Sequentielle Verifikation, Property Checking

In diesem Kapitel...

- Berechnung der erreichbaren Zustände
- Beispiele für Eigenschaften
- Sprachen für Eigenschaften
  - CTL
- Beweisen von Eigenschaften

Definitionen

- Sei f: A → B eine Funktion mit Definitionsbereich A und Bildbereich B.
  - Das Gesamtbild von f ist:
    $range(f) := \{ y \mid \exists x \in A : f(x) = y \}$
  - Sei C ⊆ A. Das Bild von C unter f ist definiert als
    $img(f, C) := \{ y \mid \exists x \in C : f(x) = y \}$
  - Sei C ⊆ B. Das Urbild von C unter f ist definiert als
    $pre(f, C) := \{ x \mid \exists y \in C : f(x) = y \}$

Beispiel

- Sei f: $\{0, 1\}^2 \rightarrow \{0, 1\}^2$ die Funktion
  $f(a, b) := (a + b, a \oplus b)$

a	b	f1	f2
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

- $range(f) = \{ (0, 0), (1, 0), (1, 1) \}$
- Sei C = { (0, 0), (0, 1) }.
  $img(f, C) = \{ (0, 0), (1, 1) \}$,
  $pre(f, C) = \{ (0, 0) \}$.

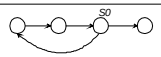
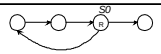
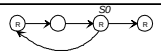
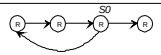
Erreichbare Zustände

- Berechne alle Zustände, die vom Anfangszustand aus erreichbar sind
- Zugrundeliegendes Automatenmodell: Mealy.
- Wichtige Operation in der Verifikation
  - Ausschließen nicht-erreichbarer Gegenbeispiele beim CEC oder beim bounded model checking
  - Überprüfen, ob eine Eigenschaft zu jedem Zeitpunkt gilt

Berechnung der erreichbaren Zustände

- Sei S eine Menge von Zuständen und $\delta : S \times I \rightarrow S$ die Zustandsübergangs-Funktion.
- Das Bild von $R \subseteq S$ bezüglich δ ist
  $image(R) = \{ s' \mid \exists\, i \exists\, s \in R : \delta(s, i) = s' \}$
- Beginnend von den Anfangszuständen wird solange das Bild berechnet und hinzugenommen, bis ein Fixpunkt erreicht wurde
  - Es muss einen Fixpunkt geben, da die Menge der Zustände endlich ist

Beispiel: Fixpunktiteration

R := *false*	
R := S_0	
R := R ∪ *image*(R)	
R := R ∪ *image*(R)	

Fixpunktiteration

- *compute_reachable_states*(S_0)

```
{
    R := S0;  // Anfangszustand
    do {
        R' := R;
        R := R ∪ image(R);
    } while (R' ≠ R);

    return R;
}
```

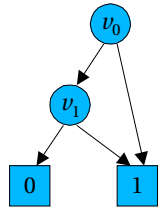
Probleme

- Explizites Auflisten der möglichen Zustände ist in der Praxis meist unmöglich („state explosion“)
- Ausweg: *implizite* (symbolische) Darstellung von Zustandsmengen mit BDDs
  - Charakteristische Funktionen von Zustandsmengen
  - Zustandsbits entsprechen Variablen der Funktion
  - Ein Zustand des Zustandsraumes entspricht einer Belegung der Variablen
  - Ein Zustand $s = (s1, ..., sk)$ ist in einer Zustandsmenge, wenn die charakteristische Funktion $\chi(s1, ..., sk) = 1$ ist.

Beispiel

- Variablen v0 und v1
- idle ≙ v0
- start ≙ ¬v0 · ¬v1
- busy ≙ ¬v0 · v1
- { idle, busy } ≙ v0 + v1
- { start, busy } ≙ ¬v0

	v0	v1
idle	1	x
start	0	0
busy	0	1



Die Übergangsrelation

- Die Übergangsrelation („transition relation“) eines endlichen Automaten ist definiert durch

$$TR(s, i, s') = [\delta(s, i) = s']$$
$$= \Pi_k: \delta_k(s, i) = s'_k$$

- Die charakteristische Funktion der Übergangsrelation kann als BDD dargestellt werden.

