

# Konformitätstests reaktiver Systeme

15. November 2005

## Beispiele für reaktive Systeme:

"Verkaufsautomaten" (für Getränke, Fahrkarten, Parktickets, etc. )

"Benutzereingaben":

*Einwerfen von Geld, Knopf drücken, ...*

Reaktion des betrachteten Systems:

*Signaltöne, Lichtsignale (Blinken), Warenausgabe, ...*

## Beispiele für reaktive Systeme:

"Message-Loop" in GUI-Programmen

"Benutzereingaben":

*Betätigen der Maustasten, Betätigen von Buttons, Tastatureingaben,*

*...*

Reaktion des betrachteten Systems:

*Öffnen von Fenstern/Dialogen (Fehlermeldungen), allgemein Funktionsaufruf, ...*

## Beispiele für reaktive Systeme:

Protokolle im Bereich der Informatik, z.B. SMTP

"Benutzereingaben":

*Festgelegte Benutzerbefehle: HELO, MAIL, RCPT, DATA, RSET, NOOP, QUIT, ...*

Reaktion des betrachteten Systems:

*Senden der Mail, Fehlermeldungen (z.B. unzulässiger Befehl im aktuellen Zustand), Statusmeldungen, ...*

## Beispiele für reaktive Systeme:

Steuerwerke in der technischen Informatik

"Benutzereingaben":

*Anliegende Signale, wie Programmzähler für Mikroprogramme, ...*

Reaktion des betrachteten Systems:

*Steuersignale, um Signalpfade von Registern zu entsprechenden funktionalen Einheiten (ALU,FPU,...) festzulegen, ...*

# Problemstellung

Wir sind nicht an der eigentlichen Implementierung oder den intern ablaufenden Prozessen (hier) interessiert.

Wir wollen sicherstellen, dass die *Interaktion* zwischen Benutzer und "Implementierung" (wie Getränkeautomat, Mailserver, ...) der *Spezifikation entsprechend* verläuft, d.h. z.B.:

*der Getränkeautomat Kaffee ausgibt, wenn Kaffee verlangt wird.*

# Problemstellung

Wir sind nicht an der eigentlichen Implementierung oder den intern ablaufenden Prozessen (hier) interessiert.

Wir wollen sicherstellen, dass die *Interaktion* zwischen Benutzer und "Implementierung" (wie Getränkeautomat, Mailserver, ...) der *Spezifikation entsprechend* verläuft, d.h. z.B.:

*der richtige Dialog aufgerufen/geöffnet wird bei Klicken auf "Datei".*

# Problemstellung

Wir sind nicht an der eigentlichen Implementierung oder den intern ablaufenden Prozessen (hier) interessiert.

Wir wollen sicherstellen, dass die *Interaktion* zwischen Benutzer und "Implementierung" (wie Getränkeautomat, Mailserver, ...) der *Spezifikation entsprechend* verläuft, d.h. z.B.:

*RCPT nur direkt nach MAIL oder RCPT akzeptiert wird.*

# Problemstellung

Wir sind nicht an der eigentlichen Implementierung oder den intern ablaufenden Prozessen (hier) interessiert.

Wir wollen sicherstellen, dass die *Interaktion* zwischen Benutzer und "Implementierung" (wie Getränkeautomat, Mailserver, ...) der *Spezifikation entsprechend* verläuft, d.h. z.B.:

*die richtigen Steuersignale anliegen.*

# Problemstellung

Wir sind nicht an der eigentlichen Implementierung oder den intern ablaufenden Prozessen (hier) interessiert.

Wir wollen sicherstellen, dass die *Interaktion* zwischen Benutzer und "Implementierung" (wie Getränkeautomat, Mailserver, ...) der *Spezifikation entsprechend* verläuft, d.h. z.B.:

*Die Implementierung wird als black box betrachtet. Nur die Reaktionen der Implementierung auf Benutzereingaben sind beobachtbar.*

# Abstraktion/Modellierung

- ▶ Gemeinsamkeiten aller Beispiele:
- ▶ endliche Menge von Zuständen: eingeworfener Betrag, ausgegraute Knöpfe, ... generell gegenwärtig erlaubte Aktionen (z.B. darf RCPT erst nach MAIL kommen).
- ▶ endliche Menge von Eingaben: Geldeinwerfen, betätigter Button, MAIL, ...
- ▶ endliche Menge von Ausgaben: aktuell eingeworfenen Betrag anzeigen, Dialog aufrufen, ...
- ▶ für einen gegebenen Zustand bestimmt die Eingabe *eindeutig* (d.h. auf jede Eingabe wird reagiert) den Folgezustand und die Ausgabe:  
50 Cent eingeworfen, Benutzer wählt Kaffee (1 Euro), Fehlermeldung, alter Zustand

