Vorwort

Die Ausarbeitungen in diesem Buch entstanden während des Seminars

AVACS:
Automatic Verification and Analysis of Complex Systems

Das Seminar wurde im Sommersemester 2003 veranstaltet.

September 2003

Bernd Becker, Marc Herbstritt
Lehrstuhl für Rechnerarchitektur
Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Inhaltsverzeichnis

Grundlagen des Model Checking

Armin Staudenmaier

Institut für Informatik
Albert-Ludwigs-Universität Freiburg
staudenm@informatik.uni-freiburg.de

Einleitung

Bei der Verifikation von Systemen will man die Korrektheit eines Systems sicherstellen, indem man gezielte Anfragen an das System stellt. Im ersten Teil der Seminararbeit wird ein geeignetes formales Modell vorgestellt, mit dem man viele unterschiedliche Systeme einheitlich beschreiben kann. Ausserdem wird gezeigt, wie man einige der Systeme, die unterschiedliche Darstellungsformen haben, in ein einheitliches System überführen kann. Um Anfragen an das System zu stellen, wird im zweiten Teil die temporale Logik eingeführt. In dieser Logik stellen Formeln Sätze dar, mit denen man die gewünschten Anfragen an das System stellen kann. Im dritten Teil wird dann gezeigt, wie man diese Anfragen auf das System anwendet, um herauszufinden, ob das System die Anfrage erfüllt oder nicht.

1 Modellierung von Systemen

Wir wollen uns mit *Reaktiven Systemen* und ihrem zeitlichen Verhalten beschäftigen. Solche Systeme interagieren häufig mit ihrer Umwelt und müssen nicht terminieren. Sie können also nicht durch ihr Input-/Outputverhalten modelliert werden. Mit den folgenden Merkmalen kann man ein Reaktives System modellieren:

1. Ein *Zustand* beschreibt einen "Schnappschuss" des Systems, das heisst z.B. die Werte der Variablen zu einem bestimmten Zeitpunkt.
2. Ein *Übergang* beschreibt eine Zustandsänderung durch den Zustand vor und nach einer Aktion des Systems.
3. Eine *Berechnung* ist eine unendliche Sequenz von zusammenhängenden Übergängen.

Eine geeignete Form der Repräsentation dieser Merkmale ist die *Kripke Struktur*. *Simultane Systeme* oder *Concurrent Systems* können sehr unterschiedlich dargestellt sein. Synchrone und asynchrone Schaltkreise werden zum Beispiel durch ihr Schaltkreisdiagramm dargestellt, wohingegen Programme mit geteilten Variablen durch ihren Programmcode dargestellt werden. Als einheitlichen Formalismus, mit dem man jedes simultane System darstellen kann, verwenden

wir die Repräsentation in Logik Erster Ordnung. Daraus kann man dann die sogenannte Kripke Struktur extrahieren. Im Folgenden wird die Kripke Struktur formal definiert und es wird gezeigt, wie man sie aus Formeln Erster Ordnung extrahiert. Außerdem wird beschrieben, wie man verschiedene Programmiersprachen durch Formeln Erster Ordnung darstellen kann.

1.1 Modellierung von simultanen Systemen

Eine Kripke Struktur besteht aus einer Menge von Zuständen, einer Menge von Übergängen zwischen Zuständen und einer Funktion, die jeden Zustand mit einer Menge von Eigenschaften markiert, die in diesem Zustand wahr sind.

Definition 1. *Sei AE eine Menge von atomaren Einheiten. Eine* ***Kripke Struktur*** *K über AE ist ein Viertupel $K = (S, S_0, R, L)$ mit*

1. *S ist eine endliche Menge von Zuständen.*
2. *$S_0 \subseteq S$ ist die Menge von Startzuständen.*
3. *$R \subseteq S \times S$ ist eine totale Übergangsrelation, das heisst für jeden Zustand $s \in S$ gibt es einen Zustand $s' \in S$, so dass $R(s, s')$*
4. *$L : S \to 2^{AE}$ ist eine Funktion, die jeden Zustand mit der Menge der atomaren Einheiten markiert, die in diesem Zustand wahr sind.*

Definition 2. *Sei $K = (S, S_0, R, L)$ eine Kripke Struktur. Ein* ***Pfad*** *in der Struktur K von einem Zustand s ist eine unendliche Sequenz von Zuständen $\pi = s_0 s_1 s_2 s_3 \dots$ so dass $s_0 = s$ und $(s_i, s_{i+1}) \in R$ für alle $i \geq 0$ gilt.*

1.2 Repräsentation in Logik Erster Ordnung

$V = \{v_1, v_2, v_3, \dots\}$ sei die Menge der Systemvariablen, die Werte aus dem Universum D annehmen können.

Repräsentation von Zustandsmengen: Ein Zustand ist eine spezifische Belegung aller Systemvariablen $s : V \to D$. Zum Beispiel $V = \{v_1, v_2, v_3\}$ mit der Belegung $s_1 = \langle v_1 \leftarrow 3, v_2 \leftarrow 2, v_3 \leftarrow 1\rangle$. Die Formel $\mathcal{F}$ repräsentiert diese Belegung.

$$\begin{aligned}\mathcal{F} &= (v_1 \leftarrow 3) \wedge (v_2 \leftarrow 2) \wedge (v_3 \leftarrow 1) \\ &= (v_1 \vee \neg 3) \wedge (v_2 \vee \neg 2) \wedge (v_3 \vee \neg 1)\end{aligned}$$

Sie ist für die atomare Einheit $v_1 = 3, v_2 = 2$ und $v_3 = 1$ erfüllt. Also wäre eine äquivalente Formulierung $\mathcal{F}' = (v_1 = 3) \wedge (v_2 = 2) \wedge (v_3 = 1)$. Wenn wir vereinbaren, dass eine Formel die Menge aller Belegungen bzw. atomarer Einheiten repräsentiert, die sie wahr machen, dann können wir Zustandsmengen durch Formeln erster Ordnung beschreiben. Sei $S = \{s_1, s_2\} = \{\langle v_1 \leftarrow 3, v_2 \leftarrow 2, v_3 \leftarrow 1\rangle, \langle v_1 \leftarrow 1, v_2 \leftarrow 2, v_3 \leftarrow 3\rangle\}$, dann kann man diese Zustandsmenge durch folgende Formel darstellen:

$$\begin{aligned}\mathcal{F} &= ((v_1 \leftarrow 3) \wedge (v_2 \leftarrow 2) \wedge (v_3 \leftarrow 1)) \vee ((v_1 \leftarrow 1) \wedge (v_2 \leftarrow 2) \wedge (v_3 \leftarrow 3)) \\ &\equiv \{\langle v_1 \leftarrow 3, v_2 \leftarrow 2, v_3 \leftarrow 1\rangle, \langle v_1 \leftarrow 1, v_2 \leftarrow 2, v_3 \leftarrow 3\rangle\} = S\end{aligned}$$

Repräsentation von Übergangsmengen: Eine Übergangsmenge ist eine Menge von geordneten Zustandspaaren. Zum Modellieren der neuen Zustände sei V' die Menge der Systemvariablen im nächsten Zustand. Jede Variable $v \in V$ hat eine zugehörige Variable $v' \in V'$. Eine Übergangsmenge kann man nun wie oben durch eine Formel angeben. Zum Beispiel sei $V = \{v_1, v_2\}$ und $V' = \{v_1', v_2'\}$ mit den Anfangszuständen $S_0 = \{s_1, s_2\} = \{\langle v_1 \leftarrow 3, v_2 \leftarrow 2\rangle, \langle v_1 \leftarrow 1, v_2 \leftarrow 2\rangle\}$. Eine Formel für eine Übergangsmenge, bei der die Variablen von zwei Zuständen um 1 erhöht werden:

$$\begin{aligned}\mathcal{R}(V, V') =& ((v_1 \leftarrow 3) \wedge (v_2 \leftarrow 2)) \wedge ((v_1' \leftarrow 4) \wedge (v_2' \leftarrow 3)) \\ &\vee ((v_1 \leftarrow 1) \wedge (v_2 \leftarrow 2)) \wedge (v_1' \leftarrow 2) \wedge (v_2' \leftarrow 3)) \\ \equiv& \{(\langle v_1 \leftarrow 3, v_2 \leftarrow 2\rangle, \langle v_1' \leftarrow 4, v_2' \leftarrow 3\rangle), \\ &(\langle v_1 \leftarrow 1, v_2 \leftarrow 2\rangle, \langle v_1' \leftarrow 2, v_2' \leftarrow 3\rangle)\}\end{aligned}$$

Erzeugen einer Kripke Struktur aus einer Formel Sei $K = (S, S_0, R, L)$ eine Kripke Struktur und seien $\mathcal{S}_0, \mathcal{R}$ Formeln, die ein simultanes System beschreiben. Die Kripke Struktur kann man dann aus den Formeln ableiten:

- Die *Zustandsmenge* S ist die Menge aller möglichen Belegungen der Systemvariablen V.
- Die *Menge der Anfangszustände* S_0 ist die Menge aller Belegungen der Systemvariablen V, für die $\mathcal{S}_0(V) = True$ gilt.
- Seien s und s' zwei Zustände. Der *Übergang* $R(s, s')$ gilt, wenn für jedes $v \in V, v' \in V'$ die Werte $s(v)$ und $s'(v')$ eingesetzt in die Formel $\mathcal{R}(V, V')$ das Ergebnis $True$ liefern.
- Für jeden Zustand s ist die Menge der Markierungsfunktion $L(s)$ die Teilmenge aller Belegungen bzw. atomaren Einheiten, für die der Zustand gilt.

Da die Übergangsrelation einer Kripke Struktur total sein muss, wird die Relation R so erweitert, dass sie, wenn ein Zustand keinen Nachfolger hat, durch $R(s, s)$ gilt.

Beispiel: Gegeben sei ein einfaches System mit zwei Variablen x und y, die Werte aus $D = \{0, 1\}$ annehmen können. Eine Belegung ist dann ein Paar $(d_1, d_2) \in D \times D$ wobei d_1 ein Wert für x und d_2 ein Wert für y ist. Das System besteht aus einem Übergang

$$x := (x + y) \bmod 2,$$

der von dem Zustand, bei dem $x = 1$ und $y = 1$ ist, beginnt. Die Formel für die Menge der Anfangszustände ist also

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

und die Formel für die Menge der Übergänge wird durch

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

beschrieben. Daraus erzeugt man die Kripke Struktur $K = (S, S_0, R, L)$:

- $S = D \times D = \{(0,0), (0,1), (1,0), (1,1)\}$
- $S_0 = \{(1,1)\}$
- $R = \{((1,1),(0,1)), ((0,1),1,1)), ((1,0),(1,0)), ((0,0),(0,0))\}$
- $L((1,1)) = \{x = 1, y = 1\}, L((0,1)) = \{x = 0, y = 1\}, L((1,0)) = \{x = 1, y = 0\}, L((0,0)) = \{x = 0, y = 0\}$

Der einzige Pfad, der an einem Startzustand beginnt, ist $(1,1)(0,1)(1,1)(0,1)$. Dieser Pfad ist die einzige Berechnung des Systems. Abbildung 1 zeigt die Kripke Struktur. Die Knoten repräsentieren die Zustände, die Kanten die Übergangsrelationen.

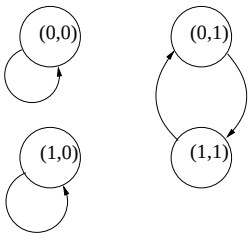


Abb. 1. Kripke Struktur mit vier Zuständen

1.3 Digitale Schaltkreise

Digitale Schaltkreise sind simultane Systeme, die aus verschiedenen Komponenten bestehen und zusammen ein System bilden. Die Komponenten sind miteinander verbunden und "kommunizieren" durch Signale. Wie diese Schaltkreise durch Formeln dargestellt werden können soll im folgenden gezeigt werden. Wir unterscheiden zwischen *synchronen* und *asynchronen Schaltkreisen*. Es sei V die Menge der Elemente eines Schaltkreises, die Zustände halten können, die zur Einfachheit durch Boolsche Variablen mit den Werten 0 und 1 dargestellt werden. Ein Zustand ist dann eine Belegung aller boolscher Variablen V. Daraus kann man eine boolsche Formel ableiten, die einen Zustand beschreibt. Also können wir die Übergänge und auch das ganze System durch Boolsche Formeln modellieren. Aus diesen Boolschen Formeln kann dann eine äquivalente Kripke Struktur erzeugt werden.

Synchrone Schaltkreise: V besteht hier aus allen Ausgängen der Register zusammen mit den Eingängen. Im allgemeinen Fall eines Schaltkreises mit n Zustandsvariablen sei $V = \{v_0, \ldots, v_{n-1}\}$ und $V' = \{v'_0, \ldots, v'_{n-1}\}$. Für jede Zustandsvariable v'_i existiert eine boolsche Funktion f_i, so dass

$$v'_i = f_i(V).$$

Deshalb beschreibt man die Relationen mit der Formel:

$$\mathcal{R}_i(V, V') \equiv (v'_i \Leftrightarrow f_i(V)).$$

Bei synchronen Schaltkreisen wird nach Anlegen eines Signals gewartet bis die Werte stabil sind, dann ändern sich die Zustandselemente gleichzeitig. Deshalb kann die Übergangsrelation durch die *Konjunktion* der einzelnen Prozesse repräsentiert werden.

$$\mathcal{R}(V, V') \equiv \mathcal{R}_0(V, V') \wedge \cdots \wedge \mathcal{R}_{n-1}(V, V').$$

Beispiel: Ein modulo 8 Zähler (siehe Abb. 2).
Es sei $V = \{v_0, v_1, v_2\}, V' = \{v'_0, v'_1, v'_2, \}$. Die Übergänge sind gegeben durch

$$\begin{aligned} v'_0 &= \neg v_0 \\ v'_1 &= v_0 \oplus v_1 \\ v'_2 &= (v_0 \wedge v_1) \oplus v_2 \end{aligned}$$

Daraus ergeben sich die Relationen:

$$\begin{aligned} \mathcal{R}_0(V, V') &\equiv (v'_0 \Leftrightarrow \neg v_0) \\ \mathcal{R}_1(V, V') &\equiv (v'_1 \Leftrightarrow v_0 \oplus v_1) \\ \mathcal{R}_2(V, V') &\equiv (v'_2 \Leftrightarrow (v_0 \wedge v_1) \oplus v_2) \end{aligned}$$

und schliesslich:

$$\mathcal{R}(V, V') \equiv \mathcal{R}_0(V, V') \wedge \mathcal{R}_1(V, V') \wedge \mathcal{R}_2(V, V')$$

Asynchrone Schaltkreise Bei asynchronen Schaltkreisen kann sich der Wert einer Komponente so schnell ändern, dass es unwahrscheinlich ist, dass sich zwei Komponenten zur selben Zeit ändern. Deshalb kann die Übergangsrelation durch folgende Formeln repräs entiert werden:

$$\mathcal{R}(V, V') \equiv \mathcal{R}_0(V, V') \vee \cdots \vee \mathcal{R}_{n-1}(V, V')$$

mit

$$\mathcal{R}_i(V, V') \equiv (v'_i \Leftrightarrow f_i(V)) \wedge \bigwedge_{j \neq i} (v'_j \Leftrightarrow v_j)$$

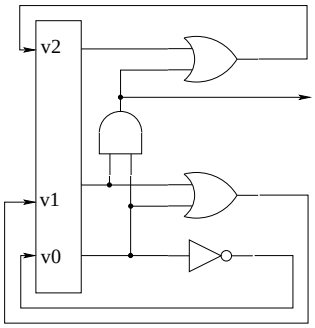


Abb. 2. Synchroner modulo acht Zähler

1.4 Programme

Sequentielle Programme Programme bestehen aus Anweisungen, die sequentiell ausgeführt werden. Wir zeigen eine Übersetzungsprozedur $\mathcal{C}$, die aus dem Text eines sequentiellen Programms P eine Formel Erster Ordnung $\mathcal{R}$ macht, welche alle Übergänge des Programms beschreibt. Wir wollen o.B.d.A. annehmen, dass jede Anweisung einen eindeutigen Einstiegspunkt und Ausstiegspunkt im Programm besitzt. Durch eine Markierungstransformation wird jeder Ein- und Ausstiegspunkt eindeutig markiert und aus dem Programm P wird ein markiertes Programm $P^{\mathcal{L}}$. Da bei sequentiellen Programmen die Ein- und Ausstiegspunkte gleich sind, genügt es, nur die Einstiegspunkte zu markieren. Für eine Anweisung P ist die markierte Anweisung $P^{\mathcal{L}}$ wie folgt definiert:

- Wenn P keine zusammengesetzte Anweisung ist (z.B. x = e, skip, lock, ...) dann ist $P^{\mathcal{L}} = P$.
- Wenn $P = P_1; P_2$, dann ist $P^{\mathcal{L}} = P_1^{\mathcal{L}}; l'' : P_2^{\mathcal{L}}$
- Wenn $P =$ **if** b **then** P_1 **else** P_2 **end if**, dann ist $P^{\mathcal{L}} =$ **if** b **then** $l_1 : P_1^{\mathcal{L}}$ **else** $l_2 : P_2^{\mathcal{L}}$ **end if**
- Wenn $P =$ **while** b **do** P_1 **end while**, dann ist $P^{\mathcal{L}} =$ **while** b **do** $l_1 : P_1^{\mathcal{L}}$ **end while**

pc sei eine spezielle *Programmzähler* -Variable, deren Wertebereich die Menge der Programmarkierungen und einen zusätzlichen Wert $\perp$, den "undefinierten Wert", beträgt. Dieser Wert wird für simultane Systeme benötigt und bedeutet, dass das Programm nicht aktiv ist. Die Formel für die Menge der Startzustände des Programms P ist

$$\mathcal{S}_0(V, pc) \equiv pre(V) \wedge pc = m,$$

wobei durch $pre(V)$ die Bedingungen für die Anfangswerte der Variablen von P festgelegt werden und m der Einstiegspunkt des Programms P ist. Die Übersetzungsprozedur $\mathcal{C}$ hängt von den Parametern l, dem Einstiegspunkt, dem Programm P und dem Ausstiegspunkt l' ab. $\mathcal{C}(l, P, l')$ wird rekursiv definiert und beschreibt die Menge der Übergänge als eine Disjunktion. Für jede Anweisung gibt es eine Übersetzungsvorschrift:

– Zuweisung:

$$\mathcal{C}(l, v \leftarrow e, l') \equiv pc = l \wedge pc' = l' \wedge v' = e \wedge \bigwedge_{x \in V \setminus \{v\}} (x' = x)$$

– Skip:

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

– Sequentielle Zusammensetzung:

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

– **if - then - else - end if** :

$$\begin{aligned}
&\mathcal{C}(l, \textbf{ if } b \textbf{ then } l_1 : P_1 \textbf{ else } l_2 : P_2 \textbf{ end if}, l') \equiv \\
&(pc = l \wedge pc' = l_1 \wedge b \wedge \bigwedge_{x \in V} (x' = x)) \vee \\
&(pc = l \wedge pc' = l_2 \wedge \neg b \wedge \bigwedge_{x \in V} (x' = x)) \vee \\
&\mathcal{C}(l_1, P_1, l') \vee \mathcal{C}(l_2, P_2, l')
\end{aligned}$$

– While:

$$\begin{aligned}
&\mathcal{C}(l, \textbf{while } b \textbf{ do } l_1 : P_1 \textbf{ end while}, l') \equiv \\
&(pc = l \wedge pc' = l_1 \wedge b \wedge \bigwedge_{x \in V} (x' = x)) \vee \\
&(pc = l \wedge pc' = l' \wedge \neg b \wedge \bigwedge_{x \in V} (x' = x)) \vee \\
&\mathcal{C}(l_1, P_1, l)
\end{aligned}$$

Die Markierungsprozedur und die Übersetzungsvorschriften können natürlich beliebig erweitert und für jede Programmiersprache angepasst werden.

Beispiel: Das kleine Programm

$$x = -1; \textbf{ if } (x < 0) \textbf{ then } x = -x; \textbf{ else } x = x; \textbf{ end if}$$

soll übersetzt werden. Zuerst wird es durch die Markierungsprozedur markiert. $P = P_1; P2$:

$$[x = -1]; l'' : [\textbf{ if } (x < 0) \textbf{ then } x = -x; \textbf{ else } x = x; \textbf{ end if }]$$

if b **then** P_1 **else** P_2 **end if**:

$$x \leftarrow -1; l'' : \textbf{ if } (x < 0) \textbf{ then } l_1 : [x = -x]; \textbf{ else } l_2 : [x = x]; \textbf{ end if}$$

Also ist das markierte Programm $P^{\mathcal{L}}$:

$$x \leftarrow -1; l'' : \textbf{ if } (x < 0) \textbf{ then } l_1 : x \leftarrow -x; \textbf{ else } l_2 : x \leftarrow x; \textbf{ end if}$$

Die Übersetzungsprozedur überführt das markierte Programm $P^{\mathcal{L}}$ in eine Formel, die die Übergänge des Programms darstellt. Hier sind die Umformungen Schritt für Schritt aufgelistet:

1. $\mathcal{C}(l, x = -1; l'' : \textbf{ if } (x < 0) \textbf{ then } l_1 : x = -x; \textbf{ else } l_2 : x = x; \textbf{ end if }, l')$
2. $\mathcal{C}(l, x \leftarrow -1, l'') \vee \mathcal{C}(l'', \textbf{ if } (x < 0) \textbf{ then } l_1 : x \leftarrow -x; \textbf{ else } l_2 : x \leftarrow x; \textbf{ end if }, l')$
3. $(pc = l \wedge pc' = l'' \wedge x' = -1) \vee (pc = l'' \wedge pc' = l_1 \wedge (x < 0) \wedge x' = x) \vee$
$(pc = l'' \wedge pc' = l_2 \wedge \neg(x < 0) \wedge x' = x) \vee \mathcal{C}(l_1, x \leftarrow -x, l') \vee \mathcal{C}(l_2, x \leftarrow x, l')$
4. $(pc = l \wedge pc' = l'' \wedge x' = -x) \vee (pc = l'' \wedge pc' = l_1 \wedge (x < 0)) \vee$
$(pc = l'' \wedge pc' = l_2 \wedge \neg(x < 0)) \vee (pc = l_1 \wedge pc' = l' \wedge x' = -x) \vee$
$(pc = l_2 \wedge pc' = l' \wedge x' = x)$

Die Formel ist also:

$$\begin{aligned}\mathcal{R} \equiv &(pc = l \wedge pc' = l'' \wedge x' = -1) \vee (pc = l'' \wedge pc' = l_1 \wedge (x < 0)) \vee \\ &(pc = l'' \wedge pc' = l_2 \wedge \neg(x < 0)) \vee (pc = l_1 \wedge pc' = l' \wedge x' = -x) \vee \\ &(pc = l_2 \wedge pc' = l' \wedge x' = x)\end{aligned}$$

Zur Einfachheit soll der Wertebereich der Variablen x die Menge $\{-1, 1\}$ betragen. Da die Variableninitialisierung Teil des Programmes ist, brauchen wir keine Formel $\mathcal{S}_0$ für die Startzustände angeben. Alle möglichen Variablenbelegungen sollen in Frage kommen. Aus der Formel kann man nun die Kripke Struktur ablesen. Die Zustände ergeben sich aus allen möglichen Variablenbelegungen und den Werten des Programmzählers. Zwei Übergänge kann man zum Beispiel aus folgender Konjunktion ablesen:

$$\begin{gathered}pc = l \wedge pc' = l'' \wedge x' = -1 \\ \equiv \\ \{(\langle x \leftarrow 1, pc \leftarrow l\rangle, \langle x \leftarrow -1, pc \leftarrow l''\rangle), (\langle x \leftarrow -1, pc \leftarrow l\rangle, \langle x \leftarrow -1, pc \leftarrow l''\rangle)\}\end{gathered}$$

Abbildung 3 zeigt die resultierende Kripke Struktur.

Simultane Programme Ein simultanes Programm oder ein "concurrent program" besteht aus einer Menge von Prozessen, die parallel ausgeführt werden können. V_i sei die Menge der Variablen, die durch den Prozess P_i verändert werden können. Die Mengen müssen nicht disjunkt sein. V sei die Menge aller Programmvariablen. Der Programmzähler eines Prozesses P_i sei pc_i. PC sei die Menge aller Programmzähler. Ein nebenläufiges Programm hat die Form

$$\textbf{cobegin } P_1 \parallel P_2 \parallel \ldots \parallel P_n \textbf{ coend}$$

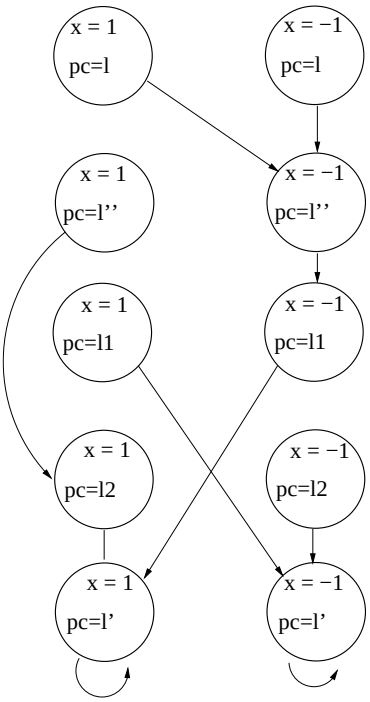


Abb. 3. Kripke Struktur des Beispielprogramms

Die Markierungstransformation von sequentiellen Programmen wird so erweitert, dass ein simultanes Programm als Anweisung in einem sequentiellen Programm auftauchen kann. Die Transformation bringt eine Markierung an jedem Eintrittspunkt und Austrittspunkt jedes Prozesses an. Im Gegensatz zu den sequentiellen Programmen sind jetzt die Eingangs- und Ausgangspunkte von den nebenläufigen Prozessen verschieden. Deshalb müssen die Ausgangs punkte explizit markiert werden. Für die Markierungstransformation gilt:

- Wenn $P = \textbf{cobegin}\ P_1 \parallel P_2 \parallel \ldots \parallel P_n\ \textbf{coend}$, dann ist
  $P^{\mathcal{L}} = \textbf{cobegin}\ l_1 : P_1^{\mathcal{L}} l_1' \parallel l_2 : P_2^{\mathcal{L}} l_2' \parallel \ldots \parallel l_n : P_n^{\mathcal{L}} l_n'\ \textbf{coend}$

Die Formel, die die Startzustände beschreibt, ist

$$\mathcal{S}_0 \equiv pre(V) \wedge pc = m \wedge \bigwedge_{i=1}^{n} (pc_i = \perp).$$

Dabei soll $pc_i = \perp$ heissen, daß Prozess P_i noch nicht aktiviert wurde und deshalb von diesem Zustand auch nicht ausgeführt werden kann. Die Übersetzungsprozedur $\mathcal{C}$ wird für simultane Programme erweitert:
$\mathcal{C}(, l, \textbf{cobegin}\ l_1 : P_1^{\mathcal{L}} l_1' \parallel l_2 : P_2^{\mathcal{L}} l_2' \parallel \ldots \parallel l_n : P_n^{\mathcal{L}} l_n'\ \textbf{coend}, l')$ ist die Disjunkti-

on von folgenden drei Formeln:

$$pc = l \wedge pc'_1 = l_1 \wedge \cdots \wedge pc'_n = l_n \wedge pc' = \bot$$

$$pc = \bot \wedge pc_1 = l'_1 \wedge \cdots \wedge pc_n = l'_n \wedge pc' = l' \wedge \bigwedge_{i=1}^{n} (pc'_i = \bot)$$

$$\bigvee_{i=1}^{n} (\mathcal{C}(l_i, P_i, l'_i) \wedge same(V \setminus V_i) \wedge same(PC \setminus \{pc_i\}))$$

Shared Variables Simultane Programme, in denen die Menge V_i sich überschneiden, heissen Shared Variable Programs. Die Translationsprozedur $\mathcal{C}$ muss für solche Programme angepaßt werden:

- $\mathcal{C}(l, \mathbf{wait}(b), l')$ ist die Disjunktion der folgenden zwei Formeln:

$$(pc_i = l \wedge pc'_i = l \wedge \neg b \wedge same(V_i))$$
$$(pc_i = l \wedge pc'_i = l' \wedge b \wedge same(V_i))$$

- $\mathcal{C}(l, \mathbf{lock}(v), l')$ ist die Disjunktion der folgenden zwei Formeln:

$$(pc_i = l \wedge pc'_i = l \wedge v = 1 \wedge same(V_i))$$
$$(pc_i = l \wedge pc'_i = l' \wedge v = 0 \wedge v' = 1 \wedge same(V_i \setminus \{v\}))$$

- $\mathcal{C}(l, \mathbf{unlock}(v), l')$:

$$(pc_i = l \wedge pc'_i = l' \wedge v' = 0 \wedge same(V_i \setminus \{v\}))$$

Ein ausführliches Beispiel steht in [1], S. 24,ff.

2 Temporale Logik

Mit temporaler Logik kann man Eigenschaften von Kripke Strukturen beschreiben, insbesondere Eigenschaften von Übergängen zwischen Zuständen. Damit lassen sich Berechnungssequenzen beschreiben, mit denen man auch die Verifikation von nichtterminierenden reaktiven Systemen durchführen kann. Zeitliche Aussagen, zum Beispiel dass ein bestimmter Zustand auf einem Pfad "in der Zukunft" erreicht wird, werden durch *temporale Operatoren* spezifiziert.

2.1 Die Logik CTL* (Computation Tree Logic)

CTL* Formeln beschreiben Eigenschaften von Berechnungsbäumen. Einen Berechnungsbaum erhält man, indem man bei einer Kripke Struktur einen Anfangszustand "festhält" und die Kripke Struktur zu einem unendlichen Baum mit dem Anfangszustand als Wurzel nach unten "aufbiegt".

In einem Berechnungsbaum kann man alle möglichen Berechnungen, die vom Startzustand ausgehen, in Form von Pfaden erkennen. Die Verzweigungsstruktur wird durch ***Pfadquantoren*** beschrieben:

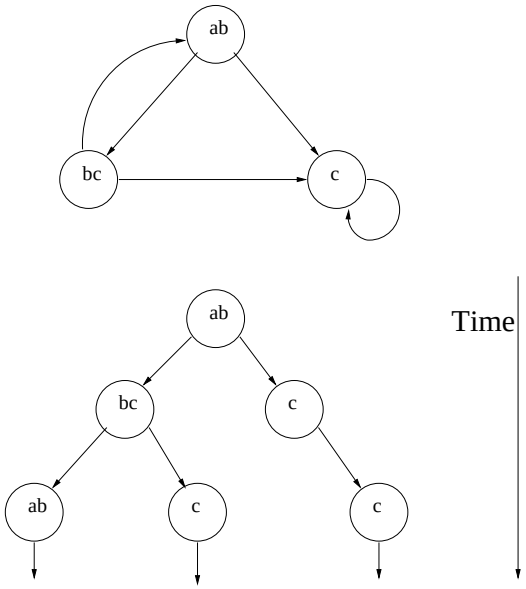


Abb. 4. Kripke Modell und Berechnungsbaum

- **A** "für alle Berechnungspfade"
- **E** "für manche Berechnungspfade"

Diese Quantoren werden auf einen speziellen Zustand angewandt, um Eigenschaften von allen (**A**) oder manchen (**E**) Pfaden, die von diesem Zustand ausgehen, zu spezifizieren. ***Temporale Operatoren*** beschreiben die Eigenschaften eines Pfades durch einen Baum. Es gibt fünf Basisoperatoren:

- **X** ("next time"): Die Eigenschaft gilt im nächsten Zustand des Pfades.
- **F** ("in the future"): Die Eigenschaft gilt in irgendeinem Zustand auf dem Pfad.
- **G** ("globally"): Die Eigenschaft gilt in allen Zuständen auf dem Pfad.
- f **U** g ("until"): Es gibt einen Zustand auf dem Pfad bei dem g gilt, auf allen vorhergehenden Zuständen auf dem Pfad gilt f.
- f **R** g ("release"): g gilt entlang eines Pfades, bis zu einem Zustand bei dem f gilt.

In CTL* gibt es zwei Arten von Formeln:

- ***Zustandsformeln*** sind wahr in einem bestimmten Zustand
- ***Pfadformeln*** sind wahr auf einem bestimmten Pfad

Definition 3. *Sei AE die Menge der atomaren Einheiten. Die* ***Syntax der Zustandsformeln von CTL**** *wird durch folgende Regeln festgelegt:*

1. *Wenn $p \in AE$, dann ist p eine* ***Zustandsformel****.*

2. *Wenn f und g* ***Zustandsformeln*** *sind, dann sind auch $\neg f, f \vee g$ und $f \wedge g$* ***Zustandsformeln****.*
3. *Wenn f eine* ***Pfadformel*** *ist, dann sind* $\boldsymbol{E}\ f$ *und* $\boldsymbol{A}\ f$ ***Zustandsformeln****.*

Die ***Syntax der Pfadformeln von CTL**** *wird durch zwei zusätzliche Regeln festgelegt:*

1. *Wenn f eine* ***Zustandsformel*** *ist, dann ist f auch eine* ***Pfadformel****.*
2. *Wenn f und g* ***Pfadformeln*** *sind, dann sind auch $\neg f, f \vee g, f \wedge g$,* $\boldsymbol{X}\ f, \boldsymbol{F}\ f, \boldsymbol{G}\ f, f\ \boldsymbol{U}\ g$, *und $f\ \boldsymbol{R}\ g$* ***Pfadformeln****.*

CTL* *ist die Menge der* ***Zustandsformeln****, die durch diese Regeln gebildet werden können.*

Die Semantik von CTL* soll in Bezug auf eine Kripke Struktur erklärt werden. Ein Zustand s_i auf einem Pfad π, wird durch die Notation π^i beschrieben.

- Wenn f eine **Zustandsformel** ist, dann bedeutet $K, s \models f$, dass die Formel f bei dem Zustand s in der Kripke Struktur K erfüllt ist.

- Wenn f eine **Pfadformel** ist, dann bedeutet $K, \pi \models f$, dass f auf dem Pfad π der Kripke Struktur K erfüllt ist.

Definition 4. *Es sei $K = (S, R, L)$ eine Kripke Struktur. Es seien f_1 und f_2 Zustandsformeln und g_1 und g_2 Pfadformeln. Die Relation $\models$ ist induktiv definiert:*

1.	$K, s \models p$	$\Leftrightarrow$	$p \in L(s)$
2.	$K, s \models \neg f_1$	$\Leftrightarrow$	$K, s \not\models f_1$
3.	$K, s \models f_1 \vee f_2$	$\Leftrightarrow$	$K, s \models f_1$ *oder* $K, s \models f_2$
4.	$K, s \models f_1 \wedge f_2$	$\Leftrightarrow$	$K, s \models f_1$ *und* $K, s \models f_2$
5.	$K, s \models \boldsymbol{E}\ g_1$	$\Leftrightarrow$	*Es gibt einen Pfad π, der bei s anfängt, sodass* $K, \pi \models g_1$
6.	$K, s \models \boldsymbol{A}\ g_1$	$\Leftrightarrow$	*Für jeden Pfad π, der bei s anfängt, gilt* $K, \pi \models g_1$
7.	$K, \pi \models f_1$	$\Leftrightarrow$	*s ist der erste Zustand von π und* $K, s \models f_1$
8.	$K, \pi \models \neg g_1$	$\Leftrightarrow$	$K, \pi \not\models g_1$
9.	$K, \pi \models g_1 \vee g_2$	$\Leftrightarrow$	$K, \pi \models g_1$ *oder* $K, \pi \models g_2$
10.	$K, \pi \models g_1 \wedge g_2$	$\Leftrightarrow$	$K, \pi \models g_1$ *und* $K, \pi \models g_2$
11.	$K, \pi \models \boldsymbol{X}\ g_1$	$\Leftrightarrow$	$K, \pi^1 \models g_1$
12.	$K, \pi \models \boldsymbol{F}\ g_1$	$\Leftrightarrow$	*Es gibt ein $k \geq 0$, sodass* $K, \pi^k \models g_1$
13.	$K, \pi \models \boldsymbol{G}\ g_1$	$\Leftrightarrow$	$\forall i \geq 0, K, \pi^i \models g_1$
14.	$K, \pi \models g_1\ U\ g_2$	$\Leftrightarrow$	*Es gibt ein $k \geq 0$, sodass* $K, \pi^k \models g_2$ *und für jedes* $0 \leq j < k, K, \pi^j \models g_1$
15.	$K, \pi \models g_1\ R\ g_2$	$\Leftrightarrow$	*Für jedes $j \geq 0$ gilt: Wenn für jedes $i < j$,* $K, \pi^i \not\models g_1$, *dann gilt* $K, \pi^j \models g_2$.

Beispiel: Die Operatoren $\vee, \neg, \mathbf{X}, \mathbf{U}$ und $\mathbf{E}$ genügen, um jede andere CTL*-Formel auszudrücken.