Bildberechnung

- Gesucht für eine Zustandsmenge S:
  $image(S) = \{ s' \mid \exists\, i \exists\, s \in S : \delta(s, i) = s' \}$.
- Berechnung mit BDDs:
  S: charakteristische Funktion der Bildmenge
  TR: charakteristische Fkt. der Übergangsrelation
  $image(S) = \exists\, i \exists\, s:\ (S(s) \cdot TR(s, i, s'))$
  Diese Operation wird *relationales Produkt* genannt.
- Oft ist es notwendig, für s und s' verschiedene Variablensätze zu verwenden und im Ergebnis des relationalen Produkts die Variablen von s' durch die von s zu ersetzen.

Erreichbarkeitsanalyse mit BDDs

- *compute_reachable_states(BDD S_0, BDD TR)*

```
{
    BDD R := S0;  // char. Fkt. Anfangszustand
    do {
        R' := R;
        Im := image(R);      // relationales Produkt
        Im' := Im |s'→s      // Variablensubstitution
        R := R ∪ Im;         //  „or“
    } while (R' ≠ R);

    return R;
}
```

alternativ: *image(*im letzten Schritt neu hinzugekommenen Zustände)

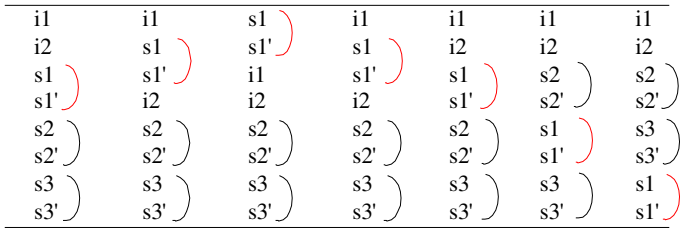
Probleme bei Bildberechnung

- *image(S)* = ∃ *i* ∃ s: (S(s) · TR(s, i, s'))
- Stellt man TR durch einen einzigen BDD dar („*monolithische Übergangsrelation*“), so ist dieser meist zu komplex
  - Der Ansatz scheitert bereits bei der Konstruktion der TR
- Wenn die TR konstruiert werden konnte, so werden meist die Zustandsmengen sehr groß

Variablenordnung

- Problem-spezifische Heuristik für eine „gute“ Variablenordnung der TR:
  - TR hängt von Eingangs-, Zustands- und Folgezustandsvariablen ab
  - Zustands- und korrespondierende Folgezustandsvariablen korrelieren stark
  - Heuristik: halte diese Variablenpaare immer benachbart (*state pair grouping*).
  - Sifting bewegt Paare von Variablen (*group sifting*).
  - Nebeneffekt: Variablensubstitution kann in O(1) erfolgen

Beispiel



Quantifizieren der Eingänge (1)

- Vertauschen von Existenzquantifizierung und Konjunktion
  - Im Allgemeinen gilt **nicht** (!):
    $\exists x: (f_1(x, y) \cdot f_2(x, y)) = (\exists x: f_1(x, y)) \cdot (\exists x: f_2(x, y))$
  - Die Situation ist anders, wenn f_2 nicht von x abhängt:
    $\exists x: (f_1(x, y) \cdot f_2(y)) = (\exists x: f_1(x, y)) \cdot f_2(y)$

Quantifizieren der Eingänge (2)

- $image(S) = \exists\, i \exists$ s: $(S(s) \cdot TR(s, i, s'))$
- Die Zustandsmenge S hängt nicht von den Eingängen i ab.
- Deshalb gilt:
  $image(S) = \exists$ s: $(S(s) \cdot (\exists\, i\, TR(s, i, s')))$
- Die Übergangsrelation kann also vereinfacht werden zu $TR(s, s') := \exists\, i\, TR(s, i, s')$, bevor die Bildberechnung beginnt.

Partitionierung der Übergangsrelation

- $TR(s, i, s') = [\delta(s, i) = s']$
  $= \Pi_k: \delta_k(s, i) = s'_k$
- Beobachtung: die Variablen des Nachfolgezustands s'_k hängen oft nur von einer kleinen Teilmenge der Zustandsvariablen s ab.
- Idee: Repräsentiere TR durch eine Liste von T Übergangsrelationen TR_t, $1 \le t \le T$, so dass
  $TR = TR_1 \cdot \ldots \cdot TR_T$.

Partitionierung der Übergangsrelation (2)

- Wie funktioniert damit die Bildberechnung?