## Modellierung

- ▶ Die Ausgabe und der Folgezustand hängen *deterministisch* von der Eingabe und dem aktuellen Zustand ab (zumindest bei der Spezifikation).
- ▶ In jedem Zustand wird auf jede Eingabe in irgendeiner Weise reagiert (Fehlermeldung, Statusmeldung, Wechsel in Fehlerzustand, alten Zustand halten, ...), d.h. das Verhalten ist vollständig spezifiziert.
- ▶ Von jedem Zustand aus ist der Startzustand erreichbar: Geld ausgeben lassen, "Programm schließen", RSET, ...
- ▶ Jeder Zustand ist vom Startzustand erreichbar: Rest uninteressant/unnötig
- ▶ ein wohldefinierter Startzustand (z.B. initiale Variablenbelegung).
- ▶ Formales Modell für diese Art von Objekten: Deterministische Mealy-Maschinen (DMM)

# Deterministische Mealy-Maschinen (DMM) - formale Definition

Sowohl Spezifikation als auch Implementierung können als Mealy-Maschinen modelliert werden:

Eine *deterministische* Mealy-Maschine (DMM)  $\mathcal{D}$  ist ein Tupel  $\mathcal{D} = (Q, I, O, \delta, \lambda, q_0)$  mit

$Q$ : endliche Menge von Zuständen

$I$ : endliche Menge von (Benutzer-)Eingaben

$O$ : endliche Menge von (Maschinen-)Ausgaben

$\delta$ : (totale) Zustandsübergangsfunktion:  $\delta : Q \times I \rightarrow Q$

$\lambda$ : (totale) Ausgabefunktion:  $\lambda : Q \times I \rightarrow O$

$q_0$ : dem Anfangszustand

- ▶ Wie für endliche Automaten (TI 1) werden DMMs als gerichtete Graphen mit beschrifteten Kanten dargestellt:  $Q$  als Knoten,  $\delta$  und  $\theta$  als beschriftete Kanten

$$q \xrightarrow{i/\lambda(q,i)} \delta(q, i), \dots$$

- ▶ Für  $\sigma = i_1 \dots i_n \in I^*$  eine endliche Eingabesequenz ist

$$\hat{\delta}(q, \sigma) = \delta(\dots \delta(q, i_1) \dots, i_n)$$

der nach Einspielen von  $\sigma$  erreichte Zustand.

- ▶ Entsprechend ist

$$\hat{\lambda}(q, \sigma) = \lambda(q, i_1)\lambda(\delta(q, i_1), i_2) \dots \in O^*$$

die beim Einspielen von  $\sigma$  beobachtete Ausgabesequenz.

## DMM vs DFA

- ▶ Wir können jeden DMM durch einen DFA über dem Alphabet  $I \times O$  beschreiben: Der DFA akzeptiert nur Kombinationen von Eingabesequenzen und zugehörigen Ausgabesequenzen, d.h. nur Paare  $(\sigma, \hat{\lambda}(q_0, \sigma))$ :
  - ▶ Transitionen  $q \xrightarrow{i/o} q'$  als  $q \xrightarrow{(i,o)} q'$  lesen.
  - ▶ Jeden DMM-Zustand als Endzustand markieren.
  - ▶ Bei nicht-zulässiger Kombination von Eingabe-/Ausgabesymbol in Fehlerzustand wechseln.
- ▶ Ein DFA kann auch durch einen DMM simuliert werden: Nur zwei Ausgaben "Licht aus", "Licht an" (=akzeptiert). "Licht an" nur, falls DFA in Endzustand wechseln würde.

## Beispiel: Nachrichtenpuffer mit Prioritäten

Spezifikation eines Nachrichtenpuffers mit Speicherplatz für zwei Nachrichten, von dem Nachrichteninhalt wird abstrahiert:

- ▶ Ablegen von Nachrichten verschiedener Priorität, hier nur zwei Stufen: *normal*, *dringend*.

