

# Formale Spezifikation und Verifikation

Ulrich Schöpp

Sommersemester 2016

# Was ist Spezifikation?

Beschreibung eines Teils des gewünschten Systemverhaltens.

- QuickSort berechnet sortierte Permutation.
- Iterator bietet alle Elemente der Menge in einer beliebigen Reihenfolge an.
- Haben zwei Objekte den gleichen `hashCode()`, so sind sie gleich bzgl. `equals()`.
- Die vom Roboterarm beschriebene Trajektorie muss innerhalb der erlaubten Toleranz mit der im Programm vorgegebenen übereinstimmen. Die Grenzggeschwindigkeiten und -beschleunigungen dürfen nicht überschritten werden.
- Die Handy-Anwendung darf nicht mehr SMS verschicken als zuvor autorisiert.

Nicht: Zweck des Systems (“Ihre Software soll in der Buchhaltung 10% Personal einsparen.”)

# Was ist Verifikation?

Nachweis, dass System eine gegebene Spezifikation erfüllt.

- Durch Tests (“unvollständig”)
- Durch informelle Argumentation
- Durch systematische Tests
- Durch rigorosen Beweis
- Durch automatisches Verifikationswerkzeug
- Durch formalisierten Beweis

# Was heißt “formal”?

Mit Mitteln der Logik ausgedrückt.

- Bedeutung einer formalen Spezifikation ist exakt festgelegt (könnte aber möglicherweise das Gewünschte nicht erfassen)
- Eine formale Verifikation liefert die Gültigkeit der zugehörigen Spezifikation mit 100%-iger Sicherheit, allerdings nur für das zugrundegelegte formale Modell des Systems.  
(Extrembeispiele: Kurzschluss, Überlastung, Compilerfehler)
- Formale Spezifikation und Verifikation kann die Systementwicklung sinnvoll unterstützen, ist aber kein Allheilmittel.

# Inhalt

- I **Aussagenlogik:** Wdh. Syntax, Semantik, Reduktion auf SAT, SAT-Solver, Anwendung auf Modellierung, BDDs.
- II **Temporallogik und Model Checking:** Temporallogik CTL, Syntax und Semantik, Model Checking Algorithmen, Temporallogik LTL und Büchi-Automaten, das Werkzeug SMV.
- III **Programmanalyse und Typsysteme:** Operationelle Semantik, Datenflussanalyse, Typsysteme.
- IV **Programmlogik:** Hoare-Logik, JML.

# Organisatorisches, Literatur

Vorlesungs- und Klausurtermine, siehe Homepage der Veranstaltung.

## Literatur

- Huth, Ryan: Logic in Computer Science.
- Clarke, Grumberg, Peled: Model checking.
- Nielson, Nielson, Hankin: Program analysis.
- Als Wdh. der Logik: Skripten von Till Tantau, Martin Lange, Martin Hofmann.

Das Folienskript sollte als Unterlage genügen; ersetzt aber nicht den Besuch der Vorlesung und Übungen.

# Kapitel I

## Aussagenlogik

# Inhalt Kapitel I

- Überblick über die Vorlesung
- Motivation
- Syntax und Semantik der Aussagenlogik
- Grundbegriffe
  - Mengennotation und indizierte Variablen
- Normalformen
- SAT-Solver
- Anwendungen
  - Sudoku
  - Schaltkreisverifikation
  - Verifikation nebenläufiger Systeme
  - Peterson Algorithmus
- BDDs (binäre Entscheidungsdiagramme)
  - Grundlegende Definitionen
  - Logische Operationen auf BDDs
  - Implementierung von BDDs



# Protokollchef

- Der Botschafter bittet Sie eine Einladungsliste für den Ball der Botschaft zusammenzustellen.
- Sie sollen Peru einladen oder Katar nicht einladen.
- Der Vizebotschafter möchte, dass Sie Katar, Rumänien, oder beide einladen.
- Aufgrund eines aktuellen Vorfalls können Sie nicht Rumänien und Peru zusammen einladen.

Wie stellen Sie die Einladungsliste zusammen?

# Modellierung in Aussagenlogik

- Peru oder nicht Katar:  $P \vee \neg K$
- Katar oder Rumänien:  $K \vee R$
- Nicht Rumänien und Peru zusammen:  $\neg(R \wedge P)$

Man muss die *aussagenlogischen Variablen*  $P, K, R$  so mit Wahrheitswerten *true*, *false* belegen, dass alle drei *Formeln* wahr werden. Ist das möglich, so ist die Formelmenge *erfüllbar*.

Die Formelmenge  $\{P \vee \neg K, K \vee R, \neg(R \wedge P)\}$  ist erfüllbar. Eine *erfüllende Belegung* ist  $P = \text{true}, K = \text{true}, R = \text{false}$ .  
Alternative:  $P = \text{false}, R = \text{true}, K = \text{false}$ .

# Syntax der Aussagenlogik

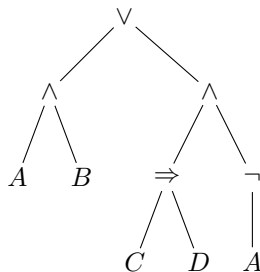
Sei eine Menge  $\mathcal{A}$  von Aussagenvariablen  $A, B, C, D \dots$  gegeben.  
Die *aussagenlogischen Formeln* über  $\mathcal{A}$  sind durch folgende BNF Grammatik definiert.

$\mathcal{F}$	$::=$	$\mathcal{A}$	
		$\neg \mathcal{F}$	(Negation, Verneinung)
		$\mathcal{F} \wedge \mathcal{F}$	(Konjunktion, logisches Und)
		$\mathcal{F} \vee \mathcal{F}$	(Disjunktion, logisches Oder)
		$\mathcal{F} \Rightarrow \mathcal{F}$	(Implikation, Wenn-Dann-Beziehung)
		$\mathcal{F} \Leftrightarrow \mathcal{F}$	(Äquivalenz, Genau-Dann-Wenn-Beziehung, "iff")
		$\top$	Verum, wahre Formel
		$\perp$	Falsum, falsche Formel

Die Symbole  $\{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$  heißen Junktoren.

# Syntaxbäume

Formal ist eine Formel ein Syntaxbaum:



Der Baum entspricht  $(A \wedge B) \vee ((C \Rightarrow D) \wedge (\neg A))$ .