  $image(S) = \exists\, i \exists$ s: $(S(s) \cdot TR_1(s, i, s') \cdot \ldots \cdot TR_T(s, i, s'))$
- Nachteile:
  - Es sind pro Bildberechnung mehr Operationen notwendig
  - Frühes Ausquantifizieren der Eingänge funktioniert nicht mehr

Early Quantification

- *image(S)* = $\exists\, i \exists$ s: (S(s) · TR_1(s, i, s') · ... · TR_T(s, i, s'))
- Quantifiziere eine Variable s_k zunächst in den TR_t, die von s_k abhängen und bilde erst danach die Konjunktion mit den TR_t, die von s_k nicht abhängen:
  - Support(TR_t): Variablen, von denen TR_t abhängt.
  - E: Variablen, bzgl. der quantifiziert werden soll
  - E_k: Variablen aus E, die in Support(TR_t) aber nicht in Support($TR_{t'}$) mit t' > t enthalten sind:

$$E_i = E \cap (Support(TR_t) \setminus \bigcup_{t'>t} Support(TR_{t'}))$$

Verbesserte Bildberechnung

- compute_image(BDD *S*, BDD TR_1, ..., BDD TR_T)

```
{
    result = S;
    for (t = 1; t < T; ++t)
    {
        result = ∃ v ∈ E_t: (result · TR_t);
    }
    return result;
}
```


Berechnung des relationalen Produkts

- *image(S)* = $\exists$ s: (S(s) · TR(s, s'))
- Ziel: Berechnung von *image*(S), ohne vorher S(s) · TR(s, s') zu berechnen.
- Betrachte $\exists$ e ∈ E: (f · g)
- Sei x die oberste Variable von f und g. Dann ist $\exists$ e ∈ E: (f · g) =

$$\begin{cases} (\exists e \in E : f_{x=1} \cdot g_{x=1}) + (\exists e \in E : f_{x=0} \cdot g_{x=0}) & falls\ x \in E \\ ite(x, \exists e \in E : f_{x=1} \cdot g_{x=1}, \exists e \in E : f_{x=0} \cdot g_{x=0}) & sonst \end{cases}$$


```
node* relprod(variable set E, node* F, node* G) {
  /* Terminalfälle, Computed Table */
  ...
  /* Rekursive Aufrufe */
  x = top_variable_of(F, G);
  low  = relprod(E, F_x', G_x');
  high = relprod(E, F_x, G_x);
  if (low == high) return low;
  if (x ∈ E)
    R = low + high;
  else {
    R = unique_table_find_or_add(x, low, high);
    /* update computed table */ ...
  }
  return R;
}
```

Approximate Reachability Analysis

- Wenn das alles nicht ausreicht...
- Approximieren der erreichbaren Zustände
- Über- und Unterapproximation
- Verschiedene Ansatzpunkte
  - Übergangsrelation
  - Zustandsmengen
  - Abstraktion der FSM
    - Kontroll-Logik

Rückwärtstraversierung

- Fragestellung ist ein bestimmter Zustand (eine bestimmte Zustandsmenge) S erreichbar?
- Statt der Bildberechnung muss eine Urbildberechnung erfolgen:
- Gesucht für eine Zustandsmenge S:
  $pre(S) = \{ s \mid \exists\, i \exists\, s' \in S : \delta(s, i) = s' \}$.
- Berechnung mit BDDs:
  $pre(S) = \exists\, i \exists\, s' :\ (S(s') \cdot TR(s, i, s'))$

Rückwärtstraversierung (2)

- *backward_reachability(BDD S, BDD TR)*
  {
  BDD $R := S$; // Zielzustand
  BDD $R_{new} := R$;
  do {
  $P := pre(R_{new}|_{s \to s'})$;
  $R_{new} := P \setminus R$;
  $R := R \cup R_{new}$;
  } while ($R_{new} \neq \emptyset$);
  return R;
  }

Einschub: Verallgemeinerter Kofaktor

- Definition:
  Sei $f: \{0,1\}^n \to \{0,1\}^m$ und $c: \{0,1\}^n \to \{0,1\}$.
  Die Funktion $f|_c$ heißt *verallgemeinerter Kofaktor* von f, falls

$$f|_c(x_1, \ldots, x_n) = \begin{cases} f(x_1, \ldots, x_n) & \text{falls } c(x_1, \ldots, x_n) = 1 \\ \text{beliebig} & \text{sonst} \end{cases}$$

