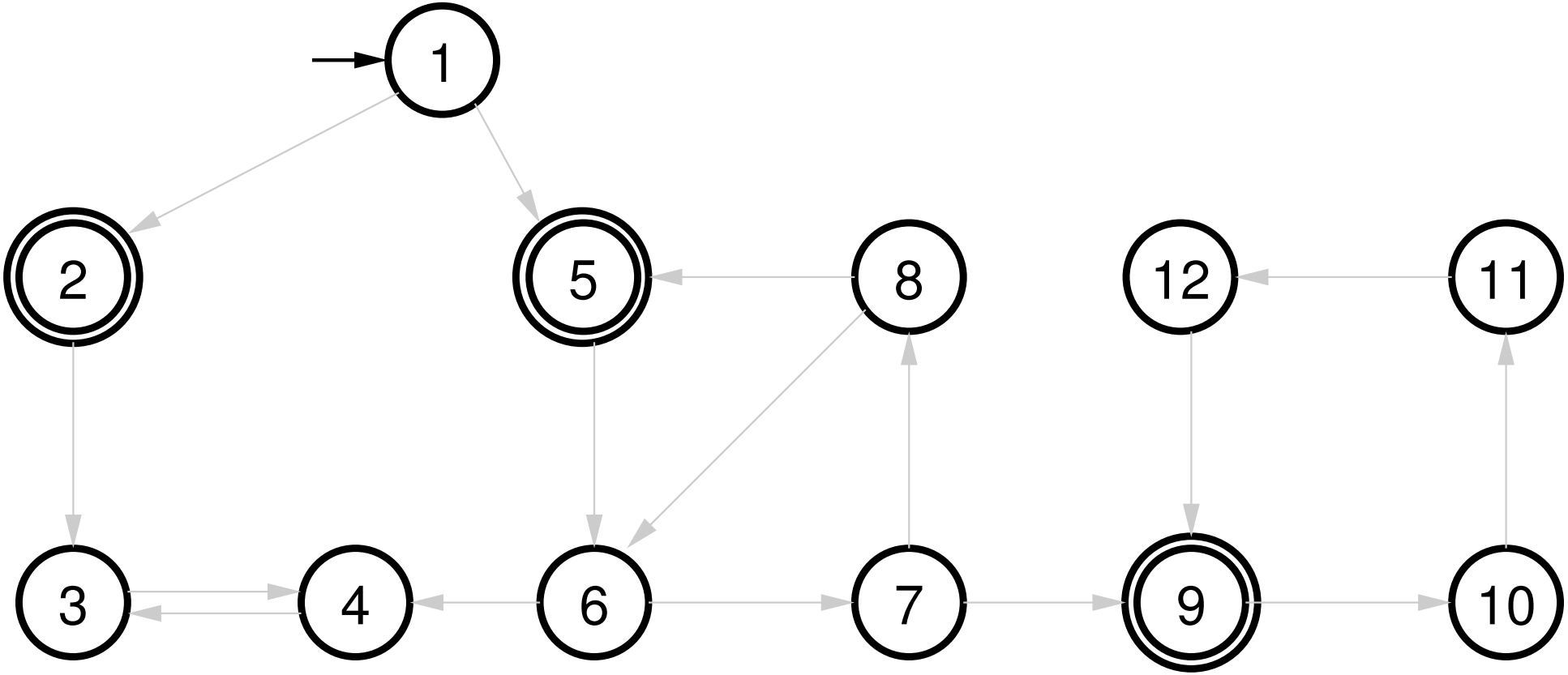
Alle Nachfolger von 4 untersucht; gehe zurück (Backtracking).

Beispiel: Tiefensuche



Backtracking bis zur 1, nächster Nachfolger erhält Nummer 5.

Beispiel: Tiefensuche



Mögliche Nummerierung am Ende der Tiefensuche.

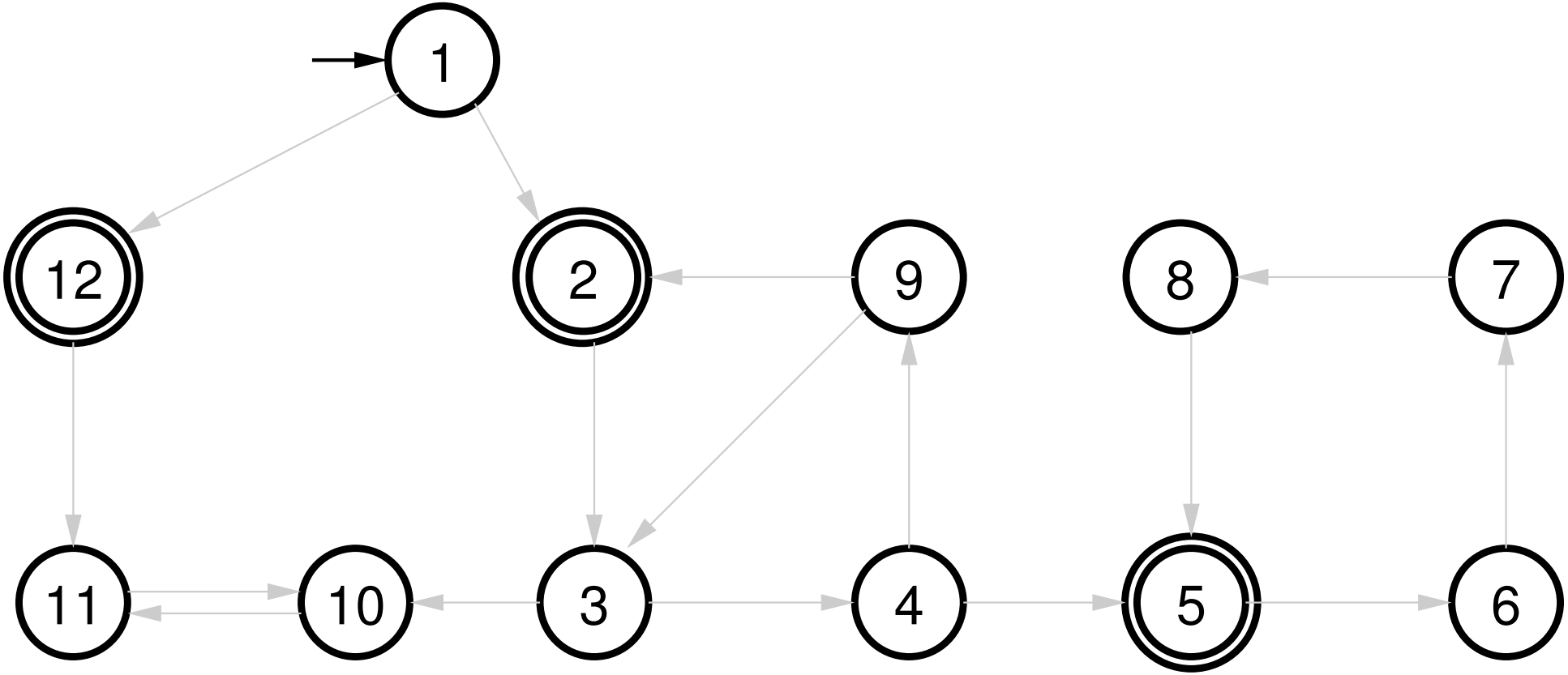
Suchordnung

Hat ein Zustand mehrere Nachfolger, so müssen sie in *irgendeiner* Reihenfolge durchlaufen werden.

Welche DFS-Nummer ein Zustand bekommt, hängt von der verwendeten Suchordnung ab.

Die Suchordnung kann auch beeinflussen, wie schnell ein Gegenbeispiel gefunden wird (wenn eins existiert)!

Beispiel: Suchordnung



Mögliche alternative Nummerierung bei anderer Suchordnung.

Suchordnung

Hat ein Zustand mehrere Nachfolger, so müssen sie in *irgendeiner* Reihenfolge durchlaufen werden.

Welche DFS-Nummer ein Zustand bekommt, hängt von der verwendeten Suchordnung ab.

Die Suchordnung kann auch beeinflussen, wie schnell ein Gegenbeispiel gefunden wird (wenn eins existiert)!

Annahme hier: Suchordnung extern gegeben.

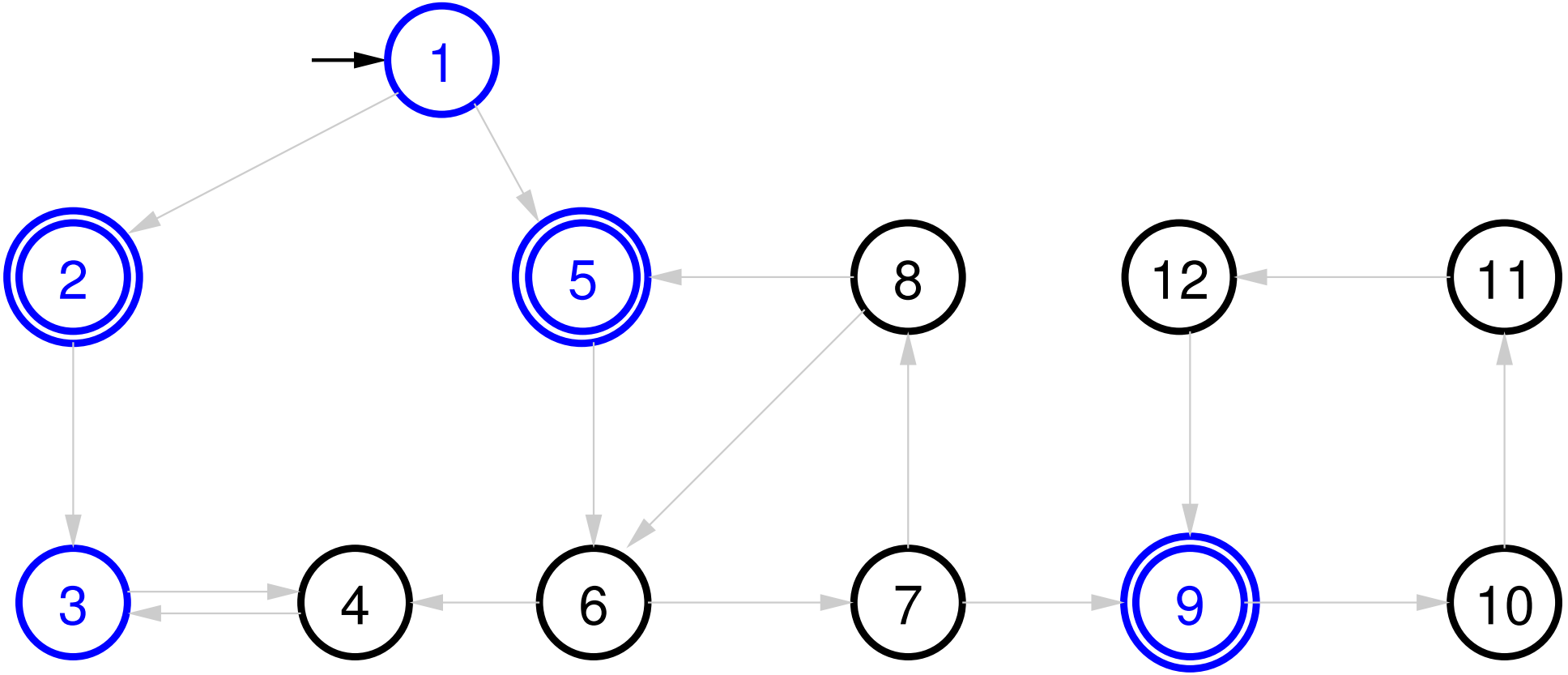
Denkbare Erweiterung: “intelligente” Suchordnung, die zusätzliches Wissen über das Modell miteinbezieht.

Wurzeln

Als Wurzel einer SCC bezeichnen wir denjenigen Zustand innerhalb einer SCC, der bei der Tiefensuche zuerst entdeckt wird.

Bemerkung: Welcher Zustand zur Wurzel wird, hängt ggfs. von der Suchordnung ab!

Beispiel: Suchordnung



Wurzeln (in blau) bei der ursprünglich verwendeten Suchordnung.

Eigenschaften von Wurzeln

(2) Innerhalb einer SCC hat die Wurzel die kleinste DFS-Nummer.

Beweis: offensichtlich

(3) Sei s eine Wurzel und $t \neq s$ ein Zustand mit $s \rightarrow^* t$.
Der Aufruf von `dfs(t)` wird beendet, bevor `dfs(s)` beendet wird.

Beweis: Entweder wurde `dfs(t)` vor `dfs(s)` begonnen. Dann muss $t \not\rightarrow^* s$ gelten, sonst wären s, t in der gleichen SCC, und s könnte nicht ihre Wurzel sein. Also muss `dfs(t)` enden, bevor `dfs(s)` überhaupt beginnen kann.

Falls `dfs(t)` nach `dfs(s)` beginnt, ist t unbesucht, wenn $dfs(s)$ beginnt. Dann wird t von s aus gefunden, und `dfs(t)` wird rekursiv von `dfs(s)` aufgerufen und terminiert daher auch früher.

(4) Innerhalb einer SCC ist die Wurzel der letzte Zustand, von dem man zurückgeht, und in diesem Moment ist die SCC komplett erforscht.

Beweis: folgt unmittelbar aus (3).

Erforschter/aktiver Untergraph

Während des Ablaufs der Tiefensuche unterscheiden wir zwei spezielle Untergraphen von $\mathcal{B}$.

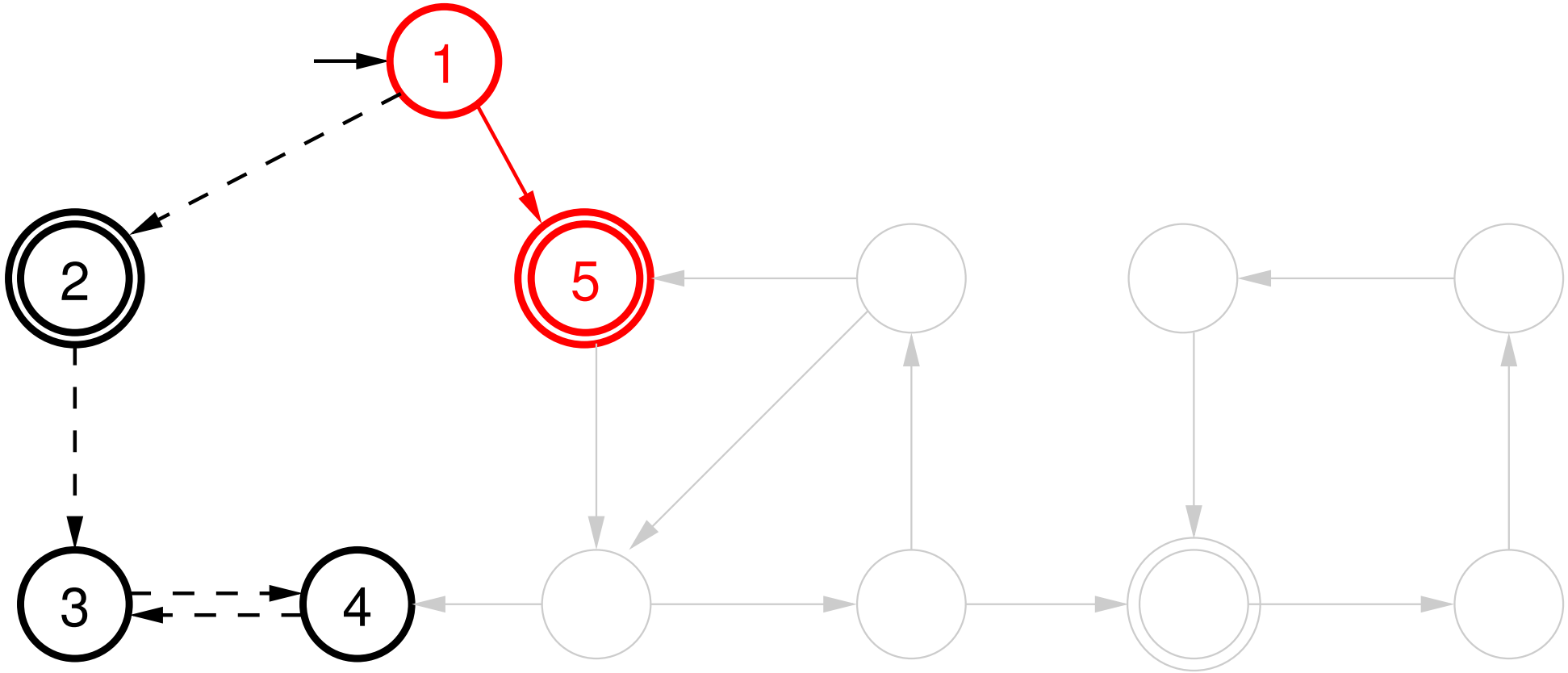
Als erforschten Graphen von $\mathcal{B}$ bezeichnen wir den Untergraphen, der durch die bereits verarbeiteten Transitionen induziert wird (enthält die Knoten im Hash).

Eine SCC *des erforschten Graphen*(!) nennen wir aktiv, wenn der Suchpfad noch mindestens einen ihrer Zustände enthält (dessen dfs-Aufruf noch nicht beendet wurde).

Ein Zustand heißt aktiv, wenn er Teil einer aktiven SCC ist.

Der aktive Graph ist der Teil des erforschten Graphen, der aus den aktiven SCCs besteht.

Beispiel: Erforscher/aktiver Untergraph



In dieser Situation: erforschter Graph in rot und schwarz, aktive SCCs: $\{1\}$ und $\{5\}$, inaktive SCCs $\{2\}$ und $\{3, 4\}$.

Eigenschaften des aktiven Graphen

(5) Eine SCC wird inaktiv in dem Moment, wo man von ihrer Wurzel zurückgeht.

Beweis: Folgt aus (4).

(6) Eine inaktive SCC des erforschten Graphen ist auch eine SCC von $\mathcal{B}$.

Beweis: Folgt unmittelbar aus (4) und (5).

(7) Die Wurzeln des aktiven Graphen sind eine Untersequenz des Suchpfads.

Beweis: Folgt aus (5), da die Wurzel einer aktiven SCC noch auf dem Suchpfad sein muss.

(8) Sei s ein aktiver Zustand und t (mit $t.num \leq s.num$) die Wurzel seiner SCC im aktiven Graphen. Dann gibt es keine aktive Wurzel u mit $t.num < u.num < s.num$.

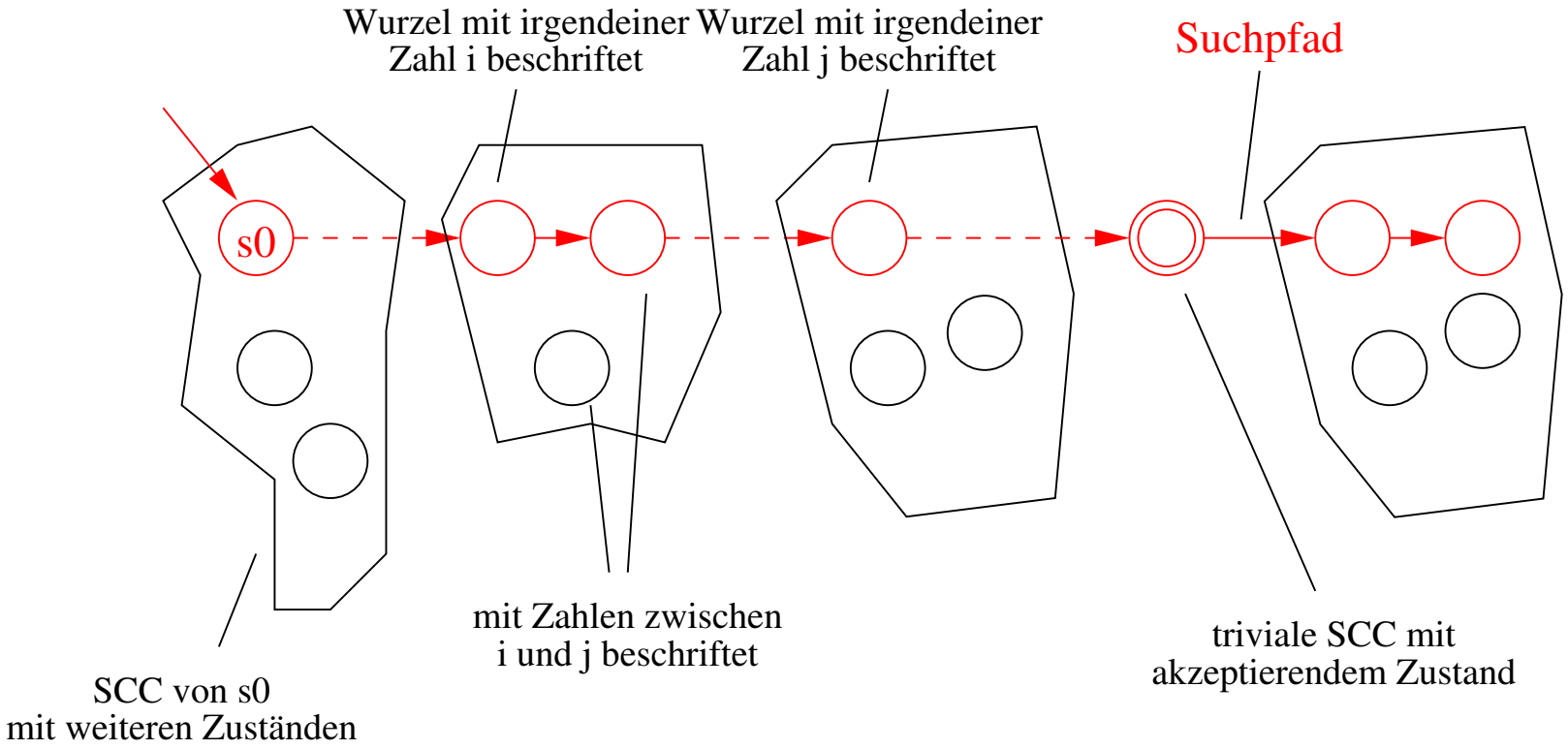
Angenommen, solch eine Wurzel u existiert. Da u aktiv ist, liegt es auf dem Suchstack, ebenso wie t, siehe (5). Dann gilt wegen (1) $t \rightarrow^* u$. Weil `dfs(u)` noch aktiv ist und $u.num < s.num$ gilt, muss s von u aus erreicht worden sein, d.h. $u \rightarrow^* s$. Da s, t in derselben SCC liegen, gilt $s \rightarrow^* t$. Dann aber liegen t, u in derselben SCC und können nicht beide ihre Wurzeln sein.

(9) Gegeben zwei aktive Zustände s und t mit $s.num \leq t.num$. Dann gilt $s \rightarrow^* t$.

Seien s', t' die (aktiven) Wurzeln zu s und t. Wegen (8) gilt $s'.num \leq t'.num$, also wegen (1) $s' \rightarrow^* t'$, und daher $s \rightarrow^* t$.

Veranschaulichung

Aus den gezeigten Eigenschaften ergibt sich, dass der aktive Graph und seine SCCs immer in etwa die untenstehende Form haben:



Eigenschaften unseres Leerheitstest-Algorithmus

Lineare Laufzeit in $|S| + |\delta|$.

Erforscht $\mathcal{B}$ mittels Tiefensuche; meldet ein Gegenbeispiel, *sobald der erforschte Graph eines enthält*. (*)

Für jeden entdeckten Zustand s wird $\mathsf{succ}(s)$ nur einmal aufgerufen.

$\rightarrow$ Zeitersparnis, da die Anzahl der succ-Aufrufe in der Praxis die Laufzeit dominiert.

Zusätzlicher Speicherbedarf

Kellerspeicher (Stack) W mit Elementen der Form (s, C), wobei

s die Wurzel einer aktiven SCC ist;

C die Menge der Zustände in der SCC von s.

(Implementierung von C: verkettete Liste, zusätzlicher Zeiger für jeden Zustand.)

Ein Bit pro Zustand, das anzeigt, ob der Zustand aktiv ist.

Arbeitsweise unseres Algorithmus

Aktionen des Algorithmus:

- Initialisierung
- Verarbeitung einer neuen Kante (zu neuem oder altem Zustand)
- Backtracking

Bei jeder Aktion wird:

- der Inhalt von *W* und der Aktiv-Bits aktualisiert;
- geprüft, ob der erforschte Graph jetzt ein Gegenbeispiel enthält.

Initialisierung

Erforschter Graph besteht nur aus dem Anfangszustand s_0, keine Kanten.

Einziges Element von W: das Tupel $(s_0, \{s_0\})$

s_0 ist aktiv.

Verarbeitung neuer Kanten

Entdeckung einer Kante $s \to t$. Fallunterscheidung:

Fall 1: t wurde vorher noch nie gesehen:

Der erforschte Graph wird um den Zustand t und die Kante $s \to t$ erweitert.

t ist aktiv und bildet eine triviale SCC im aktiven Graphen.

Erweiterung von W um $(t, \{t\})$.

Rekursiver Aufruf der Tiefensuche auf t.

Verarbeitung neuer Kanten

Entdeckung einer Kante $s \to t$. Fallunterscheidung:

Fall 2: t wurde bereits gesehen und ist inaktiv.

> t ist inaktiv, also wurde seine SCC komplett untersucht, siehe (4) und (5). s, t müssen daher zu unterschiedlichen SCCs gehören, insbesondere kann nicht $t \to^* s$ gelten. Wegen (2) sind auch alle Nachfolger von t bereits untersucht worden und haben wegen (*) keine Schleife mit akzeptierenden Zustand enthalten. Die Kante $s \to^* t$ ist also nicht Teil eines Gegenbeispiels und kann ignoriert werden.
>
> Kein rekursiver Aufruf, W und die Aktiv-Bits unverändert.

Verarbeitung neuer Kanten

Entdeckung einer Kante $s \to t$. Fallunterscheidung:

Fall 3: t wurde bereits gesehen und ist aktiv, Nummer von t höher als die von s.

Wegen (9) wissen wir bereits, dass $s \to^* t$ gilt, die SCCs des aktiven Graphen ändern sich daher nicht, und ein Gegenbeispiel kann so auch nicht entstehen. Die Kante wird also ignoriert.

Kein rekursiver Aufruf, W und die Aktiv-Bits unverändert.

Verarbeitung neuer Kanten

Entdeckung einer Kante $s \to t$. Fallunterscheidung:

Fall 4: t wurde bereits gesehen und ist aktiv, Nummer von t gleich der von s.

Dann ist $s = t$.

Ein Gegenbeispiel ist genau dann entdeckt, falls s akzeptierend.

Ansonsten: kein rekursiver Aufruf, W und die Aktiv-Bits unverändert.

Verarbeitung neuer Kanten

Entdeckung einer Kante $s \to t$. Fallunterscheidung:

Fall 5: t wurde bereits gesehen und ist aktiv, Nummer von t niedriger als von s.

> Dann gilt wegen (9) $t \to^* s$. Also gehören s, t zur selben SCC. Sei u mit $u.num \leq t.num$ die zu t gehörende Wurzel. Da s das neueste Element auf dem Suchpfad ist, können wir wegen (1) schließen, dass alle bisherigen SCCs in W von u abwärts verschmolzen werden müssen.
>
> Die richtige Wurzel u finden wir aufgrund von (8), indem wir Elemente von W entfernen, bis wir auf eine Wurzel kommen, deren Zahl kleiner gleich $t.num$ ist.
>
> Ein Gegenbeispiel entsteht genau dann, wenn zur Verschmelzung eine bisher triviale SCC mit akzeptierendem Zustand gehört. Dazu müssen wir beim “Abräumen” von W nur prüfen, ob eine akzeptierende Wurzel dabei ist.

Backtracking

Alle Nachfolger von s wurden untersucht.

Fall 1: s ist eine Wurzel.

Dann wird s sowie die gesamte SCC inaktiv, siehe (5).

W wird um sein oberstes Element verkürzt.

Fall 2: s ist keine Wurzel.

Dann ist die Wurzel der SCC, zu der s gehört, noch aktiv.

W und die Aktiv-Bits bleiben unverändert.

Fertiger Algorithmus

```
nr = 0; hash = {}; W = {}; dfs(s0); exit;

dfs(s) {
   add s to hash; s.aktiv = true;
   nr = nr+1; s.num = nr;
   push (s,{s}) onto W;
   for (t in succ(s)) {
      if (t not yet in hash) { dfs(t); }
      else if (t.aktiv) {
         D = {};
         repeat
            pop (u,C) from W;
            if u is accepting { report success; halt; }
            merge C into D;
         until u.num <= t.num;
         push (u,D) onto W;
   }  }
   if s is the top root in W {
      pop (s,C) from W;
      for all t in C { t.aktiv = false; }
   }
}
```

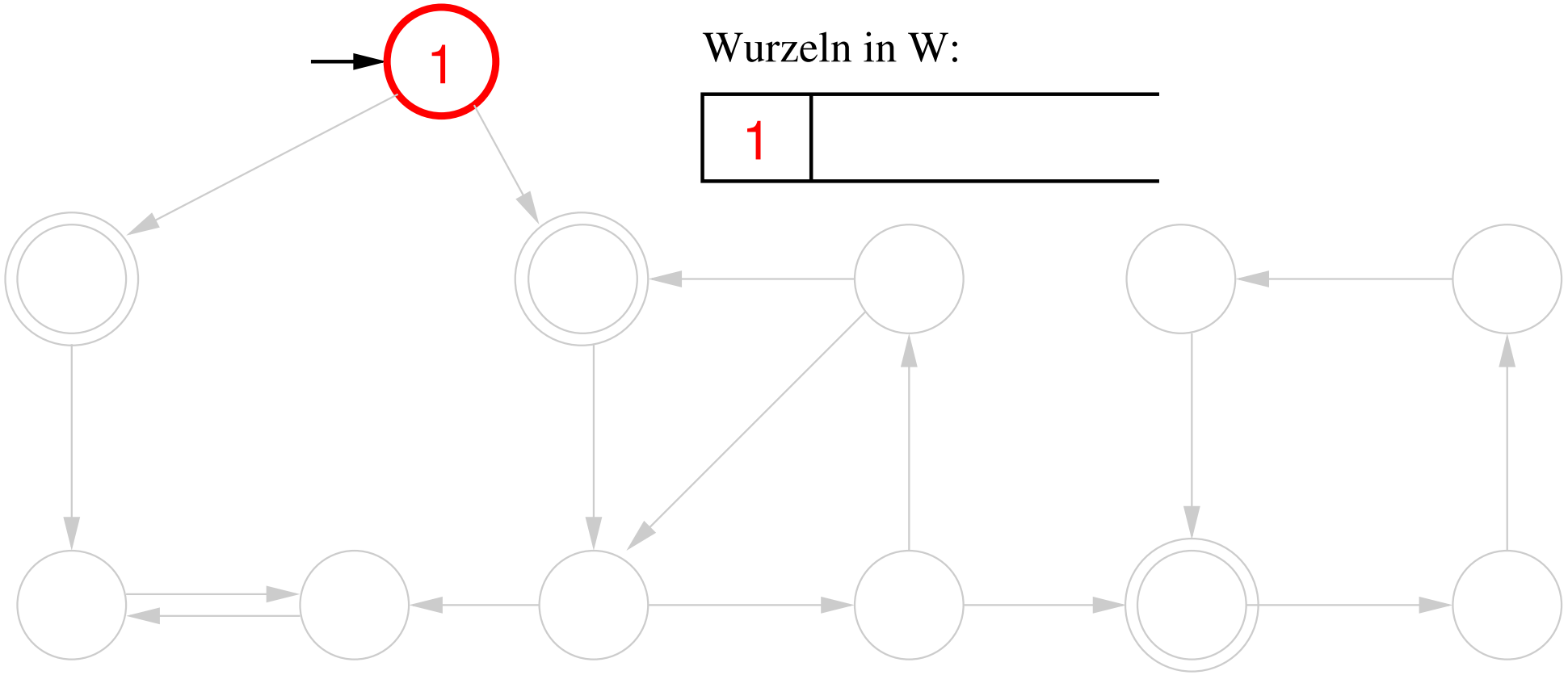
Bemerkungen zum Algorithmus

Fälle 3 bis 5 bei der Behandlung von Kanten werden einheitlich durch die repeat-until-Schleife geregelt.

Ein gefundenes Gegenbeispiel wird durch `report success` angezeigt.

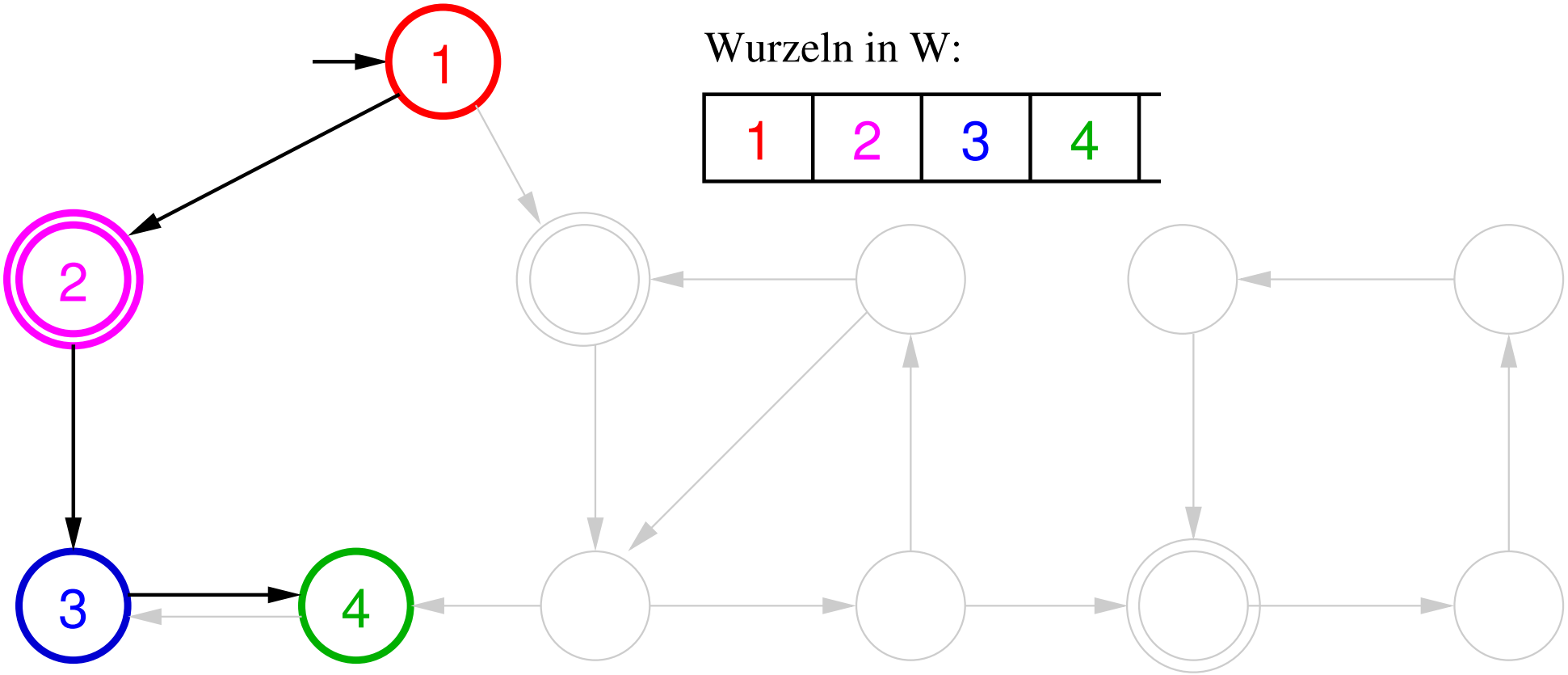
Terminiert der Aufruf `dfs(s0)`, so gibt es kein Gegenbeispiel.

Beispiel: Ablauf des Algorithmus



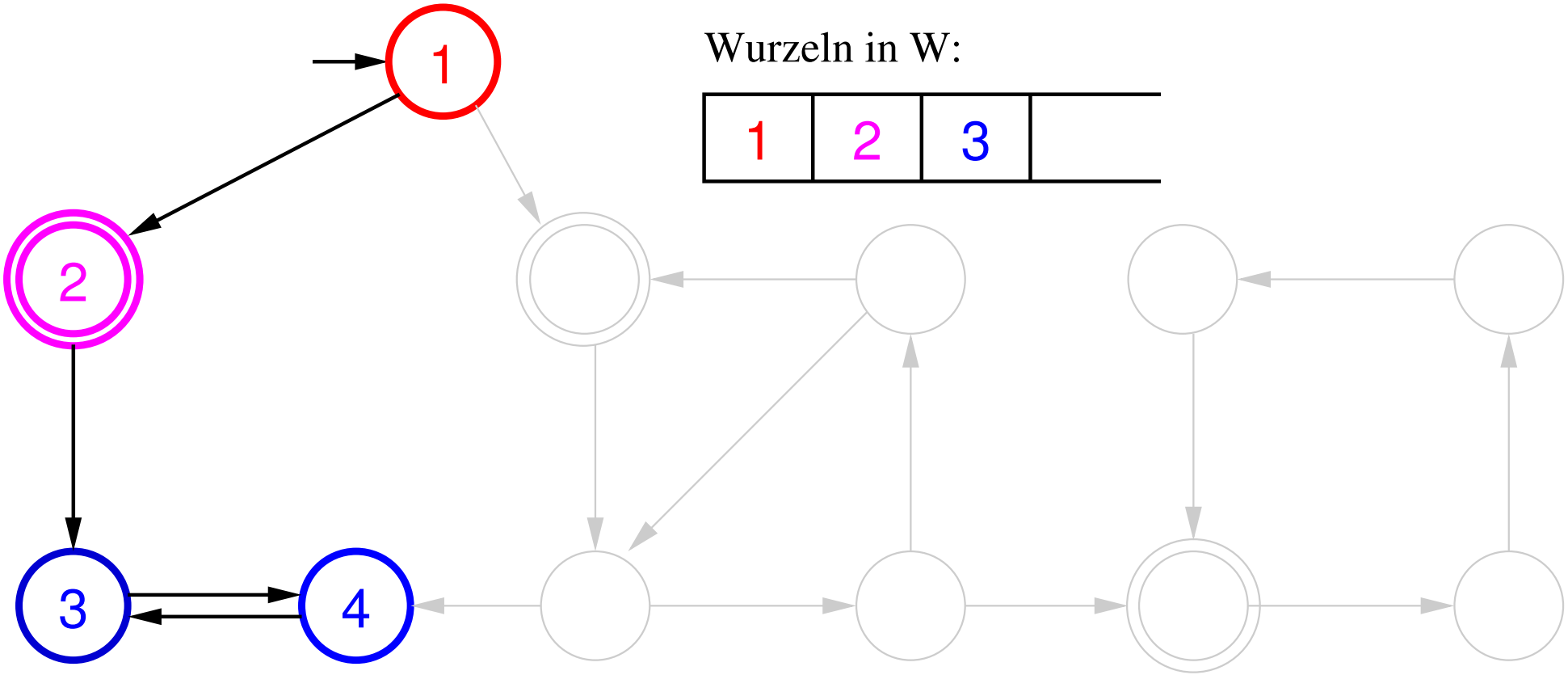
Situation zu Beginn des Ablaufs: nur s_0 erforscht.

Beispiel: Ablauf des Algorithmus



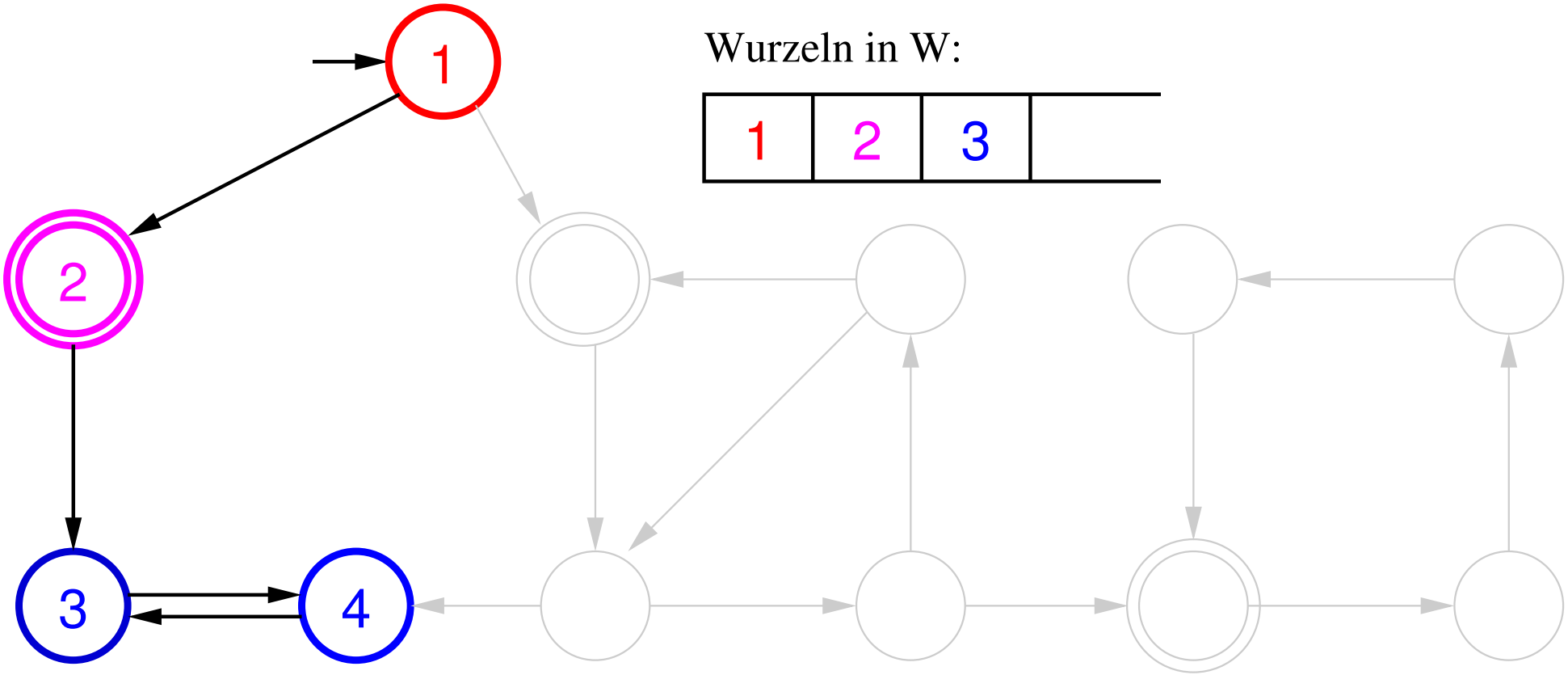
Situation nach Entdeckung dreier Kanten.

Beispiel: Ablauf des Algorithmus



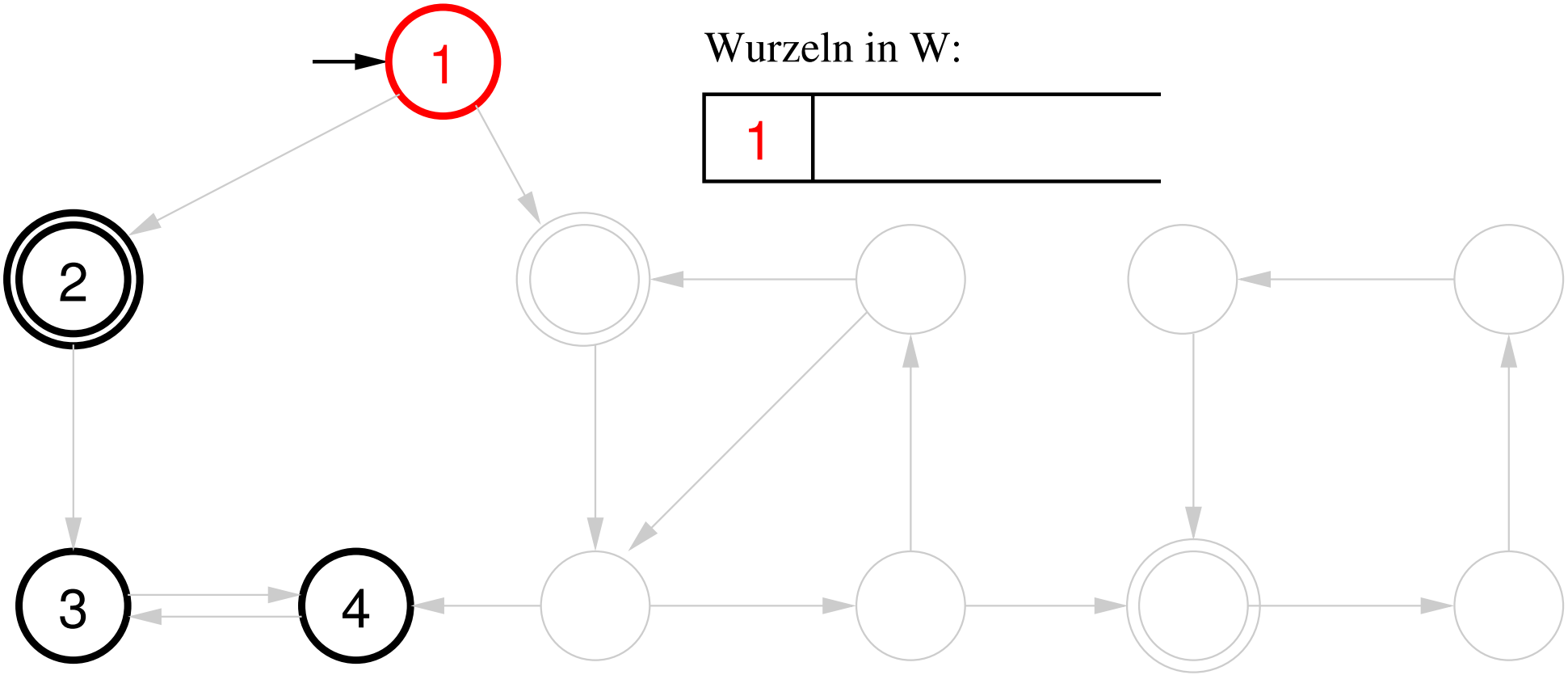
Kante $4 \rightarrow 3$ führt zur Verschmelzung zweier Komponenten.

Beispiel: Ablauf des Algorithmus



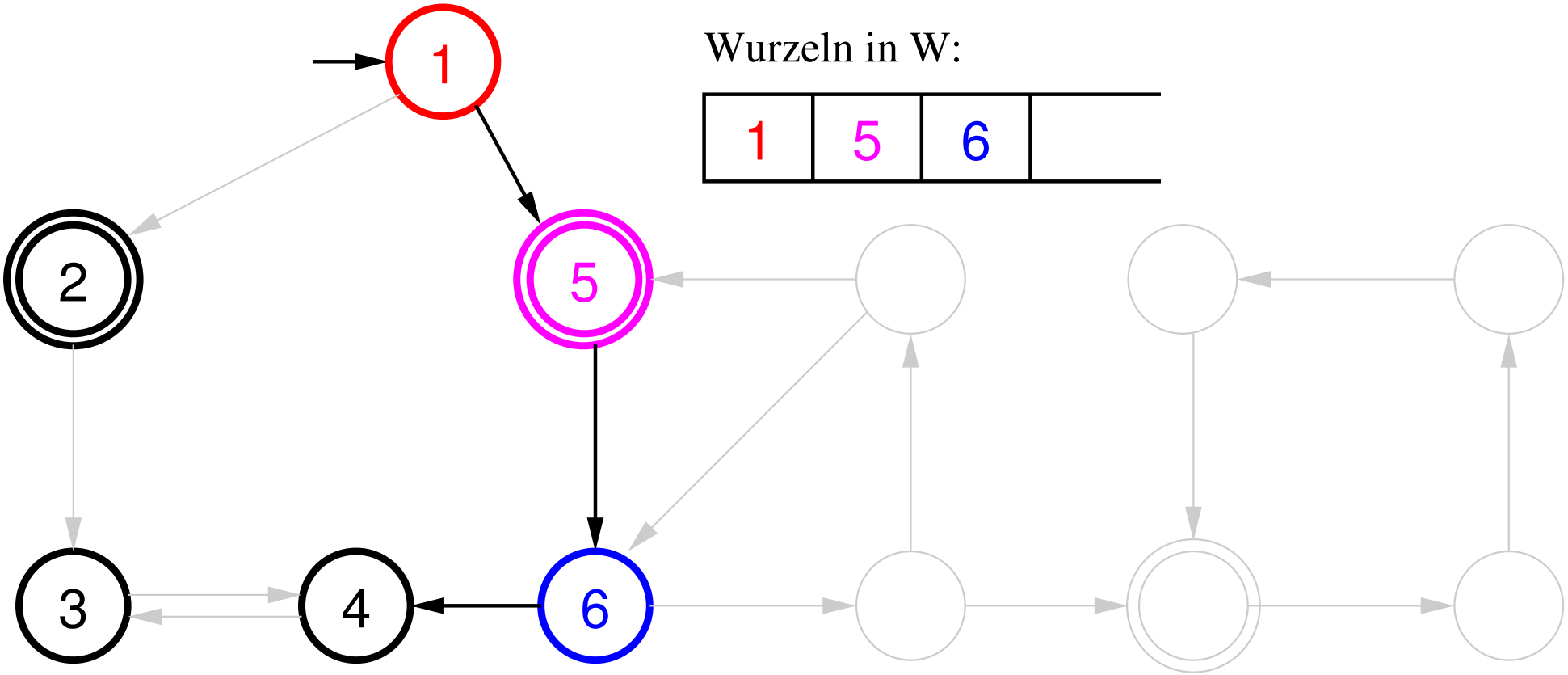
(Mengenkomponente von *W* durch gleiche Färbung von Knoten angedeutet.)

Beispiel: Ablauf des Algorithmus



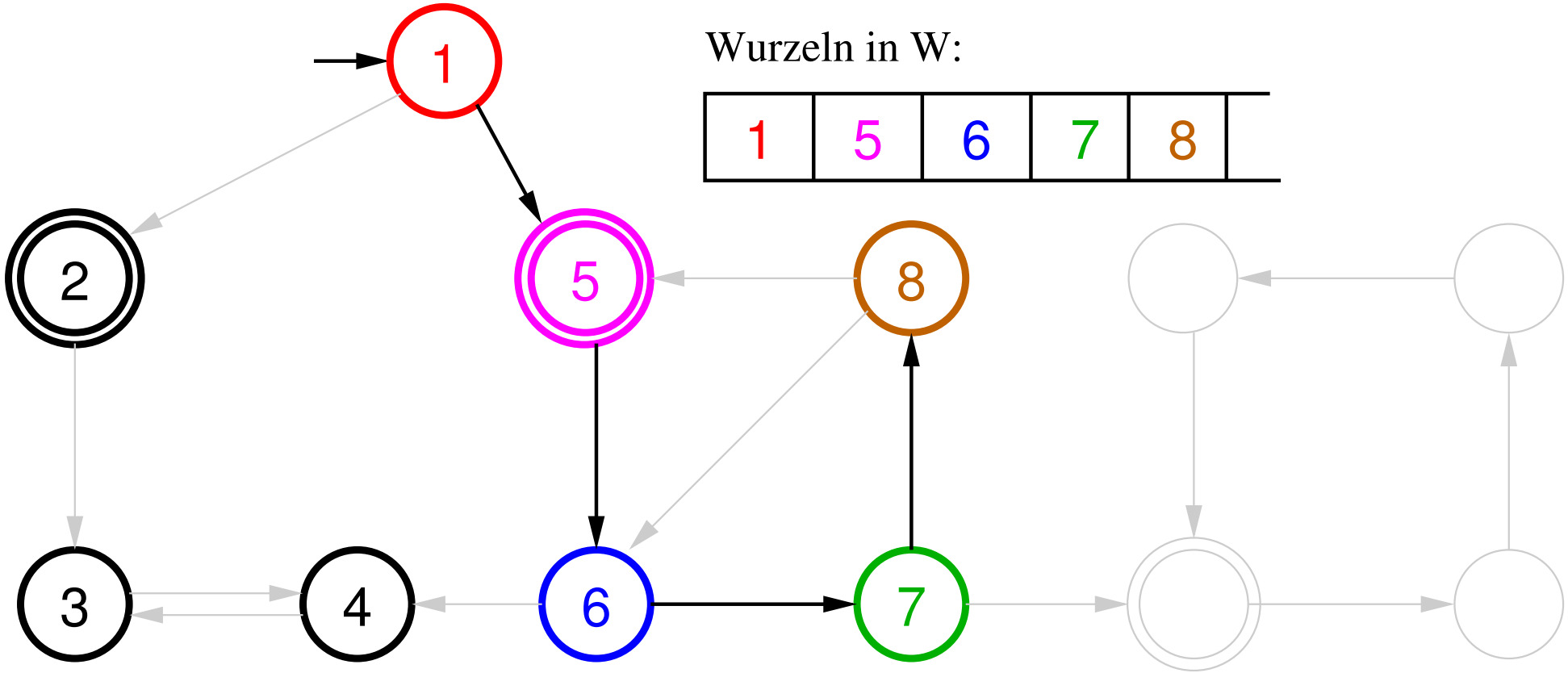
Durch Backtracking werden 2, 3 und 4 inaktiv (schwarz dargestellt).

Beispiel: Ablauf des Algorithmus



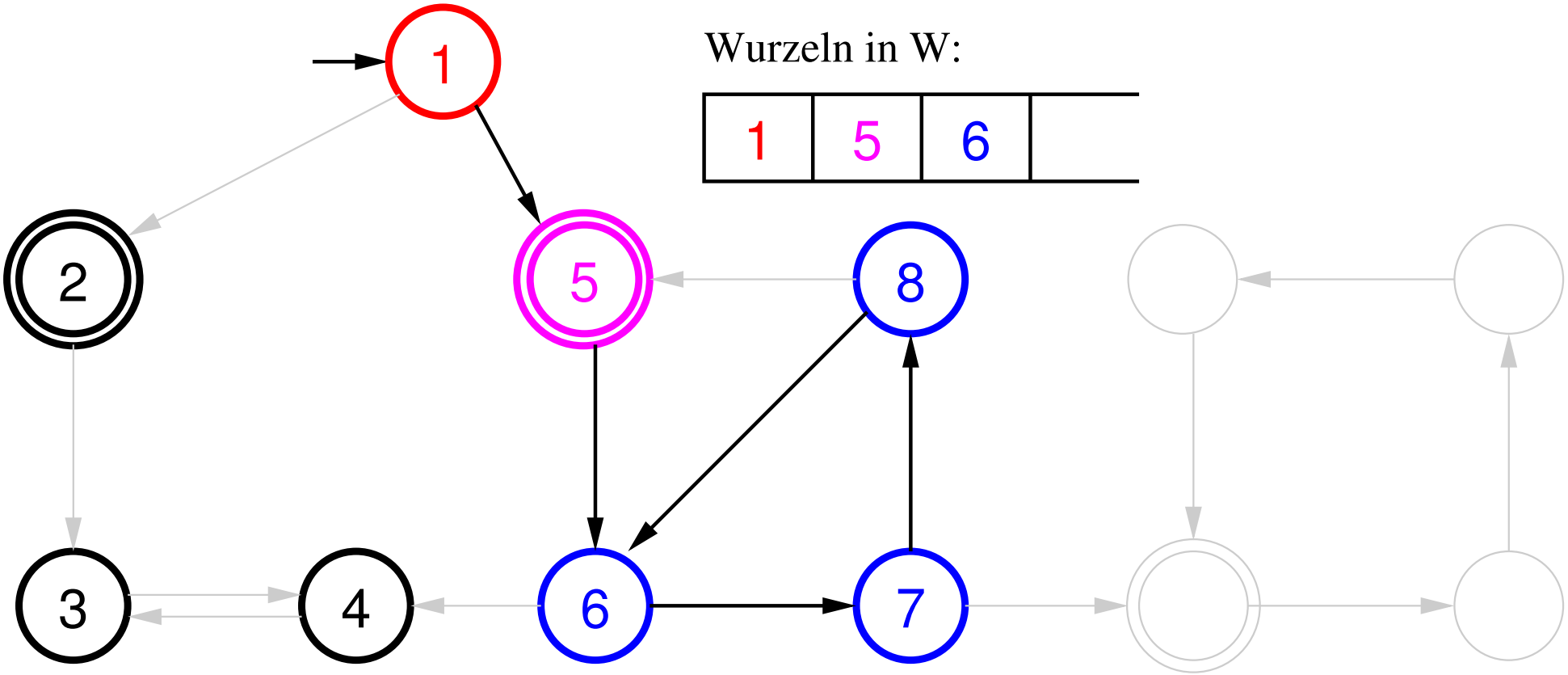
Kante $6 \rightarrow 4$ entspricht Fall 2, kann ignoriert werden.

Beispiel: Ablauf des Algorithmus



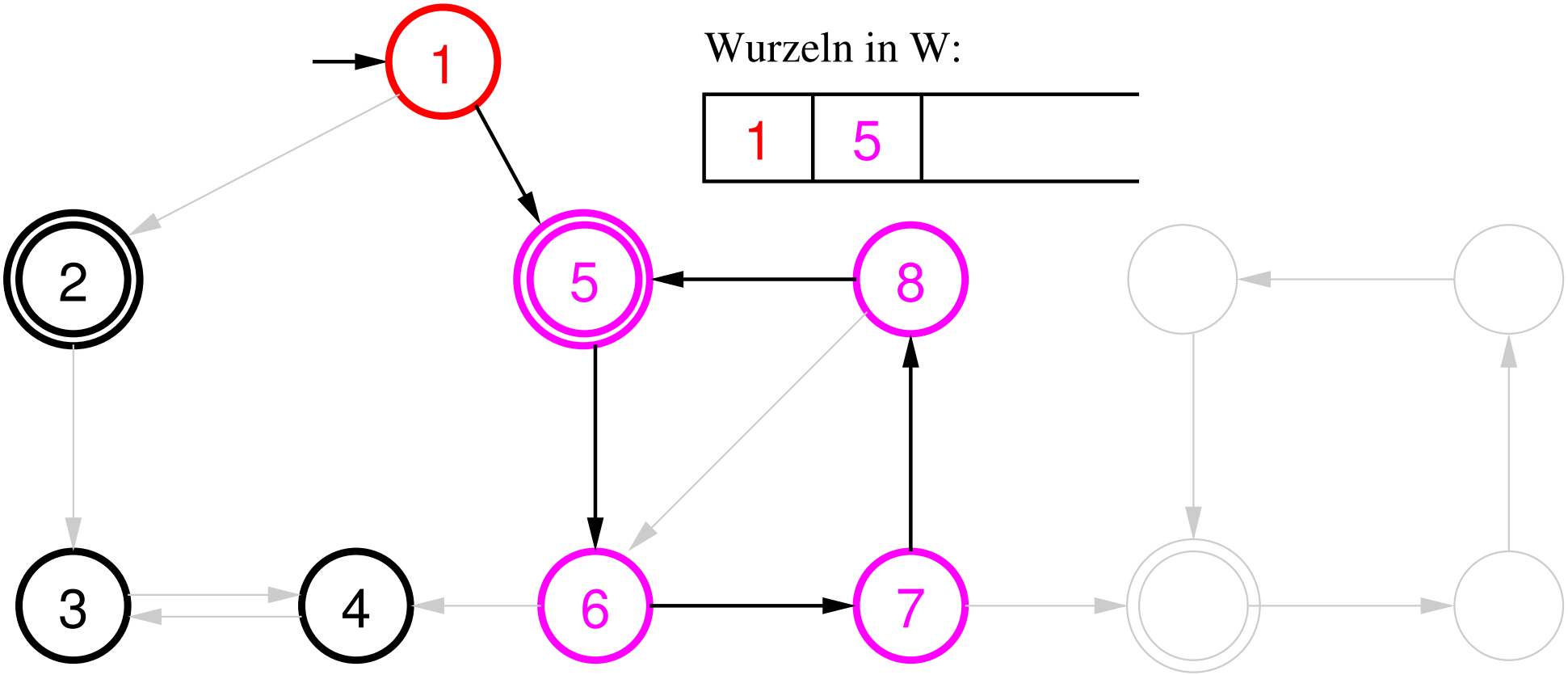
Situation bei Erreichen von 8.

Beispiel: Ablauf des Algorithmus



Kante $8 \rightarrow 6$ führt zu Verschmelzung ohne Abbruch.

Beispiel: Ablauf des Algorithmus



Kante $8 \rightarrow 5$: Gegenbeispiels entdeckt, weil Wurzel 5 akzeptierend ist.

Erweiterung auf verallgemeinerte BA

Sei $\mathcal{V}$ ein VBA mit n Akzeptanzmengen $F_1, \ldots, F_n$.

$\mathcal{L}(\mathcal{V})$ ist nicht leer gdw. es eine nicht-triviale SCC gibt, die mit jedem F_i ($1 \leq i \leq n$) einen nicht-leeren Schnitt hat.

Wir beschriften jeden Zustand s mit der Indexmenge der Akzeptanzmengen, geschrieben M_s, in denen er enthalten ist. (Ist z.B. s in F_1 und F_3, aber in keiner anderen Akzeptanzmenge, so ist $M_s = \{1, 3\}$.)

Die Elemente im Wurzelstack W werden um eine dritte Komponente erweitert, eine Indexmenge, d.h. eine Teilmenge von $\{1, \ldots, n\}$.

Enthält W einen Eintrag (s, C, M), so sind s und C wie zuvor, und $M = \bigcup_{t \in C} M_t$.

Werden zwei SCCs verschmolzen, so vereinigt man ihre Indexmengen.

Ein Gegenbeispiel ist gefunden, sobald bei der Verschmelzung die Menge $\{1, \ldots, n\}$ entsteht.

Bei “kleiner” Indexmenge lassen sich die benötigten Operationen mit Bitvektoren implementieren (konstante Zeit).

Modifikation für SCCs

Der Algorithmus kann auch verwendet werden, um den BA (oder irgendeinen gerichteten Graphen) in seine SCCs zu partitionieren.

Dazu unterlässt man einfach den Akzeptanz-Test beim Verschmelzen zweier SCCs.

Eine fertig berechnete SCC kann ausgegeben werden, sobald man von der Wurzel zurückgeht.

Teil 7: Komplexität und Ausdruckskraft von LTL

Vorschau

Im Folgenden werden wir die Ausdruckskraft von LTL auf zwei Weisen charakterisieren.

1. Durch die Komplexität des Berechnungsproblems:

   LTL-Model-Checking kann prinzipiell jedes Problem kodieren, das mit polynomiellem Platz lösbar ist.

2. Durch sprachtheoretische Betrachtung:

   Jede LTL-Eigenschaft lässt sich als Schnitt einer Sicherheits- und einer Lebendigkeits-Eigenschaft verstehen.

Komplexität von LTL-Model-Checking

Gegeben $\mathcal{K}$ und ϕ, hat der entstehende Schnitt-Automat die Größe $\mathcal{O}(|\mathcal{K}| \cdot 2^{|\phi|})$.

Als muss man (im schlimmsten Fall) zur Erforschung des Büchi-Automaten exponenzielle Zeit aufwenden.

Geht das besser?

Antwort: Nein (vermutlich).

Das Model-Checking-Problem für LTL (d.h., gegeben eine Kripke-Struktur $\mathcal{K}$ und eine Formel ϕ, gilt $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$) ist PSPACE-vollständig. (Sistla, Clarke, 1985)

Anmerkung: Es ist nicht bekannt, ob die Inklusion $\mathsf{P} \subset \mathsf{PSPACE}$ echt ist.

LTL-Model-Checking ist in NPSPACE=PSPACE

Das Kreuzprodukt hat $\mathcal{O}(|\mathcal{K}| \cdot 2^{|\phi|})$ Zustände, also genügen $\mathcal{O}(\log |\mathcal{K}| + |\phi|)$ Bits, um einen Zustand zu speichern.

Ein akzeptierender Pfad besteht aus einem Anfangsstück und einer Schleife; wir benennen deren Längen mit ℓ bzw. m. Wir können annehmen, dass ℓ und m nicht größer als die Anzahl der Zustände sind.

Eine nichtdeterministische Maschine kann ℓ und m zufällig wählen, ℓ Schritte ausführen und prüfen, ob der erreichte Zustand ein Akzeptanzzustand ist. Falls ja, merkt sie sich ihn und führt m Schritte aus. Falls nun der gemerkte Zustand erreicht ist, existiert ein akzeptierender Pfad.

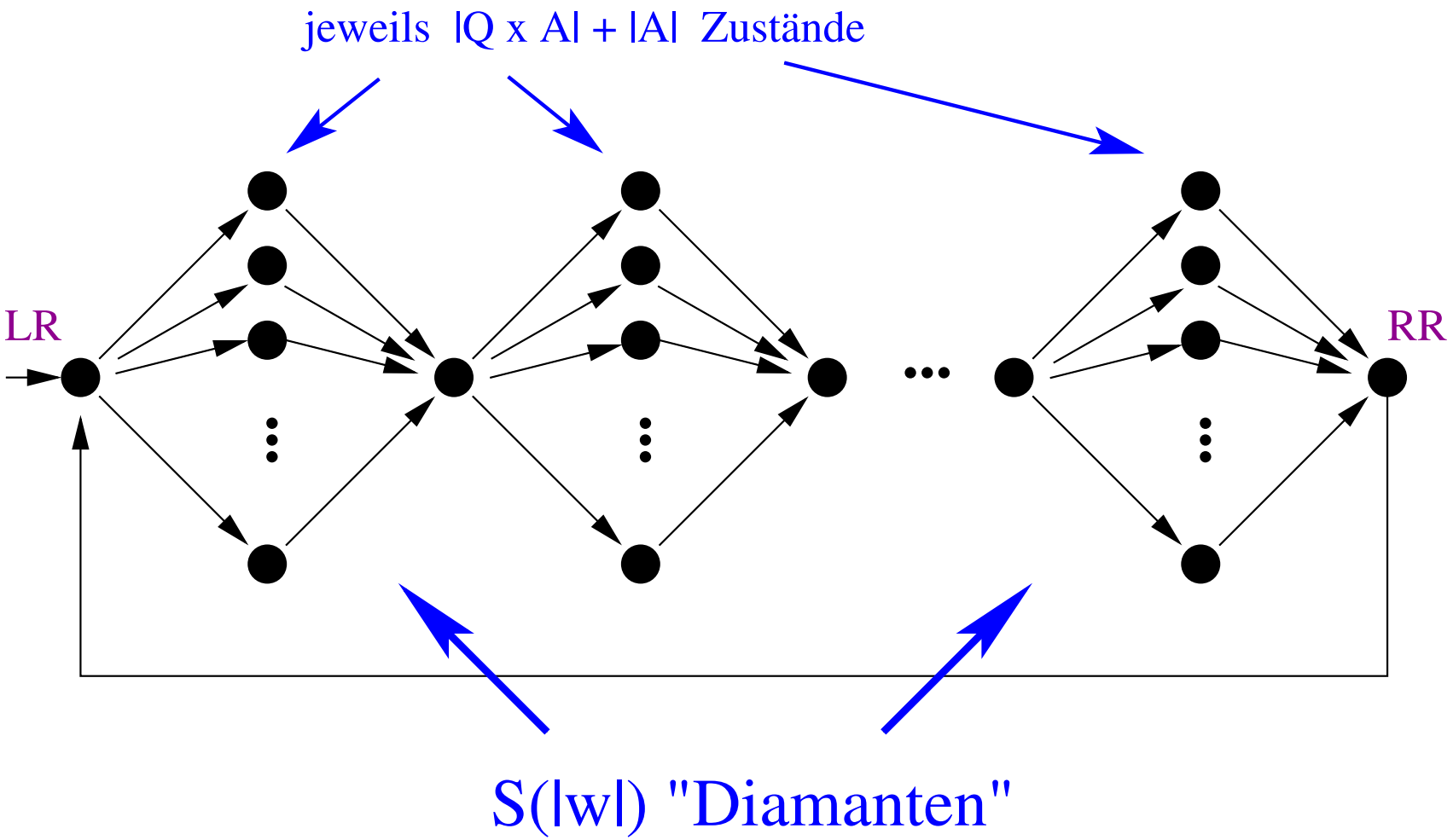
Dabei muss nie das gesamte Kreuzprodukt konstruiert werden, es reicht jeweils, aus dem aktuellen Zustand irgendeinen Folgezustand zu konstruieren.

LTL-Model-Checking ist PSPACE-hart

Das kanonische PSPACE-harte Problem ist wie folgt: Gegeben sei eine deterministische Turing-Maschine $\mathcal{M} = (Q, A, \delta, q_0, q_f, \perp)$, die durch Übergang in q_f akzeptiert, und eine Eingabe $w \in A^*$, wobei der Platzverbrauch von $\mathcal{M}$ durch ein Polynom $S(|w|)$ beschränkt ist und $\perp$ das Leerzeichen darstellt. Akzeptiert $\mathcal{M}$ die Eingabe w?

Wir zeigen, dass sich obiges Problem polynomiell auf das LTL-Problem reduzieren lässt, d.h. gegeben $\mathcal{M}$ und w konstruieren wir $\mathcal{K}$ und ϕ, so dass $\mathcal{K} \models \phi$ gdw. $\mathcal{M}$ akzeptiert w.

Das Schema für $\mathcal{K}$ ist auf der nächsten Folie zu sehen.


jeweils |Q x A| + |A| Zustände
LR
RR
S(|w|) "Diamanten"

Die Menge der Grundaussagen sei $\{LR, RR\} \cup A \times Q \times A$.

Es gelten LR und RR jeweils ausschließlich im ganz linken bzw. ganz rechten Zustand.

Die Anzahl der “Diamanten” sei $S(|w|)$, jeder von diesen stellt eine Stelle auf dem Band von $\mathcal{M}$ dar. Jeder Diamant enthält in der Mitte $|Q \times A \cup A|$ Zustände, in jedem von diesen gelte jeweils ein unterschiedliches Element aus $Q \times A \cup A$, so dass durch Auswahl eines Zustandes der Inhalt der Stelle auf dem Band dargestellt werden kann:

> Grundaussage (q, a) bedeutet, dass das Band an dieser Stelle ein a enthält, die Maschine sich im Zustand q und der Lesekopf sich über dieser Stelle befindet.
>
> Grundaussage a bedeutet, dass das Band an dieser Stelle ein a enthält und der Lesekopf anderswo steht.

Jeder Durchlauf vom LR- zum RR-Zustand soll einen Schritt von $\mathcal{M}$ simulieren.

