2.2 Bisimilarität bei reaktiven Systemen

Bei Transitionssystemen unterscheiden wir zwei Formen der Äquivalenz: Folgenäquivalenz und Akzeptanzäquivalenz. Akzeptanzäquivalenz entspricht der bei NFA gebräuchlichen Äquivalenz.

Definition 2.3 *Zwei Transitionssysteme TS_1 und TS_2 heißen* folgenäquivalent*, falls sie die gleiche Menge von Aktionsfolgen haben:*

$$TS_1 \sim_{AS} TS_2 :\Leftrightarrow AS(TS_1) = AS(TS_2)$$

Sie heißen terminal folgenäquivalent *oder* akzeptanzäquivalent*, falls gilt:*

$$TS_1 \sim_{L} TS_2 :\Leftrightarrow L(TS_1) = L(TS_2)$$

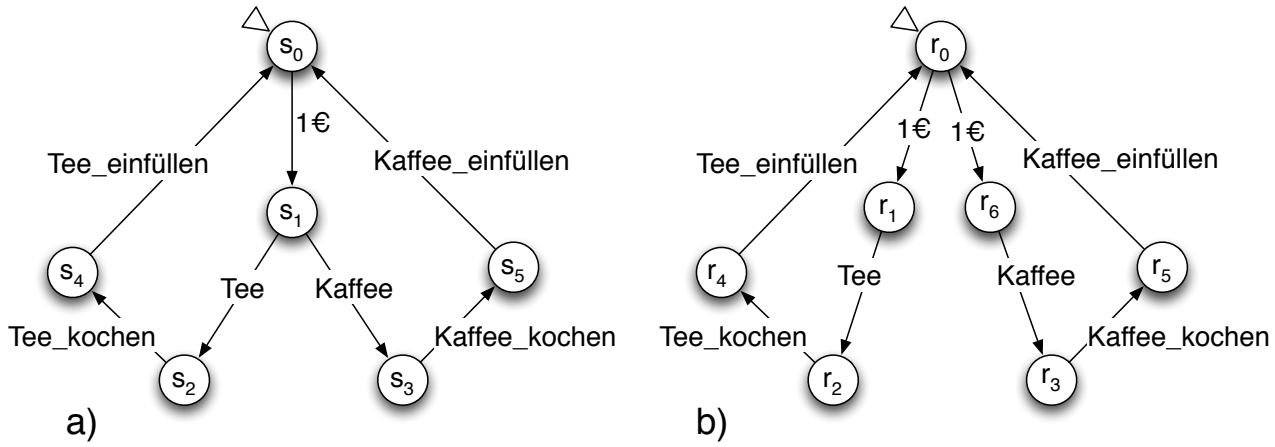


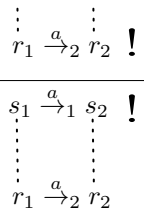
Abbildung 2.1: Zwei folgenäquivalente Transitionssysteme

Abbildung 2.1 zeigt zwei folgenäquivalente Transitionssysteme. Das Beispiel zeigt auch, dass in manchen Fällen die Folgenäquivalenz kein adäquates Mittel ist, um gleiches Systemverhalten zu definieren: Im linken System sind nach dem Münzeinwurf beide Wahlmöglichkeiten geben, was in dem rechten System nicht der Fall ist.
Eine bessere Formalisierung für gleiches Systemverhalten ist durch den Begriff der *Bisimulation* möglich.
Wir betrachten für die folgende Definition zwei Transitionssysteme mit der gleichen Aktionsmenge A. Dies ist keine echte Einschränkung, da dies durch Obermengenbildung immer zu erreichen ist.

Definition 2.4 *Gegeben seien zwei Transitionssysteme $TS_i = (S_i, A, tr_i, S_i^0, S_i^F)$ für $i \in \{1, 2\}$. Eine (aktionsbasierte)* Bisimulation *für (TS_1, TS_2) ist eine binäre Relation $\mathcal{B} \subseteq S_1 \times S_2$, für die folgendes gilt:*

a) $\forall s_0 \in S_1^0 : \exists r_0 \in S_2^0 : (s_0, r_0) \in \mathcal{B}$
$\forall r_0 \in S_2^0 : \exists s_0 \in S_1^0 : (s_0, r_0) \in \mathcal{B}$

b) Für alle $(s_1, r_1) \in \mathcal{B}$ gilt:
$s_1 \xrightarrow[1]{a} s_2 \Rightarrow \exists r_2 \in S_2 : r_1 \xrightarrow[2]{a} r_2 \wedge (s_2, r_2) \in \mathcal{B}$
$r_1 \xrightarrow[2]{a} r_2 \Rightarrow \exists s_2 \in S_1 : s_1 \xrightarrow[1]{a} s_2 \wedge (s_2, r_2) \in \mathcal{B}$



c) $\forall (s,r) \in \mathcal{B} : s \in S_1^F \Leftrightarrow r \in S_2^F$

Zustände mit $(s,r) \in \mathcal{B}$ *heißen* bisimilar *(in Zeichen* $s \underline{\leftrightarrow} r$*).*

TS_1 *und* TS_2 *heißen* bisimilar *(in Zeichen* $TS_1 \underline{\leftrightarrow} TS_2$*), falls eine solche Bisimulations-Relation* $\mathcal{B}$ *existiert.*

Die Intention der Bedingungen a) und c) in Definition 2.4 sind offensichtlich: durch sie gibt es zu jedem Anfangszustand einen bisimularen im anderen Transitionssystem. Ein Endzustand hat nur ebensolche bisimulare Partner. Die Bedingung b) wird durch die daneben stehende Graphik illustriert. Die obere dieser Graphiken entspricht dem ersten Teil der Bedingung b). Das Ausrufezeichen markiert das postulierte Element r_2, während die anderen Elemente s_1, s_2 und r_1 zur Voraussetzung der Bedingung gehören. Die durch die Relation $\mathcal{B}$ verbundenen Paare $s_1 \underline{\leftrightarrow} r_1$ und $s_2 \underline{\leftrightarrow} r_2$ sind durch die unterbrochenen Linien gekennzeichnet. Entsprechendes gilt für den zweiten Teil der Bedingung b) und die untere Graphik.

Beispiel 2.5 Wir zeigen nun, dass die (intuitiv) nicht verhaltensäquivalenten Transitionssysteme aus Abbildung 2.1 tatsächlich nicht (formal) bisimilar sind. Wären sie bisimilar, dann müssten wegen der Bedingung a) die Anfangszustände in der Relation stehen: $s_0 \underline{\leftrightarrow} r_0$. Aus der ersten Bedingung b) folgt dann, dass mit $a = 1€$ auch $s_1 \underline{\leftrightarrow} r_1$ oder $s_1 \underline{\leftrightarrow} r_6$ gelten muss. Im ersten Fall gibt es dann zwar (wieder nach Bedingung b) zu $s_1 \xrightarrow{\text{Tee}} s_2$ die Transition $r_1 \xrightarrow{\text{Tee}} r_2$, nicht aber zu $s_1 \xrightarrow{\text{Kaffee}} s_3$ eine Transition $r_1 \xrightarrow{\text{Kaffee}} r$ für irgend ein r. Da der zweite Fall in analoger Weise zum Widerspruch führt, können die Transitionssysteme nicht bisimilar sein.
Im Gegensatz dazu sind die Transitionssysteme aus Abbildung 2.2 bisimilar. Die Bisimulations-Relation ist $\mathcal{B} = \underline{\leftrightarrow} = \{(s_i, r_i) \mid i = 0, \cdots, 5\} \cup \{(s_1, r_6)\}$.

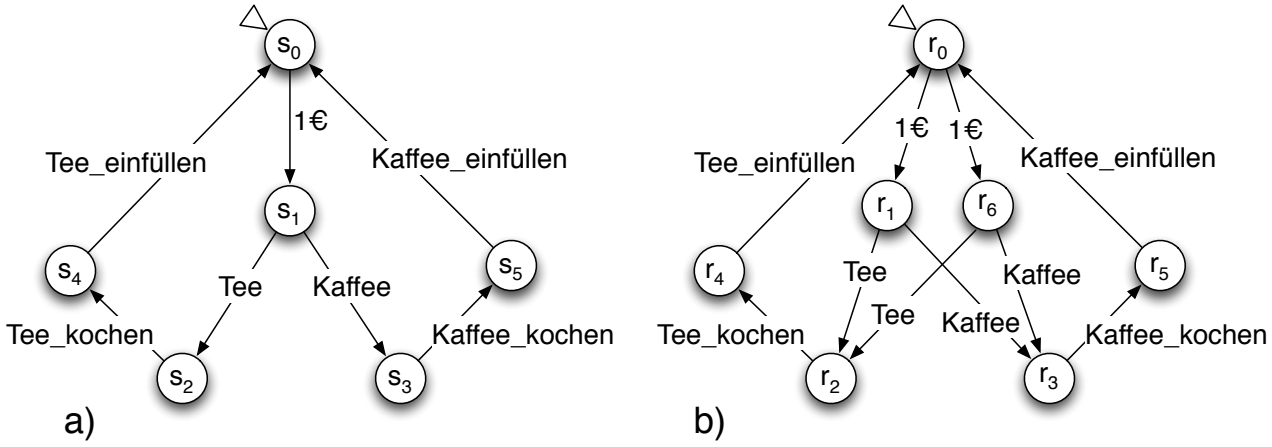


Abbildung 2.2: Zwei bisimilare Transitionssysteme

Lemma 2.6 *Seien* TS_1 *und* TS_2 *zwei beliebige Transitionssysteme mit* $TS_1 \underline{\leftrightarrow} TS_2$*, dann gilt für alle* $w \in A^*$*:*

$$(s_0 \xrightarrow[1]{w} s_1 \ \wedge \ s_0 \underline{\leftrightarrow} r_0) \ \Rightarrow \ (\exists r_1 : r_0 \xrightarrow[2]{w} r_1 \ \wedge \ s_1 \underline{\leftrightarrow} r_1)$$

Beweis: Wir beweisen dies durch Induktion über die Wortlänge $|w|$.