Manche Klammern darf man weglassen, da nach Konvention  $\neg$  am stärksten bindet und dann  $\wedge$ ,  $\vee$ ,  $\Rightarrow$ ,  $\Leftrightarrow$ .

Gleiche Junktoren werden von rechts geklammert, also

$A \Rightarrow B \Rightarrow C$  ist  $A \Rightarrow (B \Rightarrow C)$ .

# Semantik der Aussagenlogik

- Eine Formel der Form  $\phi \wedge \psi$  ist wahr, wenn  $\phi$  und  $\psi$  beide wahr sind. Ist entweder  $\phi$  oder  $\psi$  falsch, so ist  $\phi \wedge \psi$  falsch.
- Eine Formel der Form  $\phi \vee \psi$  ist wahr, wenn  $\phi$  wahr ist oder wenn  $\psi$  wahr ist. Nur wenn  $\phi$  und  $\psi$  beide falsch sind, ist die Formel  $\phi \vee \psi$  falsch.
- $\neg\phi$  ist wahr, wenn  $\phi$  falsch ist. Ist  $\phi$  wahr, so ist  $\neg\phi$  falsch.
- $\phi \Rightarrow \psi$  ist wahr, wenn entweder  $\phi$  falsch ist, oder aber  $\phi$  wahr ist und  $\psi$  dann auch wahr ist. Nur wenn  $\phi$  wahr ist und zugleich  $\psi$  falsch ist, ist  $\phi \Rightarrow \psi$  falsch.
- Eine Formel der Form  $\phi \Leftrightarrow \psi$  ist wahr, wenn  $\phi$  und  $\psi$  denselben Wahrheitsgehalt haben, also entweder beide wahr, oder beide falsch sind.
- Die Formel  $\top$  ist wahr, die Formel  $\perp$  ist nicht wahr; beides unabhängig vom Wahrheitsgehalt der Variablen.

# Formale Semantik

Eine *Belegung* ist eine Funktion, die aussagenlogischen Variablen einen Wahrheitswert zuweist.

Einer Formel  $\phi$  wird für jede Belegung  $\eta$  ein Wahrheitswert  $\llbracket \phi \rrbracket \eta$  zugeordnet.

$$\llbracket A \rrbracket \eta = \eta(A)$$

$$\llbracket \neg \phi \rrbracket \eta = !\llbracket \phi \rrbracket \eta$$

$$\llbracket \top \rrbracket \eta = \text{true}$$

$$\llbracket \perp \rrbracket \eta = \text{false}$$

$$\llbracket \phi \wedge \psi \rrbracket \eta = \llbracket \phi \rrbracket \eta \& \llbracket \psi \rrbracket \eta$$

$$\llbracket \phi \vee \psi \rrbracket \eta = \llbracket \phi \rrbracket \eta || \llbracket \psi \rrbracket \eta$$

$$\llbracket \phi \Rightarrow \psi \rrbracket \eta = !\llbracket \phi \rrbracket \eta || \llbracket \psi \rrbracket \eta$$

$$\llbracket \phi \Leftrightarrow \psi \rrbracket \eta = (\llbracket \phi \rrbracket \eta = \llbracket \psi \rrbracket \eta)$$

wobei  $!$ ,  $\&$  und  $||$  durch folgende Wahrheitstabellen gegeben sind.

$\&$	<i>false</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>false</i>
<i>true</i>	<i>false</i>	<i>true</i>

$  $	<i>false</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>true</i>
<i>true</i>	<i>true</i>	<i>true</i>

$!$	<i>false</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>false</i>
<i>true</i>	<i>false</i>	<i>true</i>

# Allgemeingültigkeit

## Definition

Eine Formel  $\phi$  ist *allgemeingültig*, wenn sie unabhängig vom Wahrheitsgehalt der Variablen stets wahr ist. Formal also, wenn für alle Belegungen  $\eta$  gilt  $\llbracket \phi \rrbracket \eta = \text{true}$ .

## Beispiele für allgemeingültige Formeln

- $A \vee \neg A$
- $(A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$
- $\neg \neg A \Rightarrow A$
- $A \wedge B \vee A \wedge \neg B \vee \neg A \wedge B \vee \neg A \wedge \neg B$

# Erfüllbare Formeln

## Erfüllbarkeit

Eine Formel  $\phi$  ist **erfüllbar** (satisfiable), wenn es eine Belegung ihrer Variablen gibt, die sie erfüllt (wahr macht). Formal heißt das, dass eine Belegung  $\eta$  existiert, derart, dass  $\llbracket \phi \rrbracket \eta = \text{true}$ .

## Unerfüllbarkeit

Eine Formel  $\phi$  ist **unerfüllbar** (unsatisfiable), wenn sie nicht erfüllbar ist.

## Beispiele erfüllbarer Formeln

- $A$
- $A \wedge B$
- $(A \vee B) \wedge (\neg A \vee B) \wedge (A \vee \neg B)$



# Äquivalenz

## Äquivalenz

Zwei Formeln  $\phi$  und  $\psi$  sind *äquivalent*, wenn für alle Belegungen  $\eta$  gilt:  $\llbracket \phi \rrbracket \eta = \llbracket \psi \rrbracket \eta$

## Beispiele äquivalenter Formeln

- $\phi \vee \psi$  ist äquivalent zu  $\psi \vee \phi$
- $\phi \Rightarrow \psi \Rightarrow \rho$  ist äquivalent zu  $\neg(\phi \wedge \psi) \vee \rho$

# Erfüllbarkeitsäquivalenz

## Erfüllbarkeitsäquivalenz

Zwei Formeln  $\phi$  und  $\psi$  sind *erfüllbarkeitsäquivalent*, wenn gilt:  $\phi$  erfüllbar gdw.  $\psi$  erfüllbar.

## Beispiele erfüllbarkeitsäquivalenter Formeln

- $A \vee B$  ist erfüllbarkeitsäquivalent zu  $\top$
- $\phi \vee \psi$  ist erfüllbarkeitsäquivalent zu  $(\neg A \vee \phi) \wedge (A \vee \psi)$

# Zusammenhänge

## Satz

- $\phi$  ist allgemeingültig, gdw.  $\neg\phi$  unerfüllbar ist.
- $\phi$  ist allgemeingültig, gdw.  $\phi$  äquivalent zu  $\top$  ist.
- $\phi$  ist erfüllbar, gdw.  $\phi$  erfüllbarkeitsäquivalent zu  $\top$  ist.

