wird oft mit Zeilennummern versehen:

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

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.

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'

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

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'

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?

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'

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

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:

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

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

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

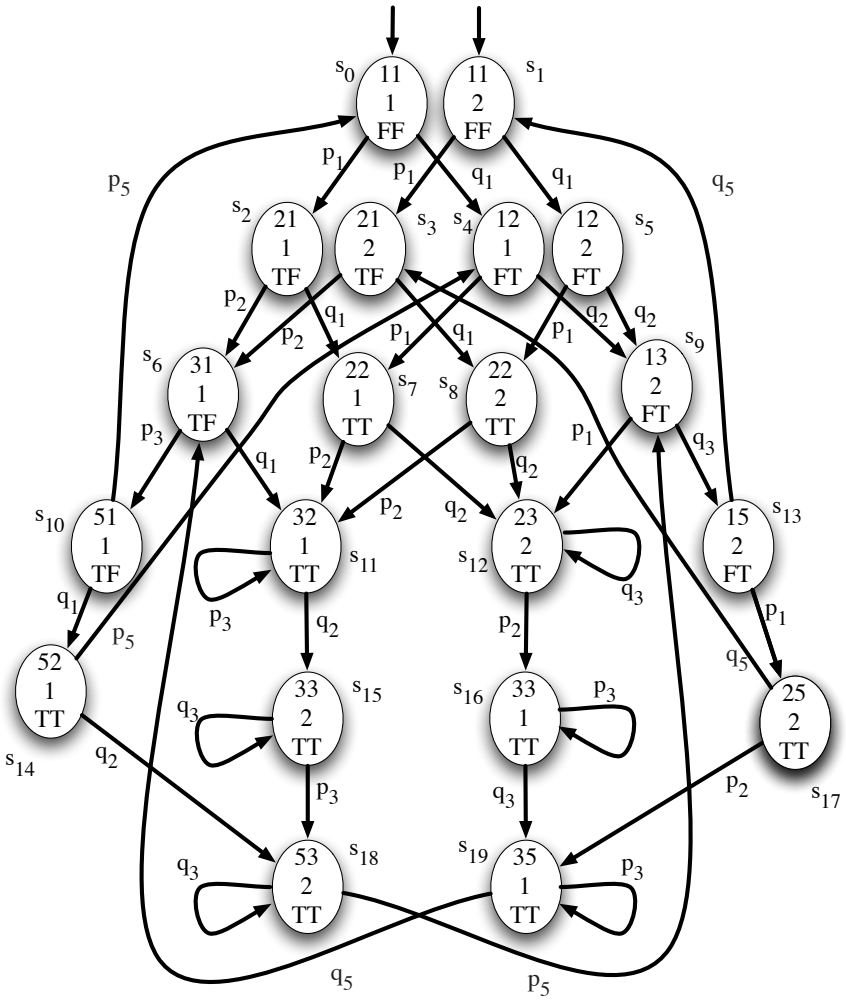


Abbildung 2.11: Kripke-Struktur zum Beispielprogramm „wechselseitiger Ausschluss“

3 Temporallogik

Die Validierung von Hardware- und Softwaresystemen wird zunehmend wichtig. Unter einer System-Validierung versteht man traditionell: **Testen und Simulation**. Dies ist am Anfang bei einfachen Fehlern sehr wirksam, ist aber im Laufe von Projekten immer weniger effektiv, da dann oftmals komplexe und verborgene Fehler auftreten. Fehler treten insbesondere bei Systemen mit Asynchronität, Parallelität, Nebenläufigkeit auf. Dies sind häufig Fehler, die von speziellen Zeitparametern/Nachrichtenlaufzeiten abhängig sind.

Eine Alternative zum Testen bietet die **formale Verifikation**, die eine umfassende Prüfung des Systemverhaltens beinhaltet. Zentraler Ansatz der formalen Verifikation sind das *Theorembeweisen* (bspw. in Zusammenhang mit der axiomatische Semantik nach Hoare-Systeme) und die Zustandsraumanalyse (engl. Model-Checking) (meist in Zusammenhang mit temporaler Logik). Ersteres wird in der Vorlesung *Semantik von Programmsprachen* (Modul FGI 3) behandelt. Eine Einführung zu letzterem wird in diesem Kapitel gegeben.

