

# Partial Order Reduction

Seminar Systementwurf

Jan Sürmeli

# Problemstellung

- Explosion des Zustandraumes
- Modelchecking zu zeitaufwendig für zu große Zustandsräume
- ▶ Reduktion des Zustandsgraphen

# Womit beschäftigen wir uns?

- Nebenläufige Prozesse, die - unabhängig von ihrer Ausführungsreihenfolge - im selben Zustand enden.
- Beispiel: Tafel

# Die Idee

- Verhalten des reduzierten Graphen ist **Teilmenge** des vollen Graphen
- Finde solche Verhaltensweisen, die für den Model Checking Algorithmus **äquivalent** sind
- Reduziere den Graphen auf **Repräsentanten**

# Forderung

Ist eine Verhaltensweise nicht im reduzierten Graphen enthalten, dann muss ein **Repräsentant**, also eine äquivalente Verhaltensweise, enthalten sein.

# Vorgehensweise

- Erzeuge **direkt** den reduzierten Graphen
- ▶ Erzeuge **niemals** den vollen Graphen

# State Transition System

- Definition:

Ein STS ist ein 4-Tupel  $(S, T, S_0, L)$

- $S$  = Menge der Zustände
  - $S_0$  = Menge der Startzustände
  - $L$  = Beschriftungsfunktion
  - $T$  = Menge der Transitionen
- } Wie bei Kripke-Strukturen

# Transitionen: Definitionen

$$\forall \alpha \in T, \alpha \subseteq S \times S$$

$$\textit{enabled}(s) \Leftrightarrow \exists s' \alpha(s, s') \in T$$

$$\textit{disabled}(s) \Leftrightarrow \neg \textit{enabled}(s)$$

# Determinismus

- Eine Transition heißt **determiniert**, wenn in jedem Zustand  $s$  **maximal ein Zustand  $s'$**  existiert, so dass

$$\alpha(s, s') \in T$$

- Indeterminiert: *Sonst.*

# Determinismus

- Wir betrachten ab jetzt nur noch Systeme mit **determinierten Transitionen**
- Neue Schreibweise:

$$s' = \alpha(s) \text{ anstatt } \alpha(s, s')$$

# Auswahl der Transitionen

- **Fragestellung:** Welche Transitionen sollen übernommen werden, welche nicht?
- **Eigenschaften**
  - Abhängigkeit
  - Unsichtbarkeit

# (Un-)Abhängigkeit

- Englisch: (In-)Dependency
- **Unabhängigkeitsrelation**  $I \subseteq T \times T$ 
  - Symmetrisch
  - Antireflexiv
- Zwei Bedingungen müssen **in allen Zuständen** erfüllt sein:
  - Enabledness
  - Commutativity

# Enabledness

- Enabledness

$$\alpha, \beta \in \textit{enabled}(s) \Rightarrow \alpha \in \textit{enabled}(\beta(s))$$

- ▶ Deaktivieren sich nicht gegenseitig

# Commutativity

- Commutativity

$$\alpha, \beta \in \text{enabled}(s) \Rightarrow \alpha(\beta(s)) = \beta(\alpha(s))$$

- ▶ Die Ausführungsreihenfolge bestimmt nicht den Zielzustand

# Unabhängigkeit auf einen Blick

Falls für alle Zustände, in denen  $\alpha$  und  $\beta$  aktiviert sind gilt:

$$\left(\alpha \in \text{enabled}(\beta(s))\right) \wedge \left(\beta \in \text{enabled}(\alpha(s))\right)$$

...und...

$$\alpha(\beta(s)) = \beta(\alpha(s))$$

... sind  $\alpha$  und  $\beta$  unabhängig.

# Abhängigkeit

- Zwei Transitionen sind abhängig voneinander, wenn sie **nicht unabhängig** voneinander sind

$$D = (T \times T) \setminus I$$

# Unsichtbarkeit

- Eigenschaft von Transitionen
- Bezieht sich immer auf eine Teilmenge von atomaren Aussagen aller im System verwendeten atomaren Aussagen
- **Intuitiv:** Eine Transition ist unsichtbar bzgl. einer Menge von atomaren Aussagen, wenn sie den Wert der Aussagen nicht ändert.