Beweis der ersten Aussage (Rest Übung):

$\phi$  ist allgemeingültig

gdw.  $\llbracket \phi \rrbracket \eta = \text{true}$  für jede Belegung  $\eta$

gdw.  $\llbracket \neg\phi \rrbracket \eta = \text{false}$  für jede Belegung  $\eta$

gdw. Es gibt keine Belegung  $\eta$  mit  $\llbracket \neg\phi \rrbracket \eta = \text{true}$

gdw.  $\neg\phi$  ist unerfüllbar

# Mengennotation

Abkürzung  $\bigwedge_{i \in I} \phi_i = \bigwedge \{ \phi_i \mid i \in I \} = \phi_{i_1} \wedge \cdots \wedge \phi_{i_n}$ , wenn  $I = \{i_1, \dots, i_n\}$ .

Analog  $\bigvee_{i \in I} \phi_i$ .

Leere Konjunktion:  $\bigwedge \emptyset = \top$ .

Leere Disjunktion:  $\bigvee \emptyset = \perp$ .

# Beispiel Sudoku

Für alle  $i, j \in \{0, \dots, 8\}$  und  $k \in \{1, \dots, 9\}$  führen wir eine Variable  $x_{ijk}$  ein. Bedeutung von  $x_{ijk}$ : In Zeile  $i$ , Spalte  $j$  steht die Zahl  $k$ .

Folgende Formeln modellieren die Sudoku Spielregeln.

- $\bigwedge_i \bigwedge_j \bigvee_k x_{ijk}$  (alle Felder sind ausgefüllt).
- $\bigwedge_i \bigwedge_j \bigwedge_{k \neq k'} \neg(x_{ijk} \wedge x_{ijk'})$  (nur eine Zahl pro Feld).
- $\bigwedge_i \bigwedge_{j \neq j'} \bigwedge_k \neg(x_{ijk} \wedge x_{ij'k})$  (nicht dieselbe Zahl zweimal in einer Zeile)
- $\bigwedge_{i \neq i'} \bigwedge_j \bigwedge_k \neg(x_{ijk} \wedge x_{i'jk})$  (nicht dieselbe Zahl zweimal in einer Spalte)
- $\bigwedge_{z \in \{0,1,2\}} \bigwedge_{w \in \{0,1,2\}} \bigwedge_k \bigwedge_{u,u' \in \{0,1,2\}} \bigwedge_{v,v' \in \{0,1,2\}}$   
 $\text{if } u \neq u' \mid \mid v \neq v' \text{ then } \neg(x_{3z+u,3w+v,k} \wedge x_{3z+u',3w+v',k}) \text{ else } \top$   
 (nicht dieselbe Zahl zweimal in einer 3x3 Box)

# Lösung eines Sudoku

Gibt man zu den so formulierten Spielregeln die schon besetzten Felder dazu (per  $\wedge$ ), so entsprechen die erfüllenden Belegungen gerade den Lösungen.

Beispiel:  $\bigwedge_{(i,j,k) \in I} x_{ijk}$  wobei

$$\begin{aligned} I = \{ & (0, 0, 6), (0, 4, 1), (0, 5, 7), (0, 6, 5), \\ & (1, 1, 8), (1, 2, 1), (1, 3, 2), (1, 7, 7), \\ & (2, 5, 5), \\ & (3, 1, 2), (3, 2, 9), (3, 3, 4), (3, 8, 1), \\ & (4, 1, 5), (4, 2, 4), (4, 4, 2), (4, 7, 3), \\ & (5, 2, 6), (5, 4, 7), (5, 5, 8), (5, 7, 5), (5, 8, 4), \\ & (6, 5, 9), (6, 6, 3), (6, 8, 7), \\ & (7, 2, 3), (7, 3, 8), (7, 6, 4), \\ & (8, 2, 5), (8, 7, 9) \} \end{aligned}$$

## Literale und Maxterme

### Literal

Ein *Literal* ist eine negierte oder nichtnegierte Variable. Ist  $\ell$  ein Literal, so definiert man  $\neg \ell$  durch  $\neg(A) = \neg A$  und  $\neg(\neg A) = A$ .

### Minterm, Und-Block

Eine Konjunktion (Ver-undung) von Literalen heißt *Minterm* oder *Und-Block*.

### Maxterm, Klausel

Eine Disjunktion (Ver-oderung) von Literalen heißt *Maxterm* oder *Klausel* oder *Oder-Block*.

# Konjunktive Normalform, Disjunktive Normalform

## Konjunktive Normalform, KNF

Eine Konjunktion von Klauseln (Oder-Blöcken) heißt *Konjunktive Normalform* (KNF).

## Disjunktive Normalform, DNF

Eine Disjunktion von Mintermen (Und-Blöcken) heißt *Disjunktive Normalform* (DNF).

## Satz von der DNF

Jede aussagenlogische Formel ist äquivalent zu einer DNF.

Beweis: Man führt je einen Minterm pro *true*-Zeile in der Wahrheitstafel ein.

Analoges gilt für die KNF (hier je ein Maxterm pro *false*-Zeile).



## Berechnung einer erfüllbarkeitsäquivalenten KNF

Die zu einer Formel äquivalente KNF kann sehr groß sein, im schlimmsten Fall exponentiell.

Besteht man nur auf Erfüllbarkeitsäquivalenz, so geht es effizienter:

### Satz

Zu jeder Formel  $\phi$  kann in polynomieller Zeit eine KNF  $\Gamma$  linearer Größe (bezogen auf die Größe von  $\phi$ ) berechnet werden, für die gilt:

- $\Gamma$  und  $\phi$  sind erfüllbarkeitsäquivalent.
- Jede Belegung, die  $\Gamma$  erfüllt, erfüllt auch  $\phi$ .

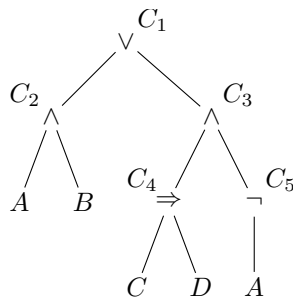
Man definiert rekursiv eine Funktion  $KNF$ , die zu jeder Formel  $\phi$  eine Variable  $A$  und eine KNF  $\Gamma$  linearer Größe liefert, so dass jede erfüllende Belegung von  $\Gamma$  auch  $A \Leftrightarrow \phi$  erfüllt.