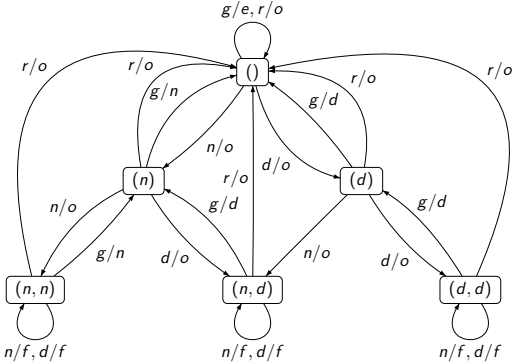
Hat der Puffer noch Platz für die Nachricht, so wird die Nachricht gespeichert und *ok* ausgegeben, ansonsten bleibt der Puffer unverändert und es wird *full* ausgegeben.

- ▶ Mittels *get* kann eine beliebige Nachricht höchster Priorität abgerufen werden.

Ist der Puffer leer, so wird *empty* ausgegeben; ansonsten: ist eine dringende Nachricht gepuffert, so wird *dringend* ausgegeben, sonst *normal*.

- ▶ Reset des Puffers: mittels *reset* wird der Puffer geleert und es wird *ok* ausgegeben.

# Beispiel: Nachrichtenpuffer mit Prioritäten



## Erweiterte DMMs (EDMM)

- ▶ Spezifikationen in Form von DMMs sind sehr umständlich, da jeder Zustand und jede Transition explizit angegeben werden muss:

Z.B. *reset* für Puffer der Größe 100: " Falls in () gib 'ok' aus und gehe nach ()", " Falls in (n) gib 'ok' aus und gehe nach ()", ...

- ▶ Für DMMs, deren Zustände die Form

(Kontrollzustand, Variablenbelegung)

haben, *wobei jede Variable nur einen endlichen Wertebereich besitzt*, dienen *Erweiterte DMMs* zur kompakten Repräsentation.

Beispiel: Der Puffer hat nur einen Kontrollzustand  $q_0$  und (z.B.) drei Variablen  $N_1, N_2, S$  mit  $N_i \in \{n, d\}$  und  $S \in \{0, 1, 2\}$

- ▶ Um zwischen den Zuständen einer EDMM und den Zuständen der von ihr repräsentierten DMM zu unterscheiden, sprechen wir in diesem Fall von *Konfigurationen* stellvertretend für DMM-Zustände.

## Erweiterte DMMs

- ▶ Die EDMM zu einer DMM fasst deren Konfigurationen bezüglich des Kontrollzustandes zusammen:  
Eine EDMM für den Puffer hätte somit nur einen Zustand  $[q_0]$ , welcher die Konfigurationsmenge  $\{q_0\} \times \{n, d\} \times \{n, d\} \times \{0, 1, 2\}$  repräsentiert.
- ▶ Der Startzustand muss jedoch immer noch explizit mit einer festen Variablenbelegung angegeben werden.  
Startzustand für den Puffer:  $(q_0, n, n, 0)$

## Erweiterte DMMs

- ▶ eine EDMM-Transition  $(q, i, g, a, o, q')$  ist um Bedingungen über den Variablenwerten (*guards*) und Zuweisungen an die Variablen (*actions*) erweitert:

q	Gegenwärtiger Kontrollzustand	z.B. $q_0$
i	Eingabe	z.B. get
g	Bedingung an Variablen ( <i>guard</i> )	z.B. $S = 1$
a	Aktion (Variablenzuweisungen)	z.B. $S = 0$
o	Ausgabe	z.B. $N_1$
q'	Neuer Kontrollzustand	z.B. $q_0$

- ▶ Die von einer EDMM repräsentierte DMM ergibt sich durch Betrachten *aller* Kombinationen von Kontrollzustand und Variablenbelegung und Anwenden der jeweiligen EDMM-Transition. ("Entrollen der EDMM")

Bsp.:  $(q_0, N_1 = d, N_2 = n, S = 1) \xrightarrow{g/d} (q_0, N_1 = d, N_2 = n, S = 0)$ .

## Erweiterte DMMs

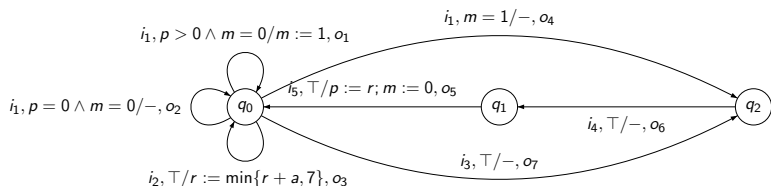
- ▶ Aus der Forderung, dass die repräsentierte DMM deterministisch ist, folgt für festen Kontrollzustand  $q$  und feste Eingabe  $i$ :  
Ist

$$G_{q,i} = \{g \mid (q, i, g, a, o, q') \text{ ist EDMM-Transition für } q \text{ und } i\}$$

die Menge alle guards für  $q$  und  $i$ ,  
dann muss jede Variablenbelegung genau einen der guards  $g \in G_{q,i}$  erfüllen, d.h.