- Induktionsanfang $w = \epsilon$: In diesem Fall gilt $s_1 = s_0$ und r_1 kann als r_0 gewählt werden.
- Induktionsschritt $w = va$ mit $v \in A^*, a \in A$:
  Sei also $s_0 \xrightarrow[1]{va} s_2$ und $s_0 \underline{\leftrightarrow} r_0$. Dann gibt es ein $s_1 \in S_1$ mit $s_0 \xrightarrow[1]{v} s_1 \xrightarrow[1]{a} s_2$.
  Nach Induktionsvoraussetzung gibt es einen Zustand r_1 mit $r_0 \xrightarrow[2]{v} r_1$ und $s_1 \underline{\leftrightarrow} r_1$.
  Aufgrund der Bisimilarität gibt es r_2 mit $r_1 \xrightarrow[2]{a} r_2$ und $s_2 \underline{\leftrightarrow} r_2$. Also gilt $r_0 \xrightarrow[2]{va} r_2$ und $s_2 \underline{\leftrightarrow} r_2$.

$$\begin{array}{ccccc} s_0 & \xrightarrow{v}_1 & s_1 & \xrightarrow{a}_1 & s_2 \\ \vdots & & \vdots & & \vdots \\ r_0 & \xrightarrow{v}_2 & r_1 & \xrightarrow{a}_2 & r_2 \; ! \end{array}$$

Damit ist die Aussage bewiesen. □

Satz 2.7 *Seien TS_1 und TS_2 zwei beliebige bisimilare Transitionssysteme, dann hat jeder erreichbare Zustand des einen Systems einen bisimilaren Partner im anderen:*

$$\forall s \in R(TS_1) : \exists r \in R(TS_2) : s \underline{\leftrightarrow} r$$
$$\forall r \in R(TS_2) : \exists s \in R(TS_1) : s \underline{\leftrightarrow} r$$

Beweis: Wegen der Symmetrie der Aussage in TS_1 und TS_2 reicht es aus, nur eine Hälfte zu zeigen:
Wir zeigen $\forall s \in R(TS_1) : \exists r \in R(TS_2) : s \underline{\leftrightarrow} r$.
Wenn s ein erreichbarer Zustand ist, dann existiert ein Wort w, mit dem der Zustand s von einem Startzustand $s_0 \in S_1^0$ erreicht werden kann: $s_0 \xrightarrow{w} s$.
Nach Def. 2.4 a) existiert zu s_0 ein bisimilarer Partner $r_0 \in S_2^0$ mit $(s_0, r_0) \in \mathcal{B}$.
Nach Lemma 2.6 gibt es $r \in S_2$ mit $r_0 \xrightarrow[2]{w} r$ und $s \underline{\leftrightarrow} r$.
Offensichtlich ist r auch erreichbar. □

Der folgende Satz zeigt, dass Bisimilarität feiner als Folgenäquivalenz ist, denn Bisimilarität trennt auch folgenäquivalente Systeme (vgl. dazu das Beispiel in Abbildung 2.1).

Satz 2.8 *Wenn zwei Transitionssysteme TS_1 und TS_2 bisimilar sind, dann sind sie auch folgen- und akzeptanzäquivalent, (aber nicht umgekehrt), d.h.:*

$$TS_1 \underline{\leftrightarrow} TS_2 \quad \Rightarrow \quad (TS_1 \sim_L TS_2) \wedge (TS_1 \sim_{AS} TS_2)$$

Beweis: Sei $TS_1 \underline{\leftrightarrow} TS_2$. Wir beweisen nun $TS_1 \sim_L TS_2$, also $L(TS_1) = L(TS_2)$:
Zu $w \in L(TS_1)$ gibt es $s_0 \in S_1^0$ und $s_1 \in S_1^F$ mit $s_0 \xrightarrow[1]{w} s_1$.

Nach Definition 2.4 a) existiert ein $r_0 \in S_2^0$ mit $s_0 \underline{\leftrightarrow} r_0$.
Nach Lemma 2.6 gibt es $r_1 \in S_2$ mit $r_0 \xrightarrow[2]{w} r_1$ und $s_1 \underline{\leftrightarrow} r_1$.

Nach Definition 2.4 c) muss auch $r_1 \in S_2^F$ gelten. Also erhalten wir $w \in L(TS_2)$.
Da $S_1^F = S_1$ und $S_2^F = S_2$ als Spezialfall enthalten ist, gilt auch $TS_1 \underline{\leftrightarrow} TS_2 \Rightarrow AS(TS_1) = AS(TS_2)$. □

Damit folgt, dass die beiden bisimilaren Transitionssysteme von Abbildung 2.2 auch akzeptanzäquivalent sind (z.B. mit $S_1^F = \{s_0\}$ und $S_2^F = \{r_0\}$).

Satz 2.9 *Wenn $\mathcal{B}$ und $\mathcal{B}'$ zwei Bisimulationen zwischen TS_1 und TS_2 sind, dann ist auch die Vereinigung $(\mathcal{B} \cup \mathcal{B}')$ eine.*

Beweis: Als Übung. □

Aus dem Satz folgt, dass die *größte* Bisimulation $\approx$ zwischen TS_1 und TS_2 existiert:

$$\approx \;:= \bigcup_{\mathcal{B} \text{ ist Bisimulation}} \mathcal{B}$$

2.3 Synchronisation von Transitionssystemen

Transitionssysteme (und damit auch Automaten) können synchronisiert werden. Dadurch erhält man Modelle für nebenläufige Systeme. Prinzipiell werden folgende Arten der Synchronisation unterschieden:

a) *Rendezvous-Synchronisation*
b) *Speicher-Synchronisation*
c) *Nachrichten-Synchronisation*

Bei der Rendezvous-Synchronisation (engl. auch „handshake synchronization“) werden zu synchronisierende Aktionen ungeteilt zusammen ausgeführt. Diese Form wird meist bei Transitionssystemen angewandt. Speicher-Synchronisation erfolgt durch Schreiben und Lesen auf gemeinsam zugreifbaren Speicherbereichen, z.B. auf gemeinsamen Variablen. Nachrichten-Synchronisation wird durch das Versenden und Empfangen von Nachrichten realisiert, wobei es oft keine oberen Schranken für die Nachrichtenlaufzeit gibt. Speicher- und Nachrichten-Synchronisation können durch Rendezvous-Synchronisation dargestellt werden, indem man eine Funktionseinheit (einen „Prozess“) zwischenschaltet, der die entsprechende Verwaltung des Speicherbereiches oder des Nachrichtenkanals modelliert.

Zwei synchronisierte Transitionssysteme ergeben wieder ein Transitionssystem, das als Zustandsmenge das Mengenprodukt $S_1 \times S_2$ („kartesisches Produkt“) der ursprünglichen Zustandsmengen hat und daher *Produkt-Transitionssystem* genannt wird. Die zu synchronisierenden Transitionspaare werden durch eine Relation $Sync \subseteq A_1 \times A_2$ bestimmt. Je nach dem Umfang dieser Relation ist das resultierende Produkt-Transitionssystem mehr oder weniger wieder sequentiell, d.h. erlaubt mehr oder weniger nebenläufige Aktionen. Vollständig sequentiell ist es zum Beispiel das Produkt von endlichen Automaten („Produktautomat“), das zur Konstruktion eines Automaten eingesetzt wird, welcher den Durchschnitt bzw. die Vereinigung regulärer Mengen akzeptiert.

Definition 2.10 *Gegeben seien zwei Transitionssysteme* $TS_i = (S_i, A_i, tr_i, S_i^0, S_i^F), i = 1, 2$ *(mit nicht notwendig gleichen Aktionsmengen* A_i*) und eine* Synchronisations-Relation $Sync \subseteq A_1 \times A_2$. *Das* Produkt-Transitionssystem *von* TS_1 *und* TS_2 *ist das Transitionssystem*

$$TS_1 \otimes_{Sync} TS_2 = (S_1 \times S_2, A_1 \cup A_2 \cup Sync, tr_3, S_1^0 \times S_2^0, S_1^F \times S_2^F),$$

wobei die Transitionsrelation tr_3 *durch die folgenden Regeln* $Sy1$, $Sy2$ *und* $Sy3$ *definiert wird.*

1. *Regel* $Sy1$*: Erste Komponente*

$$\frac{s_1 \xrightarrow{a_1}_1 s_2, \qquad a_1 \notin pr_1(Sync), \qquad r \in S_2}{(s_1, r) \xrightarrow{a_1}_3 (s_2, r)}$$

   Dabei ist $pr_1(Sync) := \{s | \exists r \in A_2 : (s, r) \in Sync\}$ *die Menge der ersten Komponenten (1. Projektion) und* $pr_2(Sync)$ *entsprechend definiert.*

2. *Regel* $Sy2$*: Zweite Komponente*

$$\frac{r_1 \xrightarrow{a_2}_2 r_2, \qquad a_2 \notin pr_2(Sync), \qquad s \in S_1}{(s, r_1) \xrightarrow{a_2}_3 (s, r_2)}$$

3. Regel Sy3: Kommunikation

$$\frac{s_1 \xrightarrow{a_1}_1 s_2, \qquad r_1 \xrightarrow{a_2}_2 r_2, \qquad (a_1, a_2) \in Sync}{(s_1, r_1) \xrightarrow{(a_1,a_2)}_3 (s_2, r_2)}$$

Statt $TS_1 \otimes_{Sync} TS_2$ schreiben wir auch kürzer $TS_1 \otimes TS_2$, wenn Sync aus dem Kontext ersichtlich ist.