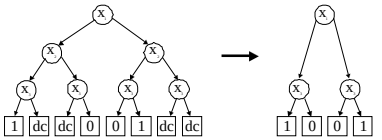
- Bemerkung 1: Die Definition ist nicht eindeutig.
- Bemerkung 2: Die bisherige Definition des Kofaktors ist ein Spezialfall dieser Definition

Verallgemeinerter Kofaktor (2)

- Der im Folgenden definierte verallgemeinerte Kofaktor hat das Ziel, die Größe des BDDs zu minimieren
- Bemerkung 3: Verallgemeinerte Kofaktoren können auch zur BDD-Minimierung eingesetzt werden, wenn *don't care*-Bedingungen gegeben sind.

Verallgemeinerter Kofaktor (3)

- Bei der Definition des verallgemeinerten Kofaktors wird die Funktion c als „*care*"-Funktion gesehen, für die der Funktionswert von f nicht verändert werden darf.
- Die don't care-Stellen werden so belegt, dass möglichst oft die Reduktionsregel zur Vermeidung unnötiger Abfragen angewendet werden kann:



Verallgemeinerter Kofaktor (4)

- Sei $c: \{0,1\}^n \to \{0,1\}$ mit $c \neq 0$. Dann ist die Abbildungsfunktion $\pi_c: \{0,1\}^n \to \{0,1\}^n$ definiert durch:

$$\pi_c(x) = \begin{cases} x & falls\ c(x) = 1 \\ MinDist(c, x) & falls\ c(x) = 0 \end{cases}$$

mit

$MinDist(c, x) = y :\Leftrightarrow$
$d(x, y) = \min_n \{ k \mid k = d(x, z) \text{ und } c(z) = 1 \}$ mit

$$d(x, y) := \sum_{i=1} |x_i - y_i| 2^{n-1}$$

- Die Funktion $f \circ \pi_c$ ist ein verallgemeinerter Kofaktor.

Verallgemeinerter Kofaktor (5)

- Satz:
Sei $f: \{0,1\}^n \to \{0,1\}^m$ und $\chi_C: \{0,1\}^n \to \{0,1\}$ die charakterist. Funktion einer Teilmenge $C \neq \emptyset$.
Dann gilt: $range(f|_{\chi C}) = img(f, C)$.
- Beweis:
Nach Def. von $\pi_{\chi C}$ ist $range(\pi_{\chi C}) = C$. Somit folgt
$range(f|_{\chi C}) = range(f \circ \pi_{\chi C}) = img(f, range(\pi_{\chi C})) = img(f, C)$.
- Satz: Seien $g: \{0,1\}^m \to \{0,1\}$, $f, c: \{0,1\}^n \to \{0,1\}$.
Dann gilt: $(g \circ f)|_C = g \circ f|_C$.

Verallgemeinerter Kofaktor (6)

```
cofactor(BDD f, BDD c) {
        assert(c != 0);
        // terminal cases
        if (c == 1 || isConstant(f)) return f;

        x = top_variable_of(f, c);
        if (c_{x=0} = 0) then return cofactor(f_{x=1}, c_{x=1});
        if (c_{x=1} = 0) then return cofactor(f_{x=0}, c_{x=0});
        return ite(x, cofactor(f_{x=1}, c_{x=1}),
                      cofactor(f_{x=0}, c_{x=0}));
}
```


Bildberechnung mit der Übergangsfunktion

- Idee: Bildberechnung ohne Verwendung der Übergangsrelation, d.h. nur mit der Übergangsfunktion
  - Vermeide die Konstruktion der (partitionierten) Übergangsrelation
  - Die charakteristische Funktion für das Bild wird direkt aus den Übergangsfunktionen δ_i konstruiert
  - Es genügt, das Gesamtbild einer Funktion berechnen zu können. Für das Bild einer Menge wird benutzt, dass $img(\delta, C) = range(\delta|_{\chi(C)})$.

Übergangsfunktion: Vorüberlegung 1

- Für eine Boolesche Funktion $f_k : \{0,1\}^n \rightarrow \{0, 1\}$ gilt:
  - Falls $f_k = 1$, dann gilt $range(f_k) = \{1\}$, also $\chi_{range(f_k)}(y_k) = y_k$.
  - Falls $f_k = 0$, dann gilt $range(f_k) = \{0\}$, also $\chi_{range(f_k)}(y_k) = \neg y_k$.
  - Falls f_k nicht konstant, dann gilt $range(f_k) = \{0, 1\}$, also $\chi_{range(f_k)}(y_k) = 1$.