- ▶ je zwei guards  $g, g' \in G_{q,i}$  sind disjunkt.
- ▶  $\bigvee_{g \in G_{q,i}} g \equiv \text{true}$

## Erweiterte DMMs - Bsp: Token Ring



Beispiel aus der Spezifikation eines Token-Ring-Protokolls.

Wertebereiche der Variablen:  $m \in \{0, 1\}$ ,  $p, r \in \{0, \dots, 7\}$ , d.h., die repräsentierte DMM hat 384 Zustände.

**Vereinbarung:** Ist in obiger EDMM in einem Zustand  $q_l$  für eine Eingabe  $i_k$  keine Transition spezifiziert, so verändert die EDMM ihren Zustand nicht und gibt eine beliebige "Fehlermeldung"  $d$  aus, d.h. wir gehen von einer Transition  $(q_l, i_k, \top, -, d, q_l)$  aus.

# Minimieren von (E)DMMs

- ▶ Minimieren der Spezifikation bzw. Implementierung durch Zusammenfassen *äquivalenter* Zustände.
- ▶ Zwei Zustände einer DMM  $q_1, q_2 \in Q$  heißen *äquivalent* (kurz:  $q_1 \sim q_2$ ), falls:

$$q_1 \sim q_2 :\Leftrightarrow \forall \sigma \in I^* : \hat{\lambda}(q_1, \sigma) = \hat{\lambda}(q_2, \sigma),$$

d.h., wenn sich die beiden Zustände unter allen möglichen Eingabesequenzen gleich verhalten.

## Minimieren von (E)DMMs

- ▶ Für  $q_1 \neq q_2$  existiert entsprechend eine Eingabesequenz  $\sigma \in I^*$ , so dass  $q_1$  und  $q_2$  verschiedene Ausgabesequenzen erzeugen. Ein solche Sequenz  $\sigma$  wird *Trennsequenz* für  $q_1$  und  $q_2$  genannt.

Beispiel: Im Fall des Puffers ist  $g(=get)$  eine Sequenz, welche den leeren Puffer  $()$  von allen nicht-leeren Puffern trennt.

- ▶ Zwei DMMs (über demselben Eingabealphabet  $I$ ) sind entsprechend (in)äquivalent, falls ihre Startzustände (in)äquivalent sind. Einfach beide DMMs nebeneinander legen und als eine große DMM auffassen.

## Minimierung von DMMs

Wir berechnen die Äquivalenzrelationen  $\sim_k$  mit:

$q \sim_k q'$  genau dann, wenn  $q$  und  $q'$  bzgl. aller Eingabesequenzen der Länge  $\leq k$  übereinstimmen.

- ▶ Bzgl. der "leeren" Eingabe  $\varepsilon$ , d.h. der Eingabe der Länge 0, sind alle Zustände ununterscheidbar, da keine Ausgabe produziert wird:

$$q \sim_0 q' :\Leftrightarrow \text{true}$$

- ▶ ( $k > 0$ ) Zwei Zustände  $q, q'$  sind bzgl. Eingabesequenzen  $\sigma = i_1 \dots i_s$  der Länge  $s \leq k$  ununterscheidbar, falls sie bzgl.  $i_1$  dieselbe Ausgabe erzeugen und in Zustände wechseln, welche bzgl. Eingabesequenzen der Länge  $\leq k - 1$  ununterscheidbar sind:

$$q \sim_k q' :\Leftrightarrow \forall i \in I : \lambda(q, i) = \lambda(q', i) \wedge \delta(q, i) \sim_{k-1} \delta(q', i)$$

- ▶  $\sim_k$  lässt sich also iterativ aus  $\sim_{k-1}$  berechnen.

# Minimierung von DMMs

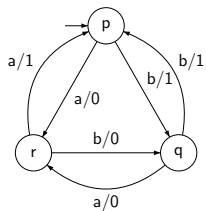
- Das Verfahren terminiert im Schritt  $k^* \geq 1$ , wenn kein neues Zustandspaar als inäquivalent markiert wurde, d.h. wenn  $\sim_{k^*} = \sim_{k^*-1}$  gilt, da:

$$\begin{array}{l} q \sim_{k^*+1} q' \\ \text{nach Def.} \\ \Leftrightarrow \forall i \in I : \lambda(q, i) = \lambda(q', i) \wedge \delta(q, i) \sim_{k^*} \delta(q', i) \\ \sim_{k^*} = \sim_{k^*-1} \\ \Leftrightarrow \forall i \in I : \lambda(q, i) = \lambda(q', i) \wedge \delta(q, i) \sim_{k^*-1} \delta(q', i) \\ \text{nach Def.} \\ \Leftrightarrow q \sim_{k^*} q', \end{array}$$

also gilt  $\sim_{k^*+1} = \sim_{k^*}$ .