Definieren wir eine eine Abbildung $\gamma : Sync \to A_3$ von $Sync$ in ein *Kommunikationsalphabet* A_3, dann kann statt (a_1, a_2) in Regel $Sy3$ auch $\gamma(a_1, a_2)$ eingesetzt werden. Die Aktionsmenge ist dann entsprechend nicht $A_1 \cup A_2 \cup Sync$, sondern $A_1 \cup A_2 \cup A_3$.

Aufgabe 2.11 Seien TS_1 und TS_2 zwei Transitionssysteme.

1. Welches Systemverhalten beschreibt $TS_1 \otimes_{Sync} TS_2$ für $Sync = \emptyset$?
2. Welches Systemverhalten beschreibt $TS_1 \otimes_{Sync} TS_2$ für $Sync = \{(a, a) \mid a \in A\}$?

Beispiel 2.12 (einfaches Sender-Empfänger-System)
Das Beispiel von Abbildung 2.3 zeigt zwei Transitionssysteme, die als Sender S und Empfänger R über einen Kanal B kommunizieren. Der Sender nimmt Daten d mit der Aktion $r_A(d)$ („receive“) von einem Benutzer über einen Kanal A auf und schreibt dann das Datum d in den Kanal B mit der Aktion $s_B(d)$ („send“). Diese Aktion kommuniziert mit der entsprechenden Aktion $r_B(d)$ des Empfängers, der das Datum über den Kanal C an den empfangenden Benutzer abgibt. Wir setzen also $Sync = \{(s_B(d), r_B(d))\}$ und $\gamma(s_B(d), r_B(d)) = c_B(d)$. Rechts oben ist das resultierende Produkt-Transitionssystem $S \otimes R$ dargestellt. Die gemäß Regel $Sy3$ konstruierte Transition ist als punktierte Linie gezeichnet.

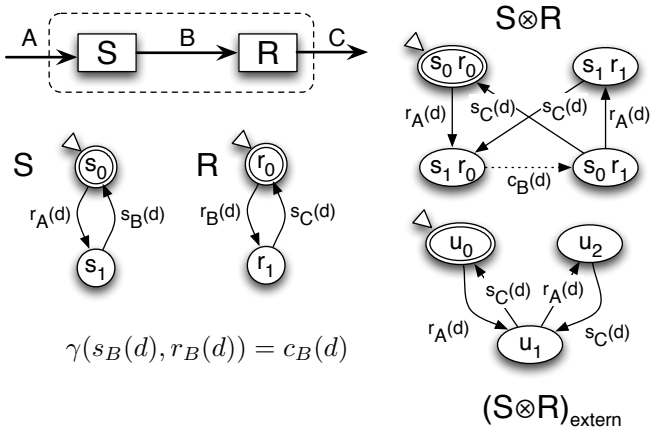


Abbildung 2.3: Ein einfaches Sender-Empfänger-System

Als „externes Verhalten“ bezeichnet man die Abläufe der extern sichtbaren Aktionen. Im vorliegenden Fall sind das die Aktionen zu den Kanälen A und C. Bezogen auf $S^F = \{(s_0, r_0)\}$ als Endzustand sollten dies alle Folgen wie $r_A(d)s_C(d)r_A(d)s_C(d) \cdots r_A(d)s_C(d)$ sein. Da $r_A(d)$ und $s_C(d)$ unabhängig voneinander sind, können aber auch Vertauschungen von diesen Aktionen in der Aktionsfolge auftreten.

Dieses Verhalten erhält man dadurch, dass die interne Transition $(s_1, r_0) \xrightarrow{c_B(d)}_3 (s_0, r_1)$ das Etikett ϵ erhält und die Bezeichner der anderen Transitionen als Etiketten übernimmt. Dieses Verhalten ist auch die Menge der akzeptablen Folgen des Transitionssystems $(S \otimes R)_{extern}$ (Abbildung 2.3 rechts unten). Diese Konstruktion entspricht der Beseitigung von ϵ-Übergängen, wie sie für Automaten in FGI-1 behandelt wurde.

Beispiel 2.13 (gestörtes Sender-Empfänger-System)
Um ein nicht korrektes Verhalten im vorstehenden Beispiel zu zeigen, soll nun der interne Kanal B gestört sein (Abb. 2.4). Die Störung wird dadurch dargestellt, dass der Sender das Fehlersignal $\perp$ in den Kanal schreibt. Hier ist $Sync = \{(s_B(d), r_B(d)), (s_B(\perp), r_B(\perp))\}$ und $\gamma(s_B(d), r_B(d)) = c_B(d)$ und $\gamma(s_B(\perp), r_B(\perp)) = c_B(\perp)$. Das externe Verhalten enthält zum Beispiel die fehlerhafte Folge $r_A(d)s_C(d)r_A(d)s_C(\perp)r_A(d)s_C(d)$.

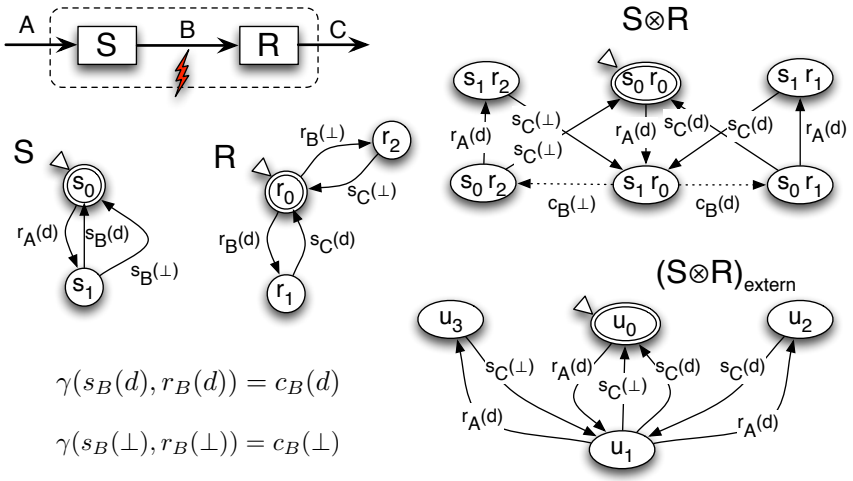


Abbildung 2.4: Ein gestörtes Sender-Empfänger-System

Beispiel 2.14 (entstörtes Sender-Empfänger-System)
Wir entwickeln im gleichen Formalismus das „Alternierbitprotokoll“. Im ersten Schritt wird eine Bestätigung über einen ungestörten Kanal D zurückgesandt (Abbildung 2.5). Wenn der Sender im Zustand s_2 ein „ok“ erhält, geht er in den Zustand s_0 und liest die nächste Eingabe. Im anderen Fall bei „*not ok*“ geht er nach s_1 zurück und sendet das Datum d erneut. Um das gewünschte externe Verhalten, wie durch $(S \otimes R)_{extern}$ dargestellt, zu erhalten, darf das System nicht immer in dieser Schleife bleiben. Man muss also (z.B. durch technische Maßnahmen) dafür sorgen, dass der Kanal B nicht permanent gestört bleibt. Dies geht in die Spezifikation des Systems als „Fairness-Bedingung“ ein. Formal wird dies dadurch ausgedrückt, dass immer der Endzustand erreicht werden muss oder (bei unendlichen Prozessen) immer wieder durchlaufen werden muss.