Model-Checking hat folgende Vor- und Nachteile: Von Vorteil ist, dass es ohne besondere Kenntnisse anwendbar ist und bei nicht korrekten Systemen Abläufe, die zu den Fehlern führen, generieren kann. Der Nachteil ist, dass praktische Systeme oftmals einen sehr großen Zustandsraum besitzen, der nicht mehr (zumindest nicht mehr unmittelbar) effizient behandelt werden kann.[1]

Literatur E.M. Clarke et al.: *Model Checking*, The MIT Press, Cambridge, 1999, [CGP99]

C. Baier, J.-P. Katoen: *Principles of Model Checking*, MIT Press, 2008, [BK08]

B. Bèrard et al.: *Systems and Software Verification: Model-Checking Techniques and Tools*, Springer, Berlin, 1999, [BBF99]

C. Girault, R. Valk: *Petri Nets for Systems Engineering*, Part III: Verification, Springer, Berlin, 2003, [GV03]

M. Huth, M. Ryan: *Logic in Computer Science*, Cambridge Univ. Press, 2004, [HR04]

[1] Um große Zustandsräume zu meistern, existieren einige Techniken (wie *symbolisches* Model-Checking, *Faltung*, Ausnutzung von Symmetrien), die wir hier aber nicht behandeln.

3.1 Temporale Logik

Die *temporale Logik* erlaubt es, Aussagen, die sich auf später einzunehmende Zustände beziehen, direkt auszudrücken. Solche Aussagen sind auch in der Prädikatenlogik möglich, wie dies zur Beschreibung von Lebendigkeits-Invarianzeigenschaften ausgeführt wird (siehe 7.1.3) oder wie die folgende Spezifikation eines Aufzuges zeigt.

Beispiel 3.1 (Spezifikation eines Aufzuges (Fragment))
Die folgenden zwei Teile einer Spezifikation des gewünschten Verhaltens eines Aufzuges seien gegeben:

I. Jede Anforderung des Aufzugs wird auch erfüllt.

II. Der Aufzug passiert kein Stockwerk mit einer nicht erfüllten Anforderung.

Die Spezifikationen I und II können mit Hilfe der Variablen t als Parameter für die ablaufende Zeit in Prädikatenlogik ausgedrückt werden, wie dies z.B. in der Physik erfolgt: $z(t) = -\frac{1}{2}gt^2$ (freier Fall des Aufzuges).

I. $\forall t, \forall n : app(n,t) \Rightarrow \exists t' > t : serv(n,t')$

II. $\forall t, \forall t' > t, \forall n \Big(\Big(app(n,t) \wedge H(t') \neq n \wedge \exists t_{trav} \,.\, t \leq t_{trav} \leq t' \wedge H(t_{trav}) = n \Big)$
$\Rightarrow \Big(\exists t_{serv} \,.\, t \leq t_{serv} \leq t' \wedge serv(n, t_{serv}) \Big) \Big)$

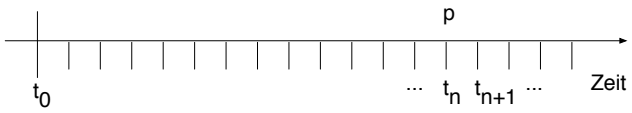
Dabei bedeuten $H(t)$ die Position des Fahrstuhls zur Zeit t, $app(n,t)$, dass eine offene Anforderung von Stockwerk n zur Zeit t besteht und $serv(n,t)$, dass der Fahrstuhl Stockwerk n bedient.

In der temporalen Logik wird der Zeit-Parameter nicht explizit benutzt. Dadurch werden die Formeln einfacher und die Entscheidungsprozeduren zur Gültigkeitsprüfung effektiv durchführbar.

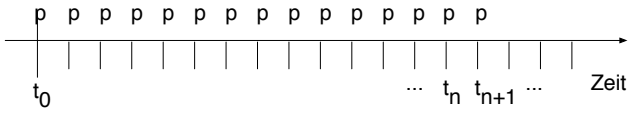
3.2 Linear Time Logic (LTL)

Amir Pnueli hat 1977 die temporale Logik erstmals für die Programmverifikation vorgeschlagen, indem er die Zeit als Programmschritte interpretierte. Von ihm stammt die *linear time logic* (LTL). Elementare *temporale Quantoren* sind $\Diamond p$ und $\Box p$:

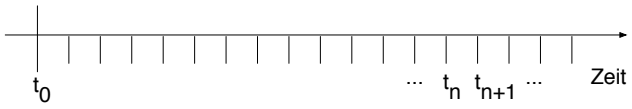
- $\Diamond p$ bedeutet: Irgendwann einmal gilt p



- $\Box p$ bedeutet: Von jetzt an gilt immer p



- $\Diamond\Box p$ bedeutet: ???