1. $f \wedge g \equiv \neg(\neg f \vee \neg g)$
2. $f\ \mathbf{R}\ g \equiv \neg(\neg f\ \mathbf{U}\ \neg g)$
3. $\mathbf{F}\ f \equiv True\ \mathbf{U}\ f$
4. $\mathbf{G}\ f \equiv \neg\mathbf{F}\ \neg f$
5. $\mathbf{A}\ (f) \equiv \neg\mathbf{E}\ (\neg f)$

Beweis zu 2.:

$$
\begin{aligned}
& K, \pi \models f\ \mathbf{R}\ g \Leftrightarrow \\
& \forall j \geq 0 \text{ gilt: Wenn für jedes } i < j:\ K, \pi^i \not\models f, \text{ dann } K, \pi^j \models g \Leftrightarrow \\
& \forall j \geq 0 : ((\forall i < j :\ K, \pi^i \not\models f) \rightarrow (K, \pi^j \models g)) \Leftrightarrow \\
& \forall j \geq 0 : (\neg(\forall i < j :\ K, \pi^i \not\models f) \vee (K, \pi^j \models g)) \Leftrightarrow \\
\neg(\exists j \geq 0 : \neg(\neg(\forall i < j :\ & K, \pi^i \not\models f) \vee (K, \pi^j \models g))) \Leftrightarrow \\
\neg(\exists j \geq 0 : ((\forall i < j :\ & K, \pi^i \not\models f) \wedge \neg(K, \pi^j \models g))) \Leftrightarrow \\
\neg(\exists j \geq 0 : ((\forall i < j :\ & K, \pi^i \models \neg f) \wedge (K, \pi^j \models \neg g))) \Leftrightarrow \\
\neg(\exists j \geq 0 : ((K, \pi^j \models \neg g) & \wedge (\forall i < j :\ K, \pi^i \models \neg f))) \Leftrightarrow \\
\neg(\text{Es gibt ein } j \geq 0, & \text{ sodass } K, \pi^j \models \neg g \\
& \text{und für jedes } 0 \leq i < j, K, \pi^i \models \neg f) \Leftrightarrow \\
\neg(K, \pi \models \neg f\ & \mathbf{U}\ \neg g)
\end{aligned}
$$

□

Beweis zu 3.:

$$
\begin{aligned}
& K, \pi \models True\ \mathbf{U}\ f \Leftrightarrow \\
& \exists k \geq 0 :\ K, \pi^k \models f\ \wedge\ \forall 0 \leq j < k,\ K, \pi^j \models True \Leftrightarrow \\
& \exists k \geq 0 :\ K, \pi^k \models f\ \wedge\ True \Leftrightarrow \\
& \exists k \geq 0 :\ K, \pi^k \models f \Leftrightarrow \\
& K, \pi \models\ \mathbf{F}\ f
\end{aligned}
$$

□

Beweis zu 4.:

$$
\begin{aligned}
K, \pi \models \mathbf{G}\ f \Leftrightarrow \forall i \geq 0, K, \pi^i \models f \Leftrightarrow \\
\neg\exists i \geq 0, \pi^i \models \neg f \Leftrightarrow \neg(K, \pi \models \mathbf{F}\ \neg f)
\end{aligned}
$$

□

Beweis zu 5.:

$K, s \models \mathbf{A}\ f \Leftrightarrow$ Für jeden Pfad π, der bei s anfängt, gilt $K, \pi \models f \Leftrightarrow$
$\neg$(Es gibt einen Pfad π, der bei s anfängt, für den gilt
$K, \pi \models \neg f) \Leftrightarrow \neg K, s \models \mathbf{E}\ \neg f$

□

2.2 CTL und LTL

CTL und LTL sind Teillogiken der CTL*-Logik. Sie unterscheiden sich in der Art von Verzweigungen, die durch ihre Formeln im Berechnungsbaum dargestellt werden können. CTL ist die Untermenge von CTL*, die durch eine Einschränkung der Syntax der Pfadformeln von CTL* entsteht.

Definition 5. *Sei AE die Menge der atomaren Einheiten. Die* ***Syntax der Zustandsformeln von CTL*** *wird durch folgende Regeln festgelegt:*

1. *Wenn $p \in AE$, dann ist p eine* ***Zustandsformel****.*
2. *Wenn f und g* ***Zustandsformeln*** *sind, dann sind auch $\neg f, f \vee g$ und $f \wedge g$* ***Zustandsformeln****.*
3. *Wenn f eine* ***Pfadformel*** *ist, dann sind* ***E*** *f und* ***A*** *f* ***Zustandsformeln****.*

Die ***Syntax der Pfadformeln von CTL*** *wird durch folgende Regel festgelegt:*

1. *Wenn f und g* ***Zustandsformeln*** *sind, dann sind* ***X*** *f,* ***F*** *f,* ***G*** *f, f* ***U*** *g, und f* ***R*** *g* ***Pfadformeln****.*

CTL *ist die Menge der Zustandsformeln, die durch diese Regeln gebildet werden können.*

Zulässige CTL-Formeln sind also Zustandsformeln. Sobald man mit dieser Logik die Eigenschaften eines Pfades durch temporale Operatoren, wie zum Beispiel $\mathbf{X}, \mathbf{F}, \ldots$ beschreiben will, kann man nur durch Anwenden der 3. Zustandsformel wieder von einer Pfadformel zu einer Zustandsformel gelangen. Also muss vor jedem temporalen Operator ein Pfadquantor, das heißt **E** oder **A**, stehen. Die temporalen Operatoren bestimmen quantitativ über die Pfade, die von einem Zustand aus möglich sind.

Definition 6. *Sei AE die Menge der atomaren Einheiten. Die* ***Syntax der Zustandsformeln von LTL*** *wird durch folgende Regel festgelegt:*

1. *Wenn f eine* ***Pfadformel*** *ist, dann sind* ***E*** *f und* ***A*** *f* ***Zustandsformeln****.*

Die ***Syntax der Pfadformeln von LTL*** *wird durch folgende Regeln festgelegt:*

1. *Wenn $p \in AE$, dann ist p eine* ***Pfadformel****.*

2. *Wenn f und g* ***Pfadformeln*** *sind, dann sind auch $\neg f, f \vee g, f \wedge g,$* $\boldsymbol{X}\ f$, $\boldsymbol{F}\ f, \boldsymbol{G}\ f, f\ \boldsymbol{U}\ g$, *und f* $\boldsymbol{R}\ g$ ***Pfadformeln****.*

LTL *ist die Menge der Zustandsformeln, die durch diese Regeln gebildet werden können.*

Nach dieser Definition kann man LTL-Formeln nur mit temporalen Operatoren erweitern, wenn man bei den Pfadformeln bleibt. Der einzige Weg aus einer Pfadformel eine Zustandsformel zu machen, ist das Anwenden der Regel für die Zustandsformel. Hat man allerdings eine Zustandsformel, dann gibt es keine Möglichkeit mehr daraus eine Pfadformel zu machen und temporale Operatoren zu verwenden. LTL-Formeln lassen sich also immer in der Form $\mathcal{F} = \mathbf{E}\ f$ oder $\mathcal{F} = \mathbf{A}\ f$ schreiben, wobei f eine Pfadformel ist.
Man kann zeigen [7], dass die drei Logiken unterschiedliche Ausdrucksstärken haben. Dazu ein paar Beispiele:

Die Formel $\mathbf{A}(\mathbf{FG}\ p)$ ist eine LTL-Formel, denn:
1. $p \Rightarrow$ Pfadformel
2. $\mathbf{G}\ p \Rightarrow$ Pfadformel
3. $\mathbf{FG}\ p \Rightarrow$ Pfadformel
4. $\mathbf{A}(\mathbf{FG}\ p) \Rightarrow$ Zustandsformel $\surd$

Diese Formel beschreibt die Eigenschaft, dass es auf jedem Pfad ($\mathbf{A}$) in der Zukunft einen Zustand gibt ($\mathbf{F}$), ab dem p für immer gilt ($\mathbf{G}\ p$).

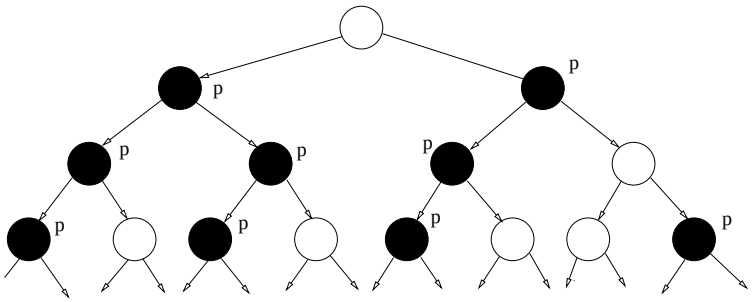


Abb. 5. Berechnungsbaum für die Formel $\mathbf{A}(\mathbf{FG}\ p)$. Bei den schwarzen Zuständen ist die atomare Einheit $p \equiv True$

Die Formel $\mathbf{A}(\mathbf{FG}\ p)$ ist keine CTL-Formel, denn:
1. $p \Rightarrow$ Zustandsformel
2. $\mathbf{G}\ p \Rightarrow$ Pfadformel $\perp$
(Eine Pfadformel kann bei CTL nur durch $\mathbf{A}$ oder $\mathbf{E}$ erweitert werden)

Die Formel $\mathbf{AG}(\mathbf{EF}\ p)$ ist eine CTL-Formel, denn:
1. $p \Rightarrow$ Zustandsformel

2. **F** $p \Rightarrow$ Pfadformel
3. (**EF** p) $\Rightarrow$ Zustandsformel
4. **G**(**EF** p) $\Rightarrow$ Pfadformel
5. **AG**(**EF** p) $\Rightarrow$ Zustandsformel $\surd$

Die Formel beschreibt die Eigenschaft, dass es von allen Zuständen aus (**AG**) einen Pfad gibt (**E**), auf dem p irgendwann gilt (**F** p).

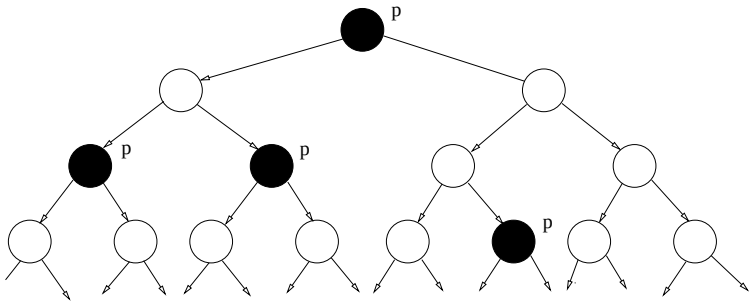


Abb. 6. Berechnungsbaum für die Formel **AG**(**EF** p).

Wenn man bei obiger Formel die Operatoren vertauscht, entsteht die Formel **EF**(**AG** p). Diese Formel beschreibt die Eigenschaft, dass es im Berechnungsbaum einen Zustand gibt, ab dem alle Zustände auf allen Pfaden, die von diesem Zustand weggehen, p erfüllen. Diese Eigenschaft kann man mit einer LTL-Formel nicht beschreiben, denn die Pfadquantifizierung in einer beliebigen horizontalen Ebene ist dort nicht möglich.

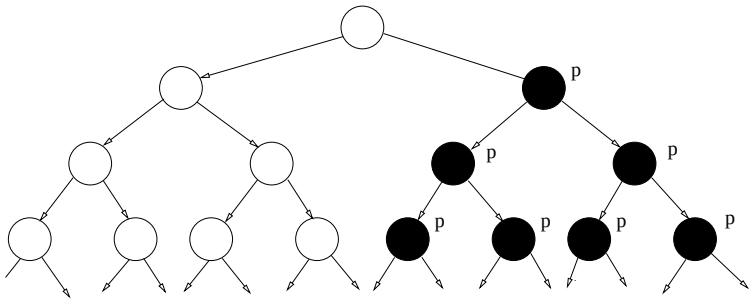


Abb. 7. Berechnungsbaum für die Formel **EF**(**AG** p).

Die Formel $\mathbf{AG}(\mathbf{EF}\ p)$ ist keine LTL-Formel, denn:
1. $p \Rightarrow$ Pfadformel
2. $\mathbf{F}\ p \Rightarrow$ Pfadformel
3. $\mathbf{E}(\mathbf{F}\ p) \Rightarrow$ Zustandsformel $\bot$
(Man kann hier keinen temporalen Operator auf eine Zustandsformel anwenden.)
Die Disjunktion der beiden Formeln $\mathbf{AG}(\mathbf{EF}\ p) \vee \mathbf{A}(\mathbf{FG}\ p)$ ergibt eine CTL*-Formel, die weder in CTL noch in LTL ausdrückbar ist.

CTL Wir haben gesehen, dass bei CTL vor jedem temporalen Operator ein Pfadquantor stehen muss. In Kombination mit den fünf aufgeführten temporalen Operatoren, ergeben sich mit den Pfadquantoren zehn CTL-Operatoren:

- **AX** und **EX**
- **AF** und **EF**
- **AG** und **EG**
- **AU** und **EU**
- **AR** und **ER**

Jeder dieser Operatoren kann durch die Operatoren **EX**, **EG** und **EU** ausgedrückt werden:

1. $\mathbf{AX}\ f \quad = \quad \neg\mathbf{EX}\ \neg f$
2. $\mathbf{EF}\ f \quad = \quad \mathbf{E}[True\ \mathbf{U}\ f]$
3. $\mathbf{AG}\ f \quad = \quad \neg\mathbf{EF}(\neg f)$
4. $\mathbf{AF}\ f \quad = \quad \neg\mathbf{EG}(\neg f)$
5. $\mathbf{A}[f\ \mathbf{U}\ g] \quad = \quad \neg\mathbf{E}[\neg g\ \mathbf{U}\ (\neg f \wedge \neg g)] \wedge \neg\mathbf{EG}\ \neg g$
6. $\mathbf{A}[f\ \mathbf{R}\ g] \quad = \quad \neg\mathbf{E}[\neg f\ \mathbf{U}\ \neg g]$
7. $\mathbf{E}[f\ \mathbf{R}\ g] \quad = \quad \neg\mathbf{A}[\neg f\ \mathbf{U}\ \neg g]$

In den folgenden Beispielen für CTL-Operatoren (siehe Abbildungen 8 bis 11) soll der oberste Zustand jeweils der Zustand s_0 sein.

$$
\begin{aligned}
K, s_0 \models \mathbf{EF}\ g \equiv\ & \text{Es existiert ein Pfad } \pi \text{ ausgehend von } s_0\text{, für den gilt:} \\
& \exists k \geq 0 : K, \pi^k \models g \\
K, s_0 \models \mathbf{AF}\ g \equiv\ & \text{Für alle Pfade } \pi \text{ ausgehend von } s_0 \text{ gilt:} \\
& \exists k \geq 0 : K, \pi^k \models g \\
K, s_0 \models \mathbf{EG}\ g \equiv\ & \text{Es existiert ein Pfad } \pi \text{ ausgehend von } s_0\text{, für den gilt:} \\
& \forall k \geq 0 : K, \pi^k \models g \\
K, s_0 \models \mathbf{AG}\ g \equiv\ & \text{Für alle Pfade } \pi \text{ ausgehend von } s_0 \text{ gilt:} \\
& \forall k \geq 0 : K, \pi^k \models g
\end{aligned}
$$

Weitere typische Beispiele für CTL-Formeln beim Verifizieren eines Systems:

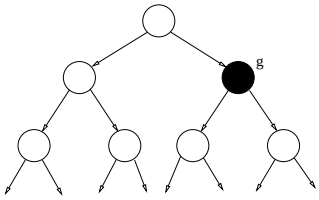


Abb. 8. $K, s_0 \models \mathbf{EF}\ g$

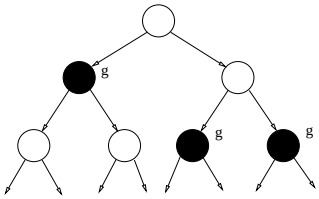


Abb. 9. $K, s_0 \models \mathbf{AF}\ g$

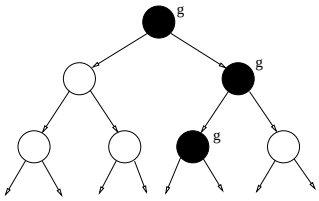


Abb. 10. $K, s_0 \models \mathbf{EG}\ g$

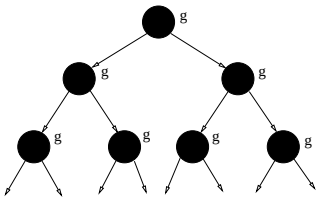


Abb. 11. $K, s_0 \models \mathbf{AG}\ g$

- **EF**($start \wedge \neg bereit$): Man kann zu einem Zustand gelangen, bei dem man eine Aktion starten will, aber das System nicht bereit ist.
- **AG**($anfrage \rightarrow$ **AF** $annahme$): Wenn eine Anfrage anfällt, dann wird sie irgendwann angenommen.
- **AG**(**EF** $restart$): Von jedem Zustand kann man zum Restart-Zustand gelangen.

ACTL, ACTL* Wenn man CTL auf Allquantoren einschränkt, also die Existenzquantoren verbietet, erhält man ACTL. Genauso erhält man bei der gleichen Einschränkung von CTL* ACTL*. Da bei der Negation Existenzquantoren auftreten könnten, nimmt man an, dass die Formeln in konjunktiver Normalform vorliegen.

Beispiel:
AF AG a und **AF AX** a sind Beispiele für ACTL-Formeln. Sie sind nicht in LTL ausdrückbar.
AG EF b und **AG**¬**AF** b sind keine ACTL-Formeln.

2.3 Fairness

Bei der Verifikation eines Systems ist man oft nur an der Korrektheit von fairen, bzw. ordentlichen Berechnungspfaden interessiert. Das heißt, Bedienungsfehler von Benutzern oder spezielle interne Zustände, bei denen die Verifikation nicht funktionieren kann, sollen ausgeschlossen werden. Da solche Ausdrücke nicht direkt in CTL, sondern in CTL* ausgedrückt werden können, muß die Semantik

von CTL angepaßt werden. Die neue Semantik soll faire Semantik heißen. Ein Fairness Constraint kann eine beliebige Menge von Zuständen sein. Wenn die Fairness Constraints als CTL-Formeln interpretiert werden, dann ist ein Pfad fair, wenn jeder Constraint unendlich oft auf dem Pfad wahr ist. Die Pfadquantifikatoren der Logik sind dann auf faire Pfade eingeschränkt.

Definition 7. *Eine **faire Kripke Struktur** ist ein 4-Tupel $K = (S, R, L, F)$, wobei S, L, und R wie bei einer Kripke Struktur definiert sind und $F \subset 2^S$ eine Menge von Fairness Constraints sind.*

Per Definition ist ein Pfad eine unendliche Sequenz von Zuständen $\pi = s_0, s_1, \ldots$. Die (unendliche) Menge dieser Zustände sei durch

$$inf(\pi) = \{s | s = s_i \text{ für unendlich viele } i\}$$

definiert.

Definition 8. *Ein **Pfad** π in einer fairen Kripke Struktur, ist **fair** $\Leftrightarrow$ Für jedes $P \in F, inf(\pi) \cap P \neq \emptyset$.*

Die Semantik von CTL* in Bezug auf einer fairen Kripke Struktur:

- Wenn f eine **Zustandsformel** ist, dann bedeutet $K, s \models_F f$, dass die Formel f bei dem Zustand s in der fairen Kripke Struktur K erfüllt ist.

- Wenn g eine **Pfadformel** ist, dann bedeutet $K, \pi \models_F g$, dass g auf dem Pfad π der fairen Kripke Struktur K erfüllt ist.

Die Semantik der $\models$ Relation ändert sich bezüglich Definition 4 in den Regeln 1,5 und 6:

1.	$K, s \models_F p$	$\Leftrightarrow$	Es gibt einen fairen Pfad, der bei s anfängt und $p \in L(s)$
5.	$K, s \models_F \mathbf{E}\ g_1$	$\Leftrightarrow$	Es gibt einen fairen Pfad π, der bei s anfängt, so dass $K, \pi \models_F g_1$
6.	$K, s \models_F \mathbf{A}\ g_1$	$\Leftrightarrow$	Für jeden fairen Pfad π, der bei s anfängt, gilt $K, \pi \models_F g_1$

Beispiel: Ein Kommunikationsprotokoll soll auf zuverlässigen Übertragungskanälen getestet werden. Es gibt einen Fairness Constraint für jeden Kanal, der die Zuverlässigkeit des Kanals beschreibt. Für Kanal i soll ein Fairness Constraint F_1 durch die Formel

$$\begin{aligned} \mathcal{F}_1 &= gesendet_i \rightarrow empfangen_i \\ &\equiv \neg gesendet_i \vee empfangen_i \end{aligned}$$

beschrieben werden. Das ist die Menge

$$\begin{aligned} F_1 = \{s \mid s \models \mathcal{F}_1\} = \{&\langle gesendet_i \leftarrow False, empfangen_i \leftarrow False, \ldots\rangle, \\ &\langle gesendet_i \leftarrow False, empfangen_i \leftarrow True, \ldots\rangle, \\ &\langle gesendet_i \leftarrow True, empfangen_i \leftarrow True, \ldots\rangle\}. \end{aligned}$$

Also die Menge der Zustände, bei denen entweder nichts gesendet wurde oder aber etwas empfangen wurde. Da die anderen Systemvariablen nicht in $\mathcal{F}_1$ auftauchen, können sie beliebige Werte annehmen und man muß alle Kombinationen in die Menge F_1 miteinbeziehen. Ausserdem soll ein zweiter Fairness Constraint F_2 durch die Formel

$$\mathcal{F}_2 = bereit_i \wedge \neg fehler_i$$

beschrieben werden. Das ist die Menge

$$F_2 = \{s \mid s \models \mathcal{F}_2\} = \{\langle bereit_i \leftarrow True, fehler_i \leftarrow False, \dots\rangle\}$$

Die Menge F der Fairness Constraints ist also

$$F = \{F_1, F_2\}.$$

Der Pfad $\pi = s_0, s_1, s_0, s_1, \dots$ mit den Zuständen

$$\begin{aligned} s_0 = \langle & gesendet_i \leftarrow False, empfangen_i \leftarrow True, \\ & bereit_i \leftarrow False, fehler_i \leftarrow False, \dots\rangle \\ s_1 = \langle & gesendet_i \leftarrow True, empfangen_i \leftarrow False, \\ & bereit_i \leftarrow True, fehler_i \leftarrow False, \dots\rangle \end{aligned}$$

ist fair, denn für $F_1 \in F$ gilt :

$$\begin{aligned} & \{s_0, s_1, s_0, s_1, \dots\} \cap F_1 \equiv \\ & \{\langle gesendet_i \leftarrow False, empfangen_i \leftarrow True, \dots\rangle \\ & \langle gesendet_i \leftarrow True, empfangen_i \leftarrow False, \dots\rangle, \dots\} \\ & \cap \\ & \{\langle gesendet_i \leftarrow False, empfangen_i \leftarrow False, \dots\rangle, \\ & \langle gesendet_i \leftarrow False, empfangen_i \leftarrow True, \dots\rangle, \\ & \langle gesendet_i \leftarrow True, empfangen_i \leftarrow True, \dots\rangle\} \\ & \equiv \\ & \{\langle gesendet_i \leftarrow False, empfangen_i \leftarrow True, \\ & bereit_i \leftarrow False, fehler_i \leftarrow False\rangle, \dots\} \equiv \{s_0, s_0, \dots\} \neq \emptyset \end{aligned}$$

und für $F_2 \in F$ gilt:

$$
\begin{aligned}
&\{s_0, s_1, s_0, s_1, \dots\} \cap F_2 \equiv \\
&\{\langle bereit_i \leftarrow True, fehler_i \leftarrow False, \dots\rangle, \dots\} \\
&\quad \cap \\
&\{\langle bereit_i \leftarrow False, fehler_i \leftarrow False, \dots\rangle, \\
&\quad \langle bereit_i \leftarrow True, fehler_i \leftarrow False, \dots\rangle\} \\
&\quad \equiv \\
&\{\langle gesendet_i \leftarrow True, empfangen_i \leftarrow False, \\
&\quad bereit_i \leftarrow True, fehler_i \leftarrow False\rangle, \dots\} \equiv \{s_1, s_1, \dots\} \neq \emptyset
\end{aligned}
$$

Der Pfad enthält also ein Element von jedem Constraint unendlich oft, nämlich $s_0 \in F_1$ und $s_1 \in F_2$. Interpretiert man die Fairness Constraints als CTL-Formeln, dann ist jeder Constraint unendlich oft wahr auf dem Pfad, denn $s_0 \models \mathcal{F}_1$ und $s_1 \models \mathcal{F}_2$.

3 Model Checking

Das Model Checking Problem besteht darin, aus einer gegebenen Kripke Struktur $K = (S, R, L)$ und einer Formel $\mathcal{F}$ der temporalen Logik, die eine Anfrage an das System darstellt, die Menge der Zustände zu finden, in denen die Formel $\mathcal{F}$ erfüllt ist, also die Menge:

$$\{s \in S \mid K, s \models \mathcal{F}\}$$

Das System erfüllt die Anforderung $\mathcal{F}$, wenn alle Anfangszustände $s_0 \in S_0$ des Systems in obiger Menge enthalten sind.

3.1 CTL Model Checking

$K = (S, R, L)$ sei eine Kripke Struktur. Wir wollen herausfinden, welche Zustände in S die CTL-Formel $\mathcal{F}$ erfüllen. Der Algorithmus markiert jeden Zustand s mit der Menge $label(s)$ der Teilformeln von $\mathcal{F}$, die für s wahr sind. Am Anfang ist $label(s) \equiv L(s)$. Der Algorithmus geht dann durch i verschiedene Stufen, bei denen Teilformeln mit $(i-1)$ geschachtelten CTL-Operatoren bearbeitet werden. Wenn eine Teilformel bearbeitet wird, wird sie zu jeder $label(s)$-Menge eines Zustands s hinzugefügt, in dem sie erfüllt ist. Wenn der Algorithmus beendet, gilt für jeden Zustand :

$$K, s \models \mathcal{F} \Leftrightarrow \mathcal{F} \in label(s).$$

Da jede CTL-Formel durch $\neg, \vee, \mathbf{EX}, \mathbf{EU}$ und $\mathbf{EG}$ ausgedrückt werden kann, reichen sechs Fälle für die Zwischenstufen, bei denen die Formel abgearbeitet wird.

- $\neg f_1$: Alle Zustände, die nicht durch f_1 markiert werden, werden markiert.

- f_1 oder f_2: Alle Zustände, die entweder durch f_1 oder durch f_2 markiert werden, werden markiert.
- **EX** f_1: Jeder Zustand, der einen Nachfolger hat, der durch f_1 markiert wird, wird markiert.
- **E**$[f_1$ **U** $f_2]$: Zuerst werden alle Zustände, die durch f_2 markiert werden, gesucht. Dann werden alle Zustände markiert, die durch die umgekehrten Übergangsrelationen R von dort aus erreicht werden können, und durch f_1 markiert werden.
- **EG** f_1: Es wird mit f_1 eine eingeschränkte Kripke Struktur $K' = (S', R', L')$ konstruiert (siehe unten). Mit dem Algorithmus von Tarjan wird der Graph (S', R') in *stark zusammenhängende Komponenten* zerlegt (siehe unten). Danach werden alle Zustände, die zu nichttrivialen Komponenten gehören, gesucht. Von diesen Zuständen werden alle Zustände gesucht, die durch die umgekehrten Übergangsrelationen R' erreicht werden können und mit f_1 markiert.

Definition 9. *Eine* ***stark zusammenhängende Komponente*** *C ist ein maximaler Teilgraph, so dass jeder Knoten in C von jedem anderen Knoten in C auf einem gerichteten Graph, der komplett in C enthalten ist, erreichbar ist. C heisst* ***nichttrivial*** *genau dann, wenn C mehr als einen Knoten oder einen Knoten mit einer Schleife enthält.*

K' sei eine eingeschränkte Kripke Struktur. Sie entsteht aus K, indem man aus S alle Zustände entfernt, in denen f_1 nicht gilt. R und L werden entsprechend eingeschränkt. Also ist $K' = (S', R', L')$ mit $S' = \{s \in S \mid K, s \models f_1\}$, $R' = R|_{S' \times S'}$ und $L' = L|_{S'}$.
Der Algorithmus hängt von folgender Beobachtung ab:

Lemma 1. *$K, s \models$ **EG** f_1 genau dann, wenn:*
1. $s \in S'$
2. Es gibt einen Pfad in K' der von s zu einem Knoten t, in einer nichtrivialen stark zusammenhängenden Komponente C des Graphen (S', R') führt.

Beweis:"⇒": Annahme: $K, s \models$ **EG** f_1. $s \in S'$ ist erfüllt, denn $S' = \{s \in S \mid K, s \models f_1\}$. π sei ein unendlicher Pfad, der bei s anfängt. Auf jedem Zustand von π soll f_1 gelten. Da die Struktur KS endlich ist, der Pfad π aber unendlich, kann man den Pfad in $\pi = \pi_0\pi_1$ aufteilen, wobei π_0 das endliche Anfangssegment ist, und π_1 ein unendliches Suffix von π ist, das sich zwangsweise immer wieder wiederholt, da wir ja nur endlich viele Zustände haben. Also taucht jeder Zustand auf π_1 unendlich oft auf. Wenn man π_0 als Menge von Zuständen sieht, dann ist π_0 natürlich in S' enthalten, da f_1 in jedem Zustand auf π_0 gilt. Es sei C die Menge von Zuständen in π_1. Aus dem gleichen Grund wie oben ist C in S' enthalten. Wir zeigen nun einen Hilfssatz: Zwischen beliebigen Zustandspaaren in C gibt es einen Pfad in C. Es seien s_1 und s_2 zwei Zustände in C. s_1 wird als beliebiger Zustand auf π_1 gewählt. Durch die Art und Weise, auf die man π_1 ausgesucht hat, kommt man irgendwann zu einem beliebigen Zustand

s_2, indem man auf π_1 von s_1 weitergeht. Das Segment zwischen s_1 und s_2 muss vollständig in C liegen. Dieses Segment ist ein endlicher Pfad von s_1 nach s_2, der in C liegt. Durch den Hilfssatz haben wir gezeigt, dass jeder Zustand von jedem anderen Zustand aus in C erreichbar ist. C ist also entweder eine stark zusammenhängende Komponente oder sie liegt in einer stark zusammenhängenden Komponente. Für den Fall, dass sie ein maximaler Teilgraph ist, ist C eine stark zusammenhängende Komponente. Wenn nicht, dann liegt C in einer stark zusammenhängenden Komponente. In beiden Fällen sind die Bedingungen (1) und (2) erfüllt.
"$\Leftarrow$": Annahme: Die Bedingungen (1) und (2) seien erfüllt. π_0 sei der Pfad von s nach t. π_1 sei ein Pfad mit einer Länge von mindestens 1, der von t wieder nach t führt. Die Existenz ist garantiert, denn t ist nach (2) ein Zustand in einer *nichttrivialen* stark zusammenhängenden Komponente C. Jeder Zustand auf dem unendlichen Pfad $\pi = \pi_0\pi_1^\omega$ erfüllt f_1, denn π ist nach (2) aus dem Graphen $K' = (S', R', L')$ und für Zustände $s \in S'$ gilt $K, s \models f_1$. Da $M' \subseteq M$ kann π auch in M beginnen. Dann haben wir einen Pfad gefunden, auf dem für jeden Zustand f_1 gilt, also gilt die Formel $K, s \models \mathbf{EG}\ f_1$.

□

Wenn eine Formel f ein Anzahl von $|f|$ Teilformeln hat, dann braucht der Algorithmus $O(|f|(|S| + |R|))$, denn die Zerlegung des Graphen mit dem Algorithmus von Tarjan benötigt $O(|S'| + |R'|)$.

Beispiel: Das Verhalten einer Mikrowelle, die durch Abbildung 12 beschrieben wird, soll untersucht werden.
Die Zustände sind durch Nummern markiert. Der Zustand mit Nummer 1 ist zum Beispiel $s_1 = \langle Start = False, Close = False, Heat = False, Error = False\rangle$.

Die Formel $\mathcal{F} = \mathbf{AG}(Start \leftarrow \mathbf{AF}Heat)$ soll in der Struktur bestätigt werden. Man will spezifizieren, dass für jeden Zustand der Maschine (**AG**) gilt : Wenn sie in Betrieb ist ($Start = True$), dann wird sie auch auf jeden Fall in der Zukunft heizen (**AF**$Heat$).

$$\begin{aligned}\mathcal{F} &= \mathbf{AG}(Start \leftarrow \mathbf{AF}Heat)\\ &\equiv \neg\mathbf{EF}(Start \wedge \neg\mathbf{AF}Heat)\\ &\equiv \neg\mathbf{EF}(Start \wedge \mathbf{EG}\neg Heat)\end{aligned}$$

wobei **EF** eine Abkürzung für $\mathbf{E}[True\ \mathbf{U}\ f]$ ist. Wir beginnen mit der Suche nach den Zuständen, die die atomaren Formeln erfüllen und fahren mit den komplizierteren Teilformeln fort. $S(g)$ sei die Menge der Zustände, die durch die Teilformel g markiert werden. Es gilt $f \in label(s) \Leftrightarrow s \models f$.

$$\begin{aligned}S(g) &= \{s \in S \mid g \in label(s)\}\\ &\equiv \{s \in S \mid s \models g\}\end{aligned}$$

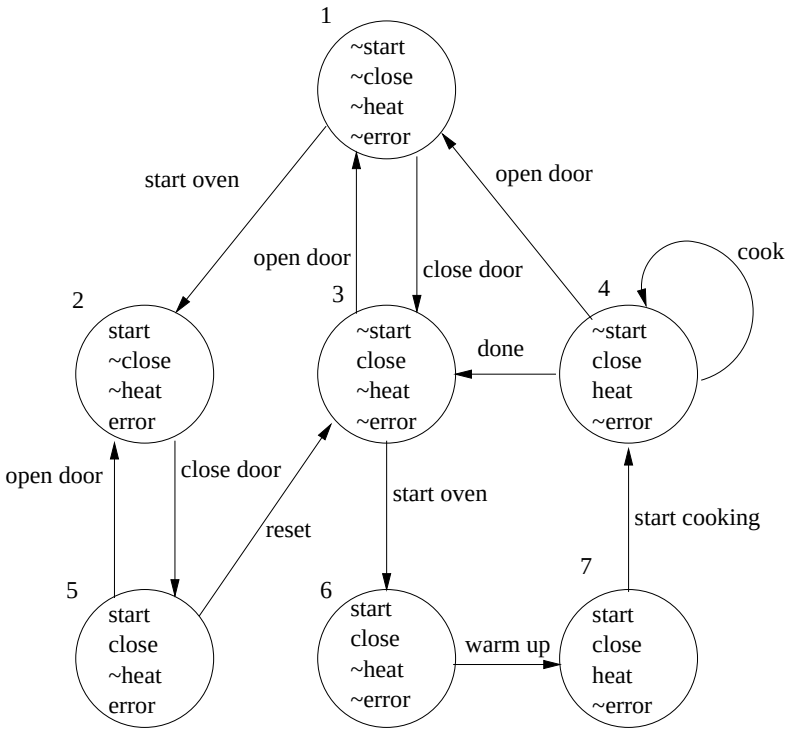


Abb. 12. Modell einer Mikrowelle

Atomare Formeln sind in diesem Fall $Start$ und $\neg Heat$.

$$S(Start) = \{2, 5, 6, 7\}$$
$$S(\neg Heat) = \{1, 2, 3, 5, 6\}$$

Für die Berechnung von $S(\mathbf{EG}\ \neg Heat)$ muss man zuerst die Menge der nichttrivial stark zusammenhängenden Komponenten SZK in $S' = S(\neg Heat)$ finden. Es gilt $SZK = \{\{1, 2, 3, 5\}\}$, da die Knoten 1,2,3 und 5 einen Kreis bilden. Der Knoten 6 ist nicht enthalten, da die anderen Knoten von ihm aus nicht erreichbar sind. Die Knoten, die zu nichttrivialen Komponenten gehören, werden nun gesucht. SZK ist eine nichttriviale Komponente, d. h. unsere Suche beginnt mit den Knoten $T = \{1, 2, 3, 5\}$. Wir suchen alle Knoten in S', die durch die umgekehrte Relation R' erreicht werden können. Der einzige Knoten, der in Frage kommt, ist der Knoten 6. Durch die umgekehrte Relation kann er aber nicht erreicht werden. Also ist

$$S(\mathbf{EG}\ \neg Heat) = \{1, 2, 3, 5\}$$

und

$$S(Start \wedge \mathbf{EG}\ \neg Heat) = \{2, 5\}$$

Die Menge $S(\mathbf{EF}(Start \wedge \mathbf{EG}\neg Heat))$ ist gleich der Menge $S(\mathbf{E}[True\ \mathbf{U}\ (Start \wedge \mathbf{EG}\neg Heat))$. Wir beginnen bei unserer Suche bei der Menge, für die gilt $S(Start \wedge \mathbf{EG}\neg Heat)) = \{2, 5\}$. Dann suchen wir alle Zustände, die durch die umgekehrten Übergangsrelationen erreicht werden können, und für die die Teilformel $True$ gilt, also ohne Einschränkung. Dadurch bekommen wir:

$$S(\mathbf{E}[True\ \mathbf{U}\ (Start \wedge \mathbf{EG}\neg Heat)) = \{1, 2, 3, 4, 5, 6, 7\}$$

Für die Negation brauchen wir die komplementäre Menge, also

$$S(\neg\mathbf{E}[True\ \mathbf{U}\ (Start \wedge \mathbf{EG}\neg Heat)) = \emptyset$$

Der Anfangszustand 1 ist nicht in der Menge enthalten, also erfüllt das System unsere Spezifikation nicht. Dass die Spezifikation nicht gilt kann man auch direkt an der Kripke Struktur erkennen. Die Aussage war, dass wenn die Mikrowelle in Betrieb ist, sie auch auf jeden Fall in Zukunft heizen wird. Es gibt jedoch einen Pfad, für den das nicht der Fall ist. Von den Zuständen 2 und 5 gibt es die Pfade $\pi_2 = s_2, s_5, s_3, s_1, \ldots$ und $\pi_5 = s_5, s_3, s_1, s_2, \ldots$, bei denen man auf keinen Zustand kommt, in dem $Heat = True$ gilt.

Fairness Constraints Um Fairness Constraints in den CTL-Modelchecking Algorithmus miteinzubeziehen, muss man den bisherigen Algorithmus erweitern.

Definition 10. *Es sei $K = (S, L, R, F)$ eine faire Kripke Struktur. Es sei $F = \{P_1, \ldots, P_k\}$ die Menge der Fairness Constraints. Eine stark zusammenhängende Komponente C des Graphen K ist **fair** bezüglich F genau dann wenn es für jede $P_i \in F$ einen Zustand $t_i \in (C \cap P_i)$ gibt.*

Der Algorithmus, um Teilformeln der Form $\mathbf{EG}\ f_1$ in Bezug auf eine faire Struktur zu bearbeiten, funktioniert ähnlich wie der uneingeschränkte CTL-Algorithmus. Wie vorher bekommt man K' aus K, indem aus S alle Zustände gelöscht werden, für die $s \models_F \neg f_i$ gilt. Also ist $K' = (S', R', L', F')$, mit $S' = \{s \in S \mid K, s \models_F f_1\}, R' = R|_{S' \times S'}, L' = L|_{S'}$ und $F' = \{P_i \cap S' \mid P_i \in F\}$.