Es sei $\phi \equiv (\phi_1 \wedge \phi_2) \rightarrow \bigvee_{a \in A} \mathbf{F}(q_f, a)$, wobei

- ϕ_1 ausdrückt, dass im ersten Durchlauf der Lesekopf im Zustand q_0 ganz links steht und das Band w enthält.
- ϕ_2 ausdrückt, dass zwei aufeinanderfolgende Durchläufe einen zulässigen Schritt in $\mathcal{M}$ darstellen.

Sei $w = a_1 \ldots a_{|w|}$. Es ist

$$\phi_1 \equiv LR \wedge \mathbf{X}(q_0, a_1) \wedge \bigwedge_{i=2}^{|w|} \mathbf{X}^{2i-1} a_i \wedge \bigwedge_{i=|w|+1}^{S(|w|)} \mathbf{X}^{2i-1} \bot$$

Seien M_L, M_R, M_N die Mengen der Paare aus $Q \times A$, für die δ einen Übergang nach links, rechts bzw. auf der Stelle vorsieht.

ϕ_{NR} und ϕ_{NL} drücken aus, dass die augenblickliche Stelle im nächsten Schritt keine Änderung nach rechts bzw. links bewirkt:

$$\phi_{NR} \equiv \bigvee_{a_1 \in A} a_1 \vee \bigvee_{(q,a_1) \notin M_R} (q, a_1) \vee RR$$

$$\phi_{NL} \equiv \bigvee_{a_2 \in A} a_2 \vee \bigvee_{(q,a_2) \notin M_L} (q, a_2) \vee LR$$

ϕ_G drückt aus, unter welchen Umständen eine Stelle in einem Schritt unverändert bleibt:

$$\phi_G \equiv \bigwedge_{a \in A} \mathbf{G}(\phi_{NR} \wedge \mathbf{X}^2 a \wedge \mathbf{X}^4 \phi_{NL}) \rightarrow \mathbf{X}^{2S(|w|)+3} a$$

Abkürzung: Im folgenden sei q' jeweils der Nachfolgezustand von (q, a) und a' das Nachfolgezeichen.

ϕ_C drückt aus, wie sich eine Stelle ändert, wenn sich der Lesekopf dort befindet:

$$\begin{aligned}\phi_C \equiv\ & \bigwedge_{a_1 \in A} \bigwedge_{(q,a) \in M_L} \mathbf{G}\left(\left(a_1 \wedge \mathbf{X}^2(q,a)\right) \rightarrow \left(\mathbf{X}^{2S(|w|)+1}(q', a_1) \wedge \mathbf{X}^{2S(|w|)+3} a'\right)\right) \\ & \wedge \bigwedge_{(q,a) \in M_N} \mathbf{G}\left((q,a) \rightarrow \mathbf{X}^{2S(|w|)+1}(q', a')\right) \\ & \wedge \bigwedge_{(q,a) \in M_R} \bigwedge_{a_1 \in A} \mathbf{G}\left(\left((q,a) \wedge \mathbf{X}^2 a_1\right) \rightarrow \left(\mathbf{X}^{2S(|w|)+1} a' \wedge \mathbf{X}^{2S(|w|)+3}(q', a_1)\right)\right)\end{aligned}$$

Es sei nun $\phi_2 \equiv \phi_C \wedge \phi_G$. Damit ist die Reduktion erreicht. $\mathcal{K}$ und ϕ haben polynomielle Größe in $|\mathcal{M}| + |w|$.

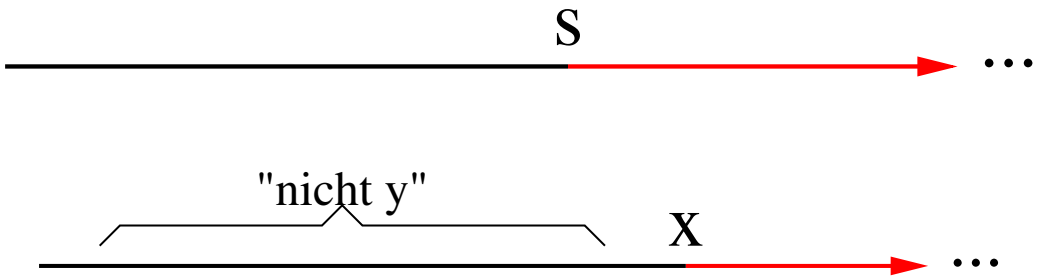
Sprachtheoretische Charakterisierung von LTL

Betrachten wir folgende beiden Eigenschaften:

> “Zustand *s* ist nicht erreichbar.”
>
> “Ereignis *x* passiert nicht, bevor *y* passiert ist.”

Die beiden Eigenschaften haben gemein, dass sie ausdrücken: Etwas “Schlechtes” wird nicht passieren.

Sobald in einem Ablauf dieses “Schlechte” passiert, wissen wir, dass der Ablauf die Eigenschaft verletzt, z.B.



Definition: Sei $\sigma \in \Sigma^{\omega}$. Wir nennen $w \in \Sigma^{*}$ einen Präfix von σ, gdw. $\sigma = w\sigma^{|w|}$.

In den betrachteten Beispielen gilt Folgendes:

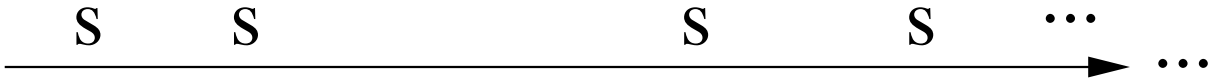
Die Eigenschaft wird verletzt gdw. sie von einem *endlichen* Präfix verletzt wird.

Es folgt: Wenn ein endlicher Präfix eine solche Eigenschaft verletzt, wird jede Erweiterung zu einer unendlichen Sequenz sie ebenfalls verletzen.

Weitere Beispiele

Betrachten wir folgende Eigenschaften:

"Ereignis *s* tritt unendlich oft auf."



"Immer, wenn *s* passiert, passiert irgendwann später *t*."

Diese beiden Eigenschaften haben Folgendes gemein:

Sie drücken aus: Etwas Gutes passiert irgendwann.

Jeder Präfix einer Sequenz kann so ins Unendliche erweitert werden, dass die Erweiterung die obigen Eigenschaften hat.

Sicherheit und Lebendigkeit

Den ersten Typ von Eigenschaften nennen wir “Sicherheits-Eigenschaft”, den zweiten “Lebendigkeits-Eigenschaft”. Formal:

Sei $\mathcal{L} \subseteq \Sigma^\omega$ (für irgendein Alphabet Σ) eine Sprache unendlicher Wörter.

Ein Wort $w \in \Sigma^*$ heißt schlechter Präfix für $\mathcal{L}$, gdw. für alle $\sigma \in \Sigma^\omega$ gilt: $w\sigma \notin \mathcal{L}$.

Definition:

> $\mathcal{L}$ ist eine Sicherheits-Eigenschaft gdw. jedes $\sigma \notin \mathcal{L}$ einen schlechten Präfix hat.
>
> $\mathcal{L}$ ist eine Lebendigkeits-Eigenschaft gdw. es keinen schlechten Präfix für $\mathcal{L}$ gibt.

LTL und Sicherheit/Lebendigkeit

Wir zeigen:

> Für jede LTL-Formel ϕ ist $[\![\phi]\!]$ der Schnitt aus einer Sicherheits- und einer Lebendigkeits-Eigenschaft.

D.h., es gibt eine Sicherheits-Eigenschaft $\mathcal{L}_s$ und eine Lebendigkeits-Eigenschaft $\mathcal{L}_\ell$, so dass $[\![\phi]\!] = \mathcal{L}_s \cap \mathcal{L}_\ell$.

Beweis: Sei $\mathcal{B} = (S, \Sigma, s_0, \delta, F)$ ein BA mit $\mathcal{L}(\mathcal{B}) = [\![\phi]\!]$. O.B.d.A. nehmen wir an, dass es zu jedem $s \in S$ und $a \in \Sigma$ ein s' mit $(s, a, s') \in \delta$ gibt. Außerdem nehmen wir o.B.d.A. an, dass es keine akzeptierenden Zustände gibt, die triviale SCCs bilden.

Seien $N \subseteq S$ die Zustände, von denen kein Zustand aus F erreicht werden kann.

Sei $B \subseteq \Sigma^*$ die Menge der (endlichen) Wörter w, so dass *jeder* mit w beschriftete Pfad von s_0 zu einem Zustand aus N führt. $\vec{B}$ bezeichne die Sprache $\{ w\sigma \mid w \in B,\ \sigma \in \Sigma^\omega \}$.

B ist genau die Menge der schlechten Präfixe von $[\![\phi]\!]$, und $\mathcal{L}(\mathcal{B}) \cap \vec{B} = \emptyset$.

Sei $\mathcal{L}_S$ das Komplement von $\vec{B}$, i.e. $\mathcal{L}_S := \Sigma^\omega \setminus \vec{B}$. Dann ist $\mathcal{L}_S$ eine Sicherheits-Eigenschaft, deren schlechte Präfixe in B sind.

Sei $\mathcal{L}_\ell := \mathcal{L}(\mathcal{B}) \cup \vec{B}$. Dann ist $\mathcal{L}_\ell$ eine Lebendigkeits-Eigenschaft.

Es gilt:
$\mathcal{L}_\ell \cap \mathcal{L}_S = (\mathcal{L}(\mathcal{B}) \cup \vec{B}) \cap (\Sigma^\omega \setminus \vec{B}) = (\mathcal{L}(\mathcal{B}) \setminus \vec{B}) \cup (\vec{B} \setminus \vec{B}) = \mathcal{L}(\mathcal{B}) = [\![\phi]\!]$.

Beispiele

$[\![\mathbf{G}\, p]\!]$ ist eine Sicherheits-Eigenschaft geschnitten mit der (trivialen) Lebendigkeits-Eigenschaft $[\![\mathbf{true}]\!]$.

$[\![\mathbf{F}\, p]\!]$ ist eine Lebendigkeits-Eigenschaft geschnitten mit der (trivialen) Sicherheits-Eigenschaft $[\![\mathbf{true}]\!]$.

$[\![p\ \mathbf{U}\ q]\!]$ ist der Schnitt aus $\mathcal{L}_s = [\![p\ \mathbf{W}\ q]\!]$ und $\mathcal{L}_\ell = [\![\mathbf{F}\, q]\!]$.

Teil 8: Halbordnungstechniken

Die Geschichte bis jetzt

Gegeben:

System S beschrieben als Promela-Modell, C-Programm, Petri-Netz, ...
$\Rightarrow$ konstruiere daraus Kripke-Struktur $\mathcal{K}$

Spezifikation als LTL-Formel ϕ
$\Rightarrow$ konstruiere daraus Büchi-Automaten $\mathcal{B}$

Ansatz:

Konstruiere Produkt von $\mathcal{K}$ und $\mathcal{B}$ und analysiere es “on the fly”

Laufzeit linear in $|\mathcal{K}| \cdot |\mathcal{B}|$

Zustandsexplosion

Größe von $\mathcal{B}$:

- schlimmstenfalls exponentiell in $|\phi|$, aber meist “harmlos”
- (im Allgemeinen) nicht vermeidbar

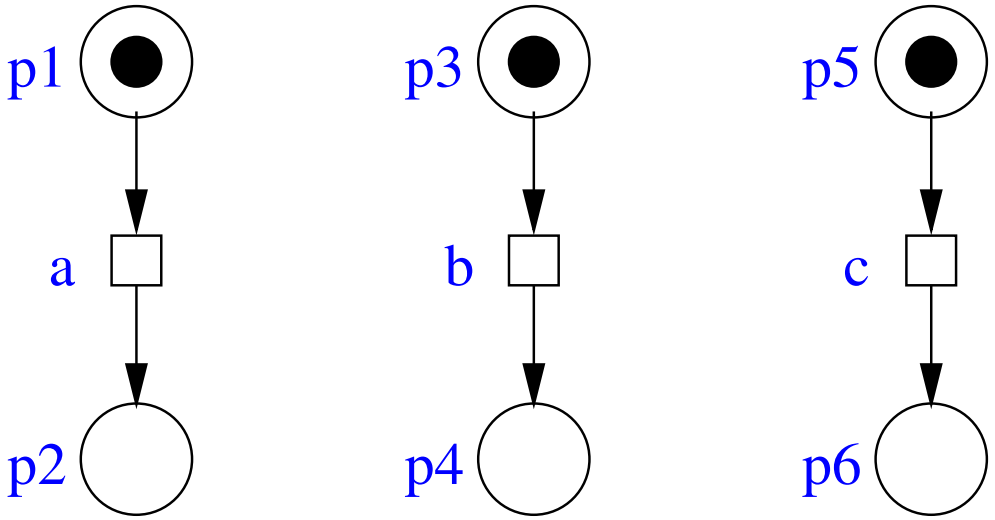
Größe von $\mathcal{K}$:

- schlimmstenfalls exponentiell in $|S|$
- “Zustandsexplosion”, hervorgerufen durch Nebenläufigkeit, Daten, . . .

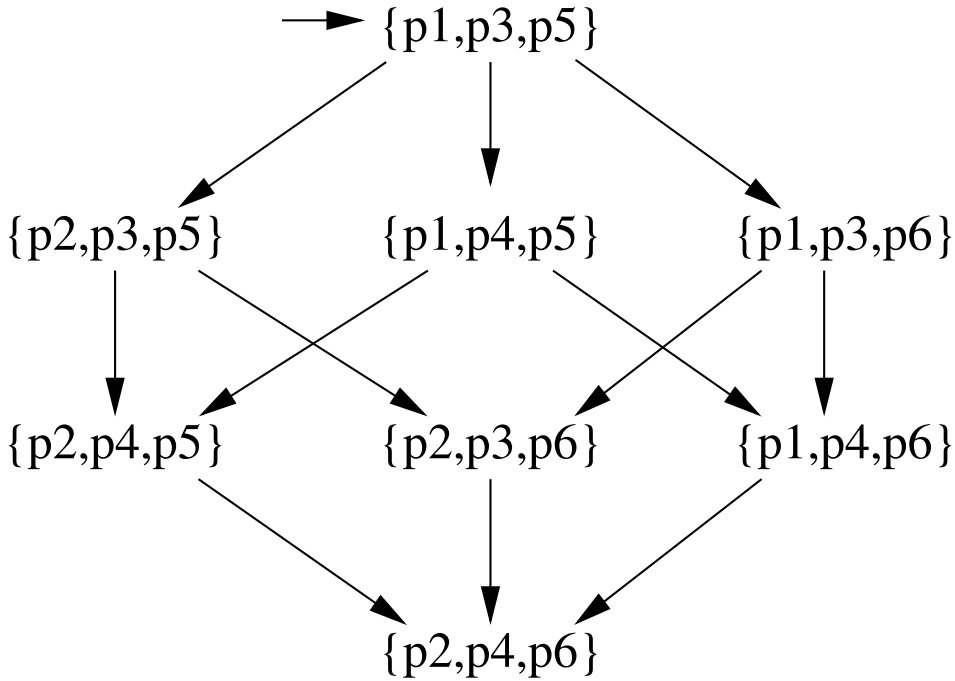
Im Folgenden werden wir eine Verbesserung betrachten, die den Effekt von Nebenläufigkeit mildert.

Beispiel

Betrachten wir das folgende Petri-Netz.

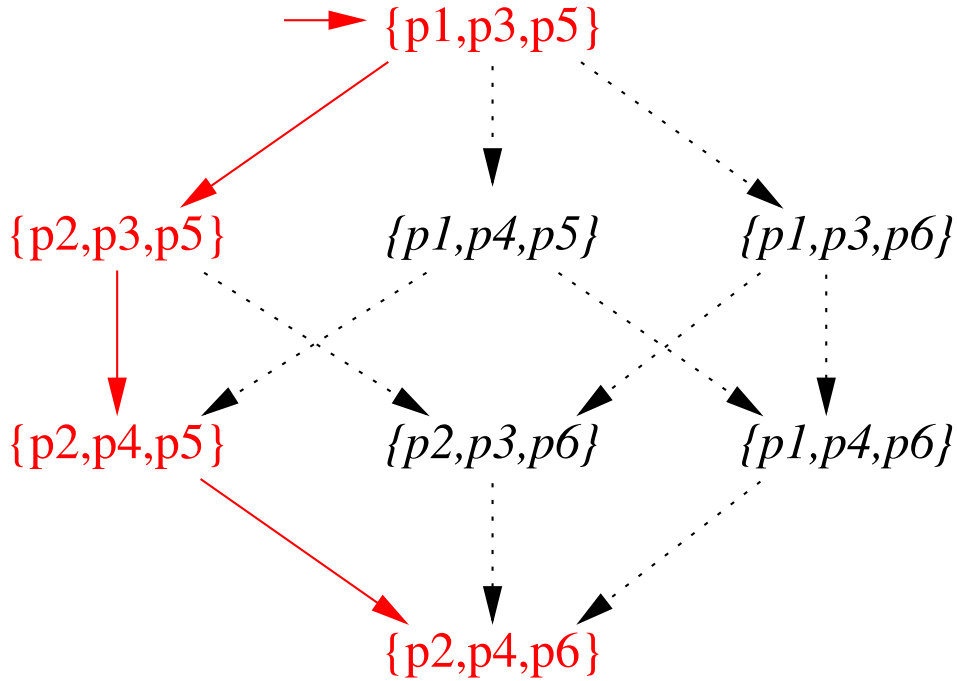


Der Erreichbarkeitsgraph hat $8 = 2^3$ Zustände und $6 = 3!$ mögliche Pfade.



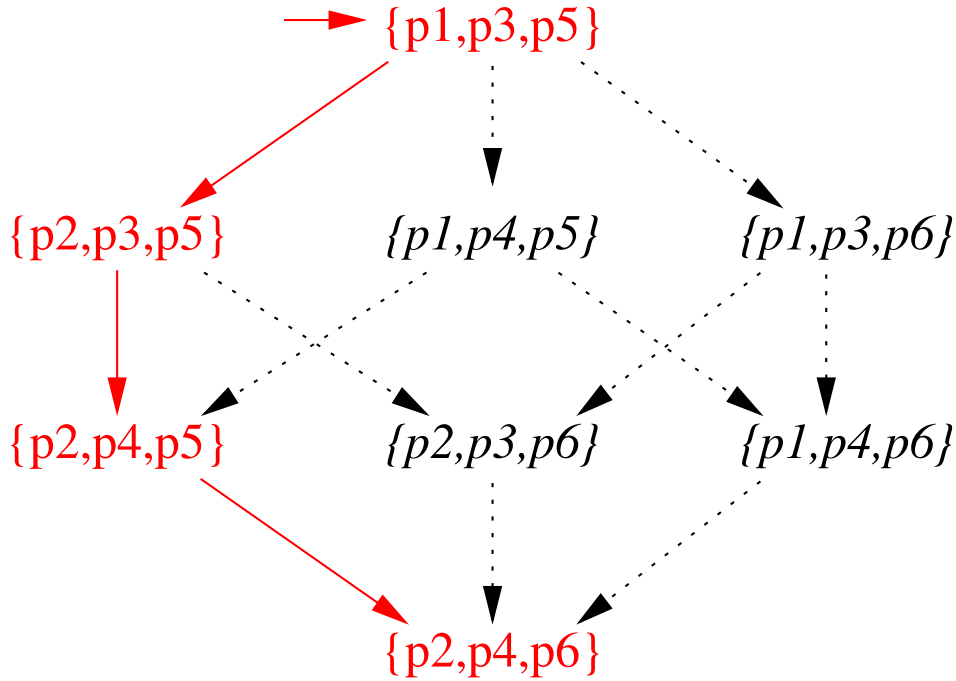
Bei n Komponenten haben wir 2^n Zustände und $n!$ Pfade.

Alle Pfade führen zu $\{p_2, p_4, p_6\}$.



Idee: System verkleinern, indem nur ein Pfad betrachtet wird.

Vorsicht: Geht nur dann, falls alle Pfade “gleichwertig” sind.



D.h., in den weggelassenen Zuständen darf nichts “Interessantes” passieren.

Halbordnungstechniken

Halbordnungstechniken finden Anwendung bei nebenläufigen Systemen.

Man versucht, Unabhängigkeiten zwischen Transitionen auszunutzen, z.B.:

Zuweisungen zu Variablen, die nicht voneinander abhängen:

$$\texttt{x} := \texttt{z} + 5 \quad \| \quad \texttt{y} := \texttt{w} + \texttt{z}$$

Senden und Empfangen auf einem Kanal, der weder voll noch leer ist.

Idee: vermeiden, alle Verschränkungen unabhängiger Transitionen zu erforschen

korrekt, falls die Eigenschaft nicht zwischen der Anordnung unabhängiger Transitionen unterscheidet und jeder Übergang irgendwann erforscht wird

kann den untersuchten Raum um einen exponentiellen Faktor verkleinern

Methoden: ample sets, stubborn sets, persistent sets, sleep sets

On-the-fly Model-Checking

Wichtig: Es wäre sinnlos, erst $\mathcal{K}$ vollständig zu konstruieren und dann zu verkleinern.

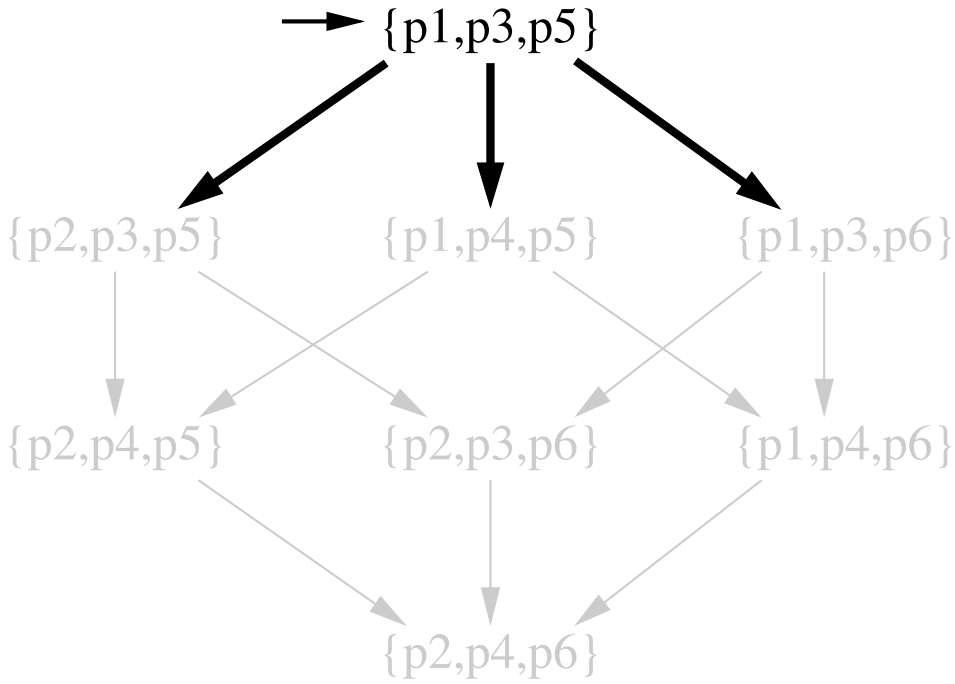
(keine Platzersparnis, Analyse erfolgt während des Aufbaus von $\mathcal{K}$)

Daher: Die Verkleinerung muss “on-the-fly” erfolgen, d.h. noch während der Erzeugung (und gleichzeitigen Analyse) von $\mathcal{K}$.

$\Rightarrow$ Kombination mit Tiefensuche-Algorithmus

Reduktion und Tiefensuche

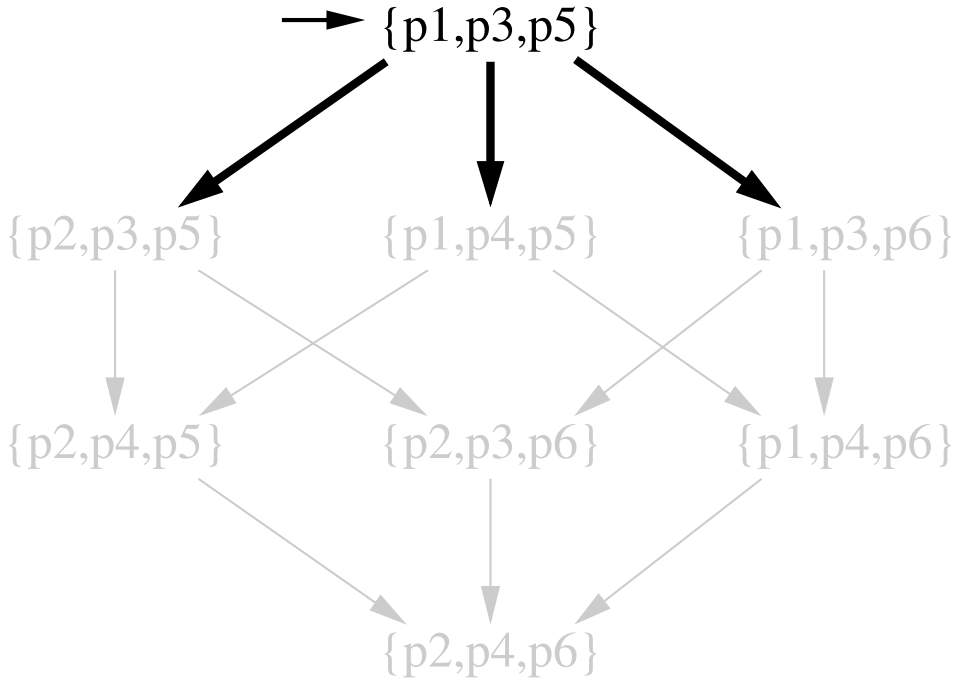
Entscheidung über die zu erforschenden Pfade muss *jetzt* getroffen werden.



D.h. noch bevor wir den Rest der Struktur konstruiert haben!

Reduktion und Tiefensuche

Entscheidung über die zu erforschenden Pfade muss *jetzt* getroffen werden.



$\Rightarrow$ nur mit Hilfe zusätzlicher Information möglich!

Zusätzliche Information

Transitionen werden mit Aktionen beschriftet.

gewonnen aus S, z.B. Anweisungen eines Programms, Transitionen eines Petri-Netzes etc.

Informationen über Unabhängigkeit zwischen Aktionen

Beeinflussen sich zwei Aktionen gegenseitig?

Informationen über Sichtbarkeit von Aktionen

Kann eine Aktion die Gültigkeit der Grundaussagen beeinflussen?

Beschriftete Kripke-Strukturen

Wir erweitern unser Modell um Aktionen:

$$\mathcal{K} = (S, A, \rightarrow, r, AP, \nu)$$

S, r, AP, ν wie zuvor

A sei eine Menge von Aktionen

$\rightarrow \subseteq S \times A \times S$

Wir nehmen forthin an, dass Transitionen deterministisch sind, d.h. für jedes $s \in S$ und $a \in A$ gibt es höchstens ein $s' \in S$, so dass $(s, a, s') \in \rightarrow$.

$en(s) = \{ a \mid \exists s' \colon (s, a, s') \in \rightarrow \}$ heißen die möglichen Aktionen in s.

Unabhängigkeit

$I \subseteq A \times A$ heißt Unabhängigkeitsrelation für $\mathcal{K}$, falls:

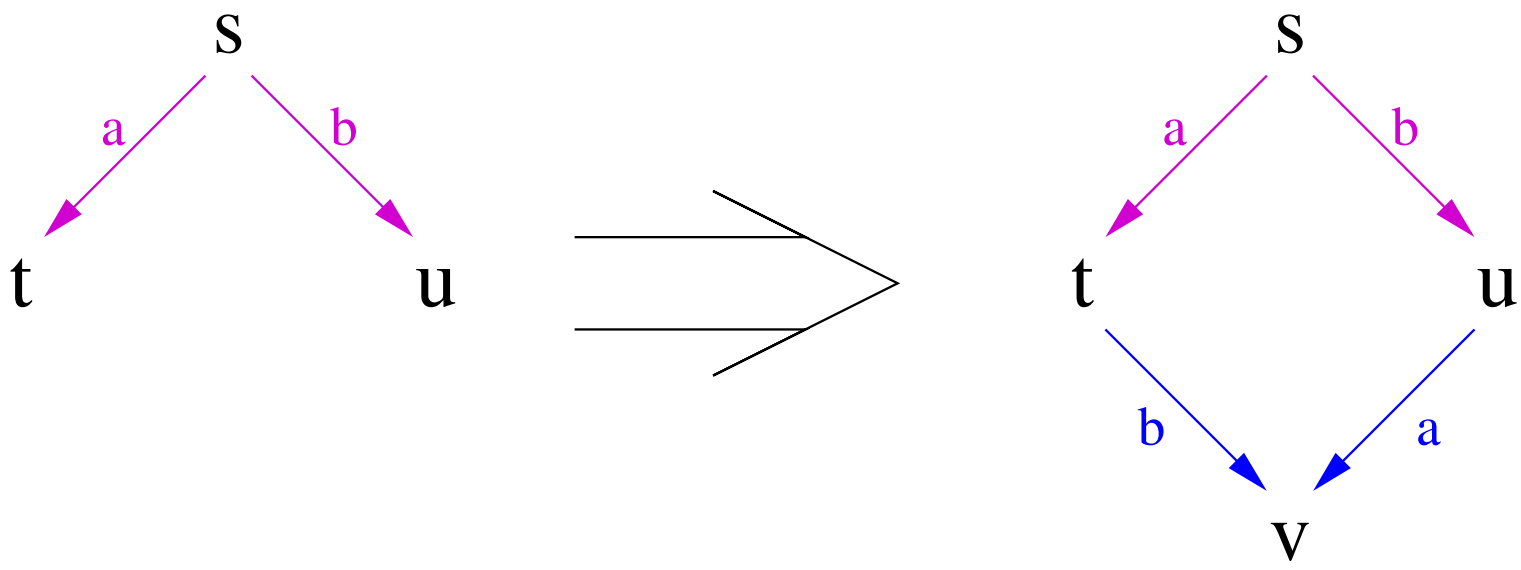
für alle $a \in A$ gilt $(a, a) \notin I$ (Irreflexivität);

für alle $(a, b) \in I$ gilt $(b, a) \in I$ (Symmetrie);

für alle $(a, b) \in I$ und alle $s \in S$ gilt:

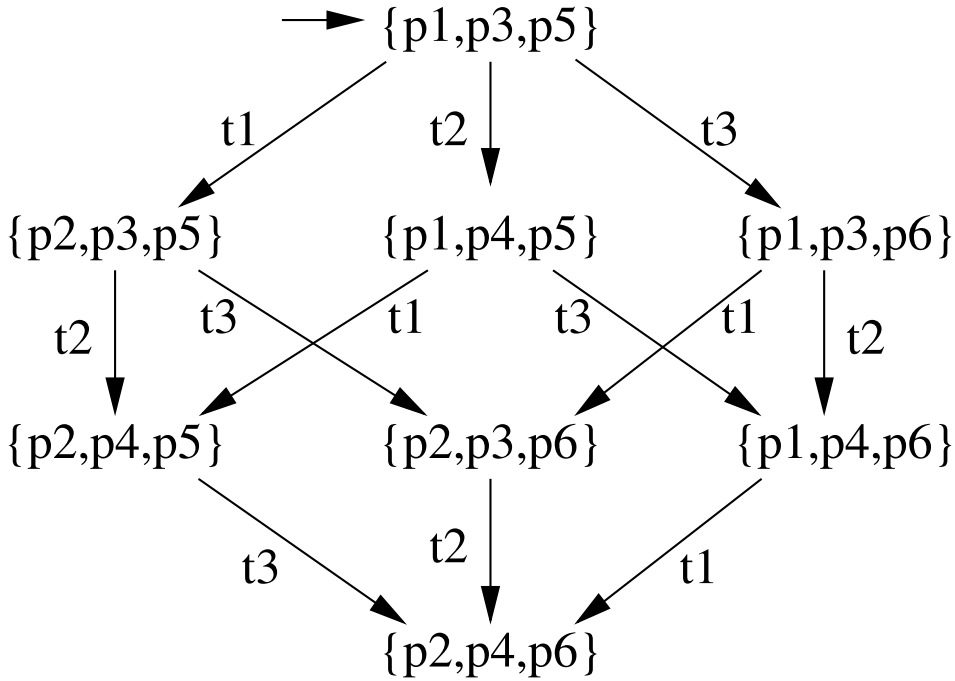
Wenn $a, b \in en(s)$, $s \xrightarrow{a} t$ und $s \xrightarrow{b} u$,
dann gibt es v mit $a \in en(u)$, $b \in en(t)$, $t \xrightarrow{b} v$ und $u \xrightarrow{a} v$.

Unabhängigkeit



Unabhängigkeit: Beispiel

Im untenstehenden Beispiel sind alle Paare von Aktionen unabhängig.



Bemerkung: Unabhängigkeitsrelation i.A. nicht transitiv!

(Un-)Sichtbarkeit

Eine Menge $U \subseteq A$ heißt Unsichtbarkeitsmenge, falls alle $a \in U$ folgende Eigenschaft haben:

> Für alle $(s, a, s') \in \rightarrow$ gilt $\nu(s) = \nu(s')$.

D.h., Aktion a führt niemals zur Änderung einer Grundaussage.

Motivation: Verschränkungen sichtbarer Aktionen dürfen nicht verloren gehen, da sie Einfluss auf die Gültigkeit von Formeln haben könnten.

Bemerkungen

Quelle für I und U: “externes” Wissen über Eigenschaften von Aktionen

z.B. Addition kommutativ, Struktur eines Petri-Netzes, . . .

werden *nicht* konstruiert, indem man sich erst die ganze Kripke-Struktur anschaut . . .

Jede (symmetrische) Untermenge einer Unabhängigkeitsrelation ist eine Unabhängigkeitsrelation, jede Untermenge einer Unsichtbarkeitsmenge ist immer noch eine Unsichtbarkeitsmenge.

$\Rightarrow$ konservative Abschätzung möglich

Aber: je größer die Relation/Menge, desto mehr Information, desto bessere Reduktion möglich

Im Folgenden nehmen wir an, dass irgendeine Unabhängigkeitsrelation I und irgendeine Unsichtbarkeitsmenge U gegeben sind.

Wir nennen a und b unabhängig, falls $(a, b) \in I$, und sonst abhängig.

Wir nennen a unsichtbar, falls $a \in U$, und sonst sichtbar.

Vorschau

Wir definieren den Begriff “gleichwertiger” (äquivalenter) Abläufe.

Wir betrachten Bedingungen, die garantieren, dass jede Äquivalenzklasse im verkleinerten System erhalten bleibt.

Wir betrachten die praktische Implementierung dieser Bedingungen in Spin.

Stotter-Äquivalenz

Definition: Seien σ, ρ unendliche Sequenzen über 2^{AP}. Wir nennen σ und ρ stotter-äquivalent gdw. es Folgen

$$0 = i_0 < i_1 < i_2 < \cdots \text{ und } 0 = k_0 < k_1 < k_2 < \cdots$$

gibt, so dass für alle $\ell \geq 0$ gilt:

$$\sigma(i_\ell) = \sigma(i_\ell + 1) = \cdots = \sigma(i_{\ell+1} - 1) =$$
$$\rho(k_\ell) = \rho(k_\ell + 1) = \cdots = \rho(k_{\ell+1} - 1)$$

(Einteilung von σ und ρ in “Blöcke” gleicher Mengen.)

Wir erweitern den Begriff auf Kripke-Strukturen:

Seien $\mathcal{K}, \mathcal{K}'$ zwei Kripke-Strukturen mit den gleichen Grundaussagen AP.

$\mathcal{K}$ und $\mathcal{K}'$ heißen stotter-äquivalent gdw. zu jeder Sequenz in $[\![\mathcal{K}]\!]$ eine stotter-äquivalente Sequenz in $[\![\mathcal{K}']\!]$ existiert, und umgekehrt.

D.h., $[\![\mathcal{K}]\!]$ und $[\![\mathcal{K}']\!]$ haben dieselben Äquivalenzklassen.

Stotter-Invarianz

Sei ϕ eine LTL-Formel. Wir nennen ϕ stotter-invariant gdw. für alle stotter-äquivalenten Sequenz-Paare σ and ρ gilt:

$$\sigma \in [\![\phi]\!] \quad \text{gdw.} \quad \rho \in [\![\phi]\!].$$

Anders ausgedrückt: ϕ kann nicht zwischen stotter-äquivalenten Sequenzen unterscheiden. (Und auch nicht zwischen stotter-äquivalenten Kripke-Strukturen.)

Satz: Alle LTL-Formeln, in denen kein $\mathbf{X}$ vorkommt, sind stotter-invariant.
Beweis: war in der Übung dran.

Strategie

Wir ersetzen $\mathcal{K}$ durch eine stotter-äquivalente, kleinere Struktur $\mathcal{K}'$.

Dann prüfen wir, ob $\mathcal{K}' \models \phi$ gilt, was äquivalent zu $\mathcal{K} \models \phi$ ist.

$\mathcal{K}'$ erzeugen wir, indem wir bei jedem Schritt während der Tiefensuche nur einen Teil der möglichen Aktionen betrachten, dazu werden I und U gebraucht.

Hier vorgestellte Methode: Ample sets (“hinreichende” Mengen)

Ample sets

Für jeden Zustand s wird eine Menge $red(s) \subseteq en(s)$ berechnet; nur die Nachfolger in $red(s)$ werden während der Tiefensuche erforscht.

(miteinander konkurrierende) Nebenbedingungen:

- $red(s)$ muss so gewählt sein, dass Stotter-Äquivalenz garantiert ist.
- Die Wahl von $red(s)$ sollte $\mathcal{K}$ stark verkleinern.
- Die Berechnung von $red(s)$ sollte effizient geschehen.

Bedingungen für Ample sets

C0: $red(s) = \emptyset$ gdw. $en(s) = \emptyset$

C1: Auf jedem Pfad von s in $\mathcal{K}$ gilt: keine Aktion, die von einer Aktion in $red(s)$ abhängt, kann vor einer Aktion in $red(s)$ ausgeführt werden.

C2: Falls $red(s) \subset en(s)$ (echte Teilmenge!), sind alle Aktionen in $red(s)$ unsichtbar.

C3: Für alle Zyklen in $\mathcal{K}'$ gilt: falls $a \in en(s)$ für einen Zustand s im Zyklus, dann $a \in red(s')$ für einen Zustand s' im Zyklus.

Idee

C0 stellt sicher, dass keine Abläufe “abgeschnitten” werden, d.h. keine zusätzlichen Verklemmungen entstehen.

C1 und C2 stellen sicher, dass keine “interessanten” Verzweigungen verpasst werden.

C3 stellt sicher, dass mögliche Aktionen nicht für immer aufgeschoben werden.

Beispiel

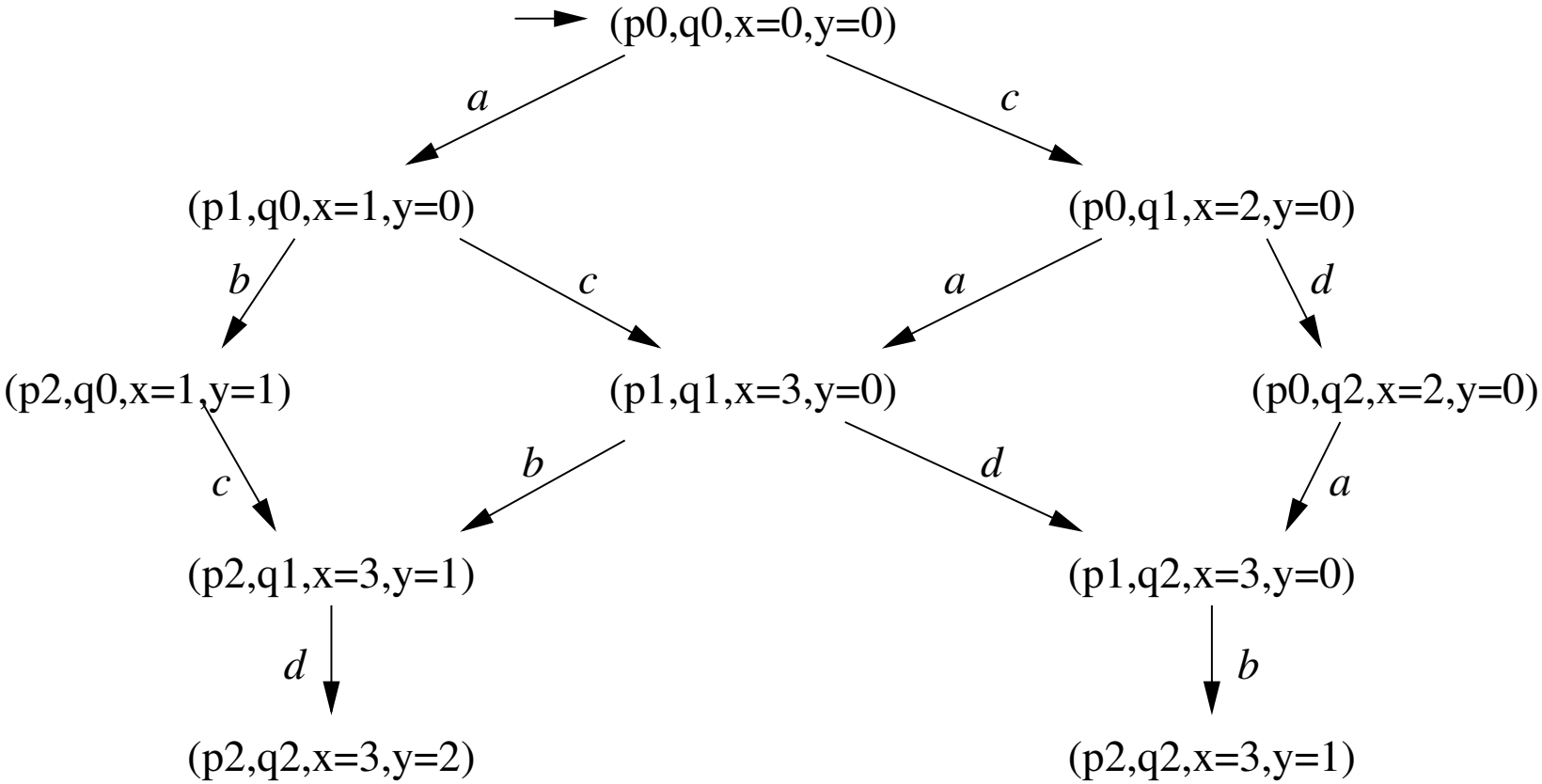
Pseudocode-Programm mit zwei Prozessen:

int x,y **init** 0;
cobegin $\{P \parallel Q\}$ **coend**

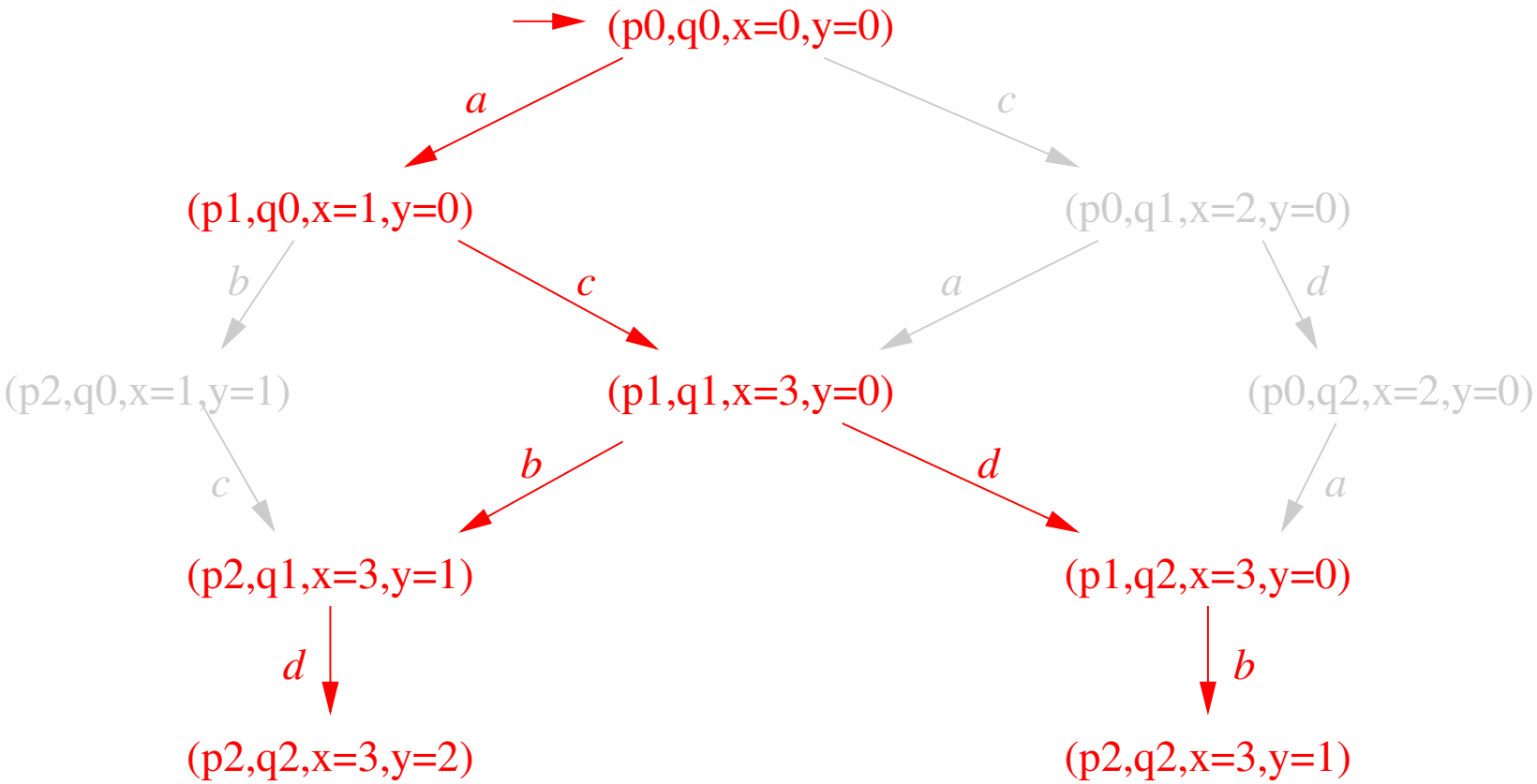
$P =$	p0: $x := x + 1;$ (Aktion a)	$Q =$	q0: $x := x + 2;$ (Aktion c)
	p1: $y := y + 1;$ (Aktion b)		q1: $y := y * 2;$ (Aktion d)
	p2: **end**		q2: **end**

b und d können nicht unabhängig sein, alle anderen Paare von Aktionen schon.

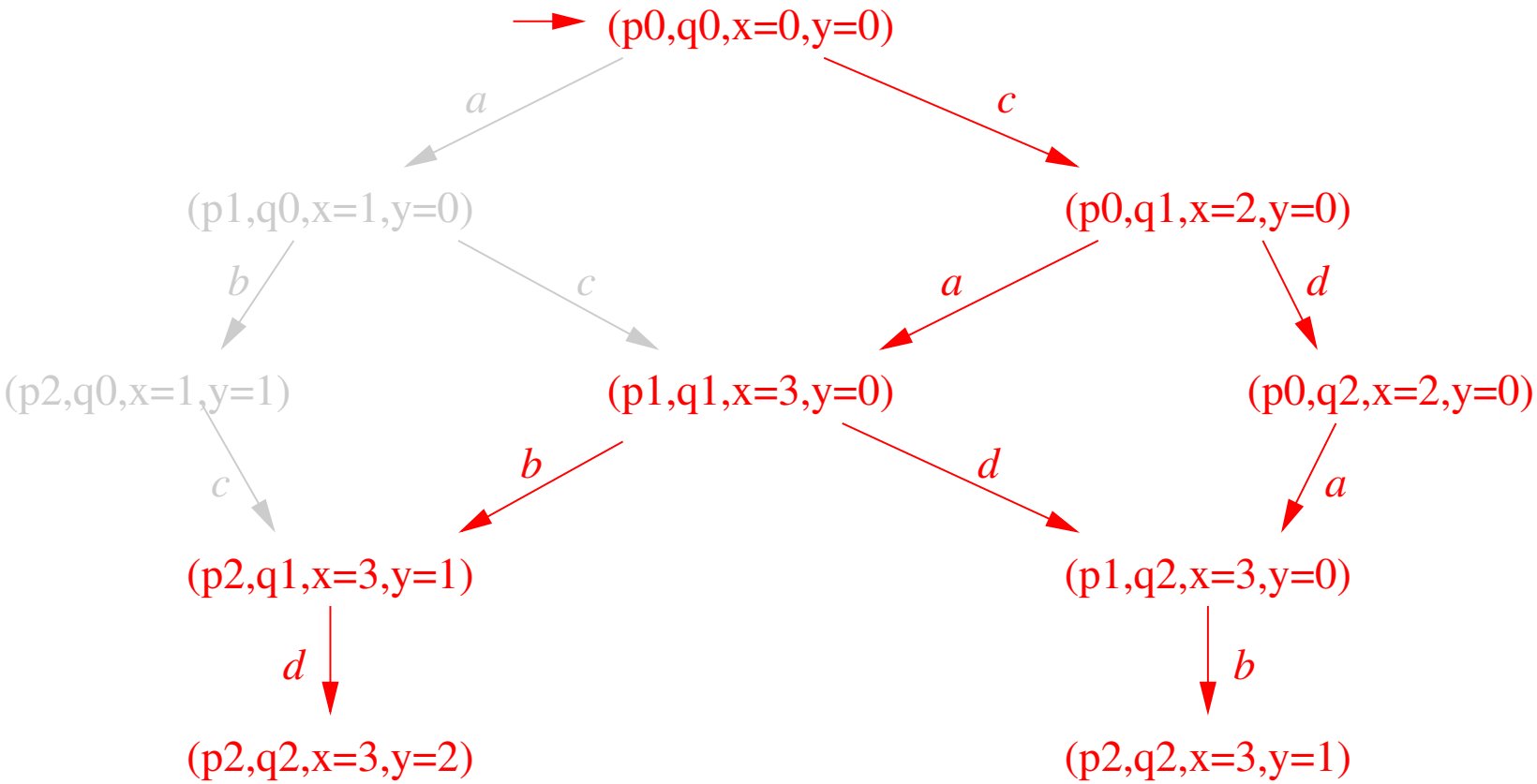
Kripke-Struktur für das Beispiel-Programm:



Mögliche reduzierte Struktur, falls alle Aktionen unsichtbar sind :



Mögliche reduzierte Struktur, falls a, d sichtbar sind :



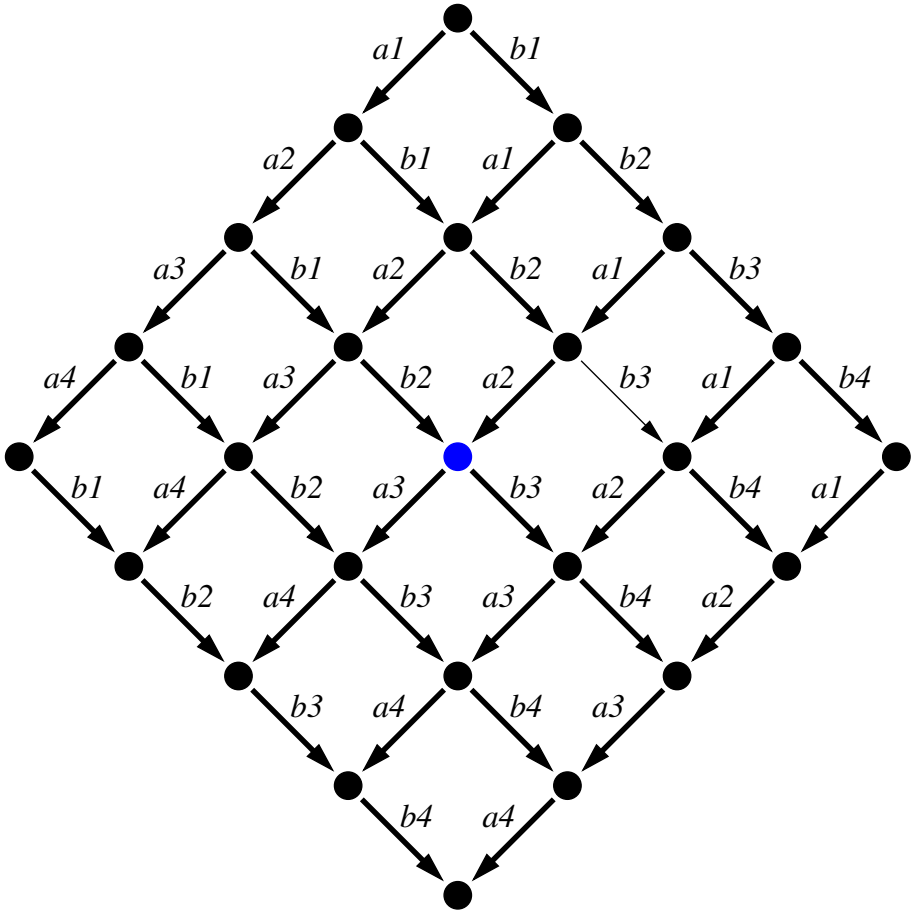
(Nicht-)Optimalität

Ideal wäre es, wenn eine Reduktion nur einen Ablauf aus jeder Stotter-Äquivalenzklasse behält.

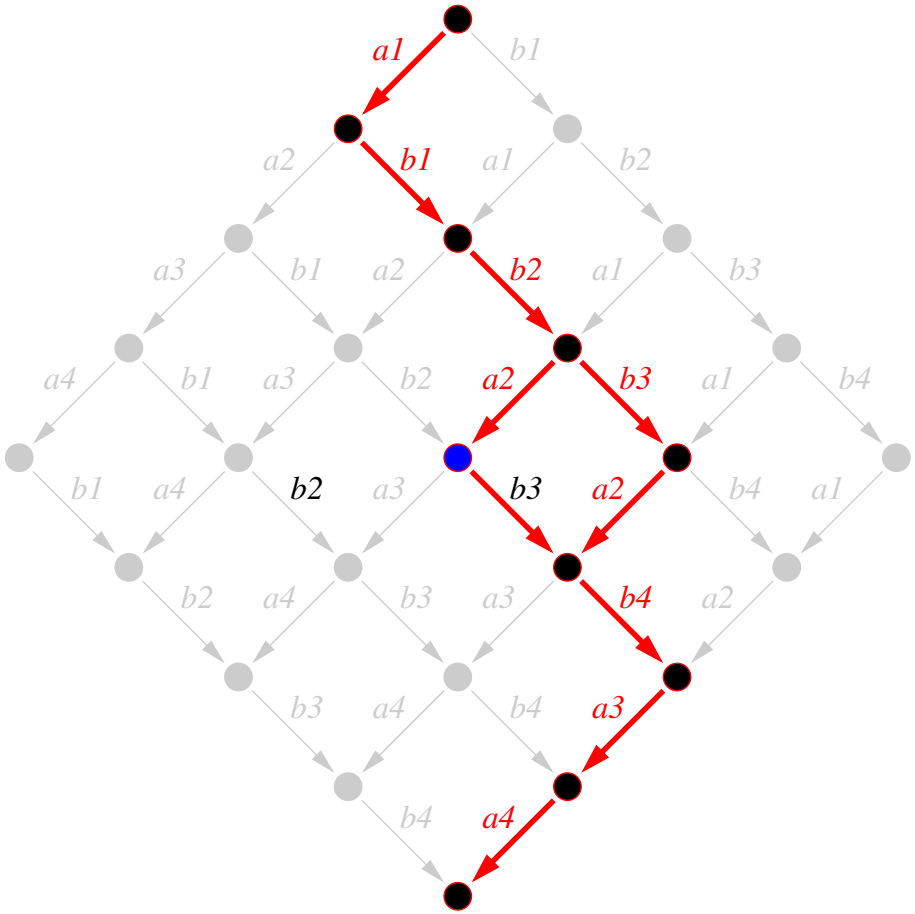
Dies wird durch C0–C3 *nicht* sichergestellt, d.h. Reduktionen, die diese Eigenschaften erfüllen, sind nicht notwendigerweise minimal.

Beispiel (siehe nächste Folie): zwei parallele Prozesse mit je vier Aktionen ($a_1, \ldots, a_4$ bzw. $b_1, \ldots, b_4$), jeweils unabhängig.

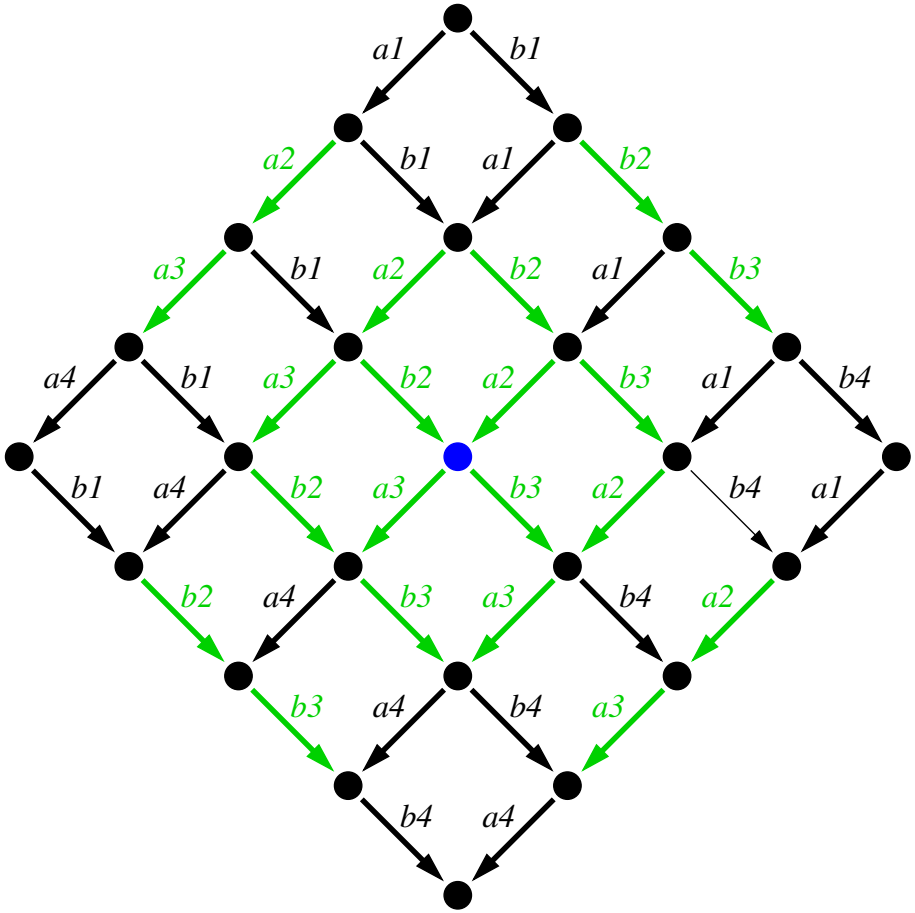
Annahme: Der blaue Zustand erfüllt andere Grundaussagen als die übrigen:



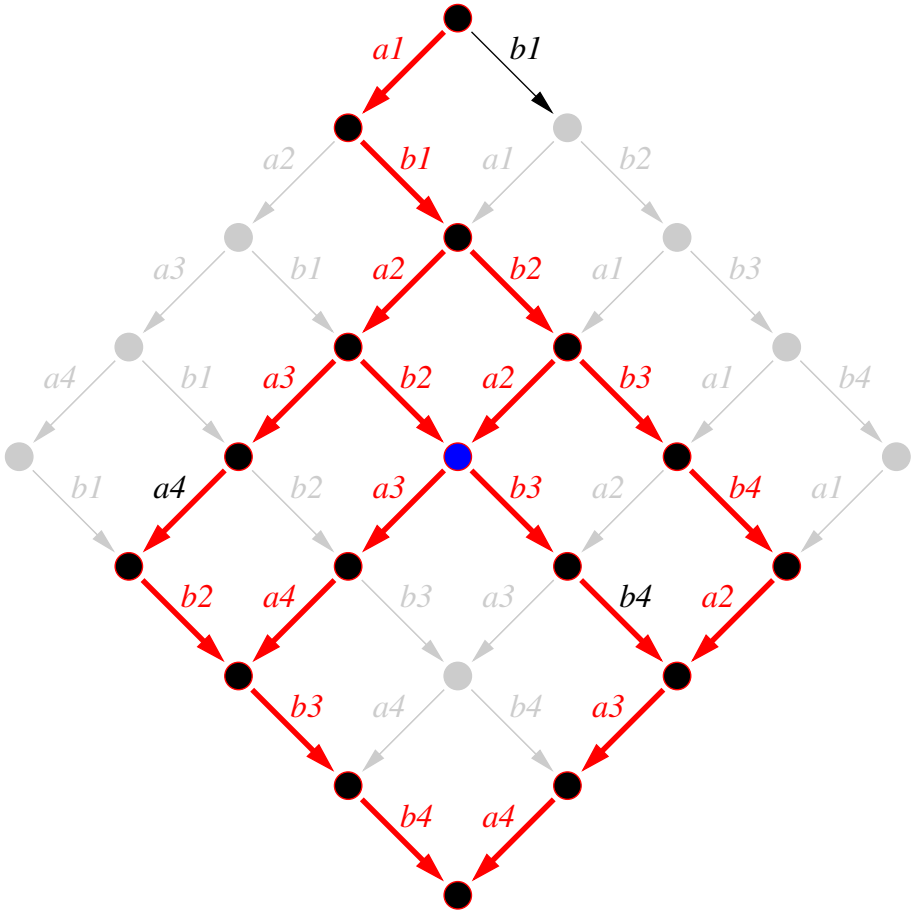
Kleinste stotter-äquivalente Struktur:



Sichtbare Aktionen: a_2, a_3, b_2, b_3 (in grün):



Kleinste durch C0–C3 erzeugbare Struktur:

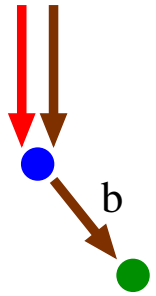


Korrektheit

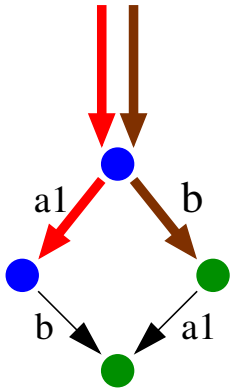
Behauptung: Wenn red die Bedingungen C0 bis C3 erfüllt, dann ist $\mathcal{K}'$ stotter-äquivalent zu $\mathcal{K}$.

Beweis (Idee): Sei σ ein unendlicher Pfad in $\mathcal{K}$. Wir zeigen, dass es dann in $\mathcal{K}'$ einen unendlichen Pfad τ gibt, so dass $\nu(\sigma)$ und $\nu(\tau)$ stotter-äquivalent sind.

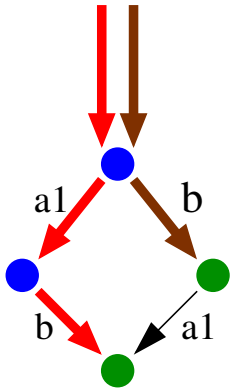
Im Folgenden ist σ braun dargestellt und τ rot. Zustände, von denen wir wissen, dass sie die gleichen Grundaussagen erfüllen, werden in der gleichen Farbe gezeichnet.



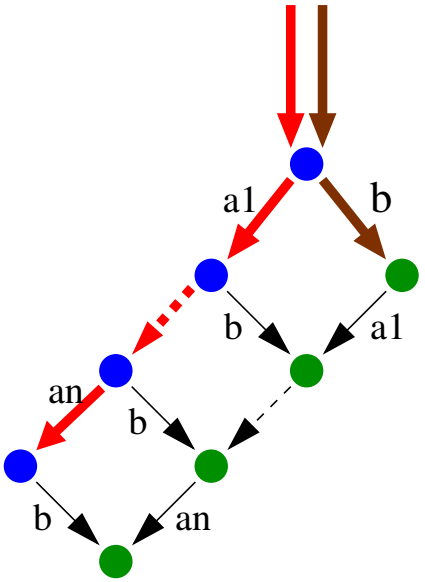
Die mit b beschriftete Transition sei die erste in σ, die es in $\mathcal{K}'$ nicht gibt.



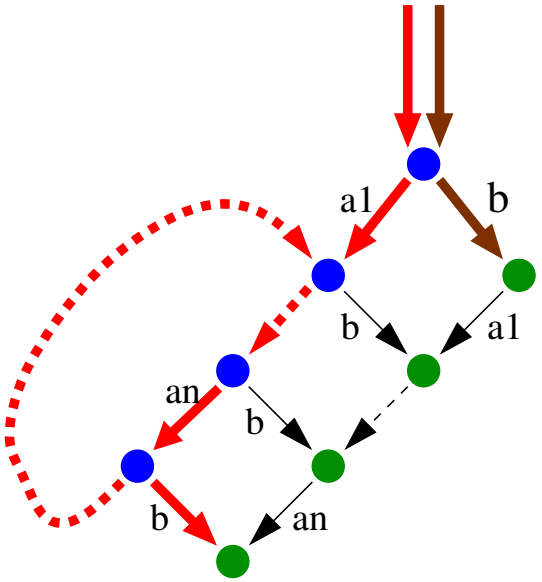
Wegen C0 muss es im blauen Zustand eine weitere Kante z.B. mit a_1 geben. a_1 ist unabhängig von b (C1) und unsichtbar (C2).



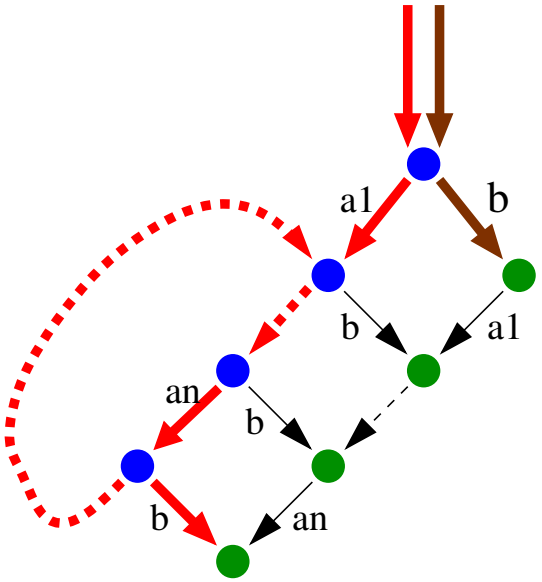
Entweder ist die zweite b-Transition in $\mathcal{K}'$ enthalten, dann ist τ die rote Kantenfolge . . .



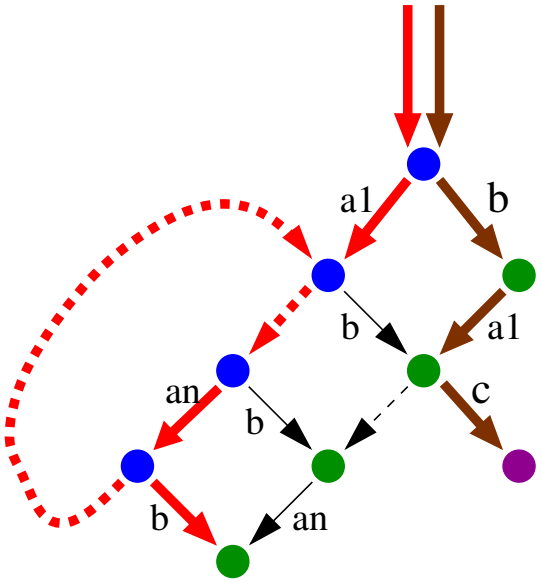
. . . oder b wird “aufgeschoben” zugunsten von $a_2, \ldots, a_n$, ebenfalls unsichtbar und unabhängig von b.



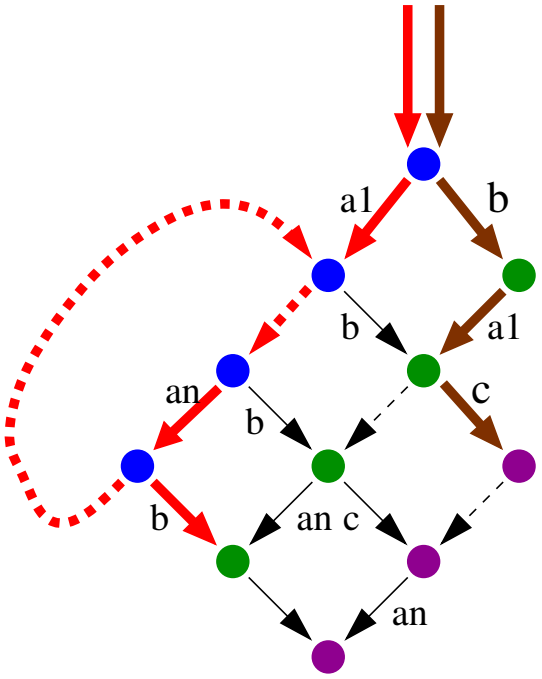
Da $\mathcal{K}$ endlich ist, führt das Aufschieben irgendwann dazu, dass ein Zyklus (in $\mathcal{K}'$) entsteht. Wegen C3 muss b in irgendeinem Zustand des Zyklus aktiviert sein.



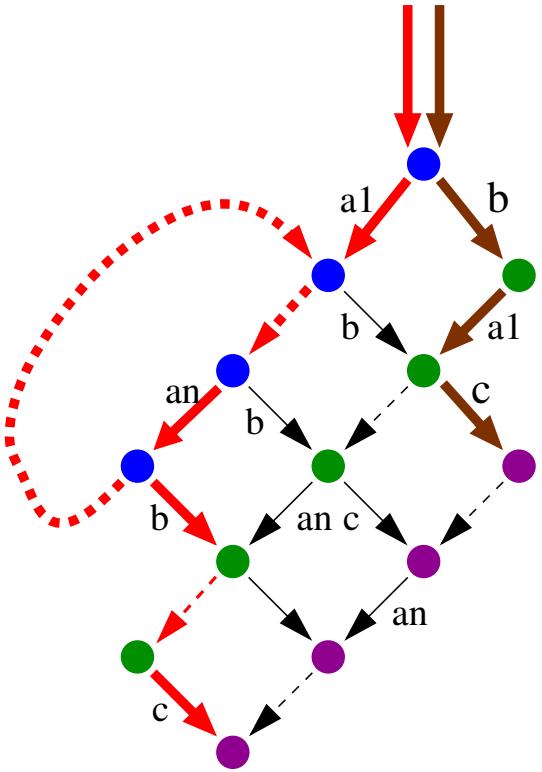
σ and τ enthalten blaue Zustände gefolgt von grünen.