Die gesuchte KNF ist dann  $A \wedge \Gamma$ .

## Beweisidee

Idee: Führe für jeden inneren Knoten des Syntaxbaums eine neue Variable ein. Erzwing die richtigen Werte durch eine Konjunktion von “kleinen” Formeln (mit je höchstens 3 Variablen).

Beispiel:  $(A \wedge B) \vee ((C \Rightarrow D) \wedge (\neg A))$



Erfüllbarkeitsäquivalente Formel:

$$\begin{aligned} C_1 \wedge (C_1 \iff C_2 \vee C_3) \wedge (C_2 \iff A \wedge B) \wedge (C_3 \iff C_4 \wedge C_5) \\ \wedge (C_4 \iff C \Rightarrow D) \wedge (C_5 \iff \neg A) \end{aligned}$$

## Beweisidee

Bringe die Teilformeln  $(C_1 \iff C_2 \vee C_3), \dots$  in KNF.

Da jede Teilformel höchstens drei Variablen hat, erhält man je höchstens acht Klauseln. Die Größe ist insgesamt linear in der Größe des Syntaxbaums.

$$\begin{aligned} C_1 \wedge (\neg C_1 \vee C_2 \vee C_3) \wedge (C_1 \vee \neg C_2) \wedge (C_1 \vee \neg C_3) \\ \wedge (\neg C_2 \vee A) \wedge (\neg C_2 \vee B) \wedge (\neg C_2 \vee A \vee B) \\ \wedge (\neg C_3 \vee C_4) \wedge (\neg C_3 \vee C_5) \wedge (\neg C_3 \vee C_5 \vee C_5) \\ \wedge (\neg C_4 \vee \neg C \vee D) \wedge (C_4 \vee C) \wedge (C_4 \vee \neg D) \\ \wedge (C_5 \vee A) \wedge (\neg C_5 \vee \neg A) \end{aligned}$$

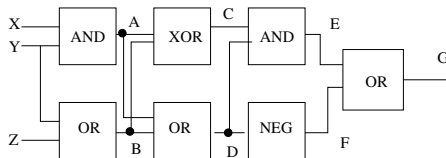
## Definition der Funktion $KNF$ (Tseitin Übersetzung)

- $KNF(A) = (A, \top)$  (leere KNF)
- $KNF(\neg\phi) = (C, \Gamma \wedge (C \vee B) \wedge (\neg C \vee \neg B))$  falls  $KNF(\phi) = (B, \Gamma)$ , wobei  $C$  eine frische Variable ist.
- $KNF(\phi_1 \vee \phi_2) = (C, \Gamma_1 \wedge \Gamma_2 \wedge (\neg C \vee B_1 \vee B_2) \wedge (\neg B_1 \vee C) \wedge (\neg B_2 \vee C))$ , falls  $KNF(\phi_1) = (B_1, \Gamma_1)$  und  $KNF(\phi_2) = (B_2, \Gamma_2)$ , wobei  $C$  eine frische Variable ist.
- Rest als Übung.

NB:  $\neg A_1 \vee \neg A_2 \vee \dots \vee \neg A_m \vee B_1 \vee \dots \vee B_n$  ist äquivalent zu  $(\bigwedge_i A_i) \Rightarrow (\bigvee_j B_j)$ .

NB: Es bietet sich an, mit dynamischer Programmierung dafür zu sorgen, dass identische Teilformeln nur einmal verarbeitet werden. Das ist insbesondere wichtig, wenn Teilformeln ge"shart" sind (Schaltkreise).

# Beispiel Schaltkreise



$$\begin{aligned}
 & (A \vee \neg X \vee \neg Y) \wedge (\neg A \vee X) \wedge (\neg A \vee Y) \wedge \\
 & (\neg B \vee Y \vee Z) \wedge (B \vee \neg Y) \wedge (B \vee \neg Z) \wedge \\
 & (\neg C \vee A \vee B) \wedge (\neg C \vee \neg A \vee \neg B) \wedge (C \vee \neg A \vee B) \wedge (C \vee A \vee \neg B) \wedge \\
 & (\neg D \vee A \vee B) \wedge (D \vee \neg A) \wedge (D \vee \neg B) \wedge \\
 & (E \vee \neg C \vee \neg D) \wedge (\neg E \vee C) \wedge (\neg E \vee D) \wedge \\
 & (\neg F \vee \neg D) \wedge (F \vee D) \wedge \\
 & (\neg G \vee E \vee F) \wedge (G \vee \neg E) \wedge (G \vee \neg F) \wedge \\
 & G
 \end{aligned}$$

# Das SAT-Problem

Das *SAT-Problem* besteht darin, für eine gegebene KNF zu entscheiden, ob sie erfüllbar ist.

Das SAT-Problem ist bekanntlich NP vollständig. Dennoch lässt es sich in vielen Fällen mittlerer Größe ( $< 10^6$  Variablen und Klauseln) praktisch lösen.

Dazu gibt es als *SAT-Solver* bezeichnete Programme, z.B. zChaff, Minisat, Picosat, ....

SAT-Solver geben nicht nur “erfüllbar” oder “unerfüllbar” als Antwort. Bei erfüllbaren Formeln wird auch eine erfüllende Belegung als Zeuge ausgegeben.

# SAT-Solver

SAT-Solver erwarten eine KNF als Eingabe.

- Mit der Tseitin-Übersetzung können wir für jede Formel  $\phi$  eine nicht zu große KNF ausrechnen.
- Gibt der SAT-Solver eine erfüllende Belegung der KNF zurück, so macht diese auch  $\phi$  wahr.

Das Äquivalenzproblem lässt sich ebenfalls auf SAT reduzieren:  
Zwei Formeln  $\phi$  und  $\psi$  sind äquivalent, genau dann wenn  
 $\neg(\phi \Leftrightarrow \psi)$  unerfüllbar ist.

## DIMACS-Format

SAT-Solver akzeptieren eine KNF in einem einfachen standardisierten Format (“DIMACS-Format”).

- Variablen werden durch Integer  $> 0$  repräsentiert; negierte Variablen durch entsprechende negative Zahlen.
- Klauseln werden durch 0-terminierte Zeilen, die ihre Literale enthalten, repräsentiert.
- Außerdem ist in einem Header die Zahl der Variablen und Klauseln anzugeben.