Lemma 2. *$K, s \models_F \mathbf{EG}\ f_1$ genau dann, wenn*
1. $s \in S'$
2. Es existiert ein Pfad in S', der in einer nichttrivialen, fairen stark zusammenhängenden Komponente des Graphen (S', R') von s zu einem Knoten t führt.

Der Beweis zu diesem Lemma ist ähnlich dem Beweis zu Lemma 1. Die Vorgehensweise, um Teilformeln der Form $\mathbf{EG}\ f_1$ in einer fairen Semantik zu bearbeiten, ist ähnlich der eben besprochenen. $\mathbf{EG}\ f_1$ soll der Menge $label(s)$ für jeden Zustand s hinzugefügt werden, für den gilt $K, s \models_F \mathbf{EG}\ f_1$. Wir gehen davon aus, daß für jeden Zustand $f_1 \in label(s) \Leftrightarrow K, s \models_F f_1$. Der einzige Unterschied ist, daß die stark zusammenhängenden Komponenten nun aus der

Menge der nichttrivialen, fairen stark zusammenhängenden Komponenten bestehen. Die Komplexität dieser Berechnung ist $O((|S| + |R|)|F|)$, da festgestellt werden muß, welche stark zusammenhängende Komponente fair ist. Jede Komponente muß aus jeder Constraint-Menge einen Zustand besitzen. Wir führen eine zusätzliche Variable $fair$ ein, die genau dann in einem Zustand s wahr ist, wenn es einen fairen Pfad π gibt, der von diesem Zustand startet. In der fairen CTL-Semantik heißt das:
In einem Zustand s ist also $fair = True$ genau dann, wenn es einen fairen Pfad π gibt, der von s startet, $\pi \models_F True \Leftrightarrow K, s \models_F \mathbf{EG}\ True$. Die Prozedur CheckFairEG(True) kann dazu benutzt werden, um Zustände mit der neuen Variablen zu markieren. Anfragen an faire Kripke-Strukturen werden folgendermassen übersetzt:

- $K, s \models_F p, p \in AE$ wird mit der CTL-Modelcheckingmethode die Formel $K, s \models p \wedge fair$ überprüft.
- $K, s \models_F \mathbf{EX}\ f_1$ wird $K, s \models \mathbf{EX}\ (f_1 \wedge fair)$ überprüft.
- $K, s \models_F \mathbf{E}[f_1\ \mathbf{U}\ f_2]$ wird $K, s \models \mathbf{E}[f_1\ \mathbf{U}\ (f_2 \wedge fair)]$ überprüft.

Für die Komplexität ergibt sich $O(|f|(|R| + |S|)|F|)$.

Beispiel: Dieses mal sollen beim Überprüfen der Formel $\mathbf{AG}(Start \leftarrow \mathbf{AF}Heat)$ nur Pfade betrachtet werden, bei denen der Benutzer die Mikrowelle korrekt bedient. Es soll auf den Pfaden unendlich oft $Start \wedge Close \wedge \neg Error$ gelten. $F = \{P\}, P = \{s | s \models Start \wedge Close \wedge \neg Error\}.S(Start)$ und $S(\neg Heat)$ bleiben gleich. Beim Berechnen der Menge der stark zusammenhängenden Komponenten von $S' = S(\neg Heat)$ gibt es jedoch einen Unterschied. Die Komponente $\{1, 2, 3, 5\}$ ist nicht fair, denn es muß gelten:
$\exists s \in \{1, 2, 3, 5\}, s \models Start \wedge Close \wedge \neg Error$
Also ist

$$S(\mathbf{EG}\ \neg Heat) = \emptyset$$
$$S(\mathbf{EF}(Start \wedge \mathbf{EG}\ \neg Heat)) = \emptyset$$
$$S(\neg(\mathbf{EF}(Start \wedge \mathbf{EG}\ \neg Heat))) = \{1, 2, 3, 4, 5, 6, 7\}$$

Da $1 \in \{1, 2, 3, 4, 5, 6, 7\}$ erfüllt das Programm die Formel mit den gegebenen Fairness Constraints!

3.2 LTL Model Checking mit der Tableautechnik

Es sei $KS = (S, R, L)$ eine Kripke Struktur mit $s \in S$. Es sei $\mathcal{F} = \mathbf{A}\ g$ eine LTL-Formel. Die Teilformel g muss also eine eingeschränkte Pfadformel sein, bei der nur Zustandsteilformeln in Form von atomaren Einheiten vorkommen. Wir wollen nun herausfinden, ob $KS, s \models \mathbf{A}\ g$. Da $\mathcal{F} \equiv \neg\mathbf{E}\ \neg g$ ist, reicht es aus, Formeln der Form $\mathbf{E}\ f$, bei denen f eine eingeschränkte Pfadformel ist, zu überprüfen. Im Allgemeinen ist das Problem **PSPACE**-Vollständig [4, 5]. Der

Beweis ist sehr kompliziert und wird hier nicht gezeigt. Es läßt sich auch zeigen, daß das Modelchecking Problem für Formeln der Form $\mathbf{E}\ f$, bei denen f eine eingeschränkte Pfadformel ist, **NP**-hart ist. Man betrachte einen beliebigen, gerichteten Graphen $G = (V, A)$, mit $V = \{v_1, v_2, v_3, \ldots v_n\}$. Wir zeigen nun, dass man das Problem, bei einem Graphen G festzustellen, ob er einen gerichteten Hamilton'schen Pfad besitzt, auf das Problem reduzierbar ist, herauszufinden, ob $KS, s \models \mathcal{F}$, bei dem gilt:

- KS ist eine endliche Kripke Struktur
- s ist ein Zustand in KS
- $\mathcal{F}$ ist eine Formel mit atomaren Einheiten $p_1, \ldots, p_n$, der Form:
  $\mathbf{E}[\mathbf{F}\ p_1 \wedge \cdots \wedge \mathbf{F}\ p_n \wedge \mathbf{G}(p_1 \rightarrow \mathbf{X}\ \mathbf{G}\ \neg p_1) \wedge \cdots \wedge \mathbf{G}(p_n \rightarrow \mathbf{X}\ \mathbf{G}\ \neg p_n)]$.

Durch den Graphen G erhalten wir eine Struktur, indem wir in allen Knoten v_i von G die Einheiten $p_i = True$ setzen und bei allen anderen Knoten $p_i = False$ setzen, für $1 \leq i \leq n$. Wir setzen einen Eingangsknoten u_1, von dem aus alle v_i erreichbar sind (aber nicht umgekehrt), und einen Ausgangsknoten u_2, der von allen v_i aus erreichbar ist (aber nicht umgekehrt). Der Eingangsknoten stellt einen eindeutigen Anfangszustand dar, von dem man den Hamilton'schen Pfad auf jedem Knoten des Graphen beginnen kann. Der Ausgangsknoten versichert uns, dass die Übergangsrelation des Graphen total ist. Formal besteht die Kripke Struktur $KS = (U, B, L)$ aus folgenden Bestandteilen:

- $U = V \cup \{u_1, u_2\}$, mit $u_1, u_2 \notin V$.
- $B = A \cup \{(u_1, v_i) | v_i \in V\} \cup \{(_i, u_2) | v_i \in V\} \cup \{(u_2, u_2)\}$
- L ist eine Zuordnung, die atomare Einheiten Zuständen zuordnet, sodass gilt:
  - $p_i = True$ in v_i für $1 \leq i \leq n$.
  - $p_j = False$ in v_i für $1 \leq i, j \leq n, i \neq j$.
  - $p_i = False$ in u_1, u_2 für $1 \leq i \leq n$.

In dieser Struktur kann man erkennen, dass $KS, u_1 \models \mathcal{F}$ genau dann, wenn es in KS einen gerichteten unendlichen Pfad gibt, der bei u_1 beginnt, genau einmal durch alle $v_i \in V$ geht und in dem Übergang (u_2, u_2) bei u_2 endet. In dieser Konstruktion hat die Formel $\mathcal{F}$ ungefähr die gleiche Größe wie der Graph G. In [6] steht eine sorgfältige Analyse, die zeigt, dass wenn die Länge der Formel viel kleiner ist, als die Größe der Kripke Struktur, dann ist die Komplexität exponentiell in der Länge der Formel aber linear in der Grösse des Zustandsgraphen. Der Algorithmus, der dort vorgeschlagen wurde behilft sich einer impliziten *Tableaukonstruktion*. Ein **Tableau** ist ein aus einer Formel abgeleiteter Graph, aus dem genau dann ein Model für die Formel extrahiert werden kann, wenn die Formel erfüllbar ist. Der Algorithmus, um zu untersuchen ob K die Formel f erfüllt, verbindet das Tableau und die Struktur, um herauszufinden, ob es eine Berechnung in der Struktur gibt, die ein Pfad im Tableau ist. Im Folgenden sei der Algorithmus aus [6] beschrieben.
Es sei $K = (S, R, L)$ eine Kripke Struktur und f eine eingeschränkte Pfadformel. Es genügt die temporalen Operatoren $\mathbf{X}$ und $\mathbf{U}$ zu betrachten, denn

$\mathbf{F}\ f = True\ \mathbf{U}\ f, \mathbf{G}\ f = \neg\mathbf{F}\ \neg f$ und $f_1\ \mathbf{R}\ f_2 = \neg[\neg f_1\ \mathbf{U}\ \neg f_2]$.
Der **Abschluss** von f, $CL(f)$, enthält Formeln, dessen Wahrheitswerte den Wahrheitswert von f beeinflussen können.

Definition 11. *Der **Abschluss** $CL(f)$ von f ist die kleinste Menge von Formeln, die f enthält und folgende Eigenschaften erfüllt:*

1. $\neg f_1 \in CL(f) \Leftrightarrow f_1 \in CL(f)$
2. $f_1 \vee f_2 \in CL(f) \Rightarrow f_1, f_2 \in CL(f)$
3. $\boldsymbol{X}\ f_1 \in CL(f) \Rightarrow f_1 \in CL(f)$
4. $\neg\boldsymbol{X}\ f_1 \in CL(f) \Rightarrow \boldsymbol{X}\ \neg f_1 \in CL(f)$
5. $f_1\ \boldsymbol{U}\ f_2 \in CL(f) \Rightarrow f_1, f_2, \boldsymbol{X}[f_1\ \boldsymbol{U}\ f_2] \in CL(f)$

Es kann gezeigt werden, dass die Größe von $CL(f)$ linear mit der Größe von f verknüpft ist.

Definition 12. *Ein **Atom** ist ein Paar $A = (s_A, K_A)$, mit $s_A \in S$ und $K_A \subseteq CL(f) \cup AE$, sodass gilt:*

1. *Für jede Einheit $p \in AE, p \in K_A \Leftrightarrow p \in L(s_A)$*
2. *Für jedes $f_1 \in CL(f), f_1 \in K_A \Leftrightarrow \neg f_1 \notin K_A$*
3. *Für jedes $f_1 \vee f_2 \in CL(f), f_1 \vee f_2 \in K_A \Leftrightarrow f_1 \in K_A$ oder $f_1 \in K_A$*
4. *Für jedes $\neg\boldsymbol{X}\ f_1 \in CL(f), \neg\boldsymbol{X}\ f_1 \in K_A \Leftrightarrow \boldsymbol{X}\ \neg f_1 \in K_A$*
5. *Für jedes $f_1\ \boldsymbol{U}\ f_2 \in CL(f), f_1\ \boldsymbol{U}\ f_2 \in K_A \Leftrightarrow f_2 \in K_A$ oder $f_1, \boldsymbol{X}[f_1\ \boldsymbol{U}\ f_2] \in K_A$*

Ein Atom (s_A, K_A) ist also so definiert, dass K_A eine maximale, übereinstimmende Menge von Formeln ist, die auch mit der Markierung des Zustands s_A übereinstimmt.
Bei der Konstruktion eines Graphen G bildet die Menge der Atome die Menge der Knoten. (A, B) ist eine Kante von G, genau dann, wenn $(s_A, s_B) \in R$ und für jede Formel $\mathbf{X}\ f_1 \in CL(f), \mathbf{X}\ f_1 \in K_A \Leftrightarrow f_1 \in K_B$.

Definition 13. *Eine **mögliche Sequenz** ist ein unendlicher Pfad π in G, so dass wenn $f_1\ \boldsymbol{U}\ f_2 \in K_A$ für ein Atom A auf π, dann existiert ein Atom B, das von A auf π erreichbar ist, mit $f_2 \in B$.*

Lemma 3. *$KS, s \models \boldsymbol{E}\ f$ genau dann, wenn es eine **mögliche Sequenz** gibt, die bei dem Atom (s, K) beginnt, sodass $f \in K$.*

Beweisidee: "$\Leftarrow$": Wir nehmen an, dass es eine mögliche Sequenz (s_0, K_0), (s_1, K_1), ... gibt, die bei $(s, K) = (s_0, K_0)$ beginnt mit $f \in K$. Durch die Definition ist $\pi = s_0, s_1, \ldots$ ein Pfad in K, der in dem Zustand $s = s_0$ beginnt. Wir wollen zeigen, dass $\pi \models f$. Wir beweisen allerdings eine stärkere Behauptung: Für jedes $g \in CL(f)$ und jedes $i \geq 0$ gilt: $\pi^i \models g \Leftrightarrow g \in K_i$. Der Beweis basiert auf Induktion über die Struktur der Teilformeln. Hier soll der Basisfall und der Induktionsschritt, wenn g entweder $\neg h_1, h_1 \vee h_2, \mathbf{X}\ h_1$ oder $h_1\ \mathbf{U}\ h_2$ gezeigt werden.

1. Wenn g eine atomare Einheit ist, also $g \in AE$, dann gilt durch die Definition des Atoms, $g \in K_i \Leftrightarrow g \in L(s_i) \Leftrightarrow s_i \models g$
2. Wenn $g = \neg h_1$, dann gilt $\pi^i \models g \Leftrightarrow \pi^i \not\models h_1$. Das ist durch die Induktionsvorraussetzung genau dann wahr, wenn $h_1 \notin K_i$. Nach der Definition der Menge K_i bei der Definition eines Atoms ist $h_1 \notin K_i \Leftrightarrow \neg h_1 \in K_i \Leftrightarrow g \in K_i$.
3. Wenn $g = h_1 \vee h_2$, dann gilt: $\pi^i \models g \Leftrightarrow \pi^i \models h_1$ oder $\pi^i \models h_2$. Durch die Induktionsvorraussetzung ist das genau dann wahr, wenn $h_1 \in K_i$ oder $h_2 \in K_i$. Nach der Definition der Menge K_i ist $h_1 \in K_i$ oder $h_2 \in K_i \Leftrightarrow h_1 \vee h_2 \in K_i \Leftrightarrow g \in K_i$.
4. Wenn $g = \mathbf{X}\ h_1$, dann gilt: $\pi^i \models g \Leftrightarrow \pi^{i+1} \models h_1$. Durch die Induktionsvoraussetzung ist das genau dann wahr, wenn $h_1 \in K_{i+1}$. Da π ein Pfad in G ist, gilt $((s_i, K_i), (s_{i+1}, K_{i+1})) \in R$ und deshalb gilt obige Aussage genau dann, wenn $\mathbf{X}\ h_1 \in K_i \Leftrightarrow g \in K_i$.
5. "$\Leftarrow$": Annahme: $g = h_1\ \mathbf{U}\ h_2 \in K_i$. Durch die Definition einer möglichen Sequenz gibt es ein $j \geq i$, sodass $h_2 \in K_j$. Da nun $g \in K_i$ folgt durch die Defintion eines Atoms, dass $g \in K_i \Leftrightarrow (h_2 \in K_i) \vee (h_1 \in K_i$ und $\mathbf{X}\ g \in K_i) \Leftrightarrow h_2 \notin K_i \Rightarrow h_1 \in K_i$ und $\mathbf{X}\ g \in K_i$. Wenn das der Fall ist, also $h_2 \notin K_i \Rightarrow h_1 \in K_i$ und $\mathbf{X}\ g \in K_i$, dann folgt durch die Definition der Übergangsrelation , dass $g \in K_{i+1}$. Die Argumentation von oben wiederholt sich: Da nun $g \in K_{i+1}$ folgt durch die Definition ..., bis zu dem j bei dem gilt: $h_2 \in K_j$. Dadurch gilt für jedes $i \leq k < j, h_1 \in K_k$. Durch die Induktionshypothese folgt aus diesen Beobachtungen: $\pi^j \models h_2$ und für jedes $i \leq k < j, \pi^k \models h_1 \Leftrightarrow \pi^i \models g$ (siehe Definition des $\mathbf{U}$ Operators).
   "$\Rightarrow$": Wenn $\pi^i \models g \Leftrightarrow \pi^i \models h_1\ \mathbf{U}\ h_2 \Leftrightarrow$ Es existiert ein $j \geq i$, so dass $\pi^j \models h_2$ und für alle $i \leq k < j, \pi^k \models h_1$. Man wähle das kleinstmögliche j. Durch die Induktionshypothese ist $h_2 \in K_j$ und für jedes $i \leq k < j, h_1 \in K_k$. Annahme: $g \notin K_i$. Durch die Definition eines Atoms gilt dann: $g \notin K_i \Leftrightarrow \neg(g \in K_i) \Leftrightarrow \neg(h_2 \in K_i \vee (h_1 \in K_i \wedge \mathbf{X}\ g \in K_i)) \Leftrightarrow h_2 \notin K_i \wedge (h_1 \notin K_i \vee \mathbf{X}\ g \notin K_i))$. Da wir oben aber festgestellt hatten, dass $h_1 \in K_i$, muss gelten $\mathbf{X}\ g \notin K_i \Leftrightarrow \mathbf{X}\ \neg g \in K_i$. Durch die Definition der Übergangsrelation ist also $\neg g \in K_{i+1} \Leftrightarrow g \notin K_{i+1}$ und wir sind wieder bei der Argumentation von oben, die sich nun fortsetzen läßt, bis man zu einem $g \notin K_j$ gelangt. $g \notin K_j \Leftrightarrow h_2 \notin K_j \wedge (h_1 \notin K_j \vee \mathbf{X}\ g \notin K_j))$. Das ist aber ein Widerspruch, da $h_2 \in K_j$.

"$\Rightarrow$": Für die andere Richtung nehmen wir an, dass $KS, s \models \mathbf{E}\ f$. Also gibt es einen Pfad $\pi = s_0, s_1, s_2, \ldots$ von $s = s_0$, sodass $\pi \models f$. Wir definieren $K_i = \{g | g \in CL(f)$ und $\pi^i \models g\}$. Dadurch gelten folgende Eigenschaften:

1. (s_i, K_i) ist ein Atom. Das folgt aus der Defintion der Relation $\models$. Zum Beispiel sei $g \in CL(f)$. Ein Atom K_i darf entweder g oder $\neg g$ enthalten, aber nicht beide. Durch die Definition von $K_i, g \in K_i \Leftrightarrow \pi^i \models g$. Aber $\pi^i \models g \Leftrightarrow \pi^i \not\models \neg g$. Durch die Definition gilt: $\pi^i \not\models \neg g \Leftrightarrow \neg g \notin K_i$.
2. Es gibt einen Übergang von (s_i, K_i) zu (s_{i+1}, K_{i+1}). Dies folgt aus folgender Beobachtung: $\mathbf{X}\ g \in K_i \Leftrightarrow \pi^i \models \mathbf{X}\ g \Leftrightarrow \pi^{i+1} \models g$. Durch die Definition von K_{i+1} gilt: $\pi^{i+1} \models g \Leftrightarrow g \in K_{i+1}$. Also gilt $\mathbf{X}\ g \in K_i \Leftrightarrow g \in K_{i+1}$.

3. Die Sequenz $(s_0, K_0)(s_1, K_1), \ldots$ ist eine **mögliche Sequenz**. Denn wenn $g = h_1 \mathbf{U} h_2 \in K_i \Leftrightarrow \pi^i \models g \Rightarrow$ Es existiert ein $j \geq i$, sodass $\pi^j \models h_2$, und deshalb ist $h_2 \in K_j$.

□

Definition 14. *Ein nichttriviale stark zusammenhängende Komponente C eines Graphen G heißt **selbsterfüllend** genau dann, wenn für jedes Atom A in C und für jedes $f_1 \ \boldsymbol{U} \ f_2 \in K_A$ ein Atom B in C existiert, so dass $f_2 \in K_B$.*

Lemma 4. *Es gibt eine mögliche Sequenz, die bei einem Atom (s, K) beginnt, genau dann wenn es in G einen Pfad von (s, K) zu einer selbsterfüllenden stark zusammenhängenden Komponenten gibt.*

Beweis:"⇒": Annahme: Es gibt eine mögliche Sequenz, die bei einem Atom (s, K) beginnt. Man betrachte nun die Menge der Atome C', die unendlich oft auf dieser Sequenz auftauchen. Da sie miteinander verbunden sind, ist die Menge C' eine Teilmenge einer stark zusammenhängenden Komponenten C des Graphen G. Man betrachte die Teilformel $g = h_1 \mathbf{U} h_2$ und ein Atom $(s_A, K_A) \in C$, für das gilt $g \in K_A$. Da C stark zusammenhängend ist, gibt es in C einen endlichen Pfad, der von (s_A, K_A) nach C' führt. Wenn schon auf diesem Pfad h_2 auftaucht, dann haben wir das $B \in C$ gefunden, für das $h_2 \in K_b$. Also ist C eine selbsterfüllende Komponente. Wenn h_2 auf dem Pfad nicht auftaucht, dann gilt nach der Definition eines Atoms Eigenschaft 5: $g \in K_A \Leftrightarrow h_1 \in K_A$ und $\mathbf{X}\ g \in K_A$. Für das nächste Atom K_{A+1} auf dem Pfad gilt deshalb $g \in K_{A+1}$. Also ist g in jedem Atom auf dem Pfad und natürlich auch in einem Atom J aus C', denn der Pfad führt von (s_A, K_A) nach C'. Da jedoch C' aus dem Pfad einer möglichen Sequenz kommt, und $g \in J$, gibt es ein Atom $F \in C'$, für das $h_2 \in K_F$ und $F \in C' \subseteq C$. Aus $F \in C$ und $h_2 \in K_F$ folgt, dass C selbsterfüllend ist.
"⇐": Annahme: Es existiert ein Pfad von (s, K) zu einer selbsterfüllenden stark zusammenhängenden Komponente C. Innerhalb von C können wir eine Sequenz konstruieren, bei der jedes Atom mit einer Teilformel $h_1 \mathbf{U} h_2$ von einem Atom mit einer Teilformel h_2 gefolgt wird. Wir haben also eine mögliche Sequenz innerhalb der selbsterfüllenden Komponente C. Für Teilformeln der Form $h_1 \mathbf{U} h_2$, die auf dem Pfad von (s, K) nach C auftauchen, gilt unter ähnlichen Beobachtungen wie oben, folgendes: Jede dieser Teilformeln muss entweder durch ein Erscheinen von h_2 gefolgt werden, oder die Teilformel bleibt in jedem Atom auf dem Pfad enthalten, bis C erreicht wird. Da C selbsterfüllend ist, ist es dort möglich zu einem Atom F zu gelangen, für das gilt $h_2 \in F$. Also haben wir die Existenz einer möglichen Sequenz nachgewiesen.

□

Korollar 1. *$KS, s \models \boldsymbol{E}\ f \Leftrightarrow$ Es gibt ein Atom $A = (s, K)$ in G, so dass $f \in K$ und in G ein Pfad existiert, der von A zu einer selbsterfüllenden stark zusammenhängenden Komponente führt.*

Das Korollar 1 kann als Basis für einen LTL-Modelchecking Algorithmus benutzt werden. Dieser Algorithmus hat die Zeitkomplexität $O((|S| + |R|)2^{O(|f|)})$. Lichtenstein und Pnueli zeigen außerdem, wie man den Basisalgorithmus erweitern kann, um einige unterschiedliche Fairnessbegriffe mit ungefähr derselben Komplexität zu behandeln [6].

Beispiel: Es soll eine Anfrage in Form einer LTL-Formel an das Mikrowellenofen-System gestellt werden. Die Spezifikation ist $\mathcal{F}_{LTL} = \mathbf{A}[(\neg Heat)\ \mathbf{U}\ Close]$, die im Model dann wahr wird, wenn es für den Ofen unmöglich ist, mit der offenen Tür heiß zu werden. Um zu zeigen, dass diese Formel im Model erfüllt ist, werden wir zeigen, dass die Negation der Formel, $\mathcal{F}' = \mathbf{E}\neg[(\neg Heat)\ \mathbf{U}\ Close]$, nicht erfüllt ist. Es sei $f = (\neg Heat)\ \mathbf{U}\ Close$.Zuerst berechnen wir den Abschluss von $\neg f$, mit den Eigenschaften von CL.

$$
\begin{aligned}
CL(\neg f) &= \{\neg f\} \\
\text{Eig. 1: } CL(\neg f) &= \{\neg f, f\} \\
\text{Eig. 5: } CL(\neg f) &= \{\neg f, f, \neg Heat, Close, \mathbf{X}\ f\} \\
\text{Eig. 1: } CL(\neg f) &= \{\neg f, f, \neg Heat, Heat, Close, \neg Close, \mathbf{X}\ f, \neg\mathbf{X}\ f\} \\
\text{Eig. 4: } CL(\neg f) &= \{\neg f, f, \neg Heat, Heat, Close, \neg Close, \mathbf{X}\ f, \neg\mathbf{X}\ f, \mathbf{X}\ \neg f\}
\end{aligned}
$$

Da $\neg\mathbf{X}\ \neg f \equiv \mathbf{X}\ f$, ist CL vollständig. Als nächstes berechnen wir die Menge der Atome, die für den Graph G die Knoten bilden. Die letzte Eigenschaft bei der Definition des Atoms ergibt: $(\neg Heat)\ \mathbf{U}\ Close \in K_A \Leftrightarrow Close \in K_A \vee \neg Heat, \mathbf{X}((\neg Heat)\ \mathbf{U}\ Close) \in K_A$. K_A muss ausserdem mit der Markierungsfunktion, $L(s_A)$ übereinstimmen. Die atomaren Formeln $\neg Close$ und $\neg Heat$ sind in den Markierungsmengen der Zustände 1 und 2 enthalten. Alle Eigenschaften der Definition des Atoms müssen gelten:

$$K_1' = \{\neg Close, \neg Heat, f, \mathbf{X}\ f\}$$

Würde man zum Beispiel $\mathbf{X}\ \neg f$ auch noch aufnehmen, dann wäre die Eigenschaft 2 verletzt, denn $\mathbf{X}\ \neg f \in K_A \Leftrightarrow \neg\mathbf{X}\ f \in K_A$ und $\neg\mathbf{X}\ f \in K_A \Leftrightarrow \mathbf{X}\ f \notin K_A$.

$$K_1'' = \{\neg Close, \neg Heat, \neg f, \mathbf{X}\ \neg f, \neg\mathbf{X}\ f\}$$

In Kombination mit den Zuständen 1 und 2 bekommt man folgende Atome: $(1, K_1')$, $(2, K_1')$, $(1, K_1'')$, $(2, K_1'')$. Genauso bekommt man für die Zustände 3,5 und 6 folgende gültige Mengen:

$$K_2' = \{Close, \neg Heat, f, \mathbf{X}\ f\} \text{ oder } K_2'' = \{Close, \neg Heat, f, \mathbf{X}\ \neg f, \neg\mathbf{X}\ f\}$$

Und für die Zustände 4 und 7:

$$K_3' = \{Close, Heat, f, \mathbf{X}\ f\} \text{ oder } K_3'' = \{Close, Heat, f, \mathbf{X}\ \neg f, \neg\mathbf{X}\ f\}$$

Um die Übergangsrelationen zwischen den Atomen zu definieren, erinnern wir uns, dass es einen Übergang von einem Atom (s_A, K_A) zu einem Atom (s_B, K_B)

gibt, wenn es einen Übergang von s_A zu s_B, also $(s_A, s_B) \in R$ in der Kripke Struktur KS gibt und für jede Formel der Form $\mathbf{X}\ g \in CL(f), \mathbf{X}\ g \in K_A \Leftrightarrow g \in K_B$. Da $(1,2) \in R$ gibt es einen Übergang von $(1, K_1')$ zu $(2, K_1')$, denn $\mathbf{X}\ f \in K_1'$ und $f \in K_1'$. Es gibt auch einen Übergang von $(1, K_1'')$ zu $(2, K_1'')$, denn $\mathbf{X}\ \neg f \in K_1''$ und $\neg f \in K_1''$. Allerdings gibt es keinen Übergang von $(1, K_1')$ zu $(2, K_1'')$, denn $\mathbf{X}\ f \in K_1'$, aber $f \notin K_1''$. Der Rest der Relation wird genauso erzeugt. Korollar 1 sagt, dass ein Zustand s eine Formel $\neg f$ erfüllt, wenn es ein Atom (s, K) gibt, mit $\neg f \in K$ und es einen Pfad von (s, K) zu einer selbsterfüllenden stark zusammenhängenden Komponente in G gibt. Wenn der Graph vollständig konstruiert ist, kann man erkennen, dass es kein solches Atom gibt, das am Anfang eines unendlichen Pfads steht. Deshalb erfüllt kein Zustand $\mathbf{E}\ \neg f$. Also erfüllen alle Zustände $\mathbf{A}\ f$.

Literatur

1. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, 1999.
2. L. Bolc, A.Szalas *Time and Logic: a computational approach* UCL Press, 1995
3. A. V. Aho, J.E. Hopcroft, and I. D. Ullman *The Design and Analysis of Computer Algorithms* Addison Wesley, 1974
4. A. P. Sistla *Theoretical Issues in the Design and Verification of Distributed Systems* PhD thesis, Harvard University, 1983
5. A. P. Sistla and E. M. Clarke *Complexity of propositional temporal logics* Journal of the ACM 32(3):733-749
6. O. Lichtenstein and A. Pnueli Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages* pp. 97 - 107. ACM, 1985
7. E. M. Clarke and I. A. Draghicescu Expressibility results for linear time and branching time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency* LNCS 354, pp. 428-437 Springer, 1988

Methoden des Model Checking

Christian Mariner

Institut für Informatik
Albert-Ludwigs-Universtität Freiburg
mariner@informatik.uni-freiburg.de

1 Einleitung

In dieser Seminararbeit werden, ausgehend von den im vorangehenden Kapitel erworbenen Grundkentnissen, weitere Techniken des Model Checking beschrieben. Im ersten Abschnitt werden die Techniken des CTL und des LTL Model Checkings kombiniert, um auch CTL* Formeln überprüfen zu können. Dieser Schritt ist notwendig, da sowohl CTL als auch LTL nur Teillogiken der CTL* darstellen und einzeln nicht über deren Ausdrucksmächtigkeit verfügen. In Abschnitt 3 werden geordnete binäre Entscheidungsdiagramme (OBDDs) betrachtet. Diese bieten eine gute Möglichkeit, reaktive Systeme mit endlich vielen Zuständen symbolisch darzustellen [3, 4]. Insbesondere lassen sich Kripke Strukturen mittels OBDDs kodieren um prägnante Darstellungen synchroner und asynchroner Systeme zu erhalten. Im Kapitel über symbolisches Model Checking wird die OBDD Darstellung von Kripke Strukturen benutzt, um einen effizienten Model Checking Algorithmus anzugeben. Da OBDDs Mengen von Zuständen und Übergängen repräsentieren, wird in Abschnitt 4.1 eine Fixpunktcharakterisierung der Operatoren der temporalen Logik vorgestellt, die es uns ermöglicht die Zustände die eine CTL Formel erfüllen als grösste oder kleinste Fixpunkte einer entsprechenden Funktion darzustellen. In Abschnitt 4.2 wird ein CTL Model Checking Algorithmus vorgestellt, der nur Standard OBDD Operationen verwendet. Dieser Algorithmus wird in Abschnitt 4.3 so erweitert, das auch Fairness Constraints beachtet werden können. Eine weitere wichtige Eigenschaft des CTL Model Checking Algorithmus — die Fähigkeit Gegenbeispiele und Zeugen finden zu können — wird in 4.4 vorgestellt. In Abschnitt 4.5 werden Möglichkeiten vorgestellt, den Algorithmus zu verbessern. Abschnitt 4.6 zeigt schliesslich, wie symbolische Techniken auch beim LTL Model Checking verwendet werden können.

2 CTL* Model Checking

Kombiniert man nun die beiden Teillogiken CTL und LTL wieder zur CTL*, so würde man eigentlich ein Ansteigen der Komplexität erwarten. Überraschenderweise ist dies nicht der Fall, es konnte gezeigt werden ([1, 7]), dass das Model Checking Problem für CTL* im wesentlichen die gleiche Komplexität hat wie für LTL.

Die Grundidee des CTL* Model Checking ist es, die Techniken der CTL und LTL zu kombinieren. Der Algorithmus für das LTL Model Checking kann Formeln der Art $\mathbf{E}f$ mit eingeschränkten Pfadformeln f bearbeiten, in denen die einzigen Zustandsteilformeln atomare Propositionen sind. Dieser Algorithmus wird nun so abgewandelt, dass beliebige Zustandsteilformeln bearbeitet werden können.
Angenommen die Zustandsteilformeln von f wurden bereits abgearbeitet und die Zustandsmarkierungen entsprechend aktualisiert. Jede Zustandsteilformel wird nun durch eine neue (bisher nicht verwendete) atomare Proposition sowohl in der Markierung des Modells als auch in der Formel ersetzt. Sei nun $\mathbf{E}f'$ diese neu entstandene Formel. Ist sie eine reine CTL Formel, so wird der CTL Model Checking Algorithmus angewendet, ist f' eine reine LTL Formel so wird der Algorithmus für LTL Model Checking benutzt. In beiden Fällen wird die Formel zu den Markierungen derjenigen Zustände hinzugefügt, die sie erfüllen. Ist $\mathbf{E}f'$ eine Teilformel einer komplexeren CTL* Formel, so wird die Prozedur wiederholt bis die komplette Formel abgearbeitet ist.
Ähnlich wie der CTL Algorithmus arbeitet auch der CTL* Algorithmus in verschiedenen Stufen, so dass in Stufe i Formeln der Ebene i bearbeitet werden. Sei nun f eine CTL* Formel. Die Zustandsteilformeln der Ebene i sind induktiv folgendermassen definiert:

- Ebene 0 enthält alle atomaren Propositionen
- Ebene $(i+1)$ enthält alle Zustandsteilformeln g, so dass alle Zustandsteilformeln von g aus der Ebene i oder niedriger sind und g nicht in einer niedrigeren Ebene enthalten ist.

Sei g eine CTL* Formel. Dann heisst eine Teilformel $\mathbf{E}h_1$ von g genau dann maximal, wenn $\mathbf{E}h_1$ keine "strikte" Teilformel einer anderen "strikten" Teilformel $\mathbf{E}h$ von g ist. Betrachten wir beispielsweise die Formel $\mathbf{E}(a \vee \mathbf{E}(b \wedge \mathbf{EF}c))$. Dann ist $\mathbf{EF}c$ eine maximale Teilformel von $\mathbf{E}(b \wedge \mathbf{EF}c)$ aber nicht von $\mathbf{E}(a \vee \mathbf{E}(b \wedge \mathbf{EF}c))$.
Sei $M = (S, R, L)$ eine Kripke Struktur, f eine CTL* Formel und g eine Zustandsteilformel von f aus der Ebene i. Wir nehmen an die Zustände von M seinen schon mit den Zustandsteilformeln der Ebenen $< i$ markiert worden. Auf der Stufe i des CTL* Algorithmus wird g zur Markierung aller Zustände hinzugefügt, die g wahr machen. Je nach Art der Formel g werden mehrere Fälle betrachtet:

- Ist g eine atomare Proposition, dann ist g genau dann in *label(s)*, wenn es in $L(s)$ ist.
- Ist $g = \neg g_1$, so wird g zu *label(s)* hinzugefügt, wenn g_1 nicht in *label(s)* ist.
- Ist $g = g_1 \vee g_2$, so wird es zu *label(s)* hinzugefügt, genau dann, wenn entweder g_1 oder g_2 in *label(s)* enthalten sind.
- Ist $g = \mathbf{E}g_1$, so wird der Algorithmus *CheckE*(g) (siehe Abb. 1) auf g angewendet, um alle Zustände zu markieren, die die Formel erfüllen. In diesem Algorithmus werden mit $\mathbf{E}h_1, \ldots, \mathbf{E}h_k$ die maximalen Teilformeln von g und mit $a_1, \ldots, a_k$ die neuen atomaren Propositionen bezeichnet. Die Formel g' im Algorithmus entsteht durch ersetzen jeder Teilformel $\mathbf{E}h_i$ durch a_i.
  Die resultierende Formel ist $\mathbf{E}g'_1$ wobei es sich bei g'_1 um eine reine LTL Pfadformel handelt. Hierbei nehmen wir an, dass der LTL Model Checker

```
procedure CheckE(g)
    if g ist eine CTL Formel then
        wende CTL Model Checking auf g an;
        return;
    end if;
    g' = g[a_1/Eh_1, ··· , a_k/Eh_k];
    for all s ∈ S
        for i = 1, ··· , k do
            if Eh_i ∈ label(s) then label(s) := label(s) ∪ {a_i};
    end for all;
    wende LTL Model Checking auf g' an;
    for all s ∈ S do
        if g' ∈ label(s)then label(s) := label(s) ∪ {a_i};
    end for all;
end procedure;
```

Abb. 1. Algorithmus CheckE

label(s) so aktualisiert, dass $M, s \models g'$ genau dann gilt, wenn $label(s) := label(s) \cup \{g'\}$.