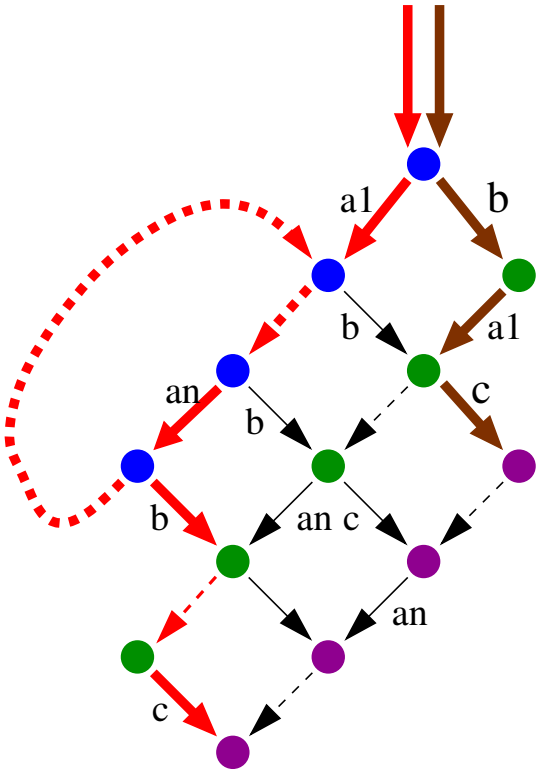
σ geht entweder mit $a_1, \ldots, a_n$ weiter, bis die Pfade “konvergieren”, oder zweigt zuvor mit einer Aktion c weiter ab.



Dann aber muss c unabhängig von $a_2, \ldots, a_n$ sein.



Mit denselben Argumenten wie zuvor können wir schließen, dass $\mathcal{K}'$ auch eine c-Kante entlang des roten Pfads hat.



Der rote und der braune Pfad enthalten wieder blaue, grüne, und lila Zustände.
Das Argument wiederholt sich ggfs. bis ins Unendliche.

Implementierung der Bedingungen

C0 und C2: klar

C1 und C3 hängen vom gesamten Zustandsgraphen ab.

Wir finden Bedingungen, die stärker sind als C1 und C3.
Diese führen ggfs. zu geringeren Reduktionen, lassen sich dafür effizient während der Tiefensuche überprüfen.

Ersetze C3 durch C3':

> Für einen Zustand s mit $red(s) \subset en(s)$ (echte Teilmenge) darf keine Aktion in $red(s)$ zu einem Zustand führen, der gerade im Stack der Tiefensuche liegt.

Heuristik für C1

Abhängig von der Beschreibung des Systems; hier Beispiel für Spin.

Abhängigkeitsrelation darf überapproximiert werden; dadurch entsteht eine sub-optimales, aber noch immer “sichere” Reduktion.

Wir betrachten ein Promela-Modell mit nebenläufigen Prozessen $P_1, \ldots, P_n$, die durch globale Variablen und Kanäle kommunizieren.

Sei $pc_i(s)$ der Kontrollzustand von Prozess P_i im Zustand s.

Heuristik für C1 (in Spin)

$pre(a)$ sei die Menge der Aktionen, die a aktivieren könn(t)en, d.h. (eine Überapproximation von)

$$\{\, b \mid \exists s : a \notin enabled(s),\ b \in enabled(s),\ a \in enabled(b(s)) \,\}$$

In Spin: Sei a eine Aktion in Prozess P_i. $pre(a)$ enthält

alle Anweisungen von P_i, die zu einem Kontrollzustand führen, in dem a ausgeführt werden kann;

wenn der Wächter für a globale Variablen abfragt, alle Anweisungen in anderen Prozessen, die diese Variablen ändern;

wenn a aus einem Kanal q liest oder in ihn schreibt, alle Anweisungen in anderen Prozessen, die das Umgekehrte mit q tun.

Heuristik für C1 (in Spin)

$dep(a) = \{ b \mid (a, b) \notin I \}$ enthält die Aktionen, die von a abhängig sind.

In Spin: Sei a eine Aktion in Prozess P_i. $dep(a)$ wird überapproximiert durch:

alle anderen Aktionen in P_i;

Aktionen in anderen Prozesse, die eine Variable beschreiben, die von a gelesen wird (oder umgekehrt);

wenn a aus einem Kanal q liest oder in ihn schreibt, alle Anweisungen in anderen Prozessen, die das Gleiche mit q tun.

Heuristik für C1 (in Spin)

A_i seien die Aktionen in Prozess P_i.

$E_i(s)$ seien die Aktionen von Prozess P_i, die in s aktiviert sind.

$$E_i(s) = en(s) \cap A_i$$

$C_i(s)$ seien alle Aktionen, die in irgendeinem Zustand mit Kontrollzustand $pc_i(s)$ in P_i möglich sind.

$$C_i(s) = \bigcup_{\{\, s' | pc_i(s) = pc_i(s') \,\}} E_i(s')$$

Der Ansatz in Spin

Teste für alle $1 \leq i \leq n$ die Mengen $E_i(s)$ als Kandidaten für $red(s)$, falls $E_i(s) \neq \emptyset$.

Falls der Test für alle i fehlschlägt, nimm $red(s) = en(s)$.

Wenn C1 verletzt ist, wird eine Aktion a ausgeführt, die von $E_i(s)$ abhängt, bevor $E_i(s)$ selbst ausgeführt wird.

Entweder gehört a zu P_j, $j \neq i$. $\Rightarrow$ Prüfen, ob $A_j \cap dep(E_i(s)) \neq \emptyset$

Sonst ist $a \in C_i(s) \setminus E_i(s)$. $\Rightarrow$ Prüfen, ob $C_i(s) \setminus E_i(s)$ durch einen anderen Prozess aktiviert werden kann.

Test für C1

function $check\_C1(s, P_i)$
  for all $P_j \neq P_i$ **do**
    if $dep(E_i(s)) \cap A_j \neq \emptyset \vee$
      $pre(C_i(s) \setminus E_i(s)) \cap A_j \neq \emptyset$ **then**
        return $False$;
    end if;
  end for all;
  return $True$;
end function

Beispiel: Leader-Election-Protokoll

Das folgende Protokoll wurde von Dolew, Klawe, Rodeh (1982) entwickelt. Die Spin-Distribution enthält ein Modell davon, als Demonstrations-Beispiel für die Reduktionsmethode.

Das Protokoll hat *n* Teilnehmer und eine ringförmige Kommunikationsstruktur, d.h. jeder Teilnehmer erhält Nachrichten von seinem “linken” Nachbarn und schickt Nachrichten an seinen “rechten” Nachbarn.

Die Kommunikation findet asynchron statt (d.h. die Nachrichten werden in Puffern zwischengespeichert), aber die Nachrichtenkanäle sind zuverlässig, d.h. keine Nachrichten gehen verloren. Jeder Teilnehmer hat eine eindeutige Nummer.

Ziel des Protokolls: Auswahl eines “Anführers”.

Leader-Election-Protokoll

Teilnehmer sind entweder aktiv oder inaktiv. Zu Beginn ist jeder Teilnehmer *aktiv*.

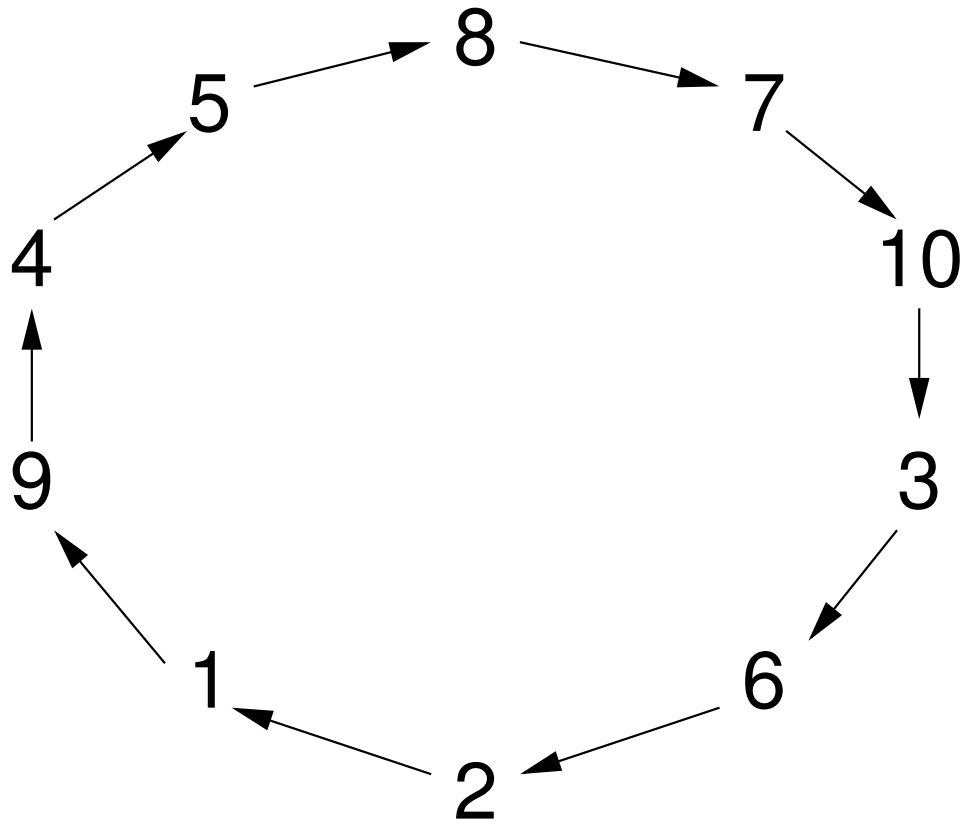
Das Protokoll besteht aus Runden. Jede Runde führt dazu, dass mindestens die Hälfte der noch aktiven Teilnehmer inaktiv wird. (Es gibt also höchstens $\log n$ Runden.)

In jeder Runde erhält jeder aktive Teilnehmer die Nummern der *zwei* nächstgelegenen aktiven Teilnehmer “von links”. Er vergleicht anschließend seine eigene Nummer mit den zwei anderen.

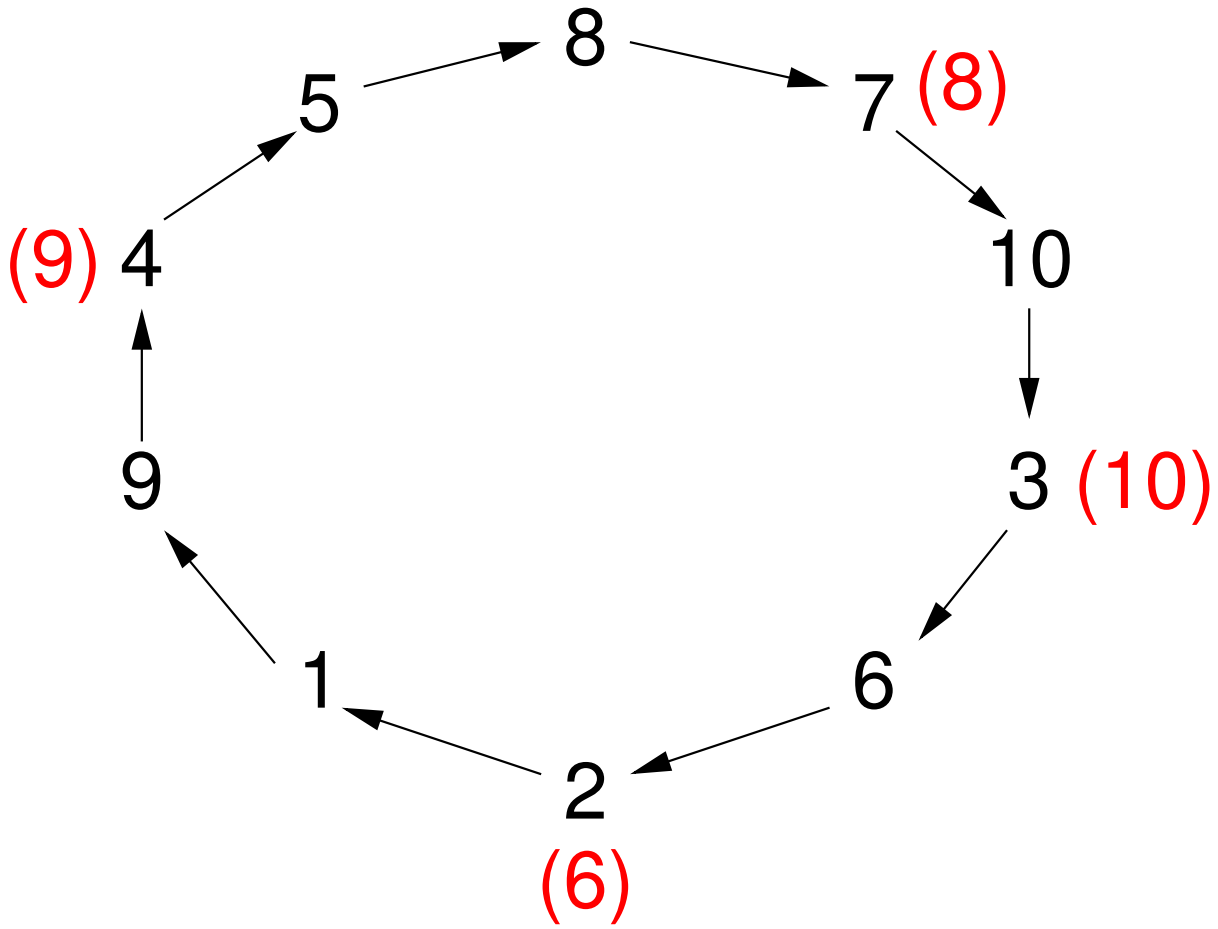
Ein Teilnehmer bleibt aktiv gdw. wenn von den drei Nummern die des nächstgelegenen Nachbarn am größten ist. Bleibt der Prozess aktiv, so nimmt er diese größte Nummer an.

Der letzte noch aktive Teilnehmer erklärt sich zum Anführer.

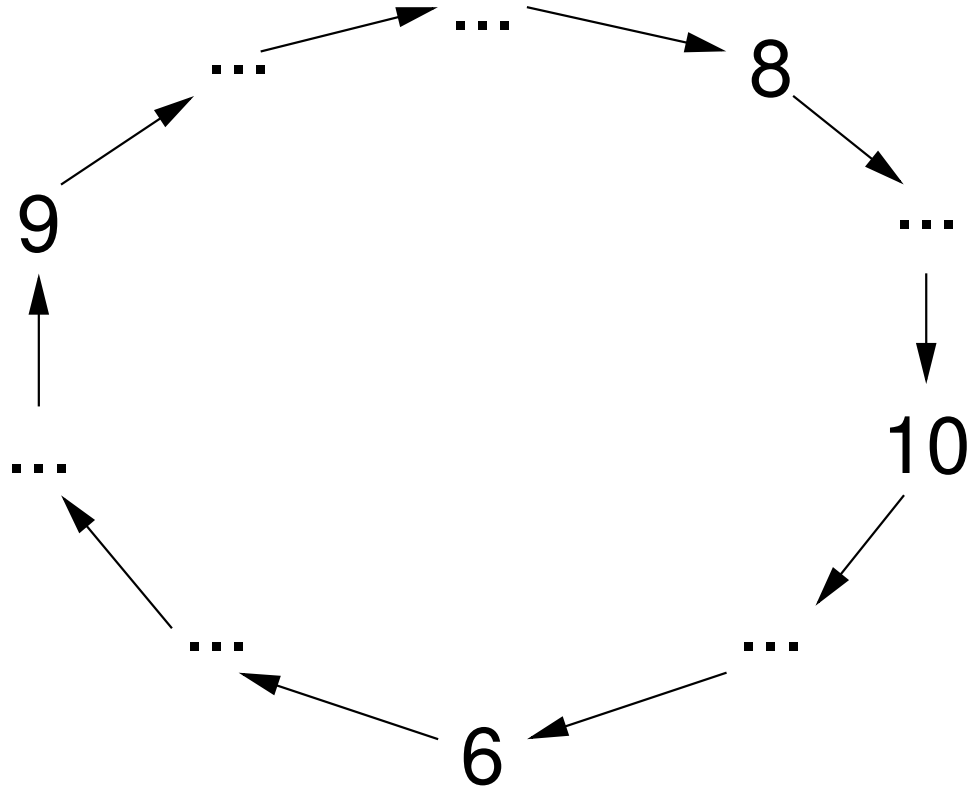
Leader Election: Beispiel



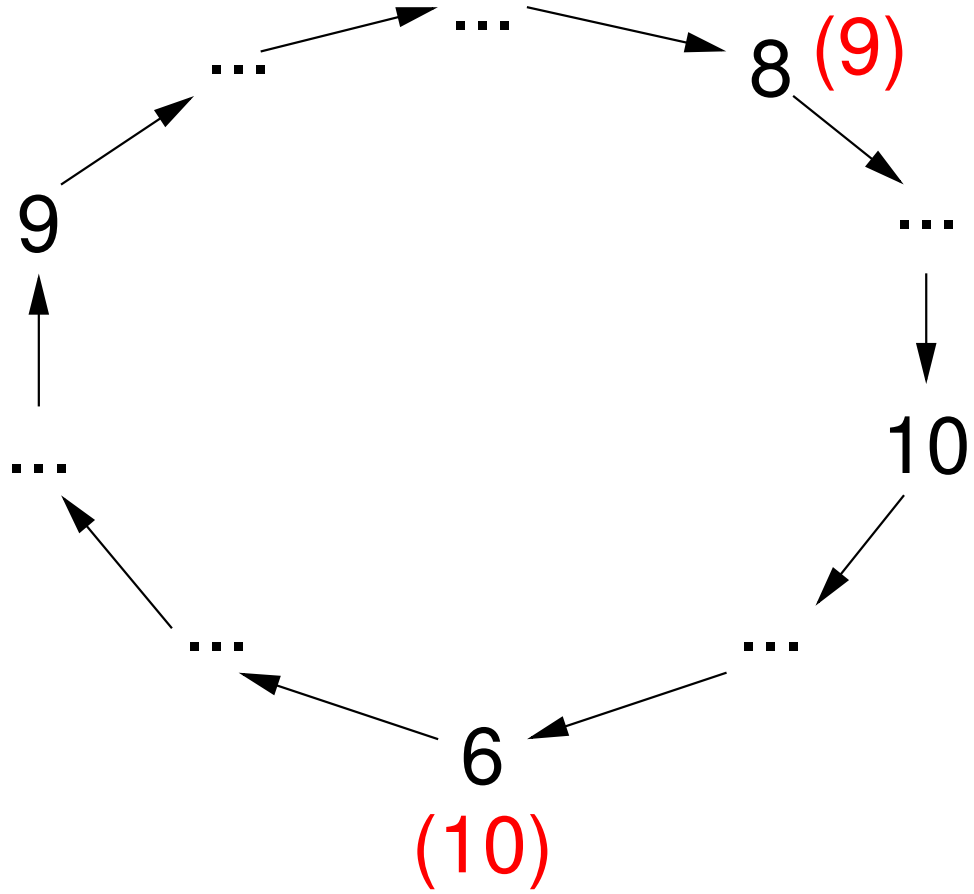
Leader Election: Erste Runde



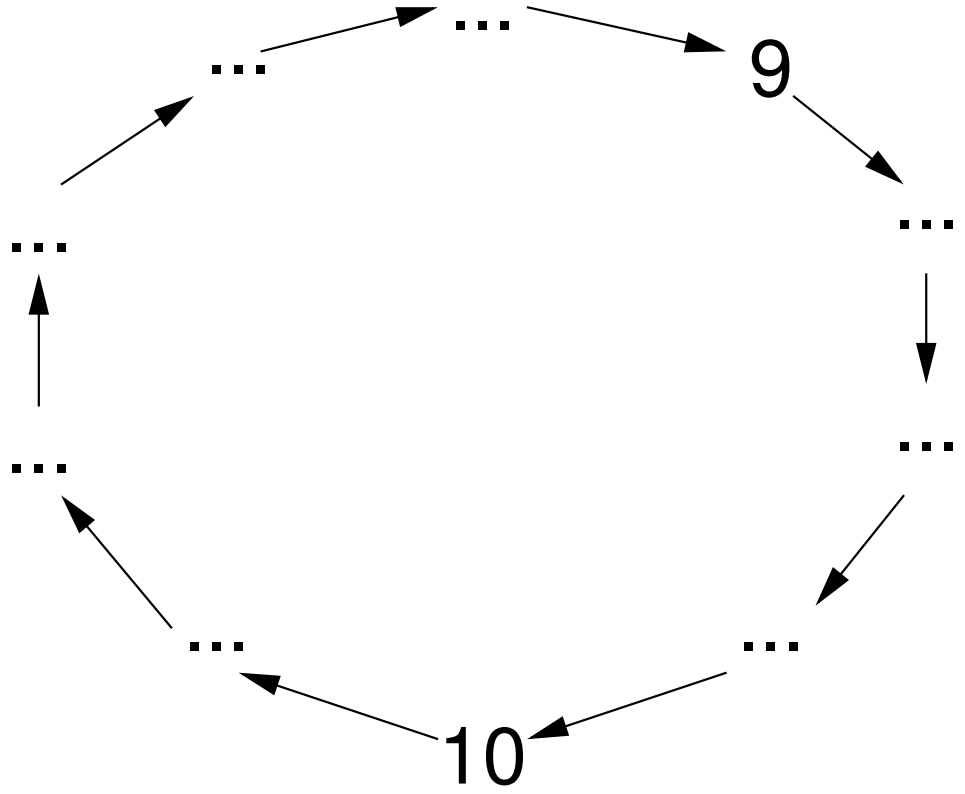
Leader Election: Ergebnis der ersten Runde



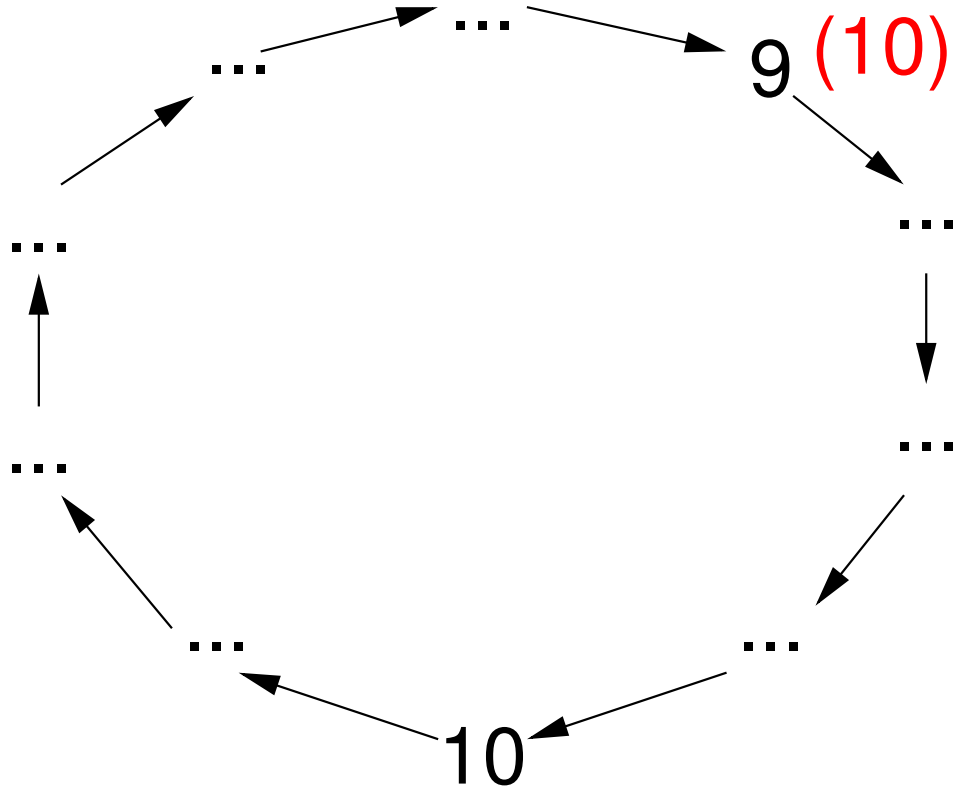
Leader Election: Zweite Runde



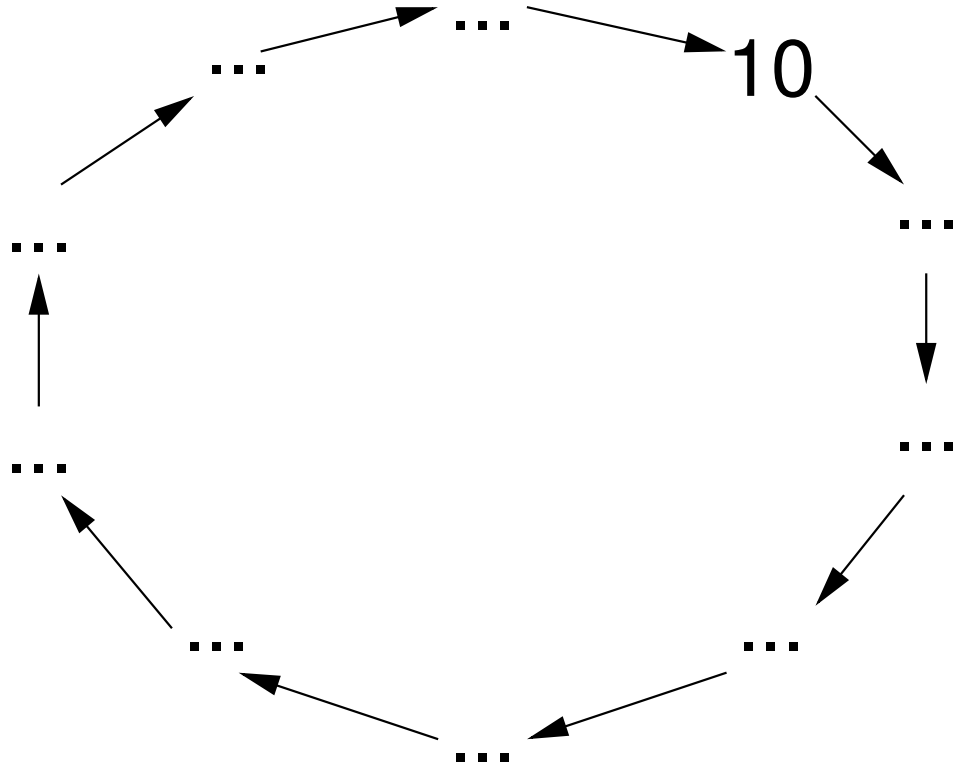
Leader Election: Ergebnis der zweiten Runde



Leader Election: Dritte Runde



Leader Election: Ergebnis



Leader-Election-Protokoll

Motivation: Nur wenige Nachrichten ($\mathcal{O}(n \log n)$ viele).
(Die meisten naiven Ansätze brauchen quadratisch viele Nachrichten!)

Wir lassen Spin prüfen, ob das Protokoll korrekt ist, d.h. es wird ein “Anführer” gefunden: $\mathbf{F}\,\mathbf{G}\,oneLeader$

Tatsächlich teilt uns Spin mit, dass diese Eigenschaft gilt.

Auswirkung der Halbordnungsreduktion

Der Zustandsraum dieses Beispiels wächst exponentiell in n, z.B. können die Prozesse ihre Nachrichten am Anfang in beliebiger Reihenfolge absenden.

Dennoch wächst die Zahl der von Spin untersuchten Zustände nur linear in n.

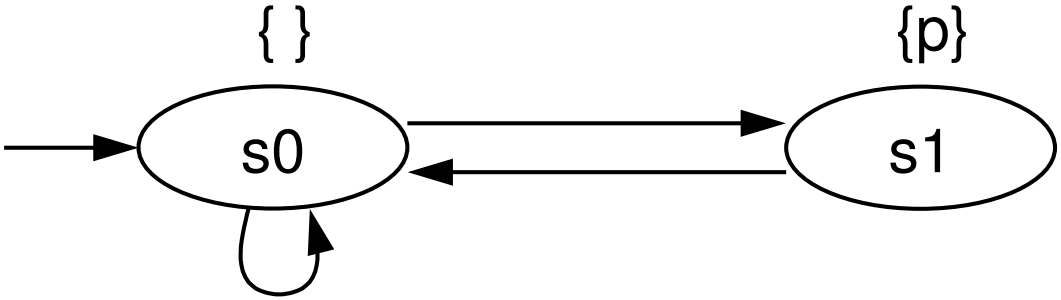
n	mit	ohne
4	136	4327
5	169	28075
6	203	185753
7	237	1.2 Mio

Teil 9: Baum-Zeit-Logik

Motivation

Linear-Zeit-Logik beschreibt Eigenschaften von Abläufen. Sie erlaubt aber nicht, über die *Möglichkeiten* zu sprechen, die ein System hat.

Beispiel: Das Verhalten der untenstehenden Struktur lässt sich nicht adäquat mit LTL beschreiben: Das System kann jederzeit, muss aber nie in den *p*-Zustand wechseln.



Als *Baum-Zeit-Logiken* bezeichnet man i.A. alle Logiken, die erlauben, über Verzweigungen zu reden – im Gegensatz zur Linear-Zeit-Logik gibt es mehrere Möglichkeiten für die Zukunft.

Als einen Vertreter von Baum-Zeit-Logik werden wir die Logik CTL betrachten.

CTL ist die wohl populärste Baum-Zeit-Logik.

- gute Ausdruckskraft
- günstige algorithmische Eigenschaften
- häufige Anwendung in Verifikation

Belegungsbaum

Wir nennen $\mathcal{T} = (V, \rightarrow, r, AP, \nu)$ einen Belegungsbaum gdw.

$(V, \rightarrow)$ ist ein ungeordneter, gerichteter Baum mit Wurzel $r \in V$ (d.h. zu jedem Knoten $v \in V$ gibt es genau einen Pfad von r nach v).

$\nu : V \rightarrow AP$ weist jedem Knoten eine Belegung zu.

Bemerkung: Ein Belegungsbaum ist eine “baumartige” Kripke-Struktur.

Mit $\lceil v \rceil$ bezeichnen wir den Unterbaum, dessen Wurzel $v \in V$ ist.

Berechnungsbaum

Sei $\mathcal{K} = (S, \rightarrow, s_0, AP, \nu)$ eine Kripke-Struktur und $s \in S$.

Mit $\mathcal{T}_{\mathcal{K}}(s)$ bezeichnen wir den Belegungsbaum mit Wurzel r und folgenden Eigenschaften:

$\nu(r) = \nu(s)$

Für jeden Zustand s' mit $s \rightarrow s'$ hat r einen Unterbaum, der isomorph zu $\mathcal{T}_{\mathcal{K}}(s')$ ist.

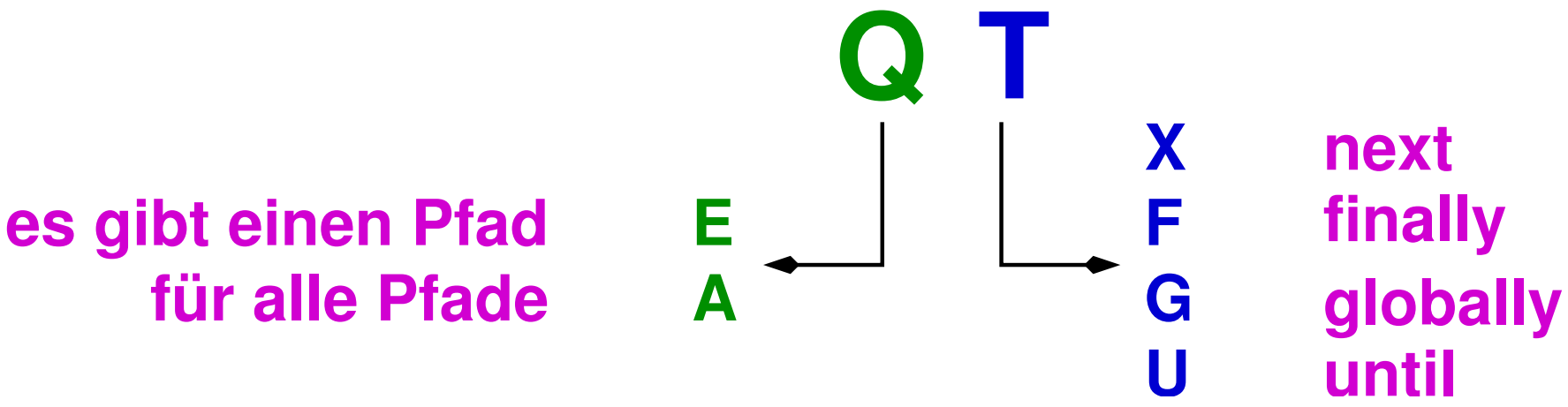
$\mathcal{T}_{\mathcal{K}}(s)$ nennen wir den Berechnungsbaum von s.

CTL: Überblick

CTL = Computation-Tree Logic

Aussagenlogik mit zusätzlichen, quantifizierten Pfad-Operatoren:

Operatoren haben die folgende Form:



CTL: Syntax

Wir definieren zunächst eine minimale Syntax, mit der sich alle Eigenschaften ausdrücken werden.

Sei AP eine Menge von Grundaussagen. Die Menge der CTL-Formeln über AP ist wie folgt:

Ist $p \in AP$, so ist p eine CTL-Formel.

Sind ϕ_1, ϕ_2 CTL-Formeln, so auch

$$\neg\phi_1, \qquad \phi_1 \vee \phi_2, \qquad \mathbf{EX}\,\phi_1, \qquad \mathbf{EG}\,\phi_1, \qquad \phi_1\,\mathbf{EU}\,\phi_2$$

CTL: Semantik

Sei ϕ eine CTL-Formel über AP und $\mathcal{T} = (V, \to, r, AP, \nu)$ ein Belegungsbaum.
Wir schreiben $\mathcal{T} \models \phi$ für "$\mathcal{T}$ erfüllt ϕ". Diese Relation ist wie folgt definiert:

$\mathcal{T} \models$	p	gdw. $p \in AP$ und $\mathbf{p} \in \nu(r)$
$\mathcal{T} \models$	$\neg\phi$	gdw. $\mathcal{T} \not\models \phi$
$\mathcal{T} \models$	$\phi_1 \vee \phi_2$	gdw. $\mathcal{T} \models \phi_1$ oder $\mathcal{T} \models \phi_2$
$\mathcal{T} \models$	$\mathbf{EX}\,\phi$	gdw. $\exists v : r \to v$ und $\lceil v \rceil \models \phi$
$\mathcal{T} \models$	$\mathbf{EG}\,\phi$	gdw. $\mathcal{T}$ hat einen unendl. Pfad $r = v_0 \to v_1 \to \ldots$,
		so dass für alle $i \geq 0$ gilt: $\lceil v_i \rceil \models \phi$
$\mathcal{T} \models$	$\phi_1\,\mathbf{EU}\,\phi_2$	gdw. $\mathcal{T}$ hat einen unendl. Pfad $r = v_0 \to v_1 \to \ldots$,
		so dass $\exists i : \lceil v_i \rceil \models \phi_2 \wedge \forall k < i : \lceil v_k \rceil \models \phi_1$

Die Semantik von ϕ ist definiert als $[\![\phi]\!] = \{\, \mathcal{T} \mid \mathcal{T} \models \phi \,\}$.

Wir bezeichnen zwei Formeln ϕ_1, ϕ_2 als äquivalent (Schreibweise: $\phi_1 \equiv \phi_2$) gdw. $[\![\phi_1]\!] = [\![\phi_2]\!]$.

Außerdem schreiben wir $\phi_1 \Rightarrow \phi_2$ gdw. $[\![\phi_1]\!] \subseteq [\![\phi_2]\!]$.

Sei $\mathcal{K} = (S, \rightarrow, s_0, AP, \nu)$ eine Kripke-Struktur und ϕ eine CTL-Formel. Wir können jetzt sagen, was es bedeutet, dass eine Kripke-Struktur eine Formel erfüllt:

Definition: $\mathcal{K}$ erfüllt ϕ (Schreibweise: $\mathcal{K} \models \phi$) gdw. $\mathcal{T}_{\mathcal{K}}(s_0) \models \phi$.
Außerdem definieren wir $[\![\phi]\!]_{\mathcal{K}} := \{ s \in S \mid \mathcal{T}_{\mathcal{K}}(s) \models \phi \}$.

Model-Checking-Problem für CTL: Gegeben $\mathcal{K}, \phi$, prüfe, ob $\mathcal{K} \models \phi$ gilt.
Globales MC-Problem für CTL: Gegeben $\mathcal{K}, \phi$, berechne $[\![\phi]\!]_{\mathcal{K}}$.

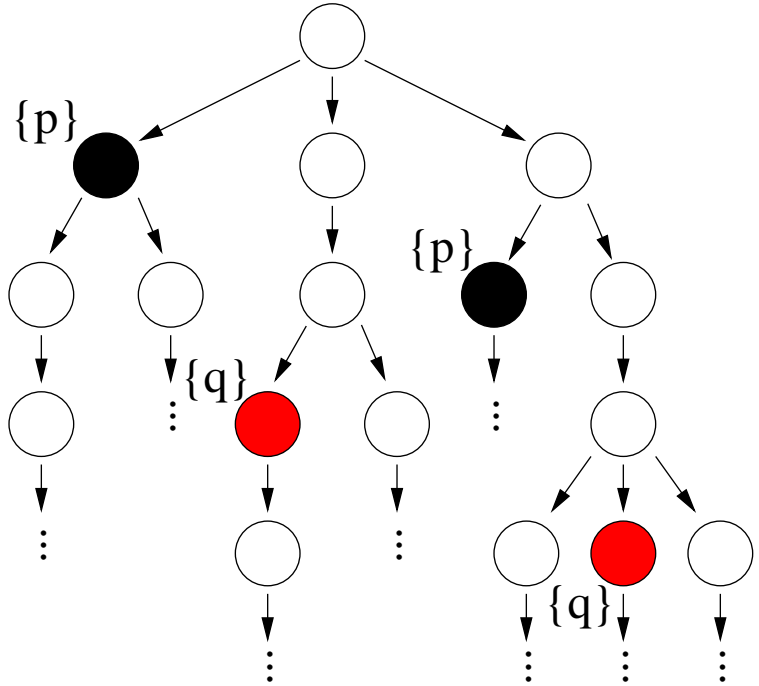
CTL: Erweiterte Syntax

Wir vereinbaren folgende abkürzende Schreibweisen: (weitere logische Operatoren können analog vereinbart werden)

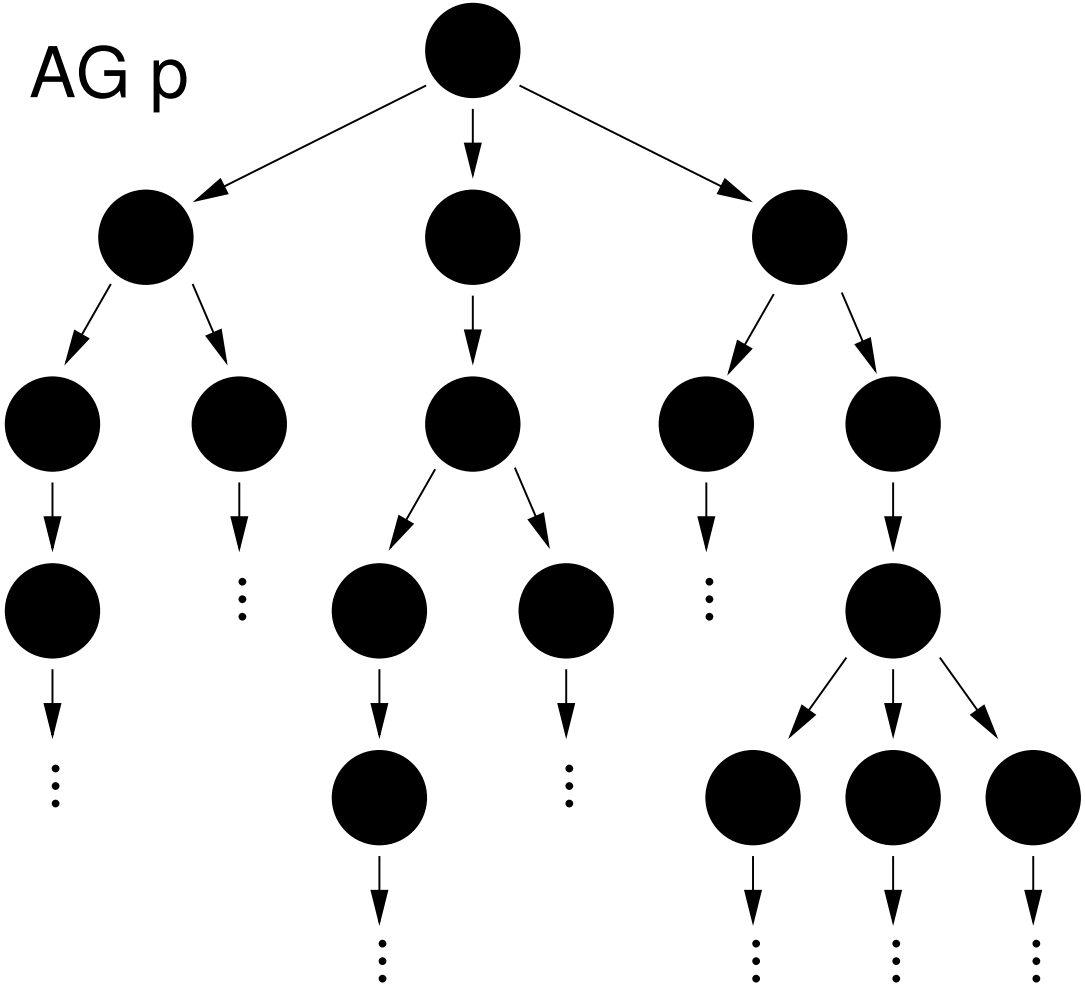
$$
\begin{aligned}
\phi_1 \wedge \phi_2 &\equiv \neg(\neg\phi_1 \vee \neg\phi_2) \\
\mathbf{true} &\equiv a \vee \neg a \\
\mathbf{false} &\equiv \neg\mathbf{true} \\
\phi_1\,\mathbf{EW}\,\phi_2 &\equiv \mathbf{EG}\,\phi_1 \vee (\phi_1\,\mathbf{EU}\,\phi_2) \\
\mathbf{EF}\,\phi &\equiv \mathbf{true}\,\mathbf{EU}\,\phi
\end{aligned}
\qquad
\begin{aligned}
\mathbf{AX}\,\phi &\equiv \neg\,\mathbf{EX}\,\neg\phi \\
\mathbf{AG}\,\phi &\equiv \neg\,\mathbf{EF}\,\neg\phi \\
\mathbf{AF}\,\phi &\equiv \neg\,\mathbf{EG}\,\neg\phi \\
\phi_1\,\mathbf{AW}\,\phi_2 &\equiv \neg(\neg\phi_2\,\mathbf{EU}\,\neg(\phi_1 \vee \phi_2)) \\
\phi_1\,\mathbf{AU}\,\phi_2 &\equiv \mathbf{AF}\,\phi_2 \wedge (\phi_1\,\mathbf{AW}\,\phi_2))
\end{aligned}
$$

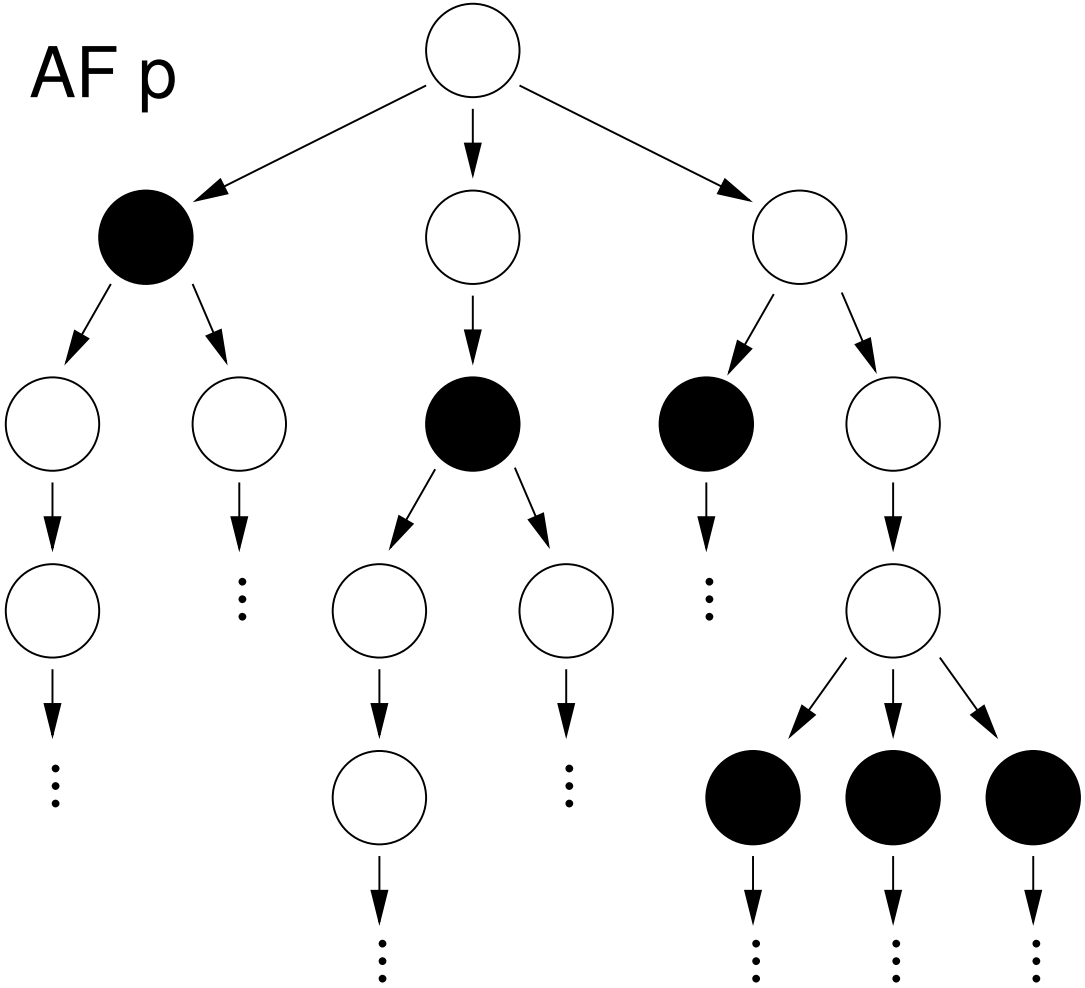
CTL: Beispiele

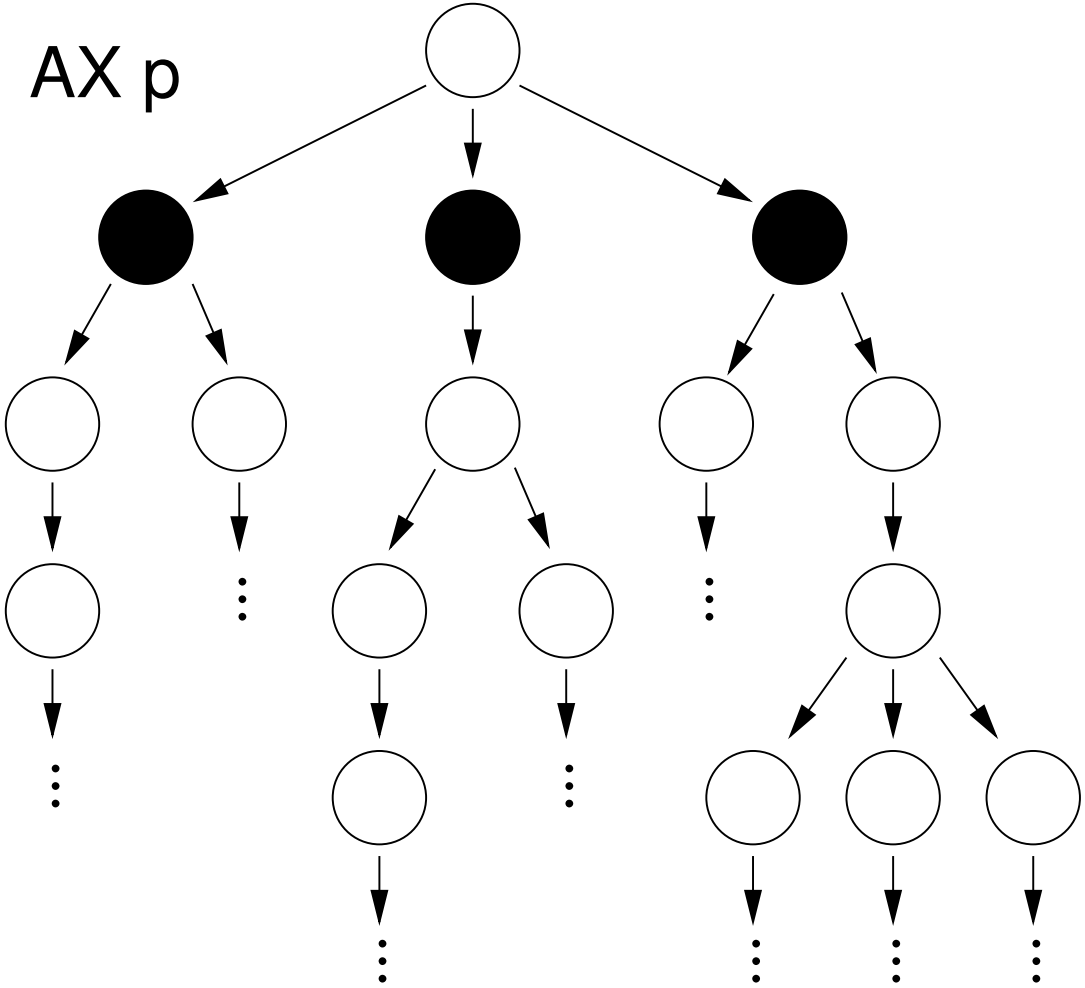
Im Folgenden wird der untenstehende Baum verwendet, um die Bedeutung verschiedener CTL-Operatoren zu verdeutlichen:

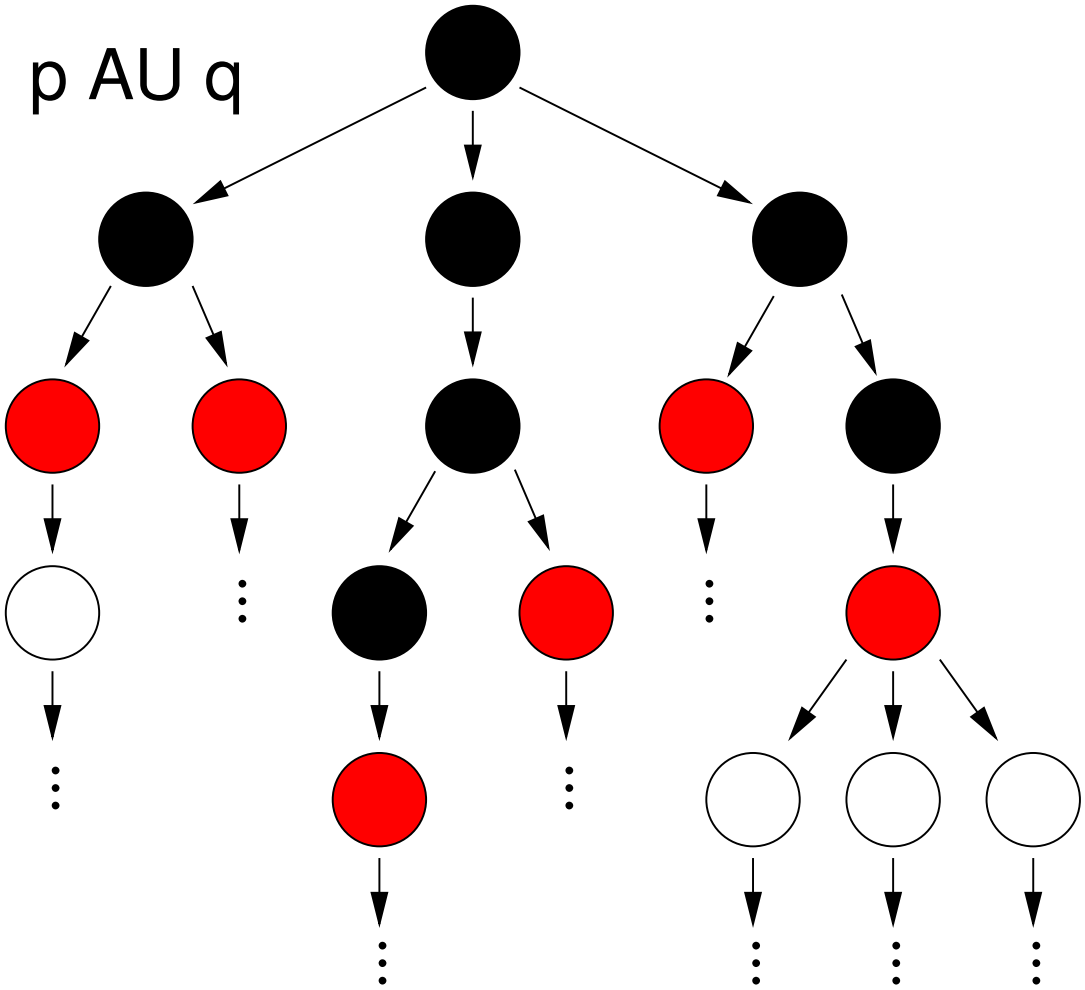


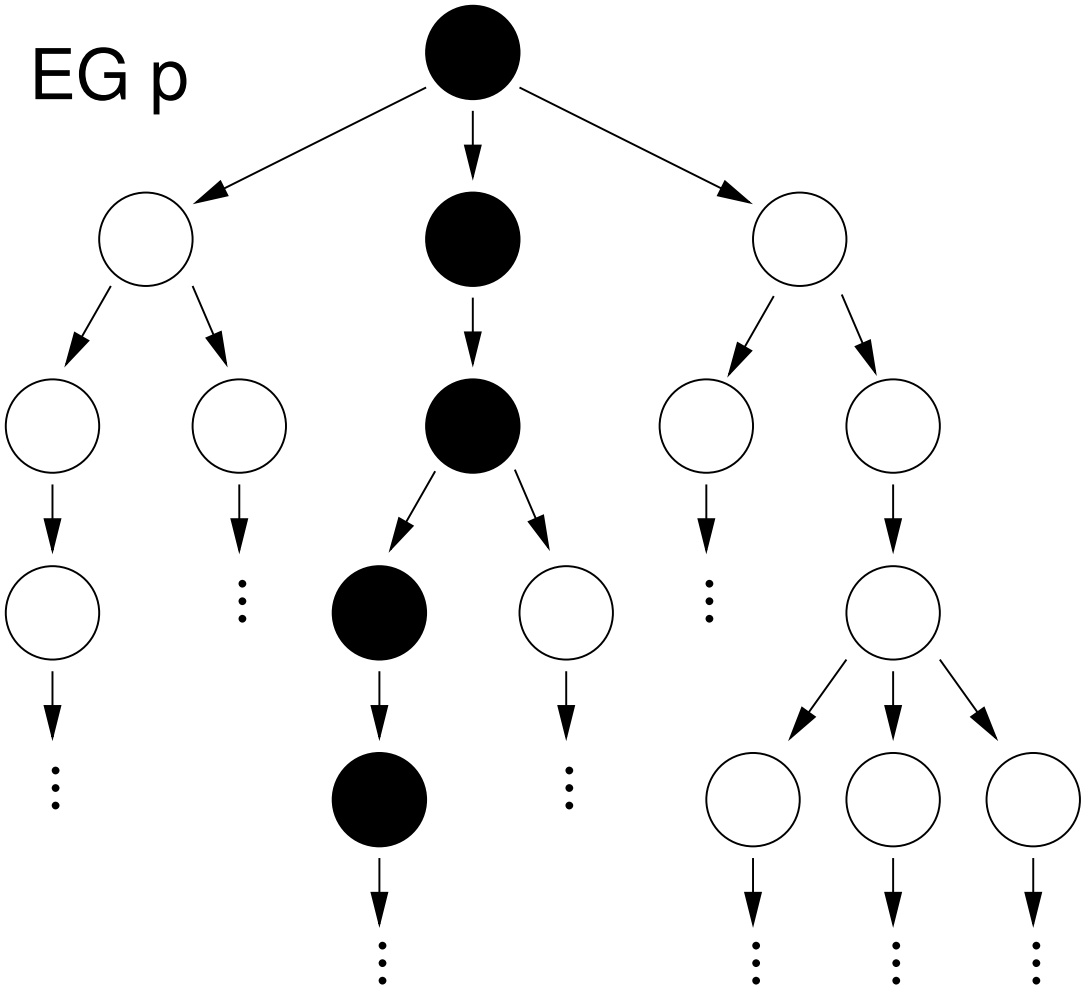
Auf den folgenden Folien erfüllen die schwarzen Zustände jeweils *p*, die roten *q*, und der Baum die angegebene Formel.

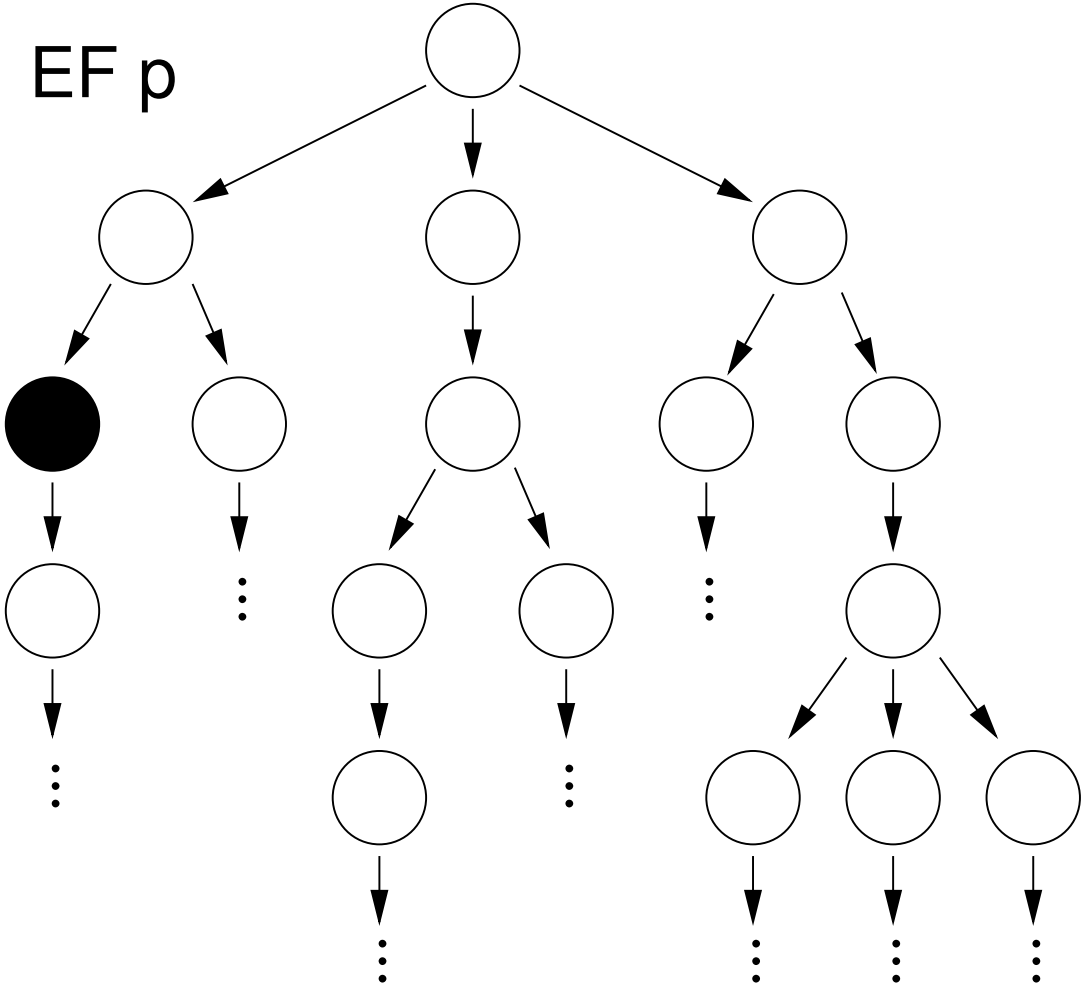

AG p

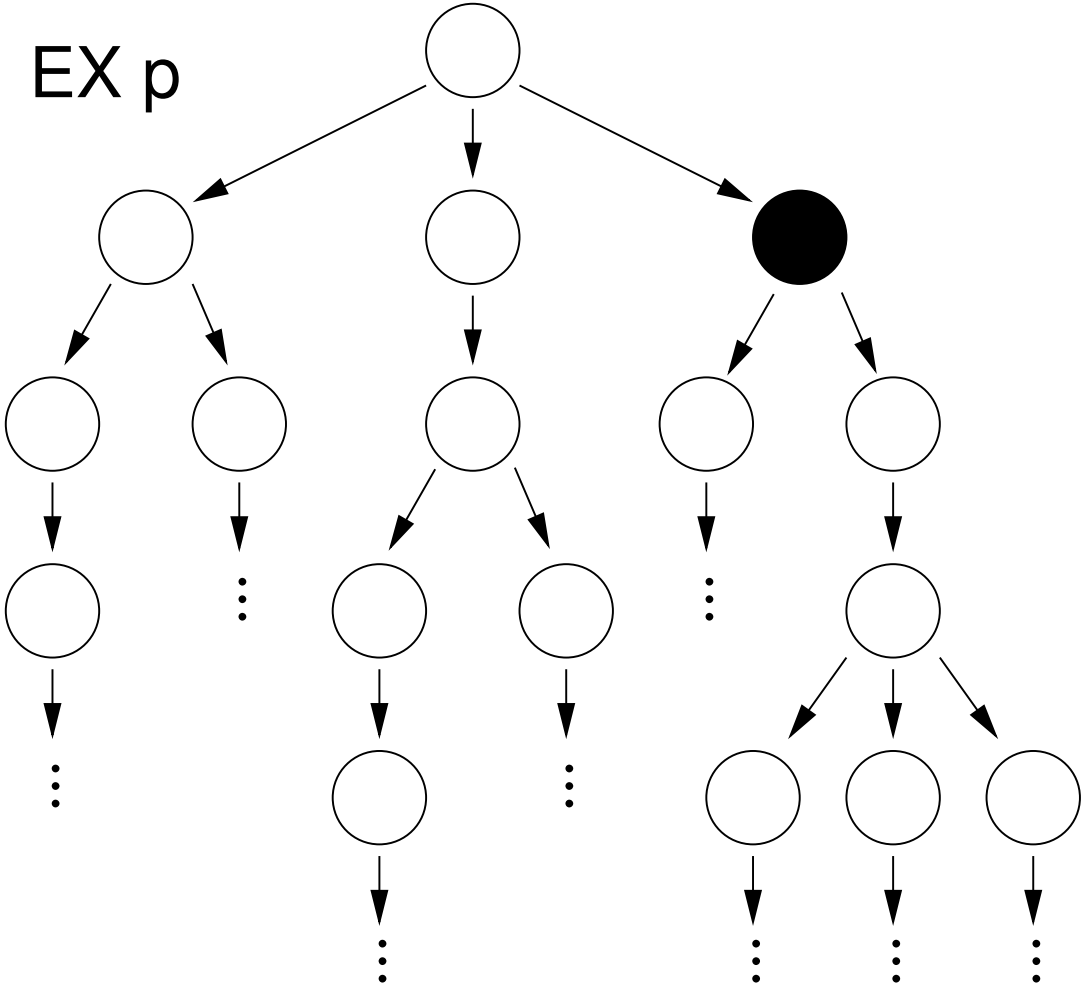

AF p

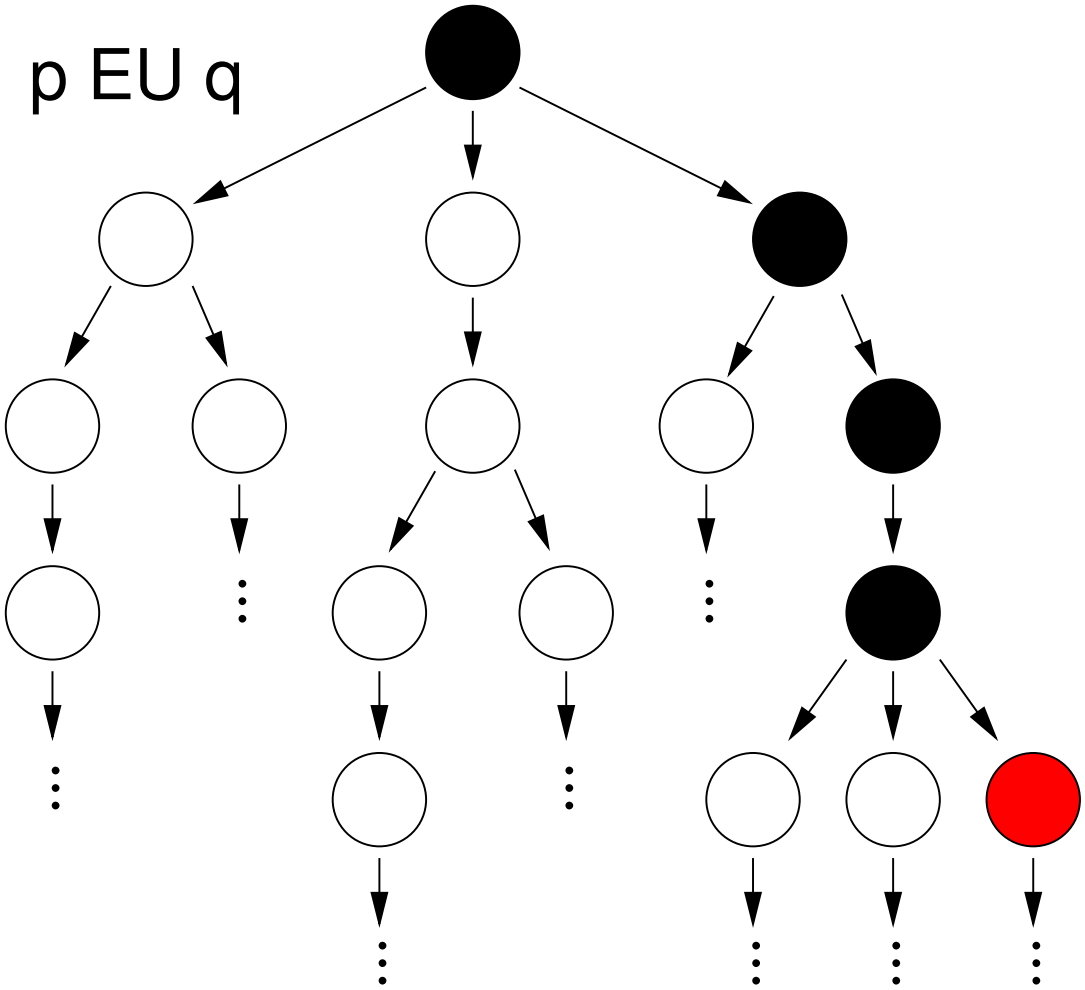

AX p


p AU q


EG p


EF p


EX p


p EU q

Lösung des (globalen) MC-Problems

Um das MC-Problem zu lösen, muss $\mathcal{T}_{\mathcal{K}}(s_0)$ nicht explizit konstruiert werden.

Die Semantik fragt ab, ob gewisse Unterbäume von $\mathcal{T}_{\mathcal{K}}(s_0)$ gewisse Teilformeln ϕ' erfüllen. Beobachtung:

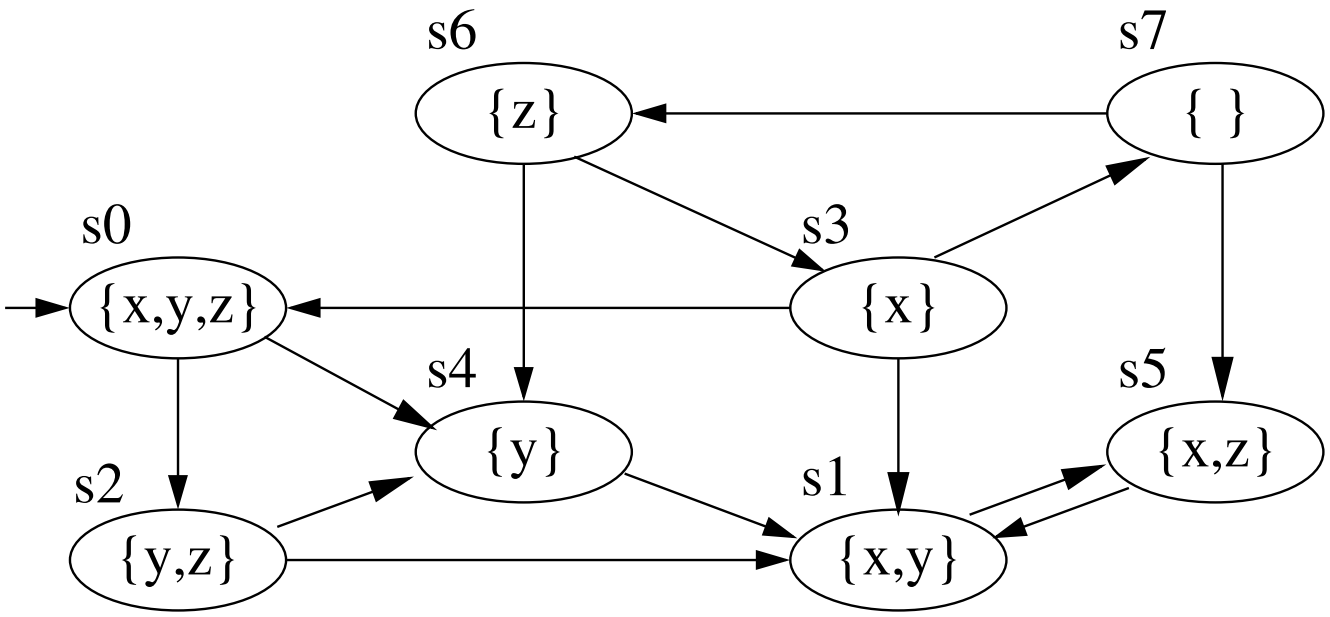
Jeder Unterbaum von $\mathcal{T}_{\mathcal{K}}(s_0)$ ist genau $\mathcal{T}_{\mathcal{K}}(s)$ für irgendein $s \in S$.