Beispiel für  $(\neg A \vee B \vee C) \wedge (A \vee \neg B \vee D \vee E)$ :

c Kommentarzeile.

p cnf 5 2

-1 2 3 0

1 -2 4 5 0

Die Zuordnung von Variablen zu Zahlen muss man sich merken.

Hier  $A = 1, B = 2, C = 3, D = 4, E = 5$ .



# DPLL-Algorithmus

Aktuelle SAT-Solver verwenden den *DPLL-Algorithmus* (Davis, Putnam, Logemann, Loveland).

Prinzip: “Vorrechnen” und wenn das nicht hilft, verzweigen.

Formal kann DPLL als rekursive Prozedur  $DPLL(\eta, \Gamma)$  beschrieben werden, die zu gegebener Belegung  $\eta$  und KNF  $\Gamma$  entweder eine erfüllende Belegung für  $\Gamma$  liefert, die  $\eta$  erweitert, oder “UNSAT” liefert, wenn es keine solche gibt.

# DPLL

$DPLL(\eta, \Gamma) =$

1. Vereinfache  $\Gamma$  und  $\eta$  mit *Unit Propagation* zu  $\Gamma'$  und  $\eta'$ .
2. Wenn  $\eta'$  alle Variablen in  $\Gamma'$  mit Werten belegt, dann gib  $\eta'$  als Ergebnis zurück.
3. Wähle andernfalls eine beliebige von  $\eta'$  noch nicht belegte Variable  $X$  aus und berechne rekursiv  $DPLL(\eta'[X \mapsto \text{false}], \Gamma')$ . Ist das Ergebnis eine Belegung, so wird sie zurückgegeben.  
Ist das Ergebnis UNSAT, dann probiere noch  $DPLL(\eta'[X \mapsto \text{true}], \Gamma')$ .  
Waren die rekursiven Aufrufe beide nicht erfolgreich, so wird UNSAT zurückgegeben.

N.B.  $\eta[X \mapsto \text{true}]$  bezeichnet die Belegung, die  $X$  auf *true* setzt und alle anderen Variablen wie  $\eta$  belegt.

# Unit Propagation

Funktionsweise der Unit Propagation:

- 1a. Vereinfache alle Klauseln in  $\Gamma$  gemäß  $\eta$ :
  - Entferne aus den Klauseln alle Literale, die durch  $\eta$  falsch gemacht werden.
  - Entferne alle Klauseln, die ein wahres Literal enthalten.
- 1b. Falls  $\Gamma$  die leere Klausel enthält, dann gib UNSAT zurück.
- 1c. Falls  $\Gamma$  die Einerklausel  $A$  enthält, so setze  $\eta(A) = \text{true}$  und beginne wieder bei Schritt 1a.
- 1d. Falls  $\Gamma$  die Einerklausel  $\neg A$  enthält, so setze  $\eta(A) = \text{false}$  und beginne wieder bei Schritt 1a.

## Beispiele für Unit Propagation

- $\Gamma = (A) \wedge (B \vee \neg C)$  und  $\eta(A) = \text{false}$

In 1a wird  $A$  aus der ersten Klausel entfernt. Es entsteht die leere Klausel, also wird in 1b. UNSAT zurückgegeben.

- $\Gamma = (\neg A) \wedge (B \vee \neg C)$  und  $\eta(A) = \text{false}$

In 1a wird die ganze erste Klausel entfernt. Das Ergebnis der Unit Propagation ist  $\Gamma' = (B \vee \neg C)$ ,  $\eta' = \eta$ .

- $\Gamma = (\neg A \wedge B) \wedge \dots$  und  $\eta(A) = \text{false}$ .

In Schritt 1a wird  $(\neg A \wedge B)$  zu  $(B)$ .

In Schritt 1c wird  $\eta(B) = \text{true}$  gesetzt.

Die Unit Propagation wird dann neu begonnen.

## Beispiel für DPLL

Ausführung von DPLL mit  $\eta = [X \mapsto \text{false}, A \mapsto \text{false}]$  und

$$\Gamma = (A \vee B \vee C) \wedge (A \vee B \vee \neg C) \wedge (A \vee \neg B \vee C \vee X) \wedge \\ (A \vee \neg B \vee \neg C \vee \neg X) \wedge (\neg A \vee \neg B \vee C \vee X)$$

- Unit Propagation liefert  $\eta' = \eta$  und

$$\Gamma' = (B \vee C) \wedge (B \vee \neg C) \wedge (\neg B \vee C) .$$

- Der Aufruf  $DPLL(\eta[B \mapsto \text{false}], \Gamma')$  gibt UNSAT zurück.
- Der Aufruf  $DPLL(\eta[B \mapsto \text{true}], \Gamma')$  liefert eine Belegung  $\eta''$  mit  $\eta''(X) = \text{false}$  und  $\eta''(A) = \text{false}$  und  $\eta''(B) = \text{true}$  und  $\eta''(C) = \text{true}$ , die als erfüllende Belegung zurückgegeben wird.

## DPLL Algorithmus: Anmerkungen

- Vorsicht: Bei der Implementierung kann man  $\Gamma$  nicht als globale Variable speichern, da sonst der Aufruf  $DPLL(\Gamma, \eta[X \mapsto false])$  das  $\Gamma$  modifiziert und es dann nicht mehr für den Aufruf  $DPLL(\Gamma, \eta[X \mapsto true])$  zur Verfügung steht.
- Bei rekursiver Implementierung werden die  $\Gamma$ s und  $\eta$ s auf dem Stack abgelegt. In der Praxis implementiert man diesen Stack explizit und speichert auf diesem nur jeweils die Änderungen ab.

# Optimierungen

- Anstatt explizit eine neue KNF  $\Gamma'$  zu erzeugen, merkt man sich nur für jede Klausel in  $\Gamma$ , welche Literale bereits gesetzt sind. Um Einerklauseln für neue Setzungen schnell zu finden, “beobachtet” jede Klausel die Variablen zweier ihrer unbesetzten Literale (*watched literals*). Wird eines von diesen gesetzt, so wird ein neues watched literal gesucht. Gibt es keines, so liegt eine Einerklausel vor ( $\leadsto$  Unit Propagation).
- Bei einem Konflikt (UNSAT) wird ermittelt, welche Variablensetzungen für den Konflikt verantwortlich waren und als neue “gelernte” Klausel der KNF hinzugefügt. War z.B.  $X = true, Y = false, Z = false$  verantwortlich, so “lernt” man  $\neg X \vee Y \vee Z$ .  
Man speichert hierzu bei jeder Setzung den entsprechenden Grund mit ab.