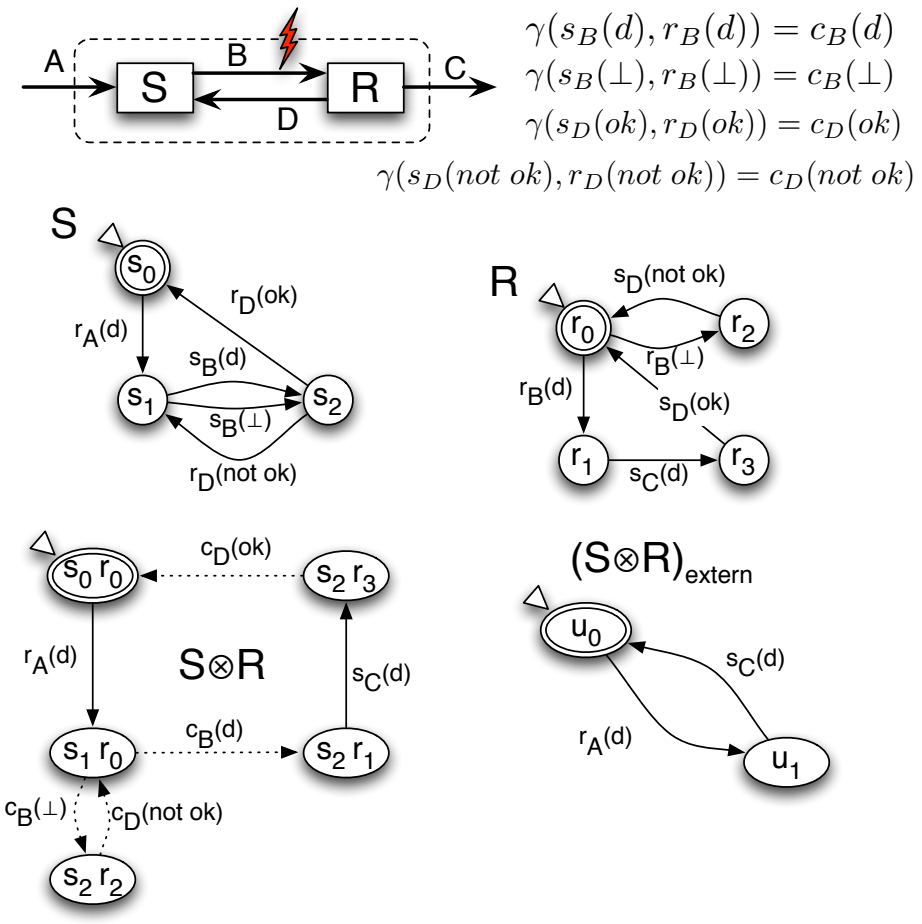


Abbildung 2.5: Entstörtes Sender-Empfänger-System

Beispiel 2.15 (Das Alternierbitprotokoll)
Die Spezifikation des Alternierbitprotokolls geht davon aus, dass auch der Bestätigungskanal D gestört ist, dies aber bei beiden Kanälen B und D nie permanent der Fall ist. Dazu erhält auch der Kanal D eine Bestätigungs-Prozedur. Diese kann aber mit derjenigen von B zusammengelegt werden, indem dem Datum d ein Bit mit den Werten 0 oder 1 beigefügt wird, wie z.B. in der Transition $s_1 \xrightarrow{s_B(d,0)} s_2$ in Abbildung 2.6. Erhält der Sender das Bit mit dem selben Wert (in den Zuständen s_3 oder s_0) zurück, dann kann er das nächste Datum mit alterniertem Bit versenden. Im anderen Fall sendet er das vorherige erneut (in den Zuständen s_1 oder s_4). Das externe Verhalten erhält man wieder, indem in dem Produkt-Transitionssystem $S \otimes R$ die (unterbrochen dargestellten) internen Transitionen der Kanäle B und D mit dem Etikett ϵ versieht. Das so betrachtete Produkt-Transitionssystem ist dann akzeptanzäquivalent zu $(S \otimes R)_{extern}$, das das gewünschte externe Verhalten hat.

Anmerkung: Die hier behandelten Beispiele haben den Spezialfall behandelt, dass die zu übermittelnden Daten d aus einer Datenmenge Δ stammen, die einelementig ist: $\Delta = \{d\}$. Der allgemeinere Fall einer endlichen Datenmenge $\Delta = \{d_1, \cdots, d_k\}$ ist im Prinzip genauso zu behandeln. In den Zuständen des Transitionssystems muss dann das jeweilige d durch die Zustände gespeichert werden. Im Beispiel des Alternierbitprotokolls (Beispiel 2.15, Abbildung 2.6) wird der Wechsel von einem Datum zum nächsten durch den Wechsel von d zu d' dargestellt. Die Transitionssysteme werden für größere Mengen Δ unpraktikabel groß. Abhilfe schaffen hier die in den folgenden Kapitel behandelten Darstellungen durch Petrinetze und Prozessalgebra.

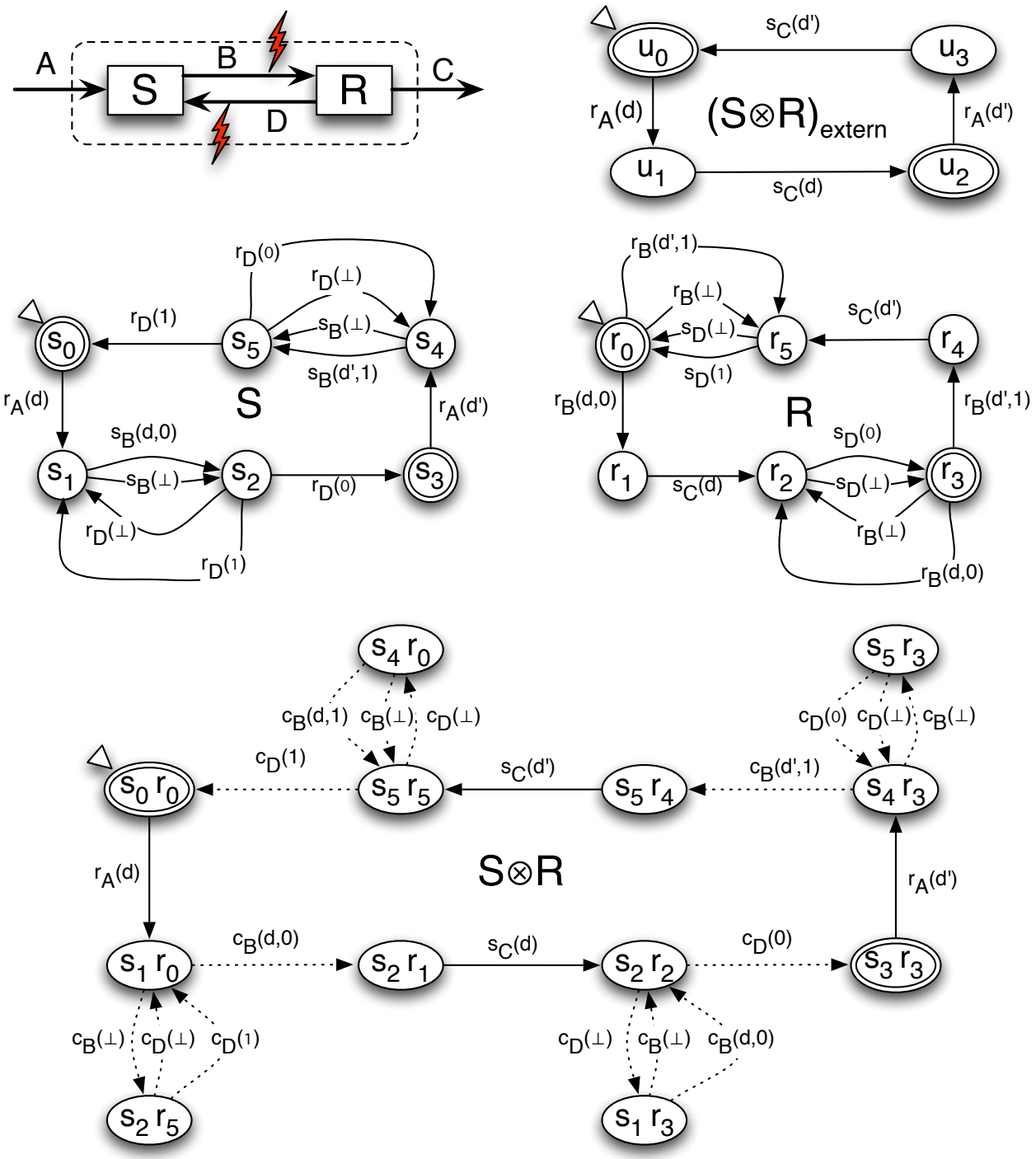


Abbildung 2.6: Das Alternierbitprotokoll

2.4 Etikettierte Transitionssysteme

Sei E eine beliebige Menge, genannt Etikettenmenge (engl. label set).
Eine *Transitionsetikettenfunktion* eines Transitionssystems TS ist eine Abbildung $E_A : A \to E$, die Aktionen auf Etikettenmenge abbildet. Wir erweitern $E_A : A \to E$ kanonisch auf Wörter $w \in A^*$:

$$E_A^*(a_1 \cdots a_n) = E_A(a_1) \cdots E_A(a_n)$$

Analog erhalten wir $E_A^\omega : A^\omega \to E^\omega$ für ω-Wörter.
Wir betrachten nun die Bilder der TS-Sprachen bzgl. der Etikettenabbildung: $E_A(TS) := E_A^*(FS(TS)) := \{E_A^*(w) \in \Sigma^* \mid w \in FS(TS)\}$ ist die *terminale Etikettensprache*.
Im Folgenden wird meist eine Transitionsetikettenfunktion $E_A : A \to E$ mit $E = \Sigma \cup \{\epsilon\}$ verwendet. Dabei ist Σ ein Etikettenalphabet und ϵ das leere Wort. E_A kann sinnvoll auch als Homomorphismus $E_A^* : A^* \to \Sigma^*$ aufgefasst werden.