$\mathcal{T}_{\mathcal{K}}(s) \models \phi'$ gdw. $s \in [\![\phi']\!]_{\mathcal{K}}$

Das MC-Problem kann also gelöst werden, indem man das *globale* MC-Problem für die Teilformeln löst (erst für einfache, dann für kompliziertere Formeln, sozusagen bottom-up).

Zum Schluss prüft man, ob $s_0 \in [\![\phi]\!]_{\mathcal{K}}$.

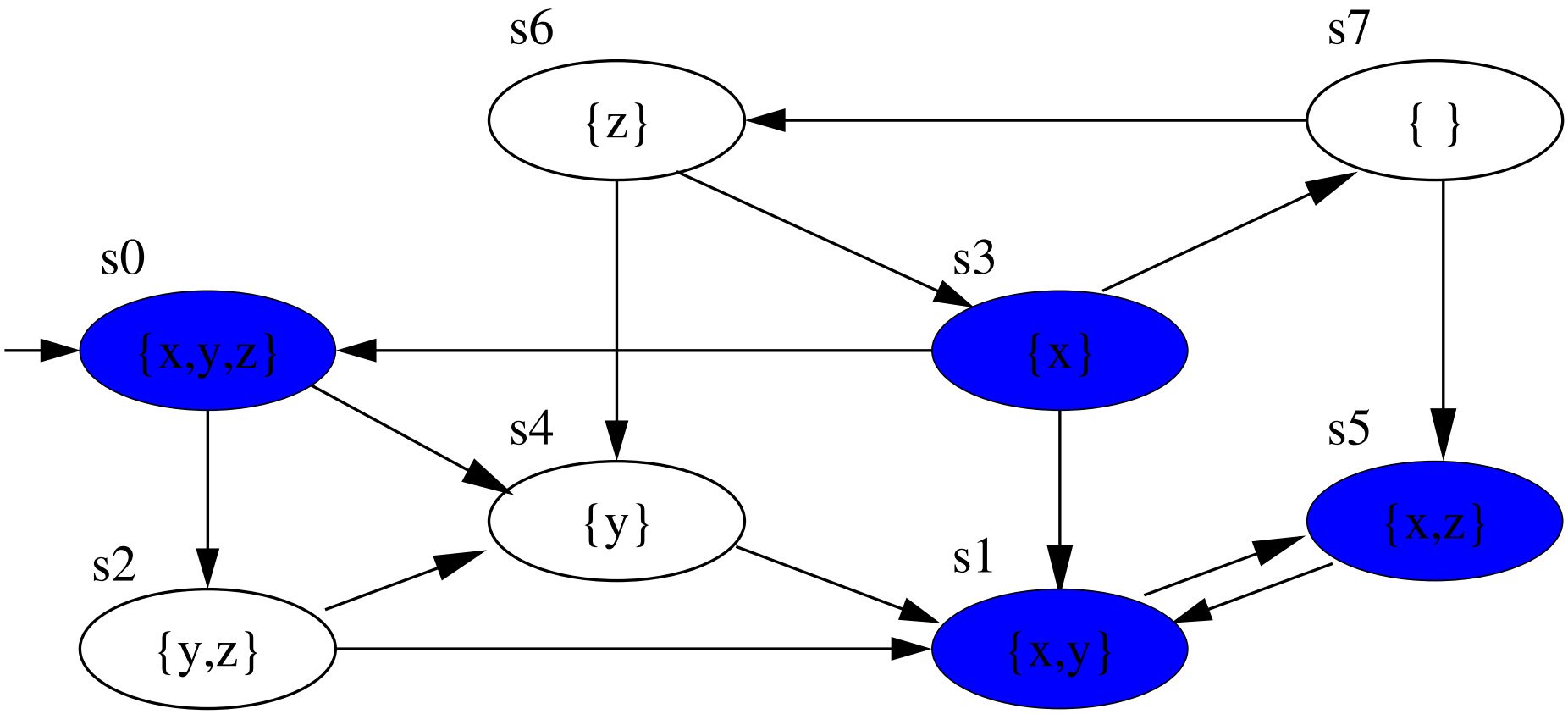
Beispiel



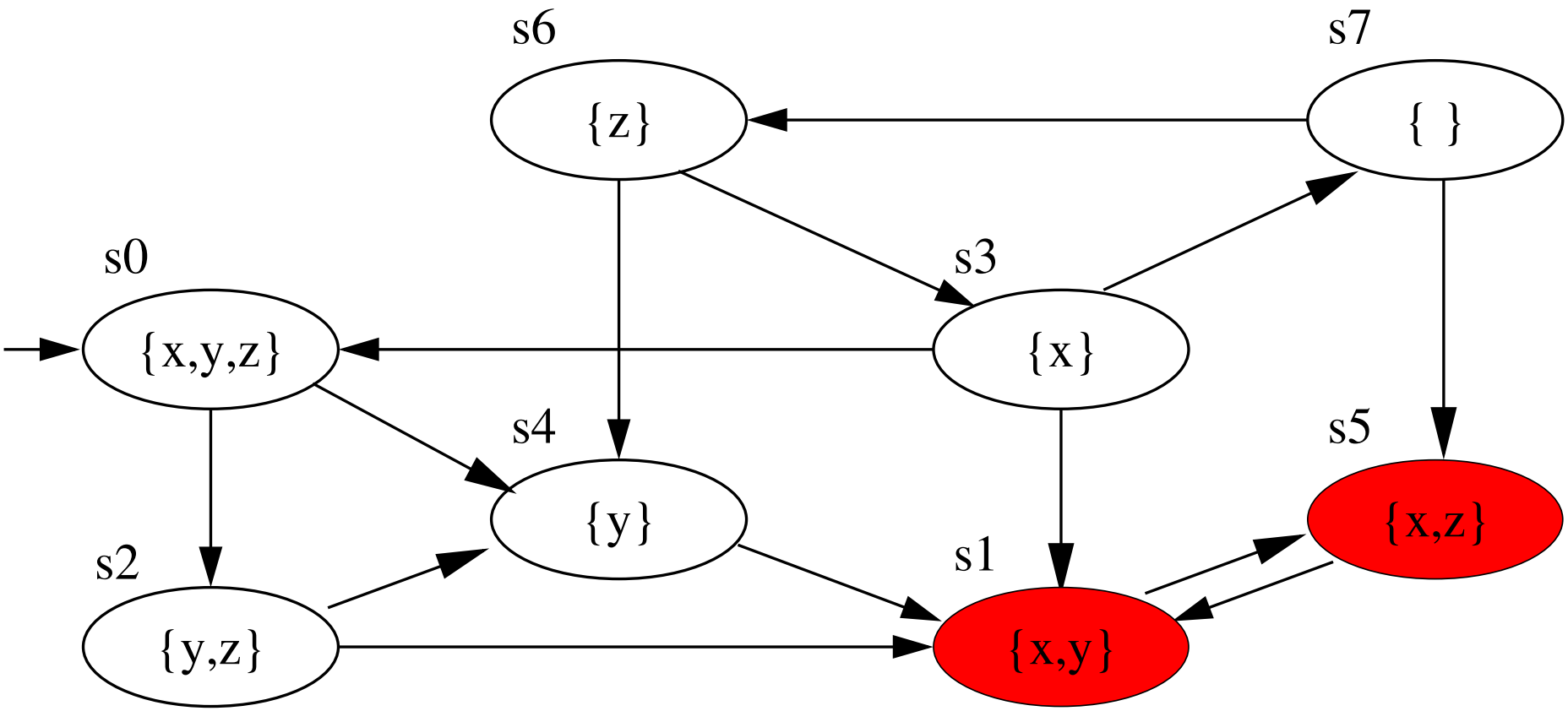
Im Folgenden wollen wir prüfen, ob die obige Struktur $\mathcal{K}$ $\mathbf{AF}\,\mathbf{AG}\,x$ erfüllt.

Dazu berechnen wir nacheinander $[\![x]\!]_{\mathcal{K}}$, $[\![\mathbf{AG}\,x]\!]_{\mathcal{K}}$ und $[\![\mathbf{AF}\,\mathbf{AG}\,x]\!]_{\mathcal{K}}$.

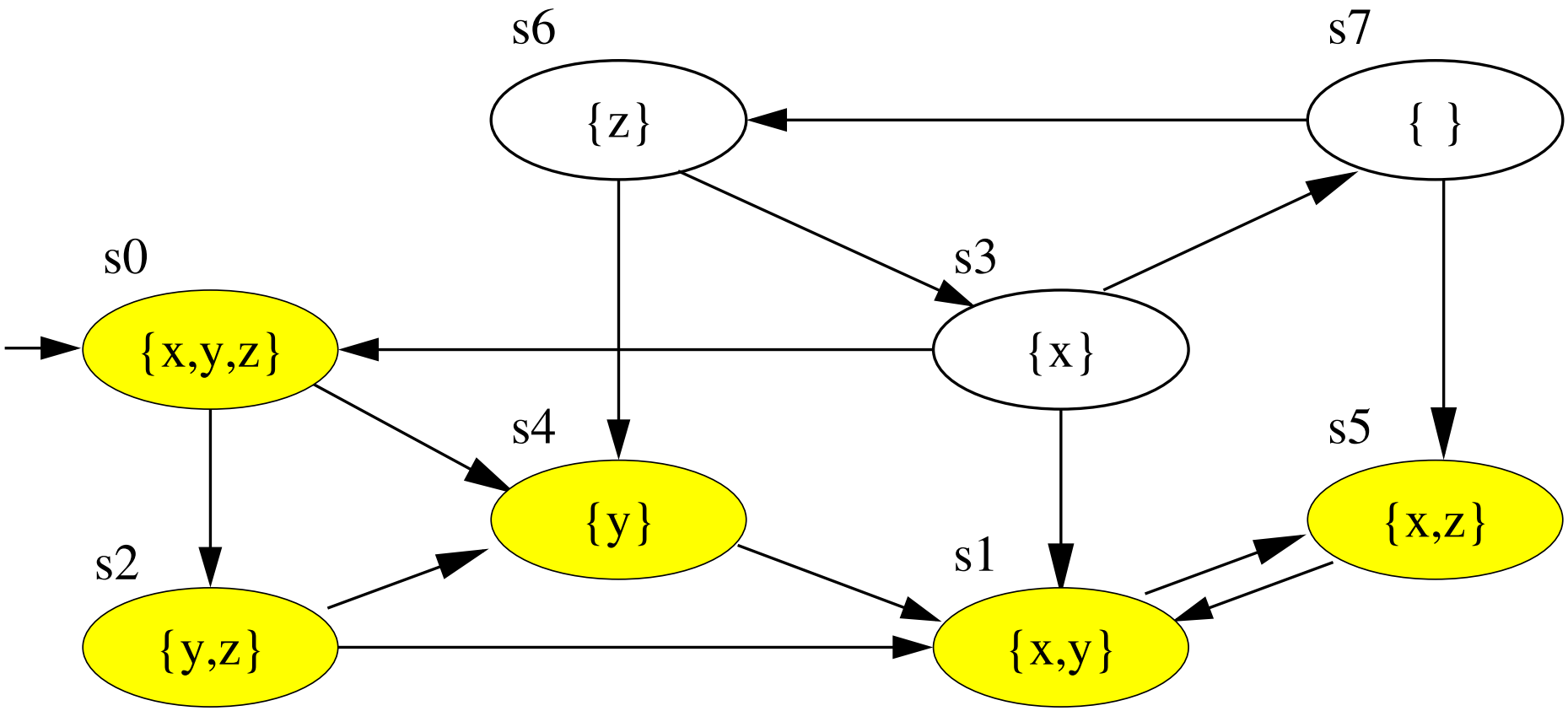
Bottom-up-Methode (1): Berechne $[\![x]\!]_{\mathcal{K}}$



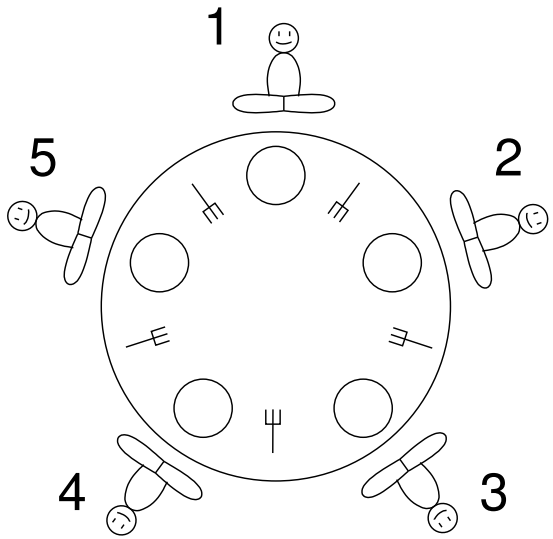
Bottom-up-Methode (2): Berechne $[\![\mathbf{AG}\, x]\!]_{\mathcal{K}}$



Bottom-Up-Methode (3): Berechne $[\![\mathbf{AF}\,\mathbf{AG}\,x]\!]_{\mathcal{K}}$



Beispiel: Speisende Philosophen



Fünf Philosophen sitzen um einen Tisch, abwechselnd denkend und essend.

Wir werden ein paar Aussagen mit Hilfe von CTL formulieren. Dabei verwenden wir folgende Grundaussagen:

$e_i \mathrel{\widehat{=}}$ Philosoph i isst gerade

Aussagen über die Speisenden Philosophen

“Philosophen 1 und 4 werden niemals zugleich essen.”

$$\mathbf{AG}\,\neg(e_1 \wedge e_4)$$

“Möglicherweise wird Philosoph 3 niemals essen.”

$$\mathbf{EG}\,\neg e_3$$

“Aus jeder Situation heraus ist es möglich, dass ein Zustand entsteht, in dem nur Philosoph 2 isst.”

$$\mathbf{AG}\,\mathbf{EF}(\neg e_1 \wedge e_2 \wedge \neg e_3 \wedge \neg e_4)$$

Teil 10: Algorithmen für CTL

CTL-Model-Checking

Im Folgenden sei $\mathcal{K} = (S, \rightarrow, s_0, AP, \nu)$ eine Kripke-Struktur (wobei S endlich ist) und ϕ eine CTL-Formel über AP.

Wir werden das *globale* Model-Checking-Problem lösen, d.h. die Berechnung von $[\![\phi]\!]_{\mathcal{K}}$ (alle Zustände von $\mathcal{K}$, deren Berechnungsbaum ϕ erfüllt).

Dazu benutzen wir den “Bottom-Up”-Ansatz (erst einfache Teilformeln betrachten, dann komplexere).

Wir zeigen nur die Algorithmen für die minimale Syntax, für größere Effizienz könnte man spezielle Algorithmen für die erweiterte Syntax betrachten.

Das ‘Bottom-Up’-Prinzip

Der Algorithmus reduziert ϕ nach und nach auf eine einzelne Grundaussage, mit folgenden Schritten. Zur Erinnerung: $[\![p]\!]_{\mathcal{K}} = \{ s \mid p \in \nu(s) \}$ für $p \in AP$. Im Folgenden schreiben wir dafür kurz $\mu(p)$.

1. Prüfe, ob ϕ bereits aus einer einzelnen Grundaussage p besteht. Falls ja, gib $\mu(p)$ aus und höre auf.

2. Andernfalls besitzt ϕ irgendeine Teilformel ψ der Form $\neg p$, $p \vee q$, $\mathbf{EX}\, p$, $\mathbf{EG}\, p$ oder $p\ \mathbf{EU}\ q$, wobei $p, q \in AP$ sind. Berechne $[\![\psi]\!]_{\mathcal{K}}$ mit den noch folgenden Algorithmen.

3. Sei $p' \notin AP$ eine “frische” Grundaussage. Erweitere AP um p' und setze $\mu(p') := [\![\psi]\!]_{\mathcal{K}}$. Ersetze alle Vorkommen von ψ in ϕ durch p' und fahre fort mit Schritt 1.

Berechnung von $[\![\psi]\!]_{\mathcal{K}}$: einfache Fälle

Fall 1: $\psi \equiv \neg p, \quad p \in AP$

Per Definition gilt $[\![\psi]\!]_{\mathcal{K}} = S \setminus \mu(p)$.

Fall 2: $\psi \equiv p \vee q, \quad p, q \in AP$

Dann ist $[\![\psi]\!]_{\mathcal{K}} = \mu(p) \cup \mu(q)$.

Fall 3: $\psi \equiv \mathbf{EX}\, p, \quad p \in AP$

Im Folgenden bezeichne $pre(X)$, für $X \subseteq S$, die Menge

$$pre(X) := \{ s \mid \exists t \in X : s \to t \}.$$

Dann ist $[\![\psi]\!]_{\mathcal{K}} = pre(\mu(p))$.

Berechnung von $[\![\psi]\!]_{\mathcal{K}}$: die Fälle $\mathbf{EU}$ und $\mathbf{EG}$

Für $\mathbf{EU}$ und $\mathbf{EG}$ werden wir zunächst eine mengentheoretische Fixpunkt-Charakterisierung geben.

$\mathbf{EU}$ ist als kleinster Fixpunkt charakterisiert: Wir gehen zunächst davon aus, dass kein Zustand die $\mathbf{EU}$-Formel erfüllt und finden dann nach und nach diejenigen, die es doch tun.

$\mathbf{EG}$ ist als größter Fixpunkt charakterisiert: Wir gehen zunächst davon aus, dass alle Zustände die $\mathbf{EG}$-Formel erfüllen und finden dann nach und nach diejenigen, die es doch nicht tun.

Danach geben wir effiziente Algorithmen an. Die Fixpunkt-Charakterisierung wird uns aber später noch sehr nützlich sein.

Berechnung von $[\![\psi]\!]_{\mathcal{K}}$: $\mathbf{EG}$

Fall 4: $\psi \equiv \mathbf{EG}\, p, \quad p \in AP$

Lemma 1: $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ ist die größte Lösung (bzgl. $\subseteq$) der Gleichung

$$X = \mu(p) \cap pre(X).$$

Beweis: Wir gehen in zwei Schritten vor.

1. Wir zeigen, dass $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ *is* eine Lösung obiger Gleichung ist, d.h.

$$[\![\mathbf{EG}\, p]\!]_{\mathcal{K}} = \mu(p) \cap pre([\![\mathbf{EG}\, p]\!]_{\mathcal{K}}).$$

Zur Erinnerung: $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}} = \{\, s \mid \exists \rho\colon \rho(0) = s \wedge \forall i \geq 0\colon \rho(i) \in \mu(p) \,\}$.

"$\Rightarrow$" Sei $s \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ und ρ ein Pfad, der dies bezeugt. Dann folgt $s \in \mu(p)$ sofort. Außerdem ist $\rho(1) \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ (wegen ρ^1), deshalb gilt $s \in pre([\![\mathbf{EG}\, p]\!]_{\mathcal{K}})$.

Fortsetzung des Beweises von Lemma 1:

1. "$\Leftarrow$" Sei $s \in \mu(p) \cap pre([\![\mathbf{EG}\, p]\!]_{\mathcal{K}})$. Dann hat s einen direkten Nachfolger t, bei dem ein Pfad ρ beginnt, der beweist, dass $t \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ ist. Dann ist $s\rho$ ein Pfad, der $s \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ beweist.

2. Wir zeigen, dass $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ auch die *größte* Lösung ist. D.h., falls M eine Lösung der Gleichung ist, dann gilt $M \subseteq [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

Sei $M \subseteq S$ eine Lösung, d.h. $M = \mu(p) \cap pre(M)$, und sei $s \in M$. Zu zeigen ist $s \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

- Da $s \in M$, gilt $s \in \mu(p)$ und $s \in pre(M)$.
- Da $s \in pre(M)$, gibt es $s_1 \in M$ mit $s \rightarrow s_1$.
- Durch Wiederholung dieses Arguments bekommen wir einen unendlichen Pfad $\rho = ss_1 \cdots$, in dem alle Zustände in $\mu(p)$ sind. Also gilt $s \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

Lemma 2: Wir betrachten die Sequenz $S,\ \pi(S),\ \pi(\pi(S)),\ \ldots$, i.e. $\left(\pi^i(S)\right)_{i\geq 0}$,
wobei $\pi(X) := \mu(p) \cap pre(X)$.
Für alle $i \geq 0$ gilt $\pi^i(S) \supseteq [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

Zunächst machen wir zwei Feststellungen:

(1) π ist *monoton*: falls $X \supseteq X'$, dann $\pi(X) \supseteq \pi(X')$.

(2) Die Sequenz ist *absteigend*: $S \supseteq \pi(S) \supseteq \pi(\pi(S)) \ldots$ (folgt aus (1)).

Beweis von Lemma 2: (Induktion über i)
Anfang: $i = 0$: offensichtlich.
Schritt: $i \rightarrow i + 1$:

$$\begin{aligned}
\pi^{i+1}(S) &= \mu(p) \cap pre(\pi^i(S)) \\
&\supseteq \mu(p) \cap pre([\![\mathbf{EG}\, \varphi]\!]_{\mathcal{K}}) \quad \text{(I.V. und Monotonizität)} \\
&= [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}
\end{aligned}$$

Lemma 3: Es gibt einen Index i mit $\pi^i(S) = \pi^{i+1}(S)$, und $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}} = \pi^i(S)$.

Beweis: Da S endlich ist, muss die absteigende Sequenz einen Fixpunkt erreichen, sagen wir nach i Schritten, so dass gilt:
$\pi^i(S) = \pi(\pi^i(S)) = \mu(p) \cap pre(\pi^i(S))$. Also ist $\pi^i(S)$ eine Lösung der Gleichung aus Lemma (1).

Wegen Lemma 1 gilt $\pi^i(S) \subseteq [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.
Wegen Lemma 2 gilt $\pi^i(S) \supseteq [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

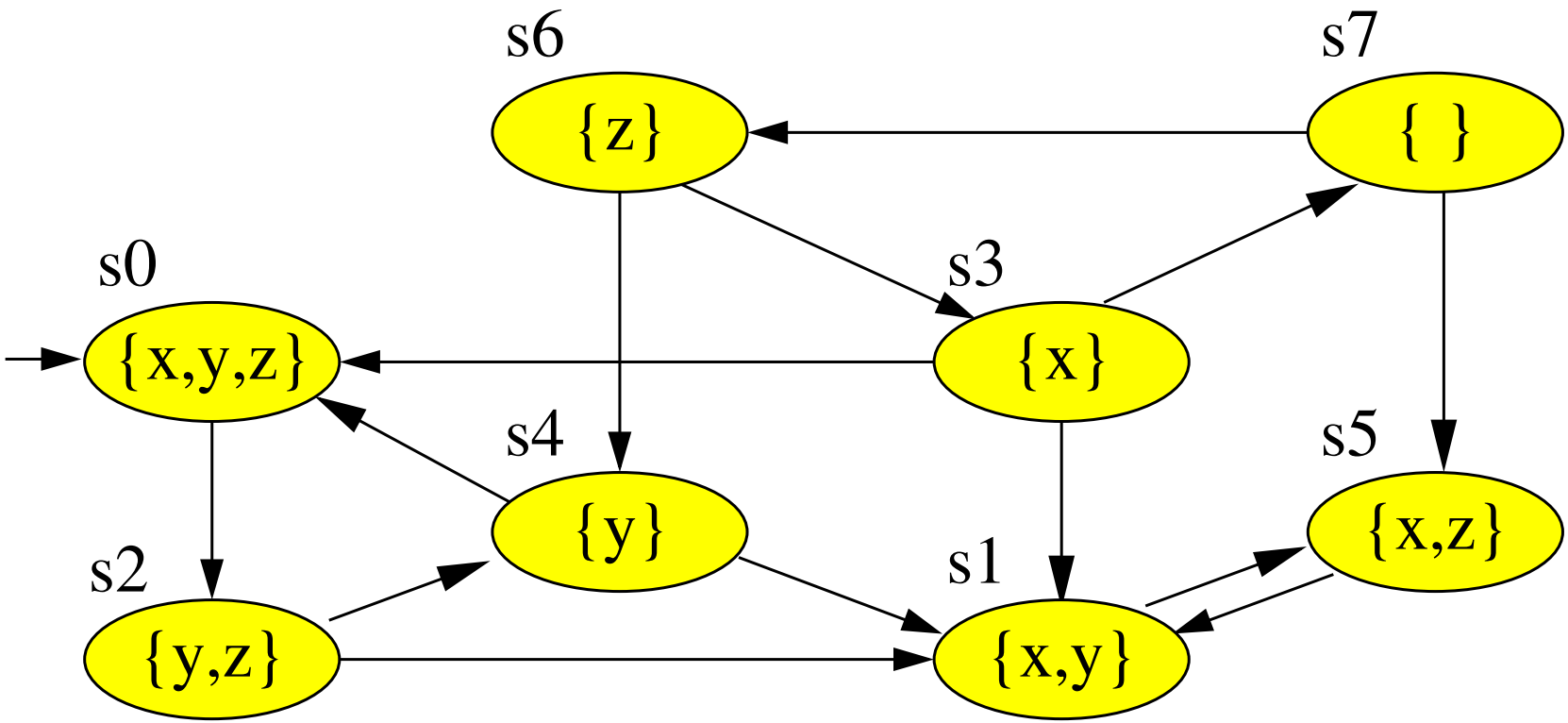
Berechnung der $\mathbf{EG}$-Semantik

Lemma 3 gibt uns einen Algorithmus vor, um $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ zu berechnen: Erstelle die Sequenz $S,\ \pi(S),\ \cdots$, bis ein Fixpunkt erreicht ist.

Praktischerweise wird man gleich mit $X := \mu(p)$ beginnen und in jeder Runde diejenigen Zustände entfernen, die keinen direkten Nachfolger in X haben.

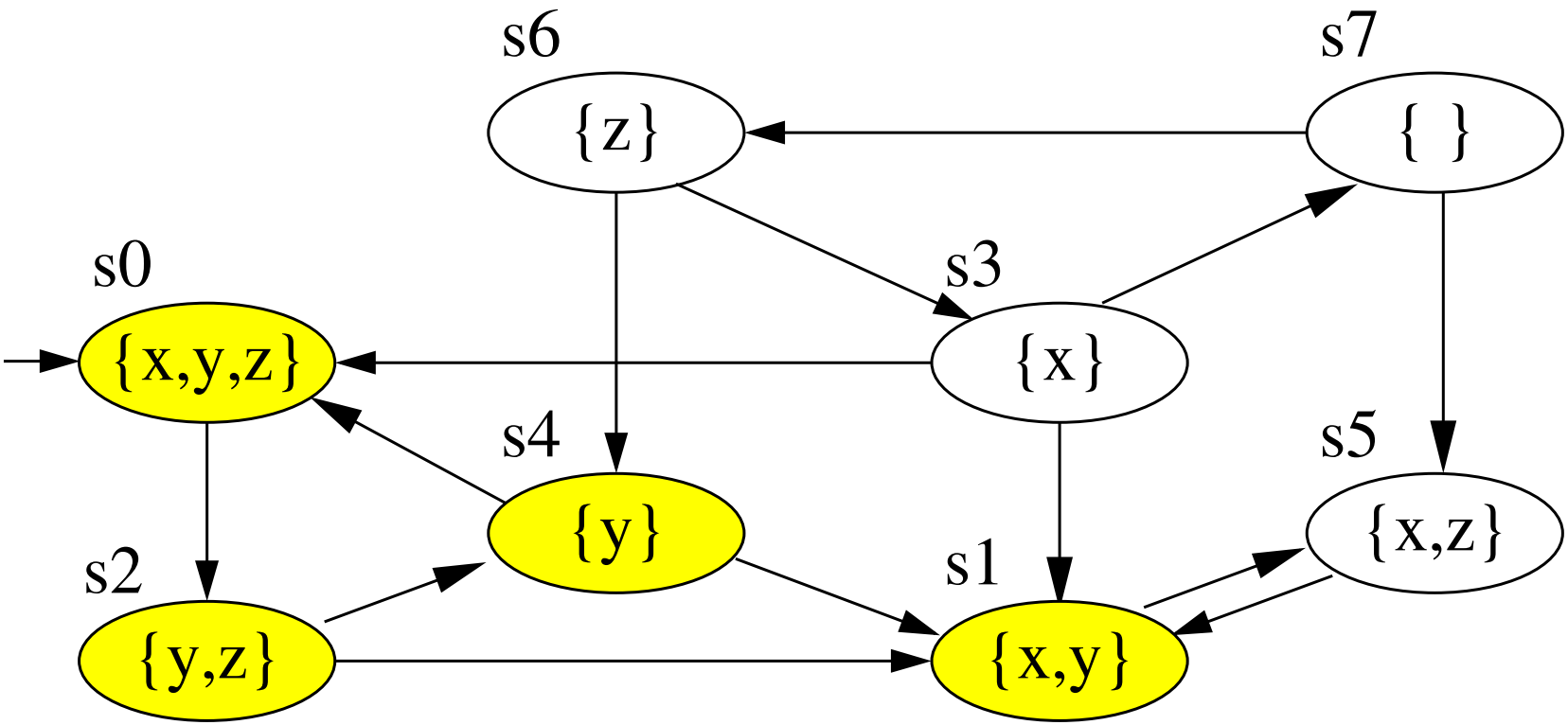
Kann effizient in $\mathcal{O}(|\mathcal{K}|)$ Zeit implementiert werden (“reference counting”).

Beispiel: Berechnung von $[\![\mathbf{EG}\, y]\!]_{\mathcal{K}}$ (1/4)



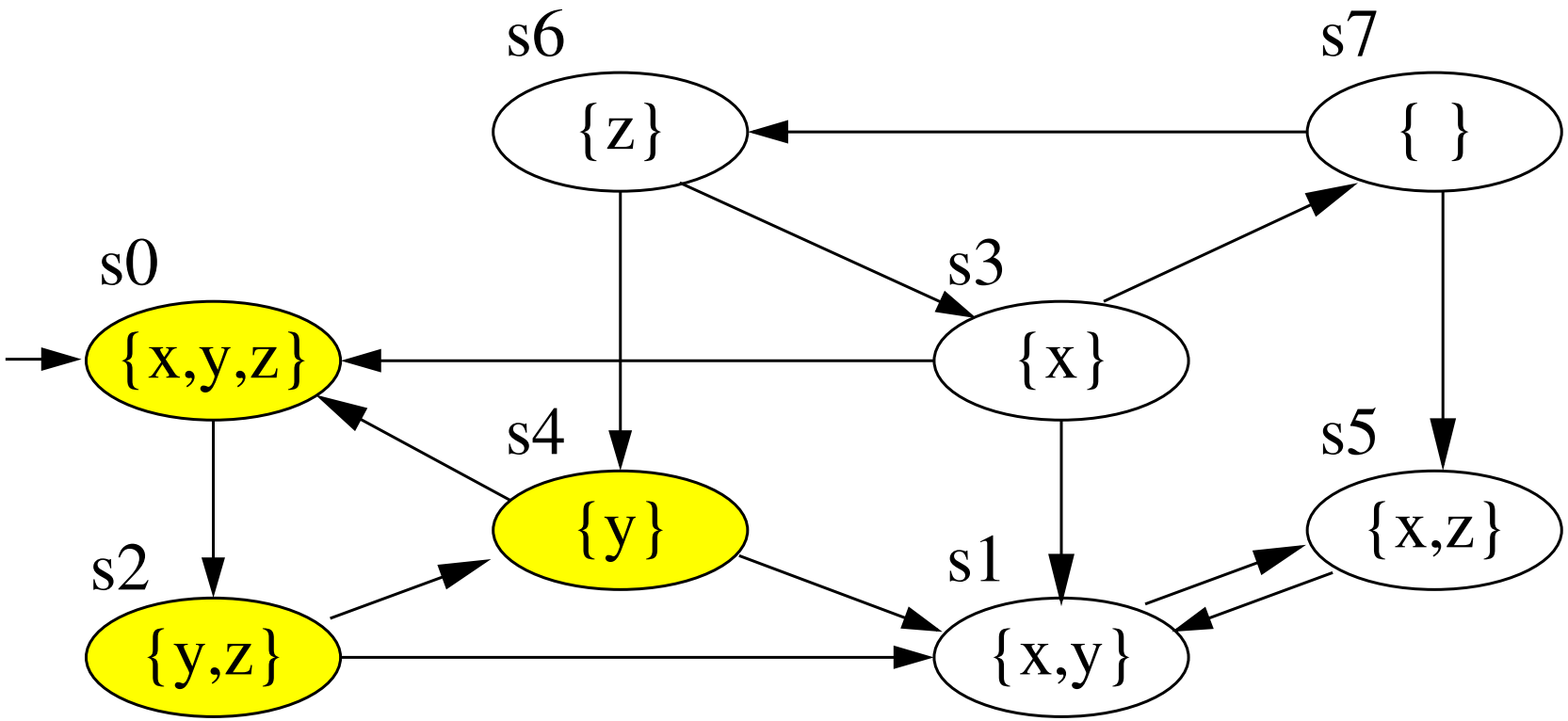
$\pi^0(s) = s$

Beispiel: Berechnung von $[\![\mathbf{EG}\, y]\!]_{\mathcal{K}}$ (2/4)



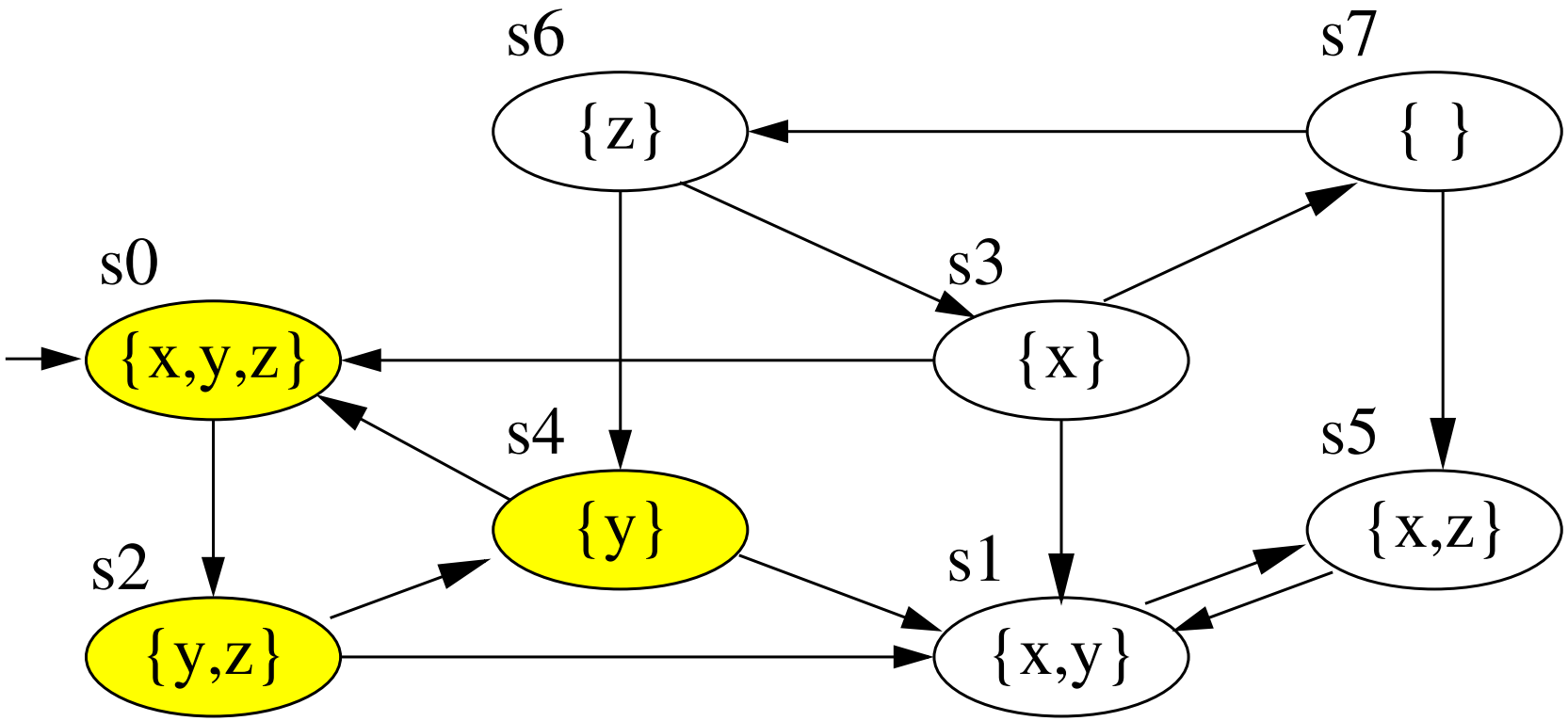
$\pi^1(S) \;=\; \mu(y) \cap pre(S)$

Beispiel: Berechnung von $[\![\mathbf{EG}\, y]\!]_{\mathcal{K}}$ (3/4)



$\pi^2(S) \;=\; \mu(y) \cap pre(\pi^1(S))$

Beispiel: Berechnung von $[\![\mathbf{EG}\, y]\!]_{\mathcal{K}}$ (4/4)



$\pi^3(S) \;=\; \mu(y) \cap pre(\pi^2(S)) \;=\; \pi^2(S)$: $\quad [\![\mathbf{EG}\, y]\!]_{\mathcal{K}} = \{s_0, s_2, s_4\}$

Berechnung von $[\![\psi]\!]_{\mathcal{K}}$: $\mathbf{EU}$

Fall 5: $\psi \equiv p\ \mathbf{EU}\ q, \quad p, q \in AP$

Analog zu $\mathbf{EG}$ (hier ohne Beweis):

Lemma 4: $[\![p\ \mathbf{EU}\ q]\!]_{\mathcal{K}}$ ist die kleinste Lösung (bzgl. $\subseteq$) von

$$X = \mu(q) \cup (\mu(p) \cap pre(X)).$$

Lemma 5: $[\![p\ \mathbf{EU}\ q]\!]_{\mathcal{K}}$ ist der Fixpunkt der Sequenz

$$\emptyset, \ \xi(\emptyset), \ \xi(\xi(\emptyset)), \ \ldots \text{where} \quad \xi(X) := \mu(q) \cup (\mu(p) \cap pre(X))$$

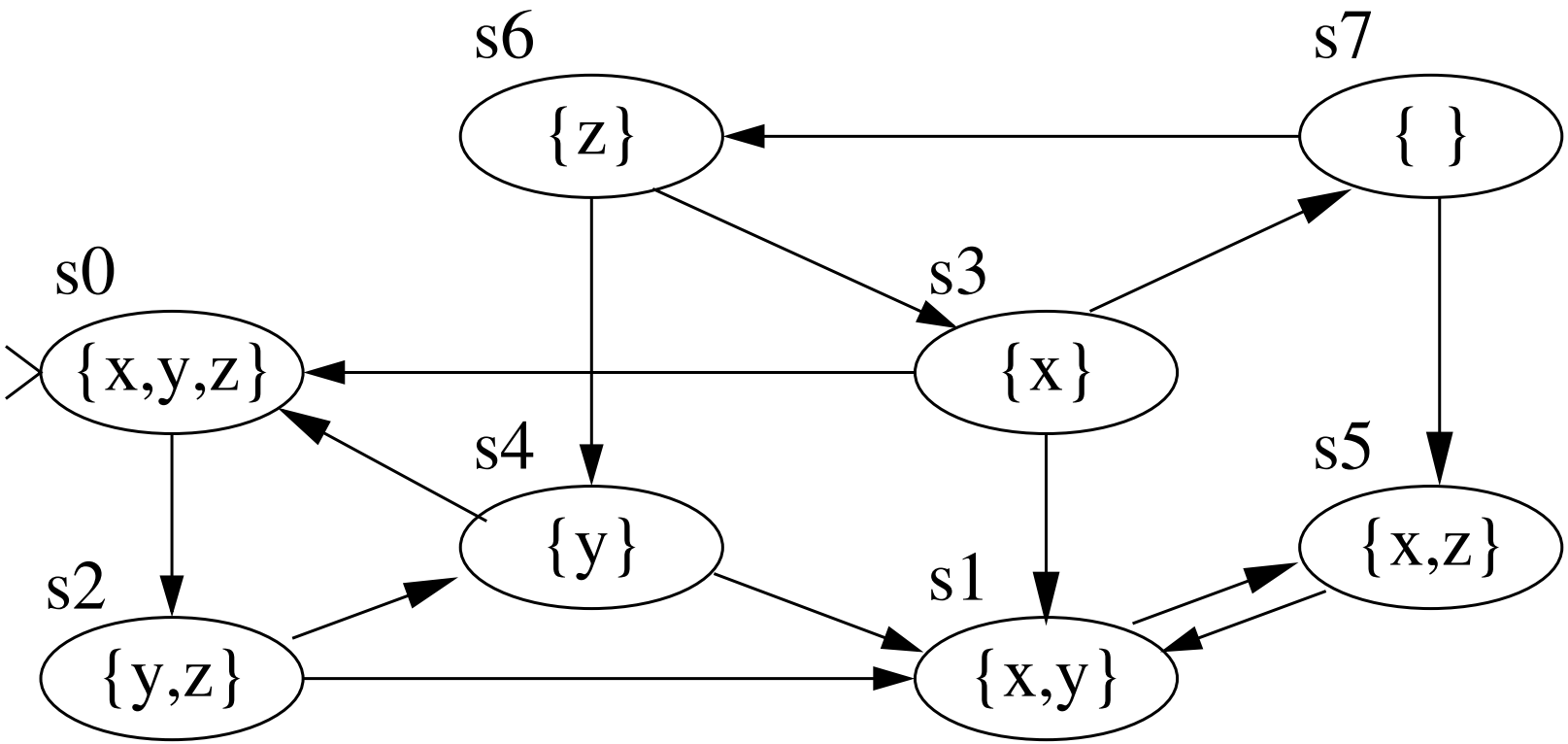
Berechnung der $\mathbf{EU}$-Semantik

Lemma 5 schlägt einen Algorithmus vor: Berechne die Sequenz $\emptyset,\ \xi(\emptyset),\ \cdots$ bis zum Erreichen eines Fixpunkts.

Praktischerweise beginnt man gleich mit $X := \mu(q)$ und fügt nach und nach die direkten Vorgänger von X-Zuständen hinzu, die in $\mu(p)$ liegen.

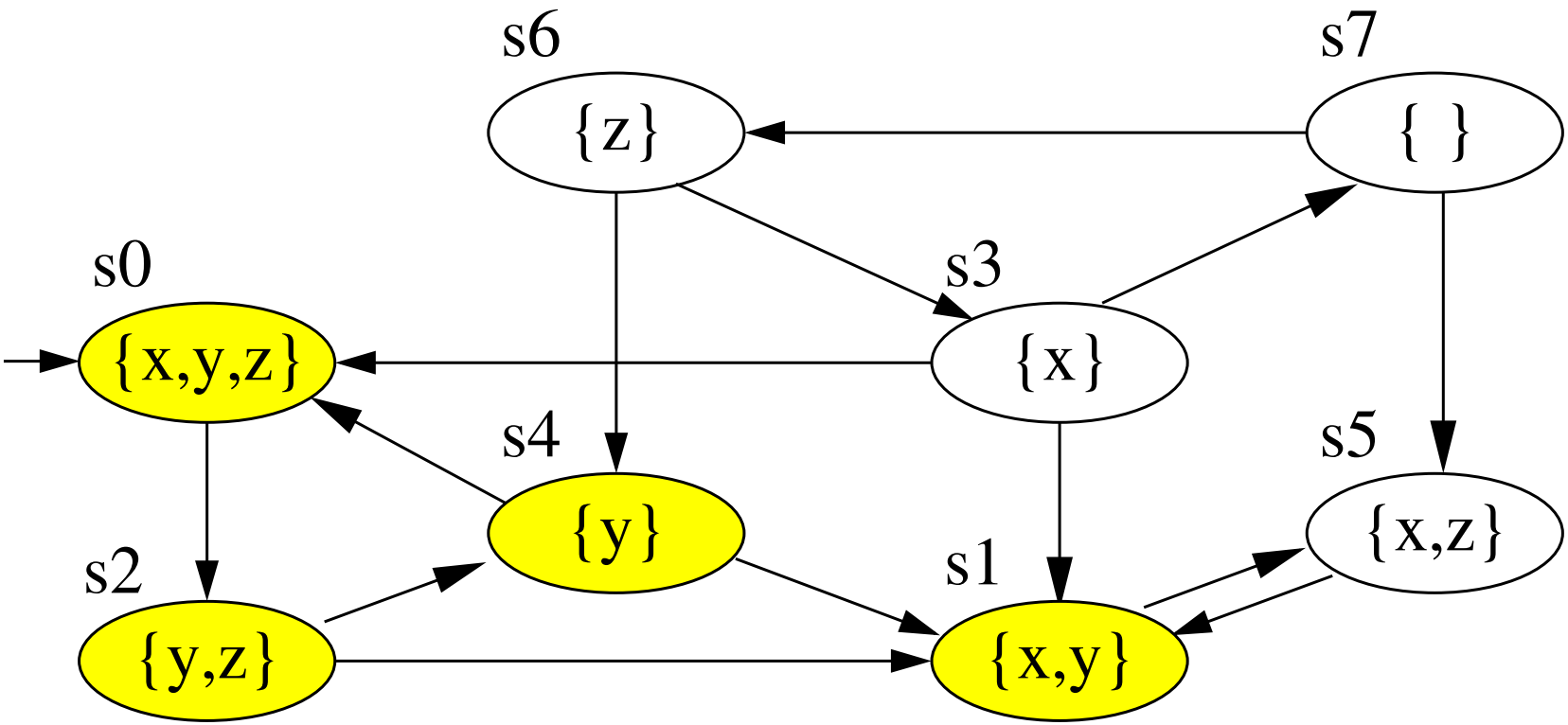
Dies kann effizient in $\mathcal{O}(|\mathcal{K}|)$ Zeit geschehen (mehrfache Rückwärts-Tiefensuche).

Beispiel: Berechnung von $[\![z \ \mathbf{EU} \ y]\!]_{\mathcal{K}}$ (1/4)



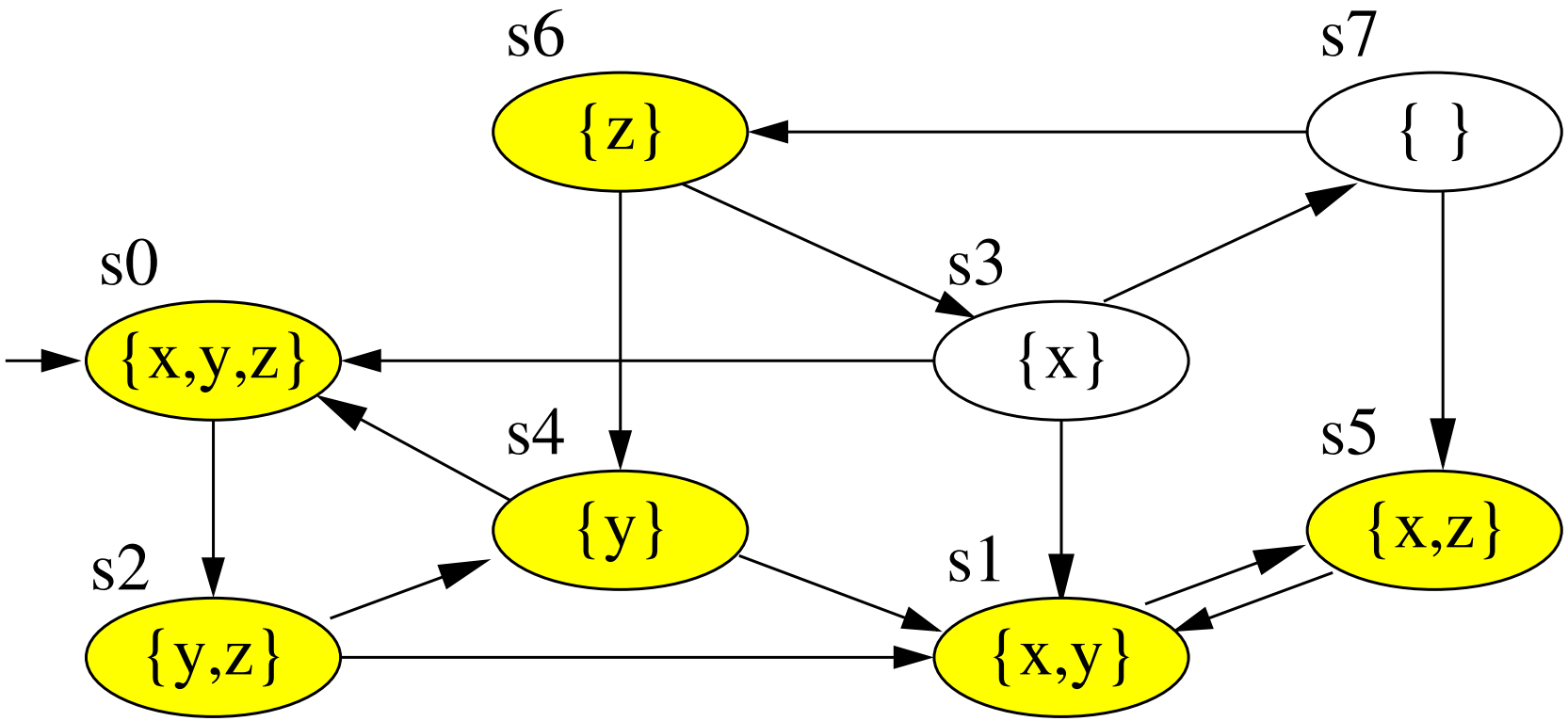
$\xi^0(\emptyset) \ = \ \emptyset$

Beispiel: Berechnung von $[\![z \ \mathbf{EU} \ y]\!]_{\mathcal{K}}$ (2/4)



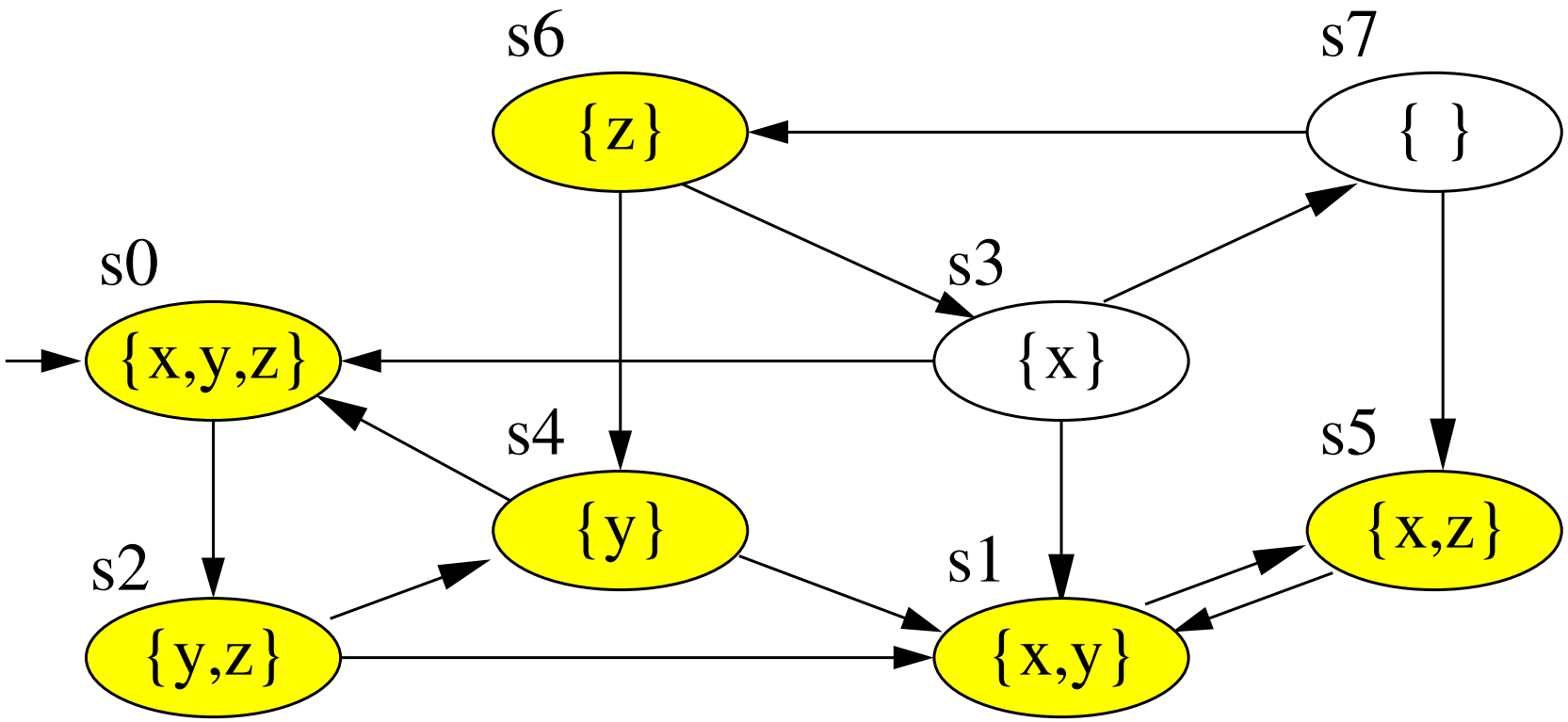
$\xi^1(\emptyset) \ = \ \mu(y) \cup (\mu(z) \cap pre(\xi^0(\emptyset)))$

Beispiel: Berechnung von $[\![z \ \mathbf{EU} \ y]\!]_{\mathcal{K}}$ (3/4)



$\xi^2(\emptyset) \ = \ \mu(y) \cup (\mu(z) \cap pre(\xi^1(\emptyset)))$

Beispiel: Berechnung von $[\![z \ \mathbf{EU} \ y]\!]_{\mathcal{K}}$ (4/4)



$\xi^3(\emptyset) \ = \ \mu(y) \cup (\mu(z) \cap pre(\xi^2(\emptyset))) \ = \ \xi^2(\emptyset)$

$[\![z \ \mathbf{EU} \ y]\!]_{\mathcal{K}} = \{s_0, s_1, s_2, s_4, s_5, s_6\}$

Beispiel: Dekkers Mutex-Algorithmus

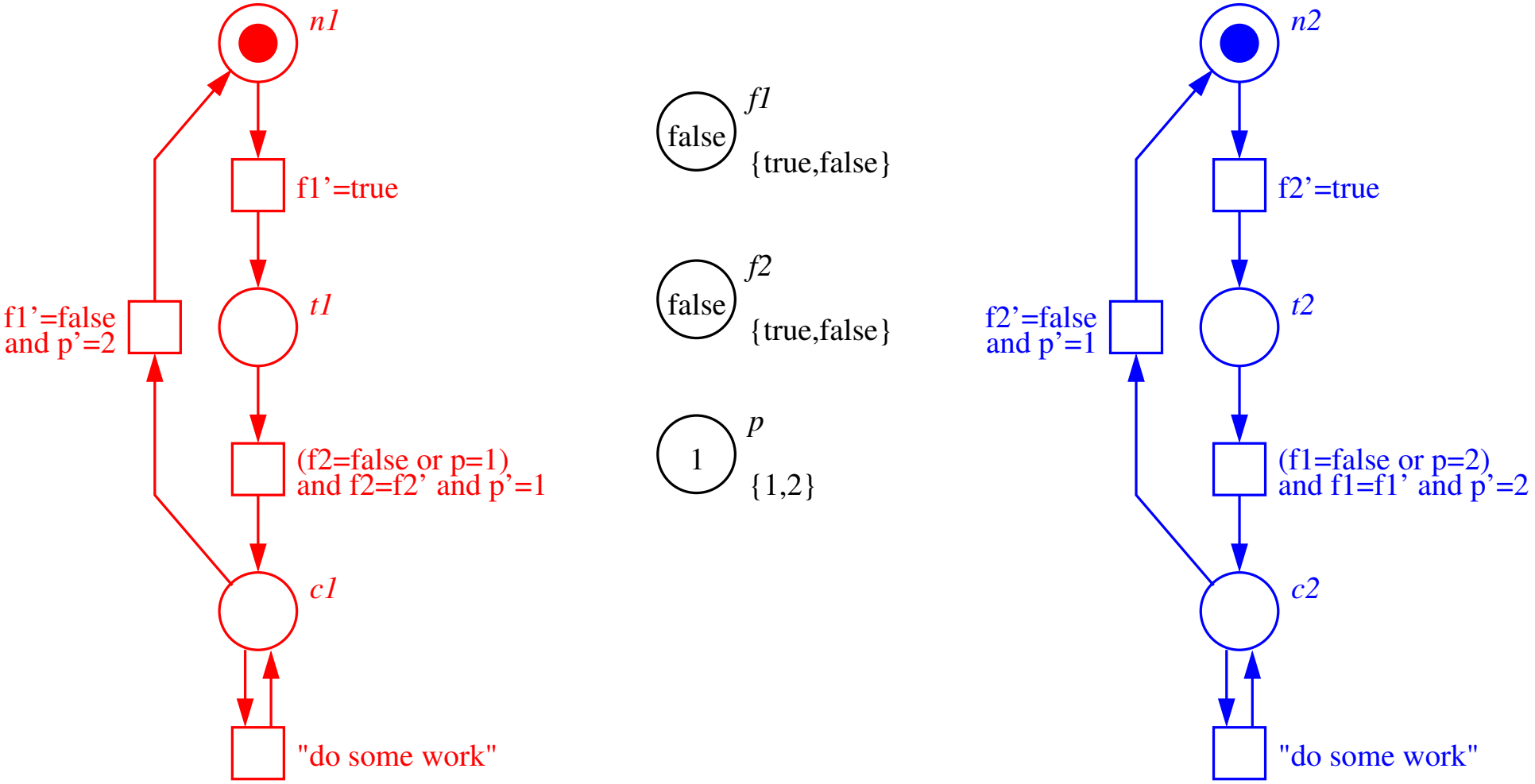
Die nächsten Folien zeigen ein Petri-Netz, welches ein Protokoll zum (fairen) gegenseitigen Ausschluss zweier Prozesse (rot und blau) modelliert, und seinen Erreichbarkeitsgraphen.

Die Stellen $n1, n2$ bezeichnen die nicht-kritischen Abschnitte, $c1, c2$ die kritischen Abschnitte, und $t1, t2$ zeigen an, dass ein Prozess in seinen kritischen Abschnitt eintreten will.

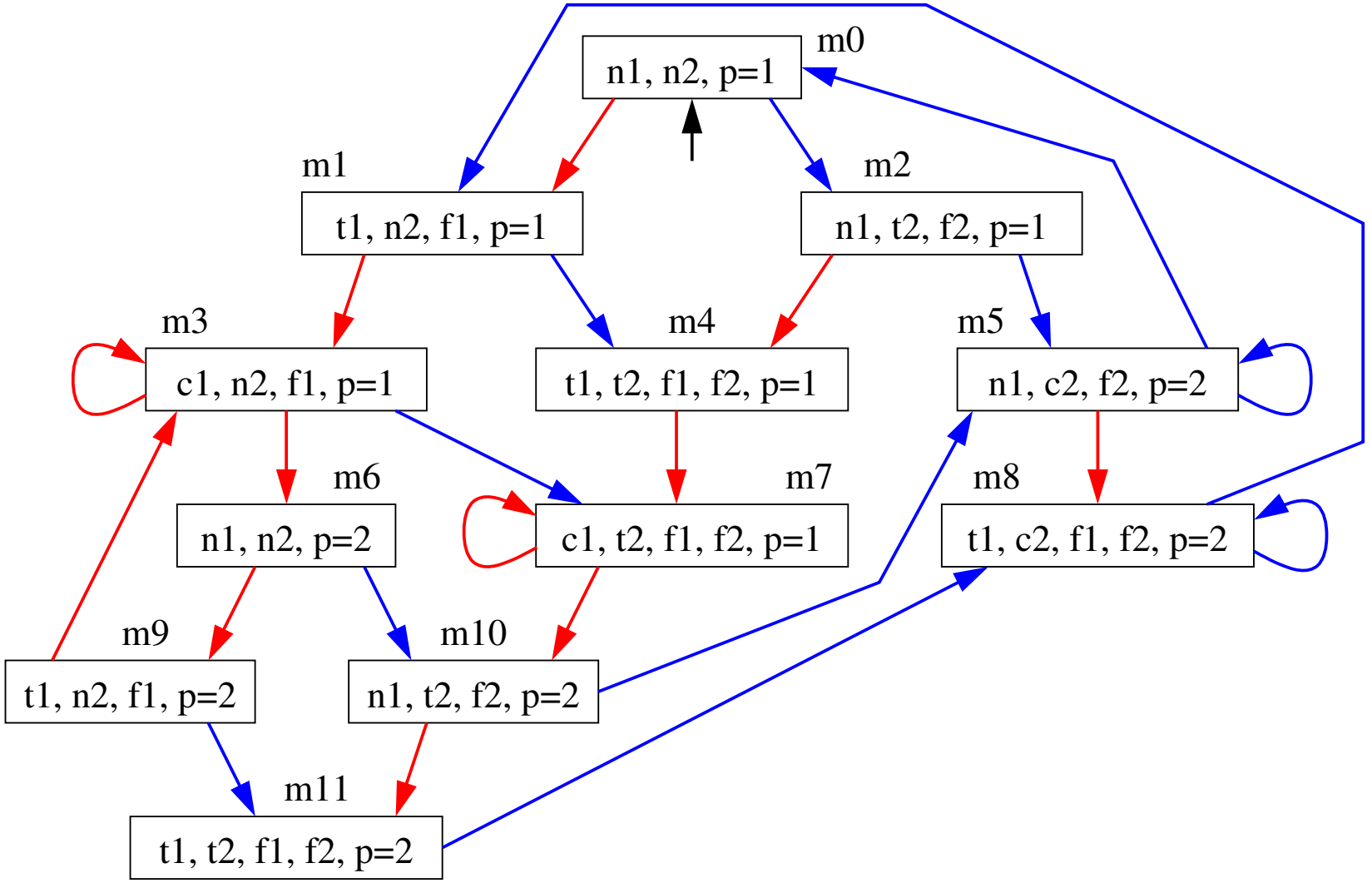
Der Übersichtlichkeit halber wurden die Kanten zu den Variablen in der Mitte $f1, f2, p$ weggelassen.

Im Erreichbarkeitsgraphen sind $f1$ and $f2$ nur dann erwähnt, wenn sie wahr sind.

Beispiel: Petri-Netz



Beispiel: Erreichbarkeitsgraph



Spezifikation

Im Erreichbarkeitsgraphen kann man sehen, dass der gegenseitige Ausschluss gewährleistet ist. Wir könnten uns zusätzlich für folgende Eigenschaft interessieren:

> "Immer, wenn ein Prozess in seinen kritischen Abschnitt eintreten will, wird ihm das irgendwann gelingen."

Formulierung in CTL:

$$\mathbf{AG}(t1 \rightarrow \mathbf{AF}\, c1) \qquad \text{and} \qquad \mathbf{AG}(t2 \rightarrow \mathbf{AF}\, c2)$$

Wir erweitern dazu den Erreichbarkeitsgraphen zur einer Kripke-Struktur mit vier Grundaussagen:

$c1$ mit $\mu(c1) := \{m_3, m_7\}$; $t1$ mit $\mu(t1) := \{m_1, m_4, m_8, m_9, m_{11}\}$

$c2$ mit $\mu(c2) := \{m_5, m_8\}$; $t2$ mit $\mu(t2) := \{m_2, m_4, m_7, m_{10}, m_{11}\}$

Überprüfung

Wir schreiben die erste Formel zunächst in die minimale Syntax um:

$$\neg\Big(\mathbf{true}\,\mathbf{EU}\,(t1 \wedge \mathbf{EG}\,\neg c1)\Big)$$

(Die zweite Formel ist analog, wir betrachten nur die erste.)

Wenn wir diese Formel überprüfen, stellen wir fest, dass die Formel nicht gilt – das System kann in den Zuständen m_5 und m_8 verbleiben.

Dies passiert, wenn ein Prozess für immer im kritischen Abschnitt verbleibt.

Fairness

Die Eigenschaft wird verletzt, da sich der blaue Prozess “unfair” verhalten kann und nicht mehr aus seinem kritischen Abschnitt austreten könnte.

Können wir – analog zu LTL – nur die Abläufe betrachten, die eine Fairness-Bedingung einhalten (Prozesse verlassen ihre kritischen Abschnitte irgendwann wieder)?

Fairness

Die Eigenschaft wird verletzt, da sich der blaue Prozess “unfair” verhalten kann und nicht mehr aus seinem kritischen Abschnitt austreten könnte.

Können wir – analog zu LTL – nur die Abläufe betrachten, die eine Fairness-Bedingung einhalten (Prozesse verlassen ihre kritischen Abschnitte irgendwann wieder)?

Antwort 1: Nein. In CTL lässt sich dies nicht formulieren (z.B. $(\mathbf{AG}\,\mathbf{AF}\,\mathit{fair}) \rightarrow \phi$ tut nicht das Richtige).

Fairness

Die Eigenschaft wird verletzt, da sich der blaue Prozess “unfair” verhalten kann und nicht mehr aus seinem kritischen Abschnitt austreten könnte.

Können wir – analog zu LTL – nur die Abläufe betrachten, die eine Fairness-Bedingung einhalten (Prozesse verlassen ihre kritischen Abschnitte irgendwann wieder)?

Antwort 1: Nein. In CTL lässt sich dies nicht formulieren (z.B. $(\mathbf{AG}\,\mathbf{AF}\,\mathit{fair}) \rightarrow \phi$ tut nicht das Richtige).

Antwort 2: Ja. Wir können CTL erweitern.

CTL mit Fairness

Seien $\mathcal{K}$ und ϕ wie zuvor. Zusätzlich seinen Fairness-Bedingungen $F_1, \ldots, F_n \subset S$ gegeben.

Im Folgenden nennen wir einen Pfad fair gdw. er jede Fairness-Bedingung unendlich oft besucht.

In unserem Beispiel hätten wir folgende Fairness-Bedingungen:

$$F_1 = S \setminus \mu(c1)$$

$$F_2 = S \setminus \mu(c2)$$

Unser Problem besteht darin, $[\![\phi]\!]_{\mathcal{K}}$ zu berechnen für den Fall, dass $\mathbf{EG}$ und $\mathbf{EU}$ nur über faire Abläufe quantifizieren. Dazu führen wir die Operatoren $\mathbf{EG_f}$ und $\mathbf{EU_f}$ ein mit folgender Bedeutung:

$\mathcal{T} \models \mathbf{EG_f}\,\phi$	gdw. $\mathcal{T}$ hat einen fairen unendl. Pfad $r = v_0 \to v_1 \to \ldots$, so dass für alle $i \geq 0$ gilt: $\lceil v_i \rceil \models \phi$
$\mathcal{T} \models \phi_1\,\mathbf{EU_f}\,\phi_2$	gdw. $\mathcal{T}$ hat einen fairen unendl. Pfad $r = v_0 \to v_1 \to \ldots$, so dass $\exists i : \lceil v_i \rceil \models \phi_2 \wedge \forall k < i : \lceil v_k \rceil \models \phi_1$

Wir machen zunächst folgende Beobachtungen:

(1) ρ ist fair gdw. ρ^i fair ist für alle $i \geq 0$.

(2) ρ ist fair gdw. es einen fairen Suffix ρ^i gibt für irgendein $i \geq 0$.

Daher können wir $\mathbf{EU_f}$ wie folgt umschreiben:

$$\phi_1 \, \mathbf{EU_f} \, \phi_2 \quad \equiv \quad \phi_1 \, \mathbf{EU} \, (\phi_2 \wedge \mathbf{EG_f} \, \mathbf{true})$$

Es genügt also, einen neuen Algorithmus für $\mathbf{EG_f}$ zu entwickeln.

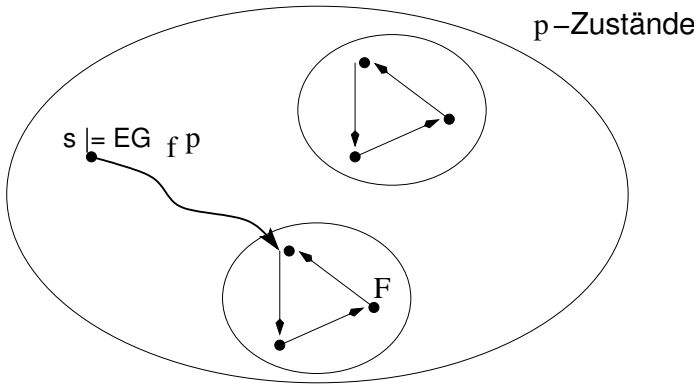
Berechnung von $[\![\mathbf{EG_f}\, p]\!]_{\mathcal{K}}$

1. Sei $\mathcal{K}_p$ die Einschränkung von $\mathcal{K}$ auf $\mu(p)$.

2. Berechne die SCCs von $\mathcal{K}_p$.

3. Finde die nicht-trivialen SCCs, die alle Fairness-Bedingungen schneiden.

4. $[\![\mathbf{EG_f}\, p]\!]_{\mathcal{K}}$ enthält genau die Zustände in $\mathcal{K}_p$, die eine solche SCC erreichen können.



Komplexität: immer noch linear in $|\mathcal{K}|$.

Vergleich von CTL und LTL

Es gibt viele Eigenschaften, die man äquivalent in CTL und in LTL ausdrücken kann, z.B.

Invarianten (e.g., "p gilt nie.")

$$\mathbf{AG}\,\neg p \qquad \text{or} \qquad G\neg p$$

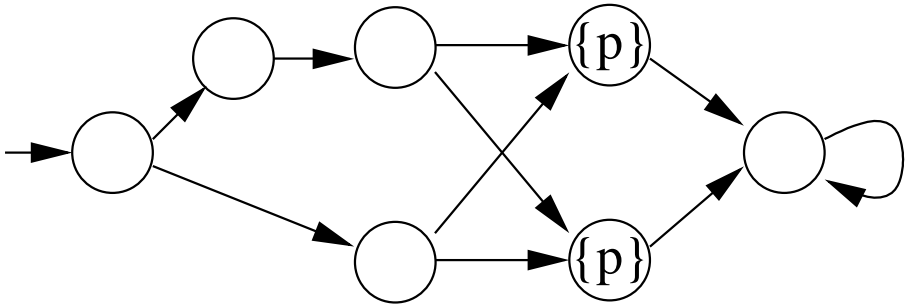
Reaktivität ("Immer, wenn p passiert, passiert irgendwann q.")