- $\Box\Diamond p$ bedeutet: ???

3.2.1 Syntax von LTL-Formeln

Wir definieren die Syntax von LTL-Formeln in Bezug auf eine Menge AP von aussagenlogischen Elementaraussagen (atomaren Formeln).

Definition 3.2 *Es sei AP eine Menge von aussagenlogischen Atomen. Dann wird die* Syntax von LTL-Formeln *wie folgt durch Backus-Naur-Form definiert:*

$$f ::= \mathbf{true}|\mathbf{false}|p|(\neg f)|(f \wedge f)|(f \vee f)|(Xf)|(Ff)|(Gf)|(fUf),$$

wobei $p \in AP$ ist.

3.2.2 Semantik von LTL-Formeln

LTL-Quantoren X, F, G und U beziehen sich auf unendliche Folgen $\alpha = A_0A_1A_2\cdots$ von Mengen $A_i \in \mathcal{P}(AP)$, d.h. jedes A_i ist eine Menge von gültigen atomaren Aussagen.

- Xp „next time“: p gilt im zweitem Element A_1 der Folge (auch $\bigcirc p$ geschrieben),
- Fp „eventually, in the future“: p gilt in einem Element einer gegebenen Folge (auch $\Diamond p$),

- Gp „always, globally“: p gilt in allen Elementen einer gegebenen Folge (auch $\Box p$),
- $p\ Uq$ „until“: es gibt ein Element der gegebenen Folge, in dem q gilt, und vor diesem Element gilt immer p. Illustriert:

Dies wird in der folgenden Definition induktiv auf LTL-Formeln übertragen.

Definition 3.3 *Sei $\alpha = A_0A_1A_2\cdots \in \mathcal{P}(\mathcal{AP})^\omega$ eine unendliche Folge von Aussagenmengen.*
Für $i \in \mathbb{N}$ sei die Folge $\alpha^i = A_iA_{i+1}\cdots$ der i-Suffix von α.
In der folgenden induktiven Definition seien $p \in AP$ und f, f_1, f_2 LTL-Formeln.

1.	$\alpha \models \mathbf{true}$.		
2.	$\alpha \not\models \mathbf{false}$.		
3.	$\alpha \models p$	$\Leftrightarrow$	$p \in A_0$.
4.	$\alpha \models \neg f$	$\Leftrightarrow$	$\alpha \not\models f$.
5.	$\alpha \models f_1 \vee f_2$	$\Leftrightarrow$	$\alpha \models f_1$ *oder* $\alpha \models f_2$.
6.	$\alpha \models f_1 \wedge f_2$	$\Leftrightarrow$	$\alpha \models f_1$ *und* $\alpha \models f_2$.
7.	$\alpha \models Xf$	$\Leftrightarrow$	$\alpha^1 \models f$.
8.	$\alpha \models Ff$	$\Leftrightarrow$	$\exists k \geq 0 : \alpha^k \models f$.
9.	$\alpha \models Gf$	$\Leftrightarrow$	$\forall k \geq 0 : \alpha^k \models f$.
10.	$\alpha \models f_1 U f_2$	$\Leftrightarrow$	$\exists k \geq 0\,.\,\alpha^k \models f_2$ *und für alle* $0 \leq j < k$ *gilt* $\alpha^j \models f_1$.

Hierbei ist $\alpha \models f$ zu lesen als „für die Folge α gilt f“ oder „α erfüllt f“.
Die Sprache $L^\omega(f) := \{\alpha \in \mathcal{P}(AP)^\omega \mid \alpha \models f\}$ heißt: Menge der α erfüllenden Folgen über AP oder: die Sprache von α.

Jede LTL-Formel f definiert also eine ω-Sprache (nämlich $L^\omega(f)$) im Sinne der Definition 1.14 (Seite 15).
Wegen der folgenden Äquivalenzen genügen die Operatoren $\vee, \neg, X, U$, um alle Formeln von LTL auszudrücken:

- $f_1 \wedge f_2 \equiv \neg(\neg f_1 \vee \neg f_2)$,
- $F\,f \equiv \mathbf{true}\,U\,f$,
- $G\,f \equiv \neg F\,\neg f$.

Zur Definition dieser Äquivalenz siehe Definition 3.5 (am Ende).