Übergangsfunktion: Vorüberlegung 2

- Notation: $f_{k..m} := (f_k, f_{k+1}, .., f_m)$, $y_{k..m} := (y_k, y_{k+1}, .., y_m)$
- Falls $k < m$, so gilt:
  - Falls $f_k = 1$, dann gilt
    $$\chi_{range(f_{k..m})}(y_{k..m}) = (y_k \wedge \chi_{range(f_{k+1..m})}(y_{k+1..m}))(y_{k..m})$$
  - Falls $f_k = 0$, dann gilt
    $$\chi_{range(f_{k..m})}(y_{k..m}) = (\neg y_k \wedge \chi_{range(f_{k+1..m})}(y_{k+1..m}))(y_{k..m})$$
  - Falls f_k nicht konstant, dann gilt
    $$\chi_{range(f_{k..m})}(y_{k..m}) = (\neg y_k \wedge \chi_{img(f_{k+1..m}, OFF(f_k))}) \vee (y_k \wedge \chi_{img(f_{k+1..m}, ON(f_k))})$$
- Und: $\chi_{img(f_{k+1..m}, ON(f_k))} = range(f_{k+1..m}|_{f_k})$
  $\chi_{img(f_{k+1..m}, OFF(f_k))} = range(f_{k+1..m}|_{\neg f_k})$

Übergangsfunktion: Algorithmus

range(BDD $f_{k..m}$, int k, int m) {
    // terminal cases
    if ($k == m$) {
        if ($f_k == 1$) return y_k;
        if ($f_k == 0$) return $\neg y_k$;
        return 1;
    }

    if ($f_k = 1$) then return $y_k \wedge range(f_{k+1\,..\,m})$;
    if ($f_k = 0$) then return $\neg y_k \wedge range(f_{k+1\,..\,m})$;
    return $ite(y_k, range(f_{k+1\,..\,m} | f_k), range(f_{k+1\,..\,m} | f_k))$;
}

Rückwärtstraversierung mit der Übergangsfunktion

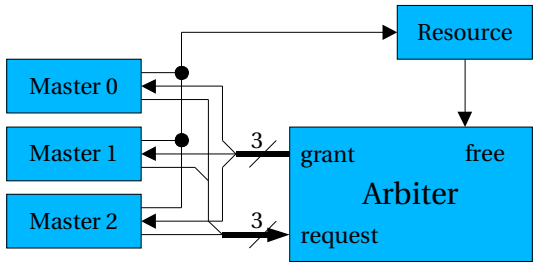
- Urbild kann durch Substitution bestimmt werden:
- *pre*(*R*) = substitute(R, current state, next state).
- Vorteil gegenüber Übergangsrelation:
  - keine Konstruktion der Relation notwendig
  - einfacher zu implementieren
- Nachteil:
  - Substitution kann teurer sein

Eigenschaften

- Im Folgenden: Verifikation durch Überprüfen von Eigenschaften auf einem Design
  - Beispiel
  - Sprachen für Eigenschaften
  - Überprüfen der Eigenschaften

Beispiel: Arbiter

- Kontrollierter Zugang von Computern zu einem Drucker



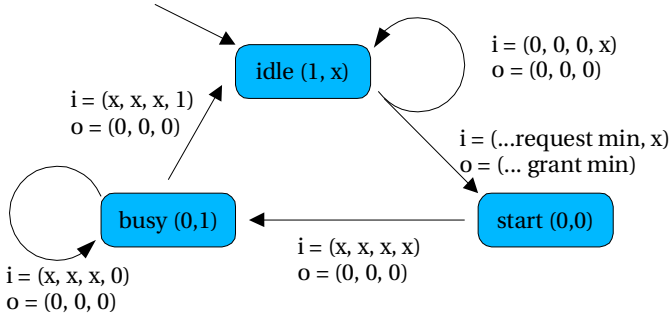
Arbiter (2)

- Prioritäten: request(1) > request(2) > request(3)
- Zwei Zustandsvariablen, drei Zustände

	inactive	active
idle	1	x
start	0	0
busy	0	1

Arbiter (3)