## Beispiel

$$\Gamma = (A \vee B \vee C) \wedge (A \vee B \vee \neg C) \wedge (A \vee \neg B \vee C \vee X) \wedge \\ (A \vee \neg B \vee \neg C \vee X) \wedge (\neg A \vee \neg B \vee C \vee \neg X)$$

Im DPLL-Algorithmus wird erst  $X = \text{false}$  und dann  $A = \text{false}$  und dann  $B = \text{false}$  gesetzt. Durch Unit Propagation erhalten wir aus den ersten beiden Klauseln einen Konflikt. Beteiligt waren nur die Setzungen  $A = B = \text{false}$ . Wir können daher die Klausel  $A \vee B$  lernen. Sie hilft uns später bei  $X = \text{true}$ .



## Anwendungen von SAT-Solvern

Auf der Vorlesungshomepage finden Sie Haskell- und Java-Programme, welche die Tseitin-Transformation, die Übersetzung ins DIMACS-Format und den Aufruf eines SAT-Solvers implementieren.

Das Haskell-Programm hat folgenden Datentyp für Formeln.

```
data Formula a = Var a
               | TT
               | FF
               | Neg (Formula a)
               | Or (Formula a) (Formula a)
               | And (Formula a) (Formula a)
               | Xor (Formula a) (Formula a)
               | Imp (Formula a) (Formula a)
               | Iff (Formula a) (Formula a)
               | BigOr [Formula a]
               | BigAnd [Formula a]
```

# Tseitin-Übersetzung und Sat-Solver in Haskell

Das Modul `Tseitin` implementiert die Tseitin-Übersetzung und kann auch gleich einen Sat-Solver aufrufen.

```
data Solution a
  = Unsatisfiable
  | Satisfiable (Map a Bool)
```

```
satisfiable :: Ord a => Formula a -> IO (Solution a)
```

Beim Ausführung von `satisfiable phi` wird `phi` mit der Tseitin-Übersetzung in KNF umgewandelt und es wird `Picosat` aufgerufen. Bei Unerfüllbarkeit wird `Unsatisfiable` zurückgegeben, sonst `Satisfiable(eta)`, wobei `eta` eine erfüllende Belegung ist.

# Beispiel

```
module Main where
import Formula
import Tseitin

phi :: Formula String
phi = And (Var "x") (Var "y")

main = do
  answer <- satisfiable phi
  case answer of
    Unsatisfiable ->
      putStr "phi ist nicht erfüllbar."
    Satisfiable eta ->
      putStr ("phi ist erfüllbar und zwar mit Belegung "
              ++ show eta)
```

# Compilierung und Ausführung

## Kompilieren und Ausführen:

```
$ ghc Main.hs  
$ ./Main
```

## Interaktiv:

```
$ ghci Tseitin.hs
```

```
*Tseitin> satisfiable (And (Var "x") (Var "y"))  
Satisfiable (fromList [("x",True),("y",True)])
```

```
*Tseitin> satisfiable (And (Var "x") (Neg (Var "x")))  
Unsatisfiable
```

# Sudoku

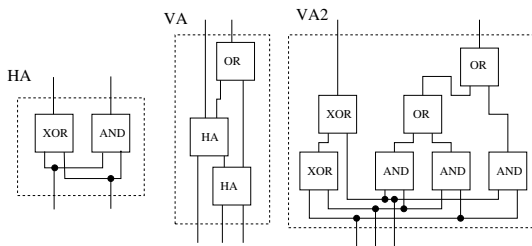
Die Sudoku-Formel kann in einem Programm erzeugt werden.

```
sudoku :: Formula (Int, Int, Int)
sudoku = BigAnd [ at_least, one_number_each_cell
                  , row_cond, col_cond,
                  , no_number_twice_in_square, givens ]
  where at_least =
        BigAnd [ BigOr [ Var (row, col, i) | i <- [1..9] ]
                | row <- [0..8], col <- [0..8] ]
  one_number_each_cell =
        BigAnd [ (Neg (Var (row, col, i)))
                `Or` (Neg (Var (row, col, j)))
                | row <- [0..8], col <- [0..8],
                  i <- [1..9], j <- [1..9], i < j ]
  ...
```

Der Aufruf `satisfiable sudoku` liefert eine Lösung des Sudokus oder sagt, dass das Problem unlösbar ist.

## Anwendung: Schaltkreisverifikation

Ein Schaltkreis ist ein DAG (gerichteter azyklischer Graph) dessen Knoten mit Junktoren beschriftet sind.



Um  $VA = VA2$  zu zeigen, beginnen wir mit drei Variablen  $X, Y, C$  und bilden die Formel

$$\neg((VA(X, Y, C)_1 \Leftrightarrow VA2(X, Y, C)_1) \wedge (VA(X, Y, C)_2 \Leftrightarrow VA2(X, Y, C)_2))$$

wobei  $VA(X, Y, C)_1$  den ersten Ausgang von  $VA(X, Y, C)$  bezeichnet, etc.

# Übersetzung in KNF

Tseitin-Transformation: KNF mit 37 Variablen, 58 Klauseln (UNSAT)

Ersetzt man das untere XOR von VA2 durch ein OR, so wird die entsprechende KNF erfüllbar. zChaff findet:

$X = Y = \text{true}, C = \text{false}$

# Gleichheit von Addierern

Halbaddierer:

```
ha x y = (x `Xor` y, x `And` y)
```

Volladdierer:

```
va x y c =  
  let (yc, d) = ha y c  
      (s, e) = ha x yc  
      cr = e `Or` d  
  in (s, cr)
```

Volladdierer (zweite Variante):

```
va2 x y c =  
  let s = (x `Xor` y) `Xor` c  
      cr = ((c `And` y) `Or` (y `And` x)) `Or` (c `And` x)  
  in (s, cr)
```

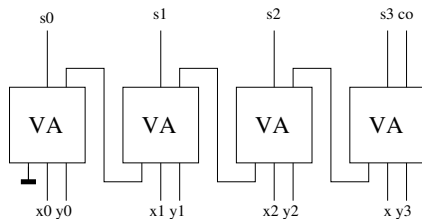


# Gleichheit von Addierern

```
equiv =  
  let (s1, c1) = va (Var "x") (Var "y") (Var "c")  
      (s2, c2) = va2 (Var "x") (Var "y") (Var "c")  
  in (s1 `Iff` s2) `And` (c1 `Iff` c2)  
  
main = do  
  answer <- satisfiable (Neg equiv)  
  case answer of  
    Unsatisfiable ->  
      putStr "equiv ist allgemeingültig."  
    Satisfiable eta ->  
      putStr ("equiv wird falsch durch die Belegung "  
              ++ show eta)
```

# Ripple-Addierer

Um Binärzahlen zu addieren, kann man Volladdierer hintereinanderschalten (Ripple-Adder):



Die Durchlaufzeit ist hier proportional zur Zahl  $n$  der addierten Bits (schlecht bei  $n = 64$ ).