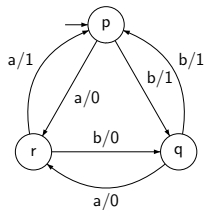
- Induktiv folgt also  $\sim_{k^*+i} = \sim_{k^*}$ , d.h. aus  $q \sim_{k^*} q'$  folgt bereits, dass  $q, q'$  für alle Eingabesequenzen die gleiche Ausgabe erzeugen.
- In jedem Schritt  $k < k^*$  wurde also mindestens ein Paar als inäquivalent erkannt, d.h.  $k^* \leq \binom{\#Q}{2}$ .

# Minimierung von DMMs - Beispiel



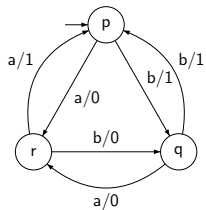
	p		-
$\sim_0$	q		
			r p

# Minimierung von DMMs - Beispiel



	p		*	-
	q		*	
$\sim_1$			r	p

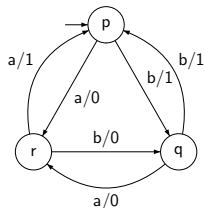
# Minimierung von DMMs - Beispiel



$\sim_2$

p	*	-
q	*	
r		p

## Minimierung von DMMs - Beispiel



Übungsaufgabe: Tabelle/Algorithmus so erweitern, dass kürzeste Trennsequenzen mitberechnet werden.

# Von Erweiterten DMMs zu minimalen DMMs

- ▶ Ziel: von einer EDMM zu einer *minimalen* DMM.
- ▶ Erste Möglichkeit:
  - ▶ EDMM in DMM entfalten und auf den einzelnen, konkreten Zuständen aus Kontrollzustand+Variablenbelegung obigen Algorithmus laufen lassen.
  - ▶ Probleme: Benötigter Speicher, Laufzeit quadratisch in Anzahl der DMM-Zustände.
- ▶ Zweite Möglichkeit:
  - ▶ Können Mengen von konkreten Zuständen (u.U.) effizient abspeichern (vgl. *BDDs* später in Vorlesung) und benötigte Operationen für Minimierungsalgorithmus auf Teilmengen ablaufen lassen, anstatt jeweils nur für einzelne Zustandspaar.
  - ▶ "Hoffnung": schneller und geringerer Speicherbedarf.

## Von Erweiterten DMMs zu minimalen DMMs - $k = 0$

- ▶ Wir wollen wieder die Äquivalenzrelation  $\sim$  berechnen mit

$$c \sim c' \Leftrightarrow \forall \sigma \in I^* : \hat{\lambda}(c, \sigma) = \hat{\lambda}(c', \sigma).$$

- ▶ Wir beginnen mit der Äquivalenzrelation  $\equiv^{(0)} := \sim_1$ , d.h.

$$c \equiv^{(0)} c' \Leftrightarrow \forall i \in I : \lambda(c, i) = \lambda(c', i).$$

*Hinweis:*  $\delta$  und  $\lambda$  bezeichnen hier die durch die EDMM implizit gegebenen Funktionen bezüglich einzelner Konfigurationen.

- ▶  $c \equiv^{(0)} c'$  stellt sicherlich eine notwendige Bedingung für  $c \sim c'$  dar, d.h.  $c \sim c' \Rightarrow c \equiv^{(0)} c'$ .

## Von Erweiterten DMMs zu minimalen DMMs - $k = 0$

- ▶ Die durch  $\equiv^{(0)}$  gegebenen Äquivalenzklassen  $A_1^{(0)}, \dots, A_n^{(0)}$  werden nun iterativ weiter unterteilt, indem notwendige Bedingungen propagiert werden.
- ▶ Nach  $u$  Unterteilungen (ausgehend von  $\equiv^{(0)} = \sim_1$ ) werden wir die Äquivalenzklassen  $A_1^{(u)}, \dots, A_{n+u}^{(u)}$  berechnet haben. Die hierdurch definierte Äquivalenzrelation wird mit  $\equiv^{(u)}$  bezeichnet werden.  
*Hinweis* : Die Äquivalenzrelation  $\equiv^{(u)}$  muss i.A. nicht mit einem  $\sim_{u'}$  übereinstimmen.
- ▶ Die Unterteilungen werden so gewählt werden, dass immer  $c \sim c' \Rightarrow c \equiv^{(u)} c'$  sichergestellt ist. D.h.  $\equiv^{(u)}$  kann höchstens zwei Konfiguration fälschlich als äquivalent annehmen.