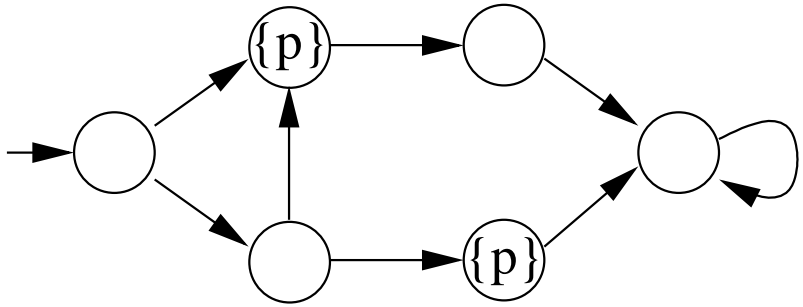
$$\mathbf{AG}(p \rightarrow \mathbf{AF}\,q) \qquad \text{or} \qquad \mathbf{G}(p \rightarrow \mathbf{F}\,q)$$

CTL betrachtet den gesamten Berechnungsbaum, LTL die Gesamtheit der Abläufe. Daher kann CTL über die verschiedenen Möglichkeiten (das *Verzweigungsverhalten*) sprechen, LTL nicht. Beispiele:

Die CTL-Eigenschaft $\mathbf{AG}\,\mathbf{EF}\,p$ (“Reset”) lässt sich in LTL nicht ausdrücken.

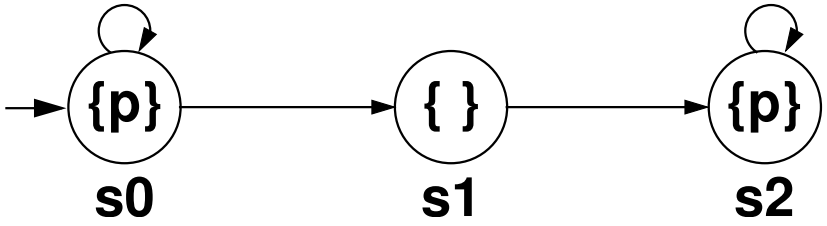
Die CTL-Eigenschaft $\mathbf{AF}\,\mathbf{AX}\,p$ unterscheidet folgende Strukturen, aber die LTL-Eigenschaft $\mathbf{F}\,\mathbf{X}\,p$ tut dies nicht:





Dennoch ist CTL nicht strikt ausdrucksmächtiger als LTL. Dies liegt daran, dass sich Quantoren und Pfad-Operatoren immer abwechseln müssen.

Die LTL-Eigenschaft $\mathbf{F}\,\mathbf{G}\,p$ lässt sich nicht in CTL ausdrücken:



$\mathcal{K} \models \mathbf{F}\,\mathbf{G}\,p$ aber $\mathcal{K} \not\models \mathbf{A}\mathbf{F}\,\mathbf{A}\mathbf{G}\,p$

(Emerson, Halpern 1986)

Komplexität

CTL-Model-Checking: $\mathcal{O}(|\mathcal{K}| \cdot |\phi|)$
LTL-Model-Checking: $\mathcal{O}(|\mathcal{K}| \cdot 2^{|\phi|})$

Ist CTL-Model-Checking also effizienter?

Antwort: Nicht unbedingt:

LTL ermöglicht On-the-fly-Model-Checking, Halbordnungsreduktion

CTL: ganze Struktur muss konstruiert und mehrmals durchlaufen werden

CTL: statt Halbordnungsreduktion $\rightarrow$ effiziente Mengendarstellungen (BDD, kommt noch)

CTL*

Um die Ausdrucksmächtigkeit von CTL und LTL zu vereinen, ist die Logik CTL* vorgeschlagen worden.

Syntax: Boolsche Kombinationen von Grundaussagen und Formeln der Art $\mathbf{A}\,\phi$ oder $\mathbf{E}\,\phi$, wobei ϕ eine LTL-Formel ist.

Kann alle CTL- und LTL-Eigenschaften ausdrücken, hat aber nicht die günstigen algorithmischen Eigenschaften, die CTL und LTL haben, und ist daher nicht populär.

Weitere Hinweise

Mehr zu diesem Thema:

M. Vardi, *Branching vs. Linear Time: Final Showdown*, 2001

Specification Patterns Database:

`http://patterns.projects.cis.ksu.edu/`

Tool-Demonstration: SMV

Das SMV-System

SMV wurde von Ken McMillan geschrieben (CMU, 1992)

Model-Checker für CTL (mit Fairness)

Geeignet zur Beschreibung endlicher Strukturen, insbesondere synchroner oder asynchroner Schaltkreise.

SMV war das erste Tool, das zur Verifikation großer Hardware-Systeme geeignet war (Gebrauch von BDDs).

Informationen/Download von SMV

Im WWW:

`http://www-2.cs.cmu.edu/~modelcheck/smv.html`

(mit ausführlicher Anleitung)

CTL-Model-Checking in SMV

Gegeben $\mathcal{K}$ (mit mehreren Anfangszuständen) und ϕ,
erfüllen alle Anfangszustände von $\mathcal{K}$ die Formel ϕ?

In SMV besteht eine Kripke-Struktur aus Modulen Prozessen,
die jeweils Variablen verwalten.

Transitionen werden spezifiziert, indem jeder Variable ihr ‘next’-Wert zugewiesen wird, in Abhängigkeit vom derzeitigen Wert.

Grundaussagen werden über Variablen getroffen und auf die ‘natürliche’ Weise interpretiert (wie in Spin).

Syntax-Beispiel (1/3)

```
-- Hier steht ein Kommentar.

MODULE main
VAR
   x : boolean;
   y : {q1,q2};

-- wird fortgesetzt...
```

Bemerkungen:

- Variablentypen: Boolean, ganze Zahlen, Aufzählungstypen
- alle Wertebereiche *endlich*

Syntax-Beispiel (2/3)

```
ASSIGN
  init(x) := 1;                          -- Anfangswert
  next(x) := case                        -- Transitionsrelation
               x: 0;
              !x: 1;
             esac;

  next(y) := case
               x & y=q1: q2;
               x & y=q2: {q1,q2};  -- Nicht-Determinismus
               1       : y;
             esac;
```

Syntax-Beispiel: Bemerkungen

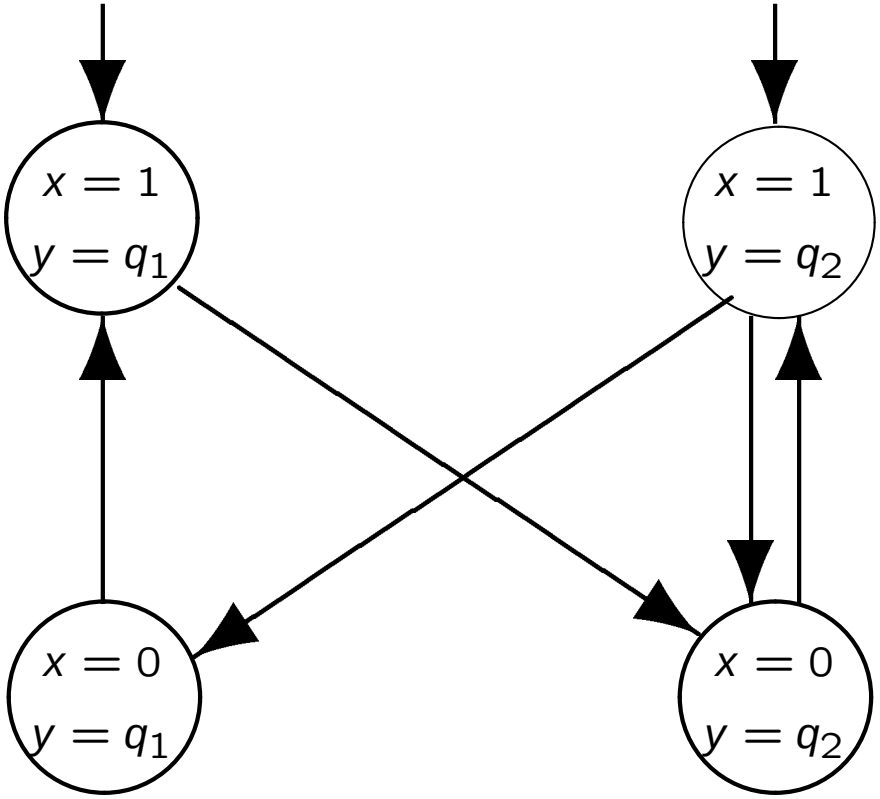
Anfangszustände sind durch die init-Prädikate gegeben; uninitialisierte Variablen können irgendeinen Wert annehmen.

Die Transition des Systems entstehen aus der synchronen Komposition der next-Prädikate.

case-Ausdrücke werden von oben nach unten ausgewertet, der erste passende Fall wird genommen.

Nicht-Determinismus kann man eingeführen, indem man eine *Menge* möglicher Nachfolgewerte angibt.

Die entstehende Struktur



Syntax-Beispiel (3/3)

```
-- Ist q1 immer von q2 erreichbar?
SPEC AG (y=q2 -> EF y=q1)

-- Wird x in jedem Ablauf unendlich oft wahr?
SPEC AG AF x
```

Bemerkungen:

- Man kann mehrere CTL-Formeln angeben, SMV überprüft sie alle nacheinander.

Demonstration: xy.smv

Module (1/2)

Module können parametrisiert sein. Beispiel:

```
MODULE counter_cell(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := 0;
  next(value) := (value + carry_in) mod 2;
DEFINE
  carry_out := value & carry_in;
```

Bemerkung:

- Dieses Modul ist durch `carry_in` parametrisiert.
- `DEFINE` schafft ein ‘Macro’.

Module (2/2)

Module werden instanziiert wie Variablen:

```
MODULE main
VAR
  bit0 : counter_cell(1);
  bit1 : counter_cell(bit0.carry_out);
  bit2 : counter_cell(bit1.carry_out);
```

Dieses System verhält sich wie ein Drei-Bit-Zähler, der von 0 bis 7 zählt und sich wieder zurücksetzt.

Bemerkung: Es muss immer ein Modul namens `main` geben, und die Spezifikationen dieses Moduls werden ausgewertet.

Demonstration: counter.smv

Asynchrone Systeme

Alle Beispiele bis jetzt waren synchron, d.h. alle Variablen und Module machen zur selben Zeit einen Schritt.

Gibt es eine Module, das mit dem Schlüsselwort `process` instanziiert sind (siehe Beispiel), dass wird in jedem Schritt ein Prozess *nicht-deterministisch* ausgewähl,t während die anderen Prozesse nichts tun (*asynchrone* Komposition).

Alternativ können alle Prozesse stillstehen (“Stottern”).

Beispiel: Mutex

var turn : {0,1};

while true do	**while true do**
q_0 non-critical section	r_0 non-critical section
q_1 **await** (turn=0);	r_1 **await** (turn=1);
q_2 critical section	r_2 critical section
q_3 turn:=1;	r_3 turn:=0;
od	**od**

Mutex in SMV (1/2)

```
MODULE main
VAR
  turn: boolean;
  p0: process p(0,turn);
  p1: process p(1,turn);

SPEC
  AG !(p0.state = critical & p1.state = critical)
```

Mutex in SMV (2/2)

```
MODULE p (nr,turn)
VAR
  state: {non_critical, critical};
ASSIGN
  init(state) := non_critical;
  next(state) := case
     state = non_critical & turn != nr: non_critical;
     state = non_critical & turn = nr : critical;
     state = critical: {critical,non_critical};
  esac;
  next(turn) := case
     state = critical & next(state) = non_critical: !nr;
     1                                            : turn;
  esac;
```

Fairness

Im Mutex-Beispiel wird folgende Eigenschaft erfolglos getestet:

```
SPEC
  AG (p0.state = non_critical -> AF p0.state = critical)
```

Spin zeigt einen Fehler an, weil das System für immer stottern kann. Mit dem Schlüsselwort `FAIRNESS` so ein Verhalten verbieten, z.B. wie folgt:

```
FAIRNESS
  p0.running & p1.running
```

Die interne Variable `running` wird jedesmal wahr, wenn der zugehörige Prozess einen Schritt ausgeführt hat. Mit dieser zusätzlichen Bedingung gelingt die Verifikation.

Teil 12: Binary Decision Diagrams

Literatur

Zusätzliche Literaturhinweise:

H.R. Andersen, *An Introduction to Binary Decision Diagrams*, Lecture notes, Department of Information Technology, IT University of Copenhagen

Im WWW: http://www.itu.dk/people/hra/bdd97-abstract.html

Tools:

DDcal (“BDD-Taschenrechner”)

Im WWW: http://vlsi.colorado.edu/~fabio/

Darstellungen von Mengen

Wie wir gesehen haben, lässt sich das MC-Problem für CTL durch Operationen auf Mengen ausdrücken:

- Zustände, die Grundaussagen erfüllen: $\mu(p)$ für $p \in AP$
- Zustände, die Teilformeln erfüllen: $[\![\psi]\!]_{\mathcal{K}}$
- Berechnung durch Mengen-Operationen: pre, $\cap$, $\cup$, ...

Wie stellen wir Mengen dar?

- explizite Auflistung: $S = \{s_1, s_2, s_4, \ldots\}$
- symbolische Darstellung: (irgendeine) kompakte Mengenschreibweise

Symbolische Darstellung

Es gibt viele Möglichkeiten, Mengen symbolisch darzustellen.
Einige davon sind Standard in der Mathematik:

Intervalle: $[1, 10]$ statt $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

Charakterisierungen: “ungerade Zahlen” statt $\{1, 3, 5, \ldots\}$

Jede symbolische Darstellung ist geeignet für manche Mengen und ungeeignet für andere (z.B. Intervall-Schreibweise für ungerade Zahlen).

Ob eine gegebene Darstellung geeignet ist, hängt davon ab, welche Eigenschaften man von den darzustellenden Mengen erwartet.

Wir werden eine Darstellung betrachten, die für den Fall geeignet ist, dass Zustände sich als boolsche Vektoren darstellen lassen d.h. wir nehmen an, dass

$$S = \{0,1\}^m \qquad \text{für ein } m \geq 1$$

Beispiele:

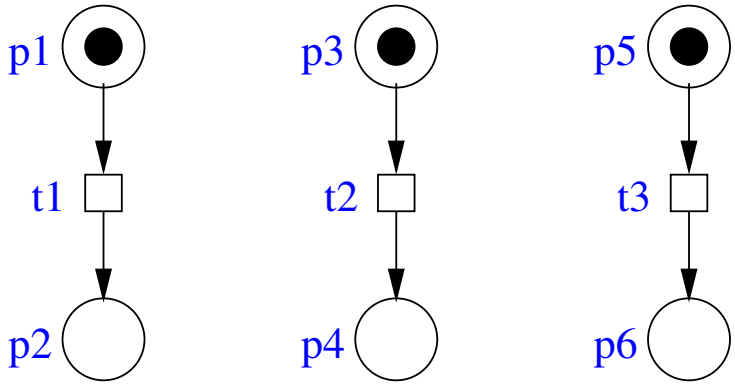
1-sichere Petri-Netze (jede Stelle mit höchstens einem Token markiert)

Schaltkreise (jede Ein- oder Ausgabe 0 oder 1)

Bemerkung: Im Allgemeinen können wir die Elemente jeder beliebigen *endlichen* Menge als boolsche Vektoren darstellen, indem wir m groß genug wählen. (Ob das immer adäquat ist, sei dahingestellt.)

Beispiel 1: Petri-Netz

Betrachten wir folgendes Petri-Netz:

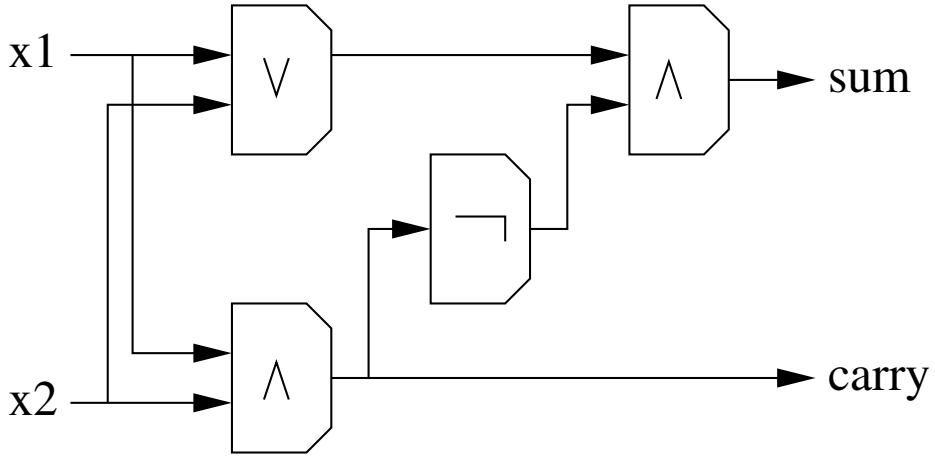


Ein Zustand kann als $(p_1, p_2, \ldots, p_6)$ geschrieben werden, wobei p_i, $1 \leq i \leq 6$ anzeigt, ob sich auf P_i ein Token befindet.

Anfangszustand$(1, 0, 1, 0, 1, 0)$;
andere erreichbare Zustände sind z.B. $(0, 1, 1, 0, 1, 0)$ oder $(1, 0, 0, 1, 0, 1)$.

Beispiel 2: Schaltkreis

Halb-Addierer:



Der Schaltkreis hat zwei Eingaben (x_1, x_2) und zwei Ausgaben ($carry, sum$). Die Kombination von Ein-/Ausgaben kann man als boolsches 4-Tupel schreiben, z.B. ist $(1, 0, 0, 1)$ (d.h. $x_1 = 1$, $x_2 = 0$, $carry = 0$, $sum = 1$) eine Kombination, die auftreten kann.

Charakteristische Funktionen

Sei $C \subseteq S = \{0,1\}^m$.
(Wir wollen schließlich *Mengen* von Vektoren darstellen.)

Die Menge C ist eindeutig durch ihre characteristische Funktion $f_C : S \to \{0,1\}$ bestimmt, die gegeben ist durch

$$f_C(s) := \begin{cases} 1 & \text{falls } s \in C \\ 0 & \text{falls } s \notin C \end{cases}$$

Bemerkung: f_C ist eine boolsche Funktion mit m Eingaben und daher "äquivalent" zu einer aussagenlogischen Formel F mit m Grundaussagen.

Die möglichen Ein-/Ausgabe-Kombinationen in Beispiel 2 haben eine charakterische Funktion, die folgender Formel der AL entspricht:

$$F \equiv \Big(carry \leftrightarrow (x_1 \wedge x_2)\Big) \wedge \Big(sum \leftrightarrow (x_1 \vee x_2) \wedge \neg carry\Big)$$

Im Folgenden werden wir

- Zustandsmengen (d.h. Mengen boolscher Vektoren)
- characteristischen Funktionen
- Formeln der AL

als unterschiedliche Darstellungen derselben Objekte behandeln.

Darstellungen boolscher Funktionen

Wahrheitstafel:

x_1	x_2	*carry*	*sum*	F
0	0	0	0	1
0	0	0	1	0
		...		
0	1	0	1	1
		...		

Die Wahrheitstafel ist *keine* kompakte Darstellung.

Im Folgenden entwickeln wir eine kompakte graphische Darstellung

Binäre Entscheidungsgraphen

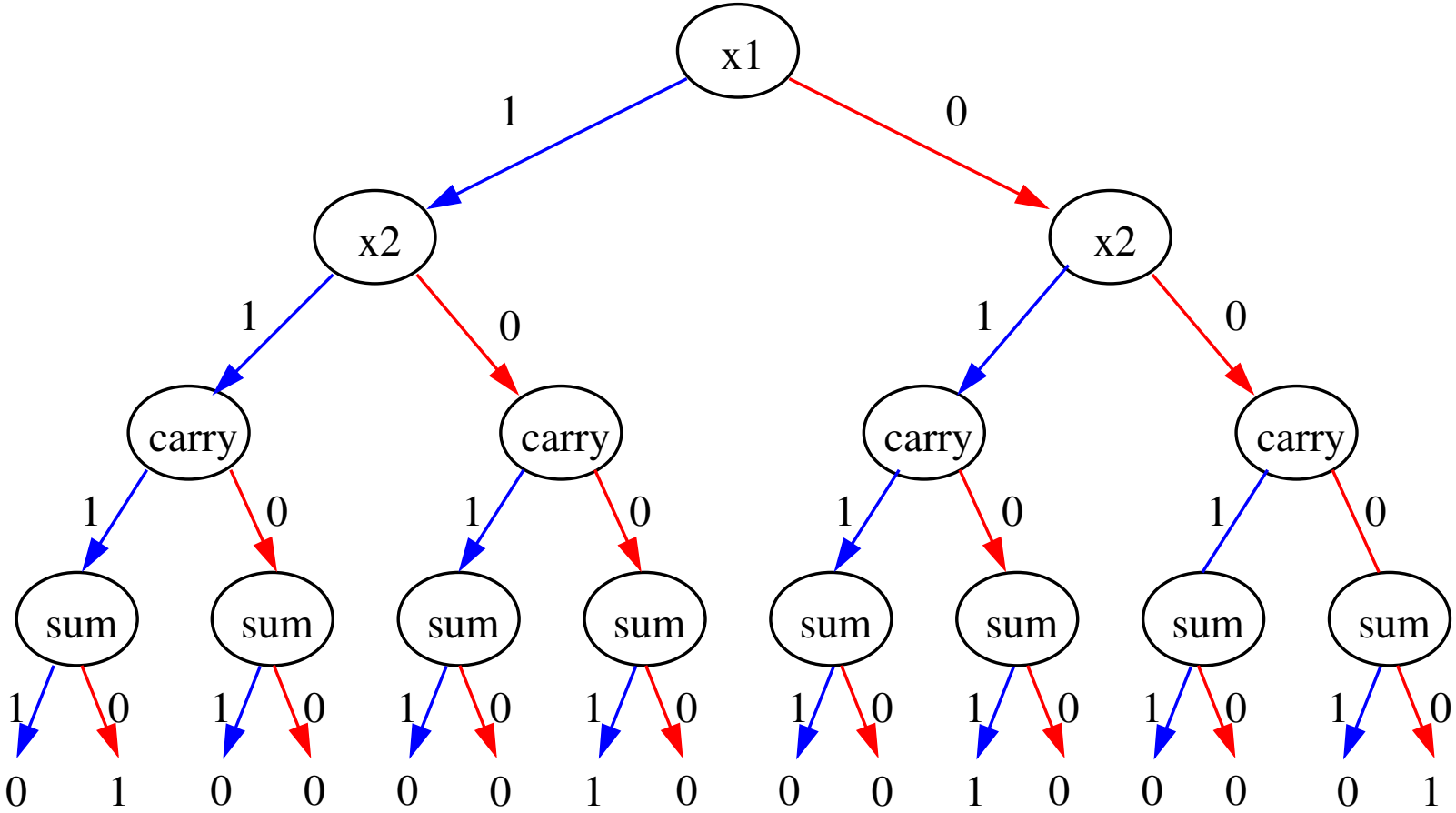
Sei V eine Menge von Variablen und $<$ eine totale Ordnung auf V, z.B.

$$x_1 < x_2 < \mathit{carry} < \mathit{sum}$$

Ein binärer Entscheidungsgraph (bzgl. $<$) ist ein zusammenhängender azyklischer gerichteter Graph mit folgenden Eigenschaften:

- es gibt genau eine Wurzel, d.h. einen Knoten ohne eingehende Kanten;
- es gibt höchstens zwei Blätter, beschriftet mit 0 oder 1;
- alle Nicht-Blätter sind mit Variablen aus V beschriftet;
- jedes Nicht-Blatt hat zwei ausgehende Kanten, die mit 0 und 1 beschriftet sind;
- gibt es eine Kante von einem x-Knoten zu einem y-Knoten, dann $x < y$.

Beispiel 2: Binärer Entscheidungsgraph (als Baum)



Pfade, die in 1 enden, entsprechen Vektoren, die in der Wahrheitstafel eine 1 haben.

Binäre Entscheidungsdiagramme

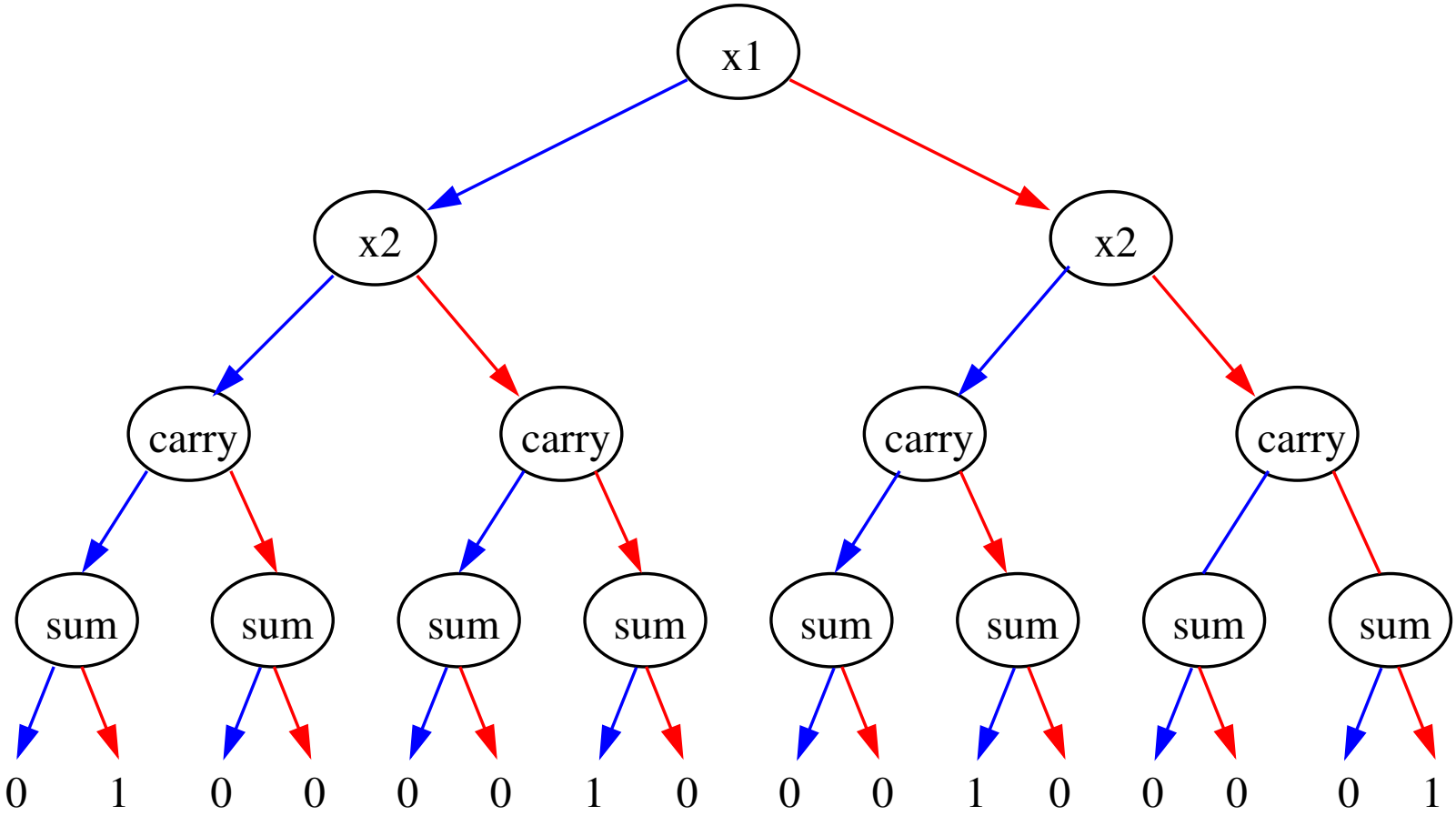
Ein binäres Entscheidungsdiagramm (binary decision diagram, BDD) ist ein binärer Entscheidungsgraph mit zwei zusätzlichen Eigenschaften:

- keine zwei Untergraphen sind isomorph;
- es gibt keine *redundanten* Knoten, bei denen beide ausgehenden Kanten zum selben Ziel führen

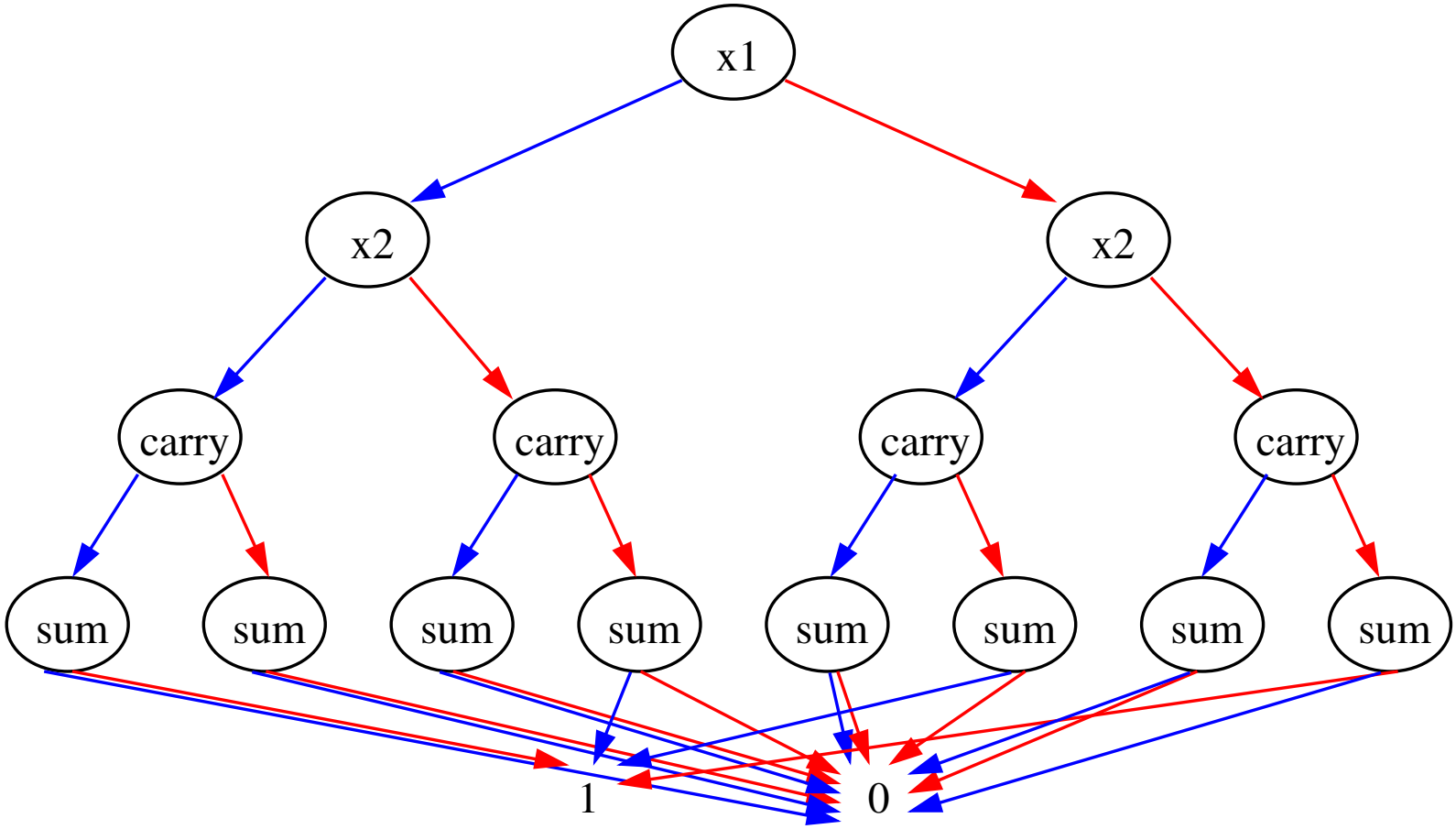
Wir erlauben auch, den 0-Knoten wegzulassen, sowie die Kanten, die dorthin führen.

Bemerkungen: Auf den folgenden Folien gelten die blauen Kanten als mit 1 beschriftet, die roten als mit 0 beschriftet.

Beispiel 2: Isomorphe Untergraphen eliminieren (1/4)

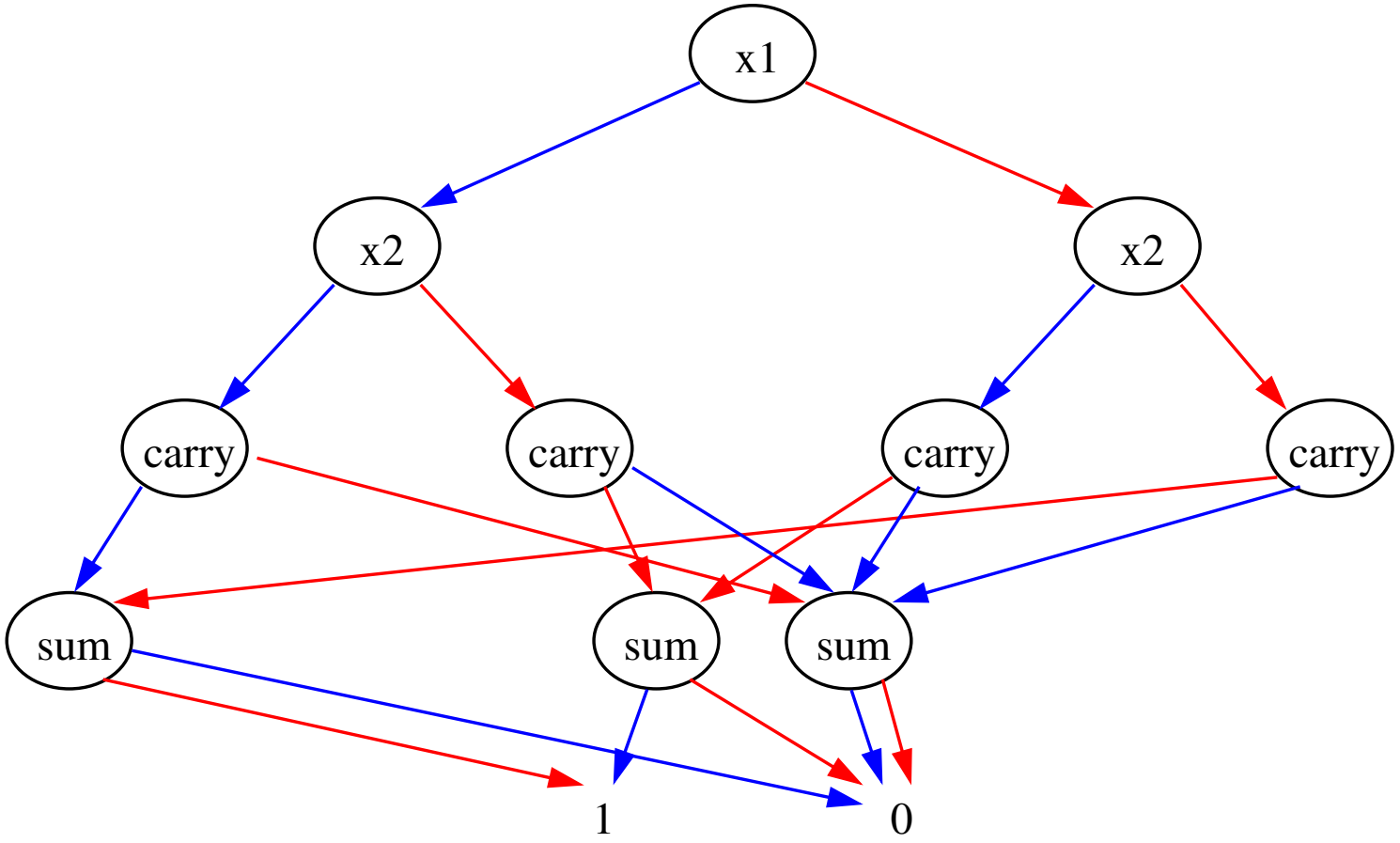


Beispiel 2: Isomorphe Untergraphen eliminieren (2/4)



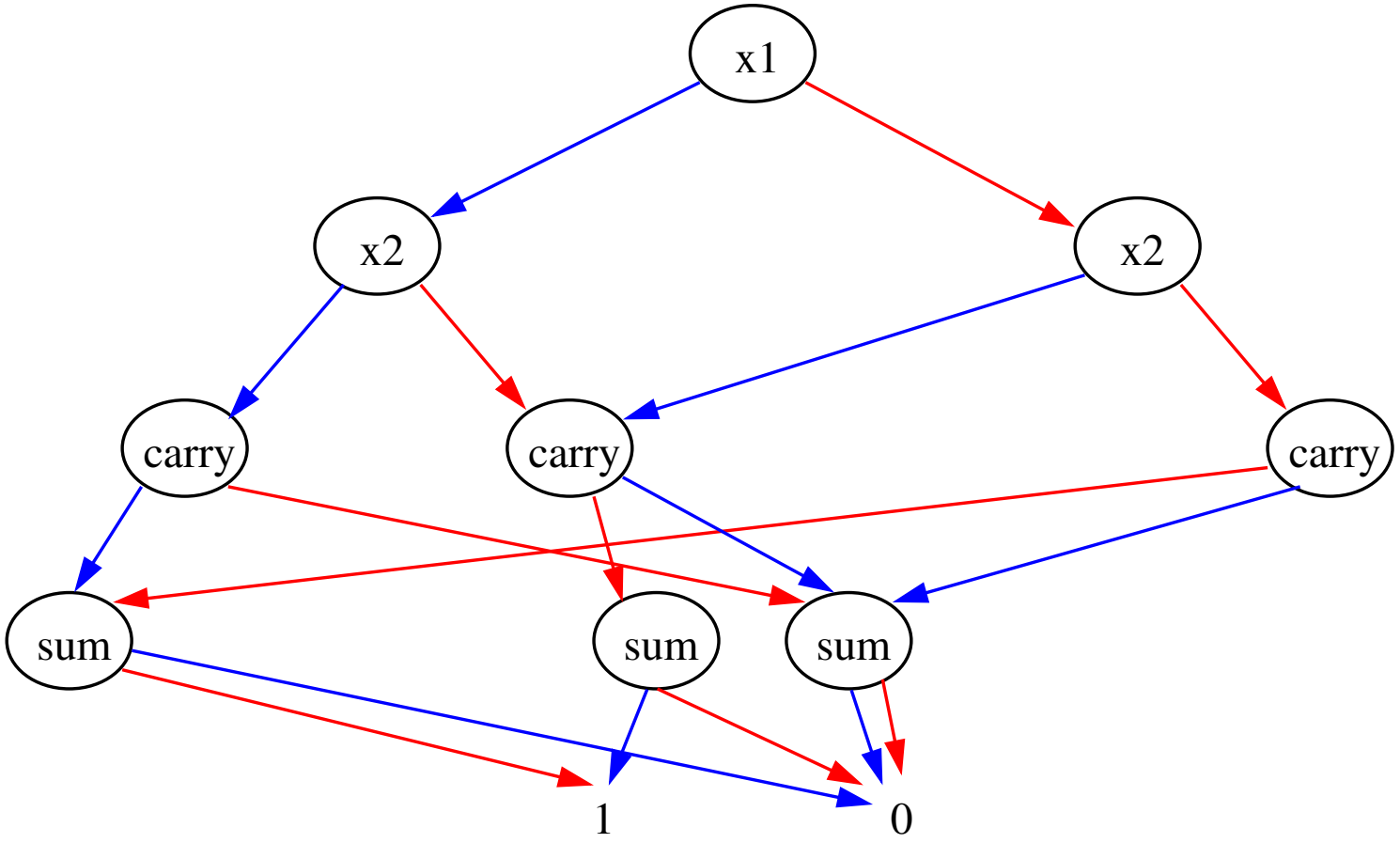
Alle 0- und 1-Knoten werden zusammengefasst.

Beispiel 2: Isomorphe Untergraphen eliminieren (3/4)



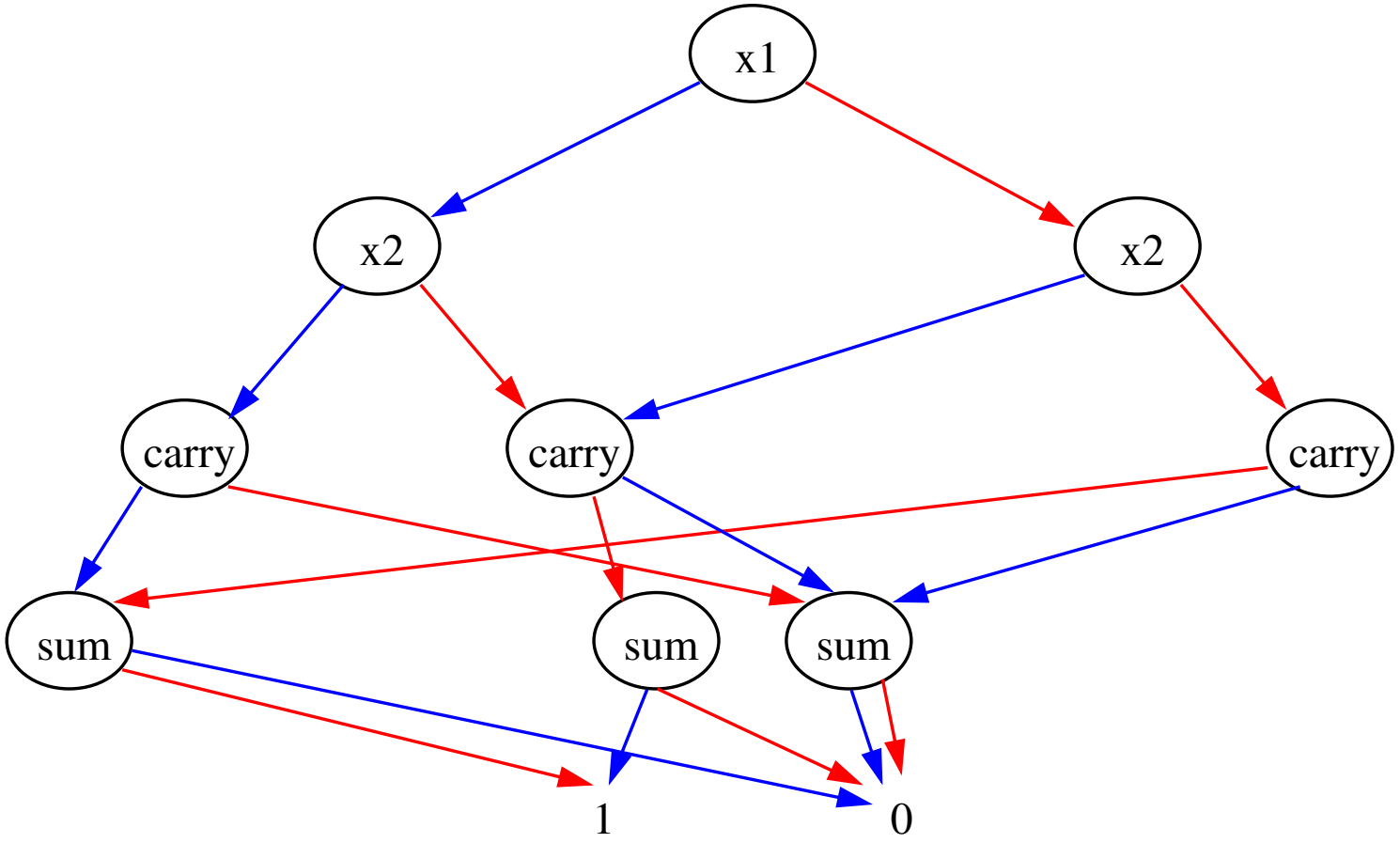
Isomorphe sum-Knoten zusammengefasst.

Beispiel 2: Isomorphe Untergraphen eliminieren (4/4)



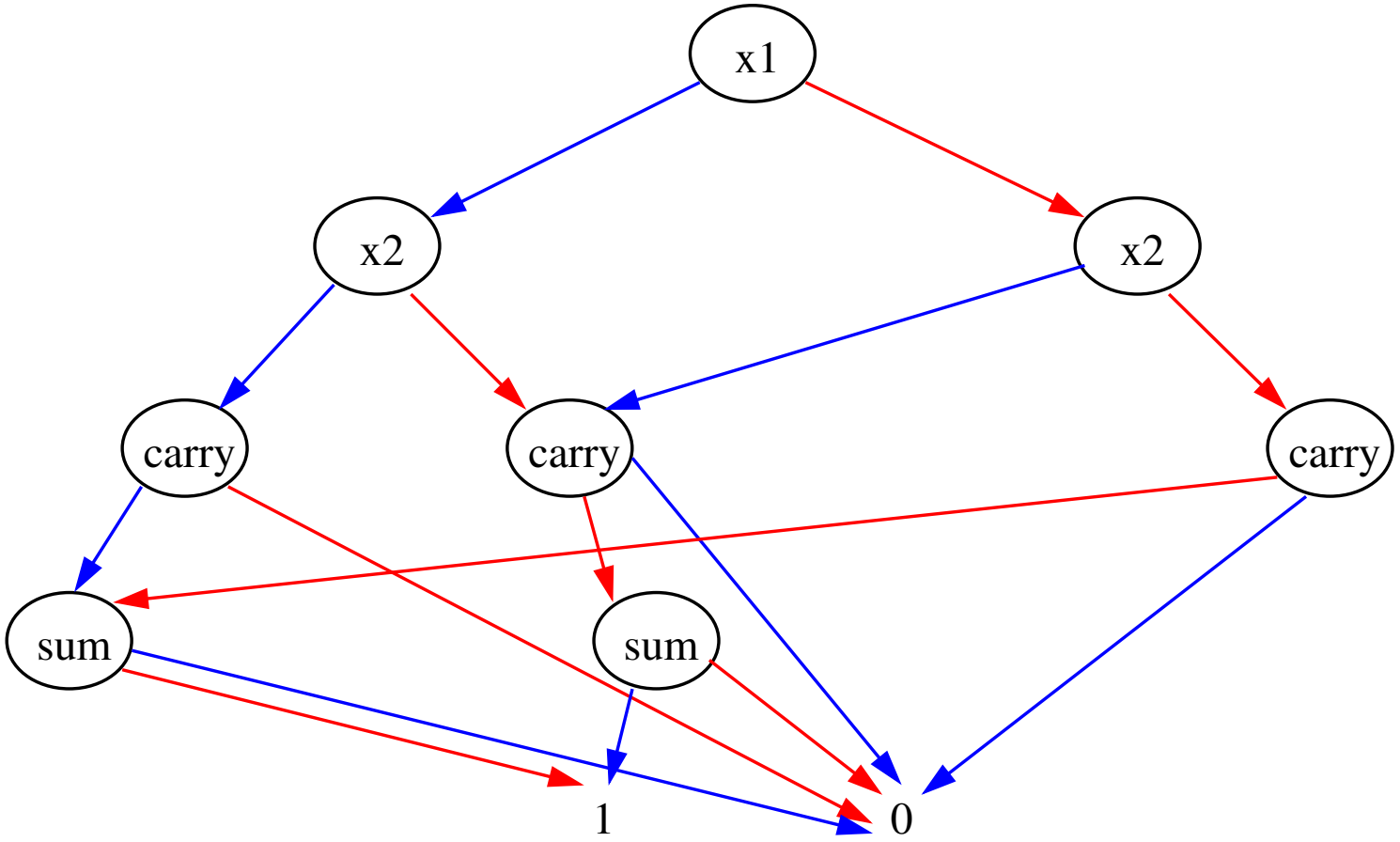
Keine isomorphen Untergraphen sind übrig → fertig

Beispiel 2: Redundante Knoten entfernen (1/2)



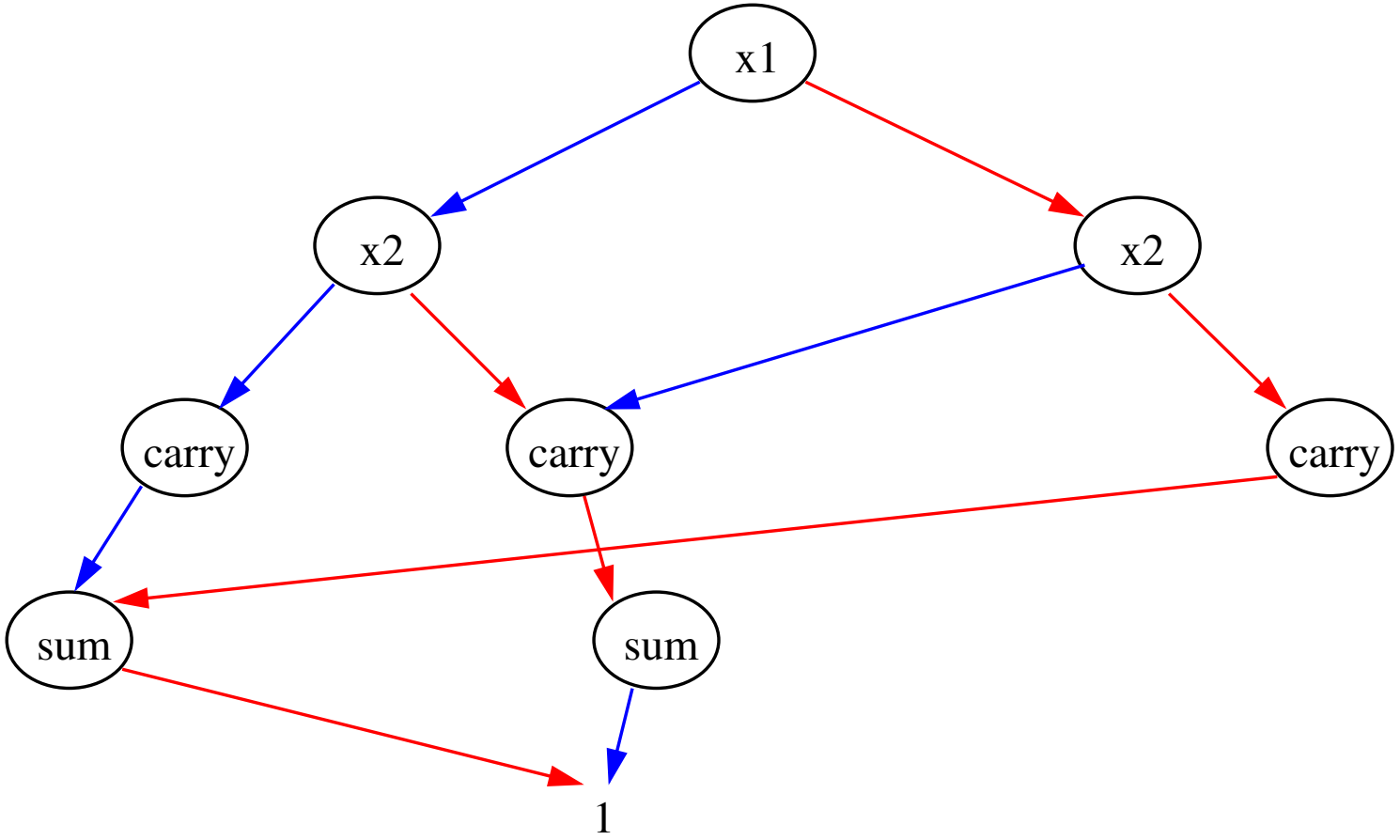
Beide Kanten des rechten *sum*-Knotens zeigen auf 0.

Beispiel 2: Redundante Knoten entfernen (2/2)



Alle redundanten Knoten entfernt → fertig.

Beispiel 2: 0-Knoten weglassen



Optional kann der 0-Knoten und die zu ihm führenden Kanten entfernt werden, der entstehende Graph ist i.d.R. übersichtlicher.

Semantik eines BDDs

Sei B ein BDD mit Ordnung $x_1 < \ldots < x_n$. Sei P_B die Menge aller Pfade in B, die von der Wurzel zum 1-Knoten führen.

Sei $p \in P_B$ so ein Pfad, z.B. $x_{i_1} \xrightarrow{b_{i_1}} \ldots \xrightarrow{b_{i_m}} \mathbf{1}$. Dann bestimmen wir als "Bedeutung" von p (geschrieben $[\![p]\!]$) die Konjunktion aller x_{i_j} mit $b_{i_j} = 1$ und aller $\neg x_{i_j}$ mit $b_{i_j} = 0$ (wobei $j = 1, \ldots, m$).

Wir sagen dann, dass B folgende Formel der AL darstellt:

$$F = \bigvee_{p \in P_B} [\![p]\!]$$

Vorschau

Wir werden im Folgenden einige Operationen auf BDDs betrachten, die man zur Implementierung von CTL-Model-Checking benötigt:

- Konstruktion eines BDDs (aus einer Formel der AL)
- Äquivalenztest
- Schnitt, Komplement, Vereinigung
- Relationen, Berechnung der Vorgänger

Aussagenlogik mit Konstanten

Im Folgenden werden wir Formeln der Aussagenlogik (AL) betrachten, in denen auch die Konstanten 0 und 1 vorkommen dürfen.

0 ist eine unerfüllbare Aussage.

1 ist eine Tautologie.

Ersetzung

Seien F und G irgendwelche Formeln der AL (möglicherweise mit Konstanten), und sei x eine Grundaussage.

Mit $F[x/G]$ bezeichnen wir die Formel der AL, die entsteht, indem jedes Vorkommen von x in F durch G ersetzt wird.

Insbesondere werden wir Formeln der Art $F[x/0]$ und $F[x/1]$ betrachten.

Beispiel: Sei $F = x \wedge y$. Dann ist $F[x/1] = 1 \wedge y \equiv y$ und $F[x/0] = 0 \wedge y \equiv 0$.

If-then-else

Wir erweitern die AL um einen neuen, ternären Operator namens *ite* (if-then-else).

Seien F, G, H Formeln der AL. Dann definieren wir

$$ite(F, G, H) := (F \wedge G) \vee (\neg F \wedge H)$$

Die Menge der Formeln, die in If-then-else-Normalform (INF) stehen, ist wie folgt:

$\mathbf{0}$ und $\mathbf{1}$ sind in INF;

wenn x eine Grundaussage ist und G, H in INF sind, dann ist auch $ite(x, G, H)$ in INF.

Shannon-Zerlegung

Sei F eine Formel der AL und x eine Grundaussage. Es gilt:

$$F \equiv ite(x, F[x/1], F[x/0])$$

Beweis: Im Folgenden wird die rechte Seite der obigen Äquivalenz mit G bezeichnet. Sei $\mathcal{B}$ eine Belegung mit $\mathcal{B} \models F$. Entweder ist $\mathcal{B}(x) = 1$, dann ist $\mathcal{B}$ auch Modell von $F[x/1]$ und von x und damit von G, und analog für $\mathcal{B}(x) = 0$. Gelte umgekehrt $\mathcal{B} \models G$. Dann ist entweder $\mathcal{B}(x) = 1$ und der "Rest" von $\mathcal{B}$ ein Modell von $F[x/1]$. Dann wird $\mathcal{B}$ immer noch ein Modell für jede Formel sein, in der einige der Einsen in $F[x/1]$ durch x ersetzt werden, insbesondere auch für F. Analog für $\mathcal{B}(x) = 0$.

Bemerkung: G wird als Shannon-Zerlegung von F bezeichnet.

Es gilt: Jede Formel der AL ist äquivalent zu einer Formel in INF.
(Beweis: mehrfache Anwendung obiger Äquivalenz)

BDD-Konstruktion

Wir können jetzt unser erstes Problem lösen: Gegeben eine Formel F der AL, konstruiere ein BDD, das F darstellt.

Wenn in F keine Grundaussagen vorkommen, dann ist $F = 0$ oder $F = 1$, und der zugehörige BDD besteht nur aus dem entsprechenden Blatt.

Ansonsten sei $<$ irgendeine gegebene Ordnung der Grundaussagen, die in F vorkommen, und x sei die kleinste dieser Variablen bzgl. $<$. Konstruiere BDDs B_0 und B_1 für $F[x/1]$ und $F[x/0]$ (diese Formeln haben eine Variable weniger).

Wegen der Shannon-Zerlegung wird F durch einen Entscheidungsgraphen repräsentiert, dessen Wurzel x mit Unterbäumen B_0 und B_1 ist. Um ein BDD zu erhalten, prüfen wir noch, ob B_0 und B_1 isomorph sind; falls ja, wird F durch B_0 repräsentiert. Ansonsten sorgen wir dafür, dass sich B_0 und B_1 isomorphe Untergraphen teilen.

Beispiel: BDD-Konstruktion

Wir betrachten die Formel aus Beispiel 2:

$$F \equiv \Big(carry \leftrightarrow (x_1 \wedge x_2)\Big) \wedge \Big(sum \leftrightarrow (x_1 \vee x_2) \wedge \neg carry\Big)$$

Es gilt z.B.:

$$F[x_1/0] \equiv \neg carry \wedge (sum \leftrightarrow x_2)$$

$$F[x_1/1] \equiv (carry \leftrightarrow x_2) \wedge (sum \leftrightarrow \neg x_2)$$

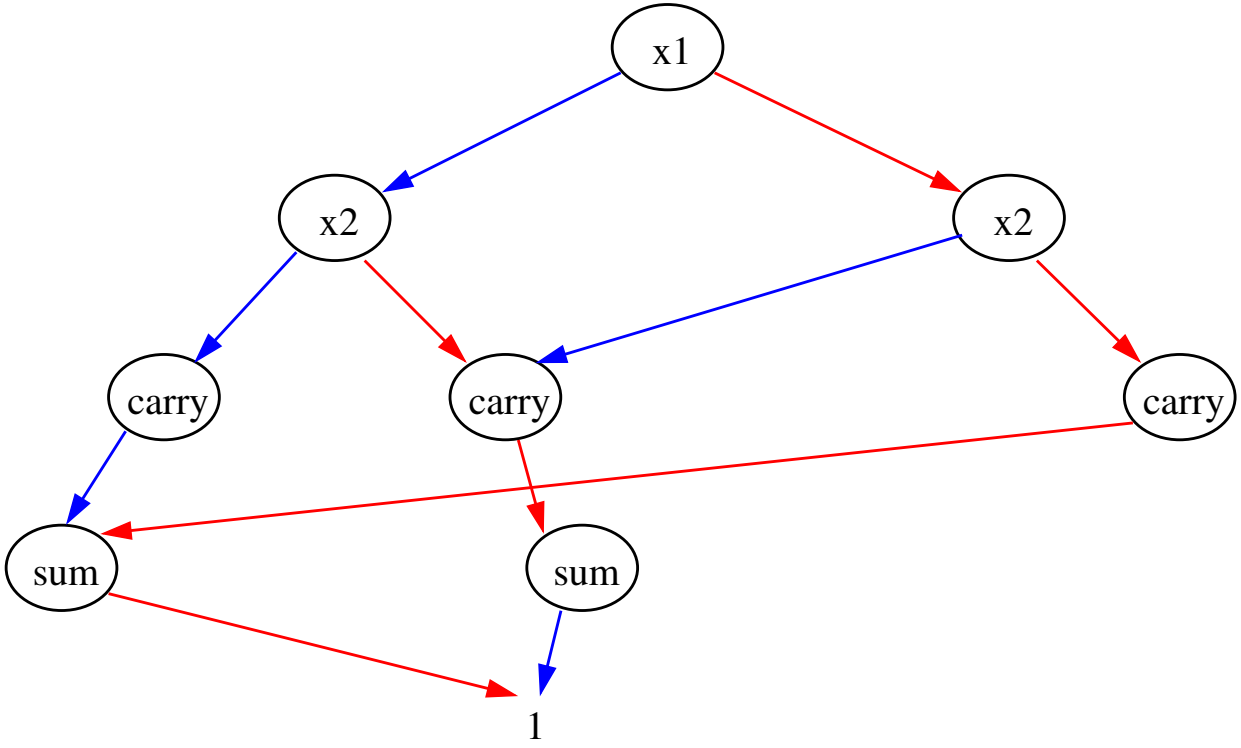
$$F[x_1/0][x_2/0] \equiv \neg carry \wedge \neg sum$$

$$F[x_1/0][x_2/1] \equiv F[x_1/1][x_2/0] \equiv \neg carry \wedge sum$$

$$F[x_1/1][x_2/1] \equiv carry \wedge sum$$

Beispiel: BDD-Konstruktion

Durch vollständige Anwendung erhalten wir denselben BDD wie vorher.



Bemerkung: Umgekehrt können wir aus jedem BDD eine INF-Formel ablesen.

Eindeutigkeit von BDDs

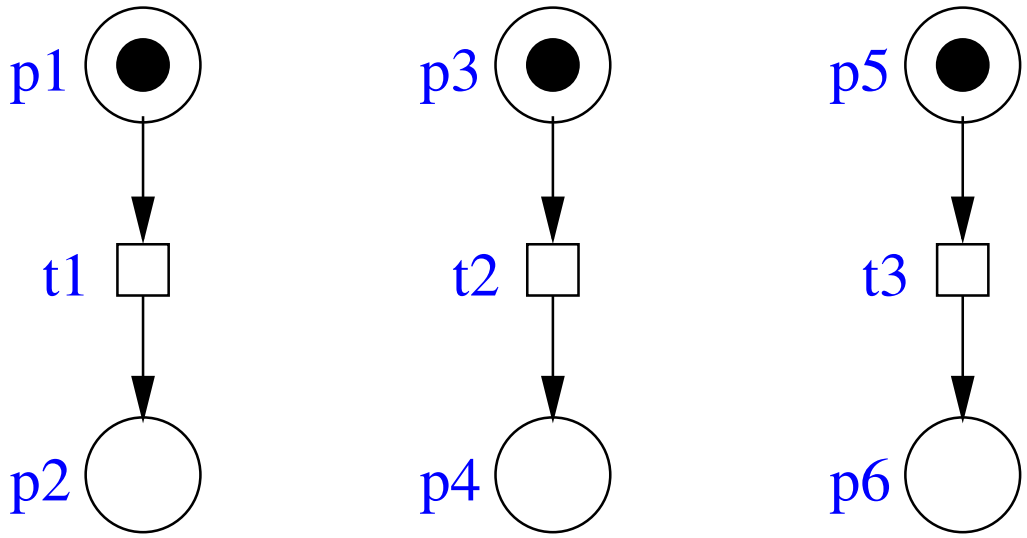
Bemerkung: Das Ergebnis der zuvor angegebenen Konstruktion ist (bis auf Isomorphie) *eindeutig*.

D.h., zu jeder Kombination von Formel F und Ordnung $<$ gibt es (bis auf Isomorphie) *genau ein* BDD.

Bemerkung: Unterschiedliche Ordnungen verursachen für unterschiedliche BDDs (mit unterschiedlicher Größe!).

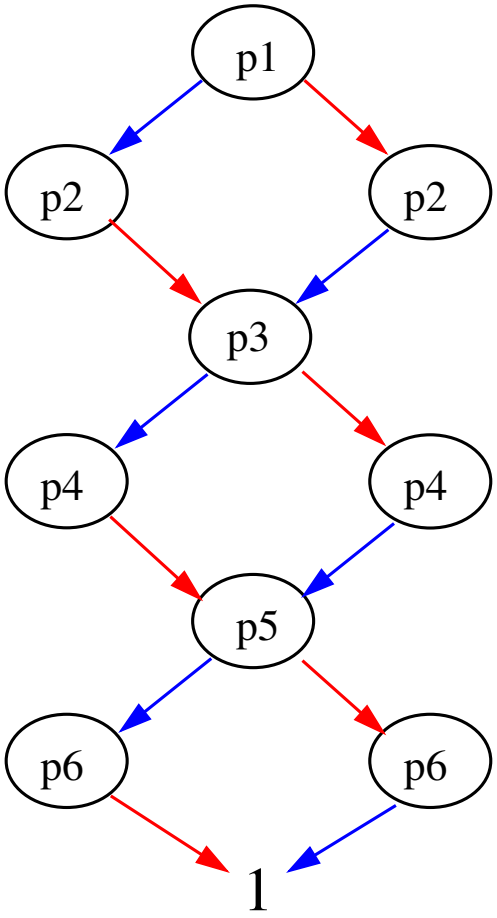
Beispiel: Variablenordnungen

Wir betrachten nochmals Beispiel 1 (Petri-Netz) und konstruieren ein BDD für die erreichbaren Markierungen:



Bemerkung: P_1 ist markiert gdw. P_2 unmarkiert ist etc.

BDD mit Variablenordnung $p_1 < p_2 < p_3 < p_4 < p_5 < p_6$:



Bemerkungen:

Wenn wir in Beispiel 1 die Anzahl der Komponenten von 3 auf n (für $n \geq 0$) ändern, wird die Größe des entsprechenden BDDs linear in n sein.

D.h., ein BDD von Größe n kann 2^n (oder gar mehr) Belegungen darstellen.

Die Größe hängt aber stark von der Ordnung ab.
Beispiel: Man führe obige Konstruktion für folgende Ordnung durch:

$$p_1 < p_3 < p_5 < p_2 < p_4 < p_6$$

Äquivalenztest

Durch die Eindeutigkeit von BDDs haben wir eine Lösung für den benötigten Äquivalenztest.

Problem: Gegeben zwei BDDs B und C. Repräsentieren B und C äquivalente Formeln?

Lösung: Teste, ob B und C isomorph sind.

Bemerkung:

Test auf Unerfüllbarkeit: Prüfe, ob der BDD nur aus dem 0-Knoten besteht.

Test auf Tautologie: Prüfe, ob der BDD nur aus dem 1-Knoten besteht.

Implementierung mit Hash-Tabellen

Angenommen, wir wollen eine Mehrzahl von BDDs verwalten. (Oder auch nur einen; jeder Untergraph ist ja wieder ein BDD.)

Implementierungen von BDDs nutzen die Eindeutigkeit aus, indem sie die Knoten in einer Hash-Tabelle speichern. (Jeder Knoten ist Wurzel eines BDDs.)

Jeder BDD wird durch einen Zeiger auf seine Wurzel repräsentiert.

Zu Beginn werden eindeutige Knoten 0 und 1 abgespeichert.

Alle anderen Knoten sind eindeutig durch ein Tripel (x, B_0, B_1) definiert, wobei x die Grundaussage auf der Wurzel ist und B_0, B_1 die Unterbäume, die durch ihre (eindeutigen) Wurzelknoten repräsentiert sind.

Eine Funktion $mk(x, B_0, B_1)$ schaut in der Hash-Tabelle nach, ob es bereits einen solchen Knoten gibt; wenn ja, wird dieser zurückgeliefert, sonst ein neuer Hash-Eintrag erzeugt.

Eine Mehrzahl von BDDs ist dann ein “Wald” (DAG mit mehreren Wurzeln).

Problem: Garbage-Collection (durch Reference Counting)

Äquivalenztest II

Wir betrachten nochmals das Äquivalenzproblem:

Problem: Gegeben zwei BDDs *B* und *C*. Repräsentieren *B* und C äquivalente Formeln?

Wenn *B* und *C* wie beschrieben in Hash-Tabellen abgespeichert werden, so sind *B* und *C* durch Zeiger auf ihre Wurzeln repräsentiert.

Der Äquivalenztest kann nun in konstanter Zeit durchgeführt werden: Teste, ob die beiden Zeiger gleich sind.

Logische Operationen I: Komplement

Sei F eine Formel der AL und B der zugehörige BDD.

Problem: Berechne ein BDD für $\neg F$.

Lösung: Vertausche die 0- und 1-Knoten von B.

(Achtung: Nicht so einfach bei Implementierung mit Hash-Tabelle!)

Logische Operationen II: Disjunktion/Vereinigung

Seien F, G Formeln der AL und B, C die zugehörigen BDDs (mit derselben Variablenordnung).

Problem: Berechne ein BDD für $F \vee G$ aus B und C.

Es gilt folgende Äquivalenz:

$$F \vee G \equiv ite(x, (F \vee G)[x/1], (F \vee G)[x/0]) \equiv ite(x, F[x/1] \vee G[x/1], F[x/0] \vee G[x/0])$$

Wenn x die kleinste Variable in F und G ist, so sind $F[x/1], F[x/0], G[x/1], G[x/0]$ entweder die Kinder der Wurzeln von B und C oder die Wurzeln selbst.

Wir konstruieren den Disjunktions-BDD nach folgender, rekursiver Vorschrift:

Wenn B und C gleich sind, gib B zurück.

Wenn entweder B oder C nur aus dem 1-Knoten bestehen, gib 1 zurück.

Wenn entweder B oder C nur aus dem 0-Knoten bestehen, gib den jeweils anderen BDD zurück.

Ansonsten vergleiche die Variablen, mit denen die Wurzeln von B und C beschriftet sind. Sei x die kleinere davon bzw. die Variable, die beide Wurzeln beschriftet.

Wenn die Wurzel von B mit x beschriftet ist, seien B_1, B_0 die Unterbäume von B; andernfalls seien $B_1, B_0 := B$; analog seien C_1, C_0 definiert.

Wende die Vorschrift rekursiv auf die Paare B_1, C_1 und B_0, C_0 an, was zu BDDs E, F führt. Falls $E = F$, gib E zurück, sonst $mk(x, E, F)$.

Logische Operationen III: Schnitt

Seien F, G Formeln der AL und B, C die zugehörigen BDDs (mit derselben Variablenordnung).

Problem: Berechne ein BDD für $F \wedge G$ aus B und C.

Lösung: Analog zu Vereinigung, mit angepassten Vorschriften bzgl. der $\mathbf{0}$- und $\mathbf{1}$-Knoten.

Komplexität: Mit dynamischer Programmierung: $\mathcal{O}(|B| \cdot |C|)$ (jede Kombination von Knoten einmal).

Berechnung der Vorgänger

Im Folgenden wollen wir die Vorgängermenge

$$pre(M) = \{ s \mid \exists s' : (s, s') \in \rightarrow \ \wedge \ s' \in M \}$$

berechnen.

Die Relation $\rightarrow$ ist eine Untermenge von $S \times S$, während $M \subset S$ ist.

Wir repräsentieren M durch ein BDD mit Variablen $y_1, \ldots, y_m$.

$\rightarrow$ wird durch ein BDD mit Variablen $x_1, \ldots, x_m$ und $y_1, \ldots, y_m$ repräsentiert (Zustand “vorher” und “nacher”).

Bemerkung: Jedes BDD für M ist zugleich ein BDD für $S \times M$!