Grund: Der Übertrag muss durch alle  $n$  Volladdierer laufen ("ripple").

## Carry-Lookahead Addierer

Günstiger ist es, den Übertrag  $c_i$  am  $i$ -ten Volladdierer als logische Funktion der Eingaben direkt zu berechnen.

carry, gen und prop

$$\begin{aligned}c_i &= \text{if } i=0 \text{ then } \perp \text{ else } gen_{0(i-1)} \\ gen_{ij} &= \text{if } i=j \text{ then } x_i \wedge y_i \text{ else } gen_{kj} \vee (gen_{i(k-1)} \wedge prop_{kj}) \\ prop_{ij} &= \text{if } i=j \text{ then } x_i \vee y_i \text{ else } prop_{i(k-1)} \wedge prop_{kj}\end{aligned}$$

wobei  $k = \lfloor (j - i + 1)/2 \rfloor + i$  der “Mittelpunkt” von  $i$  und  $j$  ist.

Idee:  $gen_{ij}$  — Am Ende des Blocks  $i \dots j$  entsteht ein Übertrag.  
 $prop_{ij}$  — Der Block  $i \dots j$  propagiert einen einlaufenden Übertrag.  
Da sich das Intervall jeweils halbiert, ist die Rekursionstiefe und damit die Tiefe der entstehenden Schaltkreise  $O(\log(n))$ .

# Implementierung in Haskell (1)

Den Ripple-Addierer implementiert man so:

```
rippleadd 0 = ([], FF)
rippleadd i =
  let (sumbits, c) = rippleadd (i - 1)
      (sum, c') = va (x (i - 1)) (y (i - 1)) c
  in (sum:sumbits, c')
```

Es gilt insbesondere:

```
rippleadd size = (co, [ssize-1, ..., s0]).
```

## Implementierung in Haskell (2)

Die gen- und prop-Schaltkreise als Formeln:

```
carry 0 = false
```

```
carry i = gen 0 (i - 1)
```

```
gen i j =
```

```
  if i == j then
```

```
    (x i) `And` (y i)
```

```
  else
```

```
    (gen k j) `Or` ((gen i (k-1)) `And` (prop k j))
```

```
    where k = (j + 1 - i) `div` 2 + i
```

```
prop i j =
```

```
  if i == j then
```

```
    (x i) `Or` (y i)
```

```
  else
```

```
    (prop k j) `And` (prop i (k-1))
```

```
    where k = (j + 1 - i) `div` 2 + i
```

## Implementierung in Haskell (3)

Lookahead-Addierer:

```
lookaheadadd 0 = ([], FF)
```

```
lookaheadadd i =
```

```
    let (ss, _) = lookaheadadd (i - 1)
```

```
        (s, d) = va2 (x (i - 1)) (y (i - 1)) (carry (i - 1))
```

```
    in (s:ss, d)
```

## Implementierung in Haskell (4)

Äquivalenz der Addierer:

```
equals size =  
  let (sumbits1, carryout1) = rippleadd size  
      (sumbits2, carryout2) = lookaheadadd size  
  in (carryout1 `Iff` carryout2) `And`  
      (AndL [s1 `Iff` s2 | (s1, s2) <- zip sumbits1 sumbits2])
```

Der Aufruf

```
satisfiable (Neg (equals 64))
```

prüft nun, ob die beiden 64-Bit-Addierwerke für *alle* Eingaben die gleichen Ergebnisse liefern: Die Formel (Neg (equals 64)) ist unerfüllbar gdw. (equals 64) allgemeingültig ist.

# Verifikation nebenläufiger Systeme

Wir wollen Eigenschaften von nebenläufigen Systemen überprüfen.

## Beispiel:

Wechselseitiger Ausschluss (Mutual Exclusion)

- $P$  Prozesse arbeiten nebenläufig.
- Die Prozesse teilen sich eine Resource.
- Es dürfen nicht zwei Prozesse gleichzeitig auf die Resource zugreifen.

Verschieden Lösungen:

- Semaphore
- Synchronisierung durch gemeinsame Variablen (z.B. Peterson Algorithmus)



# Modellierung nebenläufiger Systeme

Modellierung als Zustandsübergangssystem:

- Eine Menge  $Z$ , die Menge aller möglichen Systemzustände.
- Eine Menge  $I \subseteq Z$  von möglichen Anfangszuständen.
- Eine Relation  $\mapsto \subseteq Z \times Z$ . Die Aussage  $z \mapsto z'$  drückt aus, dass  $z' \in Z$  ein möglicher Nachfolgezustand von  $z \in Z$  ist.
- Eine Menge  $V \subseteq Z$  von verbotenen bzw. unerwünschten Zuständen.

Frage: Kann das System einen verbotenen Zustand erreichen?

$$I \ni z_0 \mapsto z_1 \mapsto \dots \mapsto z_n \in V$$

## Beispiel: Mutual Exclusion mit Semaphor

Es gibt  $k$  Prozesse mit Pseudocode

```
sleep: goto wait;  
wait:  if (sem == free) { sem = occ; goto work }  
work:  sem = free; goto sleep
```

sowie einen Semaphor mit zwei möglichen Zuständen *free* und *occ*.