- min : kleinstes k mit request(k) = 1,
  x : ein Boolescher Wert



Arbiter (4)

- Notation
  - Zeitschritte t_0, t_1, ...
  - Werte der Variablen zur Zeit t: i_t, $(grant(2))_t$, ...
- Annahmen an die Umgebung
  - Jedem „grant“ folgt genau ein „free“
  - kein „free“ ohne „grant“
  - Die Resource benötigt höchstens 100 Takte zur Bearbeitung einer Aufgabe

Arbiter (5)

- „Bounded“ Properties
  (beziehen sich auf ein endliches Zeitfenster):
  - at t: $\Sigma\ (grant(i))_t \leq 1$. // höchstens ein grant
  - $(grant(1))_t$ => $(request(1))_t$ // grant nicht grundlos
  - $\neg inactive_t \wedge \neg active_t$ => $active_{t+1}$ // auf 'Start' folgt 'Busy'
  - $\neg free_t$ => within [t, t+100] $free_t$ // Resource braucht // höchstens 100 Takte
- „Unbounded“ Properties
  - Nach einem Request bekommt Master 2 die Resource irgendwann zugeteilt

Bounded Properties

Gegeben:

- ein endliches Zeitfenster $t_0, ..., t_n$
- Design D, als Mealy-Maschine dargestellt
  - Binäre Ein- und Ausgänge sowie Zustandsbits
  - Übergangs- und Ausgangsfunktionen
- Eigenschaft P, die sich auf Ein- und Ausgänge und Zustandsbits im endlichen Zeitfenster $[t_0, ..., t_n]$ bezieht

Bounded Properties (2)

- Syntax von P:
  - Assume: Boolean Comb. $\{ temp_r: R_r \}$
    Prove: Boolean Comb. $\{temp_s: S_s \}$
  - Temporale Operatoren
    $temp_i \in \{$ at t, during $[t_1, t_2]$, ... $\}$
  - Boolesche Funktionen R_r, S_s, die Ein- und Ausgänge und Zustandsbits in Beziehung setzen
- Semantik von P:
  - at t: R $\Leftrightarrow$ R (mit Variablen zur Zeit t)
  - within $[t_1, t_2]$: S $\Leftrightarrow$ S (zu einem Zeitpunkt t $\in [t_1, t_2]$)

Beispiel (Arbiter)

- Resource braucht höchstens 100 Takte:
  Assume: at t: (¬ free)
  Prove: within [t, t+100]: (free)
- Höchstens ein „Grant“:
  Assume: true
  Prove: at t: grant(1) + grant(2) + grant(3) $\leq$ 1

Überprüfen von Bounded Properties

Idee:

- P' = P'(x) ist eine Boolesche Funktion der Bits von $i_{t0}, ..., i_{tp}, s_{t0}$
- Eigenschaft P gilt auf Design D
  gdw. P' = 1 oder beginnend bei s_0 Eingabefolge
  $i_{t0}, ..., i_t$ ($t \leq tp$) bringt D in einen Zustand in welchem P' nicht gilt
  gdw. $\exists x : P'(x) = 0$.

Somit:

- $P' = 1 \Leftrightarrow \neg(\exists x : P'(x) = 0) \Leftrightarrow \neg(\exists x : (\neg P')(x) = 1)$
  (SAT Problem)

Verbesserungen

- Bessere Verfahren für SAT
- Spezielle Verbesserungen für Property Checking
  - Vereinfachen des Booleschen Terms ¬P', der durch die Eigenschaft P und das Design D definiert ist
    - Finde Paare von Sub-Termen, die identische Funktionen bezeichnen
    - Finde Tautologien
  - Verwende zusätzliches Wissen über P, D
    - gibt es Variablen, deren Wert ¬P' nicht beeinflusst,
    - Symmetrien,
    - ...

Unbounded Properties

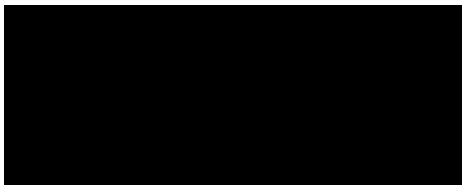
- Mit „bounded Properties“ kann nicht alles ausgedrückt werden
  - Nach einem Request bekommt Master 2 die Resource irgendwann zugeteilt
    - Diese Eigenschaft gilt nicht, wenn Master 1 höhere Priorität hat und ständig die Resource belegt ...