## Von Erweiterten DMMs zu minimalen DMMs - $k > 0$

- ▶ Wir betrachten  $\equiv^{(u)}$  und versuchen herzuleiten, wie  $\equiv^{(u+1)}$  hierausgewonnen werden kann.
- ▶ Für ein Eingabezeichen  $i \in I$  sei  $\delta_i(c) := \delta(c, i)$  die Transitionsrelation bzgl. diesem festen  $i$ .
- ▶ Sei  $K$  eine beliebige Äquivalenzklasse aus  $\equiv^{(u)}$ .
- ▶ Wir betrachten die Menge  $\delta_i^{-1}(K)$  der Vorgänger von  $K$  bzgl.  $i$ , d.h. die Menge aller Konfigurationen  $c$  die bzgl. der Eingabe  $i$  in eine Konfiguration aus  $K$  übergehen.
- ▶ Sei  $A$  eine Äquivalenzklasse aus  $\equiv^{(u)}$ , mit  $A \cap \delta_i^{-1}(K) \neq \emptyset$ , d.h.  $A$  enthält Konfigurationen  $c$ , welche bei Eingabe von  $i$  in eine Konfiguration aus  $K$  übergehen.

## Von Erweiterten DMMs zu minimalen DMMs - $k > 0$

- ▶ Es gelte  $A \not\subseteq \delta_i^{-1}(K)$ . Dann setzen wir  $A' := A \cap \delta_i^{-1}(K)$ ,  
 $A'' := A \setminus A'$ .
- ▶ Alle Konfigurationen aus  $A'$  wechseln bei Eingabe von  $i$  nach  $K$ , während alle Konfigurationen aus  $A''$  unter  $i$  in Äquivalenzklassen  $\neq K$  übergehen.
- ▶ Für jedes Paar  $c' \in A', c'' \in A''$  von Konfigurationen gilt also  $\delta(c', i) \not\equiv^{(u)} \delta(c'', i)$ .  
Nach Induktion gilt also auch  $\delta(c', i) \not\sim \delta(c'', i)$ .
- ▶ Wir finden also - wegen  $c' \not\sim c''$  - eine Trennsequenz  $\sigma$  für  $\delta(c', i)$  und  $\delta(c'', i)$ .
- ▶ (Schritt von  $u$  nach  $u + 1$ ) Dann ist  $i\sigma$  sicherlich eine Trennsequenz für  $c'$  und  $c''$ .
- ▶ D.h. alle Konfigurationen aus  $A'$  sind  $\sim$ -inäquivalent zu Konfigurationen aus  $A''$ . ( $c, c'$  waren beliebig gewählt.)

## Von Erweiterten DMMs zu minimalen DMMs - $k > 0$

- ▶ Definieren wir also  $\equiv^{(u+1)}$ , indem wir die Äquivalenzklasse  $A$  durch  $A'$  und  $A''$  ersetzen, so gilt sicherlich noch  $c \sim c' \Rightarrow c \equiv^{(u+1)} c'$ .
- ▶ Der Fall  $A \subseteq \delta_i^{-1}(K)$  ist hingegen uninteressant, da bzgl.  $i$  und  $\equiv^{(u)}$  dann keine Trennsequenz gefunden werden kann.

## Von Erweiterten DMMs zu minimalen DMMs - Terminierung

- ▶ Der Algorithmus terminiert, wenn keine Äquivalenzklasse mehr unterteilt werden muss, d.h. wenn bzgl.  $\equiv^{(u)}$  schließlich gilt:

$$\forall K, A \in \equiv^{(u)} \forall i \in I : A \cap \delta_i^{-1}(K) \in \{\emptyset, A\}$$

- ▶ Da in jedem Schritt vor der Terminierung genau eine Äquivalenzklasse in genau zwei nicht-leere Mengen aufgespalten wurde, kann es maximal so viele Schritte geben, wie die EDMM Konfigurationen repräsentiert.