**Modellierung:**

$$Z = \{(p_1, \dots, p_k, s) \mid p_i \in \{\text{sleep}, \text{wait}, \text{work}\}, s \in \{\text{free}, \text{occ}\}\}$$

$$I = \{(\text{sleep}, \dots, \text{sleep}, \text{free})\}$$

$$V = \{(p_1, \dots, p_k, s) \in Z \mid p_i = \text{work}, p_j = \text{work}, i \neq j\}$$

Zu jedem Zeitpunkt macht *genau ein* Prozess und möglicherweise der Semaphor eine Aktion. Möglich sind:

- $(\dots, \text{sleep}, \dots, s) \mapsto (\dots, \text{wait}, \dots, s)$
- $(\dots, \text{wait}, \dots, \text{free}) \mapsto (\dots, \text{work}, \dots, \text{occ})$
- $(\dots, \text{work}, \dots, s) \mapsto (\dots, \text{sleep}, \dots, \text{free})$

## Kodierung in Aussagenlogik

Man kann SAT-Solver dazu verwenden, um zu zeigen, dass ein System nach  $n$  Schritten keinen verbotenen Zustand erreicht:

Definiere dazu eine Formel, die genau dann erfüllbar ist, wenn es einen Pfad der Form  $I \ni z_0 \mapsto z_1 \mapsto \dots \mapsto z_t \in V$  für  $t \leq n$  gibt.

Wähle eine frische Variable  $q_{zt}$  für alle  $z \in Z$  und  $0 \leq t \leq n$ .

(Intention:  $q_{zt}$  sagt, dass  $z$  der  $t$ -te Zustand in einem schlechten Pfad ist.)

Klauseln:

- Für alle  $t$  ist genau ein  $q_{zt}$  wahr (vgl. Sudoku).
- $\bigvee_{z \in I} q_{z0}$
- $\bigwedge_{t=0}^{n-1} \bigvee_{z \mapsto z'} (q_{zt} \wedge q_{z'(t+1)})$
- $\bigvee_{t=0}^n \bigvee_{z \in V} q_{zt}$  (verbotener Zustand erreichbar)

## Kodierung in Aussagenlogik

Ist die Zustandsmenge durch Tupel gegeben, d.h.  $z = (z_1, \dots, z_\ell)$ , so kann man auch Variablen  $q_{1z_1t}, \dots, q_{\ell z_\ell t}$  statt  $q_{zt}$  einführen.

Bei vier Komponenten à 10 Möglichkeiten benötigt man dann  $10 + 10 + 10 + 10$  Variablen statt  $10 \cdot 10 \cdot 10 \cdot 10$ .

Die Intention von  $q_{iz_it}$  ist: Der  $t$ -te Zustand in einem schlechten Pfad hat  $z_i$  als  $i$ -te Komponente.

Die Aussage  $q_{zt}$  kann man durch  $q_{1z_1t} \wedge \dots \wedge q_{\ell z_\ell t}$  ausdrücken.

# Semaphor-Beispiel mit Aussagenlogik

## Variablen:

Für  $0 < p \leq k$  und  $z \in \{sleep, wait, work\}$  und  $t \leq n$  je eine Variable  $q_{pzt}$ . Außerdem für  $w \in \{free, occ\}$  und  $t \leq n$  je eine Variable  $s_{wt}$ .

## Klauseln:

- Für  $t < n$ ,  $0 < p \leq k$ , genau ein  $q_{pzt}$  und genau ein  $s_{wt}$  gesetzt.
- $\bigwedge_p q_{p \text{ sleep } 0} \wedge s_{free \ 0}$
- $\bigvee_t \bigvee_{p_1 \neq p_2} q_{p_1 \text{ work } t} \wedge q_{p_2 \text{ work } t}$
- $\bigwedge_{t < n} \bigvee_p \left( \bigwedge_{p_1 \neq p} \bigwedge_z q_{p_1 z t} \Leftrightarrow q_{p_1 z (t+1)} \right) \wedge$   
 $\bigvee_{(z, w, z', w') \in R} q_{pzt} \wedge q_{pz'(t+1)} \wedge s_{wt} \wedge s_{w'(t+1)}$

Hier ist  $R = \{(sleep, w, wait, w) \mid w \in \{free, occ\}\} \cup$   
 $\{(wait, free, work, occ)\} \cup \{(work, w, sleep, free) \mid w \in \{free, occ\}\}.$

# Peterson Algorithmus

Semaphor erfordert atomares Abprüfen und Setzen.

Ohne solche Unterstützung funktioniert der Peterson Algorithmus.

Zwei Prozesse 0,1. Jeder hat ein Flag  $flag[i]$  und es gibt eine Variable  $turn : \{0, 1\}$ , die angibt, wer dran ist.

Möchte ein Prozess den kritischen Bereich betreten, so signalisiert er das durch Setzen seines Flags.

Sodann setzt er  $turn$  auf den jeweils anderen Prozess und wartet dann ("busy wait"), bis er an der Reihe ist oder das Flag des anderen nicht gesetzt ist.

```
p:  0 flag[p] := 1; goto 1
    1 turn := other(p); goto 2
    2 if flag[other(p)] goto 3 else goto 4
    3 if turn = p goto 4 else goto 2
    4 (* kritischer Bereich hier *); flag[p] := 0; goto 0
```

# Peterson Algorithmus

- Variablen  $line_{p,i,t}$  für alle  $p \in \{0, 1\}, i \in \{0, 1, 2, 3, 4\}, t < n$ .  
*Zum Zeitpunkt  $t$  ist Prozess Nummer  $p$  in Zeile  $i$ .*
- Variablen  $flag_{f,p,t}$  für alle  $p, f \in \{0, 1\}, t < n$ .  
*Zum Zeitpunkt  $t$  hat  $flag[p]$  den Wert  $f$ .*
- Variablen  $turn_{p,t}$  für alle  $p \in \{0, 1\}, t < n$ .  
*Zum Zeitpunkt  $t$  hat  $turn$  den Wert  $p$ .*

Die Klauseln erzwingen, dass die Variablen die gewünschte Bedeutung haben, sowie dass zwei Prozesse den kritischen Bereich irgendwann gleichzeitig betreten (erfüllbar gdw. System inkorrekt).

- *sanity*: Zu jeder Zeit ist jeder Prozess in genau einer Zeile, und die Variablen  $flag$  und  $turn$  haben genau einen Wert.
- *initial*: Anfangszustand des Systems
- *transitions*: Aufzählung aller möglichen Zustandsübergänge von Zeit  $t$  zu  $t + 1$ , für alle  $t < n$ .
- *undesired*: Ungewünschter Systemzustand