Beispiel 1 (CTL Model Checking).* Zur Demonstration der Funktionsweise des CTL* Model Checking verwenden wir wiederum das Mikrowellenbeispiel. Die CTL* Formel

$$\mathbf{AG}((\neg Close \wedge Start) \rightarrow \mathbf{A}(\mathbf{G}\neg Heat \vee \mathbf{F}\neg Error))$$

sorgt dafür, dass bei einer illegalen Folge von Schritten die Mikrowelle entweder nie heizt oder in den Grundzustand zurückgesetzt wird. Die illegale Schrittfolge wird durch $(\neg Close \wedge Start)$ dargestellt, was bedeutet, dass bei offener Tür der Startknopf gedrückt wurde. Das Resultat des Zurücksetzens ist $\neg Error$. Diese Eigenschaft lässt sich nicht mittels CTL ausdrücken.
Um das Model Checking zu vereinfachen, schreiben wir die Formel so um, dass nur noch Existenzquantoren vorkommen:

$$\neg\mathbf{EF}(\neg Close \wedge Start \wedge \mathbf{E}(\mathbf{F}Heat \wedge \mathbf{G}Error))$$

Die Teilformeln der verschiedenen Ebenen sind:

Ebene 0: *Close, Start, Heat* und *Error*
Ebene 1: $\mathbf{E}(\mathbf{F}Heat \wedge \mathbf{G}Error)$ und $\neg Close$
Ebene 2: $\mathbf{EF}(\neg Close \wedge Start \wedge \mathbf{E}(\mathbf{F}Heat \wedge \mathbf{G}Error))$
Ebene 3: enthält die gesamte Formel.

Auf Ebene 0 werden die atomaren Propositionen abgearbeitet. Auf Ebene 1 wird zuerst die Formel $\neg Close$ zu den Markierungen der Zustände 1 und 2 hinzugefügt, die zweite Formel $\mathbf{E}(\mathbf{F}Heat \wedge \mathbf{G}Error)$ ist eine reine LTL Formel

und wird durch eine LTL Model Checking Prozedur bearbeitet. Da diese Formel von keinem Zustand erfüllt wird, wird sie zu keiner Zustandsmarkierung hinzugefügt. Auf Ebene 2 wird die Formel $\mathbf{E}(\mathbf{F}Heat \wedge \mathbf{G}Error)$ nun durch die atomare Proposition a ersetzt. Daraufhin wird die neu erhaltene reine LTL Formel $\mathbf{EF}(\neg Close \wedge Start \wedge a)$ wieder mittels LTL Model Checking verarbeitet. Da kein Zustand mit dieser Formel markiert wird, werden auf Ebene 3 alle Zustände mit

$$\neg\mathbf{EF}(\neg Close \wedge Start \wedge \mathbf{E}(\mathbf{F}Heat \wedge \mathbf{G}Error))$$

markiert. Daraus ist ersichtlich, dass diese Eigenschaft für unsere Mikrowelle immer gilt.

Die Komplexität des CTL* Model Checking Algorithmus hängt von der Komplexität der verwendeten CTL und LTL Model Checking Algorithmen ab. Wie bereits angemerkt, ist die Komplexität des CTL Model Checkings linear in der Grösse der Struktur M und der Formel f. Die beste bisher bekannte Zeitomplexität für LTL Model Checking ist $|M| \cdot 2^{O(|f|)}$. Daraus folgt:

Theorem 1. *Es existiert ein CTL* Model Checking Algorithmus mit Komplexität* $|M| \cdot 2^{O(|f|)}$.

3 Binary Decision Diagrams (BDDs)

3.1 Repräsentation von Kripke Strukturen

Sei Q eine n-stellige Relation über $\{0, 1\}$. So lässt sich Q als OBDD seiner *charakteristischen Funktion* darstellen:

$$f_Q(x_1, \cdots, x_n) = 1 \Leftrightarrow Q(x_1, \cdots, x_n)$$

Allgemeiner: Sei Q eine n-stellige Relation über einen endlichen Definitionsbereich D. Ohne Beschränkung der Allgemeinheit können wir annehmen, dass D 2^m Elemente besitzt, für ein $m > 1$. Um nun Q als OBDD darzustellen, werden die Elemente von D mittels einer Bijektion $\phi : \{0, 1\}^m \rightarrow D$ kodiert, die einen booleschen Vektor der Länge m einem Element von D zuordnet. Mit Hilfe der Kodierung ϕ konstruieren wir eine boolesche Relation $\hat{Q}$ der Stelligkeit $m \times n$ nach folgender Regel:

$$\hat{Q}(\bar{x_1}, \cdots, \bar{x_n}) = Q(\phi(\bar{x_1}, \cdots, \bar{x_n})),$$

wobei $\bar{x_i}$ einen Vektor aus m booleschen Variablen darstellt, der die Variable x_i kodiert. Q lässt sich nun als OBDD darstellen, das durch die charakteristische Funktion $f_{\hat{Q}}$ von $\hat{Q}$ bestimmt wird. Da auch Mengen als 1-stellige Relationen angesehen werden können ist es möglich, mit der gleichen Methode Mengen als OBDDs darzustellen.
Betrachten wir nun die Kripke Struktur $M = (S, R, L)$. Um diese Struktur zu repräsentieren, müssen wir die Menge S, die Relation R und die Abbildung L darstellen können.

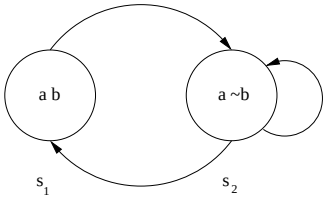


Abb. 2. Kripke Struktur mit 2 Zuständen

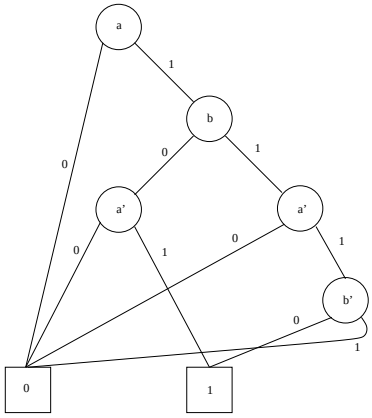


Abb. 3. OBDD der Kripke Struktur

Für die Menge S müssen zuerst die Zustände kodiert werden. Der Einfachheit halber nehmen wir an, dass es genau 2^m Zustände gibt. Wie oben beschrieben sei $\phi : \{0,1\} \rightarrow S$ die Funktion, die boolesche Vektoren auf Zustände abbildet. Da jede Zuweisung die Kodierung eines Zustandes aus S ist, stellt die charakteristische Funktion von S das gesuchte OBDD dar.
Für die Übergangsrelation R wird die gleiche Kodierung der Zustände benutzt. In diesem Fall benötigen wir zwei Mengen boolescher Variablen, eine für den Startzustand und eine für den Endzustand eines Übergangs. Wird die Übergangsrelation mittels $\hat{R}(\bar{x}, \bar{x'})$ kodiert, so wird R durch die charakteristische Funktion $f_{\hat{R}}$ repräsentiert.
Schliesslich betrachten wir die Abbildung L. Obwohl L als Abbildung von Zuständen auf Teilmengen atomarer Propositionen definiert ist, ist es in diesem Fall einleuchtender, L als Abbildung atomarer Propositionen auf Teilmengen von Zuständen anzusehen. Die atomare Proposition p wird auf diejenigen Zustände abgebildet, die sie erfüllen: $\{s|p \in L(s)\}$. Diese Menge von Zuständen, L_p, kann mit der gleichen Kodierung ϕ repräsentiert werden. Auf diesem Weg lassen sich alle atomaren Propositionen separat darstellen.
Will man zusätzlich noch mögliche Mengen von Initialzuständen einer fairen Kripke Struktur beschreiben, so wird die Menge der Initialzustände ebenso dargestellt wie jede andere Menge. Für die Fairness Constraints $F = \{P_1, \cdots, P_n\}$ repräsentiert man einfach jedes der P_i separat.

Betrachten wir nun die Figur in Abb. 2. In diesem Fall gibt es zwei Zustandsvariablen a und b. Wir führen zwei zusätzliche Variablen a' und b' ein, um Nachfolgezustände kodieren zu können. Auf diese Weise repräsentieren wir den Übergang von s_1 nach s_2 durch die Konjunktion

$$(a \wedge b \wedge a' \wedge \neg b').$$

Die boolesche Formel der gesamten Übergangsfunktion ist

$$(a \wedge b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge b').$$

Diese Formel enthält drei Disjunktionen, da die Kripke Struktur drei Übergänge enthält. Aus dieser Formel wird nun das OBDD aus Abb. 3 erstellt, um eine prägnante Darstellung der Übergangsrelation zu erhalten.

In vielen Fällen ist es nicht sinnvoll, eine explizite Repräsentation einer Kripke Struktur zu erstellen, da diese Struktur zu gross wäre. Deshalb werden in der Praxis die OBDDs direkt aus einer High-Level Beschreibung des Systems gewonnen. Im Kapitel über die Modellierung von Systemen wird die Prozedur angegeben, mit der Systeme in Formeln konvertiert werden können. Wird nun der Definitionsbereich wie oben beschrieben kodiert, so kann diese Prozedur zur Konstruktion eines OBDDs für die Übergangsrelation direkt aus einer High-Level Beschreibung des Systems verwendet werden.

4 Symbolisches Model Checking

4.1 Fixpunkt Repräsentationen

Sei wieder $M = (S, R, L)$ eine beliebige Kripke Struktur. Die Menge $P(S)$ sei die Menge aller Teilmengen von S. Jedes Element $S' \in P(S)$ kann nun als ein Prädikat von S betrachtet werden, das genau für die Zustände von S' wahr ist. Das kleinste Element von $P(S)$ ist die leere Menge, die wir mit *falsch* bezeichnen. Das grösste Element ist S selbst, das wir mit *wahr* bezeichnen.

Definition 1 (predicate transformer).
Eine Funktion, die $P(S)$ auf $P(S)$ abbildet, heisst predicate transfomer. *Sei $\tau : P(S) \to P(S)$ eine solche Funktion. Dann gilt:*

1. *τ ist* monoton *wenn $P \subseteq Q$ auch $\tau(P) \subseteq \tau(Q)$ impliziert.*
2. *τ ist $\cup$-kontinuierlich wenn $P_1 \subseteq P_2 \subseteq \cdots$ impliziert $\tau(\cup_i P_i) = \cup_i \tau(P_i)$.*
3. *τ ist $\cap$-kontinuierlich wenn $P_1 \supseteq P_2 \supseteq \cdots$ impliziert $\tau(\cap_i P_i) = \cap_i \tau(P_i)$.*

Wir werden $\tau^i(Z)$ schreiben, um i Anwendungen von τ auf Z anzuzeigen. Ein monotoner predicate transformer *τ auf $P(S)$ besitzt immer einen kleinsten Fixpunkt $\mu Z.\tau(Z)$, und einen grössten Fixpunkt $\nu Z.\tau(Z)$ (siehe Tarski [5]), wobei gilt:*

- *$\mu Z.\tau(Z) = \cap\{Z | \tau(Z) \subseteq Z\}$, wenn τ monoton ist und $\mu Z.\tau(Z) = \cup_i \tau^i(false)$, wenn τ zusätzlich noch $\cup$-kontinuierlich ist,*
- *$\nu Z.\tau(Z) = \cup\{Z | \tau(Z) \subseteq Z\}$, wenn τ monoton ist und $\nu Z.\tau(Z) = \cap_i \tau^i(true)$, wenn τ zusätzlich noch $\cap$-kontinuierlich ist.*

Folgende Lemmata sind im Umgang mit *predicate transformers* auf endlichen Kripke Strukturen von nutzen [6, 7].

Lemma 1. *Ist S endlich und τ monoton, dann ist τ auch $\cup$-kontinuierlich und $\cap$-kontinuierlich.*

Beweis. Sei $P_1 \subseteq P_2 \subseteq \cdots$ eine Folge von Teilmengen von S. Da S endlich ist, existiert ein j_0, so dass für jedes $j \geq j_0$ gilt $P_j = P_{j_0}$ und für jedes $j < j_0$ gilt $P_j \subseteq P_{j_0}$. Folglich ist $\cup_i P_i = P_{j_0}$ und als Ergebnis $\tau(\cup_i P_i) = \tau(P_{j_0})$.
Andererseits folgt aus der Tatsache das τ monoton ist, dass $\tau(P_1) \subseteq \tau(P_2) \cdots$. Deshalb ist für jedes $j < j_0, \tau(P_j) \subseteq \tau(P_{j_0})$ und für alle $j \geq j_0$ ist $\tau(P_j) = \tau(P_{j_0})$. Daraus folgt $\cup_i \tau(P_i) = \tau(P_{j_0})$ und τ ist $\cup$-kontinuierlich. Der Beweis für die $\cap$-kontinuität verläuft analog.

Lemma 2. *Ist τ monoton, dann gilt für alle i : $\tau^i(false) \subseteq \tau^{i+1}(false)$ und $\tau^i(true) \supseteq \tau^{i+1}(true)$.*

Lemma 3. *Ist τ monoton und S endlich, dann existiert eine Zahl i_0, so dass für alle $j \geq i_0$ gilt: $\tau^j(false) = \tau^{i_0}(false)$. Ausserdem existiert ein j_0, so dass für alle $j \geq j_0$ gilt: $\tau^j(true) = \tau^{j_0}(true)$.*

```
function Lfp ( τ : Predicate Transformer ) : Predicate
    Q := False;
    Q':= τ ( Q );
    while (Q ≠ Q') do
        Q := Q';
        Q':= τ ( Q');
    end while;
    return ( Q );
end function
```

Abb. 4. Algorithmus Lfp

```
function Gfp ( τ : Predicate Transformer ) : Predicate
    Q := True;
    Q':= τ ( Q );
    while (Q ≠ Q') do
        Q := Q';
        Q':= τ ( Q');
    end while;
    return ( Q );
end function
```

Abb. 5. Algorithmus Gfp

Lemma 4. *Ist τ monoton und S endlich, dann existiert eine Zahl i_0, so dass $\mu Z.\tau(Z) = \tau^{i_0}(false)$. Ebenfalls gibt es ein j_0 mit $\nu Z.\tau(Z) = \tau^{j_0}(true)$.*

Aufgrund dieser Lemmata lassen sich nun für ein monotones τ grösster und kleinster Fixpunkt durch die Algorithmen in den Abbildungen 4 und 5 berechnen.

Indem man jede CTL Formel f mit ihrem Prädikat $\{s|M, s \models f\}$ in $P(S)$ identifiziert, lassen sich alle grundlegenden CTL Operatoren durch den grössten oder kleinsten Fixpunkt eines passenden *predicate transformers* charakterisieren [9]:

- $\mathbf{AF}f_1 = \mu Z.f_1 \vee \mathbf{AX}Z$
- $\mathbf{EF}f_1 = \mu Z.f_1 \vee \mathbf{EX}Z$
- $\mathbf{AG}f_1 = \nu Z.f_1 \wedge \mathbf{AX}Z$
- $\mathbf{EG}f_1 = \nu Z.f_1 \wedge \mathbf{EX}Z$
- $\mathbf{A}[f_1\mathbf{U}f_2] = \mu Z.f_2 \vee (f_1 \wedge \mathbf{AX}Z)$
- $\mathbf{E}[f_1\mathbf{U}f_2] = \mu Z.f_2 \vee (f_1 \wedge \mathbf{EX}Z)$
- $\mathbf{A}[f_1\mathbf{R}f_2] = \nu Z.f_2 \wedge (f_1 \vee \mathbf{AX}Z)$
- $\mathbf{E}[f_1\mathbf{R}f_2] = \nu Z.f_2 \wedge (f_1 \vee \mathbf{EX}Z)$

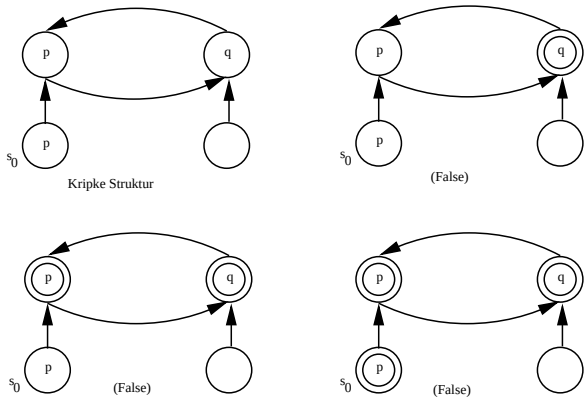


Abb. 6. Folge von Näherungen für $\mathbf{E}[p\mathbf{U}q]$ mittels der Funktion Lfp

Intuitiv kann man also sagen, dass sich kleinste Fixpunkte auf Möglichkeiten beziehen, während sich grösste Fixpunkte auf Eigenschaften beziehen, die immer gelten. Die Grafik in Abb. 6 zeigt, wie die Menge der Zustände, die $\mathbf{E}[p\mathbf{U}q]$ erfüllen, mittels der Prozedur Lfp berechnet werden kann.

Die Zustände, die die gegenwärtige Approximation von $\mathbf{E}[p\mathbf{U}q]$ erzeugen, sind doppelt eingekreist. Man sieht das $\tau^3(false) = \tau^4(false)$ gilt, deshalb ist $\mathbf{E}[p\mathbf{U}q] = \tau^3(false)$, und da s_0 in $\tau^3(false)$ liegt gilt $M, s_0 \models \mathbf{E}[p\mathbf{U}q]$.

4.2 Symbolisches CTL Model Checking

In diesem Abschnitt wird ein symbolischer Model Checking Algorithmus beschrieben, der auf Kripke Strukturen arbeitet. Die Kripke Strukturen werden in diesem Zusammenhang, wie im Abschnitt 3 erläutert, symbolisch mittels OBDDs beschrieben. Um den symbolischen Algorithmus nun in geeigneter Form schreiben zu können, benötigen wir eine prägnantere Form der Schreibweise von Operationen auf booleschen Formeln. Zu diesem Zweck werden wir *quantifizierte boolesche Formeln (quantified boolean formulas QBF)* [10, 11] benutzen.

Quantifizierte boolesche Formeln Sei $V = \{v_0, v_1, \ldots, v_{n-1}\}$ eine Menge von Variablen, QBF(V) ist die kleinste Menge von Formeln, so dass gilt:

- Jede Variable in V ist eine Formel.
- Sind f und g Formeln, so sind auch $\neg f, f \vee g$ und $f \wedge g$ Formeln.
- Ist f eine Formel und $v \in V$, dann sind auch $\exists v f$ und $\forall v f$ Formeln.

Eine Wertzuweisung für QBF(V) ist eine Funktion $\sigma : V \to \{0, 1\}$. Ist $a \in \{0, 1\}$, so benutzen wir die Schreibweise $\sigma < v \leftarrow a >$ mit

$$\sigma < v \leftarrow a > (w) = \begin{cases} a \text{ if } v = w \\ \sigma(w) \text{ sonst} \end{cases}$$

Ist f eine Formel aus QBF(V) und ist σ eine Wertzuweisung so schreiben wir $\sigma \models f$ um anzuzeigen das f unter der Zuweisung σ wahr ist. Die Funktion $\models$ ist wie bisher induktiv definiert mit den zwei Erweiterungen:

- $\sigma \models \exists$ gilt genau dann wenn $\sigma < v \leftarrow 0 > \models f$ oder $\sigma < v \leftarrow 1 > \models f$ gilt.
- $\sigma \models \forall$ gilt genau dann wenn $\sigma < v \leftarrow 0 > \models f$ und $\sigma < v \leftarrow 1 > \models f$ gilt.+

Quantoren werden am häufigsten bei relationalen Produktoperationen folgender Form benutzt:

$$\exists \bar{v}[f(\bar{v}, \bar{w}) \wedge g(\bar{v}, \bar{x})]$$

Der Algorithmus für symbolisches Model Checking Dieser Algorithmus wird in einer Prozedur *Check* implementiert, der eine CTL Formel als Argument übergeben wird und die ein OBDD zurückliefert, das exakt diejenigen Zustände des Systems repräsentiert, die die Formel erfüllen.
Check wird induktiv über den Aufbau von CTL Formeln definiert:

- Ist f eine atomare Proposition a, so ist $Check(f)$ ein OBDD, das die Menge der Zustände repräsentiert, die a erfüllen.
- Ist $f = f_1 \wedge f_2$ oder $f = \neg f_1$, so erhalten wir $Check(f)$ durch Anwenden der Funktion *Apply* (siehe Kap. über OBDDs) auf die beiden Argumente $Check(f_1)$ und $Check(f_2)$.
- Check(**EX** f) = CheckEX($Check(f)$),
- Check(**E**[f **U** g]) = CheckEU($Check(f), Check(g)$) und
- Check(**EG** f) = CheckEG($Check(f)$).

Man beachte, dass die Teilprozeduren CheckEX, CheckEU und CheckEG im Gegensatz zu *Check* OBDDs als Argumente erhalten:

- Die Prozedur CheckEX erkennt die Formel $\mathbf{EX}f$ in einem Zustand als wahr, wenn f einen Folgezustand hat, in dem f wahr ist. Es gilt also:

  $$CheckEX(f(\bar{v})) = \exists \bar{v}'[f(\bar{v'}) \wedge R(\bar{v}, \bar{v'})],$$

  wobei $R(\bar{v}, \bar{v'})$ die OBDD Darstellung der Übergangsrelation ist. Sind die OBDD Repräsentationen von f und R vorhanden, so lässt sich das OBDD von

  $$\exists \bar{v}'[f(\bar{v'}) \wedge R(\bar{v}, \bar{v'})]$$

  mittels der QBF Operationen berechnen.
- Die Prozedur CheckEU basiert auf der Charakterisierung des CTL Operators **EU** durch den kleinsten Fixpunkt:

  $$\mathbf{E}[f_1 \mathbf{U} f_2] = \mu Z.f_2 \vee (f_1 \wedge \mathbf{EX}Z)$$

  Wir benutzen also die Funktion Lfp, um eine Folge von Näherungen $Q_0, Q_1, \ldots, Q_i, \ldots$ zu berechnen, die in einer endlichen Anzahl von Schritten gegen

$\mathbf{E}[f\mathbf{U}g]$ konvergieren. Sind die OBDDs von f, g und der aktuellen Approximation Q_i schon vorhanden, so lässt sich daraus das OBDD der nächsten Approximation Q_{i+1} berechnen. Da OBDDs eine kanonische Form der Darstellung boolescher Funktionen sind, lässt sich die Konvergenz durch vergleichen benachbarter Approximationen leicht überprüfen. Sobald $Q_i = Q_{i+1}$ gilt, terminiert die Funktion Lfp. Die Menge der Zustände, die mit $\mathbf{E}[f\mathbf{U}g]$ übereinstimmen, wird durch das OBDD von Q_i repräsentiert.

– Die Prozedur CheckEG läuft ähnlich ab wie CheckEU mit dem Unterschied, das die Charakterisierung des CTL Operators **EG** mit dem grössten Fixpunkt verwendet wird. Also:

$$\mathbf{EG}f_1 = \nu Z.f_1 \wedge \mathbf{EX}Z.$$

Ist das OBDD von f vorhanden, so kann die Funktion Gfp benutzt werden, um ein OBDD zu berechnen, das die Menge der Zustände repräsentiert, die $\mathbf{EG}f$ erfüllen.

4.3 Fairnesseinschränkungen beim symbolischen Model Checking

Um nun auch beim symbolischen Model Checking Einschränkungen durch Fairness Constraints berücksichtigen zu können, wird eine neue Funktion CheckFair benötigt, die CTL Formeln im Hinblick auf Fairness Constraints überprüfen kann. Um dies zu erreichen, werden die bei *Check* benutzten Teilprozeduren zu CheckFairEX, CheckFairEU und CheckFairEG erweitert.
Angenommen die Fairness Constraints liegen als Menge von CTL Formeln $F = \{P_1, \cdots, P_n\}$ vor. Betrachten wir nun die Formel $\mathbf{EG}f$ mit den gegebenen Fairness Constraints F. Die Formel sagt aus, dass ein im gegenwärtigen Zustand beginnender Pfad existiert auf dem f immer gilt und jede Formel aus F unendlich oft entlang des Pfades gilt. Die Menge solcher Zustände Z ist die grösste Menge mit den folgenden Eigenschaften:

1. Alle Zustände in Z erfüllen f, und
2. für alle Fairness Constraints $P_k \in F$ und alle Zustände $s \in Z$ gibt es eine Zustandsfolge der Länge ≥ 1 von s in einen Zustand aus Z, die P_k erfüllt und in der alle Zustände f erfüllen.

Diese Charakterisierung ist etwas anders gewählt als die im expliziten Fall, ist aber für das symbolische Model Checking passender, da sie mit Hilfe eines Fixpunkts ausgedrückt werden kann:

$$\mathbf{EG}f = \nu Z.f \wedge \bigwedge_{k=1}^{n} \mathbf{EXE}[f\mathbf{U}(Z \wedge P_k)]$$

Man beachte, dass die Formel sowohl CTL als auch Fixpunktoperatoren enthält (es ist möglich, zu zeigen, dass diese Formel nicht direkt in CTL ausdgedrückt werden kann).

Aus der Fixpunktdarstellung folgt, dass die Menge der Zustände, die $\mathbf{EG}f$ unter den gegebenen Fairness Constraints $F = \{P_1, \cdots, P_k\}$ erfüllen, durch die Funktion $CheckFairEG(f(\bar{v}))$ gemäss folgender Fixpunktcharakterisierung berechnet werden:

$$\nu Z(\bar{v}).f(\bar{v}) \wedge \bigwedge_{k=1}^{n} \mathbf{EX}(\mathbf{EU}(f(\bar{v}), Z(\bar{v}) \wedge P_k))$$

Der Hauptunterschied zur bisher verwendeten Fixpunktauswertung ist, dass in diesem Fall jedes mal wenn der Ausdruck ausgewertet wird, innerhalb von CheckEU mehrere verschachtelte Fixpunktberechnungen ausgeführt werden.
Das Überprüfen von $\mathbf{EX}f$ und $\mathbf{E}[f\mathbf{U}g]$ unter gegebenen Fairness Constraints verläuft gleich wie im expliziten Fall. Die Menge aller Zustände, die am Anfang eines fairen Berechnungpfads stehen, ist:

$$fair(\bar{v}) = CheckFair(\mathbf{EG}True).$$

Die Formel $\mathbf{EX}f$ ist unter gegebenen Fairness Constraints in einem Zustand s genau dann wahr, wenn es einen Folgezustand s' gibt, so dass s' f erfüllt und s' der Anfang eines fairen Berechnungspfades ist. Daraus folgt, das die Formel $\mathbf{EX}f$ (unter den gegebenen Fairness Constraints) äquivalent zur Formel $\mathbf{EX}(f \wedge fair)$ (ohne Fairness Constraints) ist.Deshalb definieren wir:

$$CheckFairEX(f(\bar{v})) = CheckEX(f(\bar{v}) \wedge fair(\bar{v})).$$

In gleicher Weise ist die Formel $\mathbf{E}[f\mathbf{U}g]$ (unter den gegebenen Fairness Constraints) äquivalent zur Formel $\mathbf{E}[f\mathbf{U}(g \wedge fair)]$ (ohne Fairness Constraints). Deshalb definieren wir:

$$CheckFairEU(f(\bar{v}), g(\bar{v})) = CheckEU(f(\bar{v}), g(\bar{v}) \wedge fair(\bar{v})).$$

4.4 Gegenbeispiele und Zeugen

Eine der wichtigsten Eigenschaften des CTL Model Checking Algorithmus ist die Fähigkeit, *Gegenbeispiele* und *Zeugen* finden zu können. Ist dieses Feature aktiviert und der Model Checker findet heraus, dass eine Formel mit universellem Pfadquantor falsch ist, so wird er ebenfalls einen Berechnungspfad finden, der zeigt, dass die Negation der Formel wahr ist. Ebenso wird für eine Formel mit existenziellem Pfadquantor, die wahr ist, auch ein Berechnungspfad gefunden, der zeigt, warum die Formel wahr ist. Da ein Gegenbeispiel einer universell quantifizierten Formel auch ein Zeuge für die dazu duale existenziell quantifizierte Formel ist, reicht es aus wenn wir in der Lage sind, für die drei grundlegenden CTL-Operatoren **EX**, **EG** und **EU** Zeugen finden zu können.
Um den Vorgang näher zu erläutern, betrachten wir wieder *strongly connected components* (SCC) im Übergangsgraphen der Kripke Struktur. Mittels dieser *strongly connected components* wird ein neuer Graph gebildet, in dem es genau dann einen Übergang von einer SCC zu einer anderen gibt, wenn es einen Übergang von einem Zustand der einen in einen Zustand der anderen SCC gibt. Man

erkennt, dass der neue Graph keine echten Zyklen enthält, d.h. jeder Zykel ist in einer der SCCs enthalten und da nur endliche Kripke Strukturen betrachtet werden, muss auch jeder unendlich lange Pfad ein Suffix haben, das komplett in einer der SCCs des Übergangsgraphen liegt.
Wir beginnen mit dem Problem, einen Zeugen für die Formel $\mathbf{EG}f$ unter den gegebenen Fairness Constraints $F = \{P_1, \cdots, P_n\}$ zu finden. Jedes P_i wird mit der Menge derjenigen Zustände identifiziert, die es wahr machen. Aus dem vorangehenden Abschnitt wissen wir, dass die Menge der Zustände die $\mathbf{EG}f$ unter den gegebenen Fairness Constraints erfüllen, durch die Formel

$$\mathbf{EG}f = \nu Z.f \wedge \bigwedge_{k=1}^{n} \mathbf{EXE}[f\mathbf{U}(Z \wedge P_k)] \tag{1}$$

gegeben ist.
Sei nun s ein Zustand in $\mathbf{EX}f$. Gesucht ist ein Pfad π, der in s beginnt, f in jedem Zustand erfüllt und jede Menge $P \in F$ unendlich oft besucht. Einen solchen Pfad können wir immer finden, er besteht aus einem endlichen Präfix, dem eine unendliche Schleife (*cycle*) folgt. Wir konstruieren diesen Pfad, indem wir eine Folge von Präfixen des Pfades angeben, die solange verlängert wird bis eine Schleife gefunden wird. Dabei müssen wir in jedem Schritt darauf achten, dass das aktuelle Präfix zu einem fairen Pfad erweitert werden kann, in dem jeder Zustand f erfüllt. Diese Invariante wird dadurch garantiert, dass wir bei jedem Zustand den wir zum Präfix hinzufügen darauf achten, dass er $\mathbf{EG}f$ erfüllt.
Zuerst wird dazu die oben gegebene Fixpunktformel ausgewertet. Bei jeder Iteration der äusseren Fixpunktberechnung wird auch eine Sammlung von kleinsten Fixpunkten berechnet, die zu den Formeln $\mathbf{E}[f\mathbf{U}(Z \wedge P_1)]$ bis $\mathbf{E}[f\mathbf{U}(Z \wedge P_n)]$ gehört. Für jeden Fairness Constraint P erhalten wir eine wachsende Folge von Näherungen $Q_0^P \subseteq Q_1^P \subseteq Q_2^P \subseteq \ldots$, wobei Q_i^P die Menge derjenigen Zustände ist, von denen aus ein Zustand in $Z \wedge P$ in i oder weniger Schritten erreicht werden kann, wobei f erfüllt wird. Im letzten Schritt der Iteration des äusseren Fixpunkts, wenn $Z = \mathbf{EG}f$ gilt, speichern wir die Folge von Näherungen Q_i^P für jedes $P \in F$.
Angenommen s sei ein bekannter Initialzustand der $\mathbf{EG}f$ erfüllt, dann gehört s zur Menge der Zustände, die in Gleichung (1) berechnet werden, und somit muss s einen Nachfolger in $\mathbf{E}[f\mathbf{U}(\mathbf{EG}f \wedge P)]$ (für jedes $P \in F$) haben. Um nun die Länge des *Zeugenpfads* zu minimieren, wählen wird den ersten Fairness Constraint, die von s aus erreicht werden kann. Zusätzlich suchen wir einen Folgezustand t von s in den gespeicherten Mengen Q_0^P. Wird kein solcher Zustand t gefunden, so wird die Suche in den Mengen $Q_1^P, Q_2^P, \ldots$ fortgesetzt. Da s in $\mathbf{EG}f$ liegt, wissen wir, dass irgendwann ein Zustand $t \in Q_i^P$ gefunden wird. Man beachte dabei, dass die Pfadlänge von t zu einem Zustand in $(\mathbf{EG}f) \wedge P$ i ist und deshalb t in $\mathbf{EG}f$ liegt. Ist nun $i \geq 0$, so finden wir einen Nachfolger von t in Q_{i-1}^P, indem wir die die Menge der Nachfolger von t finden, sie mit Q_{i-1}^P schneiden und ein beliebiges Element aus der resultierenden Menge wählen. Wird dies bis $i = 0$ fortgesetzt, so erhalten wir einen Pfad vom Initialzustand s in einen Zustand u aus $(\mathbf{EG}f) \wedge P$. Danach kann P von der Betrachtung ausgeschlossen

werden und die gesamte Prozedur wird mit u wiederholt bis alle Fairness Constraints abgearbeitet wurden.
Sei s' der letzte Zustand des dadurch erhaltenen Pfades. Um nun die Schleife fertigzustellen, benötigen wir einen nichttrivialen Pfad von s' nach t, auf dem jeder Zustand f erfüllt. Anders ausgedrückt brauchen wir nun einen Zeugen für die Formel $\{s'\} \wedge \mathbf{EX}\ \mathbf{E}[f\mathbf{U}\{t\}]$. Ist diese Formel wahr, so haben wir einen Zeugenpfad für s gefunden.
Ist die Formel falsch, so gibt es mehrere mögliche Strategien:

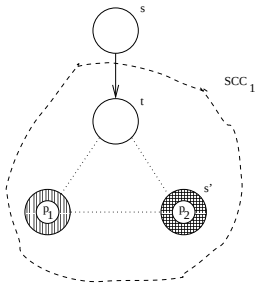


Abb. 7. Der Zeuge liegt in der ersten SCC

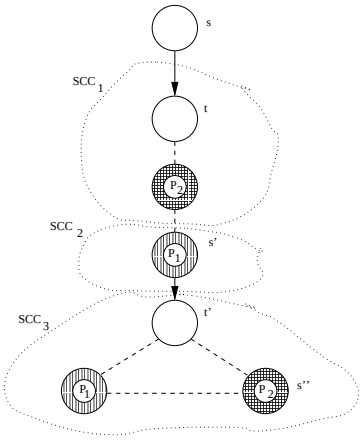


Abb. 8. Der Zeuge spannt sich über 3 SCCs

– Am einfachsten ist es, die Prozedur unter Beachtung sämtlicher Fairness Constraints aus F von s' aus erneut zu starten. Da $\{s'\} \wedge \mathbf{EX}\ \mathbf{E}[f\mathbf{U}\{t\}]$ falsch ist, wissen wir, dass s' nicht in der SCC von f liegt, die t enthält.

Trotzdem liegt s' aber in $\mathbf{EG}f$. Deshalb müssen wir, um diese Strategie weiterzuführen, tiefer in den gerichteten, azyklischen Graphen der SCCs gehen, in dem wir schliesslich entweder eine Schleife finden oder in der letzten SCC von f ankommen, in der auf jeden Fall eine Schleife gefunden wird, da diese SCC nicht mehr verlassen werden kann.

– Eine etwas fortgeschrittenere Möglichkeit wäre es, zuerst $\mathbf{E}[f\mathbf{U}\{t\}]$ zu berechnen. Da wir nach dem ersten Verlassen der Menge wissen, dass die Schleife nicht abgeschlossen wird, beginnen wir von einem anderen Zustand aus neu. Heuristisch werden mit diesem Ansatz häufig kurze Gegenbeispiele gefunden (wahrscheinlich da die Anzahl der SCCs eher klein ist), deshalb wird kein Versuch unternommen, die kleinste Schleife zu finden.

Schliesslich soll noch erklärt werden, wie Zeugen für $\mathbf{E}[f\mathbf{U}g]$ und $\mathbf{EX}f$ gefunden werden können. Erinnern wir uns, dass $fair$ die Menge der Zustände ist, die $\mathbf{EG}True$ unter Beachtung der Fairness Constraints F erfüllen. Damit ist es möglich $\mathbf{E}[f\mathbf{U}g]$ unter Beachtung von F mittels des Standard CTL Model Checking Algorithmus (ohne Fairness Constraints) zu berechnen, indem $\mathbf{E}[f\mathbf{U}(g \wedge fair)]$ berechnet wird. Ebenso wird mit $\mathbf{EX}f$ verfahren, indem $\mathbf{EX}(f \wedge fair)$ berechnet werden.