Modellierung der Zeit

- Verschiedene Möglichkeiten, Zeit zu modellieren
  - lineare Zeit vs. verzweigende Zeit
  - Zeitpunkte vs. Zeitintervalle
  - Diskrete vs. kontinuierliche Ze
  - Vergangenheit vs. Zukunft
- Hier:
  - verzweigende Zeit, diskrete Zeitpunkte, ausschließlich Zukünftige Aussagen

Berechnungsbaum

- Das „Abwickeln“ der temporalen Struktur entsprechend den Folgezuständen führt zu einem unendlichen Baum mit s_0 als Wurzel (Berechnungsbaum, computation tree)



Computation Tree Logic (CTL)

Definition CTL:

- Jede atomare Formel ist eine CTL-Formel.
- Sind p und q CTL-Formeln, dann auch
  - $\neg p$, $p \wedge q$, $p \vee q$
  - EX p, E(p U q), EG p, EF p, AX p, A(p U q), AG p, AF p
- A, E sind modale Operatoren (Pfadquantoren), F, G, U, X sind temporale Operatoren

CTL (2)

- Pfadquantoren:
  - A: „for all paths“, auf allen Berechnungspfaden, die in diesem Zustand beginnen
  - E: „there exists a path“, die Formel gilt auf einem Berechnungspfad, der in diesem Zustand beginnt
- Temporale Operatoren
  - X: „next“, im nächsten Zustand des Pfades
  - G: „globally“, in jedem Zustand des Pfades
  - F: „finally“, in mindestens einem Zustand
  - p U q: „until“, p gilt solange, bis q gilt.

CTL (3)

Abkürzende Schreibweisen

- EF p $:\Leftrightarrow$ E(*true* U p)
- AX p $:\Leftrightarrow \neg$EX($\neg p$)
- A(p U q) $:\Leftrightarrow \neg$E($\neg q$ U $\neg p \wedge \neg q$) $\wedge \neg$EG($\neg q$)
- AG p $:\Leftrightarrow \neg$EF ($\neg p$)
- AF p $:\Leftrightarrow \neg$EG ($\neg p$)

CTL – Beispiele

- Von einem Zustand aus kann man einen anderen Zustand erreichen, in dem p gilt, jedoch nicht q:
  EX EF ($p \wedge \neg q$)
- Wann immer eine Anfrage *req* erfolgt, dann erfolgt darauf irgendwann sicher eine Antwort *ack*:
  AG (*req* → AF *ack*)
- Die Berechnungsaufforderung *granted* gilt unendlich oft auf jedem Berechnungspfad:
  AG (AF *granted*)
- Von jedem Zustand im Berechnungsbaum ist ein Zustand erreichbar, in dem das *reset*-Signal gesetzt ist
  AG (EF *reset*)

CTL – Grenzen

- Auch mit CTL lässt sich nicht alles beschreiben
- Beispiel:
  Besitzt die Schaltung einen Reset-Zustand?

Vorgehensweise

- Sei *calc*(*f*) eine Prozedur zur Lösung dieser Aufgabe. Wir betrachten folgende Unterfunktionen:
  - calcEX(S_f) – *Auswertung von EX f bei bekanntem* S_f
  - calcEU(S_q, S_p, *Y*) – *Auswertung von E(q U p) bei bekannten* S_q *und* S_p *mittels Fixpunktiteration über Y*
  - calcEG(S_f, *Y*) – *Auswertung von EG f bei bekanntem* S_f *mittels Fixpunktiteration über Y*
- Die CTL-Formel wird rekursiv von innen nach außen berechnet. Alle Operationen werden auf Fixpunktiterationen basierend auf EX *y* zurückführt.


```
node* calc(formula f) {
  if ( f ist eine atomare Formel)    // S_f = { s ∈ S : s |= f };
      if ( f ist Eingangs- oder Zustandsvariable ) S_f = f;
      if ( f ist Ausgangsvariable ) S_f = λ_i;
  else if ( f ist von der Form ¬p)
      S_f = S \ calc(p);                    // ¬ calc(p)
  else if ( f ist von der Form (p ∨ q))
      S_f = calc(p) ∪ calc(q);              // calc(p) + calc(q)
  else if ( f ist von der Form EX p)
      S_f = calcEX(calc(p));
  else if ( f ist von der Form E(p U q))
      S_f = calcEU(calc(p), calc(q), false);
  else if ( f ist von der Form EG p)
      S_f = calcEG(calc(p), true);
  return S_f;
}
```