Aufgabe 3.4 Beweisen Sie diese Äquivalenzen.

Eine LTL-Formel f gilt für eine unendliche Folge $\pi = s_0s_1s_2\cdots$ von Zuständen einer Kripke-Struktur $M := (S, S_0, R, E_S)$, wenn sie für die zugehörige Zustandsetikettenfolge $E_S(\pi)$ gilt:

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

Man schreibt $M, \pi \models f$.

Definition 3.5 *Sei M eine Kripke-Struktur und $\pi \in SS(M)$ eine Pfad von M. Dann sei $M, \pi \models f :\Leftrightarrow E_S(\pi) \models f$.*

Eine LTL-Formel f gilt *(ist* gültig*) im Zustand $s \in S$ einer Kripke-Struktur M falls $M, \pi \models f$ für alle Pfade $\pi \in SS(M)$ gilt, die in s beginnen (notiert als $M, s \models f$).*

Eine LTL-Formel f gilt in M, falls sie in allen Anfangszuständen gilt, also: $M \models f :\Leftrightarrow \forall s \in S_0 : M, s \models f$.

Gilt für alle Kripke-Strukturen M die Beziehung: $M \models f$ impliziert $M \models g$, dann notieren wir dies als $f \models g$.

Gilt für alle Kripke-Strukturen M die Beziehung: $M \models f$ gdw. $M \models g$, dann notieren wir dies als $f \equiv g$.

Beispiel 3.6 MW-Ofen Die Abbildung 3.1 zeigt eine Kripke-Struktur M als Systembeschreibung eines Mikrowellenofens. Die Transitionsbezeichner und negierten Aussagen in $E_S(s)$ dienen nur der Verdeutlichung und können wie in der Definition einer Kripke-Struktur entfallen. Die als LTL-Formel $f = G(\neg Heat\ U\ Close)$ gegebene Spezifikation gilt für M, also $M, \pi \models f$ für jede unendliche Folge $\pi = s_0 s_1 s_2 \cdots \in SS(M)$, da beginnend mit $s_0 = 1$ immer $\neg Heat$ gilt bis $Close$ eintritt, was auch in jedem solchen Pfad tatsächlich geschieht. Dagegen gilt die Spezifikation $g = G(Start \rightarrow F\ Heat)$ nicht. Ein Gegenbeispiel ist die Folge $\pi = s_0 s_1 s_2 \cdots = 1(25)^\omega \in S^\omega$ oder auch $(1253)^\omega \in S^\omega$.
Es gilt $M, s \models f$ für $s = 1$, also $M \models f$, nicht jedoch $M \models g$.

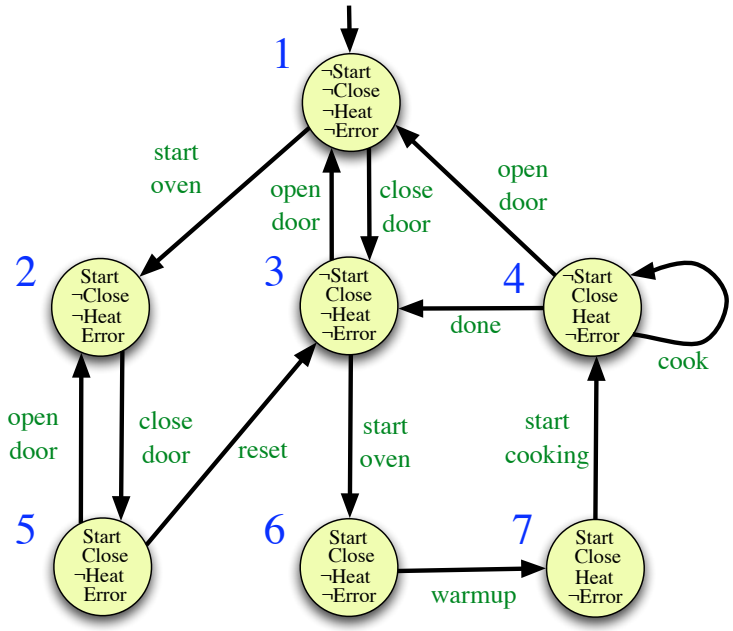


Abbildung 3.1: Kripke-Struktur für das Mikrowellenofenbeispiel

Fairnesseigenschaften wie von Definition 6.34 lassen sich gut in LTL formulieren. Jedoch nicht alle sinnvollen Eigenschaften lassen sich durch LTL-Formeln spezifizieren. Ein Beispiel ist folgende Eigenschaft: Der Aufzug kann im 3. Stock bleiben und immer die Türen auf- und zumachen. Ein weiteres Beispiel ist: Von jedem Zustand ist es möglich einen „Neustart"-Zustand zu erreichen, d.h. es gibt immer einen solchen Pfad. Diese Eigenschaft gehört zu den Lebendigkeits-Invarianzeigenschaften von Definition 7.6. Hier wird ein Existenz-Quantor für Pfade benötigt, wie er in CTL möglich ist.