4.5 Berechnungen des relationalen Produkts

Die meisten Operationen, die beim symbolischen Model Checking Algorithmus benutzt werden, sind linear im Produkt der Grössen der OBDD-Operanden. Die Hauptausnahme ist das relationale Produkt, das benutzt wird, um $\mathbf{EX}h$ zu berechnen:

$$\exists \bar{v}'[h(\bar{v}' \wedge \mathbf{R}(\bar{v}, \bar{v}')]$$

Es wäre zwar möglich, diese Operation mittels einer Konjunktion und einer Folge von existenziellen Quantifikationen zu implementieren, in der Praxis wäre diese Methode jedoch zu langsam. Zusätzlich ist das OBDD von $h(\bar{v}' \wedge \mathbf{R}(\bar{v}, \bar{v}')$ oftmals wesentlich grösser als das OBDD des Endresultats, weshalb die Konstruktion vermieden werden sollte. Aus diesem Grund wird ein spezieller Algorithmus benutzt, um das ODBB des relationalen Produkts in einem Schritt aus den OBDDs von h und R zu berechnen:

Dieser Algorithmus benutzt wie viele OBDD-Algorithmen einen Ergebnis-

Cache. In diesem Fall werden dort Einträge der Form (f, g, E, r) gespeichert, wobei E eine Menge von ausquantifizierten Variablen ist, und f, g und r OBDDs sind. Ein solcher Eintrag im Cache bedeutet, dass ein früherer Aufruf von $RelProd(f, g, E)$ r als Resultat hatte.
Obwohl der angegebene Algorithmus in der Praxis gut funktioniert, hat er im schlechtesten Fall exponentielle Komplexität.Dieser Fall tritt meistens dann auf, wenn das resultierende OBDD exponentiell grösser ist als die OBDDs der Argumente $f(\bar{v})$ und $g(\bar{v})$. In solchen Situationen muss jede Berechnungsmethode exponentielle Laufzeit haben.

```
function Relprod ( f, g : OBDD, E : Variablenmenge) : OBDD
    if f = 0 ∨ g = 0 then
        return 0;
    else if f = 1 ∨ g = 1 then
        return 1;
    else if ( f, g, E, r ) is in the result cache then
        return r;
    else    let x be the top variable of f;
        let y be the top variable of g;
        let z be the topmost of x and y;
        r0 := RelProd f|z←0, g|z←0, E);
        r1 := RelProd f|z←1, g|z←1, E);
        if z ∈ Ethen
            r := Or(r0, r1);
                /* OBDD for r0 ∨ r1 */
        else
            r := IfThenElse(z, r1, r0);
                /* OBDD for (z ∧ r1) ∨ (¬z ∧ r0) */
        end if;
        insert ( f, g, E, r ) in the result cache;
        return r;
    end if;
end function
```

Abb. 9. Algorithmus Relprod

Partitionierte Übergangsrelationen Der im vorigen Abschnitt beschriebene Algorithmus zur Berechnung des relationalen Produkts benötigt $\mathbf{R}(\bar{v}, \bar{v}')$ als sogenannte *monolitische* Übergangsrelation bestehend aus einem einzigen OBDD. Die Konstruktion eines solchen OBDDs wurde bereits im Kapitel über Binary Decision Diagrams beschrieben. Unglücklicherweise ist dieses OBDD in der Praxis oftmals sehr gross. Partitionierte Übergangsrelationen bieten zwar eine sehr viel kompaktere Repräsentation, können aber nicht mit dem Algorithmus für die Berechnung des relationalen Produkts verwendet werden. Erinnern wir uns aber nun, dass die Übergangsfunktionen für synchrone und asynchrone Schaltkreise aus Disjunktionen oder Konjunktionen mehrerer Teile $R_i(\bar{v}, \bar{v}')$ bestehen, die typischerweise als OBDDs mit weniger als hundert Knoten dargestellt werden können, so können wir den Schaltkreis als Liste dieser OBDDs darstellen, die implizit miteinander verknüpft sind. Eine solche Liste heisst *partitionierte Übergangsrelation* [15, 16].
Für synchrone Schaltkreise haben die R_i die Form
$R_i(\bar{v}, \bar{v}') = (v_i' \equiv f_i(\bar{v}))$,
und für asynchrone Schaltkreise
$R_i(\bar{v}, \bar{v}') = (v_i' \equiv f_i(\bar{v})) \wedge \bigwedge_{j \neq i}(v_i' \equiv v_j)$, wobei R im ersten Fall die Konjunktion und im zweiten Fall die Disjunktion der R_i ist. Eine Liste von R_i mit einer impli-

ziten Disjunktion heisst *disjnuktive partitionierte Übergangsrelation*, eine Liste mit impliziter Konjunktion heisst *konjunktive partitionierte Übregangsrelation*. Obwohl eine partitionierte Übergangsrelation mit einem OBDD pro Zustandsvariable oftmals effektiver ist als ein einzelnes OBDD für die gesamte Übergangsrelation, stellt es nicht immer die beste Möglichkeit dar. Solange die OBDDs nicht zu gross werden ist es besser, manche Teile R_i zu einem grösseren OBDD zu verknüpfen. Die Berechnung des relationalen Produkts kann dadurch beschleunigt werden, und wenn die R_i nahe ihrer Wurzel ähnlich sind können auf diese Weise OBDD Knoten eingespart werden. Im folgenden werden die Erweiterungen am Algorithmus beschrieben, die notwendig sind, um das relationale Produkt auch für partitionierte Übergangsrelationen berechnen zu können.

Disjunktive Partitionierung Für eine disjunktiv partitionierte Übergangsrelation ist das zu berechnende relationale Produkt von der Form:

$$\exists \bar{v}'[h(\bar{v}') \wedge R_0(\bar{v}, \bar{v}') \vee \cdots \vee R_{n-1}(\bar{v}, \bar{v}'))]$$

Dieses Produkt kann berechnet werden ohne das komplette OBDD zu konstruieren, indem man den Existenzquantor über die Disjunktionen verteilt:

$$\exists \bar{v}'[h(\bar{v}') \wedge R_0(\bar{v}, \bar{v}')] \vee \cdots \vee \exists \bar{v}'[h(\bar{v}') \wedge R_{n-1}(\bar{v}, \bar{v}')]$$

Aus diesem Grund sind wir in der Lage, das Problem der Berechnung des relationalen Produkts in eine Folge von kleineren Berechnungen zu reduzieren, und deshalb können wesentlich grössere asynchrone Schaltkreise verifiziert werden.

Konjunktive Partitionierung Wenn eine konjunktiv partitionierte Übergangsrelation vorliegt, ist das zu berechnende relationale Produkt von der Form:

$$\exists \bar{v}'[h(\bar{v}') \wedge R_0(\bar{v}, \bar{v}') \wedge \cdots \wedge R_{n-1}(\bar{v}, \bar{v}'))]$$

Die grösste Schwierigkeit bei der Berechnung des relationalen Produkts ohne die Konjunktion zu bilden ist, dass sich die Existenzquantoren nicht über die Formel verteilen. Zur Lösung dieses Problems sind zwei Beobachtungen von Bedeutung [15, 16]:

1. Schaltkreise zeigen normalerweise Lokalittseigenschaften, d.h. viele der R_i hängen nur von einer kleinen Anzahl von Variablen aus $\bar{v}$ und $\bar{v}'$ ab.
2. Obwohl sich die Existenzquantoren nicht über die Konjunktionen verteilen lassen, können Teilformeln herausgenommen werden, wenn sie von keiner der quantifizierten Variablen abhängen.

Aus diesen Tatsachen werden wir einen Vorteil ziehen, indem wir die Konjunktionen der einzelnen $R_i(\bar{v}, \bar{v}')$ mit $h(\bar{v}')$ bilden und 'früh quantifizieren', um jede Variable v'_j zu eliminieren von der keines der restlichen $R_i(\bar{v}, \bar{v}')$ mehr abhängt.

Beispiel 2. Betrachten wir wieder das Beispiel des Modulo-8 Zählers aus dem Kapitel über die Modellierung von Systemen. Es gilt:

$$R_0(\bar{v}, \bar{v_0'}) = (v_0' \Leftrightarrow \neg v_0)$$

$$R_1(\bar{v}, \bar{v_1}') = (v_1' \Leftrightarrow v_0 \oplus v_1)$$

$$R_2(\bar{v}, \bar{v_2}') = (v_2' \Leftrightarrow (v_0 \wedge v_1) \oplus v_2)$$

In diesem Fall ist das relationale Produkt für $\mathbf{EX}h$ gegeben durch:

$$\exists v_0' \exists v_1' \exists v_2' [h(\bar{v'}) \wedge (R_0(\bar{v}, \bar{v_0'}) \wedge R_1(\bar{v}, \bar{v_1}') \wedge R_2(\bar{v}, \bar{v_2}'))]$$

Diese Formel lässt sich umschreiben in:

$$\exists v_2' \exists v_1' \exists v_0' [h(\bar{v'}) \wedge (R_0(\bar{v}, \bar{v_0'}) \wedge R_1(\bar{v}, \bar{v_1}') \wedge R_2(\bar{v}, \bar{v_2}'))].$$

Da $R_2(\bar{v}, \bar{v_2}')$ nicht von v_0' oder v_1' abhängt, können wir wiederum umschreiben in:

$$\exists v_2' [\exists v_1' \exists v_0' [(h(\bar{v'}) \wedge R_0(\bar{v}, \bar{v_0'})) \wedge R_1(\bar{v}, \bar{v_1}')] \wedge R_2(\bar{v}, \bar{v_2}')].$$

Und da $R_1(\bar{v}, \bar{v_1}')$ nicht von v_0' abhängt erhalten wir schliesslich:

$$\exists v_2' [\exists v_1' [\exists v_0' [h(\bar{v'}) \wedge R_0(\bar{v}, \bar{v_0'})] \wedge R_1(\bar{v}, \bar{v_1}')] \wedge R_2(\bar{v}, \bar{v_2}')]$$

Dieses relationale Produkt lässt sich berechnen, indem man mit $h(\bar{v}')$ beginnt und das Ergebnis in jedem Schritt mit $R_i(\bar{v}, \bar{v}')$ kombiniert und die entsprechenden Variablen ausquantifiziert.

Somit haben wir das Problem der Berechnung des kompletten relationalen Produkts auf die Berechnung einer Folge von kleineren, ähnlichen Produkten reduziert. Man beachte hierbei, dass die Zwischenresultate sowohl von Variablen in $\bar{v}$ als auch von Variablen in $\bar{v}'$ abhängen können. Im Beispiel wird auch deutlich, dass die Anordnung der Konjunktionen eine wichtige Rolle spielt: Man möchte die $R_i(\bar{v}, \bar{v}')$ so anordnen, dass die Variablen in $\bar{v}'$ so schnell wie möglich ausquantifiziert werden können und die Variablen aus $\bar{v}$ so langsam wie möglich hinzugefügt werden.
Im vorangehenden Beispiel wurde das relationale Produkt von $\mathbf{EX}h$ berechnet, das heisst, es wurden Vorgänger einer Zustandsmenge berechnet. Der Fall, dass Nachfolger berechnet werden sollen verläuft ähnlich, mit dem Unterschied, dass bei der Berechnung des relationalen Produkte nicht die Variablen des nächsten Zustands ausquantifiziert werden sondern die des aktuellen Zustands.

Beispiel 3. Um die Berechnung des relationalen Produkts für die Nachfolger einer Zustandsmenge zu demonstrieren, betrachten wir wieder den Modulo-8 Zähler. In diesem Fall hat das relationale Produkt die Form:

$$\exists v_0 \exists v_1 \exists v_2 [h(\bar{v}) \wedge (R_0(v_0, \bar{v}') \wedge R_1(v_0, v_1, \bar{v}') \wedge R_2(v_0, v_1, v_2, \bar{v}'))].$$

Da die Konjunktion kommutativ und assoziativ ist, lässt sich dies umschreiben zu

$$\exists v_0 \exists v_1 \exists v_2 [((h(\bar{v}) \wedge R_2(v_0, v_1, v_2, \bar{v}')) \wedge R_1(v_0, v_1, \bar{v}')) \wedge (R_0(v_0, \bar{v}')],$$

und da nun wieder $(R_0(v_0, \bar{v}')$ nicht von v_1 oder v_2 abhängt, ergibt sich

$$\exists v_0 [\exists v_1 \exists v_2 [(h(\bar{v}) \wedge R_2(v_0, v_1, v_2, \bar{v}')) \wedge R_1(v_0, v_1, \bar{v}')] \wedge (R_0(v_0, \bar{v}')].$$

Nun sieht man noch, dass $R_1(v_0, v_1, \bar{v}')$ nicht von v_2 abhängt, wodurch man

$$\exists v_0 [\exists v_1 [\exists v_2 [h(\bar{v}) \wedge R_2(v_0, v_1, v_2, \bar{v}')] \wedge R_1(v_0, v_1, \bar{v}')] \wedge (R_0(v_0, \bar{v}')]$$

erhält.

Die Methode, die in diesen zwei Beispielen beschrieben wurde, lässt sich verallgemeinern, um beliebige konjunktiv partitionierte Übergangsrelationen mit n Zustandsvariablen zu berechnen:
Zuerst wählt der Benutzer eine Permutation ρ aus $\{0, \cdots, n-1\}$, die festlegt in welcher Reihenfolge die Partitionen $R_i(\bar{v}, \bar{v}')$ kombiniert werden. Für jedes i sei D_i die Menge der Variablen v_i' von denen $R_i(\bar{v}, \bar{v}')$ abhängt. Also sei

$$E_i = D_{\rho(i)} - \bigcup_{k=i+1}^{n-1} D_{\rho(k)}$$

Das heisst, dass E_i die Menge der Variablen ist, die in $D_{\rho(i)}$ enthalten sind, und die für alle $k > i$ nicht in $D_{\rho(k)}$ enthalten sind. Die E_i sind paarweise disjunkt und ihre Vereinigung enthält alle Variablen. Das relationale Produkt für $\mathbf{EX}h$ kann also berechnet werden als

$$h_1(\bar{v}, \bar{v}') = \exists_{v_j' \in E_0} [h(\bar{v}') \wedge R_{\rho(0)}(\bar{v}, \bar{v}')]$$

$$h_2(\bar{v}, \bar{v}') = \exists_{v_j' \in E_1} [h_1(\bar{v}, \bar{v}') \wedge R_{\rho(1)}(\bar{v}, \bar{v}')]$$

$$\vdots$$

$$h_n(\bar{v}) = \exists_{v_j' \in E_{n-1}} [h_{n-1}(\bar{v}, \bar{v}') \wedge R_{\rho(n-1)}(\bar{v}, \bar{v}')]$$

Das Resultat des relationalen Produkts ist h_n. Man beachte, dass falls ein E_i leer ist gilt

$$h_{i+1}(\bar{v}, \bar{v}') = [h_i(\bar{v}, \bar{v}') \wedge R_{\rho(i)}(\bar{v}, \bar{v}')],$$

und keine existenzielle Quantifikation auf dieser Ebene benutzt wird. Wie zu erwarten hat die Ordnung ρ einen grossen Einfluss darauf, zu welchem Zeitpunkt die Zustandsvariablen der Berechnung ausquantifiziert werden können. Deshalb ist es wichtig ,ρ vorsichtig zu wählen (ähnlich der Variablenordnung bei OBDDs). Um eine gute Ordnung für ρ zu finden, benutzen wir den Algorithmus:

while $(V \neq \emptyset)$ **do**
    Für jedes $v \in V$ berechne die Kosten um v zu eliminieren;
    Eliminiere die Variable mit den kleinsten Kosten und aktualisiere C und V;
end while;

Für jede Ordnung gibt es eine offensichtliche Ordnung der Relationen R_i, die wenn sie benutzt wird dafür sorgt, dass die Variablen in der durch den Algorithmus vorgeschlagenen Reihenfolge eliminiert werden können.
Der Algorithmus beginnt mit der Variablenmenge V und einer Sammlung C von Mengen, in der jedes $D_i \in C$ die Menge von Variablen ist von der R_i abhängt. Danach werden Schritt für Schritt alle Variablen eliminiert, wobei in jedem Schritt die Variable mit den niedrigsten Kosten eliminiert wird, und V und C aktualisiert werden. In diesem Zusammenhang ist noch die zu verwendende Kostenmetrik wichtig. Wir werden drei verschiedene Kostenmae betrachten :

1. Minimale Grösse: Die Kosten eine Variable zu eliminieren sind $|D_v|$. Mit dieser Kostenfunktion stellen wir sicher, dass die neue Relation von so wenig Variablen wie möglich abhängt.
2. Minimale Zunahme: Die Kosten eine Variable zu eliminieren sind

$$|D_v| - max_{A \in C, v \in A}|A|,$$

   was die Differenz zwischen der Grösse von D_v und der Grösse des grössten D_i das v enthält darstellt. Die Absicht hierbei ist es, die Eliminierung von Variablen zu vermeiden, die aus vielen kleinen Relationen eine Grosse machen würde. Anders ausgedrückt ziehen wir es vor, eine ohnehin schon grosse Relation nur etwas zu vergrössern, anstatt eine neue grosse Relation zu schaffen.
3. Minimale Summe: Die Kosten eine Variable zu eliminieren sind

$$\sum_{A \in C, v \in A} |A|,$$

   also einfach die Summe der Grössen aller D_i, die v enthalten. Da die Kosten einer Konjunktion von der Grösse ihrer Argumente abhängen, approximieren wir diese durch die Anzahl der Variablen von denen das Argument R_i abhängt.

Unser Ziel ist es, die Kosten des grössten BDDs, das während der Eliminierung entsteht, zu minimieren. Für unsere Abstraktion bedeutet dies eine Variablenordnung zu finden, die die Grösse der grössten während des Prozesses entstandenen Menge D_v minimiert. Die Wahl einer lokal optimalen Lösung garantiert dabei keine optimale Gesamtlösung, es können Gegenbeispiele für alle drei Kostenfunktionen gefunden werden. Tatsächlich kann gezeigt werden, dass das Problem eine optimale Ordnung zu finden **NP**-vollständig ist. In der Praxis zeigt sich aber, dass die Minimale-Summe Kostenfunktion im allgemeinen die beste Approximation der Kosten bietet.

Rekombinieren von Partitionen Wie im vorangehenden Abschnitt angesprochen, kann es in manchen Fällen sinnvoll sein, einige der R_i zu einem OBDD zu kombinieren, um eine kleinere (Gesamt-)Repräsentation zu erhalten. Kombiniert man nun auch Teile der Übergangsrelation auf die gleiche Weise, so kann die Berechnung der relationalen Produkts noch einmal erheblich beschleunigt werden.
Betrachten wir dazu das Beispiel eines n-Bit Zählers. Mit der normalen Variablenordnung ist die Anzahl der benötigten OBDD Knoten in beiden Fällen, im monolithischen und im komplett partitionierten, linear in n. Angenommen $h(\bar{v}')$ beschreibt einen einzelnen Zustand des Zählers. Berechnet man das relationale Produkt der komplett partitionierten Darstellung, so benötigt man n OBDD Operationen, von denen jede die Komplexität $O(n)$ hat, insgesamt ist die Komplexität also $O(n^2)$. Benutzen wir andererseits die monolithische Relation, so führen wir eine Opration der Komplexität $O(n)$ aus, sparen also einen Faktor n ein. In der Praxis können wir die Berechnung oftmals ohne eine signifikante Vergrösserung der Knotenanzahl beschleunigen, indem wir die OBDDs jedes einzelnen Registers kombinieren.

4.6 Symbolisches LTL Model Checking

In diesem Abschnitt wird gezeigt wie das LTL Model Checking Problem mit Hilfe symbolischer Techniken gelöst werden kann.
Sei $\mathbf{A}f$ eine Formel aus der linearen temporalen Logik. Wie wir wissen handelt es sich bei f um eine eingeschränkte Pfadformel, in der die einzigen Zustandsteilformeln atomare Propositionen sind. Wir wollen nun all diejenigen Zustände $s \in S$ finden, für die gilt: $M, s \models \mathbf{A}f$.Da $M, s \models \mathbf{A}f$ genau dann gilt, wenn $M, s \models \neg\mathbf{E}\neg f$ gilt, ist es ausreichend wenn wir Formeln der Art $\mathbf{E}f$ behandeln können.

Der früher beschriebene Algorithmus von Lichtenstein und Pnueli [17] für diese Problem war zwar linear in der Grösse des Models M, jeodch exponentiell in der Grösse der Formel f, und deshalb für grosse Beispiele nicht anwendbar, da es zu einer explosionsartigen Vermehrung der Zustände kommen kann. Da die exponentielle Komplexität des Lichtenstein-Pnueli Algorithmus durch die Tableaukonstruktion verursacht wird, werden wir das Tableau nun als OBDD darstellen, um die Komplexität zu verringern. Zusätzlich wird die Definition des Tableaus [3, 18] noch etwas modifiziert, was oftmals zu einer Darstellung mit weniger Zuständen führt.

Prinzipiell arbeitet der Model Checking Algorithmus in diesem Fall folgendermassen. Sei eine Formel $\mathbf{E}f$ und eine Kripke Struktur M gegeben. Wir beginnen damit, dass Tableau T für die Formel f zu konstruieren. T ist wieder eine Kripke Struktur und enthält *jeden* Pfad, der f erfüllt. Durch die Komposition von T und M finden wir die Menge der Pfade, die sowohl in T als auch in M enthalten sind. Ein Zustand in M erfüllt $\mathbf{E}f$ genau dann, wenn es ein Startzustand der Komposition von T und M ist, der f erfüllt. Um diese Zustände zu finden, wird

die CTL Model Checking Prozedur aus Abschnitt 4.3 benutzt.

Die Konstruktion des Tableaus geschieht dabei folgendermassen. Sei AP_f die Menge der atomaren Propositionen in f. Das mit f assoziierte Tableau ist eine Struktur $T = (S_T, R_T, L_T)$ mit AP_f als Menge ihrer atomaren Propositionen. Im Gegensatz zum Lichtenstein-Pnueli Algorithmus benutzen wir nun nicht den kompletten Abschluss der Formel. Jeder Zustand im Tableau ist eine Menge von Elementarformeln von f. Die Menge von elementaren Teilformeln von f wird als $el(f)$ bezeichnet und ist rekursiv definiert:

Definition 2.
- $el(p) = p$, *wenn* $p \in AP_f$.
- $el(\neg g) = el(g)$.
- $el(g \vee h) = el(g) \cup el(h)$.
- $el(\mathbf{X}g) = \{\mathbf{X}g\} \cup el(g)$.
- $el(g\mathbf{U}h) = \{\mathbf{X}(g\mathbf{U}h) \cup el(g) \cup el(h)$.

Dadurch ist die Menge S_T der Zustände des Tableaus gleich $P(el(f))$. Die Markierungsfunktion L_T wird so definiert, dass jeder Zustand durch die Menge seiner atomaren Propositionen markiert wird. Um nun die Übergangsrelation R_T zu konstruieren, benötigen wir eine zusätzliche Funktion sat, die mit jeder Teilformel g von f eine Menge von Zuständen aus S_T assoziiert. Intuitiv wird $sat(g)$ die Menge von Zuständen sein, die g erfüllen:

- $sat(g) = \{s | g \in s\}$, wobei $g \in el(f)$ ist.
- $sat(\neg g) = \{s | s \notin sat(g)\}$.
- $sat(g \vee h) = sat(g) \cup sat(h)$.
- $sat(g\mathbf{U}h) = sat(h) \cup (sat(g) \cap sat(\mathbf{X}(g\mathbf{U}h)))$.

Wir wollen nun, dass die Übergangsrelation die Eigenschaft hat, das jede Elementarformel eines Zustands in diesem Zustand wahr ist. Es ist offensichtlich, dass wenn $\mathbf{X}g$ in einem Zustand s vorkommt, alle Nachfolgzustände g erfüllen sollen. Weiterhin muss — da wir LTL Formeln behandeln, — gelten, dass wenn $\mathbf{X}g$ nicht in einem Zustand s vorkommt, alle Nachfolgzustände $\neg\mathbf{X}g$ erfüllen sollten, was bedeutet, dass kein Folgezustand von s g erfüllen sollte. Aus diesen Gründen definieren wir R_T als:

$$R_T(s, s') = \bigwedge_{\mathbf{X}g \in el(f)} s \in sat(\mathbf{X}g) \Leftrightarrow s' \in sat(g)$$

Betrachten wir nun noch einmal das Mikrowellenbeispiel aus dem Kapitel über die Grundlagen des Model Checking. Sei $g = (\neg heat)\mathbf{U}close$ die Spezifikation. In Abbildung 10 ist die Übergangsrelation R_T für das Tableau der Formel $\neg g$ dargestellt. Um die Anzahl der Kanten zu reduzieren, verbinden wir zwei Zusände s uns s' mit einem bidirktionalen Pfeil, wenn eine Kante in beide Richtungen existiert. Jede Teilmenge von $el(g)$ ist ein Zustand von T. Bei der Markierug der Zustände benutzen wir h (bzw. h) als Abkürzung für $heat$ (bzw. $\neg heat$) und c (bzw. c) für $close$ (bzw. $\neg close$).

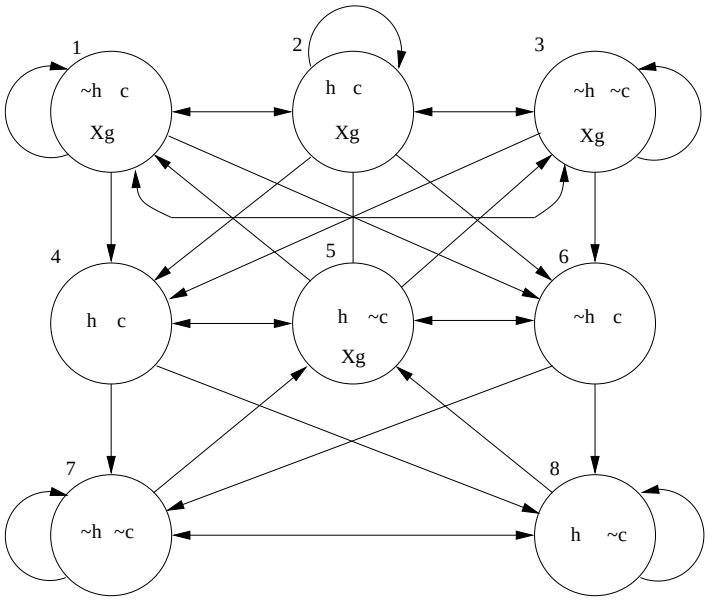


Abb. 10. Tableau für $(\neg heat)\mathbf{U}close$

Man beachte, dass $sat(\mathbf{X}g) = \{1, 2, 3, 5\}$, da jeder dieser Zustände $\mathbf{X}g$ enthält, und $sat(g) = \{1, 2, 3, 4, 6\}$, da jeder dieser Zustände entweder *close* oder $\neg heat$ und $\mathbf{X}g$ enthält. $sat(\neg g) = \{5, 7, 8\}$ ist das Komplement von $sat(g)$.
Da die Definition von R_T eine Konjunktion von 'genau dann wenn'-Bedingungen ist, gibt es einen Übergang von jedem Zustand aus $sat(\mathbf{X}g)$ in jeden Zustand von $sat(g)$ und es gibt einen Übergang vom Komplement von $sat(\mathbf{X}g)$ in jeden Zustand aus dem Komplement von $sat(g)$. Ungünstigerweise garantiert uns die Definition von R_T nicht, dass 'Eventualitäts'-Eigenschaften erfüllt werden. Dieses Verhalten sieht man in Abbildung 10. Obwohl Zustand 3 zu $sat(g)$ gehört, erfüllt die Schleife, die immer wieder in Zustand 3 führt, die Formel g nicht. Deshalb ist eine weitere Bedingung notwendig, um die Pfade zu identifizieren entlang denen f gilt. Ein Pfad π der in einem Zustand $s \in sat(f)$ beginnt erfüllt f genau dann, wenn gilt

– Für jede Teilformel $g\mathbf{U}h$ von f und jeden Zustand s aus π gilt: Ist $s \in sat(g\mathbf{U}h)$, dann ist entweder $s \in sat(h)$ oder es existiert ein Folgezustand t auf dem Pfad π für den gilt $t \in sat(h)$.

Um die Schlüsseleigenschaft der Tableaukonstruktion zu spezifizieren, benötigen wir einige neue Notationen:
Sei $\pi' = s'_0, s'_1, \cdots$ ein Pfad in einer Kripke Struktur M, dann ist $label(\pi') = L(s'_0), L(s'_1), \cdots$. Sei $l = l_0, l_1, \cdots$ eine Folge von Teilmengen der Menge AP und sei $AP' \subseteq AP$. Die Einschränkung von l auf AP' die durch $l|_A P'$ angezeigt

wird, ist die Folge $m_0, m_1 \cdots$ in der $m_i = l_i \cap AP'$ für jedes $i > 0$ gilt. Zusätzlich benutzen wir $sub(f)$, um die Menge der Teilformeln von f zu bezeichnen. Das folgende Theorem verdeutlicht die (intuitive) Behauptung, dass T jeden Pfad enthält, der f erfüllt:

Theorem 2. *Sei T das Tableau der Pfadformel f. Dann gilt für jede Kripke Struktur M und für jeden Pfad π' von M, dass wenn $M, \pi' \models f$ gilt, auch ein Pfad π in T existiert, der in einem Zustand $sat(f)$ beginnt, so dass $label(\pi')|_{AP_f} = label(\pi)$.*

Als nächstes wollen wir das Produkt $P = (S, R, L)$ des Tableaus $T = (S_T, R_T, L_T)$ und der Kripke Struktur $M = (S_M, R_M, L_M)$ berechnen:

- $S = \{(s, s') | s \in S_T, s' \in S_M$ und $L_M(s')|_{AP_f} = L_T(s)\}$.
- $R((s, s'), (t, t'))$ genau dann wenn $R_T(s, t)$ und $R_M(s', t')$ gilt.
- $L((s, s')) = L_T(s)$

Dabei kann es vorkommen, dass die entstehende Übergangsrelation nicht total ist. Wenn dies passiert, entfernen wir alle Zustände aus S, die keine Nachfolger haben, und schränken die Übergangsrelation R entsprechend ein.
Das nächste Lemma zeigt, dass in P genau diejenigen Folgen π'' liegen, für die sowohl Pfade π in T als auch Pfade π' in M existieren, die die gleichen Markierungen der Propositionen aus AP_f besitzen.

Lemma 5. *$\pi'' = (s_0, s'_0), (s_1, s'_1), \cdots$ ist genau dann ein Pfad mit $L_P((s_i, s'_i)) = L_T(s_i)$ für alle $i \geq 0$, wenn es einen Pfad $\pi = s_0, s_1, \cdots$ in T und einen Pfad $\pi' = s'_0, s'_1, \cdots$ in M gibt für die gilt: $L_T(s_i) = L_M(s'_i)|_{AP_f}$ für alle $i \geq 1$*

Die Funktion *sat* wird nun so erweitert, dass $(s, s') \in sat(g)$ genau dann gilt, wenn $s \in sat(g)$ gilt. Damit ist *sat* über die Menge der Zustände des Produktes P definiert.
Nun wird der CTL Model Checking Algorithmus angwendet, um alle Zustände V in P,$V \subseteq sat(f)$ zu finden, die $\mathbf{EG}true$ unter den Fairness Constraints

$$\{sat(\neg(g\mathbf{U}h) \vee h) | g\mathbf{U}h \text{kommt in f vor}\}$$

erfüllen.
Jeder der Zustände V liegt in $sat(f)$ und ist überdies der Startzustand eines unendlich langen Pfades, der alle Fairness Constraints erfüllt. Alle diese Pfade haben die Eigenschaft, dass keine Teilformel $g\mathbf{U}h$ entlang des Pfades fast immer gilt während h nicht gilt.

Theorem 3. *$M, s \models bfEf$ genau dann, wenn es einen Zustand s in T gibt, so dass $(s, s') \in sat(f)$ und $P(s, s') \models \mathbf{EG}True$ unter den Fairness Constraints $\{sat(\neg(g\mathbf{U}h) \vee h) | g\mathbf{U}h$kommt in f vor$\}$.*

Beispiel 4. Um unsere Konstruktion zu verdeutlichen, überprüfen wir die Formel $g = \neg((\neg heat)\mathbf{U}close)$ auf der Kripke Struktur M der Mikrowelle. Das Tableau der Formel wird aus Abb. 10 übernommen. Berechnen wir nun wie beschrieben

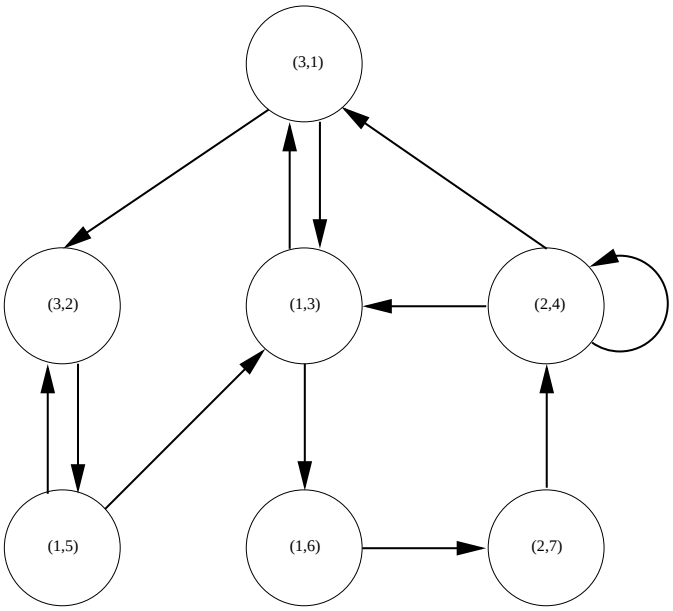


Abb. 11. Das Produkt P der Mikrowelle M und des Tableaus T

das Produkt P, so erhalten wir die Kripke Struktur aus Abb. 11. Jeder Zustand des Produkts ist durch ein Paar (s, s'), mit $s \in M$ und $s' \in T$, markiert. Die Zustände $(4,4), (4,7), (6,3), (6,5), (6,6), (7,1)$ und $(7,2)$ brauchen im Diagramm des Produkts nicht berücksichtigt zu werden, da sie keine Startzustände von unendlich langen Pfaden sind. Nun wird der CTL Model Checking Algorithmus angewendet, um die Menge V von Zuständen aus $sat(\neg g)$ zu finden, die $\mathbf{EG}true$ unter Berückichtigung der Fairness Constraint $sat(\neg((\neg heat)\mathbf{U}close) \vee close)$ erfüllen. Da $sat(\neg g) = \{(7,1),(7,2)\}$ ist, aber keiner dieser Zustände der Beginn eines unendlich langen Pfades ist, ist $V = \emptyset$. Daraus können wir folgern, dass kein Zustand in M $\mathbf{E}\neg((\neg heat)\mathbf{U}close)$ erfüllt und deshalb erfüllen alle Zustände $\mathbf{A}((\neg heat)\mathbf{U}close)$.

Nun wird beschrieben, wie diese Prozedur unter Verwendung von OBDDs implementiert werden kann. Angenommen die Übergangsrelation von M wird durch ein OBDD repräsentiert, das über die Menge seiner atomaren Propositionen defniniert ist. Um die Übergangsrelation von T als OBDD darzustellen, wird jede Elementarformel g einer Zustandsvariable v_g zugeordnet. Ist g eine atomare Proposition, so ist $v_g = g$. Dadurch sind M und T über Variablen aus AP_f sowie einiger zusätzlicher Zustandvariablen definiert.

Wir beschreiben nun die Übergangsrelation R_T als boolesche Formel mittels zweier Kopien $\bar{v}$ und $\bar{v}'$ der Zustandsvariablen und konvertieren diese Formel in ein OBDD, um eine prägnante Darstellung des Tableaus zu erhalten. Bei der

Konstruktion von P ist es günstig, Zustandvariablen, die in AP_f vorkommen, separat zu behandeln. Boolesche Vektoren, die diesen Zustandvariablen Werte zuweisen, werden durch $\bar{p}$ gekennzeichnet. Jeder Zustand in S_T wird also durch ein Paar $(\bar{p}, \bar{r})$ dargestellt, wobei $\bar{r}$ für einen booleschen Vektor steht, der Werte an Variablen zuweist, die im Tableau, aber nicht in AP_f vorkommen und ein Zustand in S_M wird mit $(\bar{p}, \bar{q})$ bezeichnet, wobei $\bar{q}$ für einen booleschen Vektor steht, der denjenigen Zustandsvariablen aus M Werte zuweist, die nicht in f vorkommen. Damit ist die Übergangsrelation R_P des Produktes zweier Kripke Strukturen gegeben durch

$$R_P(\bar{p}, \bar{q}, \bar{r}, \bar{p}', \bar{q}', \bar{r}') = R_T(\bar{p}, \bar{r}, \bar{p}', \bar{r}') \wedge R_M(\bar{p}, \bar{q}, \bar{p}', \bar{q}').$$

Wir benutzen den Algorithmus für symbolische Model Checking der Fairness Constraints berücksichtigt, um die Menge der Zustände V zu finden, die $\mathbf{EG}true$ unter den gegebenen Fairness Constraints efüllen. Da jeder Zustand aus V durch einen booleschen Vektor der Form $(\bar{p}, \bar{q}, \bar{r})$ dargestellt wird, erfüllt ein Zustand $(\bar{p}, \bar{q})$ aus M $\mathbf{E}f$ genau dann, wenn ein $\bar{r}$ existiert, so dass $(\bar{p}', \bar{q}', \bar{r}') \in V$ und $(\bar{p}', \bar{r}') \in sat(f)$.

Literatur

1. E. M Clarke, E. A. Emerson, and A. P Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Language*, January 1983.
2. D. Dolev, M. Klave and M. Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms 3*, pages 245-260.
3. J. R. Burch, E. M. Clarke, K. L. McMillian, D. L Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. *Information and Computation* 98(2),pages 142-170.
4. K. L. McMillian. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic, 1993.
5. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5: pages 285-309.
6. E. M Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, LNCS 131. Springer, 1981.
7. E. A. Emerson and C-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In LICS86 [8], pp. 276-278.
8. *Proceedings of the 1st Annual Symposium on Logic in Computer Science.* IEEE Computer Society Press, 1986.
9. E. A. Emerson and E. M Clarke. Characterzing correctness properties of parallel programs using fixpoints. In LNCS85, *Automata, Languages and Programming*, pp. 169-181. Springer, 1980.
10. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.
11. E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model checking for fragments of mu-calculus. In Courcoubetis [13] pp.385-396.

12. J. R. Burch, E. M. Clarke, K. L. McMillian, and D. L Dill. Sequential circuit verification using symbolic model checking In *Proceedings of the 27th Design Automation Conference*, pp. 46-51. IEEE, 1990.
13. C. Courcoubetis, ed. *Proceedings ot the 5th Workshop on Computer-Aided Verification*, June/July 1993.
14. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
15. J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pp. 403-407. IEEE, 1991.
16. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, eds., *Proceedings of the 1991 International Conference on VLSI*, pp. 49-58. August 1991.
17. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Language*, pp.97-107. ACM, 1985.
18. E. M. Clarke, O. Grumberg, and H. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design* 10(1): 47-71.

Äquivalenzen und Anordnungen von Kripke-Strukturen

Ralf Wimmer

Institut für Informatik
Albert-Ludwigs-Universität Freiburg
wimmer@informatik.uni-freiburg.de

Zusammenfassung. Beim Model Checking ist ein häufig auftretendes Problem, daß die Menge der Zustände der Kripke-Struktur, die untersucht wird, zu groß ist. Ziel dieser Seminararbeit ist es zu zeigen, wie Äquivalenzrelationen und Anordnungen auf Kripke-Strukturen dazu verwendet werden können, den Zustandsraum zu verkleinern.