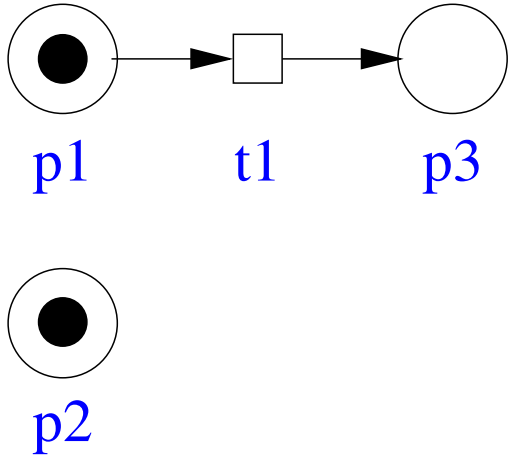
Wir können $pre(M)$ wie folgt umschreiben:

$$\{ s \mid \exists s' : (s, s') \in \rightarrow \ \cap \ (S \times M) \}$$

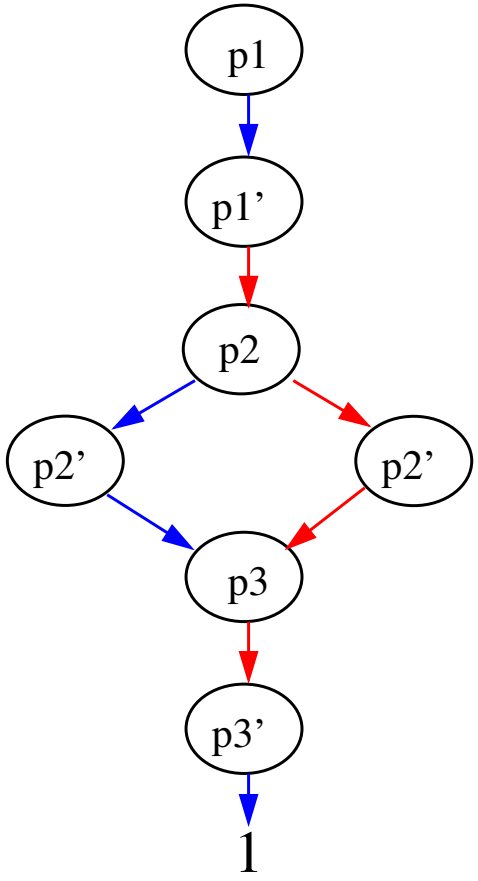
Dann reduziert sich pre auf die Operationen Schnitt und existenzielle Abstraktion.

Beispiel

Wir betrachten das folgende Petri-Netz mit einer Transition:



Das BDD F_{t_1}, das den Effekt von t_1 beschreibt, wobei p_1, p_2, p_3 den Zustand *vor* und p'_1, p'_2, p'_3 den Zustand *nach* t_1 beschreiben.



Existenzielle Abstraktion

Existenzielle Abstraktion bzgl. der Grundaussage x ist wie folgt definiert:

$$\exists x : F \quad \equiv \quad F[x/0] \vee F[x/1]$$

D.h., $\exists x : F$ ist wahr für die Belegungen, die so mit einem Wert für x erweitert werden können, dass F wahr wird.

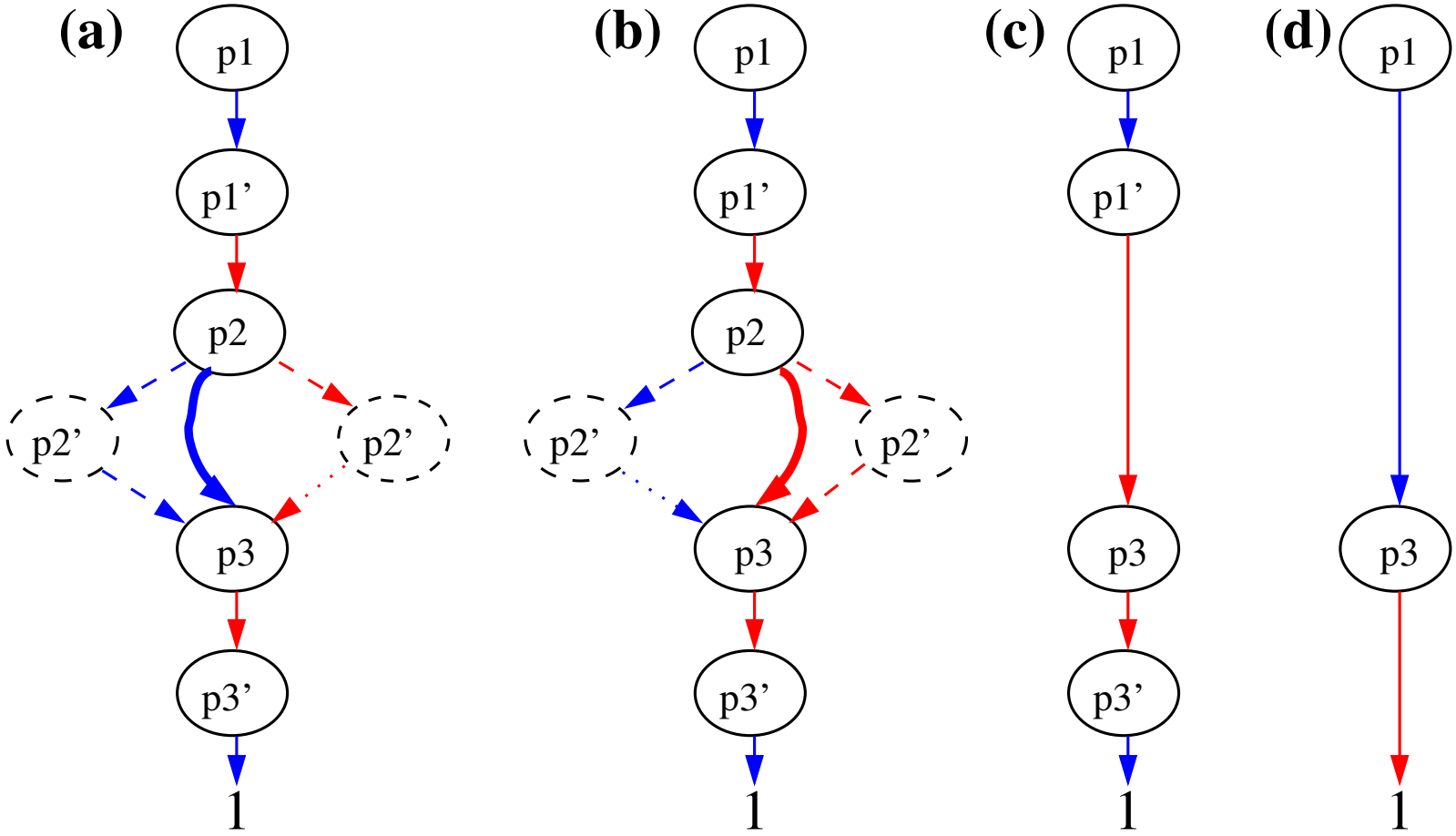
Beispiel: Sei $F \equiv (x_1 \wedge x_2) \vee x_3$. Dann

$$\exists x_1 : F \equiv F[x_1/0] \vee F[x_1/1] \equiv (x_3) \vee (x_2 \vee x_3) \equiv x_2 \vee x_3$$

Als Erweiterung können wir existenzielle Abstraktion über Mengen betrachten (abstrahiere jede einzelne Variable).

Beispiel: Existenzielle Abstraktion

(a) $F_{t_1}[p'_2/1]$; (b) $F_{t_1}[p'_2/0]$; (c) $\exists p'_2 : F_{t_1}$; (d) $\exists p'_1, p'_2, p'_3 : F_{t_1}$



BDDs mit Komplementkanten

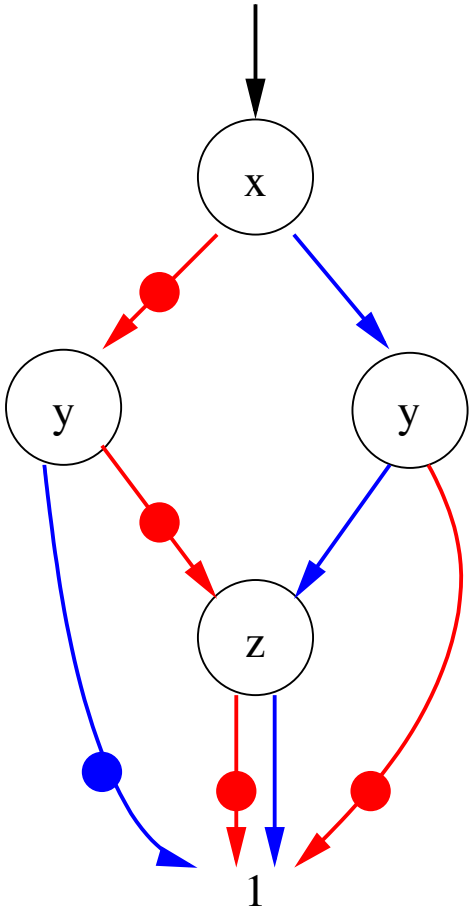
Implementierung mit Hash-Tabelle: Negation aufwändig

BDD-Bibliotheken implementieren i.d.R. BDDs mit Komplementkanten (KBDDs).

Jede Kante ist mit einem zusätzlichen Bit versehen. Wenn das Bit gesetzt ist, bedeutet dies, dass als Ziel der Kante die *Negation* des dort beginnenden BDDs gemeint ist.

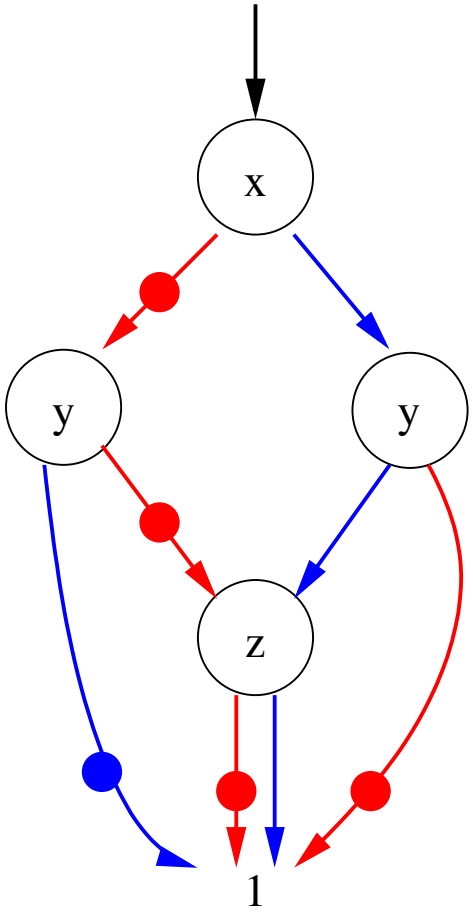
Darstellung im Folgenden: gesetztes Bit = ausgefüllter Kreis

KBDD: Beispiel



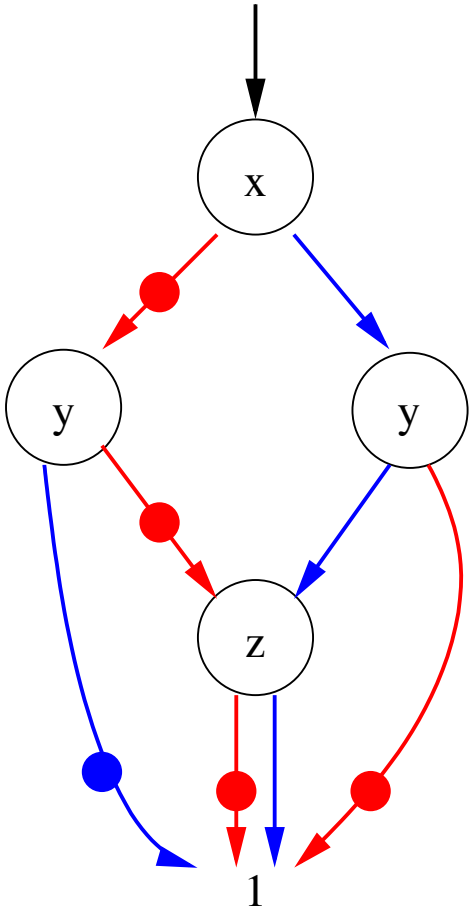
Die rote Kante des z-Knotens ist als “Komplement” markiert, sie führt daher effektiv zur 0.

KBDD: Beispiel



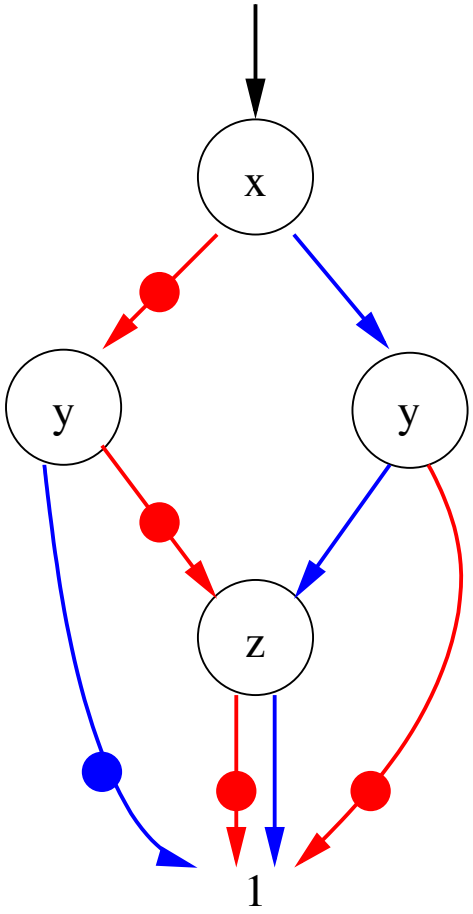
Aus diesem Grund ist der z-Knoten *nicht redundant*.
Der 0-Knoten kann entfallen.

KBDD: Beispiel



Der linke y-Knoten repräsentiert die Formel $\neg y \wedge \neg z$.

KBDD: Beispiel



Auch der Zeiger auf die Wurzel ist mit einem Komplementbit versehen (in diesem Falle nicht gesetzt).

Bemerkungen über KBDDs

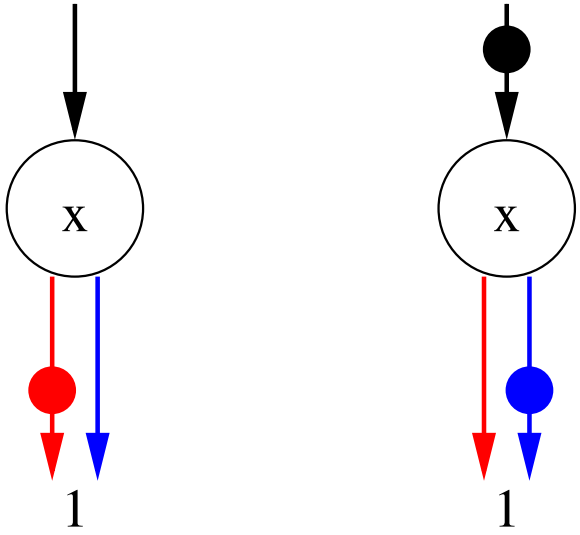
Eine Belegung ist Modell der vom KBDD repräsentierten Formel gdw. die Anzahl der Negationen auf dem entsprechenden Pfad zur 1 (inklusive Zeiger auf die Wurzel) *gerade* ist.

Negation mit KBDDs: sehr einfach, negiere das Komplementbit der Kante, die auf die Wurzel zeigt ($\mathcal{O}(1)$ Zeit).

Implementierung (in der CUDD-Bibliothek): Codierung des Bits im LSB des Zeigers

Problem: KBDDs sind nicht (ohne Weiteres) eindeutig.

Nicht-Eindeutigkeit von KBDDs



Beide obenstehenden KBDDs repräsentieren die Formel x.

Kanonizitätsbedingung

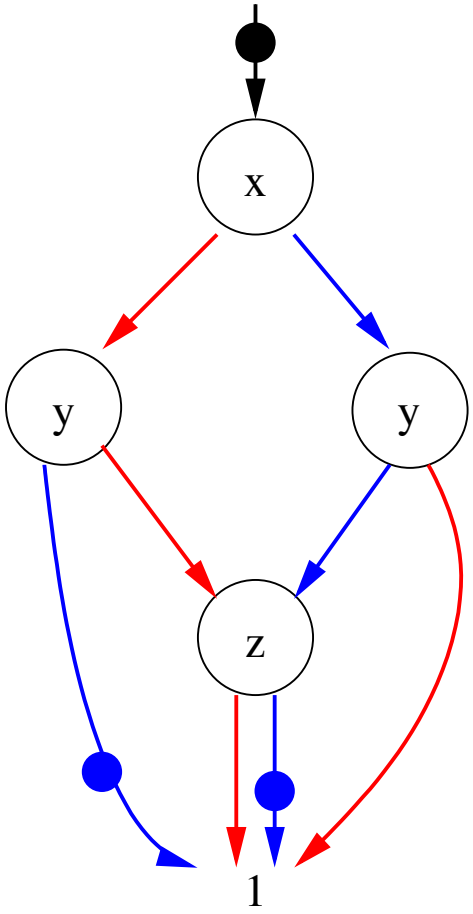
Um KBDDs dennoch eindeutig zu machen, fordert man zusätzlich, dass mit 0 beschriftete (rote) Kanten kein Komplementbit haben dürfen.

Dazu nutzt man folgende Äquivalenz aus:

$$ite(x, F, \neg G) \equiv \neg ite(x, \neg F, G)$$

Falls man also doch eine negierte 0-Kante hat, dreht man einfach alle Bits auf den Kanten um, die mit ihrem Startknoten inzident sind (Vorgehensweise: von unten nach oben).

Kanonischer BDD



Obenstehender KBDD drückt dieselbe Formel wie vorhin aus und hat keine negierten 0-Kanten.

LTL mit BDDs

Frage: Kann man auch LTL-Modelchecking mit Hilfe von BDDs effizient implementieren?

Antwort: Jein (Worst-case: quadratisch; praktisch: ganz gut)

Probleme: BDD nicht kombinierbar mit Tiefensuche; Kombination mit Halbordnungsreduktion schwierig

Symbolische Algorithmen für LTL

Idee: Finde eine nicht-triviale SCC mit einem akzeptierenden Zustand, suche rückwärts nach dem Anfangszustand

Lösungen: SCC-Dekomposition, EL, OWCTY

SCC-Dekomposition

(basierend auf Xie/Beerel 1999)

Wähle einen beliebigen Zustand s.

Berechne alle Vorgänger und alle Nachfolger von s und schneide die beiden Mengen. Das Ergebnis ist eine SCC.

Entferne die SCC aus dem Graphen und wiederhole, bis der Graph leer ist.

Stelle den Graphen wieder her und berechne rückwärts von allen nicht-trivialen SCCs mit akzeptierendem Zustand

Problem: sehr schlecht, falls das System viele triviale SCCs hat

Emerson-Lei (EL) (1986)

1. Setze M auf die Menge aller Zustände

2. Setze $B := M \cap F$.

3. Berechne C als die Menge aller Zustände, die B erreichen können.

4. Setze $M := M \cap pre(C)$.

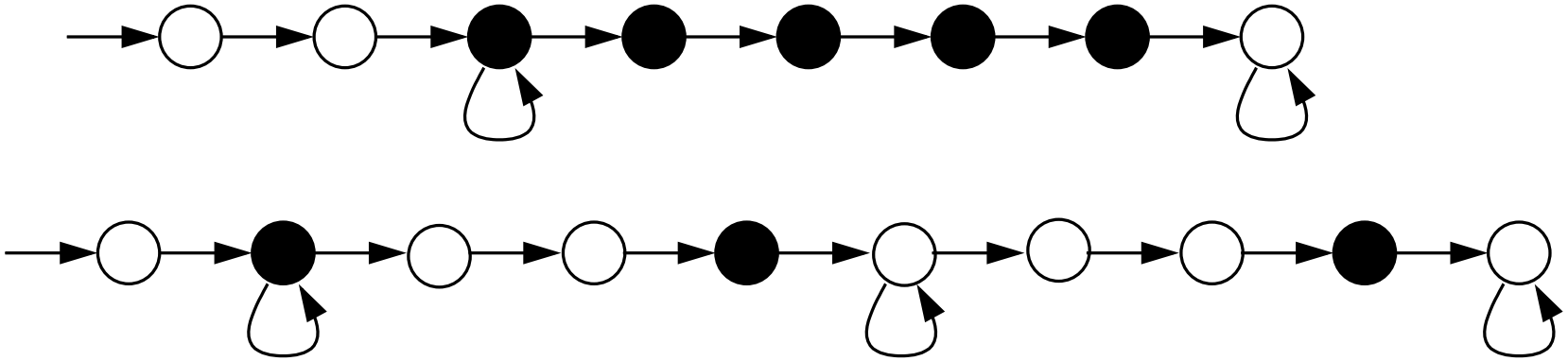
5. Gehe zu 2, bis sich M nicht mehr ändert.

One-Way-Catch-Them-Young

(Hardin et al 1997, Fisler et al 2001)

Ähnlich wie EL, zusätzlich wird Schritt 4 solange wiederholt, bis sich M nicht mehr ändert.

EL und OWCTY



Im oberen Fall funktioniert OWCTY besser als EL, im unteren Fall umgekehrt.

In praktischen Fällen scheint OWCTY besser zu funktionieren.

Teil 13: Abstraktion

Beispiel 1 (Schleife)

Problem: Programm mit drei numerischen Variablen x, y, z.

ℓ_1: `y = x+1;`

ℓ_2: `z = 0;`

ℓ_3: `while (z < 100) z = z+1;`

ℓ_4: `if (y < x) error;`

Frage: Ist der Fehler erreichbar?

Beispiel 2 (Sortierung)

Problem: Programm mit drei numerischen Variablen x, y, z.

```
ℓ1: if x > y then  swap x,y  else  skip;

ℓ2: if y > z then  swap y,z  else  skip;

ℓ3: if x > y then  swap x,y  else  skip;

ℓ4: skip
```

Anfangsbedingung: x, y, z paarweise verschieden

Frage: Sind x, y, z bei Erreichen von ℓ_4 aufsteigend sortiert?

Beispiel 3 (Gerätetreiber)

C-Code für Windows-Gerätetreiber

Operationen auf einem Semaphor: Lock, Release

Lock, Release müssen immer abwechselnd ausgeführt werden

Abstraktion

Idee: “unwichtige” Informationen wegwerfen (abstrahieren)

Behandlung *unendlicher* Zustandsräume

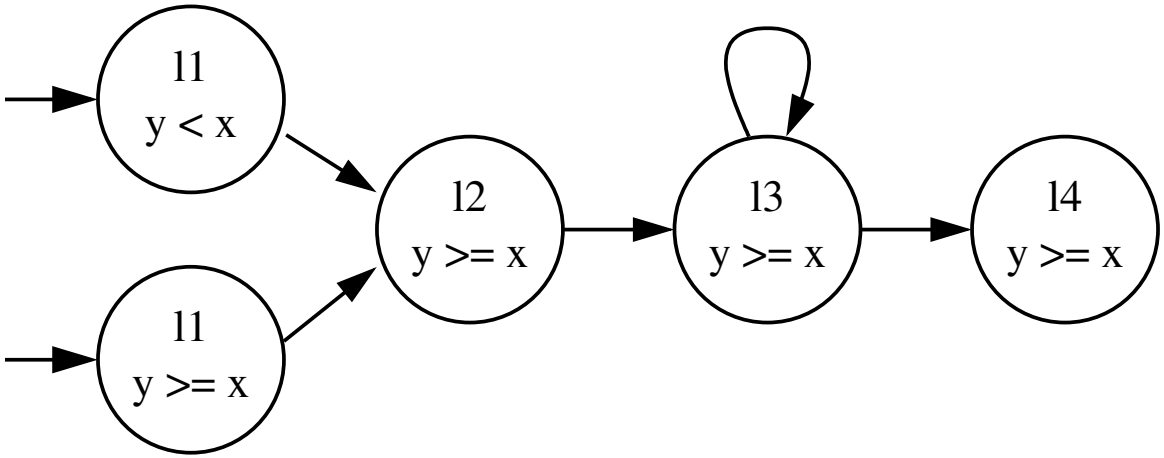
Verkleinerung bereits endlicher Zustandsräume

Alternative Sichtweise: “gleichwertige” Zustände zusammenfassen

Beispiel 1

Konkrete Werte von `x, y, z` weglassen; einzige wichtige Informationen:
Programmzähler, Information $y < x$

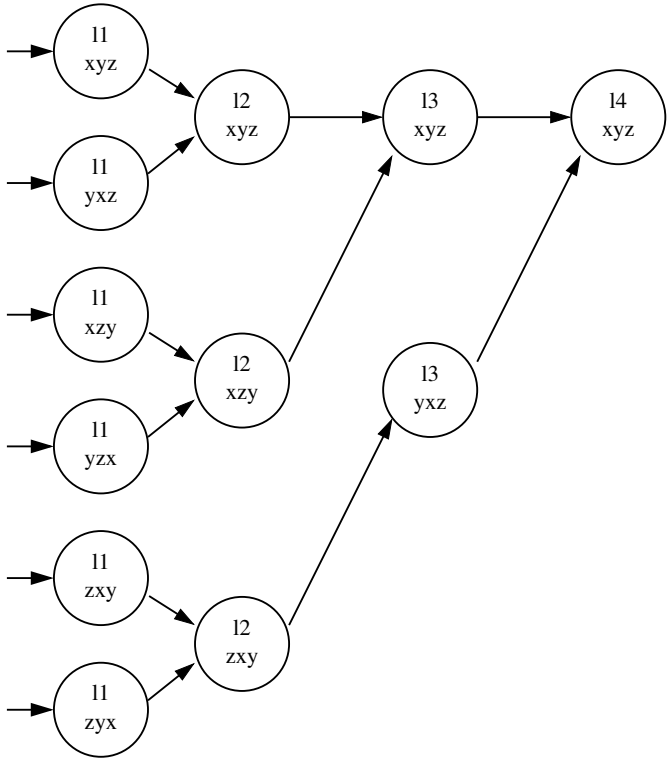
Resultierende (abstrakte) Kripkestruktur:



Ergebnis: ℓ_4 ist nur mit $y \geq x$ erreichbar, also kein Fehler

Beispiel 2

Konkrete Werte von `x`, `y`, `z` weglassen; einzige wichtige Informationen: Programmzähler, Permutation von x, y, z



Ergebnis: ℓ_4 ist nur mit *xyz* erreichbar, also kein Fehler

Aber: Was sagen uns die Abstraktionen eigentlich über das ursprüngliche Programm?

In Beispiel 1 ist der Fehler weder in der ursprünglichen noch in der abstrakten Struktur zu erreichen.

In Beispiel 1 terminiert die ursprüngliche Struktur, nicht aber die abstrakte.

Welche Bedingungen muss die abstrakte Struktur erfüllen, um noch korrekte Aussagen über die ursprüngliche Struktur zu machen?

Simulation

Seien $\mathcal{K}_1 = (S, \rightarrow_1, s_0, AP, \nu)$ und $\mathcal{K}_2(T, \rightarrow_2, t_0, AP, \mu)$ zwei Kripke-Strukturen (S, T womöglich *nicht* endlich), und sei $H \subseteq S \times T$ eine Relation.

H heißt Simulation von $\mathcal{K}_1$ nach $\mathcal{K}_2$ gdw.

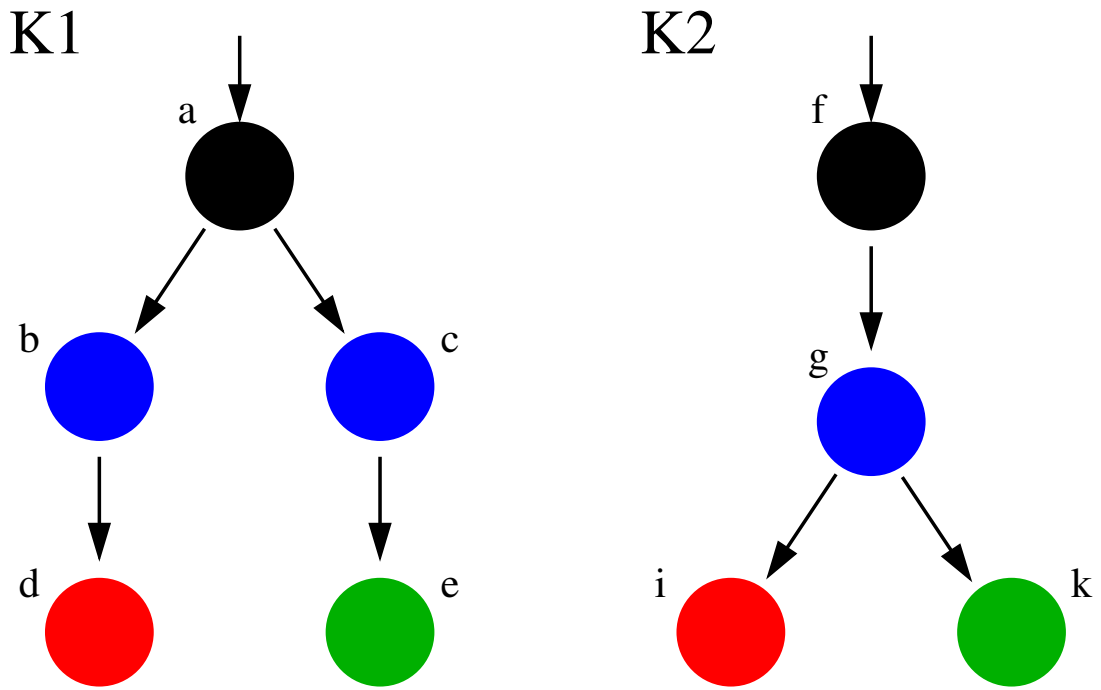
(i) $(s_0, t_0) \in H$;

(ii) für alle $(s, t) \in H$ gilt: $\nu(s) = \mu(t)$;

(iii) falls $(s, t) \in H$ und $s \rightarrow_1 s'$, dann existiert t' mit $t \rightarrow_2 t'$ und $(s', t') \in H$.

Wir sagen: $\mathcal{K}_2$ simuliert $\mathcal{K}_1$ (geschrieben $\mathcal{K}_1 \leq \mathcal{K}_2$), falls eine solche Simulation H existiert.

Intuition: In $\mathcal{K}_2$ ist alles möglich, was auch in $\mathcal{K}_1$ möglich ist.



$\mathcal{K}_2$ simuliert $\mathcal{K}_1$ (mit $H = \{(a,f),(b,g),(c,g),(d,i),(e,k)\}$).

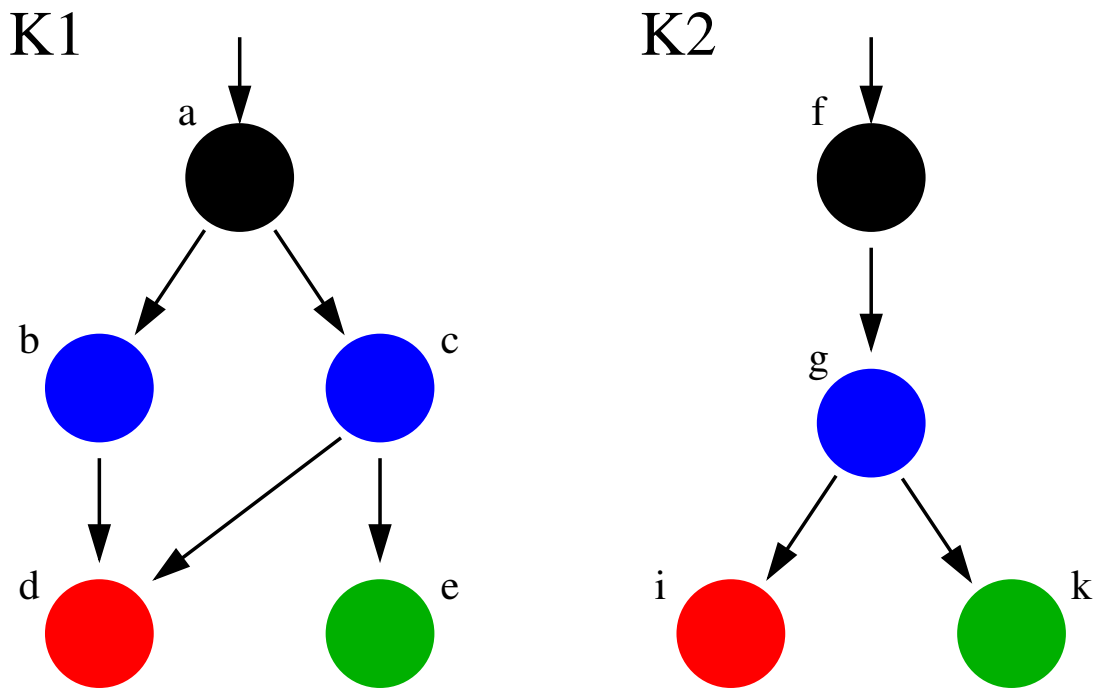
Aber: $\mathcal{K}_1$ simuliert *nicht* $\mathcal{K}_2$!

Bisimulation

Eine Relation H heißt Bisimulation zwischen $\mathcal{K}_1$ und $\mathcal{K}_2$ gdw. H eine Simulation von $\mathcal{K}_1$ nach $\mathcal{K}_2$ ist und $\{ (t,s) \mid (s,t) \in H \}$ eine Simulation von $\mathcal{K}_2$ nach $\mathcal{K}_1$.

Wir sagen: $\mathcal{K}_1$ und $\mathcal{K}_2$ sind bisimilar (geschrieben $\mathcal{K}_1 \equiv \mathcal{K}_2$) gdw. eine solche Relation H existiert.

Vorsicht: Aus $\mathcal{K}_1 \leq \mathcal{K}_2$ und $\mathcal{K}_2 \leq \mathcal{K}_1$ folgt *nicht* $\mathcal{K}_1 \equiv \mathcal{K}_2$!



(Bi-)Simulation und Model-Checking

Sei $\mathcal{K}_1 \leq \mathcal{K}_2$ und ϕ eine LTL-Formel.

Dann gilt: $\mathcal{K}_2 \models \phi$ impliziert $\mathcal{K}_1 \models \phi$.

Sei $\mathcal{K}_1 \equiv \mathcal{K}_2$ und ϕ eine CTL- oder LTL-Formel.

Dann gilt: $\mathcal{K}_1 \models \phi$ gdw. $\mathcal{K}_2 \models \phi$.

Existenzielle Abstraktion

Es sei $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ eine Kripke-Struktur (konkrete Struktur).

Sei $\approx$ eine Äquivalenzrelation auf S, so dass für alle $s \approx t$ gilt: $\nu(s) = \nu(t)$
(wir sagen: $\approx$ respektiert ν).

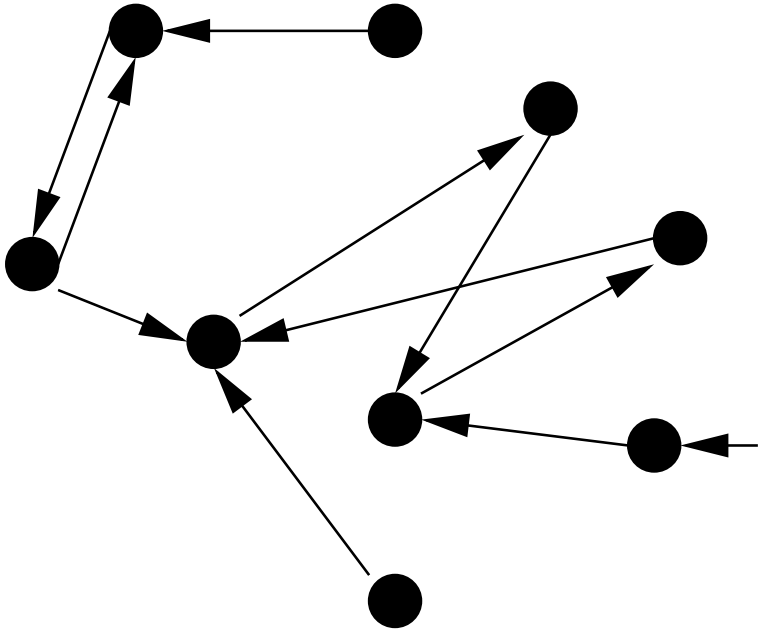
$[s] := \{ t \mid s \approx t \}$ bezeichne die Äquivalenzklasse von s;
$[S]$ bezeichne die Menge aller Äquivalenzklassen.

Die Abstraktion von S bzgl. $\approx$ ist $\mathcal{K}' = ([S], \rightarrow', [r], AP, \nu')$ mit

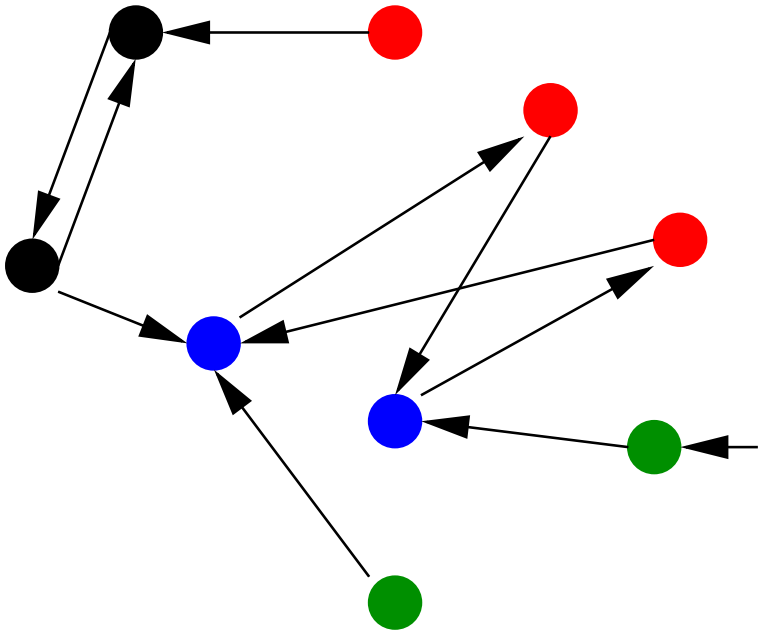
- $[s] \rightarrow' [t]$ für alle $s \rightarrow t$;
- $\nu'([s]) = \nu(s)$ (wohldefiniert).

Beispiel

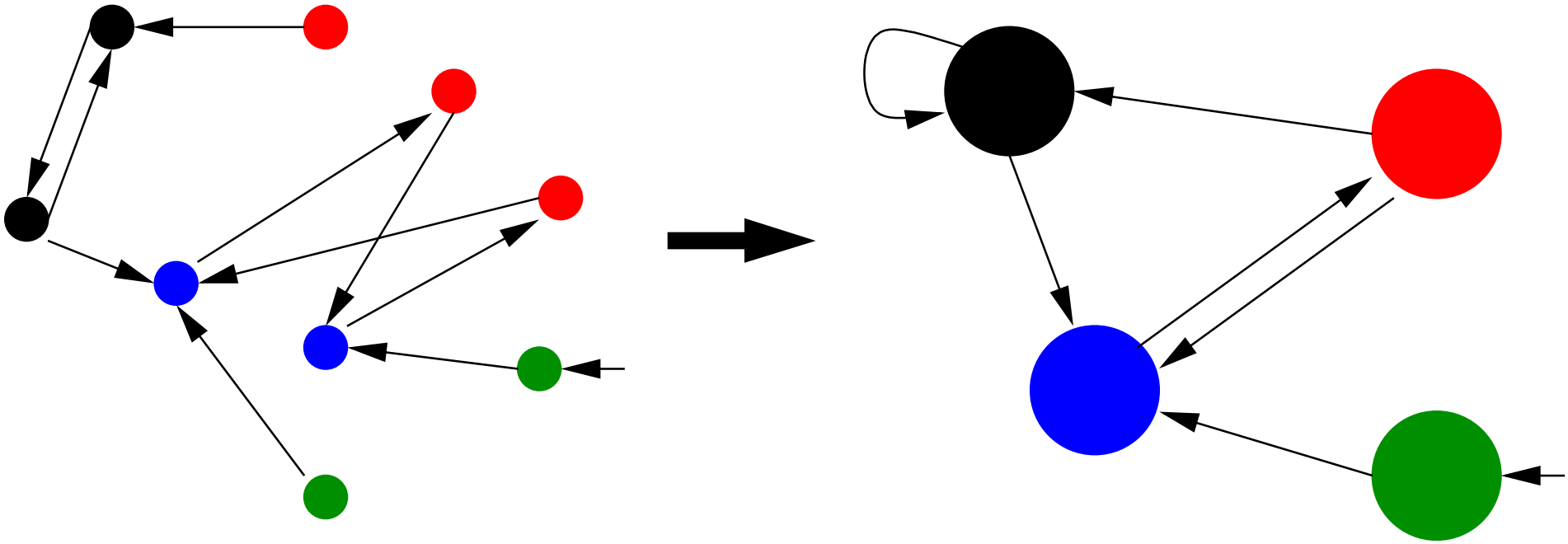
Gegebene Kripke-Struktur:



Zustände in Äquivalenzklassen partitionieren:



Abstrakte Struktur durch Quotientenbildung:



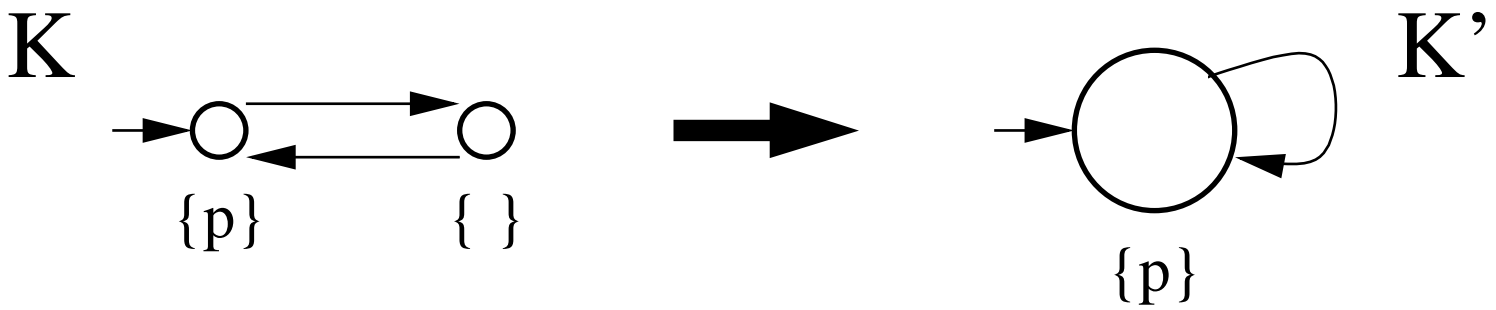
Sei $\mathcal{K}'$ eine Struktur, die durch Abstraktion aus $\mathcal{K}$ entsteht.

Dann gilt: $\mathcal{K} \leq \mathcal{K}'$.

Falls also $\mathcal{K}'$ eine beliebige LTL-Formel erfüllt, gilt dies auch für $\mathcal{K}$.

Bewahrt $\mathcal{K}'$ die Eigenschaften von $\mathcal{K}$?

Was wäre, wenn $\approx$ *nicht* ν respektiert?



Es gilt *nicht* $\mathcal{K} \not\leq \mathcal{K}'$.

Abstraktion erfüllt $\mathrm{G}\, p$, konkretes System nicht.

Sei $\mathcal{K}'$ eine Struktur, die durch Abstraktion aus $\mathcal{K}$ entsteht.

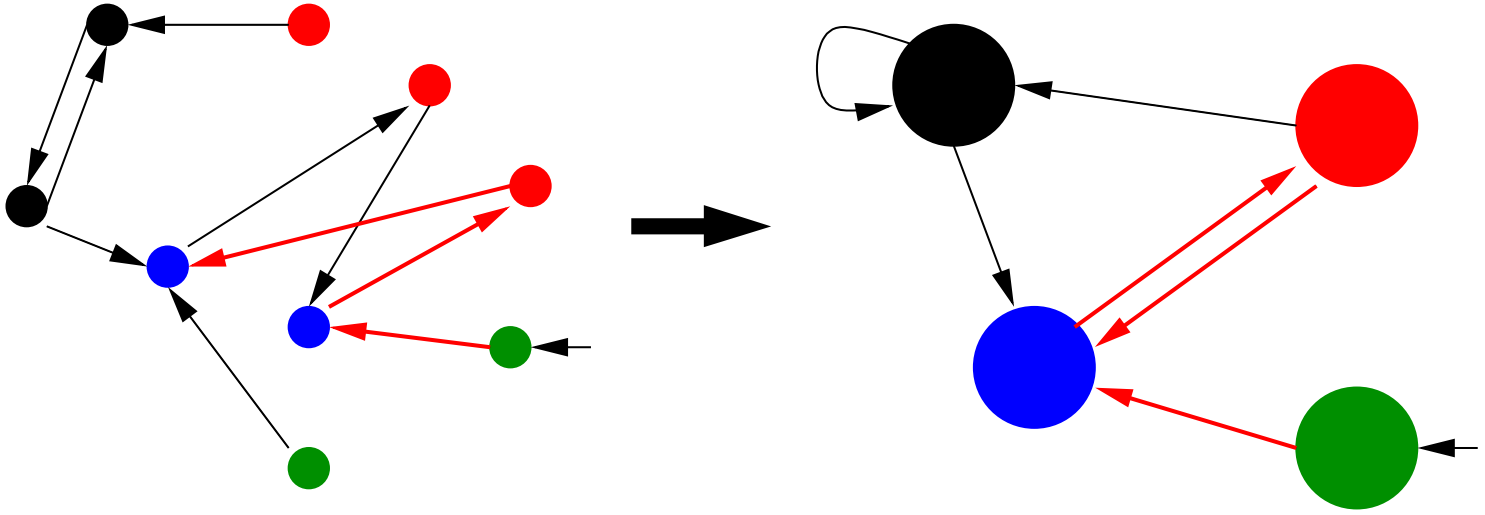
Dann gilt: $\mathcal{K} \leq \mathcal{K}'$.

Falls also $\mathcal{K}'$ eine beliebige LTL-Formel erfüllt, gilt dies auch für $\mathcal{K}$.

Aber: Falls $\mathcal{K}' \not\models \phi$, wissen wir noch nichts über $\mathcal{K} \models \phi$!

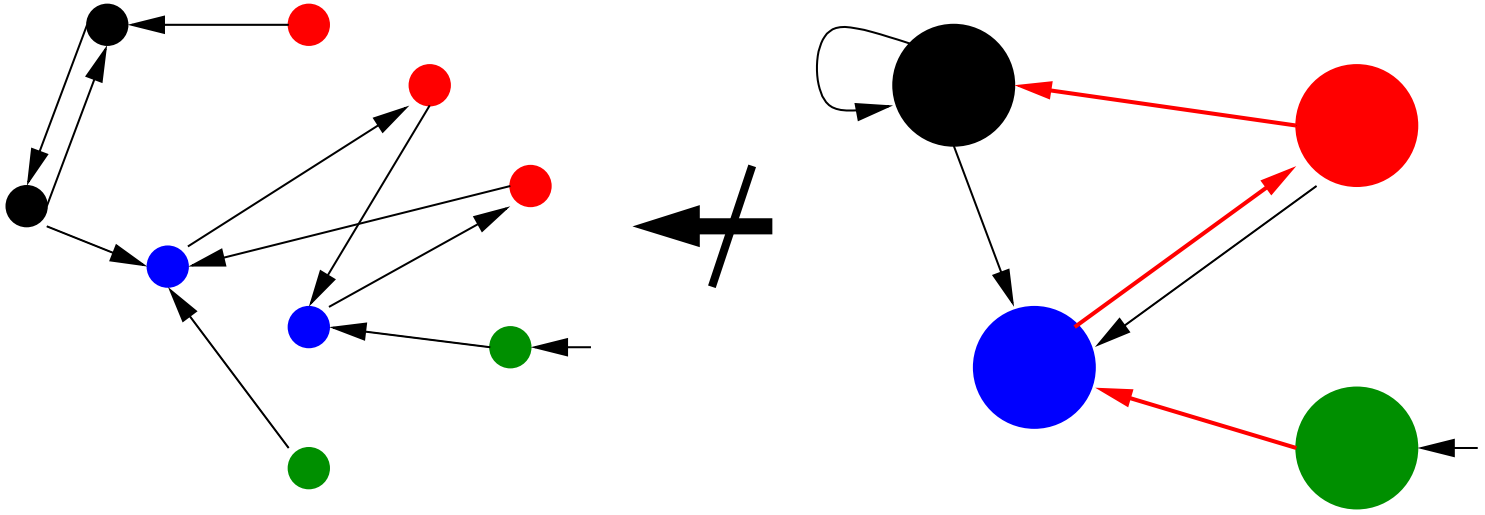
Abstraktion lässt zusätzliche Abläufe entstehen:

Jeder konkrete Ablauf hat seine Entsprechung in der Abstraktion . . .



Abstraktion lässt zusätzliche Abläufe entstehen:

. . . aber nicht jeder abstrakte Ablauf hat eine konkrete Entsprechung.



Angenommen, $\mathcal{K}' \not\models \phi$, wobei ρ als Gegenbeispiel geliefert wird.

Überprüfe, ob ρ eine konkrete Entsprechung hat.

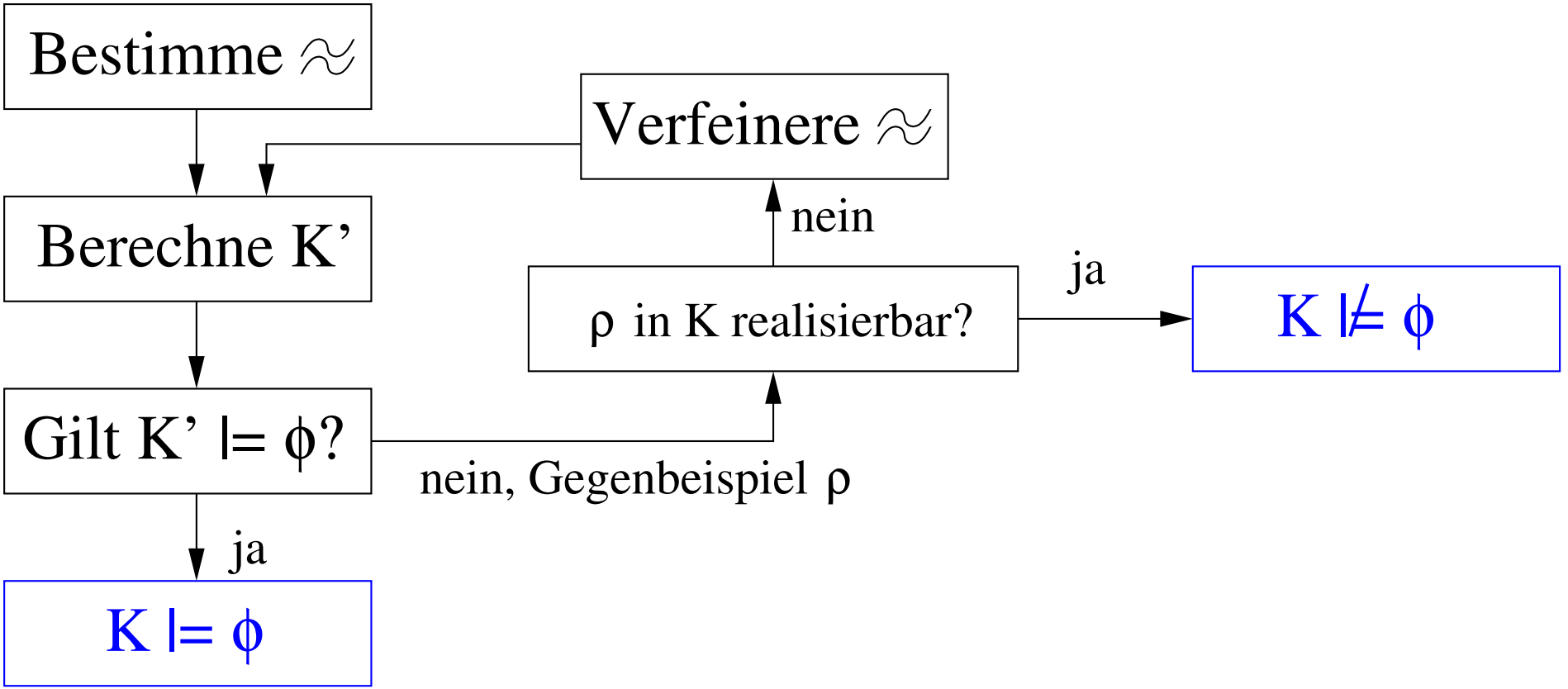
Falls ja, gilt auch $\mathcal{K} \not\models \phi$.

Falls nein, wird das Gegenbeispiel verwendet, um A zu erweitern bzw. h zu verfeinern, damit ρ nicht mehr auftritt.

Der oben beschriebene Schritt kann so lange wiederholt werden, bis eine Antwort für $\mathcal{K}$ bestimmt wurde. Diese Technik nennt man Verfeinerung durch Gegenbeispiele (counter-example guided abstraction refinement, CEGAR) [Clarke et al, 1994].

Der Abstraktions-Verfeinerungs-Zyklus

Eingabe: $\mathcal{K}, \phi$



Simulation von ρ

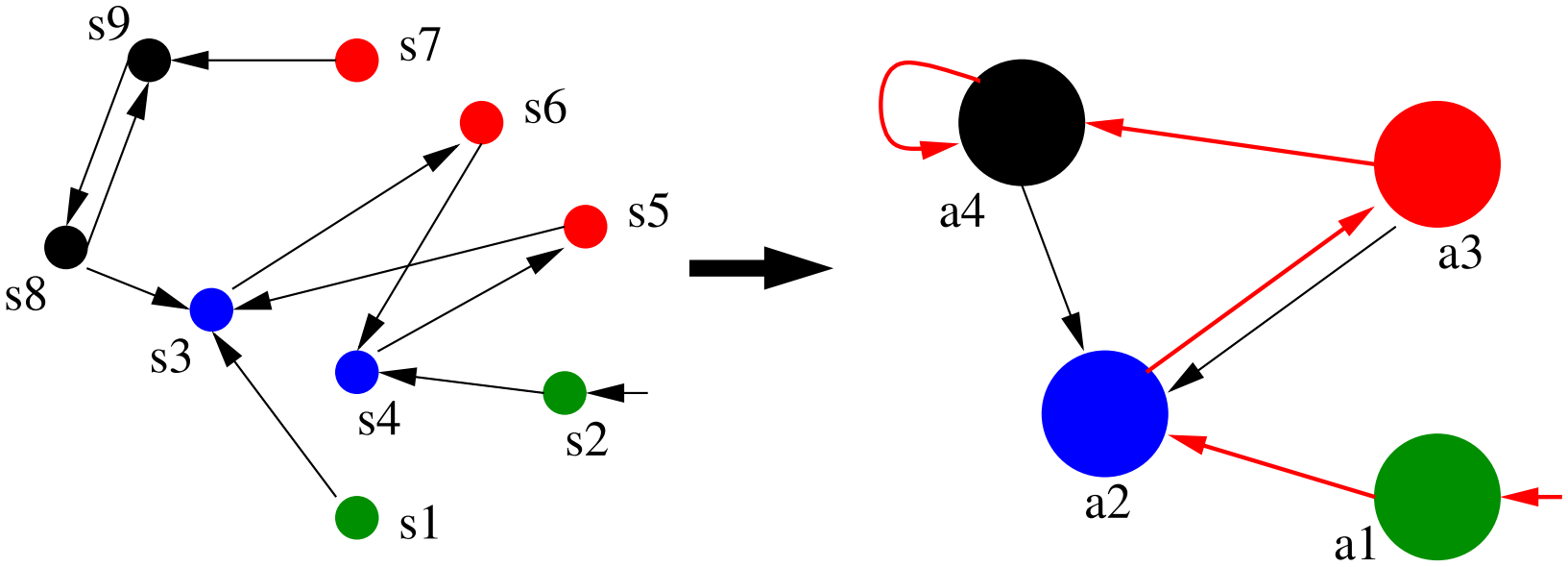
Problem: Gegeben ein Gegenbeispiel ρ, hat ρ eine Entsprechung in $\mathcal{K}$?

Lösung: ρ wird auf $\mathcal{K}$ simuliert.

Bemerkung: Ein Gegenbeispiel ρ läßt sich in einen endlichen Anfang und eine endliche Schleife zerlegen, d.h. $\rho = w_A w_S^\omega$ für geeignete w_A, w_S.

Fallunterscheidung: Simulation kann im Anfang oder in der Schleife scheitern.

Beispiel 1: $\mathrm{G}\,\neg$*schwarz*



Abstraktion liefert Gegenbeispiel mit Anfang $a_1 a_2 a_3 a_4$ und Schleife a_4.

Simulation eines Anfangsstücks

Gegeben: $w_A = b_0 \cdots b_k$.

Beginne mit $S_0 = \{r\}$. (Es gilt $b_0 = [r]$.)

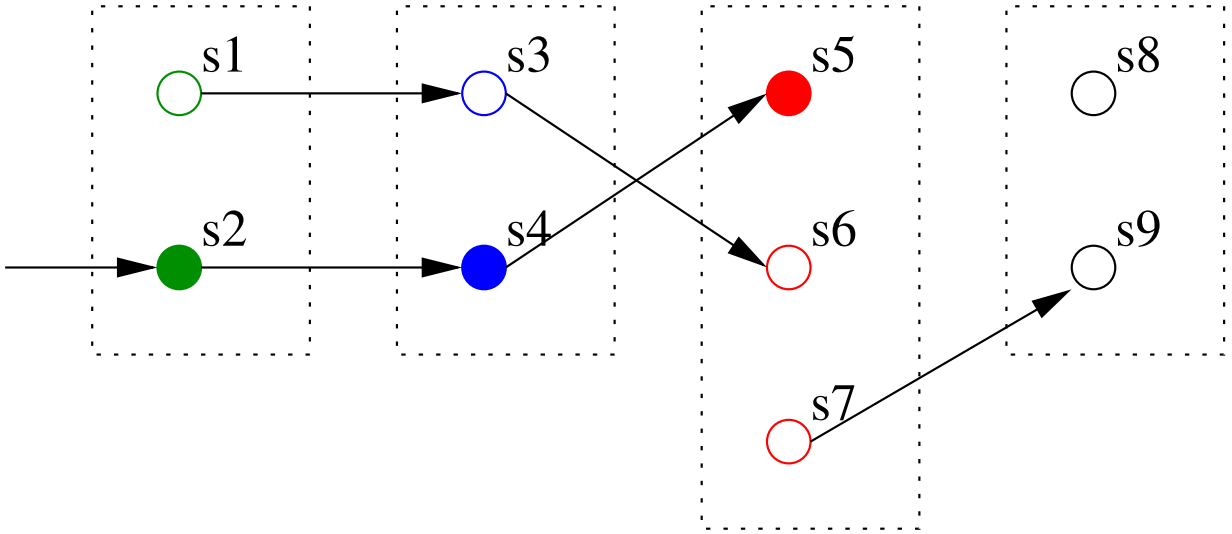
Für $i = 1, \ldots, k$: Setze $S_i = \{\, t \mid t \in b_i \wedge \exists s \in S_{i-1} : s \to t \,\}$.

Falls $S_k \neq \emptyset$, existiert eine konkrete Entsprechung für w_A.

Falls $S_k = \emptyset$: Suche kleinstes ℓ, so dass $S_\ell = \emptyset$: Unterscheide in der verfeinerten Abstraktion die Zustände aus $S_{\ell-1}$ und die $b_{\ell-1}$-Zustände mit b_ℓ-Nachfolgern.

Beispiel: $w_A = a_1 a_2 a_3 a_4$

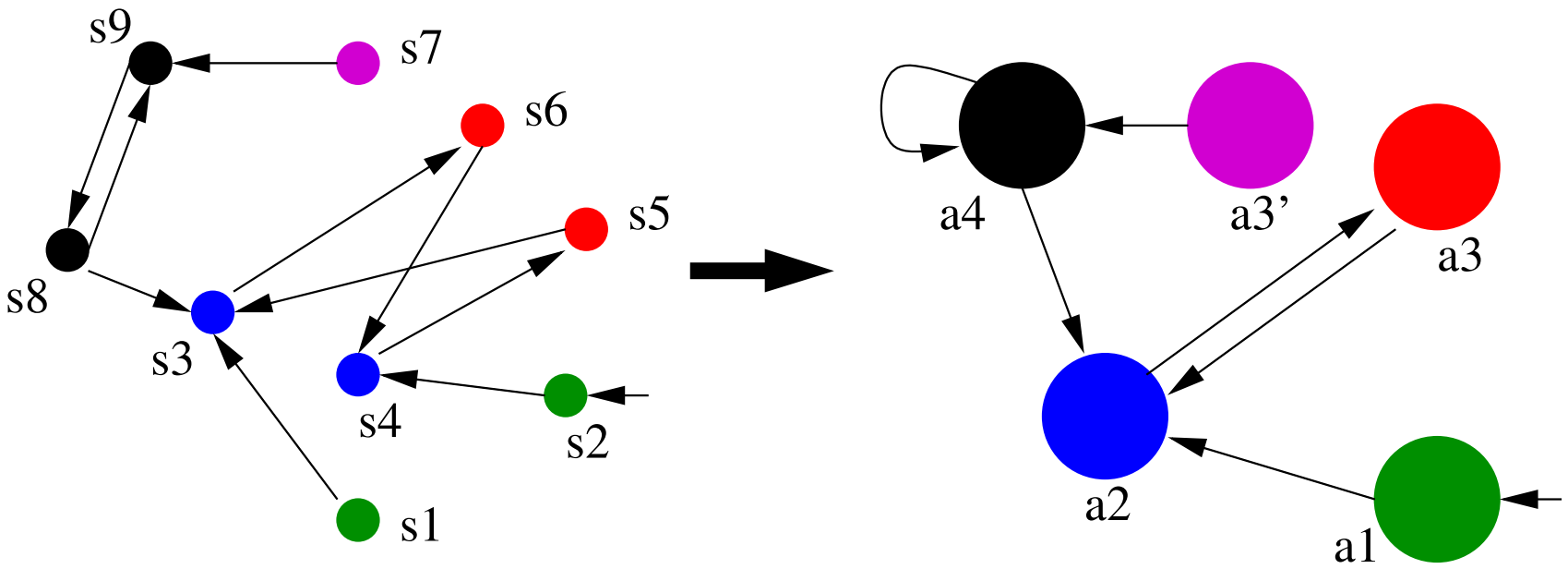
$S_0 = \{s_2\}, \quad S_1 = \{s_4\}, \quad S_2 = \{s_5\}, \quad S_3 = \emptyset.$



Unterscheide in der nächsten Abstraktion s_5 von s_7.

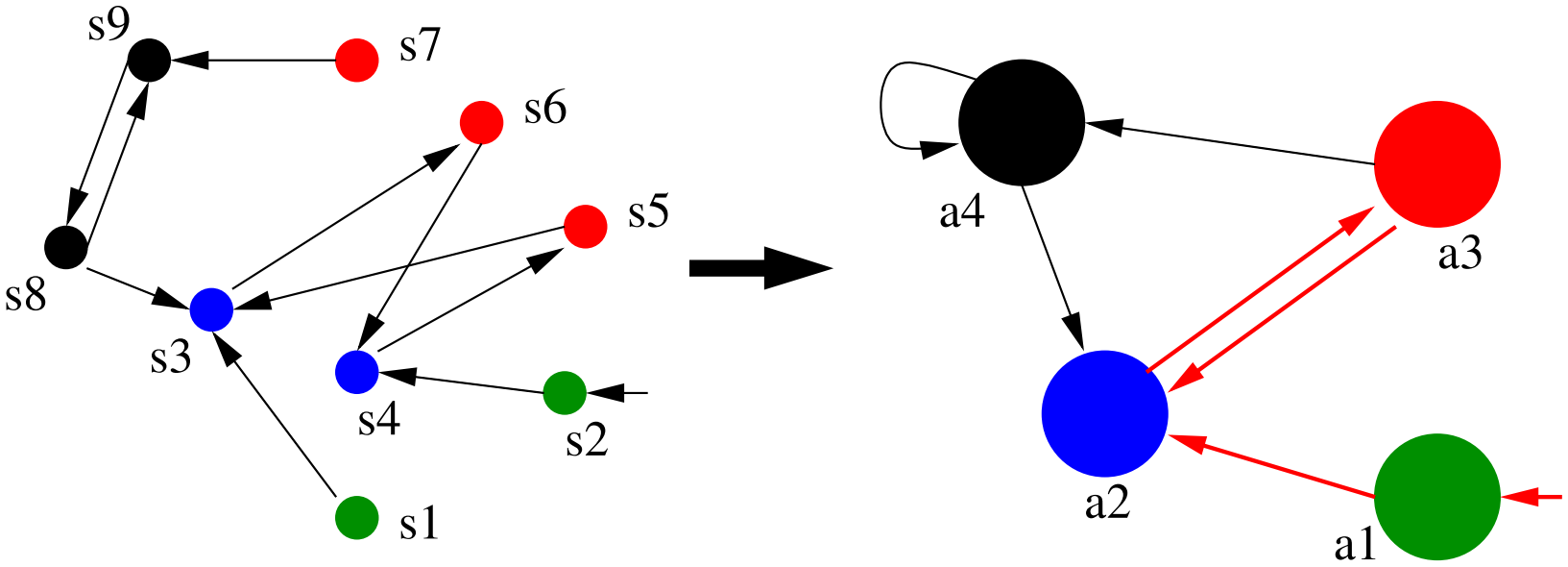
Mögliche neue Äquivalenzklassen: $\{s_5, s_6\}, \{s_7\}$ oder $\{s_5\}, \{s_6, s_7\}$.

Neuer Anlauf: $\mathrm{G}\,\neg schwarz$ mit Verfeinerung



Neue Abstraktion liefert kein Gegenbeispiel mehr, also gilt $\mathrm{G}\,\neg schwarz$ auch im konkreten System.

Beispiel 2: F G *rot*



Die Abstraktion enthält ein Gegenbeispiel mit Anfang $a_1 a_2$ und Schleife $a_3 a_2$.

Simulation einer Schleife

Gegeben: $w_A = b_0 \cdots b_k$, $\quad w_S = c_1 \cdots c_\ell$

$w_A w_S$ wird wie gehabt simuliert, jedoch muss w_S ggfs. mehrfach durchlaufen werden:

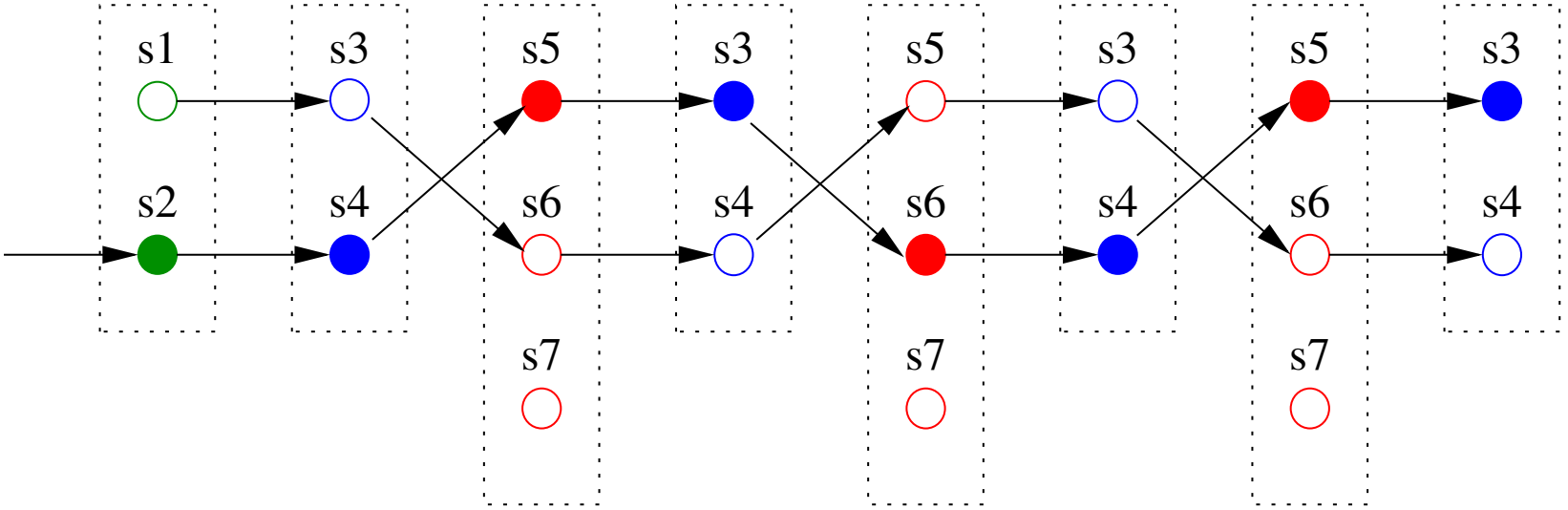
Es sei m die Größe der kleinsten Äquivalenzklasse in w:

$$m = \min_{i=1,\ldots,\ell} |c_i|$$

Simuliere den Pfad $w_A w_S^{m+1}$; dabei wird entweder eine (konkrete) Schleife entdeckt (dann handelt es sich um ein echtes Gegenbeispiel), oder die Simulation schlägt fehl.

Verfeinerung: analog zum Anfangsstück

Beispiel: $w_A = a_1 a_2$, $w_S = a_3 a_2$, $m = 2$



Die Simulation gelingt, wobei u.a. Zustand s_4 wiederholt wird, d.h. der Ablauf hat eine konkrete Entsprechung, und $\mathcal{K} \not\models \phi$.