Lemma 3.7 *Sind die etikettierten Zustandsfolgen zweier Kripke-Strukturen M_1 und M_2 gleich, dann können M_1 und M_2 durch keine LTL-Formel unterschieden werden, d.h. für alle LTL-Formeln ψ gilt:*

$$M_1 \models \psi \quad \Longleftrightarrow \quad M_2 \models \psi$$

Beweis: Wenn $E_1(SS(M_1)) = E_2(SS(M_2))$ gilt, dann folgt:

$$\begin{array}{ll} & M_1 \models \psi \\ \Longleftrightarrow & \forall \pi \in SS(M_1) : M_1, \pi \models \psi \\ \Longleftrightarrow & \forall \pi \in SS(M_1) : E_1(\pi) \models \psi \\ \Longleftrightarrow & \forall \sigma \in E_1(SS(M_1)) : \sigma \models \psi \\ \Longleftrightarrow & \forall \sigma \in E_2(SS(M_2)) : \sigma \models \psi \\ \Longleftrightarrow & \forall \pi \in SS(M_2) : E_2(\pi) \models \psi \\ \Longleftrightarrow & \forall \pi \in SS(M_2) : M_2, \pi \models \psi \\ \Longleftrightarrow & M_2 \models \psi \end{array}$$

Also sind die Modelle äquivalent. □

3.3 Computation Tree Logic (CTL)

Emerson und Halpern haben 1986 die „computation tree logic“ (CTL) eingeführt, die andere Eigenschaften hat. Später wurde dann CTL* als eine temporale Logik eingeführt, die LTL und CTL umfasst.

3.3.1 Syntax von CTL-Formeln

CTL besitzt zusätzlich *Zustandsformeln* mit Quantoren, die sich auf die von einem Zustand ausgehenden Pfade beziehen:

- *Ap* bedeutet: „Für alle Pfade, die von einem gegeben Zustand s ausgehen, gilt die Formel p.“
- *Ep* bedeutet: „Es gibt einen Pfad, der von einem gegeben Zustand s ausgeht und für den die Formel p gilt.“

Als gegebener Zustand fungiert dabei ein Anfangszustand oder ein Zustand, der durch eine umgebende Formel spezifiziert ist.
Um eine bessere Komplexität der Model-Check-Algorithmen zu erhalten, werden diese mit den bisher betrachteten Operatoren zu Formeln der Form EGf, $E[fUg]$ usw. verbunden. Dadurch ist folgende Syntaxdefinition motiviert.

Definition 3.8 *Es sei AP eine Menge von aussagenlogischen Atomen.*
Dann wird die Syntax von CTL-Zustands-Formeln *wie folgt durch Backus-Naur-Form definiert, wobei* $p \in AP$ *und* f *eine* CTL-Pfad-Formel *ist:*

$$g ::= \mathbf{true}|\mathbf{false}|p|(\neg g)|(g \wedge g)|(g \vee g)|(Ef)|(Af)$$

Eine CTL-Pfad-Formel *wird durch*

$$f ::= (Xg)|(Fg)|(Gg)|(g_1 U g_2)$$

definiert. Dabei sind g, g_1, g_2 *CTL-Zustands-Formeln.*

Beispiel 3.9

- $EF(Start \wedge \neg Ready)$
  Es ist möglich in einen Zustand zu kommen, in dem „*Start*“ aber nicht „*Ready*“ gilt.
- $AG(Req \rightarrow AF\, Ack)$
  Immer wenn ein Request *Req* erfolgt, dann wird er später einmal mit *Ack* bestätigt.
- $AG(AF\, DeviceEnabled)$
  Die Aussage „*DeviceEnabled*“ gilt unendlich oft auf jedem Pfad.
- $AG(EF\, Restart)$
  Von jedem Zustand aus ist es möglich, einen Zustand mit „*Restart*“ zu erreichen.