Einleitung

Ein wichtiges Verfahren bei der Verifikation von Hard- und Software ist heutzutage das Model Checking. Der Schaltkreis bzw. das Programm wird dabei in einen endlichen Automaten, die Kripke-Struktur, übersetzt. Die nachzuweisenden Eigenschaften werden in temporaler Logik (CTL* o. ä.) spezifiziert. Nun ist es die Aufgabe des Model Checking herauszufinden, ob die Eigenschaften in allen Anfangszuständen der Kripke-Struktur erfüllt sind.

Ein großes Problem ist dabei, daß der Zustandsraum der Kripke-Struktur extrem groß werden kann. Enthält beispielsweise der Schaltkreis 32 Flipflops, die jeweils ein Bit speichern können, so kann der Zustandsraum bereits bis zu $2^{32} = 4294967296$ Zustände umfassen.

Ziel ist es nun, Techniken zu finden, mit deren Hilfe der Zustandsraum verkleinert werden kann, ohne daß sich an der Gültigkeit der gegebenen CTL*-Formeln etwas ändert. Zu diesen Techniken zählen die **Bisimulationen** und **Simulationen**, die hier genauer untersucht werden sollen. Dabei werde ich folgendermaßen vorgehen:

Diese Seminararbeit ist in drei Abschnitt unterteilt: Im ersten werde ich mich mit Bisimulationen beschäftigen. Mit deren Hilfe kann man eine Äquivalenzrelation auf den Kripke-Strukturen definieren. Ich werde einige Eigenschaften dieser Äquivalenzrelation im Hinblick auf die Logiken CTL* und CTL beweisen. Darüber hinaus will ich Algorithmen angeben, mit denen man zum einen eine Bisimulation zwischen zwei Kripke-Strukturen konstruieren und zum anderen eine Kripke-Struktur minimieren kann.

Der zweite Abschnitt ist den Anordnungen von Kripke-Strukturen gewidmet. Dabei wird der Begriff der Simulation eine große Rolle spielen. Ich werde untersuchen, wie man diese Anordnungen benutzen kann, um den Zustandsraum

zu verkleinern. Allerdings wird sich dabei herausstellen, daß wir uns auf die Teillogik ACTL* einschränken müssen.

Im dritten Abschnitt verwende ich die Simulationen aus Abschnitt 2, um eine Äquivalenzrelation zu definieren. Diese ist schwächer als die Äquivalenzrelation, die im Abschnitt 1 mit Hilfe der Bisimulationen definiert wurde. Dafür kann aber der Zustandsraum stärker verkleinert werden. Einen Algorithmus, der dieses bewerkstelligt, will ich zum Schluß angeben.

1 Äquivalenzklassen auf Kripke-Strukturen

1.1 Grundlagen

In diesem Abschnitt wollen wir eine Äquivalenzrelation $\equiv^B$ auf der Menge der Kripke-Strukturen definieren. Dabei soll gelten, daß äquivalente Kripke-Strukturen dieselbe Menge von CTL*-Formeln erfüllen. Daß unsere Definition diese Eigenschaft tatsächlich erfüllt, weisen wir später nach.
Wenn wir zeigen können, daß zwei gegebene Kripke-Strukturen äquivalent sind, dann können wir den Model-Checking-Algorithmus auf die kleinere von beiden anwenden und Zeit und Speicherplatz sparen.

Definition 1. *Seien $M = (AP, S, S_0, R, L)$ und $M' = (AP, S', S'_0, R', L')$ zwei Kripke-Strukturen über denselben Mengen von atomaren Aussagen AP. Eine Relation $B \subseteq S \times S'$ heißt **Bisimulation** zwischen M und M', wenn für alle $s \in S$ und $s' \in S'$ gilt:*
Aus $B(s, s')$ folgt:

1. $L(s) = L'(s')$
2. $\forall s_1 \in S : \big(R(s, s_1) \Rightarrow \exists s'_1 \in S' : R'(s', s'_1) \wedge B(s_1, s'_1)\big)$
3. $\forall s'_1 \in S' : \big(R(s', s'_1) \Rightarrow \exists s_1 \in S : R(s, s_1) \wedge B(s_1, s'_1)\big)$

Definition 2. *Zwei Kripke-Strukturen M und M' heißen **Bisimulation-äquivalent** ($M \equiv^B M'$), wenn es eine Bisimulation B zwischen M und M' gibt, so daß gilt:*

1. $\forall s_0 \in S_0 \exists s'_0 \in S'_0 : B(s_0, s'_0)$
2. $\forall s'_0 \in S'_0 \exists s_0 \in S_0 : B(s_0, s'_0)$

Zwei Zustände $s \in S$ und $s' \in S'$ heißen Bisimulation-äquivalent oder, wenn aus dem Zusammenhang klar ist, daß wir uns auf eine Bisimulation beziehen, kurz äquivalent ($s \equiv^B s'$), wenn $B(s, s')$ gilt.

Beispiel 1. In Abbildung 1 sind zwei Bisimulation-äquivalente Kripke-Strukturen angegeben. Die zugehörige Bisimulation ist gegeben durch:

$$B = \{(1,1),\ (1,3),\ (2,2),\ (2,4)\}$$

Abbildung 2 zeigt ebenfalls zwei Bisimulation-äquivalente Strukturen. Hier lautet die Bisimulation

$$B = \{(1,1),\ (2,2),\ (3,3),\ (4,4),\ (5,4),\ (6,5),\ (6,6)\}$$

Abb. 1. Beispiel für zwei Bisimulation-äquivalente Strukturen

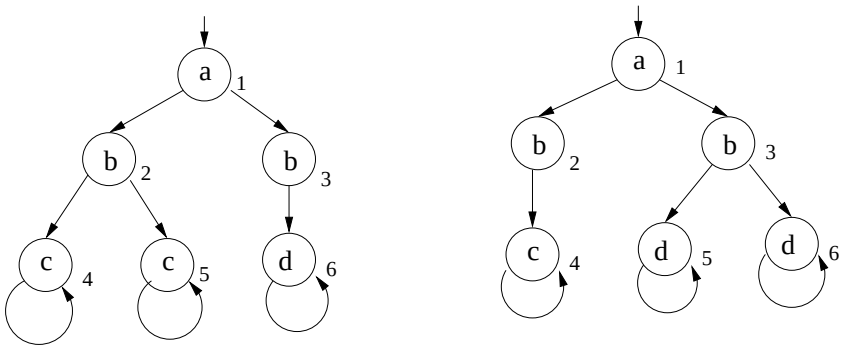


Abb. 2. Beispiel für Bisimulation-äquivalente Strukturen

Definition 3. *Zwei Pfade $\pi = s_0 s_1 s_2 \ldots$ in M und $\pi' = s'_0 s'_1 s'_2 \ldots$ in M' heißen* ***korrespondierend*** *genau dann, wenn für alle $i \geq 0$ gilt $B(s_i, s'_i)$.*

Lemma 1. *Seien s und s' zwei Zustände mit $B(s, s')$. Dann gibt es zu jedem Pfad π, der in s beginnt, einen korrespondierenden Pfad π', der in s' beginnt. Außerdem gibt es zu jedem Pfad π' von s' aus einen korrespondierenden Pfad π mit s als erstem Zustand.*

Beweis. Gelte $B(s, s')$ und sei $\pi = s_0 s_1 s_2 \ldots$ ein Pfad mit $s_0 = s$.
Wir konstruieren jetzt induktiv einen korrepondierenden Pfad $\pi' = s'_0 s'_1 s'_2 \ldots$
Nach Voraussetzung gilt: $B(s, s')$. Setze also $s'_0 = s'$.
Seien $s'_0, \ldots, s'_i$ für ein i bereits konstruiert. Es gilt somit $B(s_i, s'_i)$. Dann gibt es nach Definition einer Bisimulation und wegen $R(s_i, s_{i+1})$ einen Nachfolger t' von s'_i mit $B(s_{i+1}, t')$. Setze dann $s'_{i+1} := t'$.
Der Nachweis der 2. Behauptung erfolgt völlig analog. □

Das nächste Lemma zeigt die Bedeutung einer Bisimulation zwischen zwei Kripke-Strukturen auf, nämlich daß äquivalente Zustände und korrespondierende Pfade dieselben Zustands- bzw. Pfadformeln erfüllen.

Lemma 2. *Sei f eine CTL*-Pfad- oder -Zustandsformel. Sei B eine Bisimulation zwischen M und M' und gelte $B(s, s')$. Außerdem seien π und π' korrespondierende Pfade. Dann gilt:*

- *Falls f Zustandsformel ist, dann gilt:*

$$M, s \vDash f \quad \Leftrightarrow \quad M', s' \vDash f.$$

- *Falls f Pfadformel ist, dann gilt:*

$$M, \pi \vDash f \quad \Leftrightarrow \quad M', \pi' \vDash f.$$

Beweis. Sei $\pi = s_0 s_1 s_2 \ldots$ und $\pi' = s'_0 s'_1 s'_2 \ldots$. Bezeichne $\pi^{(k)}$ das k-te Suffix von π, d. h. $\pi^{(k)} = s_k s_{k+1} s_{k+2} \ldots$.
Wir zeigen die obige Aussage durch Induktion über den Aufbau von f:

- $f = p$ mit $p \in AP$.
  Wegen $B(s, s')$ gilt $L(s) = L'(s')$ und deshalb $M, s \vDash p \Leftrightarrow M', s' \vDash p$.
- $f = \neg f_1$ (Zustandsformel)

$$M, s \vDash f \Leftrightarrow M, s \nvDash f_1 \Leftrightarrow M', s' \nvDash f_1 \Leftrightarrow M', s' \vDash f$$

- $f = f_1 \vee f_2$ (Zustandsformel)

$$\begin{aligned} M, s \vDash f &\Leftrightarrow M, s \vDash f_1 \text{ oder } M, s \vDash f_2 \\ &\Leftrightarrow M', s' \vDash f_1 \text{ oder } M', s' \vDash f_2 \\ &\Leftrightarrow M', s' \vDash f. \end{aligned}$$

- $f = f_1 \wedge f_2$ (Zustandsformel)
  Analog zu $f = f_1 \vee f_2$.
- $f = E f_1$ (Zustandsformel)
  Gelte $M, s \vDash f$. Dann gibt es einen Pfad π_1, der in s beginnt und $M, \pi_1 \vDash f_1$ erfüllt. Dazu gibt es einen korrespondierenden Pfad π'_1 in M', der in s' beginnt. Nach Induktionsvoraussetzung gilt $M, \pi_1 \vDash f_1 \Leftrightarrow M', \pi'_1 \vDash f_1$. Daraus folgt $M', s' \vDash E f_1$.
  Die andere Richtung geht analog.
- $f = A f_1$ (Zustandsformel)
  Analog zu $f = E f_1$.
- $f = f_1$, wobei f_1 Zustands- und f Pfadformel ist.

$$\begin{aligned} M, \pi \vDash f &\Leftrightarrow M, s_0 \vDash f_1 \Leftrightarrow M', s'_0 \vDash f_1 \\ &\Leftrightarrow M', \pi' \vDash f \end{aligned}$$

- $f = X f_1$ (Pfadformel)
  Gelte $M, \pi \vDash f$. Somit gilt $M, \pi^{(1)} \vDash f_1$. Da π und π' korrespondieren, tun dies auch $\pi^{(1)}$ und $\pi'^{(1)}$. Deshalb gilt nach Induktionsvoraussetzung: $M', \pi'^{(1)} \vDash f_1$ und damit $M', \pi' \vDash f$.
  Andere Richtung analog.
- $f = F f_1$ (Pfadformel)
  Gelte $M, \pi \vDash f$. Somit gilt für ein $k \geq 0$: $M, \pi^{(k)} \vDash f_1$. Da π und π' korrespondieren, tun dies auch $\pi^{(k)}$ und $\pi'^{(k)}$. Deshalb gilt nach Induktionsvoraussetzung: $M', \pi'^{(k)} \vDash f_1$ und damit $M', \pi' \vDash f$.
  Andere Richtung analog.

- $f = Gf_1$ (Pfadformel)
  Analog zu $f = Ff_1$.
- $f = f_1 \, U \, f_2$ (Pfadformel)
  Gelte $M, \pi \vDash f_1 \, U \, f_2$. Nach Definition des Operators U gibt es ein k mit $M, \pi^{(k)} \vDash f_2$ und für $0 \leq i < k$: $M, \pi^{(i)} \vDash f_1$.
  Wenn π und π' korrespondieren, dann auch $\pi^{(j)}$ und $\pi'^{(j)}$ für jedes $j \geq 0$. Folglich gilt $M', \pi'^{(k)} \vDash f_2$ und $M', \pi'^{(i)} \vDash f_1$ und somit $M', \pi' \vDash f$.
  Andere Richtung analog.
- $f = f_1 \, R \, f_2$ (Pfadformel)
  Analog zu $f = f_1 \, U \, f_2$.

□

Theorem 1. *Wenn $M \equiv^B M'$, dann gilt für jede CTL*-Formel f:*

$$M \vDash f \quad \Leftrightarrow \quad M' \vDash f.$$

Es gilt sogar die umgekehrte Aussage: Wenn zwei Kripke-Strukturen genau die gleichen CTL*-Formeln erfüllen, dann sind sie Bisimulation-äquivalent[4].

1.2 CTL* vs. CTL

Wir wollen nun zeigen: Gibt es eine CTL*-Formel, die in der einen Kripke-Struktur erfüllt ist, in der anderen aber nicht, dann gibt es auch eine CTL-Formel, die in der einen Struktur gilt und in der anderen nicht. Wir werden dazu zu jeder Kripke-Struktur eine CTL-Formel angeben, die die Struktur bis auf Bisimulation-Äquivalenz charakterisiert. Dabei werden wir eine weitere Eigenschaft von Bisimulationen, die auf dem Begriff des Berechnungsbaumes beruht, beweisen. Wir folgen dabei der Darstellung in [1].

Definition 4. *Sei $s \in S$ ein Zustand einer Kripke-Struktur M. Für jedes $n \geq 0$ sei $Tr_n(s)$ der **Berechnungsbaum** zu s, der induktiv definiert ist durch*

- *$Tr_0(s)$ ist ein einzelner Knoten mit derselben Beschriftung wie s.*
- *$Tr_{n+1}(s)$ ist ein Baum mit Wurzel m, die dieselbe Beschriftung trägt wie s. Besitzt s die Nachfolgeknoten $s_1, \ldots, s_k$, dann besitzt m die Teilbäume $Tr_n(s_1), \ldots, Tr_n(s_k)$.*

Definition 5. *Zwei Berechnungsbäume $Tr_n(s)$ und $Tr_n(s')$ **korrespondieren** ($Tr_n(s) \equiv Tr_n(s')$) genau dann, wenn*

- *ihre Wurzeln dieselben Beschriftungen tragen und*
- *es zu jedem Teilbaum der Tiefe $n-1$ der Wurzel des einen Berechnungsbaums einen korrespondierenden Teilbaum des anderen gibt.*

Jetzt können wir eine weitere Eigenschaft von Bisimulationen angeben:

Lemma 3. *Es gilt:*

$$s \equiv^B s' \quad \Leftrightarrow \quad Tr_j(s) \equiv Tr_j(s') \text{ für alle } j \geq 0.$$

Beweis. Wir müssen zwei Richtungen zeigen:

„⇒“ Gelte $s \equiv^B s'$. Dann gibt es eine Bisimulation B mit $B(s, s')$. Wir zeigen durch Induktion über n, daß dann $Tr_n(s) \equiv Tr_n(s')$ ist.

- $n = 0$:
  Wegen $B(s, s')$ gilt $L(s) = L'(s')$ und damit $Tr_0(s) \equiv Tr_0(s')$.
- Gelte die Behauptung für ein n.
- $n \to n + 1$:
  Aus $B(s, s')$ folgt nach Definition einer Bisimulation:
  1. $L(s) = L'(s')$
  2. $R(s, s_1) \Rightarrow \exists s_1' : R(s', s_1') \wedge B(s_1, s_1')$.
     Nach Konstruktion besitzt $Tr_{n+1}(s)$ einen Teilbaum $Tr_n(s_1)$ und $Tr_{n+1}(s')$ einen Teilbaum $Tr_n(s_1')$ mit $Tr_n(s_1) \equiv Tr_n(s_1')$.
  3. $R(s', s_1') \Rightarrow \exists s_1 : R(s, s_1) \wedge B(s_1, s_1')$.
     Wie bei 2. folgt hier, daß es zu jedem Teilbaum von $Tr_{n+1}(s')$ einen entsprechenden Teilbaum von $Tr_{n+1}(s)$ gibt.

  Damit gilt $Tr_{n+1}(s) \equiv Tr_{n+1}(s')$.

„⇐“ Gelte $Tr_n(s) \equiv Tr_n(s')$. Offensichtlich erfüllen die beiden Bäume und damit auch die Zustände s und s' dieselben CTL*-Formeln. Folglich sind sie Bisimulation-äquivalent.

□

Lemma 4. *Sei $S = \{s_0, s_1, \ldots, s_k\}$ eine Menge von Zuständen. Dann gibt es eine Zahl $c \geq 0$, so daß für alle $0 \leq i, j \leq k$ mit $i \neq j$ gilt:*

$$s_i \not\equiv^B s_j \quad \Rightarrow \quad Tr_c(s_i) \not\equiv Tr_c(s_j)$$

*c heißt **charakteristische Zahl** von S.*

Beweis. Da für jede Bisimulation B gilt $\neg B(s_i, s_j)$, korrespondieren die Berechnungsbäume $Tr_n(s_i)$ und $Tr_n(s_j)$ nach Lemma 3 für ein n nicht. Setze

$$c := \min\{n \geq 0 \mid \forall 0 \leq i < j \leq k : \neg B(s_i, s_j) \wedge Tr_n(s_i) \not\equiv Tr_n(s_j)\}$$

□

Zu einem Berechnungsbaum $Tr_n(s)$ konstruieren wir jetzt induktiv eine CTL-Formel $\mathfrak{F}(Tr_n(s))$:

–

$$\mathfrak{F}(Tr_0(s)) = \bigwedge_{p \in L(s)} p \ \wedge \bigwedge_{q \in AP \setminus L(s)} \neg q$$

– Seien $s_1, \ldots, s_k$ die Nachfolgezustände von s. Dann setze

$$\mathfrak{F}(Tr_{n+1}(s)) = \bigwedge_{i=1}^{k} EX\mathfrak{F}(Tr_n(s_i)) \wedge AX\big(\bigvee_{i=1}^{k} \mathfrak{F}(Tr_n(s_i))\big) \wedge \mathfrak{F}(Tr_0(s))$$

Lemma 5. *Für alle $n \geq 0$ gilt:*

$$M, s \vDash \mathfrak{F}(Tr_n(s)).$$

Beweis. Induktion über n:

– $n = 0$:
$$M, s \vDash p \Leftrightarrow p \in L(s) \text{ und } M, s \vDash \neg q \Leftrightarrow q \in AP \setminus L(s)$$
Daraus folgt unmittelbar $M, s \vDash \mathfrak{F}(Tr_0(s))$.
– Gelte für alle $s \in S$ und ein $n \geq 0$ $M, s \vDash \mathfrak{F}(Tr_n(s))$. Dann gilt für jeden Nachfolgezustand s_i: $M, s_i \vDash \mathfrak{F}(Tr_n(s_i))$ und damit $M, s \vDash \bigwedge_{i=1}^{k} EX\mathfrak{F}(Tr_n(s_i))$.
Da in jedem Nachfolgezustand eine der Formeln $\mathfrak{F}(Tr_n(s_i))$ wahr ist, gilt $M, s \vDash AX\big(\bigvee_{i=1}^{k} \mathfrak{F}(Tr_n(s_i))\big)$.
Damit folgt, daß $M, s \vDash \mathfrak{F}(Tr_{n+1}(s))$ gilt. □

Lemma 6. *Falls $s \vDash \mathfrak{F}(Tr_n(s'))$ gilt, dann gilt auch $Tr_n(s) \equiv Tr_n(s')$.*

Beweis. Induktion über n:

– Der Fall $n = 0$ ist klar.
– $n - 1 \to n$: Seien $s_1, \ldots, s_p$ die Söhne von s in $Tr_n(s)$ und $s'_1, \ldots, s'_q$ die Söhne von s' in $Tr_n(s')$. Es ist leicht einzusehen, daß s und s' dieselbe Beschriftung tragen.
Wir müssen zeigen, daß es zu jedem $Tr_{n-1}(s_i)$ einen korrespondierenden Teilbaum $Tr_{n-1}(s'_j)$ gibt und umgekehrt.
Wegen $s \vDash \mathfrak{F}(Tr_n(s'))$ gilt auch $s \vDash AX(\bigvee_j \mathfrak{F}(Tr_{n-1}(s'_j))$. Da s_i ein Nachfolger von s ist, muß $s_i \vDash \mathfrak{F}(Tr_{n-1}(s'_j))$ für ein j gelten.
Entsprechend gilt auch $s \vDash \bigwedge_j EX\mathfrak{F}(Tr_{n-1}(s'_j))$. Da s'_j ein Nachfolger von s' ist, gilt $s \vDash EX\mathfrak{F}(Tr_{n-1}(s'_j))$. Deshalb muß es einen Nachfolger s_i von s geben mit $s_i \vDash \mathfrak{F}(Tr_{n-1}(s'_j))$.
Mit der Induktionsvoraussetzung folgt dann die Behauptung. □

Lemma 7. *Wenn s ein Zustand einer Kripke-Struktur M ist, dann gibt es eine CTL-Formel $\mathfrak{C}(M, s)$, die s bis auf Bisimulation-Äquivalenz innerhalb von M eindeutig bestimmt, d. h. $\mathfrak{C}(M, s)$ ist wahr in s und jedem Zustand, der Bisimulation-äquivalent zu s ist, und falsch sonst.*

Beweis. Wir wählen $\mathfrak{C}(M, s) = \mathfrak{F}(Tr_c(s))$, wobei c die charakteristische Zahl von M ist. $\mathfrak{C}(M, s)$ ist wahr in s und deshalb auch in jedem Zustand, der äquivalent zu s ist. Sei s' ein Zustand, der nicht äquivalent zu s ist. Dann gilt $Tr_c(s) \not\equiv Tr_c(s')$. Nach Lemma 6 gilt dann $s' \nvDash \mathfrak{C}(M, s)$. □

Jetzt haben wir alle Vorbereitungen getroffen, um folgenden Satz zu beweisen:

Theorem 2. *Sei M eine Kripke-Struktur mit Anfangszustand s_0. Dann gibt es eine CTL-Formel $F(M, s_0)$, die M bis auf Bisimulation-Äquivalenz charakterisiert, d. h.*

$$M', s_0' \vDash F(M, s_0) \Leftrightarrow M \equiv^B M'.$$

Beweis. Sei $s \in S$ und seien $s_1, \ldots, s_p$ die Nachfolgezustände von s in M. Wir definieren

$$G(M, s) := AG\left(\mathfrak{C}(M, s) \rightarrow \bigwedge_i EX\, \mathfrak{C}(M, s_i) \wedge AX \bigvee_i \mathfrak{C}(M, s_i)\right)$$

$G(M, s)$ beschreibt alle möglichen Übergänge von s aus. Dann sei

$$F(M, s_0) := \mathfrak{C}(M, s_0) \wedge \bigwedge_{s \in S} G(M, s).$$

Wenn zwei Strukturen M und M' äquivalent sind, dann erfüllen sie dieselben Formeln. Da $M, s_0 \vDash F(M, s_0)$ gilt, gilt auch $M', s_0' \vDash F(M, s_0)$.
In der anderen Richtung zeigt man mit Induktion über n, daß $Tr_n(s_0) \equiv Tr_n(s_0')$ für jedes $n \geq 0$. Nach Lemma 3 gilt dann $M \equiv M'$. □

Korollar 1. *Seien zwei Kripke-Strukturen M und M' mit den Anfangszuständen s_0 bzw. s_0' gegeben. Wenn es eine CTL*-Formel f gibt, die wahr in M und falsch in M' ist, dann gibt es auch eine CTL-Formel f', die wahr in M und falsch in M' ist.*

Diese Aussage ist um so interessanter, da CTL und CTL* bekanntermaßen unterschiedliche Ausdrucksstärke haben. Beispielsweise gibt es zu der CTL*-Formel $EGF\, p$ mit $p \in AP$ keine äquivalente CTL-Formel ([5]).

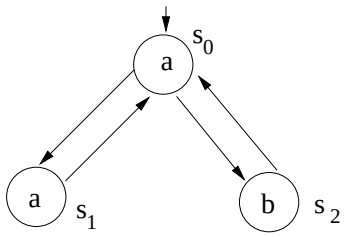


Abb. 3. Kripke-Struktur zur Konstruktion der charakt. CTL-Formel

Beispiel 2. Wir wollen für die einfache Kripke-Struktur M aus Abbildung 3 die charakteristische CTL-Formel berechnen:
Die charakteristische Zahl c von M ist 1, denn $Tr_0(s_0) \not\equiv Tr_0(s_2)$, $Tr_0(s_1) \not\equiv Tr_0(s_2)$ und $Tr_1(s_0) \not\equiv Tr_1(s_1)$.
Wir setzen

$$\begin{aligned}
\mathfrak{C}(M, s_0) &:= a \wedge \neg b \wedge EX(a \wedge \neg b) \wedge EX(\neg a \wedge b) \wedge AX(a \wedge \neg b \vee \neg a \wedge b) \\
\mathfrak{C}(M, s_1) &:= a \wedge \neg b \wedge EX(a \wedge \neg b) \wedge AX(a \wedge \neg b) \\
\mathfrak{C}(M, s_2) &:= \neg a \wedge b \wedge EX(a \wedge \neg b) \wedge AX(a \wedge \neg b)
\end{aligned}$$

Dann erhalten wir für die charakteristische CTL-Formel von M:

$$\begin{aligned}
F(M, s_0) = \mathfrak{C}(M, s_0) & \\
& \wedge AG\big(\mathfrak{C}(M, s_0) \Rightarrow EX\mathfrak{C}(M, s_1) \wedge EX\mathfrak{C}(M, s_2) \wedge AX(\mathfrak{C}(M, s_1) \vee \mathfrak{C}(M, s_2))\big) \\
& \wedge AG\big(\mathfrak{C}(M, s_1) \Rightarrow EX\mathfrak{C}(M, s_0) \wedge AX\mathfrak{C}(M, s_0)\big) \\
& \wedge AG\big(\mathfrak{C}(M, s_2) \Rightarrow EX\mathfrak{C}(M, s_0) \wedge AX\mathfrak{C}(M, s_0)\big)
\end{aligned}$$

1.3 Deterministische Kripke-Strukturen

Definition 6. *Sei M eine Kripke-Struktur und sei $s \in S$. Dann setze*

$$\mathfrak{L}(s) := \{\langle \sigma_0\sigma_1\sigma_2 \ldots \rangle \mid \textit{Es gibt einen Pfad } \pi = s_0 s_1 s_2 \ldots \textit{ mit } s_0 = s \textit{ und } \sigma_i = L(s_i) \textit{ für } i \geq 0\}$$

*Die **Sprache** von M ist gegeben durch*

$$\mathfrak{L}(M) := \bigcup_{s_0 \in S_0} \mathfrak{L}(s_0).$$

Definition 7. *Eine Kripke-Struktur M heißt **deterministisch**, wenn*

- *sie genau einen Anfangszustand s_0 hat und*
- *für alle $s \in S$ gilt: $\big(R(s,t) \wedge R(s,u)\big) \Rightarrow L(t) \neq L(u)$.*

Lemma 8. *Sei $\sigma = \langle \sigma_0\sigma_1\sigma_2 \ldots \rangle \in \mathfrak{L}(M)$ und M deterministisch. Sei $\pi = s_0 s_1 s_2 \ldots$ ein Pfad, der σ erzeugt, d. h. $s_0 \in S_0$ und $\sigma_i = L(s_i)$ für $i \geq 0$. Dann ist jedes Präfix von π eindeutig bestimmt.*

Beweis. Induktion über die Länge i des Präfixes:

- Da M deterministisch ist, gibt es genau einen Anfangszustand, d. h. s_0 ist eindeutig bestimmt.
- Sei das i-te Präfix $s_0 s_1 \ldots s_i$ eindeutig bestimmt. Da M deterministisch ist, gilt: Aus $R(s_i, t)$ und $R(s_i, t')$ folgt $L(t) \neq L(t')$. Das heißt, es kann höchstens einen Nachfolgezustand s_{i+1} von s_i geben mit $L(s_{i+1}) = \sigma_{i+1}$. Einen solchen Zustand muß es aber geben, sonst wäre σ nicht in $\mathfrak{L}(M)$.
  Also ist auch s_{i+1} eindeutig.

□

Theorem 3. *Für deterministische Kripke-Strukturen M und M' gilt:*

$$M \equiv^B M' \quad \Leftrightarrow \quad \mathfrak{L}(M) = \mathfrak{L}(M').$$

Beweis. Wir zeigen die beiden Richtungen der Behauptung:

„⇒“ Gelte $M \equiv^B M'$ und sei $\sigma = \langle \sigma_0\sigma_1\sigma_2 \ldots \rangle \in \mathfrak{L}(M)$. Dann gibt es einen Pfad $\pi = s_0 s_1 s_2 \ldots$ mit $s_0 \in S_0$ und $\sigma_i = L(s_i)$ für $i \geq 0$.
Wegen $M \equiv^B M'$ gibt es eine Bisimulation B zwischen M und M', so daß gilt:
- $\exists s'_0 \in S'_0 : B(s_0, s'_0)$.
- Es gibt einen zu π korrespondierenden Pfad $\pi' = s'_0 s'_1 s'_2 \ldots$, d. h. $B(s_i, s'_i)$ für alle $i \geq 0$ und $s'_0 \in S'_0$.
- Nach Definition einer Bisimulation gilt dann $\sigma_i = L(s_i) = L'(s'_i)$.

Folglich ist $\sigma \in \mathfrak{L}(M')$. Die andere Inklusion zeigt man analog.

„⇐“ Gelte $\mathfrak{L}(M) = \mathfrak{L}(M')$. Wir konstruieren eine Bisimulation B zwischen M und M':
Gelte $B(s, s')$ genau dann, wenn es ein $\sigma = \langle \sigma_0\sigma_1\sigma_2 \ldots \rangle \in \mathfrak{L}(M)$ gibt, so daß gilt: Es gibt einen Pfad $\pi = s_0 s_1 s_2 \ldots$ in M mit $\sigma_i = L(s_i)$ und einen Pfad $\pi' = s'_0 s'_1 s'_2 \ldots$ in M' mit $\sigma_i = L'(s'_i)$, wobei s_0 und s'_0 jeweils die (eindeutigen) Startzustände der Strukturen sind.
B ist eine Bisimulation, denn:
- Aus $B(s, s')$ folgt $L(s) = L'(s')$.
- Gelte $B(s, s')$ und $R(s, t)$. Dann gibt es Pfade $\pi = s_0 s_1 s_2 \ldots$ und $\pi' = s'_0 s'_1 s'_2 \ldots$ mit $s = s_i$ und $s' = s'_i$ und $s_0 \in S_0$ sowie $s'_0 \in S'_0$.
  Da R total ist, gibt es einen Pfad $t_0 t_1 t_2 \ldots$ mit $t_0 = t$. Dann ist

  $$\pi^* = s_0 s_1 s_2 \ldots s_i t_0 t_1 t_2 \ldots$$

  ein Pfad in M der im Anfangszustand s_0 beginnt. Es muß in M' einen entsprechenden (eindeutigen) Pfad $\pi^{*\prime} = s'_0 s'_1 s'_2 \ldots t'_0 t'_1 t'_2 \ldots$ geben, der dasselbe Wort erzeugt und bei dem $s'_i = s'$ gilt. Da $L(t'_0) = L(t_0) = L(t)$ ist, gilt: $R(s'_i, t'_0)$ und $B(t, t'_0)$.
- Die andere Richtung geht analog.

Da für die beiden Startzustände $B(s_0, s'_0)$ gilt, sind die beiden Strukturen Bisimulation-äquivalent.

□

Anmerkung 1. Für nichtdeterministische Strukturen gilt Theorem 3 nicht, wie Abbildung 4 zeigt: Die beiden Strukturen besitzen die Sprache $\mathfrak{L}(M) = \mathfrak{L}(M') = (a + b)^\omega$, aber es gibt keine Bisimulation zwischen ihnen. Zum linken Anfangszustand von M gibt es keinen äquivalenten Anfangszustand von M'. M erfüllt beispielsweise die Formel $a \rightarrow EXb$, M' dagegen nicht.

Für deterministische Kripke-Strukturen existieren effiziente Algorithmen, die testen, ob die zugehörigen Sprachen gleich sind ([3]).

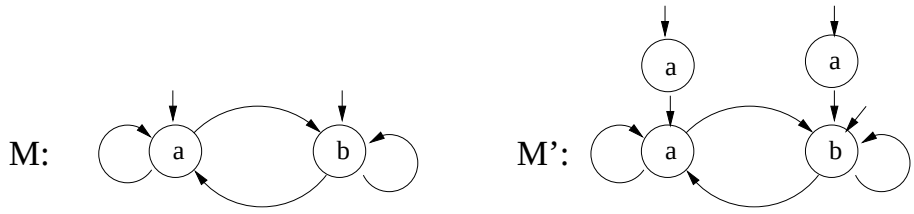


Abb. 4. Gegenbeispiel zu Anmerkung 1

1.4 Algorithmus zur Berechnung einer Bisimulation

Im folgenden wollen wir ein Verfahren angeben, das auch bei nichtdeterministischen Strukturen eine Bisimulation liefert, falls eine solche exisitiert. Ansonsten ist $B^* = \emptyset$

Seien M und M' zwei Kripke-Strukturen mit derselben Menge AP von atomaren Aussagen. Wir definieren eine Folge von Relationen $B_0^*, B_1^*, B_2^*, \ldots$ wie folgt:

1. $B_0^*(s, s') \quad \Leftrightarrow \quad L(s) = L'(s')$
2. $B_{n+1}^*(s, s') \quad \Leftrightarrow$
   - $B_n(s, s')$ und
   - $\forall s_1 \in S : \big(R(s, s_1) \Rightarrow \exists s_1' \in S' : R'(s', s_1') \wedge B_n^*(s_1, s_1')\big)$ und
   - $\forall s_1' \in S' : \big(R'(s', s_1') \Rightarrow \exists s_1 \in S : R(s, s_1) \wedge B_n^*(s_1, s_1')\big)$

Per Definition gilt für alle $i \geq 0$: $B_{i+1}^* \subseteq B_i^*$. Da M und M' endlich sind, gibt es ein n mit $B_n^* = B_{n+1}^*$. Setze $B^* := B_n^*$.

Theorem 4. *B^* ist die größte Bisimulation zwischen M und M', d. h. für jede Bisimulation B' zwischen M und M' gilt $B' \subseteq B^*$.*

Beweis. Zeigen wir zuerst, daß B^* eine Bisimulation ist.
Es gilt: $B^*(s, s') \Rightarrow L(s) = L'(s')$, da $B^* \subseteq B_0^*$.
Gelte $R(s, s_1)$ und $B^*(s, s')$. Mit $B^* = B_n^* = B_{n+1}^*$ erhalten wir: Es existiert ein $s_1' \in S'$ mit $R'(s', s_1')$ und $B^*(s_1, s_1')$.
Die dritte Bedingung geht analog zur zweiten.

Jetzt müssen wir noch zeigen, daß jede Bisimulation B zwischen M und M' in B^* enthalten ist. Dazu genügt es zu zeigen, daß B in jedem B_i^* enthalten ist. Das beweisen wir durch Induktion über i:

- $B \subseteq B_0^*$ gilt, weil per Definition gilt: $B(s, s') \Rightarrow L(s) = L'(s')$.
- Nehmen wir an, daß $B \subseteq B_i^*$ und $B(s, s')$.
  Sei $R(s, s_1)$ ein Übergang in M. Da B eine Bisimulation ist, gibt es einen Zustand s_1' mit $R'(s', s_1')$ in M' und $B(s_1, s_1')$. Da $B \subseteq B_i^*$ ist, gilt $B_i^*(s_1, s_1')$. Nach Definition von B_{i+1}^* gilt dann $B_{i+1}^*(s, s')$.

□

1.5 Algorithmus zur Minimierung einer Kripke-Struktur

Der folgende Algorithmus berechnet zu einer gegebenen Kripke-Struktur M eine zu M Bisimulation-äquivalente Kripke-Struktur M' mit einer minimalen Anzahl an Zuständen.

Definition 8. *Sei $M = (AP, S, S_0, R, L)$ eine Kripke-Struktur und $B \subseteq R \times R$ die größte Bisimulation auf M, wie sie im vorigen Abschnitt konstruiert wurde. Wir definieren die **Quotientenstruktur** $M_q = (AP, S_q, S_{0_q}, R_q, L_q)$ bezüglich B wie folgt:*

- $S_q = \{[s] \mid s \in S\}$. *Dabei bezeichnet* $[s] = \{s' \in S \mid B(s, s')\}$ *die Äquivalenzklasse von* s.
- $S_{0_q} = \{[s_0] \mid s_0 \in S_0\}$.
- $L_q([s]) = L(s)$.
- $R_q([s], [t]) \Leftrightarrow \forall s' \in [s]\, \exists t' \in [t] : R(s', t')$

Lemma 9. *Es gilt $M \equiv^B M_q$.*

Beweis. Definiere eine Relation $B' \subseteq S \times S_q$:

$$B' = \{(s, [t]) \mid s \in [t]\}$$

Wir müssen zeigen, daß B' eine Bisimulation ist. Gelte also $B'(s, [t])$. Dann folgt daraus:

- Weil $s \in [t]$ ist, gilt $B(s, t)$. Deshalb muß auch $L(s) = L(t)$ sein und damit $L(s) = L_q([t])$.
- Sei $R(s, s_1)$ ein Übergang in M. Wir müssen zeigen, daß $R_q([s], [s_1])$ ein Übergang in M_q ist.
  Sei $s' \in [s]$. Dann gilt $B(s, s')$. Nach Definition einer Bisimulation gilt für alle Nachfolger s_1 von s, daß es einen Nachfolger s'_1 von s' gibt mit $B(s_1, s'_1)$, d. h. $s'_1 \in [s_1]$. Folglich gibt es zu jedem Zustand in $[s]$ einen Nachfolger, der in $[s_1]$ enthalten ist. Somit gilt $R_q([s], [s_1])$.
- Sei $R_q([s], [s_1])$ ein Übergang in M_q. Dann gilt nach Definition von R_q:

  $$\forall t \in [s]\, \exists t_1 \in [s_1] : R(t, t_1).$$

  Da $s \in [s]$ ist, gibt es also ein $s' \in [s_1]$ mit $R(s, s')$ und $B'(s', [s_1])$.