Unterprozeduren zur Auswertung einer CTL-Formel

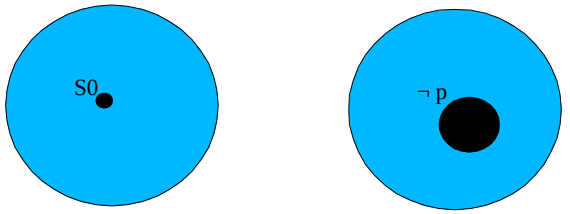
```
calcEX(S_p)
{
    return (alle unmittelbaren Vorgänger der Elemente von S_p);
}
```

```
calcEU(S_p, S_q, Y)  // E (p U q)
{
    Y_it := Y;
    do {
        Y := Y_it;
        Y_it := S_q ∪ (S_p ∩ calcEX(Y));
    } until (Y_it == Y)
    return Y;
}
```

```
calcEG(S_p, Y)  // EG p
{
    Y_it := Y;
    do {
        Y := Y_it;
        Y_it := S_p ∩ calcEX(Y));
    } until (Y_it == Y)
    return Y;
}
```


Optimierung für AG p

- Nachweis, dass p in allen erreichbaren Zuständen gilt
- Parallele Ausführung von Vorwärts- und Rückwärtstraversierung



Vorwärtstraversierung Rückwärtstraversierung

Überprüfen von Invarianten

- Wichtiger Spezialfall: Verifikation von Sicherheitseigenschaften (*safety properties*) durch Nachweis von Invarianten (*invariant checking*).
- Definition: Eine Zustandsmenge *Inv* eines endlichen Automaten *M* heißt *Invariante*, wenn sie im Startzustand gilt und alle von *Inv* aus erreichbaren Zustände in *Inv* enthalten sind.
  - $S_0 \subseteq Inv$
  - $image(Inv) \subseteq Inv$.

Beispiele für Invarianten

(1) Von S_0 aus erreichbare Zustände

(2) Beim Arbiter: es ist nie mehr als ein „grant“-Signal aktiv

(3) Ausgehend von einem Fehlerzustand, der nicht erreichbar sein soll, wird rückwärts traversiert, bis ein Fixpunkt erreicht wurde.
Das Komplement dieser Zustandsmenge ist Invariante, wenn der Initialzustand enthalten ist.

BDD vs. SAT: Logik

Symbolic Model Checking

- große Klasse von temporalen Anweisungen für „safety“ und „lifeness“
- Semantik der Eigenschaften schwierig
- Erlaubt das Beschreiben des Verhaltens in Bezug auf die Ein- und Ausgänge

SAT-based Model Checking

- beschränkt auf „safety“-Anweisungen, die sich auf 10-100 Takte beziehen
- Intuitive Semantik, Verifikationspläne
- Oft ist es notwendig, sich auf interne Zustände zu beziehen

BDD vs. SAT: Algorithmen

Symbolic Model Checking

- größerer Speicherbedarf
- Verifikations-Problem muss auf wenige 1000 Gatter / wenige 100 Zustandsbits abstrahiert werden
- Unvorhersagbare Berechnungszeiten
- Keine „false negatives“

SAT-based Model Checking

- geringer Speicherbedarf
- Designs mit 30K-100K Gattern und >10K Zustandsbits handhabbar
- Berechnungszeiten im Bereich Sekunden bis Minuten
- „false negatives“ sind zu verhindern

Weiterführende Literatur

- J.R. Burch *et al.*: Symbolic model checking for sequential circuit verification. IEEE Transactions on CAD, 1994.
- A. Narayan *et al.*: Reachability analysis using Partitioned-ROBDDs. Int'l Conference on Computer-Aided Design, 1997.
- B. Yang, R.E. Bryant *et al.*: A performance study of BDD-based model checking. Int'l Conference on Formal Methods in CAD, volume 1522 of LNCS, 1998.
- H. Higuchi, F. Somenzi: Lazy group sifting for efficient symbolic state traversal of FSMs. Int'l Conference on Computer-Aided Design, 1999.
- I.-H. Moon *et al.*: To split or to conjoin: the question of image computation. Design Automation Conference, 2000.