Die Logik CTL* trennt bei der Definition der Formeln nicht mehr zwischen Pfad- und Zustands-Formeln. Somit ist CTL syntaktisch eine Teillogik von CTL*. Aber auch LTL ist syntaktisch eine Teillogik von CTL*, nämlich indem wir jede LTL-Formel f als die CTL*-Formel Af auffassen.

3.3.2 Semantik von CTL-Formeln

Die Bedeutung von CTL-Formeln wird gerne als Beschreibung von Eigenschaften eines *Berechnungsbaumes* (computation tree) formuliert. Letzterer ist eine schleifenfreie Darstellung des Systemverhaltens. Berechnungsbäume entstehen durch „Abwickeln“ der Kripke-Struktur-Beschreibung des Systemverhaltens wie in Abbildung 3.2.

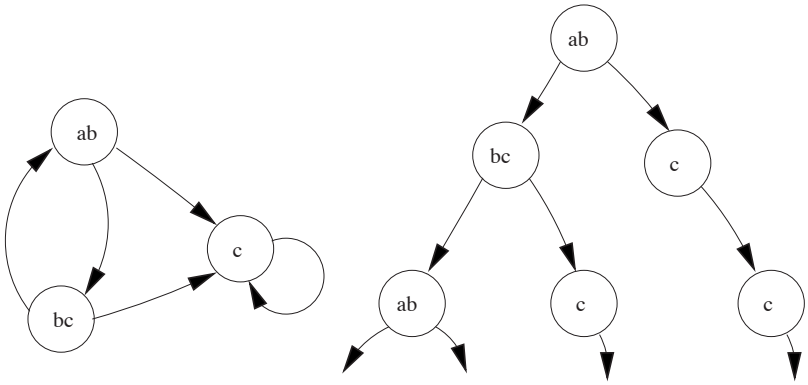


Abbildung 3.2: Abwicklung einer Kripke-Struktur

Definition 3.10 *Sei $M := (S, S_0, R, E_S)$ eine Kripke-Struktur und $s \in S$ ein Zustand. Dann wird die* Gültigkeit *einer CTL-Zustandsformel wie folgt induktiv definiert:*

1. $M, s \models p \quad \Leftrightarrow \quad p \in E_S(s)$.
2. $M, s \models \neg g \quad \Leftrightarrow \quad M, s \not\models g$.
3. $M, s \models g_1 \vee g_2 \quad \Leftrightarrow \quad M, s \models g_1$ *oder* $M, s \models g_2$.
4. $M, s \models g_1 \wedge g_2 \quad \Leftrightarrow \quad M, s \models g_1$ *und* $M, s \models g_2$.
5. $M, s \models Ef \quad \Leftrightarrow \quad$ *Es gibt einen in s beginnenden Pfad π mit* $M, \pi \models f$.
6. $M, s \models Af \quad \Leftrightarrow \quad$ *Für alle in s beginnenden Pfade π gilt* $M, \pi \models f$.

Dabei ist $p \in AP$ und g, g_1, g_2 sind CTL-Zustandsformeln. f ist entsprechend der Syntax eine CTL-Pfad-Formel. Die dort enthaltenen Quantoren X, F, G und U werden analog wie in LTL definiert.

Satz 3.11 *Es gelten die folgenden Äquivalenzen:*

- $AXg \equiv \neg EX(\neg g)$,
- $EFg \equiv E(\mathbf{true}\, U g)$,
- $AGg \equiv \neg EF(\neg g)$,
- $AFg \equiv \neg EG(\neg g)$,
- $A[g_1 U g_2] \equiv \neg E[\neg g_2 U(\neg g_1 \wedge \neg g_2)] \wedge \neg EG\neg g_2$

Beweis: Als Übung. □

3.3.3 Reduktion der Operatoren

In CTL müssen vor den Pfad-Quantoren X, F, G, U immer Zustands-Quantoren A oder E stehen. Es gibt damit acht Kombinationen:

- AXg und EXg,
- AFg und EFg,
- AGg und EGg,
- $A[g_1 U g_2]$ und $E[g_1 U g_2]$,

Aufgrund der Äquivalenzen aus Satz 3.11 können die 8 Kombination mittels EXg, EGg, $E[g_1 U g_2]$ ausgedrückt werden.

Satz 3.12 *Mit Hilfe der Operatoren EXg, EGg und $E[g_1 U g_2]$ können alle CTL-Formeln ausgedrückt werden.*

Ohne Beweis erwähnen wir folgende Eigenschaft, die die Trennschärfe der Temporallogik CTL charakterisiert.