Teil 13b: Praktische Aspekte von CEGAR

Überblick

CEGAR = counterexample-guided abstraction refinement

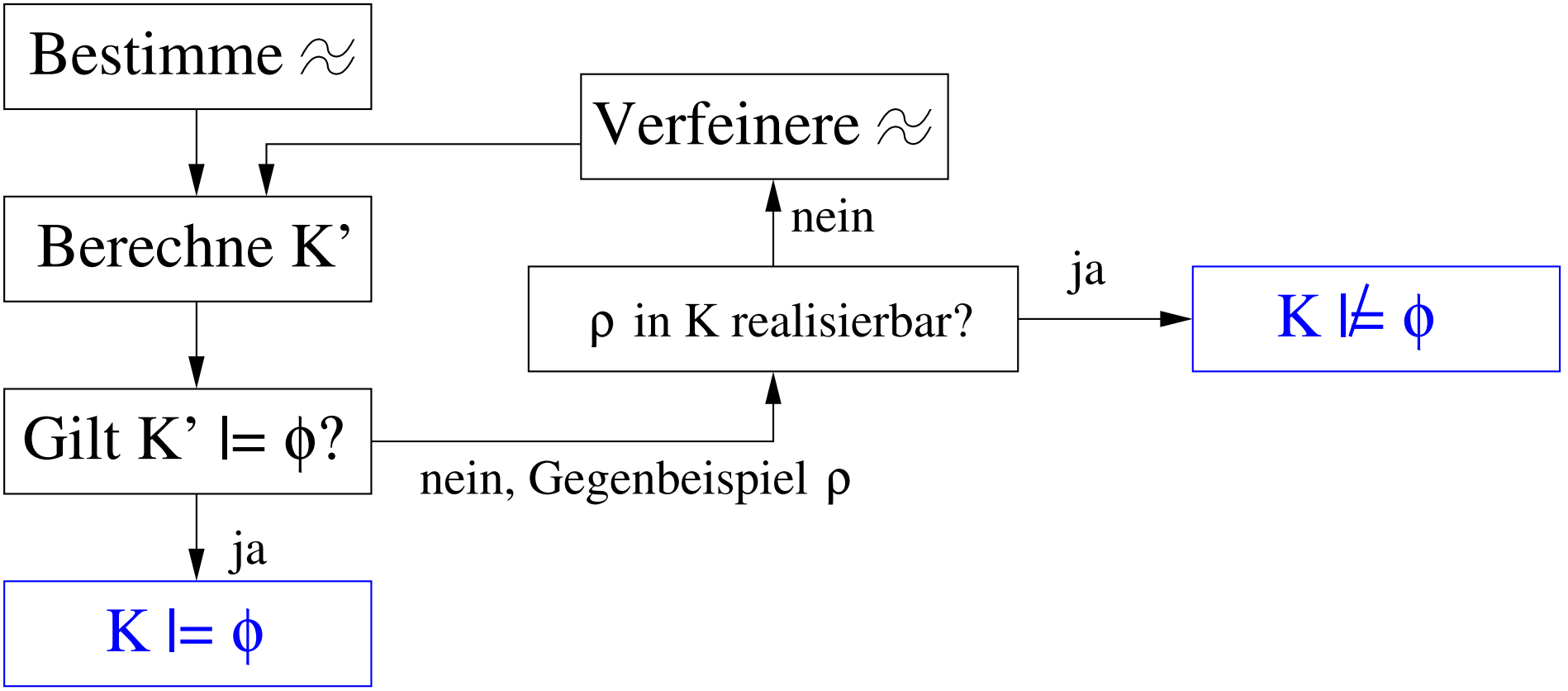
Bislang behandelt: Grundlagen von CEGAR

Heute: Aspekte praktischer Anwendung

- Konstruktion des abstrakten Modells
- Überprüfung von Gegenbeispielen
- Wahl der Prädikate (Verfeinerung)

Der Abstraktions-Verfeinerungs-Zyklus

Eingabe: $\mathcal{K}, \phi$



Konstruktion der Abstraktion

Problem: Abstraktion soll erstellt werden, ohne dass das konkrete System voll konstruiert wird (zu groß oder gar unendlich)

Hier: Beispiel für *endliches* konkretes System mit BDDs

$\mathcal{K} = (S, \rightarrow, r, AP, \nu)$: konkretes System, $\rightarrow \subseteq S \times S$ gegeben als BDD R mit Variablen $\vec{x}, \vec{y}$

Äquivalenzklassen gegeben durch BDDs $V_1, \ldots, V_m$ (über $\vec{x}$)

Führe je m neue BDD-Variablen $\vec{v}$, $\vec{w}$ ein.

Berechne $R' := \exists \vec{x}, \vec{y} : R \wedge \bigwedge_{i=1}^{m} \left(v_i \leftrightarrow V_i \right) \wedge \bigwedge_{i=1}^{m} \left(w_i \leftrightarrow V_i[\vec{x}/\vec{y}] \right)$

Dann ist R' die abstrakte Transitionsrelation.

Prüfung des Gegenbeispiels: geeignete Operationen zur Berechnung von $S_i = \{ t \mid t \in b_i \wedge \exists s \in S_{i-1} : s \rightarrow t \}$.

Verfeinerung: Falls $S_{\ell+1} = \emptyset$, wähle neues Prädikat S_ℓ.

CEGAR: Prinzip erstmals eingeführt 1994

Seit ca. 2000: Software-Model-Checker mit CEGAR

- SLAM / SDV (Microsoft Research)
- Blast (Standford / EPFL)
- Magic (CMU)

“kanonische Anwendung:” Gerätetreiber (viel Kontrolle, wenig Datenabhängigkeiten)

aktuelles Forschungsgebiet!

Typische Eigenschaften existierender CEGAR-Systeme:

Eingabe: C-Programm

Eigenschaft: Erreichbarkeit oder Sicherheitseigenschaft

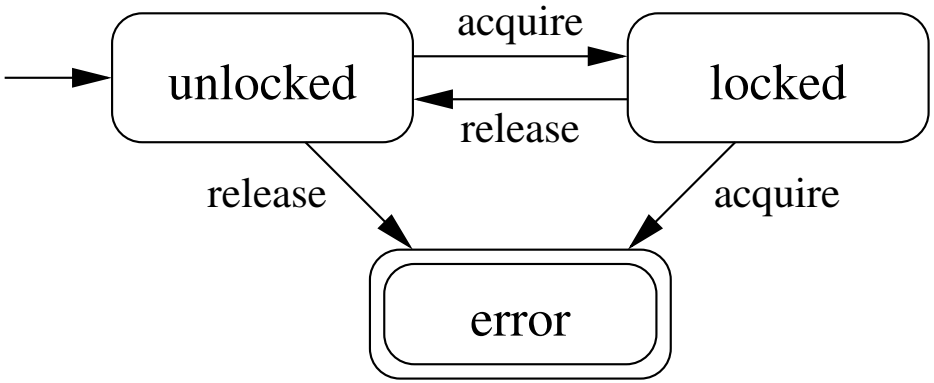
Abstraktion: Prädikate, boolsche Programme

Arbeitsweise von SLAM / SDV

Entwickelt von Ball, Rajamani et al, ab ca 2000

Eingabe: C-Programm (Windows-Gerätetreiber)

Sicherheitseigenschaft durch einen endlichen Automaten gegeben,
z.B. “Acquire/Release” müssen in richtiger Reihenfolge stehen:



Weitergehende Eigenschaften: Bis zu 30 Zuständen

Vor Model-Checking: Einbettung des Automaten in das C-Programm

Abstraktion in SLAM

Menge von Prädikaten *P* (booleschen Ausdrücken, z.B. x==y)

Äquivalenzklassen unterscheiden nach Kontrollpunkten und erfüllten Prädikaten

Implementierung: Übersetzung des C-Programms in ein “Boolesches Programm”, in dem alle Variablen vom Typ Boolean sind.

BDD-basierter Modelchecker, der auf Erreichbarkeit testet und ggfs. eine Ausführung liefert, die einen Fehlerzustand (in der Abstraktion) erreicht.

Erstellung der Abstraktion

Anfängliche Prädikate: Zustände des Sicherheitsautomaten

Jede Anweisung des C-Programms wird in eine Zuweisung an die booleschen Variablen umgewandelt.

Berechne die schwächste Vorbedingung (weakest precondition) für Gültigkeit bzw. Ungültigkeit jedes Prädikats.

Stelle fest, aus welchen Prädikaten die schwächste Vorbedingung folgt.

Beispiel: Prädikat $p := (x = y)$, Anweisung $x = x + 1$

Schwächste Vorbedingung: $x + 1 = y$

Aus p folgt $\neg(x + 1 = y)$.

Weder aus p noch aus $\neg p$ folgt $x + 1 = y$.

$\Rightarrow$ Übersetze die Anweisung in:

```
if p then p:=false else p:=unbekannt
```

Einsatz von Theorembeweisern, um die richtigen Vorbedingungen zu finden (für arithmetische Aussagen, Zeiger, . . .)

Genaue Bedingungen nicht immer feststellbar!
(Der Theorembeweiser könnte “weiß nicht” antworten.)

Viele Aufrufe des Theorembeweisers, Erstellung der Abstraktion wird zum Flaschenhals.

Überprüfung eines Gegenbeispiels

Ein Gegenbeispiel besteht aus einer Folge von Anweisungen. Beispiel:

```
x = c;

d = c+1;

y = d;

if (y != x+1) error;
```

Offensichtlich kann diese Sequenz von Anweisungen nicht zum Fehler führen.

Übersetze die Anweisungen in eine Formel $F = F_1 \wedge F_2 \wedge F_3 \wedge F_4$:

$$F_1 = x_1 = c_0 \wedge y_1 = y_0 \wedge d_1 = d_0 \wedge c_1 = c_0$$

$$F_2 = x_2 = x_1 \wedge y_2 = y_1 \wedge d_2 = c_1 + 1 \wedge c_2 = c_1$$

$$F_3 = x_3 = x_2 \wedge y_3 = d_2 \wedge d_3 = d_2 \wedge c_3 = c_2$$

$$F_4 = y_3 \neq x_3 + 1$$

Beobachtung: Das Beispiel kann zum Fehler führen gdw. F erfüllbar ist.

Aber: Überprüfung einer solch großen Formel aufwändig!

Interpolanten

Seien F, G zwei Formeln, so dass $F \wedge G$ unerfüllbar ist.

J heißt (Craig-)Interpolant zwischen F und G gdw.

(i) J folgt aus F;

(ii) $J \wedge G$ ist unerfüllbar;

(iii) in J kommen nur Grundaussagen vor, die in F *und* G vorkommen.

Seien Z die Variablen, die in F, aber nicht in G vorkommen.
Dann ist $\exists Z : F$ ein Craig-Interpolant zwischen F und G.

Seien Z die Variablen, die in G, aber nicht in F vorkommen.
Dann ist $\forall Z : \neg G$ ein Craig-Interpolant zwischen F und G.

Anwendung von Interpolation

Berechne die Interpolanten zwischen F_1 und $F_2 \wedge F_3 \wedge F_4$,
zwischen $F_1 \wedge F_2$ und $F_3 \wedge F_4$ usw.

Im Beispiel: (mit existenzieller Quantifizierung)

$$x_1 = c_1$$

$$d_2 = x_2 + 1$$

$$y_3 = x_3 + 1$$

Füge die gefundenen Interpolanten als Prädikate hinzu.

Optimierung: Reduziere die Anweisungssequenz zuvor auf diejenigen Anweisungen, die auf die Fehlerbedingung Einfluss haben (Slicing).

Sei J ein Interpolant zwischen (z.B.) F_1 und $F_2 \wedge F_3 \wedge F_4$.

J ist nur von den Variablen mit Index 1 abhängig, d.h. es ist ein Prädikat über den Zustand nach der ersten Anweisung.

J wird von F_1 impliziert, d.h. wir wissen, dass nach Ausführung der ersten Anweisung das Prädikat J gilt

$\Rightarrow$ In der neuen (booleschen) Abstraktion wird nach der ersten Anweisung J den Wert *true* haben.

$J \wedge G$ ist unerfüllbar, d.h. aus den von J repräsentierten Zuständen ist der Fehlerzustand nicht erreichbar.

$\Rightarrow$ Die neue Abstraktion wird nicht wieder denselben Fehlerpfad finden.

Fazit: Das Hinzufügen von J als Prädikat wird die Abstraktion verbessern.

Berechnung von Interpolanten

Genaue Berechnung hängt von der Logik ab, in der F und G ausgedrückt sind!

Drcken F, G den Effekt von Anweisungen aus, so gilt allgemein:

Die stärkste Nachbedingung von F bzgl. *true* ist ein Interpolant.

Die schwächste Vorbedingung von G bzgl. *false* ist ein Interpolant.

Im Folgenden betrachten wir den (einfachen) Fall, wo F und G Formeln der Aussagenlogik sind.

Interpolanten in der Aussagenlogik

Seien F, G Formeln der Aussagenlogik.

Bemerkungen:

Sind J und K Interpolanten zwischen F und G, so haben auch $J \wedge K$ und $J \vee K$ diese Eigenschaft.

Es gibt einen stärksten Interpolanten (d.h. einen Interpolanten J, so dass $J \rightarrow K$ für alle Interpolanten K gilt).

Es gibt einen schwächsten Interpolanten (d.h. einen Interpolanten J, so dass $K \rightarrow J$ für alle Interpolanten K gilt).

Beispiel

Gegeben sei die (unausführbare) Folge von Anweisungen

x:=true; y:=x; assume($\neg y \wedge z$);

Die entsprechenden Formeln:

$$F_1 = x_1 \wedge (y_1 \leftrightarrow y_0) \wedge (z_1 \leftrightarrow z_0)$$

$$F_2 = (x_2 \leftrightarrow x_1) \wedge (y_2 \leftrightarrow x_1) \wedge (z_2 \leftrightarrow z_1)$$

$$F_3 = (\neg y_2 \wedge z_2) \wedge (x_3 \leftrightarrow x_2) \wedge (y_3 \leftrightarrow y_2) \wedge (z_3 \leftrightarrow z_2)$$

Beispiel (2)

Stärkste Interpolanten: (Nachbedingungen)

zwischen F_1 und $F_2 \wedge F_3$: $J_1 := x_1$

zwischen $F_1 \wedge F_2$ und F_3: $J_2 := x_2 \wedge y_2$

Schwächste Interpolanten: (Vorbedingungen)

zwischen $F_1 \wedge F_2$ und F_3: $K_2 := y_2 \vee \neg z_2$

zwischen F_1 und $F_2 \wedge F_3$: $K_1 := x_1 \vee \neg z_1$

Beispiel (3)

Wir stellen fest, dass J_1, J_2 und K_1, K_2 Hoare-Annotationen bilden, die die Unausführbarkeit der Anweisungsfolge beweisen (unter Weglassung der Variablenindizes):

$\{\mathsf{true}\}$ x:=true; $\{x\}$ y:=x; $\{x \wedge y\}$ assume($\neg y \wedge z$); $\{\mathsf{false}\}$

$\{\mathsf{true}\}$ x:=true; $\{x \vee \neg z\}$ y:=x; $\{y \vee \neg z\}$ assume($\neg y \wedge z$); $\{\mathsf{false}\}$

Würde man eine Abstraktion mit J_1, J_2 bzw. K_1, K_2 als Prädikaten bilden, so wäre die Anweisungsfolge auch in der Abstraktion unausführbar.

Einfachere Prädikate

Aber: J_1, J_2 bzw. K_1, K_2 sind nicht die “einfachsten” Prädikate für diesen Zweck!

K_1, K_2 sind unnötig kompliziert, da es auf z gar nicht ankommt.

J_2 ist unnötig kompliziert, da es nach der zweiten Anweisung auf x gar nicht mehr ankommt.

Heuristik für einfachere Prädikate:

J_1 impliziert K_1 (dito für J_2, K_2)

Daher: $J_1 \wedge \neg K_1$ bzw. $J_2 \wedge \neg K_2$ unerfüllbar.

Interpoliere zwischen $J_1, \neg K_1$ und $J_2, \neg K_2$.

Interpolation zwischen J_1 und $\neg K_1$ liefert $L_1 := x$.

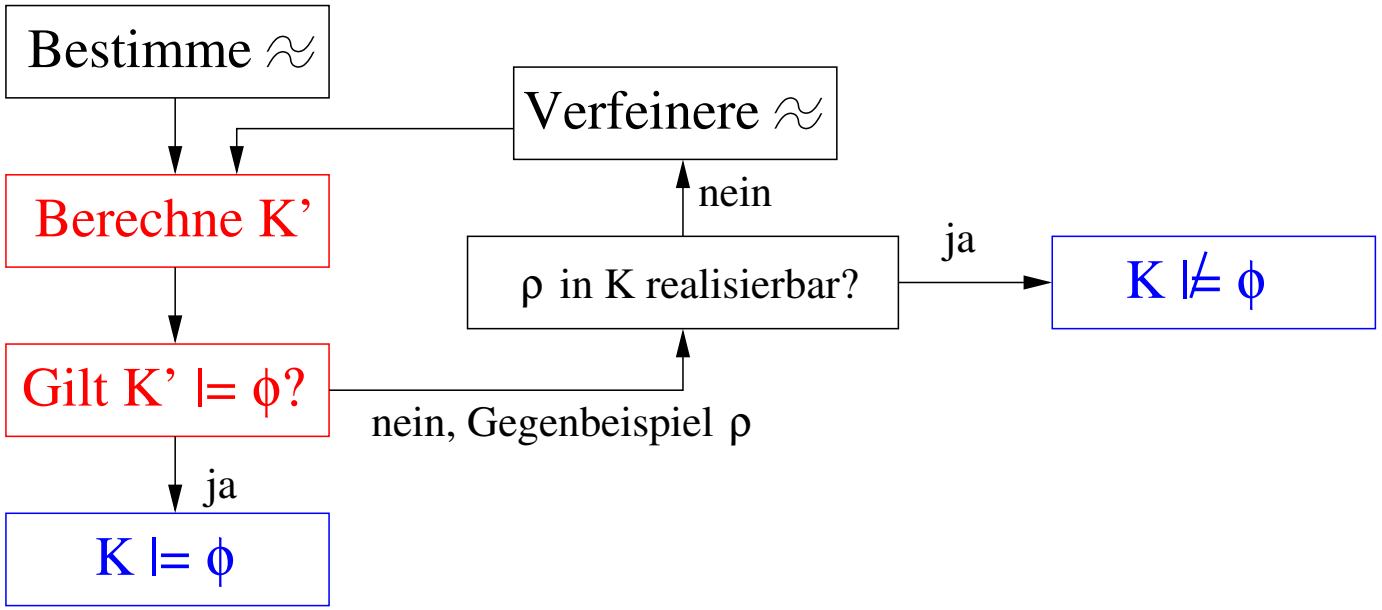
Interpolation zwischen J_2 und $\neg K_2$ liefert $L_2 := y$.

L_1, L_2 bilden eine einfachere Hoare-Annotation:

$\{\mathsf{true}\}$ x:=true; $\{x\}$ y:=x; $\{y\}$ assume($\neg y \wedge z$); $\{\mathsf{false}\}$

Lazy Abstraction

Abstraktions-Verfeinerungs-Zyklus:



Rot markierte Schritte sind teuer; oft wird dieselbe Arbeit mehrfach wiederholt.

Idee von Lazy Abstraction: [Henzinger et al 2002]

Wiederholte Arbeit vermeiden

Abstraktion nur in Gegenden verfeinern, die von falschen Gegenbeispielen betroffen sind.

Erreichbarkeitsprüfung ebenfalls nur in diesen Gegenden wiederholen.

ggfs. unterschiedliche Prädikate in verschiedenen Teilen eines Programms

Arbeitsweise von Lazy Abstraction

Erforschung des Programms durch Tiefensuche

Parameter der Tiefensuche: Programmzeile, Formel über derzeitige Prädikate

Speichere zu jeder erreichten Programmzeile die Disjunktion aller Prädikate

Tiefensuche abschneiden, wenn derzeitige Formel in bestehender Disjunktion “enthalten” ist

Entdeckter Fehler $\widehat{=}$ Pfad von der Wurzel zur einem Blatt im Tiefensuche-Baum

Verfeinerung bei Lazy Abstraction

Pfad von der Wurzel zu einem Blatt entspricht einer Anweisungsfolge

Berechne schwächste Vorbedingungen rückwärts vom Blatt aus.

Finde den am nächsten zum Blatt liegenden Knoten p, an dem die schwächste Vorbedingung nicht erreichbar ist.

Lösche den Baum unterhalb von p; füge neue Prädikate in p ein und starte die Tiefensuche dort neu.

Implementierung

Lazy Abstraction implementiert in BLAST
(Berkeley Lazy Abstraction Software Verification Tool)

URL: `http://mtc.epfl.ch/software-tools/blast/`

Eingabe: C-Programm mit Assertions

Ausgabe: “safe” (wenn keine Assertion verletzt wird)
“unsafe” (wenn eine Assertion verletzt sein könnte)

Teil 14: Bounded Model Checking

Motivation

Abstraktion arbeitet mit Überapproximation

Es werden mehr Abläufe betrachtet, als das System in Wirklichkeit hat.

Fragestellung: Eigenschaft erfüllt?

Antwort: Ja (falls tatsächlich erfüllt), nein (falls Eigenschaft verletzt sein könnte)

Antwort ist präzision auf der “sicheren” Seite

Ziel: Verifikation (Beweis der Korrektheit)

Komplementärer Ansatz: Unterapproximation

Betrachte weniger Verhaltensweisen als tatsächlich vorhanden

Antwort: nein (falls Fehler vorhanden), “weiß nicht” (falls keiner entdeckt werden konnte)

Ziel: Falsifikation (Aufspüren von Fehlern)

Komplementär zu Testing-Methoden, aber systematischer

Bounded Model Checking

Vertreter einer Unterapproximationstechnik: Bounded Model Checking (BMC)

Betrachte nur Abläufe der Länge k, für ein fixes k

Eingeführt durch Biere et al 1999

Übersichtsartikel: Biere at al, *Bounded Model Checking*. In: Advances in Computers, 2003

Übersicht

Gegeben: (kompakte Beschreibung einer) Kripke-Struktur $\mathcal{K}$, LTL-Formel ϕ, Schranke $k \geq 0$

Übersetzung von $\neg\phi$ und aller Pfade der Länge k in $\mathcal{K}$ in eine Formel F der AL

Wenn F erfüllbar, dann erfüllt $\mathcal{K}$ ϕ *nicht* bzgl. der modifizierten Semantik. (umgekehrter Schluss *nicht* zulässig!)

Einsatz eines SAT-Checkers

Pfade als AL-Formel

Im Folgenden sei $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ eine Kripke-Struktur, S endlich.
Annahme: $S = \{0,1\}^m$ für irgendein m (o.B.d.A., wie bei BDDs)
Weitere Annahme: S hat keine Deadlocks.

Beliebiger Zustand aus S kann durch einen Vektor $\vec{s}$ mit m Grundaussagen beschrieben werden.

Benutze Vektoren $\vec{s_0}, \ldots, \vec{s_k}$, um Pfade der Länge k zu beschreiben ($(k+1) \cdot m$ Grundaussagen).

Im Folgenden identifizieren wir $\vec{s_i}$ mit dem dadurch repräsentierten Zustand s_i.

Sei $I(s_0)$ wahr gdw. $s_0 = r$.

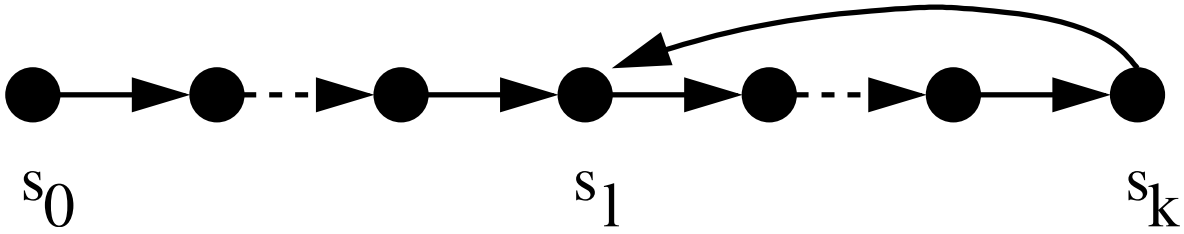
Sei $T(s_i, s_j)$ wahr gdw. $s_i \rightarrow s_j$ gilt.

Dann ist $F_P := I(s_0) \wedge \bigwedge_{i=1}^{k} T(s_{i-1}, s_i)$ wahr gdw. die Zustände einen Pfad der Länge k darstellen.

k-Schleifen

Sei $p = s_0, \ldots, s_k$ ein Pfad der Länge k.

Sei $0 \leq \ell \leq k$. Wir nennen p eine (k, ℓ)-Schleife, falls $s_k \rightarrow s_\ell$.



Wir nennen p eine Schleife, falls p eine (k, ℓ)-Schleife für irgendein ℓ ist.

Wir nennen p schleifenlos, falls p keine k-Schleife ist.

LTL-Semantik auf Schleifen

Sei p eine (k, ℓ)-Schleife, und sei ϕ eine LTL-Formel in NNF.

Durch unendliche Wiederholung der Schleife in p erhalten wir einen unendlichen Pfad $\pi = s_0 \dots s_{\ell-1}(s_\ell \dots s_k)^\omega$.

Wir schreiben: $p \models^k \phi$ gdw. $\pi \models \phi$.

Sei $\sigma(i) := i + 1$ für $i < k$ und $\sigma(k) := \ell$.

Seien $0 \leq i, j \leq k$. Wir schreiben $i \preceq_\ell j$ gdw. $i \leq j$ oder $i, j \geq \ell$.
(Bedeutung: s_j tritt in π nach s_i auf.)

Sei $i \leq k$. Beobachtung:

$$\pi^i \models \phi \quad \Leftrightarrow \quad \pi^{i+j\cdot(k-\ell+1)} \models \phi \qquad \text{für jedes } j \geq 0 \text{ und } i \geq \ell$$

$$\pi^i \models a \quad \Leftrightarrow \quad a \in \nu(s_i) \qquad \text{für } a \in AP$$

$$\pi^i \models \neg a \quad \Leftrightarrow \quad a \notin \nu(s_i) \qquad \text{für } a \in AP$$

$$\pi^i \models \phi \vee \psi \quad \Leftrightarrow \quad \pi^i \models \phi \;\vee\; \pi^i \models \psi$$

$$\pi^i \models \phi \wedge \psi \quad \Leftrightarrow \quad \pi^i \models \phi \;\wedge\; \pi^i \models \psi$$

$$\pi^i \models \mathbf{X}\,\phi \quad \Leftrightarrow \quad \pi^{\sigma(i)} \models \phi$$

$$\pi^i \models \phi\,\mathbf{U}\,\psi \quad \Leftrightarrow \quad \pi^i \models \psi \;\vee\; (\pi^i \models \phi \;\wedge\; \pi^{\sigma(i)} \models \phi\,\mathbf{U}\,\psi)$$

$$\pi^i \models \phi\,\mathbf{R}\,\psi \quad \Leftrightarrow \quad \pi^i \models \phi \wedge \psi \;\vee\; (\pi^i \models \psi \;\wedge\; \pi^{\sigma(i)} \models \phi\,\mathbf{R}\,\psi)$$

Fazit: $\pi \models \phi$ (und damit $p \models^k \phi$) hängt nur von p ab.

LTL-Semantik auf endlichen Pfaden

Sei p ein (möglicherweise schleifenloser) Pfad der Länge k und ϕ eine LTL-Formel in NNF.

Da p keine Deadlocks hat, kann p zu mindestens einem unendlichen Pfad π verlängert werden.

Wir wollen $p \models^k \phi$ so definieren, dass aus $p \models^k \phi$ folgt: $\pi \models \phi$ für unendliche Verlängerungen π von p.

Folgende Definition erreicht diesen Zweck (mit $i \leq k$):

$$
\begin{array}{lcll}
p^{k+1} \not\models^k \phi & & \text{für jede Formel } \phi & \\
p^i \models^k a & \Leftrightarrow & a \in \nu(s_i) & \text{für } a \in AP \\
p^i \models^k \neg a & \Leftrightarrow & a \notin \nu(s_i) & \text{für } a \in AP \\
p^i \models^k \phi \vee \psi & \Leftrightarrow & p^i \models^k \phi \ \vee \ p^i \models^k \psi & \\
p^i \models^k \phi \wedge \psi & \Leftrightarrow & p^i \models^k \phi \ \wedge \ p^i \models^k \psi & \\
p^i \models^k \mathbf{X}\,\phi & \Leftrightarrow & p^{i+1} \models^k \phi & \\
p^i \models^k \phi\,\mathbf{U}\,\psi & \Leftrightarrow & p^i \models^k \psi \ \vee \ (p^i \models^k \phi \ \wedge \ p^{i+1} \models^k \phi\,\mathbf{U}\,\psi & \\
p^i \models^k \phi\,\mathbf{R}\,\psi & \Leftrightarrow & p^i \models^k \phi \wedge \psi \ \vee \ (p^i \models^k \psi \ \wedge \ p^{i+1} \models^k \phi\,\mathbf{R}\,\psi) &
\end{array}
$$

Dabei bezeichne p^i die Sequenz $s_i \ldots s_k$.

Aussagen über BMC

Sei p ein Pfad der Länge k und ϕ eine LTL-Formel.

Es gilt: Wenn $p \models^k \neg\phi$, dann $\mathcal{K} \not\models \phi$.

Es gilt: Wenn $\mathcal{K} \not\models \phi$, dann gibt es $k \geq 0$ und p, so dass $p \models^k \neg\phi$.

Übersetzung nach AL

Wir wollen eine Formel F erstellen, so dass gilt:

F ist erfüllbar gdw. es p gibt mit $p \models^k \neg\phi$.

Beobachtung: Sei p gegeben. Um festzustellen, ob $p \models^k \neg\phi$ gilt, müssen wir nur die Wahrheitswerte der Unterformeln von ϕ an endlich vielen Stellen testen.

Übersetzung von Schleifen

Sei $\ell \leq k$ und p eine (k, ℓ)-Schleife, und sei π der zugehörige unendliche Pfad.
Sei ψ eine Teilformel der NNF von $\neg\phi$ und $i \leq k$.

Wir führen Grundaussagen der Form ${}_\ell[\![\psi]\!]^i_k$ ein.

${}_\ell[\![\psi]\!]^i_k$ soll wahr sein gdw. $\pi^i \models \psi$.

Erstelle Teilformeln unter Rückgriff auf vorige Beobachtungen, z.B.

$$
\begin{aligned}
{}_\ell[\![\psi_1 \vee \psi_2]\!]^i_k &\leftrightarrow {}_\ell[\![\psi_1]\!]^i_k \vee {}_\ell[\![\psi_2]\!]^i_k \\
{}_\ell[\![\mathbf{X}\,\psi]\!]^i_k &\leftrightarrow {}_\ell[\![\psi]\!]^{\sigma(i)}_k
\end{aligned}
$$

Am Ende: Alle diese Teilformeln durch Konjunktion verbinden.

Vorsicht: Diese Vorgehensweise ist nicht ausreichend für $\mathbf{U}$:

$$\ell[\![\psi_1 \,\mathbf{U}\, \psi_2]\!]_k^i \;\leftrightarrow\; \Big(\ell[\![\psi_2]\!]_k^i \vee (\ell[\![\psi_1]\!]_k^i \wedge \ell[\![\psi_1 \,\mathbf{U}\, \psi_2]\!]_k^{\sigma(i)})\Big) \wedge \bigvee_{i \preceq_\ell j} \ell[\![\psi_2]\!]_k^j$$

Ohne die letzte Disjunktion wäre es möglich, dass die rechte Seite der $\mathbf{U}$-Formel niemals erfüllt ist.

Analog führen wir Grundaussagen $[\![\psi]\!]_k^i$ ein für beliebige Pfade p.

Fertige Übersetzung

F sei definiert wie folgt:

$$F := I(s_0) \wedge \bigwedge_{i=1}^{k} T(s_{i-1}, s_i) \wedge \left([\![\neg\phi]\!]_k^0 \vee \bigvee_{\ell=0}^{k} (T(s_k, s_\ell) \wedge {}_\ell[\![\neg\phi]\!]_k^0) \right)$$

Wenn F erfüllbar ist, so gilt $\mathcal{K} \not\models \phi$.

Bemerkung: Größe von F ist polynomiell in $|\mathcal{K}|$, $|\phi|$ und k.

Erfüllbarkeitstest der AL

1. Möglichkeit: Generiere einen BDD für F, teste, ob der BDD nur aus dem $\mathbf{0}$-Knoten besteht.

2. Möglichkeit: Benutze einen SAT-Checker.

SAT-Checking

Findet *irgendeine* erfüllende Belegung, oft schnell.
(BDDs würden *alle* erfüllenden Belegungen konstruieren.)

Seit etwa 5 Jahren gewaltige Fortschritte bei SAT-Checking

- Hunderttausende Variablen und Klauseln noch gut behandelbar

- Beispiele für SAT-Solver: zChaff, MiniSAT, . . .

Vorteil: Speichereffizienz!

SAT und BMC

Vorteil für Software-Model-Checking: Keine (prinzipiellen) Probleme, alle Sprachfeatures zu behandeln.

Gewöhnliche Vorgehensweise: Benutzer gibt vor, wie oft while-Schleifen entfaltet werden sollen, Checker fängt mit kleinen Werten für k an und versucht einen Fehler zu finden oder so weit wie möglich zu kommen.

Sehr effizient für Werte von k bis ca. 80.

Bekannte Implementierung: CBMC (für C-Programme)

```
http://www.cprover.org/cbmc/
```

Zusammenfassung

Bisher behandelter Stoff

Model-Checking (Erreichbarkeit, LTL, CTL)

endliche (womöglich sehr große) Kripke-Strukturen

Optimierungen

- Reduktion: System verkleinern unter Erhaltung aller Eigenschaften
- Kompression: kompakte Darstellung großer Zustandsmengen
- Abstraktion: Information wegwerfen

Approximation:

- Überapproximation (existenzielle Abstraktion) $\longrightarrow$ Verifikation
- Unterapproximation (BMC) $\longrightarrow$ Falsifikation

Approximation auch/insbesondere anwendbar auf Software bzw. unendliche Zustandsräume

Unendliche Zustandsräume

Einige Gründe für unendliche Zustandsräume:

- Daten: ganze Zahlen, Listen, Bäume, Zeigerstrukturen, ...
- Kontrolle: Prozeduren, dynamische Prozesserzeugung, ...
- Asynchrone Kommunikation: unbeschränkte FIFO-Kanäle
- Unbekannte Parameter: Zahl von Protokoll-Teilnehmern, ...
- Echtzeit: diskrete oder stetige Zeit

Viele (nicht alle!) dieser Merkmale (bzw. Kombinationen davon) ergeben Turing-mächtige Berechnungsmodelle.

Teil 15: Kellerautomaten

Beispiel 1

Ein kleines Programm (mit Parameter $n \geq 1$):

```
bool g=true;
void main() {
   level_1();
   level_1();
   assume(g);
}
void level_n() {
   g:=not g;
}
```

```
void level_i() {
   for j:=1 to 8 do skip;
   level_{i+1}();
   level_{i+1}();
}
```

Frage: Ist g am Ende des Programms true?

Das vorige Beispiel hat *endlich* viele Zustände.
(Der Stack der Prozedur-Aufrufe hat Länge $\mathcal{O}(n)$.

Behandelbar durch “Inlining” (Prozeduraufruf ersetzen durch Kopie der Prozedur).

Inlining sorgt für exponenzielle Vergrößerung des Programms.

Inlining ist ineffizient: Jede Kopie eine Prozedur wird gesondert untersucht.

Inlining nicht anwendbar bei rekursiven Aufrufen.

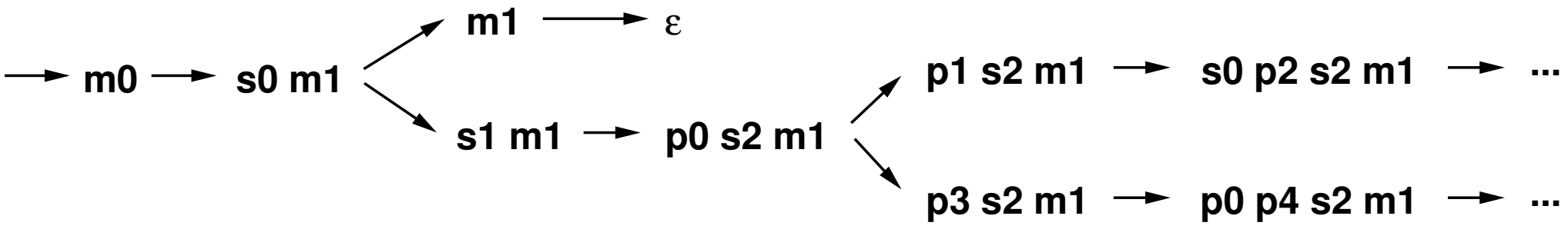
Beispiel 2: Rekursives Programm (Plotter)

procedure p;
p_0: **if** ? **then**
p_1: **call** s;
p_2: **if** ? **then call** p; **end if**;
else
p_3: **call** p;
end if
p_4: **return**

procedure s;
s_0: **if** ? **then return**; **end if**;
s_1: **call** p;
s_2: **return**;

procedure $main$;
m_0: **call** s;
m_1: **return**;

$S = \{p_0, \ldots, p_4, s_0, \ldots, s_2, m_0, m_1\}^*$, Anfangszustand m_0



Beispiel 2 hat unendlich viele Zustände.

Nicht durch Inlining behandelbar!

Nicht durch naives Auflisten der Zustände behandelbar.

Endliche Darstellung von unendlichen Zustandsmengen erforderlich.

Beispiel 3: Quicksort

```
void quicksort (int left, int right) {
   int lo,hi,piv;
   if (left >= right) return;
   piv = a[right]; lo = left; hi = right;
   while (lo <= hi) {
     if (a[hi]>piv) {
       hi = hi - 1;
     } else {
       swap a[lo],a[hi];
       lo = lo + 1;
     }
   }
   quicksort(left,hi);
   quicksort(lo,right);
}
```

Frage: Sortiert Beispiel 3 korrekt? Terminiert es immer?

Beispiel 3 kann man nicht ansehen, wie viele Zustände es hat:

endlich viele, falls es korrekt ist

womöglich *unendlich* viele, falls es einen Fehler hat

Terminierung nur durch direkte Behandlung unendlicher Mengen feststellbar.

Ein Modell für prozedurale Programme

Kontrollfluss:

- sequentielles Programm (keine parallelelen Threads)
- Prozeduren
- gegenseitige Aufrufe (womöglich rekursiv)

Daten:

- globale Variablen (Einschränkung: nur endlich viel Speicher)
- lokale Variablen in jeder Prozedur (eine Kopie pro Aufruf)

Kellerautomaten

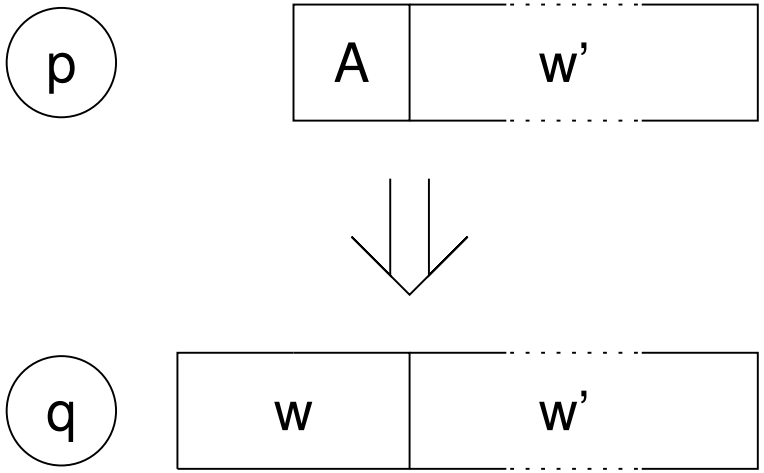
Ein Kellerautomat / Pushdown-System (PDS) ist ein Tripel (P, Γ, Δ), wobei

P eine endliche Menge von Kontrollzuständen ist;

Γ ein endliches Stackalphabet ist;

Δ eine endliche Menge von Regeln ist.

Regeln haben die Form $pA \hookrightarrow qw$ mit $p, q \in P,\ A \in \Gamma,\ w \in \Gamma^*$.



Wie Akzeptoren für kontextfreie Sprachen, nur ohne Eingabe!

Semantik eines PDS

Sei $\mathcal{P} = (P, \Gamma, \Delta)$ ein PDS und $c_0 \in P \times \Gamma^*$.

Das zu $\mathcal{P}$ gehörige Transitionssystem $\mathcal{T}_\mathcal{P} = (S, \rightarrow, r)$ ist wie folgt:

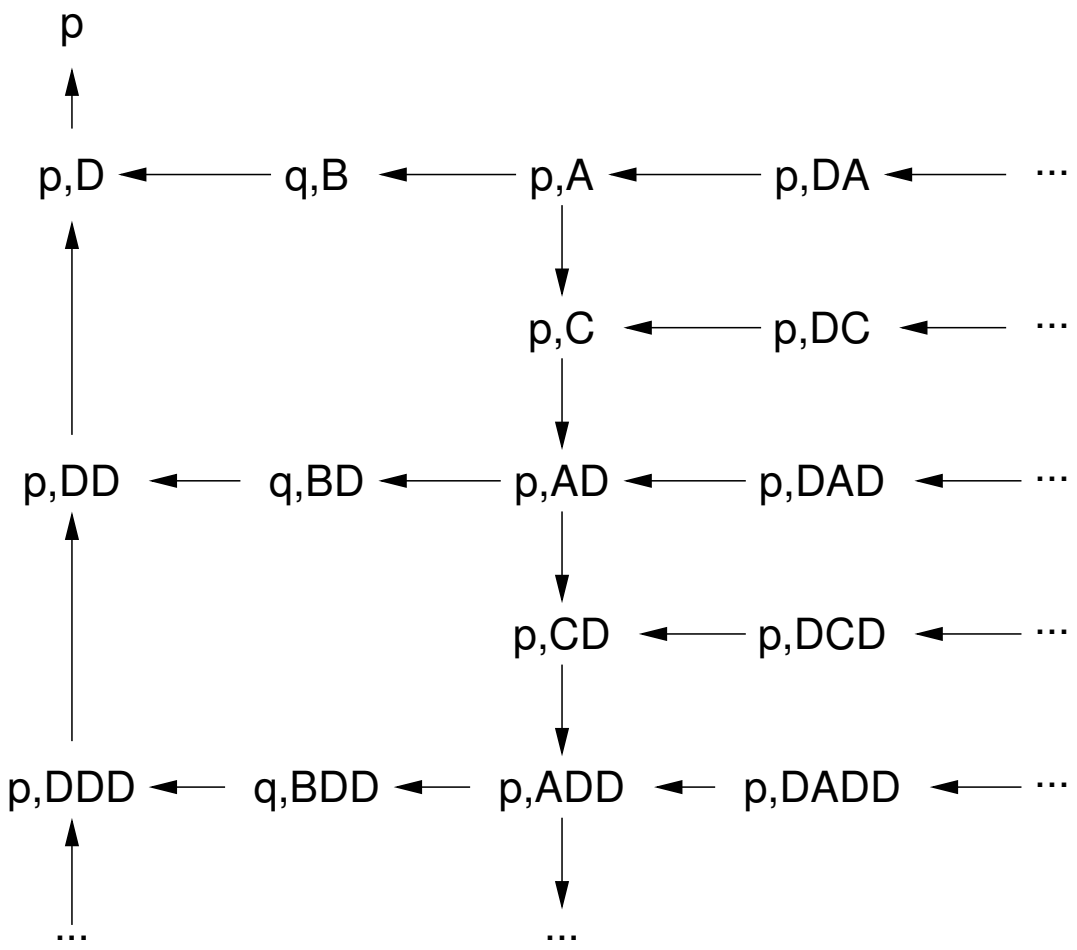
$S = P \times \Gamma^*$ sind die Zustände (Konfigurationen);

wir haben $pAw' \rightarrow qww'$ für alle $w' \in \Gamma^*$ gdw. $pA \hookrightarrow qw \in \Delta$;

$r = c_0$ ist die Anfangskonfiguration.

Transitionssystem eines PDS

$pA \hookrightarrow qB$
$pA \hookrightarrow pC$
$qB \hookrightarrow pD$
$pC \hookrightarrow pAD$
$pD \hookrightarrow p\varepsilon$



Korrespondenz Programme / PDS

P seien die Belegungen der globalen Variablen

Γ enthalte Tupel der Form $(\mathit{Programmzeiger}, \mathit{lokale\ Belegung})$

Interpretation einer Konfiguration pAw:

globale Werte in p, aktuelle Prozedur mit lokalen Variablen in A

“wartende” Prozeduren in w

Regeln:

$pA \hookrightarrow qB \mathrel{\widehat{=}}$ Anweisung innerhalb einer Prozedur

$pA \hookrightarrow qBC \mathrel{\widehat{=}}$ Prozeduraufruf

$pA \hookrightarrow q\varepsilon \mathrel{\widehat{=}}$ Rückkehr aus einer Prozedur

Erreichbarkeit in PDS

Sei $\mathcal{P}$ ein PDS und seien c, c' zwei Konfigurationen.

Frage: Gilt $c \to^* c'$ in $\mathcal{T}_\mathcal{P}$?

Merke: $\mathcal{T}_\mathcal{P}$ hat unendlich viele Zustände.

Dennoch ist die Frage entscheidbar!

Endliche Automaten

Um (unendliche) Mengen von Konfigurationen darzustellen, benutzen wir endliche Automaten.

Sei $\mathcal{P} = (P, \Gamma, \Delta)$ ein PDS. Wir nennen $\mathcal{A} = (\Gamma, Q, P, \delta, F)$ einen $\mathcal{P}$-Automaten.

Eingabealphabet von $\mathcal{A}$ ist das Stackalphabet Γ.

“Anfangszustände” von $\mathcal{A}$ sind die Kontrollzustände P.

$\mathcal{A}$ akzeptiert die Konfiguration pw, falls man bei p beginnend die Eingabe w lesen kann und an einem Endzustand anlangt.

Sei $\mathcal{L}(\mathcal{A})$ die Menge der von $\mathcal{A}$ akzeptierten Konfigurationen.

Eine Menge C (von Konfigurationen) heißt regulär gdw. sie von einem $\mathcal{P}$-Automaten akzeptiert wird.

Ein Automat heißt normal, falls keine Kanten in Zustände aus P führen.

Bemerkung: Im Folgenden unterscheiden wir die transitiven Erreichbarkeitsrelationen durch unterschiedliche Pfeile:

$pw \Rightarrow p'w'$ (im PDS $\mathcal{P}$) bzw. $p \xrightarrow{w} q$ (in $\mathcal{P}$-Automaten)

Erreichbarkeit in PDS

Mit $pre^*(C) = \{ c' \mid \exists c \in C : c' \Rightarrow c \}$ bezeichnen wir die Vorgänger von C, mit $post^*(C) = \{ c' \mid \exists c \in C : c \Rightarrow c' \}$ die Nachfolger.

Bekanntes Resultat (Büchi 1964):

> Sei C eine reguläre Menge und $\mathcal{A}$ ein (normaler) $\mathcal{P}$-Automat, der C akzeptiert.
>
> Ist C regulär, so auch $pre^*(C)$ und $post^*(C)$.
>
> Ist C regulär, so kann $\mathcal{A}$ in einen Automaten transformiert werden, der $pre^*(C)$ bzw. $post^*(C)$ akzeptiert.

Vorgehensweise

Sättigungsregel: Erweitere $\mathcal{A}$ wie folgt um Transitionen:

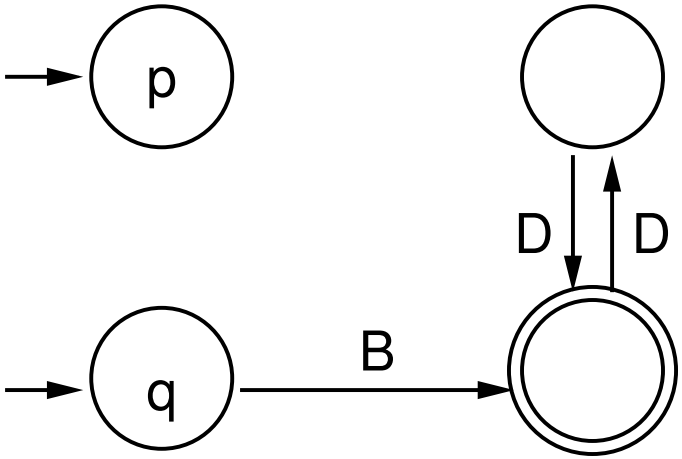
> Falls $q \xrightarrow{w} r$ im Automaten $\mathcal{A}$ geht und $pA \hookrightarrow qw$ eine Regel ist, füge die Transition (p, A, r) hinzu.

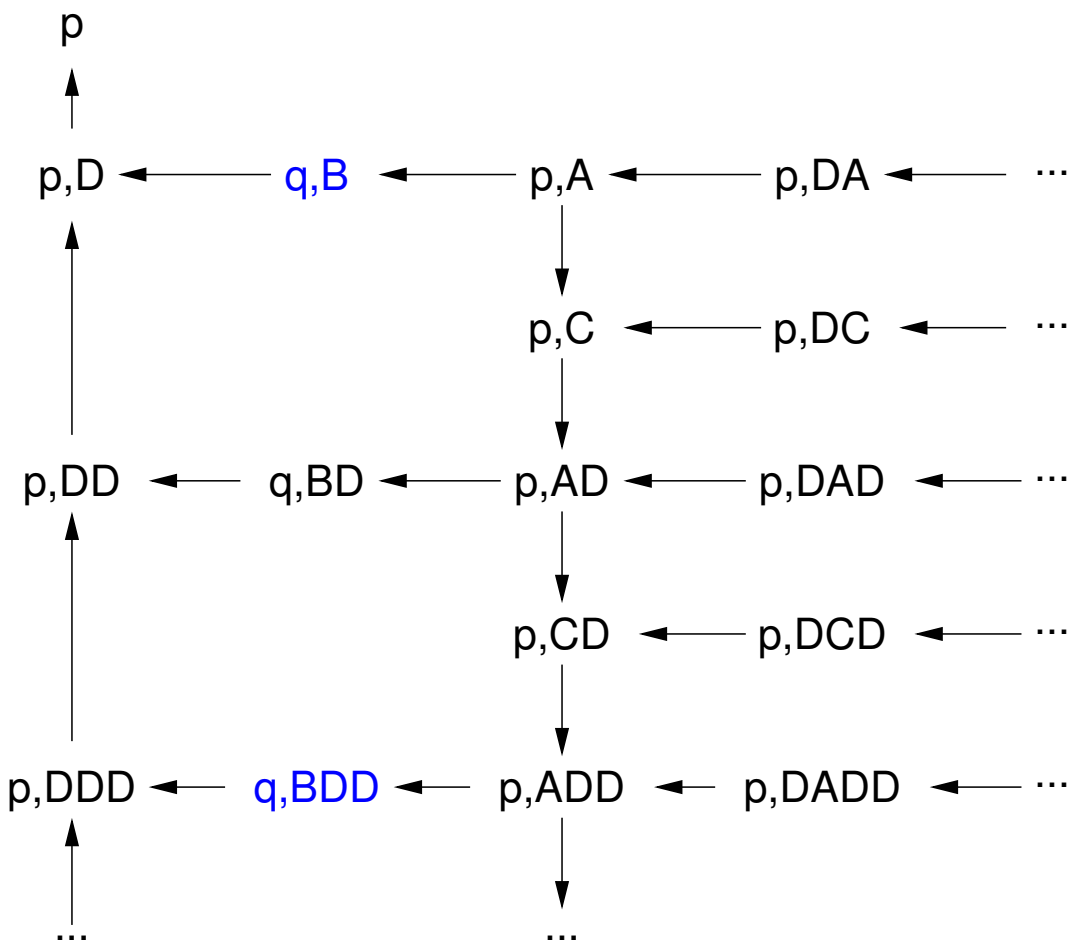
Wiederhole dies, bis keine Transition mehr hinzugefügt werden kann.

Am Ende akzeptiert der Automat $pre^*(C)$.

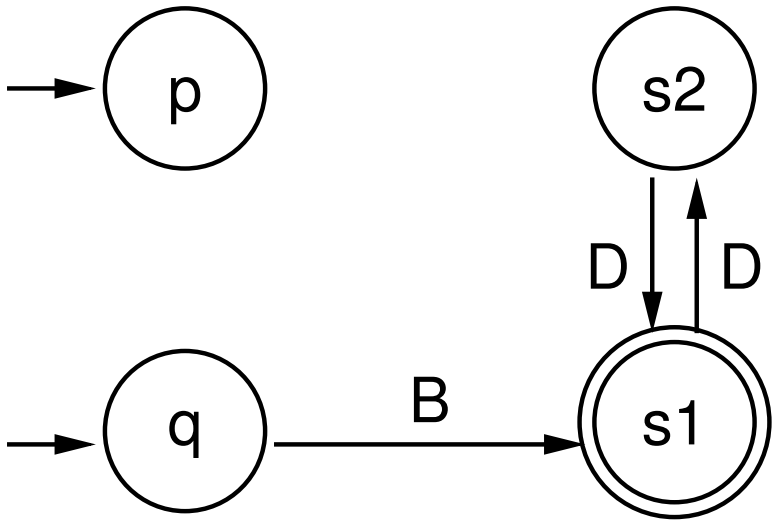
Für $post^*(C)$: Ähnlich.

Automat $\mathcal{A}$ für C



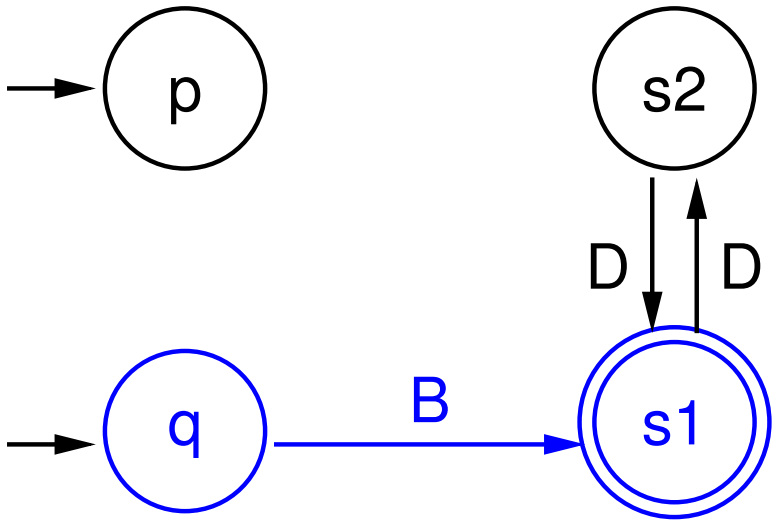


Erweitere $\mathcal{A}$



Erweitere $\mathcal{A}$

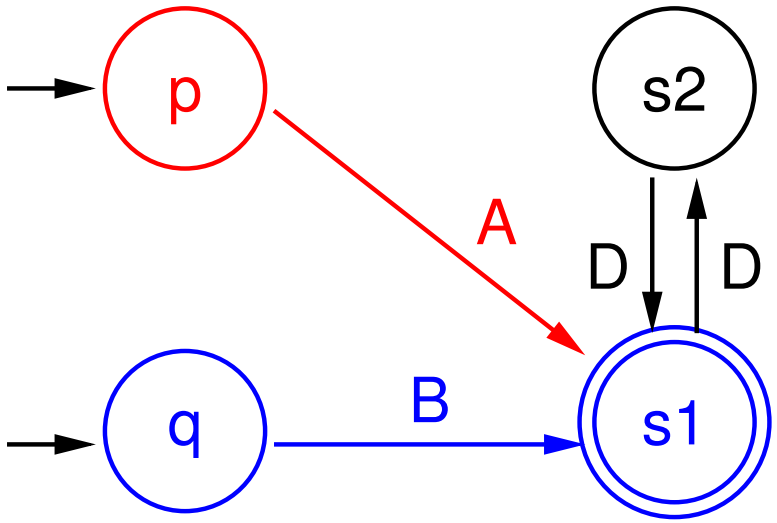
Wenn die rechte Seite einer Regel gelesen werden kann,



Regel: $pA \hookrightarrow qB$ Pfad: $q \xrightarrow{B} s_1$

Erweitere $\mathcal{A}$

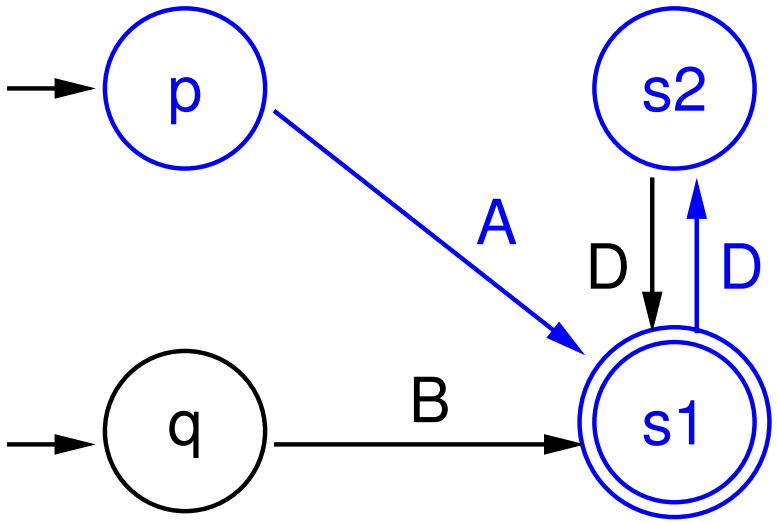
Wenn die rechte Seite einer Regel gelesen werden kann, ergänze die linke Seite.



Regel: $pA \hookrightarrow qB$ Pfad: $q \xrightarrow{B} s_1$ Neuer Pfad: $p \xrightarrow{A} s_1$

Erweitere $\mathcal{A}$

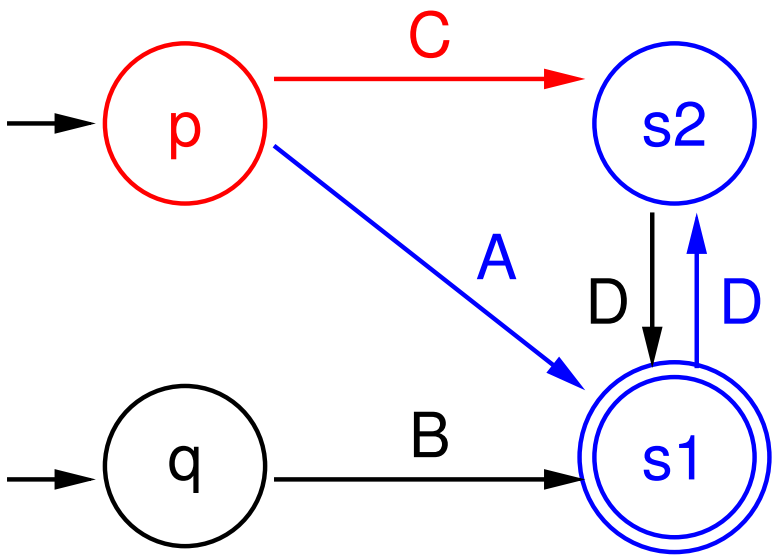
Wenn die rechte Seite einer Regel gelesen werden kann,



Regel: $pC \hookrightarrow pAD$ Pfad: $p \xrightarrow{A} s_1 \xrightarrow{D} s_2$

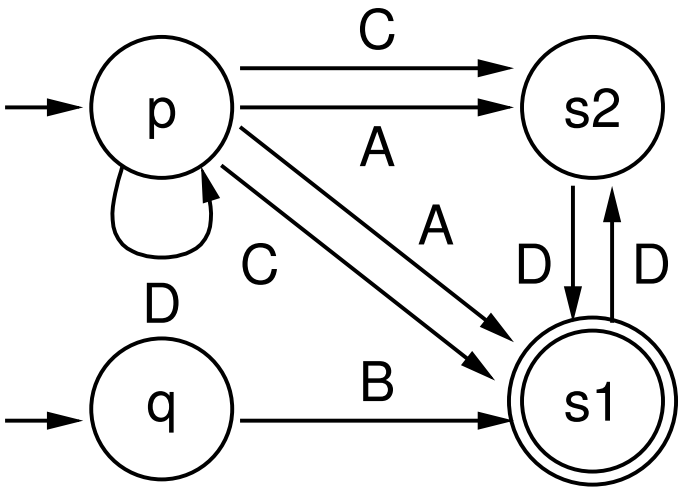
Erweitere $\mathcal{A}$

Wenn die rechte Seite einer Regel gelesen werden kann, ergänze die linke Seite.



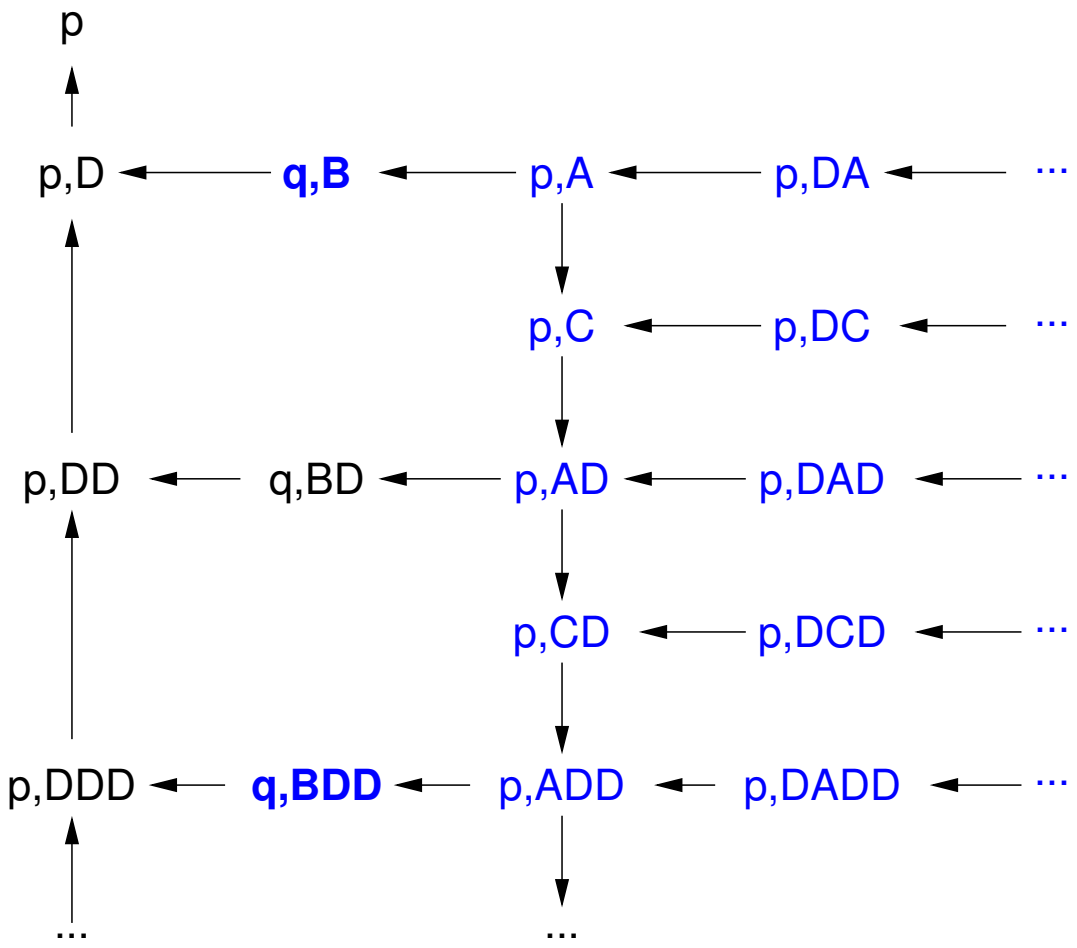
Regel: $pC \hookrightarrow pAD$ Pfad: $p \xrightarrow{A} s_1 \xrightarrow{D} s_2$ Neuer Pfad: $p \xrightarrow{C} s_2$

Endergebnis



Aufwand:

$\mathcal{O}(|Q|^2 \cdot |\Delta|)$ Zeit.



Beweis der Korrektheit

Wir zeigen (für pre^*):

> Sei $\mathcal{B}$ der Automat, der aus $\mathcal{A}$ durch die Sättigungsregel entsteht. Dann ist $\mathcal{L}(\mathcal{B}) = pre^*(C)$.

1. Teil: Terminierung

> Die Anwendung der Sättigungsregel terminiert, da man nur endlich viele Transitionen hinzufügen kann.

2. Teil: $pre^*(C) \subseteq \mathcal{L}(\mathcal{B})$

> Sei $c \in pre^*(C)$ und $c' \in C$, so dass c' von c in k Schritten erreichbar ist. Beweis durch Induktion über k (einfach).

3. Teil: $\mathcal{L}(\mathcal{B}) \subseteq pre^*(C)$

Sei $\underset{i}{\rightarrow}$ die Transitionsrelation des Automaten nach i-facher Anwendung der Sättigungsregel.

Wir zeigen allgemeiner: Wenn $p \xrightarrow[i]{w} q$ gilt, dann gibt es $p'w'$ mit $p' \xrightarrow[0]{w'} q$ und $pw \Rightarrow p'w'$; falls $q \in P$, so gilt zusätzlich $w' = \varepsilon$.

Beweis durch Induktion über i: (IA mit $i = 0$ trivial)

Induktionsschritt: Sei $t = (p_1, A, q')$ die i-te hinzugefügte Transition und j die Anzahl der Male, die t im Pfad $p \xrightarrow[i]{w} q$ vorkommt.

Induktion über j: Für $j = 0$ trivial. Sei also $j > 0$.

Es gibt p_2, p', u, v, w', w_2 mit folgenden Eigenschaften:

(1) $p \xrightarrow[i-1]{u} p_1 \xrightarrow[i]{A} q' \xrightarrow[i]{v} q$ (Aufteilung des Pfads $p \xrightarrow[i]{w} q$)

(2) $p_1 A \hookrightarrow p_2 w_2$ (Vorraussetzung für Sättigungsregel)

(3) $p_2 \xrightarrow[i-1]{w_2} q'$ (Vorraussetzung für Sättigungsregel)

(4) $p_1 u \Rightarrow p_1 \varepsilon$ (Ind.-Annahme auf i)

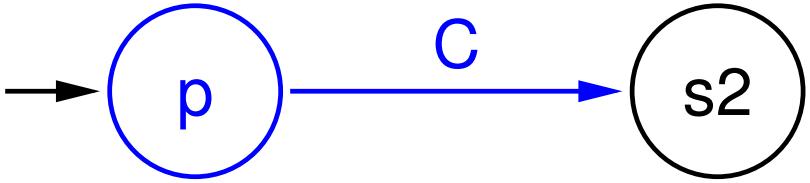
(5) $p_2 w_2 v \Rightarrow p' w'$ (Ind.-Annahme auf j)

(6) $p' \xrightarrow[0]{w'} q$ (Ind.-Annahme auf j)

Die gewünschte Aussage folgt aus (1), (4), (2) und (5).
Falls $q \in P$ ist, so folgt die Aussage aus (6) und der Tatsache, dass $\mathcal{A}$ normal ist.

Beispiel: $post^*$ (ohne Beweis)

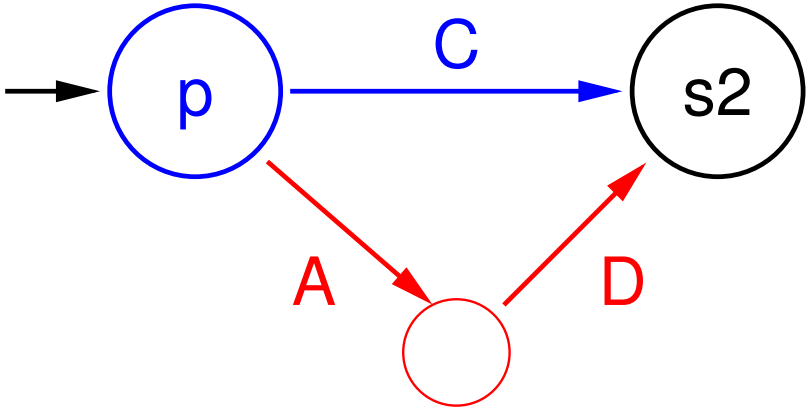
Wenn die *linke* Seite einer Regel gelesen werden kann, .



Regel: $pC \hookrightarrow pAD$ Pfad: $p \xrightarrow{C} s_2$

Beispiel: $post^*$ (ohne Beweis)

Wenn die *linke* Seite einer Regel gelesen werden kann, ergänze die *rechte* Seite.



Regel: $pC \hookrightarrow pAD$ Path: $p \xrightarrow{C} s_2$ Neuer Pfad: $p \xrightarrow{AD} s_2$

Gewichtete PDS

Regeln eines PDS mit Gewichten erweitert.

$$pA \overset{d}{\hookrightarrow} qBC$$

Gewichte können Entfernungen, Aktionen etc. ausdrücken (“was passiert während der Ausführung der Regel?”)

Gewichte werden einem Semiring entnommen (mit gewissen algebraischen Eigenschaften).

Gewichte sammeln sich entlang eines Pfads an, und wir berechnen eine “Zusammenfassung” des Gewichts aller Pfade, die von einer regulären Menge C ausgehen (verallgemeinertes Kürzeste-Wege-Problem).