# Unsichtbarkeit

## Formale Definition:

- Es seien:

$AP$  = Menge aller atomaren Aussagen

$$AP' \subseteq AP$$

- Für alle Zustände  $s, s'$  mit  $s' = \alpha(s)$  gilt:

$$L(s) \cap AP' = L(s') \cap AP'$$

# Sichtbarkeit

- Eine Transition ist sichtbar bzgl. einer Menge von atomaren Aussagen, wenn sie den Wert ändern kann.
- Eine Transition ist sichtbar **wenn sie nicht unsichtbar** ist.

# Pfade

- Ein Pfad von einem Zustand  $s_0$  ist eine Sequenz  $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots$
- Mit:  $\alpha_i(s_i, s_{i+1}) \in T, s_i \in S, i = 0, 1, 2, \dots$
- Die Länge eines endlichen Pfades ist die Anzahl der Transitionen darin

# Stuttering

- Stuttering: Englisch für „stotternd“
- Stuttering bezeichnet die Eigenschaft, dass in einem Pfad eine Sequenz von **identisch bezeichneten** Zuständen auftritt.

$$L(s_0) = L(s_1) = L(s_2) \dots$$

# Stuttering Equivalent

- Siehe Beispiel an der Tafel
- Die Sequenzen der Blöcke sind äquivalent.

# Stuttering Equivalent

**Schreibweise:**

$$\pi \sim_{st} \pi'$$

# Invariant under Stuttering

- Eine LTL-Formel  $f$  ist „invariant under stuttering“, wenn für alle stuttering equivalent Pfade gilt:

$$\pi \models f \Leftrightarrow \pi' \models f$$

# Stuttering und LTL<sub>-X</sub>

Betrachte LTL ohne den Next-Time-Op.

**Satz:**

Alle LTL<sub>-X</sub>-Eigenschaften sind  
**invariant under stuttering.**

# Übertragung auf Strukturen

- Bisher: Stuttering Equivalence nur auf Pfade bezogen.
- Jetzt: Betrachte zwei Strukturen
- Drei Bedingungen müssen erfüllt sein

# $\sim_{st}$ bei Strukturen

Zwei Strukturen  $M$  und  $M'$  sind genau dann stuttering equivalent, wenn gilt:

1.  $M$  und  $M'$  haben dieselbe Menge an Startzuständen
2. Für jeden Pfad  $\sigma$  aus  $M$ , der von einem Startzustand ausgeht, existiert in  $M'$  ein Pfad  $\sigma'$ , mit  $\sigma \sim_{st} \sigma'$
3. Analog zu 2. nur anders herum

# Wozu das Ganze?

- Stuttering Equivalent Strukturen sind für alle  $LTL_X$ -Eigenschaften invariant unter Stottern.
- Der Partial Order Reduction-Algorithmus erstellt einen zum vollen Graphen unter Stottern äquivalenten Graphen!

# Der POR Algorithmus

- Erstellt den reduzierten Zustandsgraphen
- Erstellt *niemals* den vollen Zustandsgraphen
- Arbeitet mit Tiefensuche

# POR-Algorithmus - Schritte

1. Beginne beim Startzustand
2. Bilde ein sog. *ample-Set* für den Zustand
3. Füge alle über die Transitionen aus dem *ample-Set* erreichbaren Zustände hinzu
4. Rekursiver Aufruf für diese Zustände ab Schritt 2
5. Erstelle die Kanten

# POR im Pseudocode

- 1 `set onStack(s0)`
- 2 `expandState(s0)`

Ein Zustand ist mit „onStack“ markiert, wenn er auf dem Tiefensuche-Stack liegt

# POR im Pseudocode

```
4 void expandState(State s)
5   workSet(s) := ample(s)
6   while not workSet(s).isEmpty
7     let  $\alpha \in$  workSet(s)
8     delete  $\alpha$  from workSet
9      $s' = \alpha(s)$ 
```

# POR im Pseudocode

```
10     if new(s') then
11         set onStack(s')
12         expandState(s')
13     end if
14     createEdge(s,  $\alpha$ , s')
15 end while
16 set completed(s)
17 end
```

# Das *ample*-Set

- Menge der „ausreichenden“ Transitionen in einem Zustand
- Teilmenge aller aktivierten Transitionen in einem Zustand
- Kern des POR-Algorithmus‘

# Erzeugung des *ample*-Sets

## Herangehensweise:

1. Stelle Bedingungen für *ample* auf
2. Finde Algorithmus, der *ample* hinreichend genau berechnet

## Hinweis:

Der Algorithmus muss *ample* **nicht** optimal errechnen. Die errechnete Menge muss korrekt sein, also die Bedingungen erfüllen.