Satz 3.13 *Bisimilare Modelle können von keiner CTL-Formel unterschieden werden.*

3.4 Ausdrucksmächtigkeit: LTL vs. CTL

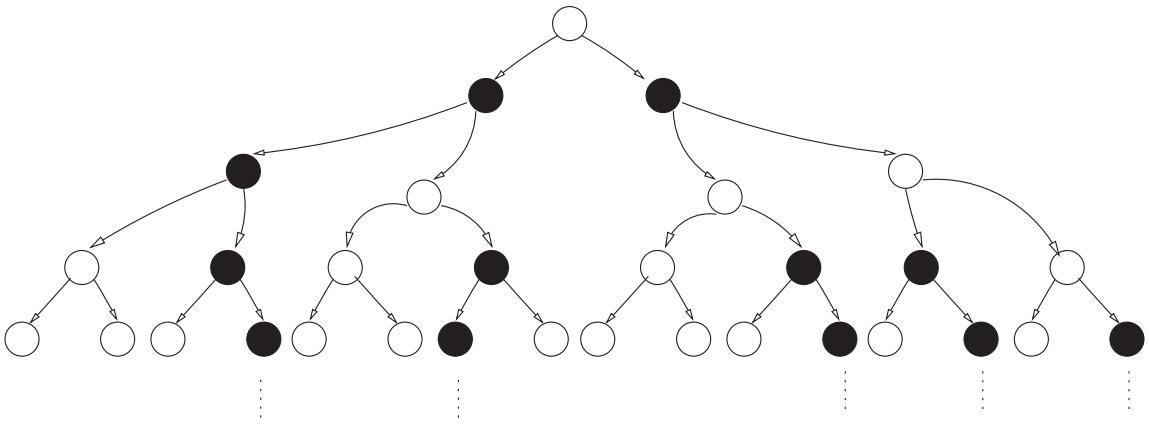
Zwei Formeln f und g heißen *äquivalent* (als Relationszeichen: $f \equiv g$ (siehe auch am Ende von Def. 3.5)), wenn sie durch kein Modell unterschieden werden können, d.h. wenn für jedes Modell M gilt:

$$M \models f \quad \text{gdw.} \quad M \models g$$

Auch in CTL kann man nicht alle LTL-Spezifikationen ausdrücken.

Satz 3.14 *Es gibt keine CTL-Formel, die äquivalent zur LTL-Formel $A(FGp)$ ist.*

Die Formel $A(FGp)$ bedeutet: „Auf jedem Pfad gibt es einen Zustand, ab dem p immer gilt“.



Satz 3.15 *Es gibt keine LTL-Formel, die äquivalent zur CTL-Formel $AG(EFp)$ ist.*

Die Formel $AG(EFp)$ bedeutet: „Von jedem Zustand ist ein Zustand erreichbar, in dem p gilt“.
(Ist das äquivalent zu folgender Aussage: „Alle Pfade enthalten unendlich viele Zustände, in den p gilt“?)
Beweis: Wir wollen zeigen, dass es zu der CTL-Formel $\phi = AGEFp$ keine äquivalente LTL-Formel ϕ' gibt.
(1) Wir zeigen zuerst: Seien M_1 und M_2 zwei Kripke-Strukturen und ψ eine beliebige LTL-Formel. Wenn $E_1(SS(M_1)) \subseteq E_2(SS(M_2))$ gilt, dann folgt aus $M_2 \models \psi$ auch $M_1 \models \psi$. Es gilt:

$$\begin{array}{ll} & M \models \psi \\ \iff & \forall \pi \in SS(M) : M, \pi \models \psi \\ \iff & \forall \pi \in SS(M) : E(\pi) \models \psi \\ \iff & E(SS(M)) \subseteq L^\omega(\psi) \end{array}$$

Sei $M_2 \models \psi$, d.h. $E_2(SS(M_2)) \subseteq L^\omega(\psi)$. Dann ist wegen der Inklusion auch $E_1(SS(M_1)) \subseteq E_2(SS(M_2))$ auch $E_1(SS(M_1)) \subseteq E_2(SS(M_2)) \subseteq L^\omega(\psi)$ und damit gilt dann $M_1 \models \psi$.