Somit ist B' eine Bisimulation zwischen M und M_q. Wegen $S_{0_q} = \{[s_0] \mid s_0 \in S_0\}$ und $B'(s_0, [s_0])$ sind M und M_q Bisimulation-äquivalent. □

Im nächsten Abschnitt wollen wir eine entsprechende Äquivalenzrelation wie hier auf fairen Kripke-Strukturen definieren und einige analoge Aussagen wie bei normalen Kripke-Strukturen herleiten.

1.6 Kripke-Strukturen mit Fairness-Constraints

Bevor wir mit der Definition der Äquivalenzrelation auf fairen Kripke-Strukturen beginnen, wollen wir nochmals kurz die Definition einer fairen Kripke-Struktur wiederholen.

Definition 9. *Eine **faire Kripke-Struktur** über einer Menge AP von atomaren Aussagen ist ein 6-Tupel $M = (AP, S, S_0, R, L, F)$, wobei AP, S, S_0, R und L wie bei normalen Kripke-Strukturen definiert sind, und $F \subseteq \mathcal{P}(S)$ eine Menge von Fairness-Bedingungen ist.*
Sei $\pi = s_0 s_1 s_2 \ldots$ ein Pfad in M und sei

$$\inf(\pi) = \{s \mid s = s_i \text{ für unendlich viele } i\}.$$

*π heißt **fair**, wenn für jedes $P \in F$ gilt: $P \cap \inf(\pi) \neq \emptyset$.*

Bei der Semantik von CTL* müssen wir drei Punkte ändern:

- $M, s \vDash_F p$:⇔ Es gibt einen *fairen* Pfad, der in s beginnt und $p \in L(s)$.
- $M, s \vDash_F Ef_1$:⇔ Es gibt einen *fairen* Pfad π, der in s beginnt und $M, \pi \vDash_F f_1$ erfüllt.
- $M, s \vDash_F Af_1$:⇔ Alle *fairen* Pfade π mit Startpunkt s erfüllen $M, \pi \vDash_F f_1$.

Sonst ändert sich gegenüber der bisherigen Definition der Semantik von CTL* nichts.

Jetzt definieren wir eine Bisimulation auf fairen Kripke-Strukturen analog zum vorigen Abschnitt. Der einzige Unterschied ist, daß nur faire Pfade berücksichtigt werden.

Definition 10. *Eine Relation $B \subseteq S \times S$ heißt **faire Bisimulation** zwischen M und M' genau dann, wenn für alle $s \in S$ und alle $s' \in S'$ gilt:*
Wenn $B(s, s')$, dann gilt:

1. $L(s) = L'(s')$
2. *Zu jedem* fairen *Pfad $\pi = s_0 s_1 s_2 \ldots$ in M von $s_0 = s$ aus gibt es einen* fairen *Pfad $\pi' = s'_0 s'_1 s'_2 \ldots$ in M' mit $s'_0 = s'$, so daß für alle $i \geq 0$ gilt: $B(s_i, s'_i)$.*
3. *Zu jedem* fairen *Pfad $\pi' = s'_0 s'_1 s'_2 \ldots$ in M' von $s'_0 = s'$ aus gibt es einen* fairen *Pfad $\pi = s_0 s_1 s_2 \ldots$ in M mit $s_0 = s$, so daß für alle $i \geq 0$ gilt: $B(s_i, s'_i)$.*

Definition 11. *Zwei faire Kripke-Strukturen M und M' heißen **fair Bisimulation-äquivalent** ($M \equiv_F^B M'$), wenn es eine faire Bisimulation B zwischen M und M' gibt mit*

1. $\forall s_0 \in S_0 \exists s'_0 \in S'_0 : B(s_0, s'_0)$
2. $\forall s'_0 \in S'_0 \exists s_0 \in S_0 : B(s_0, s'_0)$

Analog zur Bisimulation ohne Fairness gilt auch hier das folgende Theorem:

Theorem 5. *Wenn $M \equiv_F^B M'$ gilt, dann gilt für jede CTL*-Formel f, die auf fairen Pfaden interpretiert wird:*

$$M \vDash_F f \quad \Leftrightarrow \quad M' \vDash_F f.$$

Beweis. Der Beweis verläuft analog zum Beweis von Theorem 1. □

Die Umkehrung von Theorem 5 gilt auch hier: Erfüllen zwei faire Strukturen M und M' dieselben CTL*-Formeln, so gilt $M \equiv_F^B M'$. Dies gilt sogar dann, wenn sie dieselben CTL-Formeln erfüllen.

Haben wir zwei Kripke-Strukturen M und M' gegeben (z. B. Spezifikation und Implementierung eines Systems), von denen wir nachweisen können, daß sie Bisimulation-äquivalent sind, dann wissen wir, daß sie dieselben Eigenschaften besitzen, d. h. wenn eine der beiden Strukturen korrekt ist, dann auch die andere. Beim Nachweis von Eigenschaften können wir uns also auf die kleinere Struktur beschränken.

Lemma 10. *Zwei deterministische Strukturen sind fair Bisimulation-äquivalent genau dann, wenn ihre Sprachen im Hinblick auf faire Pfade gleich sind.*

2 Anordnungen von Kripke-Strukturen

2.1 Kripke-Strukturen ohne Fairness-Bedingungen

Leider stellt es sich heraus, daß der Zustandsraum alleine mit Hilfe einer Bisimulation oft nicht genügend verkleinert werden kann. Deshalb müssen wir zum schwächeren Begriff der *Simulation* greifen.

Definition 12. *Gegeben seien zwei Kripke-Strukturen M und M' mit $AP \supseteq AP'$. Eine Relation $H \subseteq S \times S'$ heißt **Simulation** zwischen M und M', wenn für alle $s \in S$ und $s' \in S'$ gilt:*
Aus $H(s, s')$ folgt:

1. $L(s) \cap AP' = L'(s')$
2. $\forall s_1 \in S : \big(R(s, s_1) \Rightarrow \exists s_1' \in S' : R'(s', s_1') \wedge H(s_1, s_1')\big)$.

Definition 13. *M' **simuliert** M ($M \preceq M'$), wenn es eine Simulation H zwischen M und M' gibt mit*

$$\forall s_0 \in S_0 \exists s_0' \in S_0' : H(s_0, s_0').$$

Beispiel 3. Betrachte Abbildung 5. Für die beiden Strukturen M und M' gilt $M \preceq M'$. Die zugehörige Simulation ist:

$$H_1 = \{(1,1), (2,2), (3,2), (4,4), (5,5), (6,4)\} \subseteq S \times S'$$

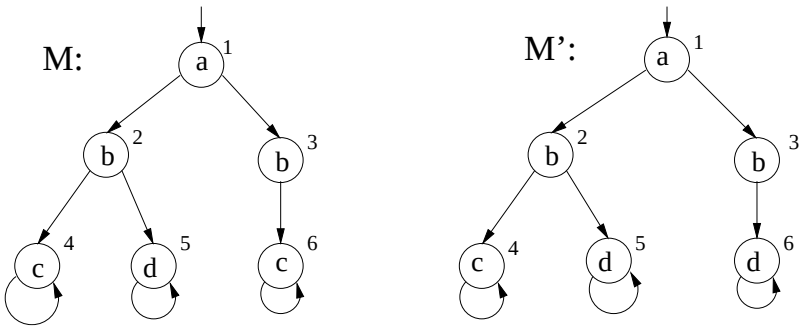


Abb. 5. Beispiel für zwei Strukturen mit $M \preceq M'$

Außerdem gilt auch $M' \preceq M$ mit der folgenden Simulation:

$$H_2 = \{(1,1),(2,2,),(3,2),(4,4),(5,5),(6,5)\} \subseteq S' \times S$$

Man sieht aber leicht, daß trotzdem nicht $M \equiv^B M'$ gilt, weil es keine Bisimulation zwischen M und M' geben kann, da es jeweils zum Zustand 3 keine Entsprechung in der anderen Struktur gibt.

Im folgenden Lemma zeigen wir, daß die oben definierte Relation $\preceq$ eine partielle Ordnung auf der Menge der Kripke-Strukturen darstellt, d. h. reflexiv und transitiv ist.

Lemma 11. *Die Relation $\preceq$ ist reflexiv und transitiv.*

Beweis. Wir weisen die beiden Eigenschaften nach:

- Reflexivität:
  Definiere $H := \{(s,s) \mid s \in S\}$. Dann ist H eine Simulation zwischen M und M, d. h. $M \preceq M$.
- Transitivität:
  Gelte $M \preceq M'$ und $M' \preceq M''$. Sei H_0 eine Simulation zwischen M und M' und H_1 eine Simulation zwischen M' und M''. Setze dann

  $$H_2 := \{(s,s'') \mid \exists s' : H_0(s,s') \wedge H_1(s',s'')\}.$$

  Damit gilt:
  - $H_2(s_0, s_0'')$, denn $s_0 \in S_0$. Deshalb gibt es ein $s_0' \in S_0'$ mit $H_0(s_0, s_0')$, da $M \preceq M'$. Wegen $M' \preceq M''$ gibt es ein $s_0'' \in S_0''$ mit $H_1(s_0', s_0'')$.
  - Gelte $H_2(s,s'')$ und sei s' so, daß $H_0(s,s')$ und $H_1(s',s'')$ gilt. Per Definition einer Simulation gilt dann:
    $L(s) \cap AP' = L'(s')$ und $L'(s') \cap AP'' = L''(s'')$. Weil $AP \supseteq AP' \supseteq AP''$, folgt $L(s) \cap AP'' = L''(s'')$.
  - Sei $R(s,s_1)$ ein Übergang in M. Dann gibt es einen Übergang $R'(s',s_1')$ in M' mit $H_0(s_1,s_1')$. Da H_1 eine Simulation ist, gibt es einen Übergang $R''(s'',s_1'')$ in M'' mit $H_1(s_1',s_1'')$. Damit gilt auch $H_2(s_1,s_1'')$.

Also ist H_2 eine Simulation zwischen M und M''. □

Definition 14. *Seien $\pi = s_0 s_1 s_2 \ldots$ und $\pi' = s'_0 s'_1 s'_2 \ldots$ zwei Pfade. Sie heißen **korrespondierend** genau dann, wenn für jedes $i \geq 0$ gilt: $H(s_i, s'_i)$.*

Lemma 12. *Seien s und s' zwei Zustände mit $H(s, s')$. Dann gibt es für jeden in s beginnenden Pfad einen korrespondierenden Pfad, der in s' beginnt.*

Beweis. Sei $\pi = s_0 s_1 s_2 \ldots$ ein Pfad mit $s_0 = s$. Wir konstruieren induktiv einen korrespondierenden Pfad $\pi' = s'_0 s'_1 s'_2 \ldots$ von $s' = s'_0$ aus.
Nach Voraussetzung gilt $H(s, s')$. Setzen wir also $s'_0 := s'$.
Sei s'_i für ein $i \geq 0$ schon definiert. Nach Definition einer Simulation folgt aus $R(s_i, s_{i+1})$, daß es ein t' gibt mit $R'(s'_i, t')$ und $H(s_{i+1}, t')$. Setze also $s'_{i+1} := t'$. □

Wir würden jetzt gerne für Simulationen ein entsprechendes Theorem wie Theorem 1 für Bisimulationen herleiten, d. h. wir würden gerne zeigen, daß gilt: $M \preceq M' \Rightarrow (M' \vDash f \Rightarrow M \vDash f)$ für jede CTL*-Formel f. Leider stellt sich heraus, daß dies so nicht richtig ist, sondern daß wir uns auf ACTL*-Formeln beschränken müssen.

Die Teillogik ACTL* entsteht aus der Logik CTL*, indem man auf den Existenzquantor E für Pfade verzichtet. Damit dies eine tatschliche Einschränkung darstellt, muß man auch die Negation $\neg$ einschränken: Es dürfen in ACTL* nur atomare Aussagen negiert werden. Ansonsten könnte eine Formel Ef durch $\neg A \neg f$ ausgedrückt werden.

Um zu verdeutlichen, daß die Einschränkung auf ACTL* tatsächlich nötig ist, betrachten wir nochmals Abbildung 5. Es gilt $M \preceq M'$ und $M' \preceq M$, wie wir bereits in Beispiel 3 gezeigt haben. Würde obige Behauptung gelten, dann müßten beide Strukturen dieselben CTL*-Formeln erfüllen, wären also Bisimulation-äquivalent. Man sieht jedoch leicht ein, daß M die Formel $AG(b \rightarrow EXc)$ erfüllt, M' jedoch nicht. Außerdem erfüllt M' die Formel $AG(b \rightarrow EXd)$, die in M nicht erfüllt ist.

Theorem 6. *Sei $M \preceq M'$. Dann gilt für jede ACTL*-Formel f mit atomaren Aussagen in AP':*

$$M' \vDash f \quad \Rightarrow \quad M \vDash f.$$

Beweis. Es genügt, folgende Behauptung zu zeigen: Gelte $H(s, s')$ und seien π und π' korrespondierende Pfade. Dann gilt für jede ACTL*-Zustands- oder Pfadformel f:

$$M', s' \vDash f \Rightarrow M.s \vDash f, \text{ falls } f \text{ Zustandsformel}$$

bzw.

$$M', \pi' \vDash f \Rightarrow M, \pi \vDash f, \text{ falls } f \text{ Pfadformel}$$

Da der Beweis durch Induktion über den Aufbau von ACTL*-Formeln zu einem großen Teil dem Beweis von Lemma 2 entspricht, zeigen wir nur einige Punkte, bei denen sich Unterschiede ergeben.

– $f = p$ mit $p \in AP'$. Es ist

$$\begin{aligned} M', s' \vDash p &\Leftrightarrow p \in L'(s') \Leftrightarrow p \in L(s) \cap AP' \\ &\Leftrightarrow p \in L(s), \text{ da } p \in AP' \\ &\Leftrightarrow M, s \vDash p \end{aligned}$$

– $f = \neg p$ mit $p \in AP'$.

$$\begin{aligned} M', s' \vDash \neg p &\Rightarrow p \notin L'(s') \Rightarrow p \notin L(s) \cap AP' \\ &\Rightarrow p \notin L(s), \text{ da } p \in AP' \\ &\Rightarrow M, s \vDash \neg p \end{aligned}$$

– $f = f_1 \vee f_2$ und $f = f_1 \wedge f_2$ gehen analog wie in Lemma 2.
– $f = Af_1$. Aus $M', s' \vDash Af_1$ folgt, daß jeder Pfad π' in M' mit Startpunkt s' die Formel f_1 erfüllt, d. h. $M', \pi' \vDash f_1$.
Sei π ein beliebiger Pfad in M, der in s beginnt. Dann gibt es einen korrespondierenden Pfad π' in M' mit Start in s'. Dieser erfüllt f_1 und damit erfüllt nach Induktionsvoraussetzung auch π f_1. Also gilt: $M, s \vDash Af_1$.

Die übrigen Punkte mit Ausnahme von $f = Ef_1$ zeigt man wie in Lemma 2. □

Jetzt können wir auch sehen, warum der Existenzquantor für Pfade nicht zugelassen werden kann. Nehmen wir an, in M' würde die Formel $f = Ef_1$ in einem Zustand s' gelten. Dann gibt es in M' einen Pfad π', der in s' beginnt und $M', \pi' \vDash f_1$ erfüllt. Sei s ein Zustand in M mit $H(s, s')$ und wollen zeigen, daß $M, s \vDash f$ gilt. Wir wissen zwar, daß es zu jedem Pfad π in M von s aus einen korrespondierenden Pfad in M' von s' aus gibt, aber wir wissen nicht, ob π' zu einem π korrespondiert. Das dies nicht gelten muß, sehen wir in Abbildung 5.

2.2 Algorithmen für Simulationen

Seien M und M' zwei Kripke-Strukturen mit $AP \supseteq AP'$. Wir definieren induktiv eine Folge von Relationen $H_0^*, H_1^*, H_2^* \ldots$ zwischen M und M':

1. $H_0^*(s, s') \quad \Leftrightarrow \quad L(s) \cap AP' = L'(s')$
2. $H_{n+1}^*(s, s') \quad \Leftrightarrow$
   – $H_n^*(s, s')$ und
   – $\forall s_1 \in S : \big(R(s, s_1) \Rightarrow \exists s_1' \in S' : R'(s', s_1') \wedge H_n^*(s_1, s_1')\big)$

Da M und M' endlich sind, gibt es auch hier ein n, so daß $H_n^* = H_{n+1}^*$. Setze dann $H^* := H_n^*$.

Theorem 7. *H^* ist die größte Simulation zwischen M und M', d. h. für alle Simulationen H' zwischen M und M' gilt $H' \subseteq H^*$.*

2.3 Anordnungen von fairen Kripke-Strukturen

Jetzt definieren wir eine entsprechende Anordnung für faire Kripke-Strukturen.

Definition 15. *Seien M und M' zwei faire Kripke-Strukturen. Sei außerdem $AP \supseteq AP'$. Die Relation $H \subseteq S \times S'$ ist eine* ***faire Simulation*** *zwischen M und M', wenn gilt:*
Aus $H(s, s')$ folgt:

- $L(s) \cap AP' = L'(s')$
- *Zu jedem fairen Pfad $\pi = s_0 s_1 s_2 \ldots$ in M gibt es einen fairen Pfad $\pi' = s'_0 s'_1 s'_2 \ldots$ in M' mit $H(s_i, s'_i)$ für alle $i \geq 0$.*

Wir schreiben $M \preceq_F M'$, wenn es eine faire Simulation zwischen M und M' gibt mit

$$\forall s_0 \in S_0 \exists s'_0 \in S'_0 : H(s_0, s'_0).$$

Theorem 8. *Falls $M \preceq_F M'$ gilt, dann gilt für jede ACTL*-Formel f, die auf fairen Pfaden interpretiert wird:*

$$M' \models_F f \quad \Rightarrow \quad M \models_F f.$$

Kupfermann und Vardi beweisen in [6], daß das Problem, für zwei *faire* Kripke-Strukturen M und M' zu überprüfen, ob $M \preceq_F M'$ gilt, PSPACE-vollständig ist. Hält man die Größe von M' fest und betrachtet sie als konstant, so ist das Problem auch noch PTIME-vollständig, d. h. wir haben es mit einem komplexitätstheoretisch sehr schwierigen Problem zu tun.

3 Äquivalenz bezüglich Simulation

Wir können die Simulationsrelationen aus Kapitel 2 dazu benutzen, eine weitere Äquivalenzrelation auf Kripke-Strukturen zu definieren ([2]).

Definition 16. *Zwei Kripke-Strukturen M und M' heißen* ***Simulation-äquivalent*** *($M \equiv^S M'$), wenn gilt:*

$$M \preceq M' \quad \textit{und} \quad M' \preceq M$$

Daß die so definierte Relation $\equiv^S$ eine Äquivalenzrelation ist, folgt unmittelbar aus der Definition (Symmetrie) und den Eigenschaften der Relation $\preceq$ (Reflexivität, Transitivität). Weiterhin folgt aus Theorem 6 sofort:

Korollar 2. *Seien M und M' Simulation-äquivalent, d. h. $M \equiv^S M'$. Dann gilt für jede ACTL*-Formel f:*

$$M \models f \quad \Leftrightarrow \quad M' \models f$$

Der Unterschied zwischen Simulation-äquivalent und Bisimulation-äquivalent ist also, daß bei der ersteren die beiden Kripke-Strukturen dieselben CTL*-Formeln erfüllen, bei der letzteren aber nur dieselben ACTL*-Formeln.

Wir wollen jetzt zu einer gegebenen Kripke-Struktur M eine Simulation-äquivalente Kripke-Struktur M' berechnen, die minimal viele Zustände und Übergänge besitzt. Dazu brauchen wir folgende Begriffe:

Definition 17. *Sei M eine Kripke-Struktur und H die maximale Simulation über $M \times M$. Ein Zustand s_1 heißt **kleiner Bruder** eines Zustandes s_2, genau dann, wenn es einen Zustand s_3 gibt mit*

- *$R(s_3, s_2)$ und $R(s_3, s_1)$*
- *$H(s_1, s_2)$, aber nicht $H(s_2, s_1)$.*

Definition 18. *Eine Kripke-Struktur M heißt **reduziert**, wenn gilt*

- *Es gibt keine Simulation-äquivalenten Zustände in M, d. h. keine Zustände $s_1 \neq s_2$ mit $H(s_1, s_2)$ und $H(s_2, s_1)$.*
- *Es gibt keine Zustände s_1 und s_2, so daß s_1 kleiner Bruder von s_2 ist.*
- *Alle Zustände sind vom Startzustand aus erreichbar.*

Wir können jetzt zeigen, daß zu jeder Kripke-Struktur M die reduzierte Kripke-Struktur M bis auf Isomorphie eindeutig bestimmt ist. Dies sagt das nächste Theorem aus:

Theorem 9. *Seien M und M' reduzierte Kripke-Strukturen. Dann sind die folgenden Aussagen äquivalent:*

1. *M und M' sind Simulation-äquivalent.*
2. *M und M' sind isomorph.*

Beweis. Die Richtung von 2 nach 1 ist trivial. Beweisen wir also die Richtung von 1 nach 2. Seien M und M' reduzierte Kripke-Strukturen. Mit $H_{MM'}$ bezeichnen wir die größte Simulation zwischen M und M' und mit $H_{M'M}$ die größte Simulation zwischen M' und M. Die zusammengesetzte Relation $H_{MM'M} \subseteq S \times S$ ist definiert durch:

$$H_{MM'M} = \left\{(s_1, s_2) \,|\, \exists\, s_1 \in S' : (s_1, s') \in H_{MM'} \wedge (s', s_2) \in H_{M'M}\right\}$$

Wie im Beweis von Lemma 11 zeigt man, daß $H_{MM'M}$ eine Simulation ist.
Wir definieren jetzt eine Funktion zwischen M und M' und zeigen, daß diese ein Isomorphismus ist.

Sei $f : S' \to S$ definiert wie folgt:

$$f(s') = s \Leftrightarrow H_{M'M}(s', s) \text{ und } H_{MM'}(s, s').$$

Beh.: f ist wohldefiniert, d. h. aus $f(s') = s_1$ und $f(s') = s_2$ folgt $s_1 = s_2$.
Bew.: Angenommen, f wäre keine Funktion, d. h. $s_1 \neq s_2$.
Sei $H_{MM'M}$ die zusammengesetzte Simulation. Da $H_{MM'M}$ eine Simulation ist, ist sie in der größten Simulation $H \subseteq S \times S$ enthalten. Wir zeigen, daß $(s_1, s_2) \in H_{MM'M}$ und $(s_2, s_1) \in H_{MM'M}$, was der Annahme widerspricht, daß M reduziert ist.

- Aus $f(s') = s_1$ folgt $H_{M'M}(s', s_1)$ und $H_{MM'}(s_1, s')$.
- Aus $f(s') = s_2$ folgt $H_{M'M}(s', s_2)$ und $H_{MM'}(s_2, s')$.
- Aus $H_{MM'}(s_1, s')$ und $H_{M'M}(s', s_2)$ folgt $H_{MM'M}(s_1, s_2)$.
- Aus $H_{MM'}(s_2, s')$ und $H_{M'M}(s', s_1)$ folgt $H_{MM'M}(s_2, s_1)$.

Daß die Umkehrung $f^{-1} : S \to S'$ ebenfalls eine Funktion, d. h. daß f injektiv ist, zeigt man analog.

Beh.: f ist surjektiv.
Bew.: Wir müssen zeigen, daß es zu jedem Zustand $s \in S$ einen Zustand $s' \in S'$ gibt mit $f(s') = s$. Den Beweis führen wir durch Induktion über den Abstand zum Anfangszustand. Da alle Zustände erreichbar sind, ist der Abstand beschränkt durch $|S|$.

- Induktionsanfang: Sei der Abstand zum Anfangszustand 0. Aus $M \preceq M'$ folgt $H_{MM'}(s_0, s'_0)$, und aus $M' \preceq M$ folgt $H_{M'M}(s'_0, s_0)$.
- Induktionsschritt: Nehmen wir an, die Behauptung gilt für alle Zustände s mit einem Abstand kleiner oder gleich n von einem Anfangszustand. Wir zeigen die Behauptung dann für Zustände mit Abstand $n + 1$.
  Sei t_1 ein Zustand mit Abstand $n+1$ von einem Anfangszustand. Dann gibt es einen Zustand s mit Abstand n und $R(s, t_1)$. Nach Induktionsannahme gibt es einen Zustand s' mit $f(s') = s$, d. h. $H_{MM'}(s, s')$ und $H_{M'M}(s', s)$. Nach Definition einer Simulation gibt es zu jedem Nachfolger von s, also auch für t_1, einen Nachfolger t'_1 von s' mit $H_{MM'}(t_1, t'_1)$. Jetzt müssen wir nur noch zeigen, daß auch $H_{M'M}(t'_1, t_1)$ gilt.
  Nehmen wir an, das wäre falsch. Dann folgt aus $R'(s', t'_1)$ und $H_{M'M}(s', s)$ aber, daß es ein t_2 gibt mit $R(s, t_2)$ und $H_{M'M}(t'_1, t_2)$. Wegen $H_{MM'}(t_1, t'_1)$ und $H_{M'M}(t'_1, t_2)$ folgt dann $H_{MM'M}(t_1, t_2)$. Also sind t_1 und t_2 entweder äquivalent oder t_1 ist kleiner Bruder von t_2. Dies ist ein Widerspruch zur Voraussetzung. Damit gilt $H_{M'M}(t'_1, t_1)$.

Analog dazu kann man zeigen, daß auch f^{-1} surjektiv ist. Damit haben wir also gezeigt, daß f total und bijektiv ist.

Beh.: Seien $s', t' \in S'$. Dann gilt $R'(s', t') \Leftrightarrow R(f(s'), f(t'))$.
Bew.: Wir zeigen hier nur, daß $R(s', t') \Rightarrow R(f(s'), f(t'))$ gilt. Die andere Richtung geht ähnlich.
Seien $s', t'_1 \in S'$ zwei Zustände mit $R'(s', t'_1)$ und seien $s, t_1 \in S$ mit $f(s') = s$ und $f(t'_1) = t_1$. Nehmen wir an, es würde nicht $R(s, t_1)$ gelten. Dann folgt aus $H_{M'M}(s', s)$ aber, daß es ein t_2 gibt mit $R(s, t_2)$ und $H_{M'M}(t'_1, t_2)$. Außerdem folgt aus $H_{MM'}(s, s')$, daß es ein t'_2 gibt mit $R'(s', t'_2)$ und $H_{MM'}(t_2, t'_2)$. Wir unterscheiden zwei Fälle:

1. Falls $t'_2 = t'_1$ ist, dann ist $f(t'_1) = t_2$, was einen Widerspruch zur Wohldefiniertheit von f darstellt.
2. Sonst folgt aus $H_{M'M}(t'_1, t_2)$ und $H_{MM'}(t_2, t'_2)$, daß $H_{M'MM'}(t'_1, t'_2)$ gilt. Daraus folgt wiederum daß entweder t'_1 und t'_2 äquivalent sind oder t'_1 ein kleiner Bruder von t'_2 ist. Widerspruch.

Da außerdem nach Definition einer Simulation für alle $s' \in S'$ die Gleichung $L'(s') = L(f(s'))$ gilt, ist f ein Isomorphismus zwischen M und M'. □

3.1 Minimierung bezüglich Simulationsäquivalenz

Nachdem wir gerade gezeigt haben, daß zu jeder Kripke-Struktur die reduzierte Struktur eindeutig bestimmt ist, wollen wir nun ein Verfahren angeben, wie eine Kripke-Struktur reduziert werden kann. Den Algorithmus geben wir an, ohne die Korrektheit zu beweisen. Den Beweis kann man in [2] nachlesen.

Die Reduktion verläuft in drei Schritten:

1. Berechnung der ∀-Quotientenstruktur bezüglich Simulationsäquivalenz, um äquivalente Zustände zu entfernen.
2. Die Verbindungen zu kleinen Brüdern werden aufgetrennt.
3. Alle von Startzuständen aus unerreichbaren Knoten werden gelöscht.

Wir wollen nun die ∀-Quotientenstruktur definieren.

Definition 19. *Die* ∀***-Quotientenstruktur*** $M_q = (S_q, S_{0_q}, R_q, L_q)$ *zu einer Kripke-Struktur* M *ist gegeben durch*

- $S_q = \{[s] \mid s \in S\}$, *wobei* $[s]$ *die Äquivalenzklasse bezeichnet, die* s *enthält.*
- $S_{0_q} = \{[s_0] \mid s_0 \in S_0\}$.
- $R_q = \{([s],[t]) \mid \forall s_1 \in [s]\ \exists t_1 \in [t] : R(s_1, t_1)\}$.
- $L_q([s]) = L(s)$

Die Zustände sind gerade die Äquivalenzklassen bzgl. Simulation. Es gibt einen Übergang von einer Äquivalenzklasse $[s]$ zu einer Klasse $[t]$, wenn es von jedem Element von $[s]$ einen Übergang zu einem Element von $[t]$ gibt (deshalb ∀-Quotientenstruktur).

Als nächstes löschen wir die Übergänge in M_q, die zu Zuständen führen, die kleine Brüder von anderen Zuständen sind.

Definition 20. *Sei* $M_q = (S_q, S_{0_q}, R_q, L_q)$ *die* ∀*-Quotientenstruktur zu* M*. Dann ist die Struktur* $M_b = (S_b, S_{0_b}, R_b, L_b)$ *ohne kleine Brüder definiert durch*

- $S_b = S_q$, $S_{0_b} = S_{0_q}$ *und* $L_b = L_q$.
- $R_b = R_q \setminus \{(s_1, s_2) \mid \exists s_3 : (s_1, s_2) \in R \wedge H(s_2, s_3) \wedge \neg H(s_3, s_2)\}$.

Wenn M_q und H symbolisch mit Hilfe von OBDDs repräsentiert werden, dann kann M_b effizient berechnet werden.

Im letzten Schritt werden noch diejenigen Zustände von M_b eliminiert, die von keinem Startzustand aus erreichbar sind. Auch diese Operation kann effizient mit OBDDs durchgeführt werden.

Lemma 13. *Der obige Algorithmus konstruiert zu einer Kripke-Struktur* M *eine reduzierte Kripke-Struktur* M'*, so daß gilt:*

- $M \equiv^S M'$

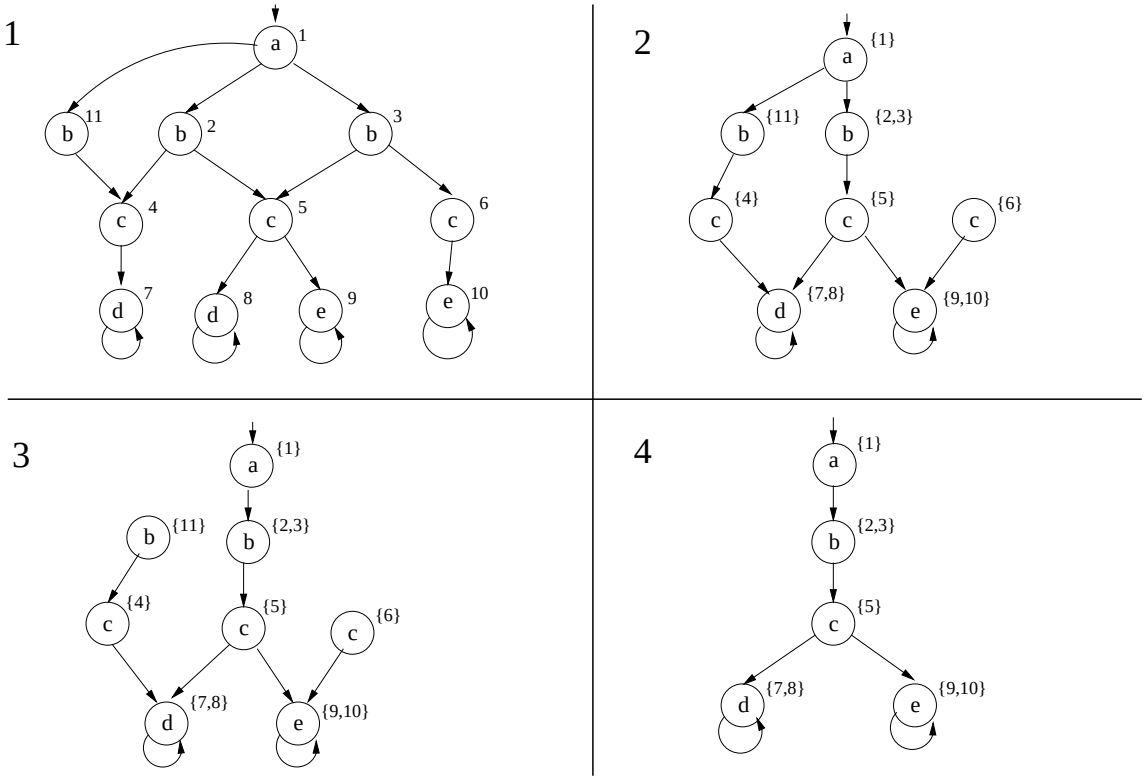


Abb. 6. Minimierung einer Kripke-Struktur

– *M' besitzt eine minimale Anzahl an Knoten und Übergängen.*

Beweis. siehe [2]. □

Die Minimierung wollen wir jetzt an einem Beispiel verdeutlichen:

Beispiel 4. Abbildung 6 zeigt die einzelnen Schritte der Minimierung der Kripke-Struktur M, die im Teil 1 der Abbildung angegeben ist. Die maximale Simulation $H \subseteq S \times S$ ist gegeben durch

$$H = \big\{(2,3),(3,2),(11,2),(11,3),(4,5),(6,5),(7,8),(8,7),(9,10),(10,9)\big\},$$

wobei die trivialen Paare weggelassen wurden.
Die Äquivalenzklassen sind also gegeben durch:

$$\big\{\{1\},\{2,3\},\{11\},\{4\},\{5\},\{6\},\{7,8\},\{9,10\}\big\}.$$

Die $\forall$-Quotientenstruktur M_q zeigt Teil 2 der Abbildung.
Im zweiten Schritt der Minimierung stellt man fest, daß $\{11\}$ kleiner Bruder von $\{2,3\}$ mit Vater $\{1\}$ ist. Deshalb kann die Kante $(\{1\},\{11\})$ entfernt werden.
Im letzten Schritt stellen sich die Zustände $\{11\}$, $\{4\}$ und $\{6\}$ als unerreichbar heraus und werden entfernt. Die reduzierte Struktur ist in Teil 4 zu sehen.

Die Zeitkomplexität der einzelnen Schritte des Algorithmus ergibt sich folgedermassen (siehe [2]):

– Der Algorithmus muß zuerst die maximale Simulation berechnen. Dies kann in Zeit $O(|S| \cdot |R|)$ geschehen. Die Äquivalenzklassen können danach in Zeit $O(|S|^2)$ berechnet werden. Die Übergangsrelation R_q wird in Zeit $O(|S|+|R|)$ berechnet.
– Die Verbindung zu kleinen Brüdern kann in Zeit $O(|S|^3)$ aufgetrennt werden.
– Das Löschen unerreichbarer Zustände kann in Zeit $O(|R|)$ erfolgen.

Der Platzbedarf ist beschränkt durch $|S|^2$. Der kritische Schritt ist hierbei die Berechnung der größten Simulation. Bustan und Grumberg schlagen in [2] deshalb einen Algorithmus zur Berechnung der $\forall$-Quotientenstruktur vor, der ohne die Berechnung der größten Simulation auskommt, weniger Speicherplatz benötigt, aber dafür eine etwas größere Laufzeit besitzt.

Zusammenfassung

Wir wollen nochmals kurz zusammenfassen, was wir in den letzten drei Abschnitten gesehen haben.

Wir haben damit angefangen, daß wir definiert haben, was eine Bisimulation zwischen zwei Kripke-Strukturen sein soll und wann zwei Strukturen Bisimulation-äquivalent sein sollen. Davon ausgehend haben wir bewiesen, daß zwei Bisimulation-äquivalente Kripke-Strukturen dieselbe Menge von CTL*-Formeln erfüllen.

Wir haben dann den Begriff des Berechnungsbaumes eingeführt und bewiesen, daß zwei äquivalente Zustände dieselben Berechnungsbäume besitzen. Diese Bäume haben wir dazu verwendet, um Kripke-Strukturen bis auf Bisimulation-Äquivalenz mit Hilfe von CTL-Formeln zu charakterisieren. Dadurch konnten wir erkennen, daß man zwei Kripke-Strukturen, die durch eine CTL*-Formel unterschieden werden können, auch durch eine CTL-Formel unterscheiden kann.

Deterministische Kripke-Strukturen sind das Thema des nächsten Abschnittes gewesen. Wir haben gezeigt, daß zwei deterministische Kripke-Strukturen äquivalent sind genau dann, wenn sie dieselben Sprachen besitzen.

Wir haben auch zwei Algorithmen kennengelernt: Der erste gestattet uns, die größte Bisimulation zwischen zwei Strukturen zu konstruieren, der zweite, eine gegebene Kripke-Struktur zu minimieren.

Zum Abschluß des ersten Abschnitts haben wir uns mit fairer Bisimulation beschäftigt und einige analoge Resultate wie für Kripke-Strukturen ohne Fairness-Constraints angegeben.

Im zweiten Abschnitt gingen wir zum schwächeren Begriff der Simulation über. Mit Hilfe einer Simulation haben wir eine partielle Ordnung auf der Menge der Kripke-Strukturen definiert. Dabei stellte sich heraus, daß eine ACTL*-Formel, die in der „größeren“ von zwei Kripke-Strukturen gilt, auch in der „kleineren“ erfüllt ist. Ein Algorithmus gestattet uns, die größte Simulation zwischen zwei Strukturen zu konstruieren.