# Bedingungen für *ample*: $C_0$

## Definition $C_0$ :

*Ample* ist nur dann für einen Zustand die leere Menge, wenn die Menge der aktivierten Transitionen leer ist und vice versa.

$$C_0 : ample(s) = \emptyset \Leftrightarrow enabled(s) = \emptyset$$

# Bedingungen für *ample*: $C_1$

## Definition $C_1$ :

Entlang eines jeden von  $s$  ausgehenden Pfades im **vollen Zustandsgraphen** gilt:

Eine Transition, die abhängig von einer Transition in *ample* ist, kann nicht ausgeführt werden, ohne dass eine Transition aus *ample* ausgeführt wurde.

# Folgerungen aus $C_1$

## **Lemma:**

Alle in  $enabled(s) \setminus ample(s)$  enthaltenen Transitionen sind unabhängig von denen in  $ample(s)$ .

## **Problem:**

Die Prüfung von  $C_1$  erfordert einen vollen Zustandsgraphen - Widerspruch! Lösung: Konstruiere so, dass die Bedingung erfüllt wird -> Kein optimaler reduzierter Zustandsgraph.

# Bedingungen für ample: $C_2$

- Wenn nicht alle aktivierten Transitionen eines Zustandes in *ample* übernommen werden, dann muss jede Transition in *ample* unsichtbar sein.
- Beispiel: Tafel

## Bedingungen für *ample*: $C_3$

Ein Zyklus ist nicht erlaubt, wenn in einem seiner Zustände eine Transition aktiviert ist, die in keiner der *ample*-Mengen der Zustände des Zyklus enthalten ist.

**Problem:** Ebenfalls global definiert

# Lösung für $C_3$

- Möglichkeit 1:
  - Erstelle den reduzierten Zustandsgraphen mit den Zyklen
  - korrigiere dann solange, bis  $C_3$  erfüllt ist.
- Möglichkeit 2:
  - Finde hinreichende Bedingung für  $C_3$
  - $C_3'$ : Wenn nicht alle Transitionen aus  $s$  verwendet werden, darf keine Transition aus  $ample(s)$  zu einem Zustand auf dem Search Stack führen.

# Algorithmus für *ample(s)*

- Überprüfe in jedem Zustand welche möglichen Prozesse existieren
- Überprüfe, ob dieser Prozess die Bedingungen erfüllt
- Falls ja: Gebe diesen Prozess zurück
- Falls kein Prozess die Bedingungen alleine erfüllt: Gebe alle aktivierten Transitionen zurück

# Heuristik für *ample(s)*

- + Niedrigere Komplexität
- + Schneller in der Ausführung
- + Leichter zu beweisen
- + Gutes Verhältnis zwischen Reduktionsgrad und Performance
- Kein optimaler Zustandsgraph

# Ausblick

- Im Vortrag beschrieben:  
Zweiphasenprozess
  1. Erstelle reduzierten Graphen
  2. Modelchecking
- On-the-Fly Partial Order Reduction während Modelchecking
- Vorteil: Bevor der reduzierte Graph vollständig aufgebaut wurde, kann gezeigt werden, dass er die Spezifikation nicht erfüllt
- SPIN von Bell Laboratories

# Kleine Statistik für SPIN auf *SGI Challenge*

	Non-reduced			Reduced		
# <i>P</i>	# <i>s</i>	Mem	Time	# <i>s</i>	Mem	Time
3	15929	1,801	13,8 s	1435	1,493	0,6 s
4	522.255	15.727	9,3 min	8475	1,698	3,5 s
5		> 128	> 40 h	57555	3,234	28,7 s
6				434083	15,625	4,1 min

Aus: *Clarke, Grumberg, Peled: Model Checking.*

**Vielen Dank**

**für Eure Aufmerksamkeit**

# Quelle

*Clarke, Grumberg, Peled:*

**Model Checking**

(MIT Press, 3. Auflage, 2001)