Beispiel 2.16 Transitionsetiketten dienen oft dazu, interne von extern sichtbaren Aktionen zu unterscheiden. So könnte man hier die Aktionen aus {Tee_kochen,Kaffee_kochen} als interne Aktionen deklarieren, indem man definiert:

$$E_A(a) := \begin{cases} \epsilon, & \text{falls } a \in \{\text{Tee\_kochen,Kaffee\_kochen}\} \\ a, & \text{sonst} \end{cases}$$

Wählt man statt der Aktionsfolgen die Bilder unter E_A, dann sind die internen Aktionen {Tee_kochen,Kaffee_kochen} unsichtbar. Abbildung 2.7 b) zeigt eine entsprechende Darstellung.

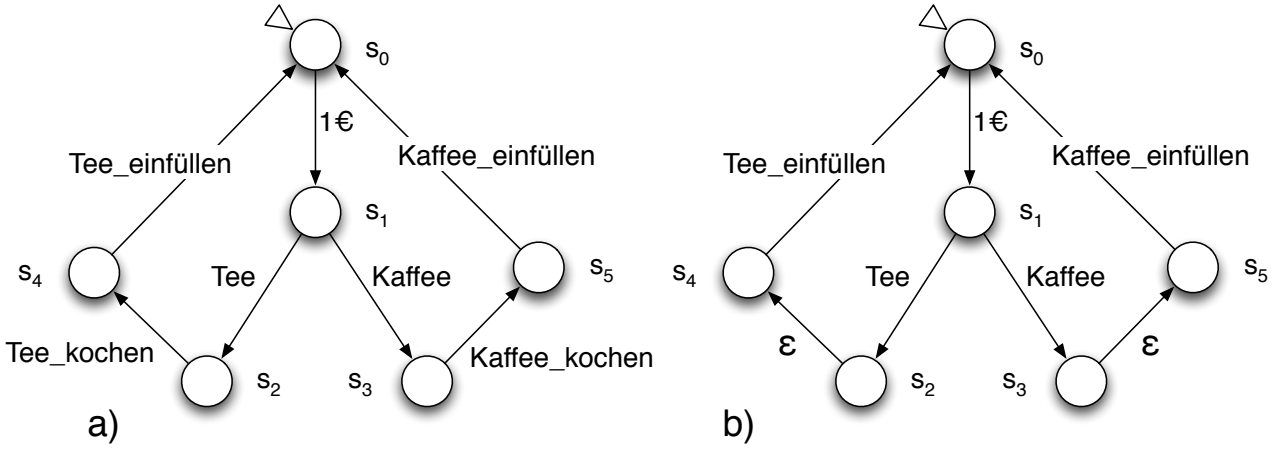


Abbildung 2.7: a) Getränkeautomat als Transitionssystem und mit internen Transitionen in b)

Analog definieren wir eine Abbildung für die Zustände. Eine *Zustandsetikettenfunktion* eines Transitionssystems $TS = (S, A, tr, S^0, S^F)$ ist eine Abbildung $E_S : S \to E$, die Zustände auf Etikettenmenge abbildet. Die *Zustandsetikettensprache* von TS ist $E_S(TS) := E_S^\omega(SS(TS)) := \{E_S^\omega(\pi) \in \Sigma^* \mid \pi \in SS(TS)\}$. Zwei TS heißen *trace-äquivalent*, wenn die Mengen ihrer Zustandsetikettenfolgen gleich sind.

2.5 Kripke-Strukturen

Wir kommen nun zu Analyseverfahren, die auf Transitionssystemen von beliebigen Systemen anwendbar sind. Wegen ihrer Wurzeln in der Logik heißen sie Kripke-Strukturen. Dabei werden weniger die Aktionsfolgen als vielmehr die Eigenschaften der Zustände betrachtet. Solche Eigenschaften werden durch atomare Aussagen in den Zuständen beschrieben.

2.5.1 Grundlegende Überlegungen

Wir betrachteten daher eine *Zustands*-Etikettenfunktion, die jedem Zustand, die Menge der lokal gültigen atomaren Aussagen zuweist (Sei AP eine Menge von atomaren Aussagen.):

$$E_S : S \to \mathcal{P}(AP)$$

Beispiel 2.17 Durch eine Zustandsetikettenfunktion können den Zuständen Mengen von atomaren Aussagen zugeordnet werden, die in diesem Zustand gültig sind. Beispielsweise könnte man im Transitionssystem von Abbildung 2.7 b) die atomaren Aussagen $AP := \{\alpha_1, \alpha_2, \alpha_3\}$ folgendermaßen definieren:
$\alpha_1 =$ „1€ gezahlt“, $\alpha_2 =$ „Tee gewählt“ und $\alpha_3 =$ „Kaffee gewählt“.
Die Zuordnung $E_S(s_0) = \emptyset$, $E_S(s_1) = \{\alpha_1\}$ usw. ist in Abbildung 2.8 angegeben. Das bedeutet, dass zum Beispiel bei der Analyse (Model-Checking) im Zustand s_3 nur die Eigenschaften $\{\alpha_1, \alpha_3\}$ zu prüfen sind.

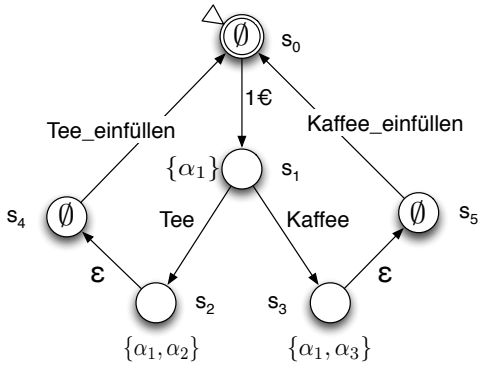


Abbildung 2.8: Getränkeautomat als Transitionssystem mit Zustandsetikettenfunktion

Eine Relation $R \subseteq X \times Y$ heißt *linkstotal*, wenn gilt: $\forall x \in X : \exists y \in Y : (x, y) \in R$.
Wir wollen nur linkstotale Transitionsrelationen betrachten, damit jede endliche Zustandsfolge Anfangsstück einer unendlichen Folge ist, d.h. keine „Sackgassen“ entstehen können. Wir können aber stets einen speziellen Sackgassen-Zustand einführen, der jedes TS mit Sackgassen in eines ohne überführt.

2.5.2 Definitionen

Eine *Kripke-Struktur* ist ein Transitionssystem $M := (S, A, tr, S_0, S^F)$ mit $S^F = \emptyset$ einelementiger Aktionenmenge (also $|A| = 1$), linkstotaler Transitionsrelation tr und zusätzlich einer Zustandsetikettenfunktion $E_S : S \to \mathcal{P}(AP)$.

Kripke-Strukturen werden üblicherweise in der folgenden Form notiert, die wir im Rest dieses Kapitels auch benutzen. Dabei wird die (hier nicht benötigte) Menge A der Aktionen weggelassen und daher die Transitionsrelation als 2-stellige, linkstotal Relation $R \subseteq S \times S$ dargestellt.

Definition 2.18 *Eine* Kripke-Struktur $M := (S, S_0, R, E_S)$ *besteht aus*

a) einer Zustandsmenge S,
b) einer Menge $S_0 \subseteq S$ *von* Anfangszuständen,
c) einer linkstotalen Transitionsrelation $R \subseteq S \times S$ *und*
d) einer Zustandsetikettenfunktion $E_S : S \to \mathcal{P}(AP)$, *die jedem Zustand* s *eine Menge* $E_S(s) \subseteq AP$ *von* aussagenlogischen atomaren Formeln *zuordnet.*

Anmerkung: In Darstellungen und in Model-Checking-Werkzeugen werden zur besseren Lesbarkeit oft Transitionsbezeichner benutzt (wie in den Abbildungen 2.8 und 3.1). Diese gehören aber nicht zur formalen Definition.

Da eine Kripke-Struktur keine Aktionen hat, werden zur Beschreibung des Verhaltens statt $L^\omega(M)$ die unendlichen Zustandsfolgen betrachtet.
Für Transitionssysteme ist ein Pfad eine Folge $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \cdots$, die in einem Startzustand beginnt. Für Kripke-Strukturen ist die Angabe von Aktionen überflüssig. Für eine Kripke-Struktur $M := (S, S_0, R, E_S)$ ist ein *Pfad* aus $s \in S$ (auch: eine *Rechnung* aus s) eine unendliche Folge:

$$\pi = s_0 s_1 s_2 \ldots \in S^\omega \quad \text{mit } s_0 = s \text{ und } \forall i \geq 0 : (s_i, s_{i+1}) \in R$$

Die Menge aller Rechnungen von M ist dann (analog zu TS):

$$SS(M) := \{\pi \mid \exists s \in S^0 : \pi \text{ ist eine Rechnung aus } s\}$$

Die zu $\pi = s_0 s_1 s_2 \ldots$ zugehörige Zustandsetikettenfolge ist dann:

$$E_S(\pi) = E_S(s_0) E_S(s_1) E_S(s_2) \ldots \in \mathcal{P}(AP)^\omega$$

Zwei Kripke-Strukturen heißen *trace-äquivalent*, wenn die Mengen ihrer Zustandsetikettenfolgen gleich sind.