Beispiel: Gewichtete PDS

$$pA \overset{a}{\hookrightarrow} qB$$
$$pA \overset{b}{\hookrightarrow} pC$$
$$qB \overset{c}{\hookrightarrow} pD$$
$$pC \overset{d}{\hookrightarrow} pAD$$
$$pD \overset{e}{\hookrightarrow} p\varepsilon$$

Beispiel: Gewichtete PDS

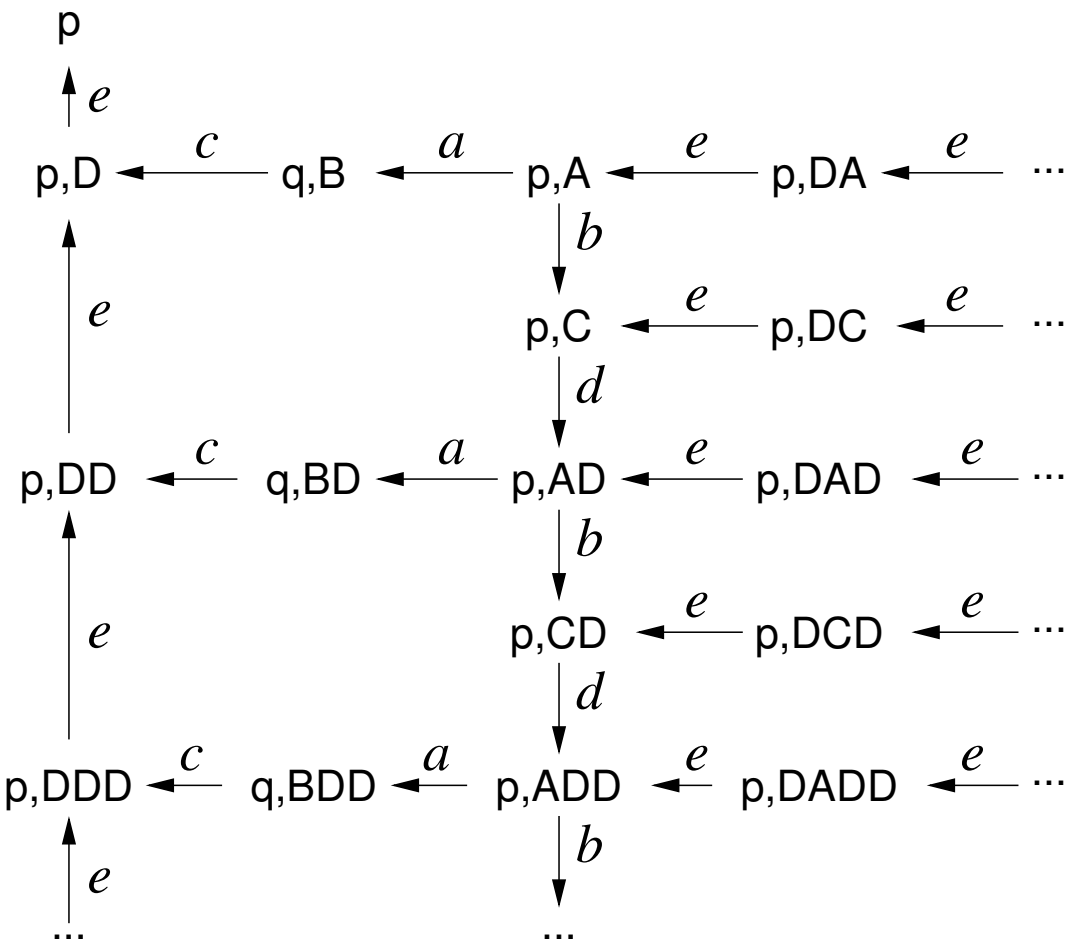
$pA \overset{a}{\hookrightarrow} qB$

$pA \overset{b}{\hookrightarrow} pC$

$qB \overset{c}{\hookrightarrow} pD$

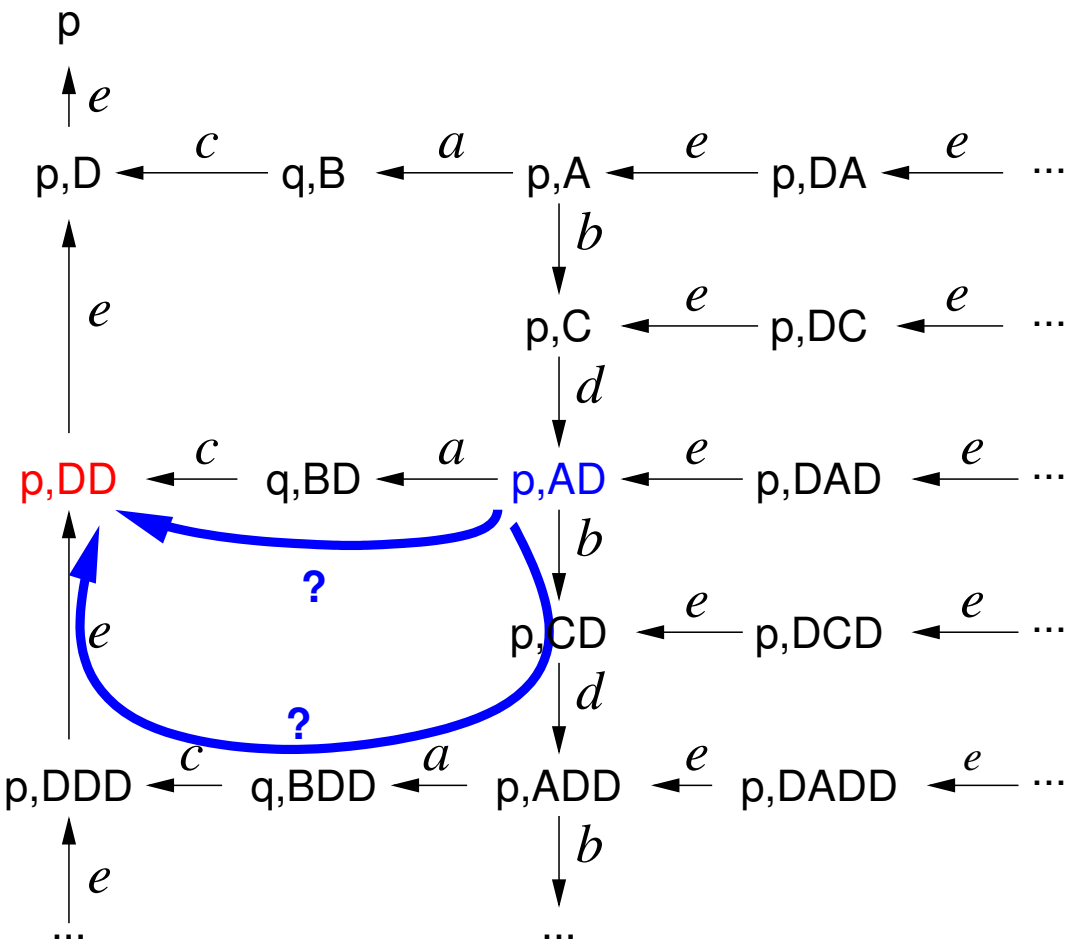
$pC \overset{d}{\hookrightarrow} pAD$

$pD \overset{e}{\hookrightarrow} p\varepsilon$



Beispiel: Gewichtete PDS

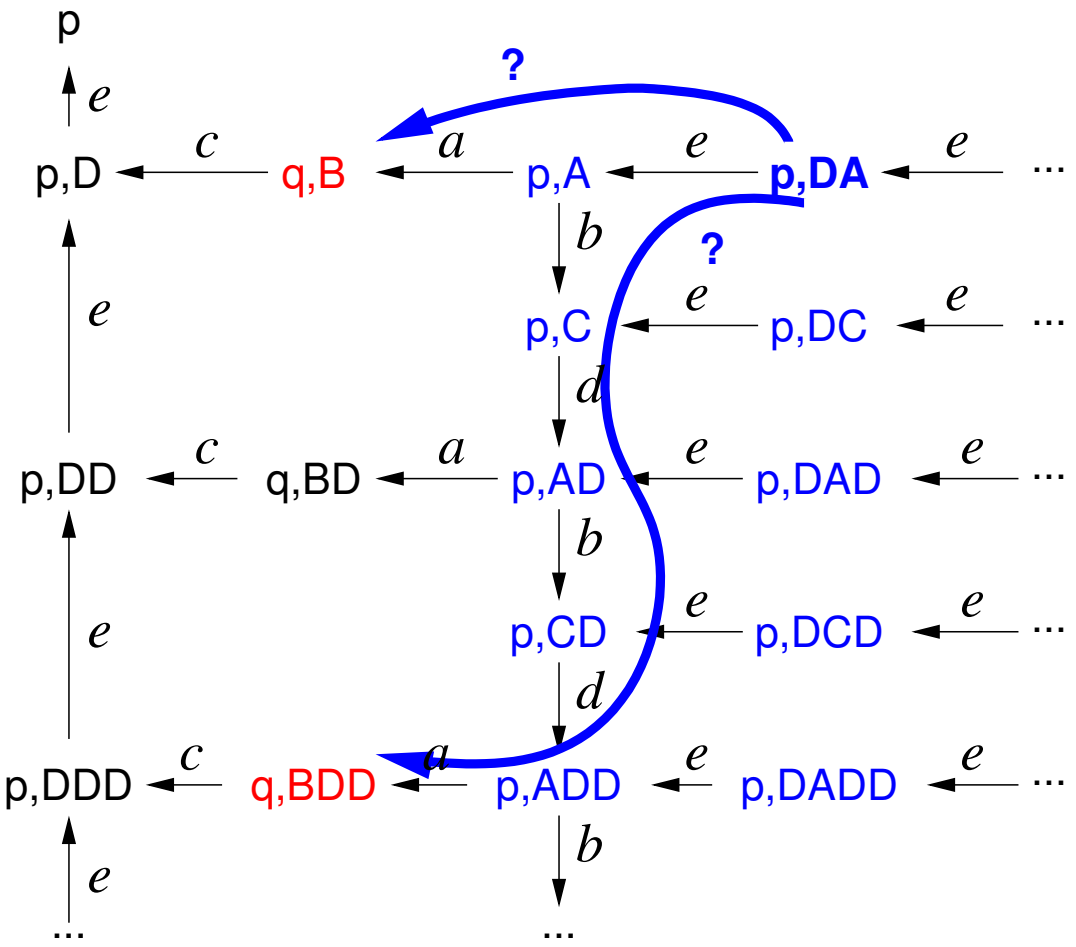
$pA \overset{a}{\hookrightarrow} qB$
$pA \overset{b}{\hookrightarrow} pC$
$qB \overset{c}{\hookrightarrow} pD$
$pC \overset{d}{\hookrightarrow} pAD$
$pD \overset{e}{\hookrightarrow} p\varepsilon$



Gewicht der Pfade, die von *pAD* nach *pDD* führen?

Beispiel: Gewichtete PDS

$pA \overset{a}{\hookrightarrow} qB$
$pA \overset{b}{\hookrightarrow} pC$
$qB \overset{c}{\hookrightarrow} pD$
$pC \overset{d}{\hookrightarrow} pAD$
$pD \overset{e}{\hookrightarrow} p\varepsilon$



Wert aller Pfade von c nach C, für jedes $c \in pre^*(C)$?

Semiring: Definition

$\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ ist ein beschränkter idempotenter Semiring gdw.:

$(D, \oplus)$ ist ein kommutatives Monoid mit neutralem Element $\bar{0}$
und $a \oplus a = a$ für alle $a \in D$.

$(D, \otimes)$ ist ein Monoid mit neutralem Element $\bar{1}$.

Distributivität, d.h. für alle $a, b, c \in D$:

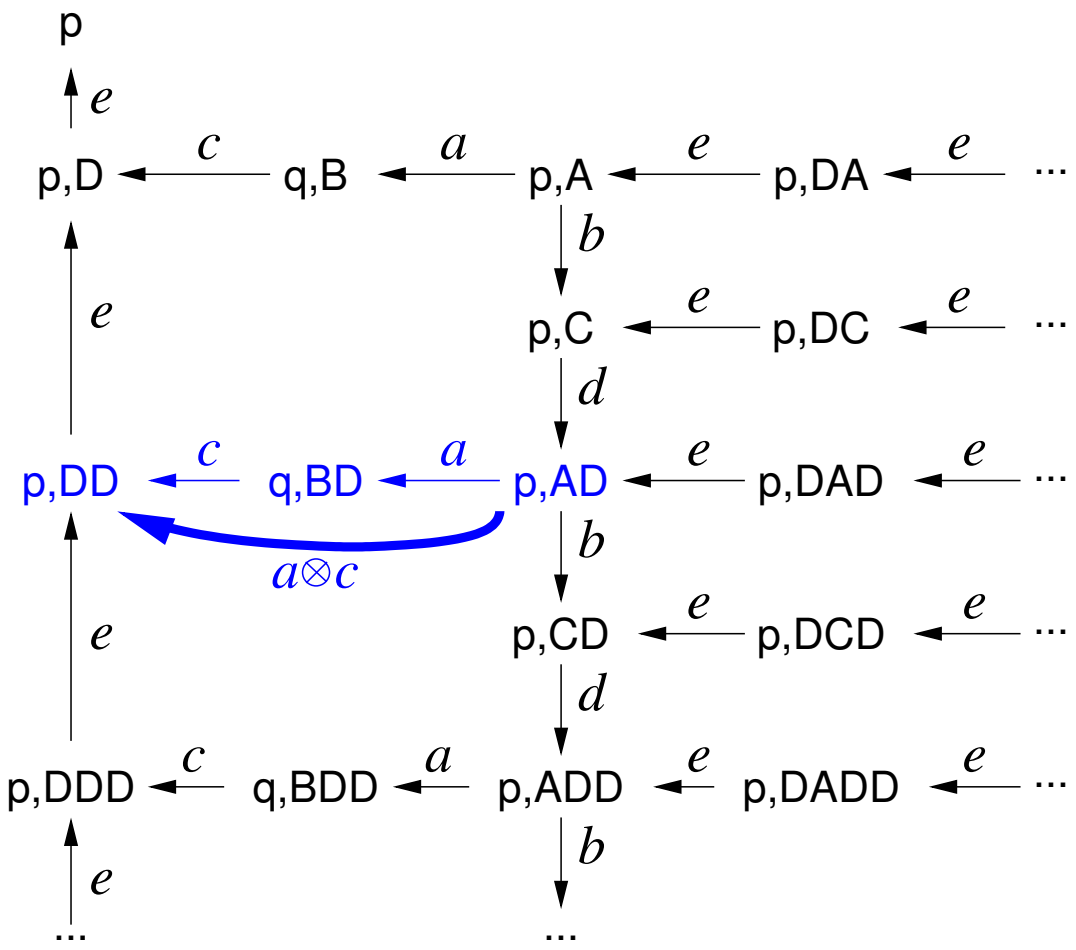
$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \text{ und}$$
$$(a \oplus b) \otimes c) = (a \otimes c) \oplus (b \otimes c)$$

"Annihilator:" $a \otimes \bar{0} = \bar{0} = \bar{0} \otimes a$ für alle $a \in D$

Keine unendlichen absteigenden Ketten in der Halbordnung, die durch $a \sqsubseteq b$ gdw. $a \oplus b = a$ gegeben ist.

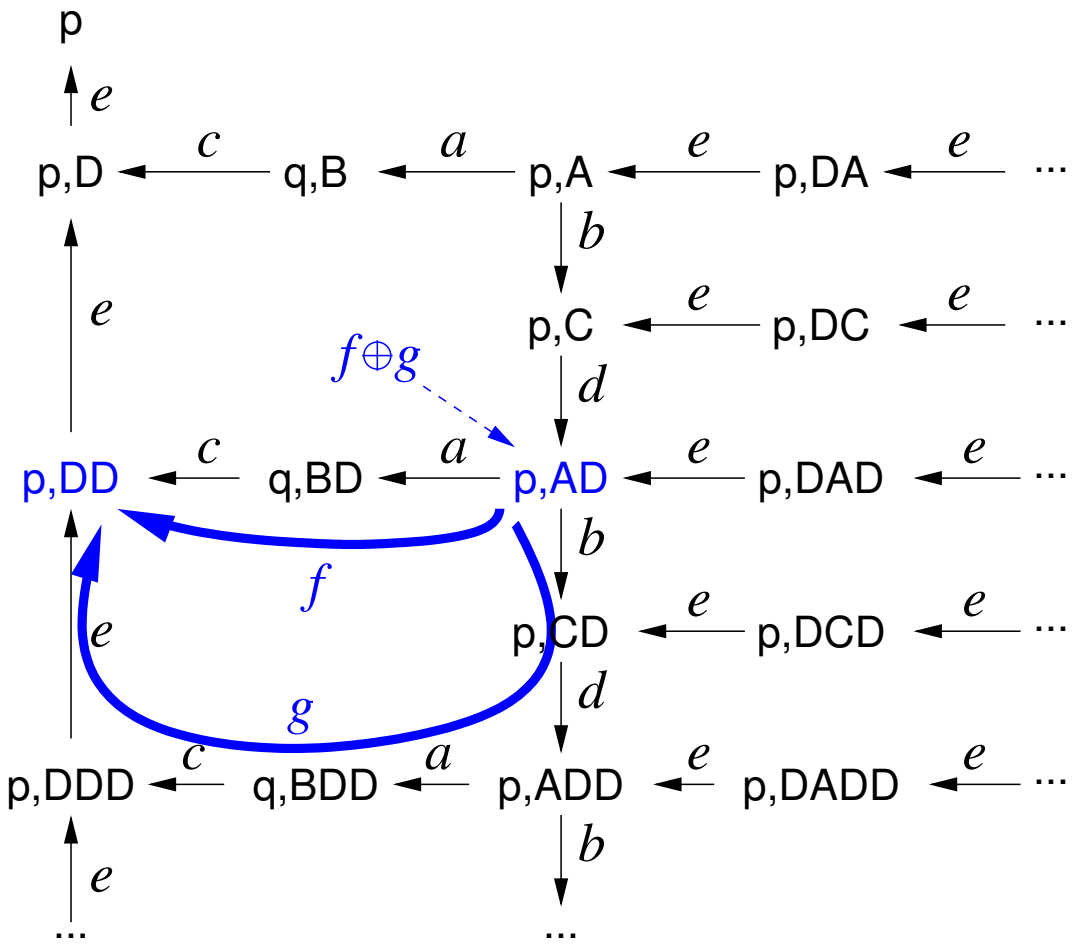
“Extend”: $\otimes$

Fügt Werte entlang eines Pfads zusammen:



“Combine”: $\oplus$

Fasst zwei Pfade zusammen:



Gewichtete PDS

Ein gewichtetes Pushdown-System $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ besteht aus:

einem PDS $\mathcal{P} = (P, \Gamma, \Delta)$;

einem beschränkten idempotenten Semiring $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$;

einer Gewichtsfunktion $f : \Delta \to D$.

Verallgemeinerte Erreichbarkeit

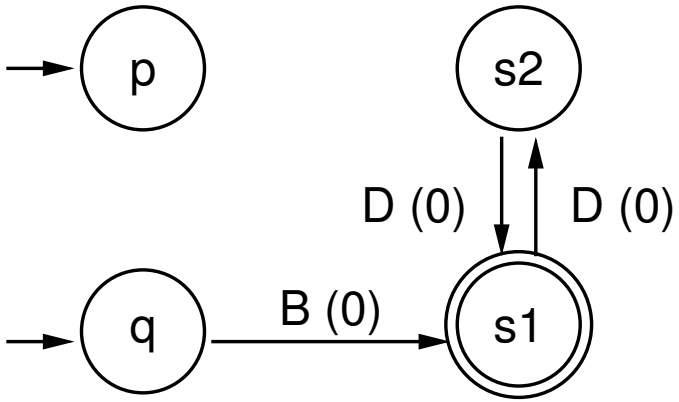
Sei $\pi = c_0 \xrightarrow{d_1} \cdots \xrightarrow{d_n} c_n$ ein Pfad in $\mathcal{P}$. Das Gewicht von π ist $v(\pi) = \bigotimes_{i=1}^{n} d_i$, und $\Pi(c, C)$ bezeichnet die Menge der Pfade von Konfiguration c zu Konfigurationen aus der Menge C.

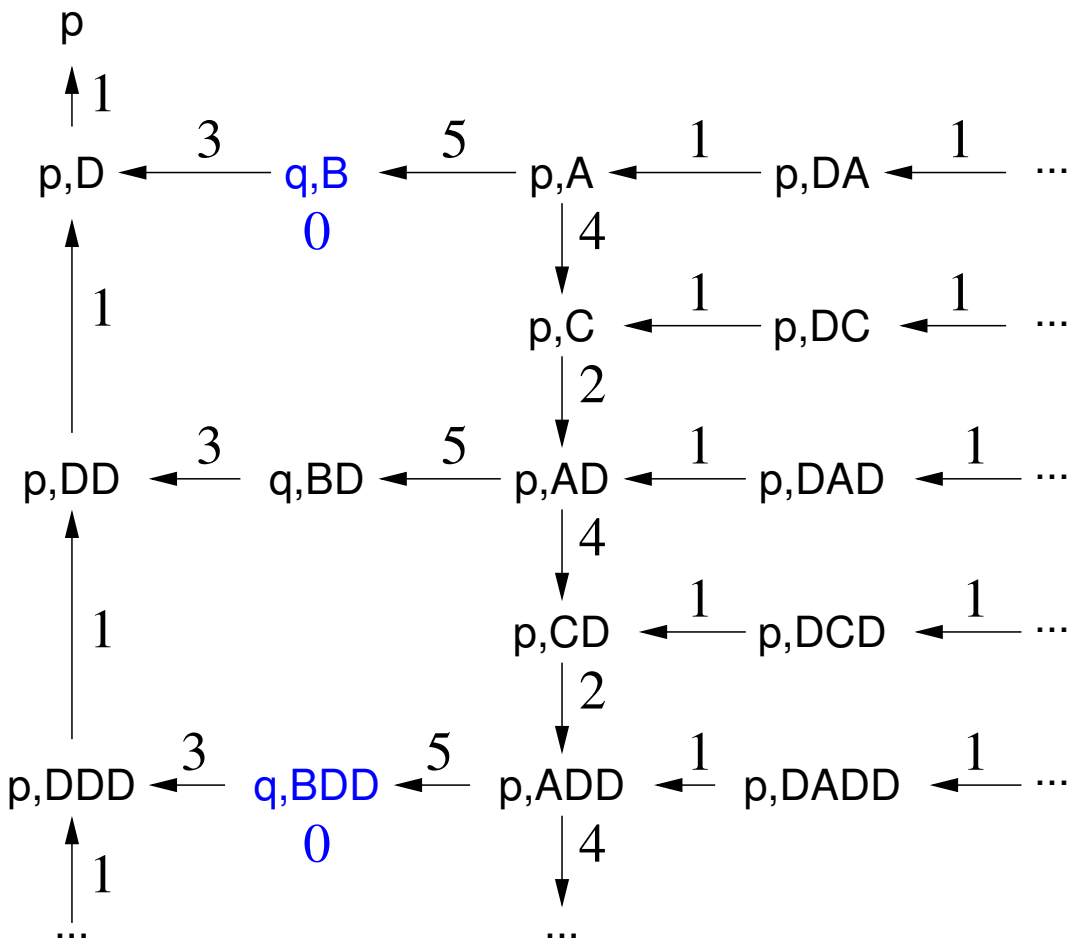
(für pre^*): Gegeben ein gewichtetes PDS $\mathcal{W} = (P, \mathcal{S}, f)$ und eine reguläre Menge C, berechne

$$\delta(c) = \bigoplus \{ v(\pi) \mid \pi \in \Pi(c, C) \}$$

für alle c in $pre^*(C)$.

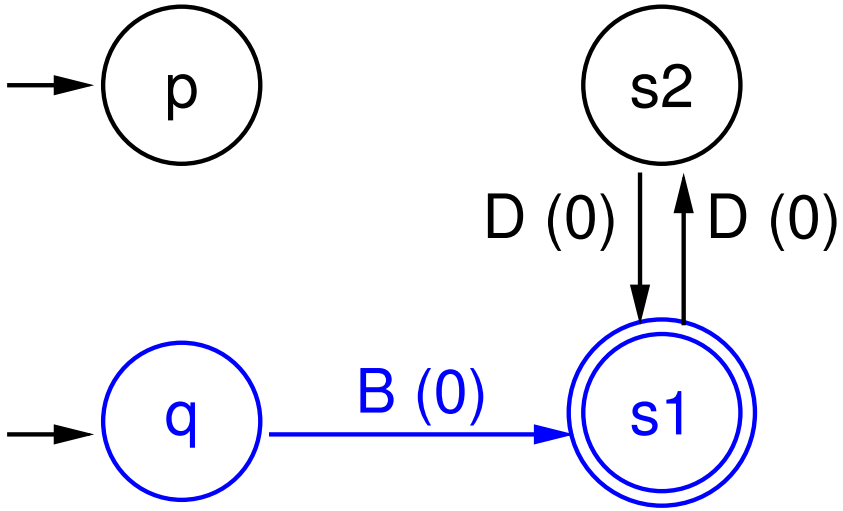
Beispiel: $\mathcal{S} = (\mathbb{N}, \min, +, \infty, 0)$, anfänglicher Automat





Erweitere $\mathcal{A}$ um zusätzliche Transitionen

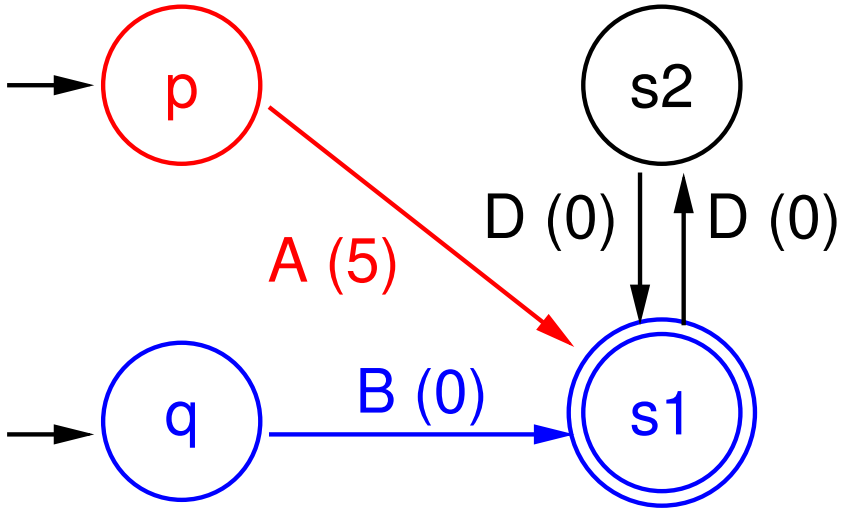
Wenn die rechte Seite (mit Gewicht d) mit Gewicht e gelesen wird



Regel: $\langle p, A\rangle \overset{5}{\hookrightarrow} \langle q, B\rangle$ Pfad: $q \overset{B\ (0)}{\rightarrow} s_1$

Erweitere $\mathcal{A}$ um zusätzliche Transitionen

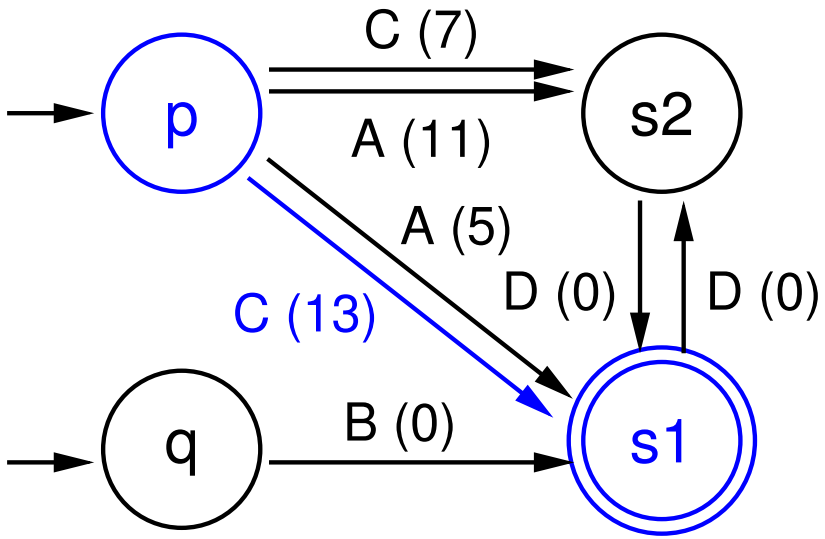
Wenn die rechte Seite (mit Gewicht d) mit Gewicht e gelesen wird
füge die linke Seite mit Gewicht $d \otimes e$ hinzu.



Regel: $\langle p, A\rangle \overset{5}{\hookrightarrow} \langle q, B\rangle$ Pfad: $q \overset{B\ (0)}{\rightarrow} s_1$ Neuer Pfad: $p \overset{A\ (5+0)}{\rightarrow} s_1$

Erweitere $\mathcal{A}$ um zusätzliche Transitionen

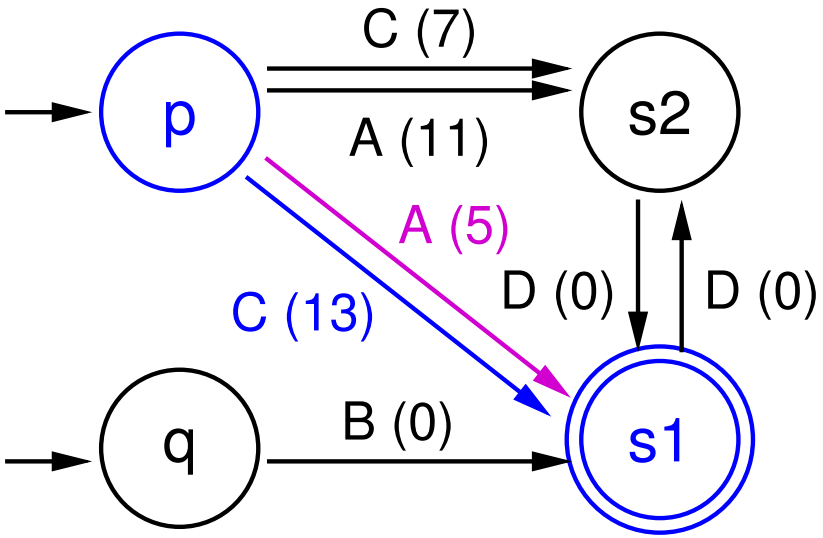
Wenn die rechte Seite (mit Gewicht d) mit Gewicht e gelesen wird



Regel: $pA \overset{4}{\hookrightarrow} pC$ Pfad: $p \overset{C\ (13)}{\rightarrow} s_1$,

Erweitere $\mathcal{A}$ um zusätzliche Transitionen

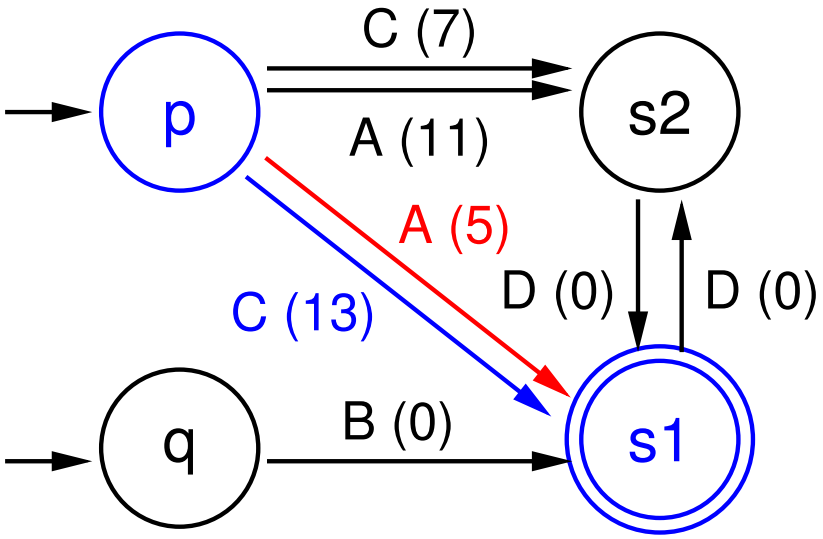
Wenn die rechte Seite (mit Gewicht d) mit Gewicht e gelesen wird
und die linke Seite schon Wert f hat,



Regel: $pA \overset{4}{\hookrightarrow} pC$ Pfad: $p \overset{C\ (13)}{\rightarrow} s_1$, $p \overset{A\ (5)}{\rightarrow} s_1$

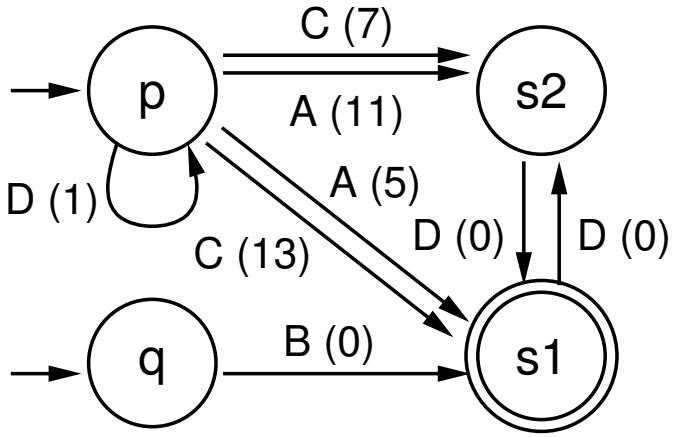
Erweitere $\mathcal{A}$ um zusätzliche Transitionen

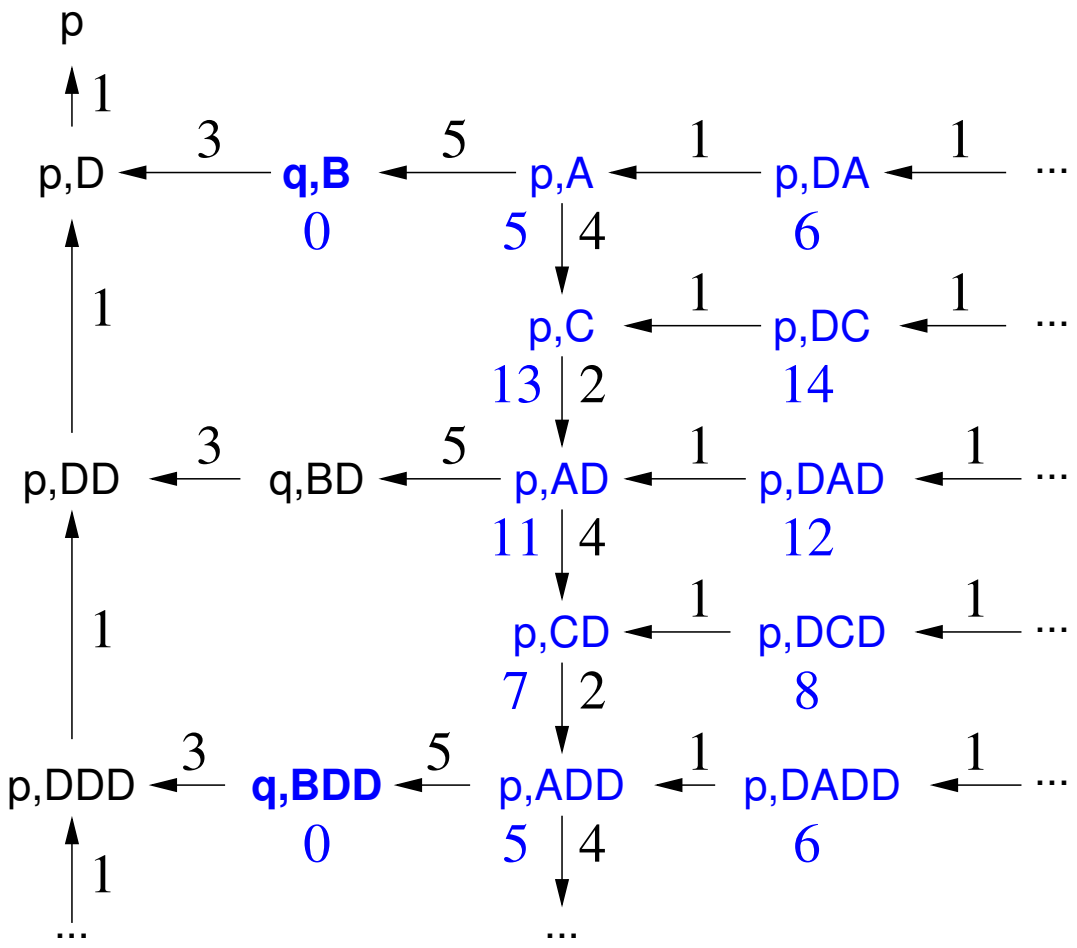
Wenn die rechte Seite (mit Gewicht d) mit Gewicht e gelesen wird
und die linke Seite schon Wert f hat, ändere f zu $f \oplus (d \otimes e)$.



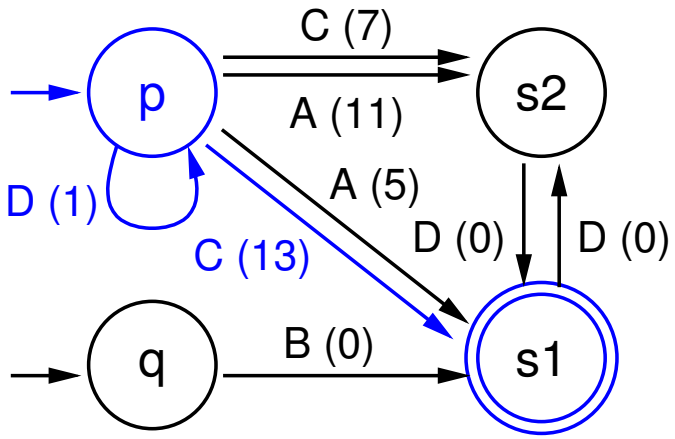
Regel: $pA \overset{4}{\hookrightarrow} pC$ Pfad: $p \overset{C\ (13)}{\rightarrow} s_1$, $p \overset{A\ (5)}{\rightarrow} s_1$ $5 \rightarrow \min\{5, 4 + 13\}$

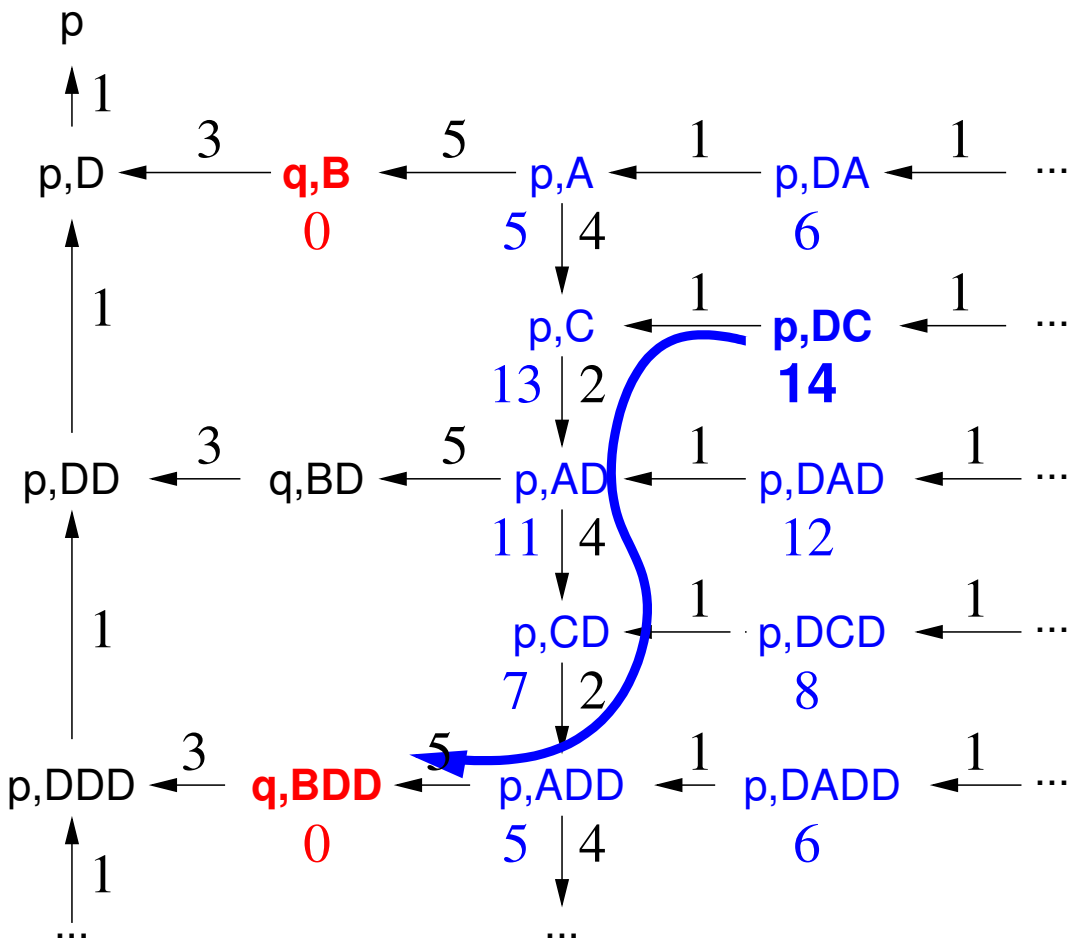
Beispiel: $\mathcal{S} = (\mathbb{N}, \min, +, \infty, 0)$, Endergebnis





Beispiel: $\mathcal{S} = (\mathbb{N}, \min, +, \infty, 0)$, Endergebnis





Anwendung: Interprozedurale Datenfluss-Analyse

Untersuchung von Programmen mit (rekursiven) Prozeduren

Anweisungen werden auf ihren Effekt hin reduziert
(bzgl. irgendeiner “interessanten” Eigenschaft)

Ziel: Für jede Programmzeile ℓ die Menge der Datenfluss-Fakten berechnen, die immer gelten, wenn eine Programm-Ausführung ℓ erreicht.

Ansatz:

Programme $\widehat{=}$ Pushdown-Systeme

Analysen $\widehat{=}$ Semiringe

Beispiel-Programm

```
int x;

void main() {
  n1: x = 5;
  n2: p();
  n3: return;
}

void p() {
  n4: if (...) {
  n5:    return;
  n6: } else if (...) {
  n7:    x = x + 1;
  n8:    p();
  n9:    x = x - 1;
      } else {
  n10:   x = x - 1;
  n11:   p();
  n12:   x = x + 1;
      }
}
```

Durch Semiring darstellbare Analysen

Bitvektor-Probleme (z.B. lebendige Variablen)

Linear-constant propagation

Relationen, z.B. BDDs

Zeiger-Analysen

. . .

Durch Semiring darstellbare Analysen

Bitvektor-Probleme (z.B. lebendige Variablen)

Linear-constant propagation

Relationen, z.B. BDDs

Zeiger-Analysen

. . .

Linear-Constant Propagation als Semiring

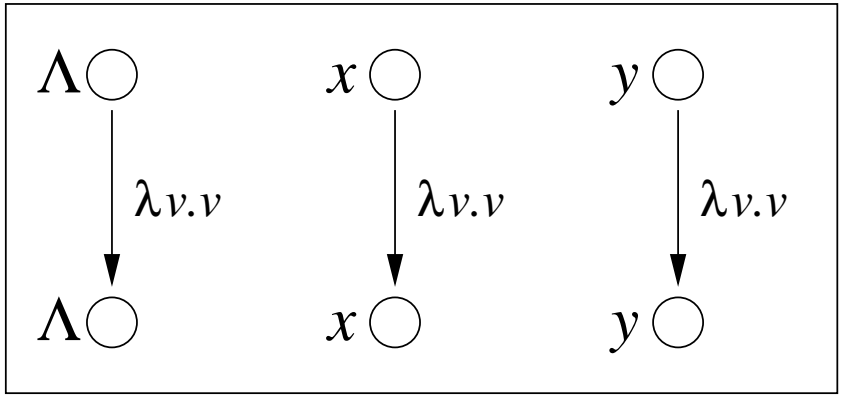
Fragestellung: Gilt sicher $x = ay + b$ (für irgendwelche Konstanten a, b und Variablen x, y), wenn eine Ausführung Zeile ℓ erreicht? (Anwendung: Programmoptimierung)

Methode: Analysiere Zuweisungen der Form `x := 2*y+5` oder `x := 7;` .

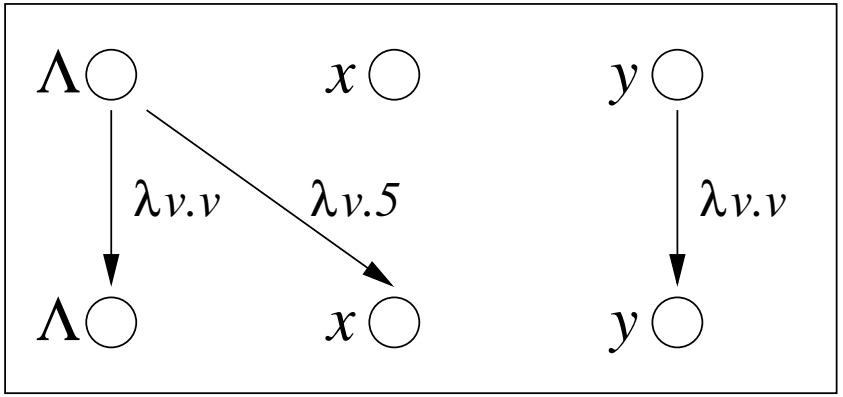
Alle anderen Zuweisungen werden zu `x := ⊥;` vereinfacht.

Zuweisungen werden durch Semiring-Werte dargestellt
(hier: bipartite Graphen).

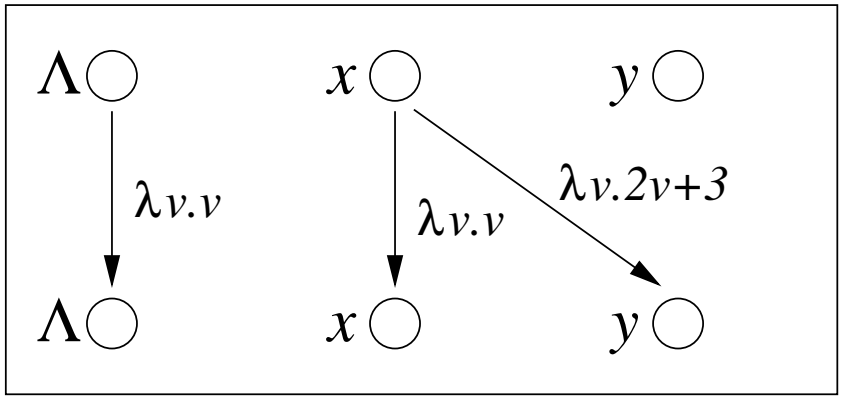
Beispiele für Semiring-Werte



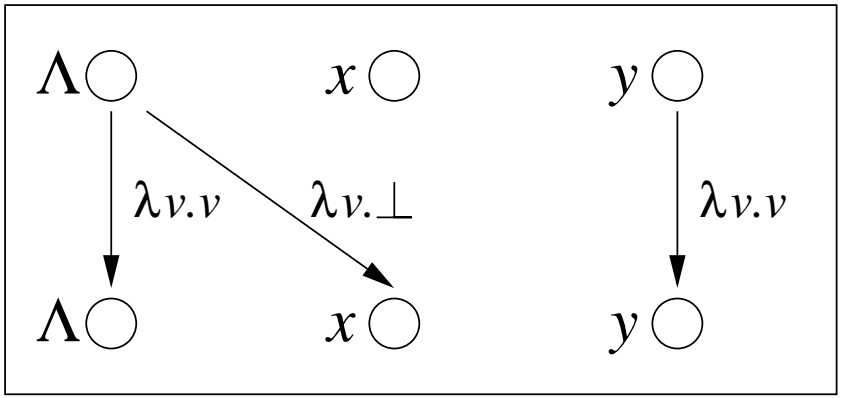
skip



x := 5

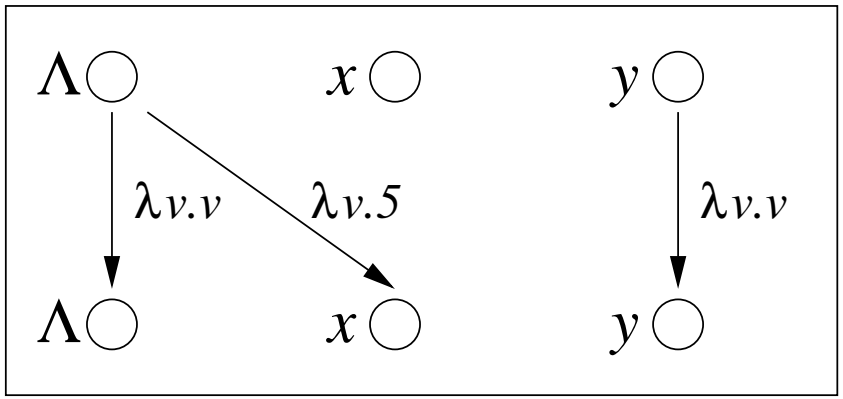


y := 2*x + 3

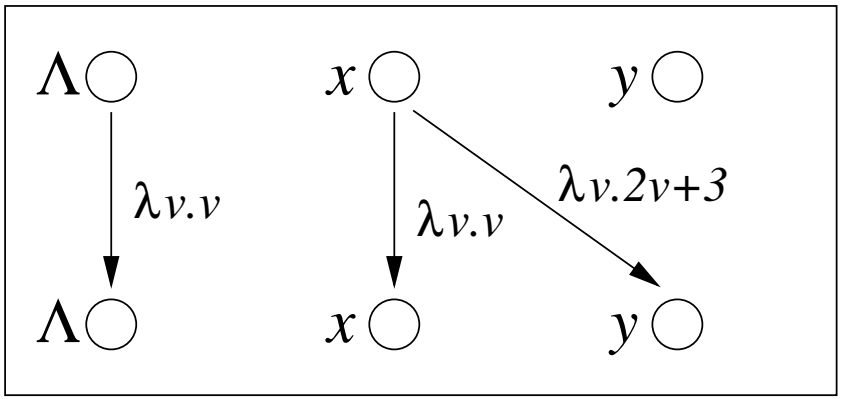


x := sin(y)

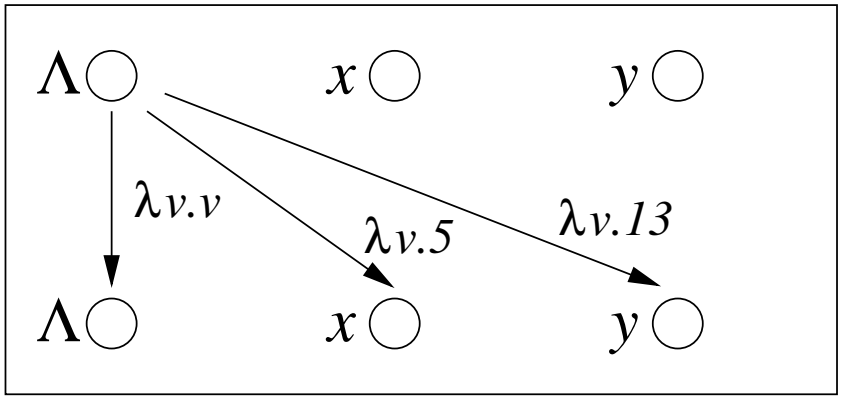
Extend: Graphen “aneinanderkleben”



x := 5

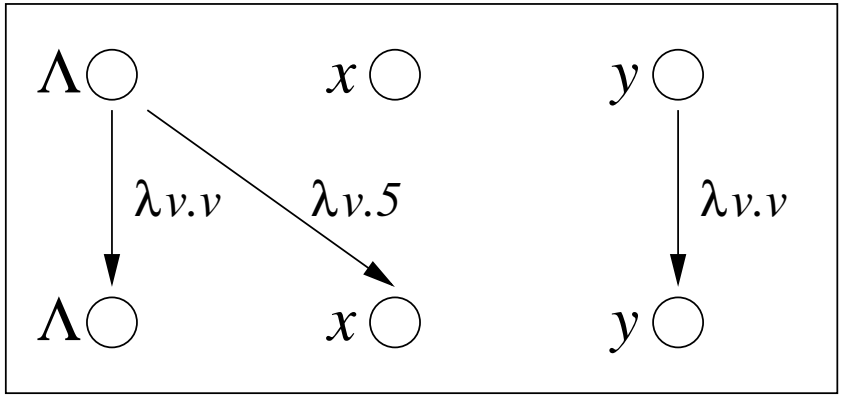


y := 2*x + 3

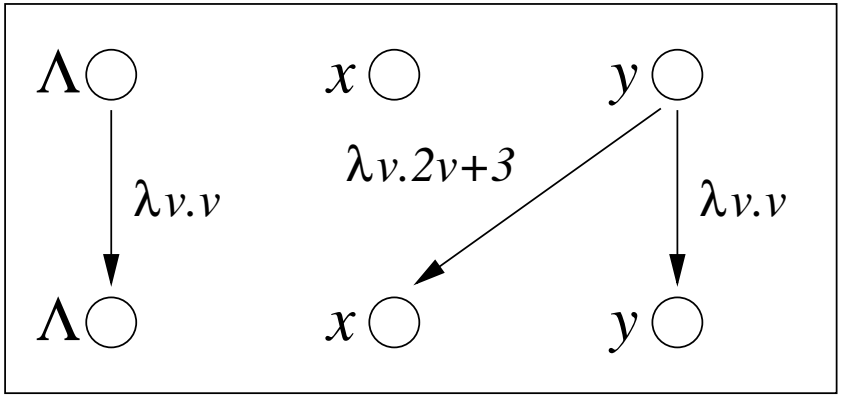


x := 5 ⊗ y := 2*x + 3

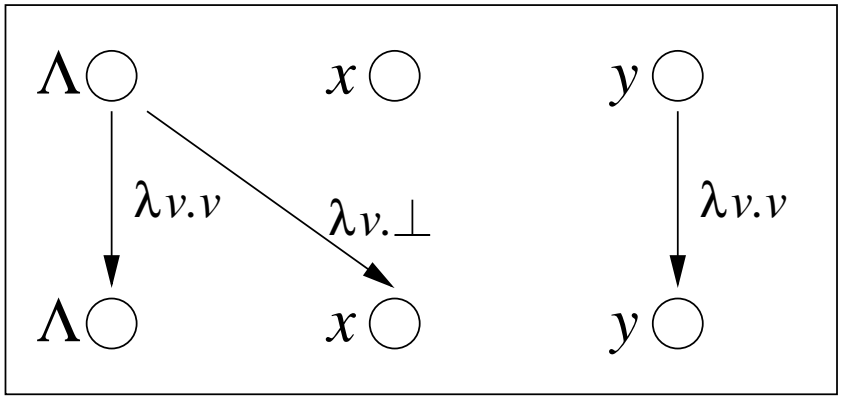
Combine: Prüfen, ob die Zuweisungen “kompatibel” sind



x := 5



x := 2*y + 3

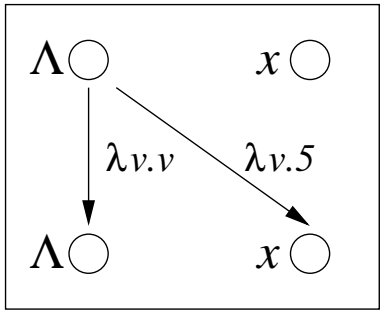


x := 5 ⊕ x := 2*y + 3

Beispiel

```
int x;

void main() {
  n1: x = 5;
  n2: p();
  n3: return;
}

void p() {
  n4: if (...) {
  n5:    return;
  n6: } else if (...) {
  n7:    x = x + 1;
  n8:    p();
  n9:    x = x - 1;
      } else {
  n10:   x = x - 1;
  n11:   p();
  n12:   x = x + 1;
      }
}
```

Gewichtete *post**-Berechnung für $C = \{\langle q, n_1 \rangle\}$ ergibt $\delta(\langle q, \varepsilon \rangle) =$



d.h., am Ende des Programm ist x immer 5.

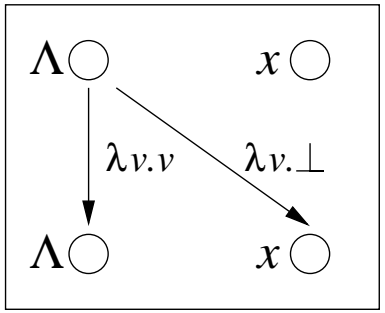
Beispiel

```
int x;

void main() {
  n1: x = 5;
  n2: p();
  n3: return;
}

void p() {
  n4: if (...) {
  n5:    return;
  n6: } else if (...) {
  n7:    x = x + 1;
  n8:    p();
  n9:    x = x - 1;
      } else {
  n10:   x = x - 1;
  n11:   p();
  n12:   x = x + 1;
      }
}
```

Gewichtete *pre**-Berechnung für

$C = \{ \langle q, n_4 w \rangle \mid w \in \Gamma^* \}$

ergibt $\delta(\langle q, n_1 \rangle) =$



d.h., p kann mit unterschiedlichen Werten von x erreicht werden.

LTL-Model-Checking

Sei $\mathcal{P} = (P, \Gamma, \Delta)$ ein PDS mit Anfangskonfiguration c_0, $\mathcal{T}_\mathcal{P}$ das dazugehörige Transitionssystem, AP eine Menge von Grundaussagen und $\nu : P \times \Gamma^* \to 2^{AP}$. eine Belegungsfunktion.

$\mathcal{K}$ sei die aus $\mathcal{T}_\mathcal{P}$, AP und ν gebildete Kripke-Struktur und ϕ eine LTL-Formel (über AP).

Frage: Gilt $\mathcal{K} \models \phi$?

LTL-Model-Checking

Sei $\mathcal{P} = (P, \Gamma, \Delta)$ ein PDS mit Anfangskonfiguration c_0, $\mathcal{T}_\mathcal{P}$ das dazugehörige Transitionssystem, AP eine Menge von Grundaussagen und $\nu : P \times \Gamma^* \to 2^{AP}$. eine Belegungsfunktion.

$\mathcal{K}$ sei die aus $\mathcal{T}_\mathcal{P}$, AP und ν gebildete Kripke-Struktur und ϕ eine LTL-Formel (über AP).

Frage: Gilt $\mathcal{K} \models \phi$?

Ist das überhaupt entscheidbar?

LTL-Model-Checking

Sei $\mathcal{P} = (P, \Gamma, \Delta)$ ein PDS mit Anfangskonfiguration c_0, $\mathcal{T}_\mathcal{P}$ das dazugehörige Transitionssystem, AP eine Menge von Grundaussagen und $\nu : P \times \Gamma^* \to 2^{AP}$. eine Belegungsfunktion.

$\mathcal{K}$ sei die aus $\mathcal{T}_\mathcal{P}$, AP und ν gebildete Kripke-Struktur und ϕ eine LTL-Formel (über AP).

Frage: Gilt $\mathcal{K} \models \phi$?

Ist das überhaupt entscheidbar?

$\Rightarrow$ Im Allgemeinen: Nein!
(für beliebige ν könnte man unentscheidbare Probleme modellieren)

Im Folgenden betrachten wir daher diese Einschränkung:

Sei $\widehat{\nu}$ eine Funktion von $P \times \Gamma$ (ohne Stern!) nach 2^{AP}.

Wir verlangen, dass $\nu(pAw) = \widehat{\nu}(pA)$, für alle $p \in P$, $A \in \Gamma$ und $w \in \Gamma^*$.

D.h. man kann dem “Anfang” einer Konfiguration ansehen, ob sie eine Grundaussage erfüllt.

Unter dieser Einschränkung ist LTL auf PDS entscheidbar!

Vorgehensweise

Prinzip genau wie bei endlichen Kripke-Strukturen:

Übersetze $\neg\phi$ in einen Büchi-Automaten $\mathcal{B}$.

Bilde das Kreuzprodukt von $\mathcal{K}$ und $\mathcal{B}$.

Teste das Kreuzprodukt auf Leerheit!

Problem: Das Kreuzprodukt ist kein Büchi-Automat! (nicht endlich)

Büchi-PDS

Das Kreuzprodukt wird wie folgt gebildet:

Sei $\mathcal{P} = (P, \Gamma, \Delta)$ ein PDS, $p_0 w_0$ eine Anfangskonfiguration und AP, ν wie üblich.

Sei $\mathcal{B} = (2^{AP}, Q, q_0, \delta, F)$ der Büchi-Automat.

Wir bilden ein Büchi-PDS wie folgt:

$\mathcal{BP} = (P \times Q, \Gamma, \Delta', P \times F)$, wobei

$(p, q)A \hookrightarrow (p', q')w \in \Delta'$ gdw.

– $pA \hookrightarrow p'w \in \Delta$ und

– $(q, L, q') \in \delta$, so dass $\nu(pA) = L$.

Anfangskonfiguration $c_0 := (p_0, q_0)w_0$

$\mathcal{BP}$ ist ein Pushdown-System mit einer Menge akzeptierender Kontrollzustände (im vorliegenden Fall $P \times F$).

Ein Ablauf (Folge von Konfigurationen) von $\mathcal{BP}$ heißt akzeptierend, wenn darin unendlich viele akzeptierende Kontrollzustände vorkommen.

Proposition: Die zu $\mathcal{P}, AP, \nu$ gehörige Kripke-Struktur erfüllt ϕ *nicht* gdw. es in $\mathcal{BP}$ einen akzeptierenden Ablauf beginnend bei c_0 gibt.

Charakterisierung akzeptierender Abläufe

Frage: Gibt es einen solchen akzeptierenden Ablauf beginnend bei c_0?

Im Folgenden betrachten wir diese verallgemeinerte Fragestellung, das globale Model-Checking-Problem:

> Berechne *alle* Konfigurationen c, so dass es einen akzeptierenden Ablauf beginnend bei c gibt.

Charakterisierung akzeptierender Abläufe

Frage: Gibt es einen solchen akzeptierenden Ablauf beginnend bei c_0?

Im Folgenden betrachten wir diese verallgemeinerte Fragestellung:

> Berechne *alle* Konfigurationen c, so dass es einen akzeptierenden Ablauf beginnend bei c gibt.

Lemma: Für c gibt es einen solchen Ablauf gdw. es $p \in P$, $A \in \Gamma$ gibt mit folgendenden Eigenschaften:

(1) $c \Rightarrow pAw$ für irgendein $w \in \Gamma^*$

(2) $pA \Rightarrow pAw'$ für irgendein $w' \in \Gamma^*$, wobei

der Pfad von pA nach pAw' mindestens einen Schritt enthält;

auf diesem Pfad mindestens ein akzeptierender Kontrollzustand vorkommt.

Schleifenköpfe

Wir nennen pA einen Schleifenkopf, wenn pA die Eigenschaften (1) und (2) besitzen.

Strategie:

1. Berechne die Menge der Schleifenköpfe.
(naiv: prüfe für jedes pA, ob $pA \in pre^*(\{ pAw \mid w \in \Gamma^* \})$ mit Gewichten aus boolschem Semiring, um Besuch akzeptierender Zustände zu überprüfen)

2. Berechne die Menge $pre^*(\{ pAw \mid pA$ ist Schleifenkopf, $w \in \Gamma^* \})$

Verbesserung für 1.

Berechne zunächst *einmal* $pre^*(\{ p\varepsilon \mid p \in P \})$ (gewichtet mit boolschem Semiring wie vorher).

Bilde den folgenden *endlichen* gerichteten Graphen $G = (V, \rightarrow)$:

$V = P \times \Gamma$

$pA \rightarrow qB$ gdw. es gibt $pA \hookrightarrow rvBw$ mit $r \in P$, $v, w \in \Gamma^*$ und $rv \Rightarrow q\varepsilon$

Beschrifte die Kante $pA \rightarrow qB$ mit 1 gdw. der Pfad von rv nach $q\varepsilon$ einen akzeptierenden Zustand besucht.

Finde die SCCs in G, die eine 1-Kante enthalten.

pA ist ein Schleifenkopf gdw. es in einer solchen SCC enthalten ist.

Tool-Demonstration: Moped

Moped: Modelchecker für PDS (Erreichbarkeit und LTL)

Eingabe: PDS oder boolsche Programme (mit Prozeduren)

jMoped: Erweiterung für Java

```
http://www7.in.tum.de/tools/jmoped/
```

CTL-Model-Checking auf PDS: Strategie

Betrachte die Logik CTL*
(Verallgemeinerung von LTL *und* CTL)

Erweitere den LTL-Algorithmus in einen Algorithmus für CTL*
(allgemeines Verfahren, funktioniert auch für endliche Strukturen)

CTL*: Syntax

Sei AP eine Menge von Grundaussagen.

Es gibt Zustandsformeln und Pfadformeln:

- a für $a \in AP$ ist eine Zustandsformel.
- Wenn ϕ, ψ Zustandsformeln sind, dann auch $\neg\phi$ und $\phi \vee \psi$.
- Wenn ϕ eine Pfadformel ist, dann ist $\mathbf{E}\,\phi$ eine Zustandsformel.
- Jede Zustandsformel ist auch eine Pfadformel.
- Wenn ϕ, ψ Pfadformeln sind, dann auch $\neg\phi$, $\phi \vee \psi$, $\mathbf{X}\,\phi$, $\phi\,\mathbf{U}\,\psi$.

Die Menge der CTL*-Formel ist die Menge der Zustandsformeln.

CTL*: Semantik (informell)

Jeder Zustandsformel wird eine Menge von Zuständen zugeordnet:

- Grundaussagen und boolsche Kombinationen: offensichtlich
- $\mathbf{E}\,\phi$: die Menge der Zustände, von denen aus einen Pfad gibt, der die Pfadformel ϕ erfüllt.

Jeder Pfadformel wird eine Menge von Pfaden zugeordnet:

- Zustandsformeln: die Menge der Pfade, die mit einem solchen Zustand beginnt.
- alles andere: wie bei LTL

Am Ende schaut man, ob der Anfangszustand in der Semantik enthalten ist.

CTL* als Obermenge von LTL und CTL

Wenn ϕ eine LTL-Formel ist, dann ist $\neg\,\mathbf{E}\,\neg\phi$ eine CTL*-Formel mit der Eigenschaft: $\mathcal{K} \models \phi$ gdw. der Anfangszustand von $\mathcal{K}$ in der Semantik von $\neg\,\mathbf{E}\,\neg\phi$ enthalten ist.

Wenn ϕ eine CTL-Formel ist, so ist ϕ auch eine CTL*-Formel mit gleicher Semantik.

CTL* ist also mächtiger als CTL und LTL, hat aber weniger effiziente MC-Algorithmen, daher nicht so populär.

CTL*-Modelchecking für endliche Systeme

Sei $\mathcal{K}$ eine Kripke-Struktur mit *endlicher* Zustandsmenge S und ϕ sei eine CTL*-Formel.

Gegeben sei ein globaler MC-Algorithmus, der zu einer LTL-Formel die Menge *aller* Zustände liefert, von denen aus *irgendein* Ablauf die Formel erfüllt.

Bottom-Up-Strategie:

1. Wenn ϕ keine Unterformel mit $\mathbf{E}$ enthält, berechne die Semantik von ϕ durch geeignete Mengenoperationen.

2. Ansonsten gibt es eine Teilformel $\mathbf{E}\,\psi$, so dass ψ kein $\mathbf{E}$ enthält und daher eine LTL-Formel ist. Anwendung des globalen MC-Algorithmus auf ψ liefert eine Menge $M \subseteq S$. Ersetze alle Vorkommen von $\mathbf{E}\,\psi$ in ϕ durch eine 'frische' Grundaussage b, und sorge dafür, dass b auf genau den Zuständen aus M gilt.

3. Fahre mit 1. fort.

CTL*-Modelchecking für PDS

Gleiches Prinzip wie bei endlichen Systemen.

Problem: (bei Schritt 2) Unser globaler MC-Algorithmus für PDS liefert eine *reguläre* Menge von Konfigurationen, die $\mathbf{E}\,\psi$ erfüllen.

D.h. die Gültigkeit der ‘frischen’ Grundaussage *b* hängt nicht nur vom Kontrollzustand und dem obersten Stackzeichen ab, sondern auch vom Stack; inkompatibel mit unserer zuvor gemachten Einschränkung.

Lösung: Kodiere einen endlichen Automaten ins PDS hinein.

Vorgehensweise

Sei $\mathcal{A}$ der Automat, der beim globalen MC von $\mathbf{E}\,\psi$ entsteht.

Wandle $\mathcal{A}$ per Potenzmengenkonstruktion in einen *deterministischen*, vollständigen Automaten um, der den Stack rückwärts liest.

Sei Γ das Stackalphabet des PDS und Q die Zustände von $\mathcal{A}$. Wir modifizieren das PDS, indem wir das Stackalphabet zu $\Gamma \times Q$ erweitern.

Ziel: Im modifizierten PDS soll $p(A_0, q_0)(A_1, q_1) \ldots (A_n, q_n)$ erreichbar sein gdw. $pA_1 \ldots A_n$ im ursprünglichen PDS erreichar war, q_n der Anfangszustand von $\mathcal{A}$ ist und $A_n \ldots A_1$ von q_0 nach q_n führt.

Anfangskonfiguration geeignet anpassen

Änderung der PDS-Regeln:

Falls $pA \hookrightarrow p'B$ im PDS, dann $p(A,q) \hookrightarrow p'(B,q)$ für alle $q \in Q$.

Falls $pA \hookrightarrow p'\varepsilon$ im PDS, dann $p(A,q) \hookrightarrow p'\varepsilon$ für alle $q \in Q$.

Falls $pA \hookrightarrow p'BC$ im PDS, dann $p(A,q) \hookrightarrow p'(B,q')(C,q)$ so dass q in $\mathcal{A}$ mit C nach q' übergeht.

Jetzt kann man festlegen: $p(A,q)w$ erfülle b gdw. q mit A in einen Zustand s übergeht, so dass $p \in s$.

Komplexitätsbetrachtungen

Stackalphabet wächst mit jeder verschachtelten $\mathbf{E}$-Formel weiter an, jedesmal schlimmstenfalls um den Faktor $2^{|P|}$.

Man kann zeigen: Das Modelchecking-Problem für PDS und CTL* (sogar für CTL) ist EXPTIME-vollständig.

Reduktion auf Akzeptanzproblem in beschränkten, alternierenden Turing-Maschinen (Walukiewicz, 2000)

PDS und Nebenläufigkeit

PDS sind sequentielle Systeme (nur ein Prozess).

Oft sind nebenläufige Systeme von Interesse.

Mögliches formales Modell für Systeme mit Rekursion und Nebenläufigkeit:

Kellerautomat (PDS) mit einem Kontrollzustand und zwei oder mehr Stacks

Problem: Dieses Modell ist Turing-mächtig!

Ansätze

Unterapproximation, z.B. “context-bounded reachability” (Qadeer, Rehof 2005)

Überapproximation kombiniert mit CEGAR, z.B. Touili, Reps et al 2006

Einschränkung, z.B. Kommunikation nur über Locks, auf die in FIFO-Manier zugegriffen wird (synchronized-Konstrukt in Java!), siehe Kahlon, Ivancić, Gupta 2005

Von Stacks zu Bäumen

Zustände (Konfigurationen) in PDS sind Wörter

Allgemeines Prinzip der Theoretischen Informatik:

> Was mit Wörtern geht, kann man auch mit Bäumen probieren.
> (z.B. LTL $\rightarrow$ CTL).

PDS (Systeme mit Wörtern) $\widehat{=}$ imperative Programmiersprachen

Systeme mit Bäumen $\widehat{=}$ funktionale Programmiersprachen (Terme!)

Baumersetzungssysteme (Skizze)

Zustände sind Bäume.

Regeln bestimmen, welche Teilbäume durch andere Teilbäume ersetzt werden können.

Mengen von Bäumen werden durch Baumautomaten dargestellt.

Regularität bleibt unter Erreichbarkeit erhalten, pre^*-Algorithmus kann angepasst werden.

Mehr Details: Löding, 2006