(2) Sei nun M die Kripke-Struktur mit $S = \{s_1, s_2\}$ und den Übergängen $s_1 \to s_1$, $s_1 \to s_2$ und $s_2 \to s_2$ sowie $S^0 = \{s_1\}$. Die Zustandsetiketten sind $E(s_1) = \emptyset$ und $E(s_2) = \{p\}$. Es gilt: $M \models \phi$ für $\phi = AGEFp$.
(3) Sei M' die Einschränkung der Kripke-Struktur M auf s_1, d.h. $S' = \{s_1\}$, $s_1 \to s_1$ und $E'(s_1) = \emptyset$. Es gilt: $M' \not\models \phi$ für $\phi = AGEFp$, denn M' besitzt nur noch den Pfad $\pi = s_1^\omega$ und es folgt $E'(\pi) = \emptyset^\omega$. Hier gilt p also nie.
Damit zeigen wir: Es gibt keine zu der CTL-Formel $\phi = AGEFp$ äquivalente LTL-Formel ϕ'.
Angenommen ϕ' wäre eine zu der CTL-Formel $\phi = AGEFp$ äquivalente LTL-Formel.
Aus der Äquivalenz folgt, dass ϕ' in M gilt, da ϕ – nach (2.) – in M gilt.
Da die Etiketten in M' und M identisch sind, ist jede ω-Folge $E(\pi)$ von M' auch eine von M. Also gilt nach (1.) auch ϕ' in M'.
Aus der Äquivalenz folgt nun, dass auch ϕ in M' – Widerspruch zu (3.).
Also gibt es keine solche äquivalente LTL-Formel ϕ'. □

CTL^* ist eine temporale Logik, die syntaktisch beide Logiken LTL und CTL umfasst. Es gibt aber auch Formeln in CTL*, z.B. $A(FGp) \vee AG(EFp)$, die weder in CTL noch in LTL ausdrückbar sind, wodurch CTL* die Logiken LTL und CTL auch semantisch echt enthält (vgl. Abb. 3.3)

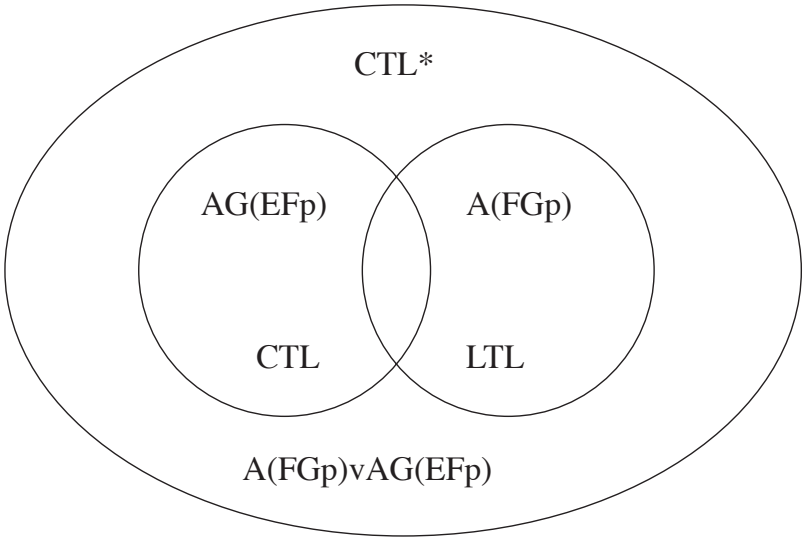
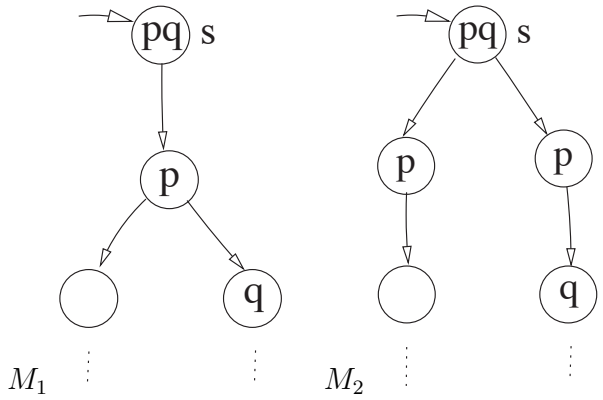


Abbildung 3.3: Beziehungen zwischen den Temporallogiken

Aufgabe 3.16 Gegeben sind die folgenden Kripke-Strukturen M_1 und M_2:



a) Gibt es Formeln in LTL, die die Strukturen unterscheiden, d.h. nur in einem Modell gelten?
b) Das gleiche für CTL.