## Von Erweiterten DMMs zu minimalen DMMs - Korrektheit

- ▶ Sei  $\equiv^*$  die nach Terminierung berechnete Äquivalenzrelation. Wir behaupten  $\equiv^* = \sim$ . Nach Konstruktion gilt  $c \sim c' \Rightarrow c \equiv^* c'$ . Bleibt also nur  $c \sim c' \Leftarrow c \equiv^* c'$ . Oder äquivalent:  $c \not\sim c' \Rightarrow c \not\equiv^* c'$ :
- ▶ Wegen  $c \not\sim c'$  finden wir eine kürzeste Trennsequenz  $\sigma = i_1 \dots i_n$  zu  $c$  und  $c'$ . Es sei  $c_r := \hat{\delta}(c, i_1 \dots i_r)$  und entsprechend  $c'_r$  mit  $c = c_0$  und  $c' = c'_0$ .
- ▶ Insbesondere werden also  $c_{n-1}$  und  $c'_{n-1}$  von  $i_n$  getrennt. (Ansonsten könnten wir  $\sigma$  verkürzen)

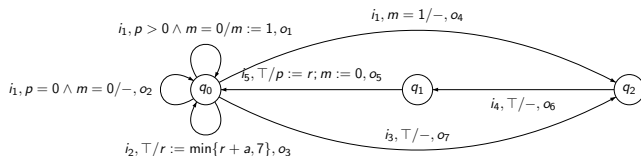
## Von Erweiterten DMMs zu minimalen DMMs - Korrektheit

- ▶ Es gilt also  $c_{n-1} \not\sim_1 c'_{n-1}$ . Wegen  $\equiv^{(0)} := \sim_1$  folgt damit auch  $c_{n-1} \not\equiv^* c'_{n-1}$ , da Äquivalenzklassen nur weiter unterteilt werden.
- ▶ Seien  $c_{n-2}$  und  $c'_{n-2}$  in einer gemeinsamen Äquivalenzklasse  $A$  bzgl.  $\equiv^*$  enthalten, d.h.  $c_{n-2} \equiv^* c'_{n-2}$ . Ist  $K$  dann die Äquivalenzklasse von  $c_{n-1}$  bzgl.  $\equiv^*$ , so könnte  $A$  jedoch durch  $\delta_{i_{n-1}}^{-1}(K)$  verfeinert werden, da  $\delta(c'_{n-2}, i_{n-1}) = c'_{n-1} \notin K$ . Der Algorithmus könnte somit nicht terminiert haben - entgegen der Annahme.
- ▶ Es muss also doch  $c_{n-2} \not\equiv^* c'_{n-2}$  gelten.
- ▶ Entsprechend folgt, dass  $c_{n-3} \not\equiv^* c'_{n-3}$  gelten muss, ... und schließlich  $c_0 \not\equiv^* c'_0$ .

## Von Erweiterten DMMs zu minimalen DMMs - Anmerkungen

- ▶ Der Algorithmus benötigt als Operationen die Berechnung von  $\delta_i^{-1}$  und die Schnitt- und Komplementbildung für Mengen. Wie später in der Vorlesung gezeigt werden wird, werden all diese Operationen von BDDs unterstützt.
- ▶ Bei geeigneter Implementierung (bereits ohne BDDs) führt dieser Ansatz zu einem Algorithmus mit Laufzeit  $O(\#Q \cdot \log \#Q)$ , wobei  $Q$  die Menge der *Konfigurationen* sei, vgl. *Aho, Hopcroft, Ullmann, The Design and Analysis of Computer Algorithms, Algorithm 4.5, p.158+ .*

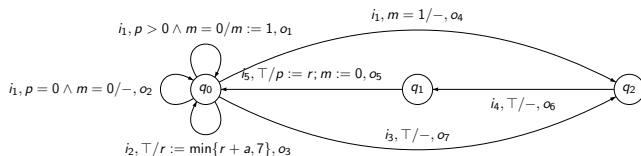
## Von Erweiterten DMMs zu minimalen DMMs - Bsp.



Ziel: Minimierung der EDMM aus der Token-Ring-Spezifikation.

*Hinweis* : Zur Vereinfachungen, werden die Transitionen zu  $i_2$  und  $i_3$  (zunächst) nicht betrachtet.

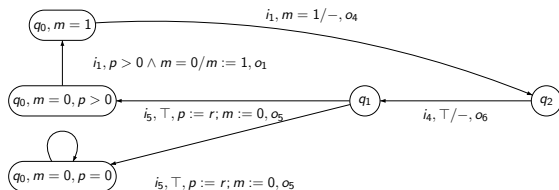
## Von Erweiterten DMMs zu minimalen DMMs - Bsp.