Beispiel 2.19 Die Menge aller Rechnungen der Kripke-Struktur von Abb. 2.8 ist

$$(s_0 s_1 (s_2 s_4 + s_3 s_5))^\omega$$

Die Etikettensprache ist dann:

$$\begin{array}{ll} & E_S((s_0 s_1 (s_2 s_4 + s_3 s_5))^\omega) \\ = & (E_S(s_0) E_S(s_1)(E_S(s_2) E_S(s_4) + E_S(s_3) E_S(s_5)))^\omega \\ = & (\emptyset\{\alpha_1\}(\{\alpha_1, \alpha_2\}\emptyset + \{\alpha_1, \alpha_3\}\emptyset))^\omega \end{array}$$

Hinweis: Man beachte, dass hier Mengen als Symbole verwendet werden. Man könnte die Etikettensprache (mit abgewandelten Etiketten der Form $a_X, X \in \mathcal{P}(AP)$ statt nur X) auch folgendermaßen schreiben:

$$E_S((s_0 s_1 (s_2 s_4 + s_3 s_5))^\omega) \quad = \quad (a_\emptyset a_{\{\alpha_1\}}(a_{\{\alpha_1,\alpha_2\}} a_\emptyset + a_{\{\alpha_1,\alpha_3\}} a_\emptyset))^\omega$$

Nach diesem Hinweis bleiben wir aber bei der vorherigen (und üblichen) Darstellung.

Darstellung von Kripke-Strukturen durch logische Formeln

Eine Kripke-Struktur kann mittels Prädikaten erster Ordnung repräsentiert werden. Die Darstellung durch logische Formeln ist für die Verarbeitung in Modelchecking-Werkzeugen wichtig und wird dort zum Beispiel für die Komplexitätsreduktion durch „binäre Entscheidungsdiagramme“ (*binary decision diagrams, BDDs*) eingesetzt.

Zustand: $s : V \to D$ ist eine Abbildung von der Menge der Variablen in eine Wertemenge. Z.B. für die Variablenmenge $V = \{v_1, v_2, v_3\}$, wird der Zustand $s =< v_1 \leftarrow 2, v_2 \leftarrow 3, v_3 \leftarrow 5 >$ durch die zugehörige Formel $(v_1 = 2) \wedge (v_2 = 3) \wedge (v_3 = 5)$ repräsentiert. $\mathcal{S}_0$ bezeichnet die Formel für den Anfangszustand.

Transition: Beziehung zwischen Werten der Variablen vor der Transition (Menge V) und Werten der Variablen nach der Transition (Menge V'). Wenn R eine Transitionsrelation ist, so bezeichnet $\mathcal{R}(V, V')$ die entsprechende Formel.

Definition 2.20 *Eine* Kripke-Struktur in logischer Darstellung *ist ein Tupel* $M := (S, S_0, R, E_S)$ *mit:*

1. S *Menge der* Belegungen,
2. $S_0 \subseteq S$ *Menge von* Zuständen, *die die* Anfangsbedingung $\mathcal{S}_0$ *erfüllen,*
3. $\forall s, s' \in S : R(s, s') \Leftrightarrow \mathcal{R}(V, V')$ *mit* Belegung s *für* V *und* s' *für* V',
4. $E_S(s)$ *Menge der* Formeln, *die bei* Belegung s *gelten.*
   ($v \in E_S(s)$ *GDW.* $s(v) = wahr$, $v \notin E_S(s)$ *GDW.* $s(v) = falsch$*)*

Beispiel 2.21 (System mit Kripke-Struktur)
Variablenmenge: $V = \{x, y\}$,
Wertemenge: $D = \{0, 1\}$,
System: $x := (x + y) \bmod 2$,
Anfangszustand: $x = 1, y = 1$,
Das System kann mit zwei Prädikatenformeln beschrieben werden:

$$\mathcal{S}_0(x, y) \equiv x = 1 \wedge y = 1$$

$$\mathcal{R}(x, y, x', y') \equiv x' = (x + y)\, mod\, 2 \wedge y' = y$$

Daraus die Kripke-Struktur: $M = (S, S_0, R, E_S)$
$S = D \times D$,
$S_0 = \{(1, 1)\}$,
$R = \{((1, 1), (0, 1)), \quad ((0, 1), (1, 1)), \quad ((1, 0), (1, 0)), \quad ((0, 0), (0, 0))\}$,
$E_S((1, 1)) = \{x = 1, y = 1\}, \ldots$
Ein Pfad vom Anfangszustand: (1, 1), (0, 1), (1, 1), (0, 1), ...

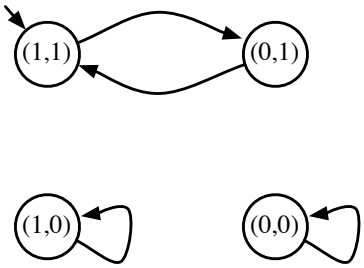


Abbildung 2.9: Kripke-Struktur zu Beispiel 2.21

2.6 Kripke-Strukturen von Programmen

Exemplarisch für Programme werden nun Kripke-Strukturen für eine einfache Programmsprache definiert. Zunächst wird dies für sequentielle Programme durchgeführt und dann auf nebenläufige Programme erweitert. Illustriert wird diese Formalisierung am Beispiel eines Programmes für den wechselseitigen Ausschluss. Dieses Beispiel erläutert zudem grundsätzliche Fehlerquellen, die typisch für nebenläufige Prigramme sind.

2.6.1 Sequentielle Programme als Kripke-Strukturen

Es werden nun Kripke-Strukturen für eine einfache Programmsprache definiert, die die elementaren Anweisungen *Zuweisung*, *Skip* (d.i. die leere Anweisung), *Hintereinanderausführung*, *bedingte Anweisung* und *Schleife* enthält.

Zur Kennzeichnung von Zuständen erhalten die Programme Zeilennummern.

Für Zeilennummern gibt es die Variable pc (Befehlszähler, program counter). Mit $pc = \perp$ ist gemeint, dass das Programm nicht aktiv ist.

Das Prädikat $same(Y) \equiv \forall y, y' \in Y : (y' = y)$ (wobei $Y \subseteq V$) beschreibt die nicht veränderten Variablen.

Das Prädikat $pre(V)$ beschreibt die Anfangsbelegung der Variablen V.

Der Anfangszustand ist $\mathcal{S}_0 \equiv pre(V) \wedge pc = m$ mit m als Startzeilennummer.

Die Prozedur $\mathcal{C}(l, P, l')$ liefert rekursiv für eine Programmbeschreibung P die Formel $\mathcal{R}$ zur Repräsentation der Transitionsrelation der gewünschten Kripke-Struktur, dabei sind:

l, l' Zeilennummer jeweils vor und nach der Anweisung

pc, pc' Befehlszähler - Variable

Zuweisung:

$\mathcal{C}(l, v \leftarrow e, l') \equiv pc = l \wedge pc' = l' \wedge v' = e \wedge same(V \backslash \{v\})$

Skip:

$\mathcal{C}(l, skip, l') \equiv pc = l \wedge pc' = l' \wedge same(V)$

Hintereinanderausführung: [2]

$\mathcal{C}(l,\ (P_1; l'' : P_2),\ l') \equiv \mathcal{C}(l, P_1, l'') \vee \mathcal{C}(l'', P_2, l')$

Bedingte Anweisung:

$\mathcal{C}(l,\ \textbf{if}\ b\ \textbf{then}\ l_1 : P_1\ \textbf{else}\ l_2 : P_2\ \textbf{endif},\ l') \equiv$
$(pc = l \wedge pc' = l_1 \wedge b \wedge same(V)) \vee$
$(pc = l \wedge pc' = l_2 \wedge \neg b \wedge same(V)) \vee$
$\mathcal{C}(l_1, P_1, l') \vee \mathcal{C}(l_2, P_2, l')$

Schleifen-Anweisung:

$\mathcal{C}(l,\ \textbf{while}\ b\ \textbf{do}\ l_1 : P_1\ \textbf{endwhile},\ l') \equiv$
$(pc = l \wedge pc' = l_1 \wedge b \wedge same(V)) \vee$
$(pc = l \wedge pc' = l' \wedge \neg b \wedge same(V)) \vee$
$\mathcal{C}(l_1, P_1, l)$

[2] Das innere Klammerpaar dient der besseren Lesbarkeit.

Beispiel 2.22
(ein kleines sequentielles Programm)
Wir bilden die Formeln für die Zuweisungsfolge
$l_1 : x := 2 \cdot y;$
$l_2 : y := y - 1;$
$l_3 \ \cdots$ mit $V = \{x, y\}$ und den Anfangswerten $x = 1, y = 2$.

Anfangszustand: $\mathcal{S}_0 \equiv (x = 1 \wedge y = 2 \wedge pc = l_1)$

Transitionsrelation: $\mathcal{R} \equiv \mathcal{C}(l_1, x \leftarrow 2 \cdot y, l_2) \ \vee \ \mathcal{C}(l_2, y \leftarrow y - 1, l_3)$
mit $\mathcal{C}(l_1, x \leftarrow 2 \cdot y, l_2) \equiv pc = l_1 \wedge pc' = l_2 \wedge x' = 2 \cdot y \wedge y' = y$
und $\mathcal{C}(l_2, y \leftarrow y - 1, l_3) \equiv pc = l_2 \wedge pc' = l_3 \wedge y' = y - 1 \wedge x' = x$

Daraus wird folgendermaßen der Anfang einer Kripke-Struktur generiert:

Anfangsschritt: Bilde alle Anfangszustände, die $\mathcal{S}_0$ erfüllen. Das können mehrere sein. In unserem Beispiel (siehe Abbildung 2.10) ist dies nur ein einziger $s_0 = (pc, x, y) = (l_1, 1, 2)$. Wäre z eine weitere Programmvariable, dann gäbe es so viele Anfangszustände wie Werte von z.