Einteilen der Konfigurationen mit gleichem  
Eingabe/Ausgabe-Verhalten. Wegen  $i_1$  erhalten wir die Einteilung:

$$\begin{aligned}[q_1] &:= \{(q_1, m, p, r)\} \\ [q_2] &:= \{(q_2, m, p, r)\} \\ [q_0, m = 1] &:= \{(q_0, 1, p, r)\} \text{ (wegen } \sigma_4) \\ [q_0, m = 0, p = 0] &:= \{(q_0, 0, 0, r)\} \text{ (wegen } \sigma_2) \\ [q_0, m = 0, p > 0] &:= \{(q_0, 0, p, r) \mid p > 0\} \text{ (wegen } \sigma_1)\end{aligned}$$

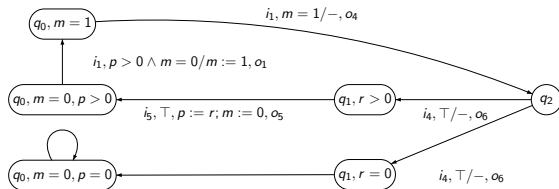
## Von Erweiterten DMMs zu minimalen DMMs - Bsp.



$[q_0, m = 0, p > 0]$  (bzw.  $[q_0, m = 0, p = 0]$ ) führt zur Spaltung von  $[q_1]$ .

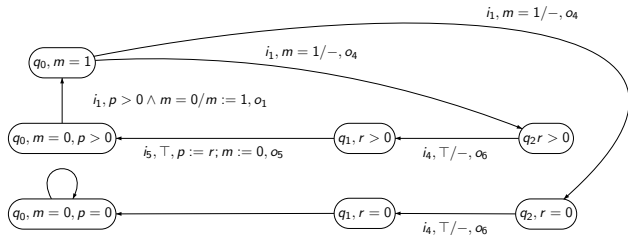
Wegen der Aktion  $p := r; m := 0$  zu  $i_5$  ist  $[q_0, m = 1]$  unerreichbar und  $[q_1]$  wird unterteilt in  $[q_1, r > 0]$  und  $[q_1, r = 0]$ .

## Von Erweiterten DMMs zu minimalen DMMs - Bsp.



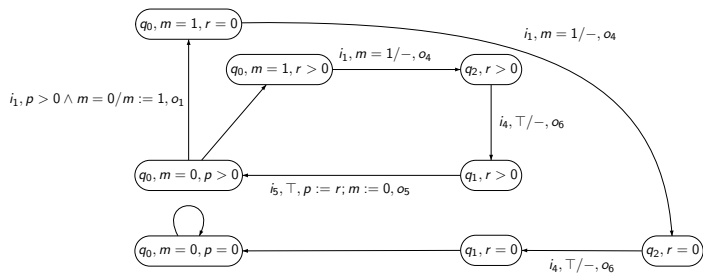
Wegen  $[q_1, r > 0]$  (bzw.  $[q_1, r = 0]$ ) muss  $[q_2]$  in  $[q_2, r > 0]$  und  $[q_2, r = 0]$  aufgeteilt werden.

## Von Erweiterten DMMs zu minimalen DMMs - Bsp.



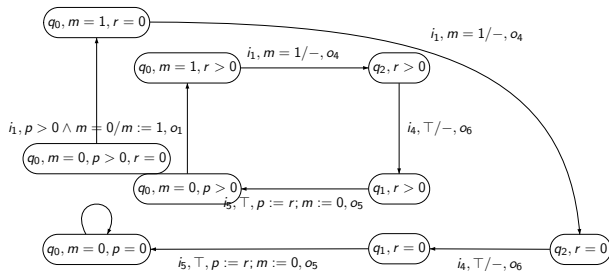
Mittels  $[q_2, r > 0]$  wird  $[q_0, m = 1]$  aufgespalten in  $[q_0, m = 1, r > 0]$  und  $[q_0, m = 1, r = 0]$ .

## Von Erweiterten DMMs zu minimalen DMMs - Bsp.



Schließlich Split von  $[q_0, m = 0, p > 0]$  in  $[q_0, m = 0, p > 0, r > 0]$  und  $[q_0, m = 0, p > 0, r = 0]$ .

## Von Erweiterten DMMs zu minimalen DMMs - Bsp.



Entsprechend müssten nun noch die bis jetzt vernachlässigten Transitionen zu  $i_2$  und  $i_3$  betrachtet werden, wofür die Fälle  $a = 0$  und  $a > 0$  betrachtet werden müssten.