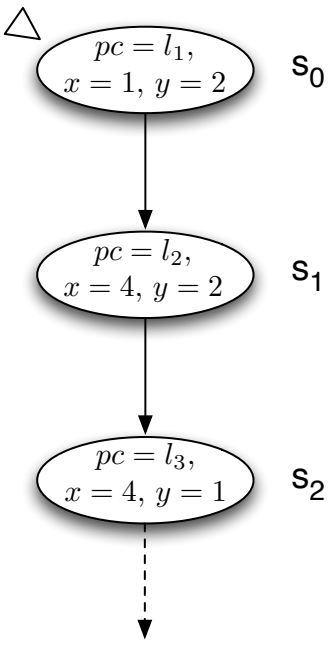


Abbildung 2.10: Anfang einer Kripke-Struktur zum Programm von Beispiel 2.22

Rekursionsschritt: Wähle einen bereits konstruierten Zustand (pc, x, y). Bestimme (pc', x', y') derart, dass die Formel der Transitionsrelation erfüllt ist.

Im Beispiel haben wir nach dem Anfangsschritt nur $(pc, x, y) = (l_1, 1, 2)$ und $s_1 = (pc', x', y') = (l_2, 4, 2)$, da nur für einen Term der Disjunktion $pc = l_1$ erfüllt ist[3]. Entsprechend wird aus $s_1 = (l_2, 4, 2)$ der Nachfolgezustand $s_2 = (l_3, 4, 1)$ konstruiert.

Insgesamt gesehen entspricht die Disjunktion $\mathcal{R}$ der Vereinigung der einzelnen Transitionsübergänge $\{(s, s')\} = \{((pc, x, y), (pc', x', y'))\}$.

2.6.2 Nebenläufige Programme als Kripke-Strukturen

Zu den Anweisungen von Abschnitt 2.6.1 werden nun noch die Anweisungen **cobegin/coend** für nebenläufige Ausführung und **await** für bedingtes Warten hinzugenommen.

Die Programmbeschreibung

$$P = l : \textbf{cobegin } P_1 \| P_2 \| \ldots \| P_n \textbf{ coend}; \ l'$$

wird oft mit Zeilennummern versehen:

$$P = l : \textbf{cobegin } l_1 : P_1 \ l_1' \| \ldots \| l_n : P_n \ l_n'; \textbf{coend}; \ l'$$

[3]Würde bei der Definition der Hintereinanderausführung $\vee$ durch $\wedge$ ersetzt, so wäre wegen $l_1 \neq l_2$ keine gültige Belegung der Formel möglich!

Dafür ergeben sich die Formeln, wobei $PC = \{pc, pc_i | pc_i\text{-Befehlszähler von } P_i\}$):

$$\mathcal{S}_0(V, PC) \equiv pre(V) \wedge pc = l \wedge \bigwedge_{i=1}^{n} (pc_i = \bot)$$

P_i sind also am Anfang nicht aktiv. Damit ergibt sich folgende Repräsentation:

$\mathcal{C}(l, P, l') \equiv$
$(pc = l \wedge pc'_1 = l_1 \wedge \ldots \wedge pc'_n = l_n \wedge pc' = \bot) \vee$ (Initialisierung)
$(pc = \bot \wedge pc_1 = l'_1 \wedge \ldots \wedge pc_n = l'_n \wedge pc' = l' \wedge \bigwedge_{i=1}^{n}(pc'_i = \bot)) \vee$ (Termination)
$(\bigvee_{i=1}^{n}(\mathcal{C}(l_i, P_i, l'_i) \wedge same(V \backslash V_i) \wedge same(PC \backslash \{pc_i\})))$ (Transitionen von P_i)

V_i ist die Menge der Variablen die von Programm P_i geändert werden.

await-Anweisung:

$\mathcal{C}(l, \mathbf{await}(b), l') \equiv$
$(pc_i = l \wedge pc'_i = l \wedge \neg b \wedge same(V_i))$ "busy waiting"
$\vee\ (pc_i = l \wedge pc'_i = l' \wedge b \wedge same(V_i))$

Als Beispiel behandeln wir den „wechselseitigen Ausschluss“ (*mutual exclusion*), eine Erscheinung, die von grundsätzlicher Bedeutung ist.

2.6.3 Wechselseitiger Ausschluss

Parallele Programme oder Prozesse können zum konsistenten Schreiben auf gemeinsame Daten einen *kritischen Abschnitt* enthalten, der nicht überlappend ausgeführt werden darf. Dies kommt im nicht kritischen Abschnitt nicht vor.

Der Algorithmus von Dekker/Petersen war der erste, der dies ohne betriebssystemunterstützte Operationen (wie Semaphore oder synchronized-Methoden in Java) realisierte.

Die Programme sollen dabei (für den Fall zweier Prozesse P und Q) folgende Eigenschaften erfüllen:

A) Die Befehlszähler von P und Q sind nie gleichzeitig in ihren kritischen Abschnitten.

B) Meldet der Prozess P oder Q den Wunsch zum Eintritt in den kritischen Abschnitt an ($wantP = True$ oder $wantQ = True$), so kann er nach einer gewissen endlichen Zeit tatsächlich in seinen kritischen Abschnitt eintreten.

Bei diesen Programmen gehen wir davon aus, dass alle elementaren Anweisungen (also Zuweisungen und Tests) durch einen Speichersperrmechanismus ungeteilt (atomar) ausgeführt werden. Vorausgesetzt wird natürlich auch, dass der kritische Abschnitt nach einer gewissen Zeit auch wieder verlassen wird. A) und B) sind Minimalforderungen. Eine weitere Forderung C) ist zum Beispiel, dass der Eintritt in den kritischen Abschnitt nicht abwechselnd erfolgen muss.

Wir geben hier eine Einführung nach Dijkstra wieder und benutzen dabei das Schema von Algorithmus 2.1.

Ein erster Lösungsversuch folgt der Vorstellung einer Ampel, die auf grün und rot gesetzt wird (Algorithmus 2.2). Was geht hier schief?

Der zweite Versuch in Algorithmus 2.3 führt das Anmelden eines Zutrittswunsches durch $wantP$ und $wantQ$ ein. Ein Prozess prüft, ob dieser bei dem anderen vorliegt. Was läuft hier falsch?

Der dritte Versuch in Algorithmus 2.4 versucht die Verklemmung durch das temporäre Aufheben des Zutrittswunsches zu vermeiden. Warum funktioniert das nicht?

Algorithmus 2.1 (Wechselseitiger Ausschluss nach Dijkstra - Programmschema)

P0: Initialisierung
m : **cobegin** $P \| Q$ **coend**
wobei

P: l_0 : **while** True **do**
p_i : non-critical section;
$Eintrittsprotokoll$
p_j : critical section;
$Austrittsprotokoll$
endwhile l_0'

Q: l_1 : **while** True **do**
q_i : non-critical section;
$Eintrittsprotokoll$
q_j : critical section;
$Austrittsprotokoll$
endwhile l_1'

Algorithmus 2.2 (Wechselseitiger Ausschluss nach Dijkstra - 1)

P1: *Ampel = grün:* {*rot, grün*},
m : **cobegin** $P \| Q$ **coend**
wobei

P: l_0 : **while** True **do**
p_0 : non-critical section;
p_2 : **await***(Ampel = grün)*;
p_3 : $Ampel := rot$;
p_4 : critical section;
p_5 : *Ampel := grün*;
endwhile l_0'

Q: l_1 : **while** True **do**
q_0 : non-critical section;
q_2 : **await***(Ampel = grün)*;
q_3 : $Ampel := rot$;
q_4 : critical section;
q_5 : *Ampel := grün*;
endwhile l_1'

Algorithmus 2.3 (Wechselseitiger Ausschluss nach Dijkstra - 2)

P2: $wantP = wantQ = False$: *boolean*,
m : **cobegin** $P \| Q$ **coend**
wobei

P: l_0 : **while** True **do**
p_0 : non-critical section;
p_1 : $wantP := True$;
p_3 : **await**$(wantQ = False)$
p_4 : critical section;
p_5 : $wantP := False$;
endwhile l_0'

Q: l_1 : **while** True **do**
q_0 : non-critical section;
q_1 : $wantQ := True$;
q_3 : **await**$(wantP = False)$
q_4 : critical section;
q_5 : $wantQ := False$;
endwhile l_1'

Algorithmus 2.4 (Wechselseitiger Ausschluss nach Dijkstra - 3)

P3: $wantP = wantQ = False$: *boolean*,
m : **cobegin** $P \| Q$ **coend**
wobei

P: l_0 : **while** True **do**
p_0 : non-critical section;
p_1 : $wantP := True$;
p_3 : **while** $wantQ = True$ **do**
$wantP := False$;
$wantP := True$
endwhile
p_4 : critical section;
p_5 : $wantP := False$;
endwhile l_0'

Q: l_1 : **while** True **do**
q_0 : non-critical section;
q_1 : $wantQ := True$;
q_3 : **while** $wantP = True$ **do**
$wantQ := False$;
$wantQ := True$
endwhile
q_4 : critical section;
q_5 : $wantQ := False$;
endwhile l_1'

Der vierte Versuch, der ursprünglich von Dekker stammt und von Peterson auf die vorliegende Form verbessert wurde, löst die Verklemmung im 2. Algorithmus durch eine „tie break rule“, indem derjenige Prozess im Konfliktfall in den kritischen Abschnitt eintreten darf, der sich nicht zuletzt für den Eintritt angemeldet hat (Variable *last* im Algorithmus 2.5). Ob dieser Algorithmus das Problem löst, wird im Folgenden mit seiner Kripke-Struktur untersucht.

Algorithmus 2.5 (Wechselseitiger Ausschluss nach Dekker/Peterson (Dijkstra - 4))

P4: $wantP = wantQ = False : boolean$,
$last = 1 \vee last = 2 : integer$,
m : **cobegin** $P \| Q$ **coend**
wobei

P:
l_0 : **while** True **do**
[p_0 : non-critical section;]
$p_1 : wantP := True$;
$p_2 : last := 1$;
p_3 : **await**$(wantQ = False \vee last = 2)$;
[p_4 : critical section;]
$p_5 : wantP := False$;
endwhile l_0'

Q:
l_1 : **while** True **do**
[q_0 : non-critical section;]
$q_1 : wantQ := True$;
$q_2 : last := 2$;
q_3 : **await**$(wantP = False \vee last = 1)$;
[q_4 : critical section;]
$q_5 : wantQ := False$;
endwhile l_1'

Kripke-Strukturen (d.h. Zustandsräume) von parallelen Programmen wachsen sehr schnell. Daher sucht man nach Methoden, um diese Größe zu reduzieren, ohne die Analysemöglichkeit einzuschränken. Dies ist im kleinen Maßstab auch bei diesem Beispiel möglich. Wir können nämlich die Zeilen p_0, p_4, q_0 und q_4 im Algorithmus 2.5 streichen und für dieses reduzierte Programm immer noch den wechselseitigen Ausschluss prüfen, indem wir beweisen, dass die Befehlszähler nie gleichzeitig in p_5 und q_5 sind. Das gilt auch für andere Eigenschaften.
Im Folgenden ist das (reduzierte) Programm

$$P = l : \textbf{cobegin } l_0 : P\ l_0' \| \ldots \| l_1 : Q\ l_1'; \textbf{coend};\ l'$$

in der zuvor eingeführten Notation dargestellt.
$PC = \{pc, pc_0, pc_1\}, \quad V = V_0 = V_1 = \{wantP, wantQ, last\}$
pc_0 nimmt die Werte $\{p_1, p_2, p_3, p_5\}$ an und pc_1 die Werte $\{q_1, q_2, q_3, q_5\}$.

Der Anfangszustand ist:

$$\mathcal{S}_0(V, PC) \equiv pc = l\ \wedge\ pc_0 = \perp\ \wedge pc_1 = \perp$$

Die Transitionsrelation ist repräsentiert durch $\mathcal{R}(V, PC, V', PC')$ als Disjunktion der folgenden Formeln:

- $pc = l \wedge pc_0' = l_0 \wedge pc_1' = l_1 \wedge pc' = \perp$
- $pc_0 = l_0' \wedge pc_1 = l_1' \wedge pc' = l' \wedge pc_0' = \perp \wedge pc_1' = \perp$
- $\mathcal{C}(l_0, P, l_0') \wedge same(V \backslash V_0) \wedge same(PC \backslash \{pc_0\})$ also $\mathcal{C}(l_0, P, l_0') \wedge same(pc, pc_1)$
- $\mathcal{C}(l_1, Q, l_1') \wedge same(V \backslash V_1) \wedge same(PC \backslash \{pc_1\})$ also $\mathcal{C}(l_1, Q, l_1') \wedge same(pc, pc_0)$

Dabei ist $\mathcal{C}(l_0, P, l_o')$ die Disjunktion von folgenden Formeln:

- $pc_0 = l_0 \wedge pc_0' = p_1 \wedge True \wedge same(V)$ (Schleifen-Anweisung)
- $pc_0 = p_1 \wedge pc_0' = p_2 \wedge wantP' = True \wedge same(V \backslash \{wantP\})$ (Zuweisung)
- $pc_0 = p_2 \wedge pc_0' = p_3 \wedge last' = 1 \wedge same(V \backslash \{last\})$ (Zuweisung)

- $pc_0 = p_3 \wedge pc_0' = p_3 \wedge \neg(wantQ = False \vee last = 2) \wedge same(V)$ (await-Anweisung)
- $pc_0 = p_3 \wedge pc_0' = p_5 \wedge (wantQ = False \vee last = 2) \wedge same(V)$ (await-Anweisung)
- $pc_0 = p_5 \wedge pc_0' = l_0 \wedge wantP' = False \wedge same(V \backslash \{wantP\})$ (Zuweisung)

$\mathcal{C}(l_1, Q, l_1')$ ist entsprechend definiert.

Abbildung 2.11 zeigt die zugehörige Kripke-Struktur. Dabei soll eine Knotenbeschriftung (i, j, n, b_1, b_2) den Zustand $(p_i, q_j, last = n, wantP = b_1, wantQ = b_2)$ bezeichnen.

Gelten die aufgestellten Forderungen?

A) Die Befehlszähler von P und Q sind nie gleichzeitig in ihren kritischen Abschnitten. Für die Kripke-Struktur heißt dies, dass in keinem Zustand der Prozess P im Zustand $p_i = p_5$ und gleichzeitig der Prozess Q im Zustand $q_j = q_5$ ist. In der temporalen Logik wird dies als $\Box\neg(p_5 \wedge q_5)$ ausgedrückt.

B) Meldet der Prozess P oder Q den Wunsch zum Eintritt in den kritischen Abschnitt an ($wantP = True$ oder $wantQ = True$), so kann er nach einer gewissen endlichen Zeit tatsächlich in seinen kritischen Abschnitt eintreten. Dies bedeutet z.B. für den Prozess P in der Kripke-Struktur folgendes: von jedem Knoten mit $p_i = p_1$ stößt jeder Pfad irgendwann einmal auf einen Knoten mit $p_j = p_5$. In der temporalen Logik wird dies für den Prozess P als $\Box(p_1 \Rightarrow \Diamond(p_5))$ ausgedrückt.

Dies kann in der Kripke-Struktur von Abbildung 2.11 verifiziert werden. Der folgende Abschnitt stellt die Grundlagen für die algorithmische Lösung solcher Probleme da.

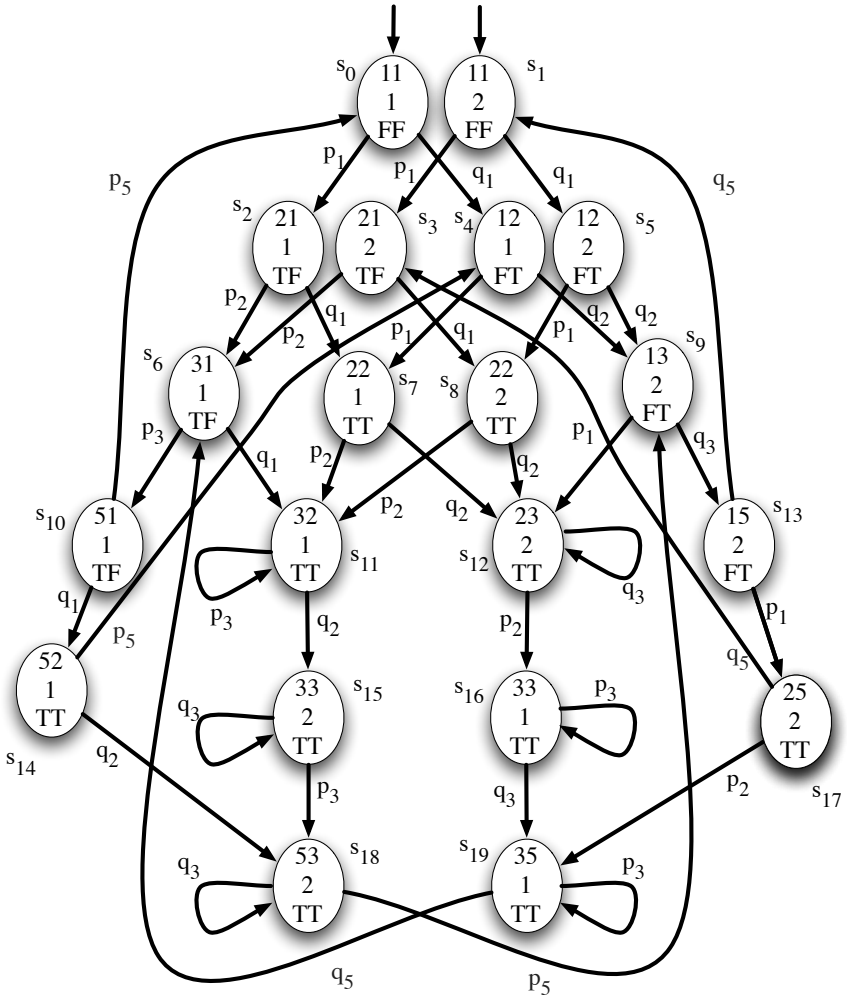


Abbildung 2.11: Kripke-Struktur zum Beispielprogramm „wechselseitiger Ausschluss“