Den Begriff der Simulation kann man auch auf faire Kripke-Strukturen übertragen. Allerdings ist das Problem, für zwei faire Kipke-Strukturen M und M' zu entscheiden, ob $M \preceq_F M'$ gilt, komplexitätttheoretisch sehr schwierig.

Ausgehend von Simulationen haben wir in Abschnitt 3 eine weitere Äquivalenzrelation definiert. Im Vergleich zu der auf Bisimulationen aufbauenden Äquivalenzrelation aus Abschnitt 1 gestattet uns diese, den Zustandsraum stärker zu verkleinern, wenn wir uns auf ACTL*-Formeln einschränken. Ein Algorithmus, der zu einer gegebenen Kripke-Struktur eine äquivalente mit minimal vielen Zuständen berechnet, wurde angegeben.

Literatur

1. M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. In *Theoretical Computer Science 59(1–2)*, pages 115–131, 1988.
2. Doran Bustan and Orna Grumberg. Simulation based minimization. In *Conference on Automated Deduction*, pages 255–270, 2000.
3. E. M. Clarke, I. A. Draghicescu, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. In A. Arnold and N. D. Jones, editors, *Proceedings of the 15th Colloquium on Trees in Algebra and Programming, LNCS 431*, pages 103–116. Springer, 1990.
4. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999.

5. E. A. Emerson and J. Y. Halpern. „Sometimes“ and “Not Never“ revisited: on branching versus linear time temporal logic. In *Journal of the ACM, 33(1)*, pages 151–178, 1986.
6. Orna Kupfermann and Moshe Y. Vardi. Verification of fair transition systems. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 1996 Workshop on Computer-Aided Verfication, LNCS 1102*, pages 372–382. Springer, 1996.

Komposition von fairen Kripke-Strukturen

Markus Degen

Institut für Informatik
Albert-Ludwigs-Universtiät Freiburg
degen@informatik.uni-freiburg.de

Zusammenfassung. Große Systeme können nicht mehr ohne weiteres verifiziert werden, da die Zustandsmengen exponentiell anwachsen. Oft ist es aber gar nicht nötig, ein System als ganzes zu betrachten. Wir wollen statt des kompletten Systems Teilsysteme verifizieren, um danach zu zeigen, dass die zusammengesetzten Teilsysteme, wenn sie parallel ausgeführt werden, auch korrekt sind. Hierzu bedienen wir uns der Komposition von Kripke-Strukturen und beweisen mittels Assume-Garantee-Formeln und eines Kalküls. Um dieses Paradigama verwenden zu können, brauchen wir eine Möglichkeit aus ACTL-Formeln ein Tableau, also eine Kripke-Struktur, zu konstruieren.

Da man Kripke-Strukturen ohne Fairnessbedingungen als ein Spezialfall von fairen Kripke-Strukturen auffassen kann ($F = \{S\}$), wollen wir uns hier auf faire Kripke-Strukturen beschränken.

Motivierendes Beispiel: Als Beispiel bedienen wir uns in diesem Kapitel einer einfachen Datenübermittlung eines Clients an den Server. Um die korrekte Datenübertragung und -verarbeitung sicherzustellen, müßte eine Kripkestruktur über den Client, das Netzwerk und den Server erstellt werden. Diese eine Struktur wäre sicher zu groß, um darauf sinnvoll arbeiten zu können, zumal viele Clients an einem Netz hängen, aber nicht alle betrachtet werden müssen. Wir wollen also statt dessen jede Komponente einzeln betrachten, hier also den Client, das Netzwerk und den Server jeweils als eigenständige Kripke-Struktur. Nachdem die Korrektheit der einzelnen Systeme gesichert ist (vorige Kapitel), wollen wir nun versuchen zu beweisen, dass diese korrekten Einzelsysteme mit bestimmten Voraussetzungen und Annahmen auch zusammen korrekt funktionieren.

1 Vereinigung von Kripke-Strukturen

In diesem Abschnitt wollen wir die Vereinigung kleinerer Strukturen zu einer großen Kripke-Struktur formal einführen. Um die einzelnen Komponenten zusammenführen zu können, benötigen wir die Möglichkeit einer parallelen Vereinigung. Wir wollen später vermeiden, diese Komposition auszuführen, benötigen

sie aber, um zu beweisen, dass die parallele Ausführung der Komponenten korrekt funktioniert.

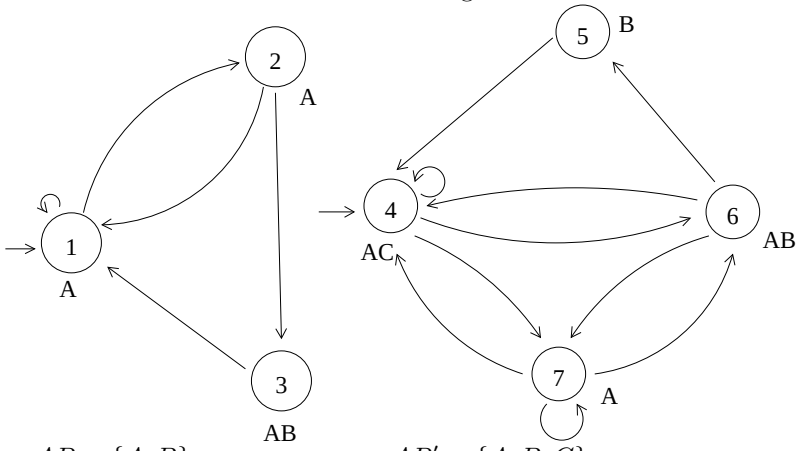
Definition 1. *Seien $M = (S, S_0, AP, L, R, F)$ und $M' = (S', S'_0, AP', L', R', F')$ zwei Kripke-Strukturen. Dann ist die* ***parallele Vereinigung*** *von M und M', geschrieben $M||M'$, folgende Kripke-Struktur M'':*

1. $S'' = \{(s, s')|L(s) \cap AP' = L'(s') \cap AP\}$.
2. $S''_0 = (S_0 \times S'_0) \cap S''$.
3. $AP'' = AP \cup AP'$.
4. $L''((s, s')) = L(s) \cup L'(s')$.
5. $R''((s, s'), (t, t')) \Leftrightarrow R(s, t) \wedge R'(s', t')$.
6. $F'' = \{(P \times S') \cap S''|P \in F\} \cup \{(S \times P') \cap S''|P' \in F'\}$.

Diese Definition modelliert ein synchrones Verhalten zweier Kripkestrukturen. Zudem ist sie relativ einfach. Lediglich die Definition der Fairnessbedingung F'' scheint etwas komplizierter. F'' muß sicherstellen, dass die Fairnessbedingungen von M und M' gewahrt bleiben. Dies ist genau dann der Fall, wenn jeder faire Pfad in $M||M'$ reduziert auf M oder M' für sich schon fair ist.

1.1 Beispiel zur parallelen Komposition

Hier ein Beispiel, um die obige Definition zu verdeutlichen. Es sollen die folgenden zwei Strukturen M und M' vereinigt werden:



$AP = \{A, B\}$
$F = \{\{3\}\}$

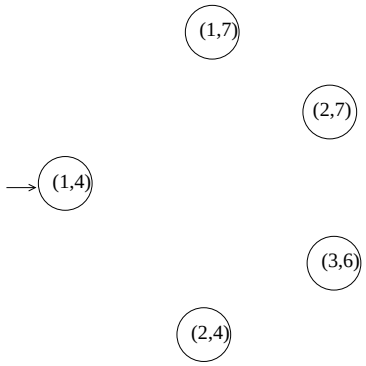
$AP' = \{A, B, C\}$
$F' = \{\{6, 4\}, \{7\}\}$

Die Zustandsmenge S'' ergibt sich aus obiger Definition. Hierbei muß beachtet werden, dass die Zustände nur dann verschmelzen, wenn die Schnittmengen der Beschriftung mit der jeweils anderen atomaren Aussagen (AP) identisch sind.

$$S'' = \{(1, 4), (1, 7), (2, 4), (2, 7), (3, 6)\}$$

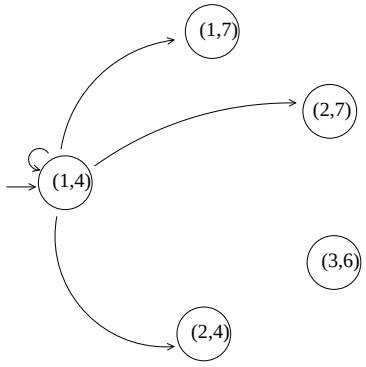
Daraus folgt auch die Menge der Startzustände:

$$S_0'' = \{(1,4)\}$$

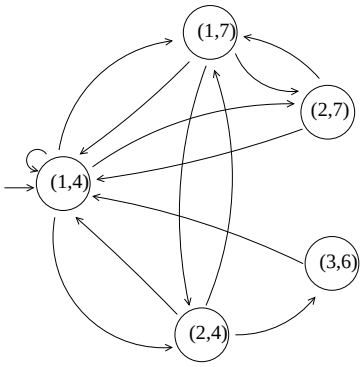


Die Übergangsrelation R'' ergibt sich aus den Übergängen der beiden ursprünglichen Strukturen. Hier als Beispiel alle Übergänge von dem Startzustand $(1,4)$ aus. In R gibt es für den Startzustand folgende Übergänge: $R(1,1)$ und $R(1,2)$. In R' existieren $R'(4,4)$, $R'(4,6)$ und $R'(4,7)$. Es würden also folgende Übergänge entstehen: $R''((1,4),(1,4))$, $R''((1,4),(1,6))$, $R''((1,4),(1,7))$, $R''((1,4),(2,4))$, $R''((1,4),(2,6))$ und $R''((1,4),(2,7))$. Davon werden aber natürlich nur solche berücksichtigt, deren Zustande auch in S'' existieren.

$$R''((s,s'),(t,t')) \Leftrightarrow R(s,t) \wedge R'(s't').$$

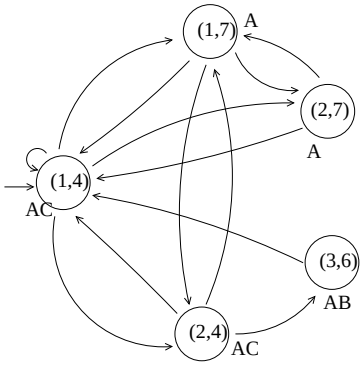


Für die anderen Zustände vervollständigt ergibt sich



L'' folgt aus der Vereinigung der Beschriftungen der ursprünglichen Beschriftungen, also

$$L''((s, s')) = L(s) \cup L'(s').$$



Fehlen noch die atomaren Aussagen (Vereinigung von AP und AP')

$$AP'' = \{A, B, C\}$$

und die Fairnessbedingungen

$$F'' = \{(P \times S') \cap S'' | P \in F\} \cup \{(S \times P') \cap S'' | P' \in F'\}.$$

Für dieses Beispiel also

$$F'' = \{\{(3,6)\}, \{(3,6),(1,4),(2,4)\}, \{(1,7),(2,7)\}\}$$

1.2 Eigenschaften der parallelen Komposition:

Wir beginnen mit den Basiseigenschaften dieser Definition.

Theorem 1. *Die oben vorgestellte Komposition ist* ***kommutativ*** *und* ***assoziativ.***

Beweis. Exemplarisch der Beweis zur Kommutativität:
Seien $M = (S, S_0, AP, L, R, F)$ und $M' = (S', S'_0, AP', L', R', F')$ zwei Kripke-Strukturen.
Z.z.: $M||M' = M'||M$.
Um eine einheitliche Benennung der Zustände zu erreichen, werden während des Beweises die Zustände (s', s) aus $M'||M$ zu (s, s') umbenannt.

$$\begin{aligned} S_{M'||M} &= \{(s', s) | L'(s') \cap AP = L(s) \cap AP'\} \\ &= \{(s, s') | L(s) \cap AP' = L'(s') \cap AP\} \\ &= S_{M||M'} \end{aligned}$$

$$\begin{aligned} S_0^{M'||M} &= (S'_0 \times S_0) \cap S_{M'||M} \\ &= (S_0 \times S'_0) \cap S_{M||M'} \\ &= S_0^{M||M'} \end{aligned}$$

$$\begin{aligned} AP_{M'||M} &= AP' \cup AP \\ &= AP \cup AP' \\ &= AP_{M||M'} \end{aligned}$$

$$\begin{aligned} L_{M'||M} &= L'(s') \cup L(s) \\ &= L(s) \cup L'(s') \\ &= L_{M||M'} \end{aligned}$$

$$\begin{aligned} R_{M'||M}((s', s), (t', t)) &\Leftrightarrow R'(s', t') \wedge R(s, t) \\ &\Longleftrightarrow \\ R_{M'||M}((s, s'), (t, t')) &\Leftrightarrow R(s, t) \wedge R'(s', t') \\ &\Longleftrightarrow \\ &\quad R_{M||M'}((s, s'), (t, t')) \end{aligned}$$

$$\begin{aligned} F_{M'||M} &= \{(P' \times S) \cap S_{M'||M} | P' \in F'\} \cup \{(S' \times P) \cap S_{M'||M} | P \in F\} \\ &= \{(S \times P') \cap S_{M'||M} | P' \in F'\} \cup \{(P \times S') \cap S_{M'||M} | P \in F\} \\ &= \{(P \times S') \cap S_{M||M'} | P \in F\} \cup \{(S \times P') \cap S_{M||M'} | P' \in F'\} \\ &= F_{M||M'} \end{aligned}$$

Somit gilt $M||M' = M'||M$. Der Beweis zur Assoziativität verläuft genauso direkt. □

Das folgende Lemma verdeutlicht und beweist die Verbindung der Fairnessbedingungen. Ein Pfad in $M||M'$ ist also genau dann fair, wenn er reduziert auf M und M' auch fair ist.

Lemma 1. *Sei $M'' = M||M'$. Dann sind die folgenden beiden Aussagen äquivalent.*

(1) $\pi'' = (s_0, s'_0), (s_1, s'_1), ...$ ist ein fairer Pfad in M''.
(2) $\pi = s_0, s_1, ...$ und $\pi' = s'_0, s'_1, ...$ sind faire Pfade in M und M' und (s_i, s'_i) ist ein Zustand von M'' für alle i.

Beweis. Es gelte (1). Durch die Definition der parallelen Vereinigung ist $\pi = s_0, s_1, ...$ ein Pfad in M. Sei $(P, Q) \in F$ und $\inf(\pi) \cap P \neq \emptyset$. Dann ist $(P'', Q'') = ((P \times S') \cap S'', (Q \times S') \cap S'') \in F''$ und $\inf(\pi'') \cap P'' \neq \emptyset$. Durch die Definition der fairen Pfade ist $\inf(\pi'') \cap Q'' \neq \emptyset$. Also ist $\inf(\pi) \cap Q \neq \emptyset$ und π ein fairer Pfad in M. Auf die gleiche Weise läßt sich zeigen, dass $\pi' = s'_0, s'_1, ...$ ein fairer Pfad in M' ist.
Es gelte (2). Aus der Definition der parallelen Vereinigung folgt, dass $\pi'' = (s_0, s'_0), (s_1, s'_1)...$ ein Pfad in M'' ist. Sei nun $(P'', Q'') \in F''$ und $\inf(\pi'') \cap P'' \neq \emptyset$. So gilt entweder $(P'', Q'') = ((P \times S') \cap S'', ((Q \times S') \cap S'')$ für ein $(P, Q) \in F$ oder $(P'', Q'') = ((S \times P') \cap S'', ((S \times Q') \cap S'')$ für ein $(P', Q') \in F'$. Im ersten Fall ergibt sich $\inf(\pi) \cap P \neq \emptyset$ und somit $\inf(\pi) \cap Q \neq \emptyset$. Das impliziert $\inf(\pi'') \cap Q'' \neq \emptyset$. Der zweite Fall ist analog. Daraus folgt, dass π'' ein fairer Pfad in M'' ist. Siehe auch [1] □

Es folgen drei Theoreme, die die Verbindung der parallelen Komposition und der Ordnungsrelation $\preceq$ beschreiben.

Theorem 2. *Seien M und M' zwei Kripke-Strukturen, dann gilt $M||M' \preceq_F M$.*

Beweis. Sei S'' die Zustandsmenge von $M||M'$. Dann definieren wir H wie folgt:

$$H = \left\{((s, s'), s) \,\middle|\, (s, s') \in S''\right\}$$

Falls (s_0, s'_0) ein Startzustand von $M||M'$ ist, so ist $s_0 \in S_0$. Die Beschriftung von (s, s') ist gegeben durch $L(s) \cup L(s')$. Außerdem gilt $(L(s) \cup L(s')) \cap AP = L(s)$. Wenn nun $(s_0, s'_0), (s_1, s'_1), ...$ ein fairer Pfad in $M||M'$ ist, so ist nach Definition $s_0, s_1, ...$ ein fairer Pfad in M. Durch die Definition von H gilt $H((s_i, s'_i), s_i)$ für alle i. Folglich ist H eine Simulation und somit gilt $M||M' \preceq_F M$. □

Dieses Theorem verdeutlicht, dass die parallele Komposition $M||M'$ nur M in seinem Verhalten einschränkt. Dies wird intuitiv klar, wenn man sich vorstellt, dass die Funktionalität des Netzwerks egal ist, wenn nicht mindestens der Client korrekt funktioniert. Damit ist dieses Theorem das eigentliche Kernstück der parallelen Vereinigung. Wir können daraus folgern, dass es genügt zuerst über M allein zu argumentieren anstatt sofort die gesamte Vereinigung $M||M'$ zu betrachten.

Theorem 3. *Seien M, M' und M'' drei Kripke-Strukturen und es gelte $M \preceq_F M'$, dann gilt $M||M'' \preceq_F M'||M''$.*

Beweis. Sei H_0 die Simulation von M auf M'. Dann definieren wir

$$H_1 = \big\{((s,s''),(s',s'')) \,\big|\, H_0(s,s')\big\}$$

Wir müssen zeigen, dass H_1 eine Simulation von $M||M''$ auf $M'||M''$ ist.
Sei (s_0,s_0'') ein Startzustand von $M||M'' \Rightarrow s_0 \in S_0$ und $s_0'' \in S_0''$. Da $M \preceq_F M'$, gibt es einen Zustand $s_0' \in S_0'$ mit $H_0(s_0,s_0')$. Nun folgt aus

$$\begin{aligned} L'(s_0') \cap AP'' &= (L(s_0) \cap AP') \cap AP'' \\ &= (L(s_0) \cap AP'') \cap AP' \\ &= (L''(s_0'') \cap AP) \cap AP' \\ &= L''(s_0'') \cap AP', \end{aligned}$$

dass (s_0',s_0'') ein Zustand von $M'||M''$ ist. Dieser Zustand ist nach der Definition auch ein Startzustand. Nach unserer Definition von H_1 gilt somit auch $H_1((s_0,s_0''),(s_0',s_0''))$.
Es gelte $H_1((s,s''),(s',s''))$. Dann ist

$$\begin{aligned} (L(s) \cup L''(s'')) \cap (AP' \cup AP'') &= (L(s) \cap AP') \cup (L(s) \cap AP'') \\ &\quad \cup (L''(s'') \cap (AP' \cup AP'')) \\ &= L'(s') \cup (L''(s'') \cap AP) \cup L''(s'') \\ &= L'(s') \cup L''(s'') \end{aligned}$$

Sei $(s_0,s_0''),(s_1,s_1''),...$ ein fairer Pfad in $M||M''$ mit $(s,s'')=(s_0,s_0'')$. Dann gilt für jedes i: $L(s_i) \cap AP'' = L''(s_i'') \cap AP$. Nun ist mit Lemma 1 $\pi = s_0,s_1,...$, beginnend mit s ein fairer Pfad in M und $\pi'' = s_0'',s_1'',...$, beginnend mit s'' ein fairer Pfad in M''. Wegen $H_0(s,s')$ gibt es einen Pfad $\pi' = s_0',s_1',...$ beginnend mit s' in M', so dass für alle i gilt $H_0(s_i,s_i')$. Nach Definition gilt $L(s_i) \cap AP' = L'(s_i')$ für alle i. Auf die gleiche Weise zeigen wir $L'(s_i') \cap AP'' = L''(s_i'') \cap AP'$ für alle i. Somit ist jedes dieser (s_i',s_i'') ein Zustand in $M'||M''$. Nun gilt aber wegen der Definition von H_1: $H_1((s_i,s_i''),(s_i',s_i''))$. Mit Lemma 1 ist also $(s_0',s_0''),(s_1',s_1''),...$ ein mit (s',s'') startender Pfad, der mit dem Pfad $(s_0,s_0''),(s_1,s_1''),...$ korrespondiert. Somit ist H_1 eine Simulation von $M||M''$ auf $M'||M''$. □

Mit Hilfe dieses Theorems können wir also jede Kripke-Struktur durch eine Abstraktion ersetzen. Dieses Theorem wird klar, wenn man Theorem 2 zuhilfe nimmt. An die Abstraktion werden keinerlei Bedingungen gestellt, da sie nur die bereits vorhandenen Strukturen einschränkt. Somit existiert aber auch noch eine Simulation der parallelen Vereinigungen.

Theorem 4. *Sei M eine Kripke-Struktur, dann gilt $M \preceq_F M||M$.*

Beweis. Für jeden Zustand s in M ist (s,s) ein Zustand in $M||M$. Wir definieren

$$H = \big\{(s,(s,s)) \big| s \in S\big\}$$

Nach der Definition ist für jedes $s_0 \in S_0$ der Zustand (s_0,s_0) Startzustand in $M||M$. Zudem hat (s,s) immer die gleiche Beschriftung wie s. Mit Lemma 1 und

der Definition der parallelen Komposition gilt, dass falls $\pi = s_0, s_1, ...$ ein fairer Pfad in M ist, so ist $\pi' = (s_0, s_0), (s_1, s_1), ...$ ein fairer Pfad in $M||M$. Durch die Definition von H ergibt sich $H(s_i, (s_i, s_i))$ für alle i. Somit ist H eine Simulation und es gilt $M \preceq_F M||M$. □

Dieses dritte Theorem ist lediglich technischer Art, wir werden es später für Beweise verwenden.

2 Tableau Konstruktion für ACTL-Formeln

In diesem Kapitel wollen wir eine Tableau Konstruktion für ACTL-Formeln angeben. Wir werden zeigen, dass dieses Tableau unter der Ordnungsrelation $\preceq_F$ maximal ist. Mit dieser Eigenschaft können wir die Tableaus als Annahme für das Assume-Garantee-Paradigma verwenden. Dieses Paradigma wird später eingeführt und benötigt, um mithilfe der parallelen Komposition große Strukturen zu verifizieren. Wichtiger als die Konstruktion selbst sind hierbei vor allem die Eigenschaften der Tableau-Kronstruktion. Für dieses Kapitel sei f eine beliebige ACTL-Formel.

Um ein Tableau $\mathcal{T}_f$ von f zu konstruieren, benötigen wir zwei zusätzliche Definitionen: $el(f)$ und $sat(f)$. Hierbei ist $el(f)$ die Menge der elementaren Teilformeln von f und $sat(f)$ die Menge der Zustände, die f erfüllen.

Definition 2. *$el(f)$ definiert sich induktiv durch*

1. $el(true) = el(false) = \emptyset$
2. $el(p) = el(\neg p) = \{p\}$ *falls p eine atomare Aussage ist*
3. $el(g_1 \vee g_2) = el(g_1 \wedge g_1) = el(g_1) \cup el(g_2)$
4. $el(\boldsymbol{AX} g_1) = \{\boldsymbol{AX} g_1\} \cup el(g_1)$
5. $el(\boldsymbol{A}[g_1 \boldsymbol{U} g_2]) = \{\boldsymbol{AX} false, \boldsymbol{AX}(\boldsymbol{A}[g_1 \boldsymbol{U} g_2])\} \cup el(g_1) \cup el(g_2)$
6. $el(\boldsymbol{A}[g_1 \boldsymbol{R} g_2]) = \{\boldsymbol{AX} false, \boldsymbol{AX}(\boldsymbol{A}[g_1 \boldsymbol{R} g_2])\} \cup el(g_1) \cup el(g_2)$

Definition 3. *$sat(g)$, wobei g Teilformel von f ist, wird definiert durch*

1. $sat(true) = \mathcal{P}(el(f)) \quad [= S_T]$
2. $sat(false) = \emptyset$
3. $sat(g) = \{s \big| g \in s\}$ *falls* $g \in el(f)$
4. $sat(\neg g) = \{s \big| g \notin s\}$ *falls g eine atomare Aussage ist*
5. $sat(g \vee h) = sat(g) \cup sat(h)$
6. $sat(g \wedge h) = sat(g) \cap sat(h)$
7. $sat(\boldsymbol{A}[g \boldsymbol{U} h]) = \big(sat(h) \cup \big(sat(g) \cap sat(\boldsymbol{AX}(\boldsymbol{A}[g \boldsymbol{U} h]))\big)\big) \cup sat(\boldsymbol{AX} false)$
8. $sat(\boldsymbol{A}[g \boldsymbol{R} h]) = \big(sat(h) \cap \big(sat(g) \cup sat(\boldsymbol{AX}(\boldsymbol{A}[g \boldsymbol{R} h]))\big)\big) \cup sat(\boldsymbol{AX} false)$

Mit Hilfe dieser beider Definitionen können wir nun die Konstruktion des Tableaus $\mathcal{T}_f$ der ACTL-Formel f angeben:

Definition 4. *Das Tableau $\mathcal{T}_f$ der ACTL-Formel f wird durch folgende Kripke-Struktur $(S_T, S_0^T, AP_T, L_T, R_T, F_T)$ gegeben.*

- $S_T = \mathcal{P}(el(f))$
- $S_0^T = sat(f)$
- AP_T *besteht aus der Menge der atomaren Aussagen von* f
- L_T*, die Beschriftung wird durch die atomaren Aussagen, die der jeweilige Zustand enthält, gegeben.*
- $R_T(s_1, s_2) = \bigwedge_{\boldsymbol{AX}g \in el(f)} s_1 \in sat(\boldsymbol{AX}g) \Rightarrow s_2 \in sat(g)$
- $F_T = \big\{\big((S_T - sat(\boldsymbol{AX}\ \boldsymbol{A}[g\,\boldsymbol{U}h])) \cup sat(h)\big)\big|\boldsymbol{AX}\ \boldsymbol{A}[g\,\boldsymbol{U}h] \in el(f)\big\}$

Diese Konstruktion ist korrekt und hat die gewünschten Eigenschaften [2].

Ein einfaches aber wichtiges Lemma, dessen Beweis aus der Definition des Tableaus und dessen Korrektheitsbeweis direkt hervorgeht [2], ist folgende Eigenschaft:

Lemma 2. $\mathcal{T}_f \models_F f$

Die wichtigste Eigenschaft, die uns erst die spätere Verwendung der Tableaus in dem Assume-Garantee-Paradigma ermöglicht, beschreibt das folgende Theorem. Damit wird auch bewiesen, dass $\mathcal{T}_f$ das maximale Modell für f ist.

Theorem 5. *Für jede Struktur M gilt $M \models_F f \Leftrightarrow M \preceq_F \mathcal{T}_f$.*

Beweis. Es gelte $M \preceq_F \mathcal{T}_f$. Wegen $\mathcal{T}_f \models_F f$ (s.o.) folgt mit der Annahme direkt $M \models_F f$.
Es gelte $M \models_F f$. Dann erfüllt nach der Definition jedes $s_0 \in S_0$ f. Wir definieren eine faire Simulationsrelationrelation [2]:

$$H = \big\{(s, s_T)\big|s_T = \{g|g \in el(f), s \models g\}\big\}$$

Dadurch erhält jedes s_0 ein s_0^T. Außerdem gilt nach der Definition von $sat()$ und wegen der Simulationsrelation H: $s_0^T \in sat(f)$, und mit der Definition des Tableaus $s_0^T \in S_0^T$. Somit gilt mit H die zu beweisende Eigenschaft $M \preceq_F \mathcal{T}_f$. □

Wir können mithilfe der Tableaus auch direkt auf Formeln arbeiten. Wir schreiben $g \models f$, um auszudrücken, dass jedes Modell von g auch ein Modell von f ist. Damit ergibt sich für unsere Tableaukonstruktion direkt:

Lemma 3. $g \models f \Leftrightarrow \mathcal{T}_g \models_F f$

Beweis. Es gelte $g \models f$. Dann ist jedes Modell von g auch Modell von f, also auch $\mathcal{T}_g$.
Es gelte $\mathcal{T}_g \models_F f$. Sei $M \models_F g$. Durch Theorem 5 gilt dann $M \preceq_F \mathcal{T}_g$. Mit Theorem 6 aus Kapitel 03 gilt auch $M \models_F f$. Da dies für jedes $M \models_F g$ gilt, gilt auch $g \models f$. □

3 Asumee-Garantee-Paradigma und Kalkül

Um Einzelkomponenten in einem größeren Kontext zu betrachten, setzen wir Annahmen für einzelne Komponeten voraus und garantieren dafür, dass die Komponente mit einem gewünschten Ergebnis/Zustand terminiert. Dieses Verfahren nennt man Assume-Garantee-Paradigma. In dem Client-Netzwerk-Server Beispiel wird also bei der Verifikation des Netzwerks angenommen, dass eine korrekte Anfrage des Clients vorliegt. Dazu wird natürlich bei der Verifikation des Clients verlangt, dass nachher eine korrekte Anfrage formuliert und abgeschickt wurde.

Definition 5. *Seien g und f ACTL-Fromeln und M eine Kripke-Struktur, dann ist $\langle f \rangle M \langle g \rangle$ eine Assume-Garantee-Formel. Sie drückt aus, dass M unter der Vorgabe das f gilt, korrekt abläuft und nach dem Ablauf g erfüllt ist.*

Mit Hilfe dieser Definition können auch Ausdrücke ohne Vorgabe ($\langle true \rangle M \langle g \rangle$) oder ohne Abschlußbedingung ($\langle f \rangle M \langle true \rangle$) beschrieben werden. Um eine größere Flexibilität zu haben, werden oft auch Kripke-Strukturen als Garantie-Bedingungen zugelassen. Dies bedeutet dann, dass die zu prüfende Kripkestruktur die Garantie-Struktur simulieren muß (siehe unten).

Für unser Client/Netzwerk/Server Beispiel lassen sich also folgende einfache Formeln bilden:

1. Der Client (C) funktioniert korrekt und liefert eine sinnvolle Anfrage f:
$$\langle true \rangle C \langle g \rangle \tag{1}$$
2. Das Netzwerk (N) funktioniert, falls es eine richtige Anfrage bekommt (f) und reicht diese an den Server weiter (g):
$$\langle f \rangle N \langle g \rangle \tag{2}$$
3. Der Server (S) empfängt die Anfrage und verarbeitet sie, erfüllt also die Bedingung h:
$$\langle g \rangle S \langle h \rangle \tag{3}$$

Ziel ist es nun, mit diesen Formeln nachzuweisen, dass das komplette System auch funktionsfähig ist. In unserem Beispiel gilt es also nachzuweisen, dass man aus obigen Formeln folgende Aussage ableiten kann:

$$\langle true \rangle C||N||S \langle h \rangle \tag{4}$$

Damit wäre gewährleistet, dass die Kripke-Struktur $C||N||S$ (Client parallel zu Netzwerk und Server ausgeführt) korrekt funktioniert und die Daten verarbeitet werden.

Um aus den vorgegebenen Formeln Schlußfolgerungen zu ziehen, bedienen wir uns einen einfachen Kalküls. Die Regeln des Kalküls leiten sich direkt aus

den oben vorgestellte Theoremen und Lemmata ab. Hierbei werden vor allem Theorem die2, 3 und 4 benutzt. Um aber diese Theoreme verwenden zu können, müssen die Assume-Garantee-Formeln erst umgewandelt werden. Dies geschieht mit Hilfe der vorgestellten Tableaus. Damit ergeben sich folgende einfachen Umformungen für die Assume-Garantee-Formeln: Seien f und g ACTL-Formeln und M und A Kripke-Strukturen.

1. $\langle g \rangle\, M\, \langle f \rangle \quad \hat{=} \quad \mathcal{T}_g || M \models_F f$
2. $\langle true \rangle\, M\, \langle A \rangle \quad \hat{=} \quad M \preceq_F A$

Andere Assume-Garantee-Formeln lassen sich dann entsprechend umwandeln. Angewendet auf unser Client/Netzwerk/Server Beispiel ergibt sich:

(1)	$\langle true \rangle\, C\, \langle f \rangle$	$C \models_F f$
(2)	$\langle f \rangle\, N\, \langle g \rangle$	$\mathcal{T}_f \| N \models_F g$
(3)	$\langle g \rangle\, S\, \langle h \rangle$	$\mathcal{T}_g \| S \models_F h$
(4)	$\overline{\langle true \rangle\, C \| N \| S\, \langle h \rangle}$	$\overline{C \| N \| S \models_F h}$

Nun können wir die oben vorgestellten Theoreme verwenden, um (4) aus den gegebenen Formeln (1)-(3) herzuleiten:

1. $C \models_F f$	(1)
2. $C \preceq_F \mathcal{T}_f$	Definition der Tableaus
3. $C \| N \preceq_F T_f \| N$	2. und Theorem 3
4. $T_f \| N \models_F g$	(2)
5. $C \| N \models_F g$	4., 5. und faire Simulation
6. $C \| N \preceq_F T_g$	Definition der Tableaus
7. $C \| N \| S \preceq_F T_g \| S$	6. und Theorem 3
8. $T_g \| S \models_F h$	(3)
9. $C \| N \| S \models_F h$	7., 8. und faire Simulation

Damit haben wir nachgewiesen, dass falls die einzelnen Komponenten ihre Aufgabe korrekt ausführen, auch das komplette System korrekt funktioniert. Diesen Beweis kann man natürlich auf viele andere Szenarien ausweiten und direkt verwenden.

4 Schlußfolgerungen

Mit der parallelen Komposition unter Zuhilfenahme des vorgestellten Kalküls können wir also aneinanderhängende Teilsysteme als Ganzes verifizieren. Es sollte also, wann immer möglich, auf kleineren Systemen gearbeitet werden, so dass die Kripke-Strukturen nicht zu groß werden. Viele einfache Beweise zur parallelen Ausführung können zudem einmal exemplarisch geführt werden. Es ergibt

sich also eine enorme Zeiteinsparung, wenn das System in Teilen vorliegt, eine Verschmelzung sollte so weit wie möglich vermieden werden.

Leider ist es nicht ohne weiteres möglich, große Systeme in mehrere kleine zu zerteilen. Liegt also bereits ein System vor, muß dieses für sich verifiziert werden.

Literatur

1. O. Grumberg and D. E. Long. Model Checking and Modular Verification. In *ACM Transactions on Programming Languages ans Systems, Vol16, No 3*, pages 843–871, May 1994.
2. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, 1999.

Abstraktion

Anselm Vossen

Institut für Informatik
Albert-Ludwigs-Universität Freiburg
vossen@informatik.uni-freiburg.de

Zusammenfassung. In diesem Kapitel werden zwei Techniken zur Reduktion der Zustände in einem endlichen Automaten analysiert. Ausgangspunkt ist dabei die Kripke Struktur dieses Automaten, diese wird durch eine im Allgemeinen nicht umkehrbare Abbildung in eine Struktur überführt, die einen Automaten mit einer reduzierten Zustandszahl beschreibt. Das Ziel ist eine einfachere, also effizientere Verifikation von Bedingungen, die für den Automaten gelten. Aus logischen Formeln, die über die reduzierte Struktur gelten, muss also auf Erfüllung der Bedingungen in der Originalstruktur gefolgert werden können. Die Logiken, die hier benutzt werden, sind $CTL*$ und $ACTL*$. Nötig werden diese Techniken durch die Ausweitung der Model Checking Techniken auf neue Anwendungsgebiete und dadurch, daß die Systeme, die verifiziert werden sollen, komplexer werden. Z.B. wird Model Checking nun auch auf Software Systeme angewandt, die fast beliebig komplex werden können. Außerdem werden zum Beispiel durch VLSI Techniken Hardware Komponenten komplexer, Stichwort Moores Law. Da der Zustandsraum auch exponentiell in der Anzahl der Zustandsvariablen wächst, wird ersichtlich, daß Techniken zur Reduzierung des Zustandsraumes durch Reduktion der Zustandsvariablen gebraucht werden. Clarke stellt in [1] ein System vor, das die von ihm beschriebenen Techniken nutzt, um so auch einen Begriff von dem quantitativen Gewinn durch die Nutzung von Reduktionstechniken zu bekommen. In diesem Beitrag wird auch auf diesen praktischen Aspekt eingegangen, sowie welche Abstraktionen in der Praxis gemacht werden, um den Zustandsraum zu reduzieren.
Zuerst wird die **Cone of Influence Reduction** und dann die Technik der **Data Abstraction** beschrieben. Der Schwerpunkt liegt dabei auf der Technik der Abstraktion. Hier werden auch noch Techniken zur symbolischen Abstraktion sowie Beispiele aus der Praxis vorgestellt.

1 Cone of Influence Reduction

Die "Cone of Influence Reduction" ist ein offensichtlicher Weg, die Variablen in einem System, das durch eine Kripke-Struktur beschrieben wird, zu reduzieren. Ein Verifikationsproblem besteht in der allgemeinsten hier behandelten Form aus einer Formel f in $CTL*$, deren Wahrheitswert für ein bestimmtes System festgestellt werden soll. Dieses System sei ein synchroner Schaltkreis. Dann ist intuitiv ersichtlich, daß eine Reduktion der Struktur durch Elimination von Variablen,

die die Spezifikation nicht beeinflussen, keinen Einfluss auf den Wahrheitswert der Formel f hat. Mit dem "Cone of Influence" von V' werden nun die Menge derjeniger Variablen, die die Variablen in V' beeinflussen bezeichnet. Jetzt kann man die Kripkestruktur des Systems bezüglich der Variablen im "Cone of Influence" reduzieren.

Da wir uns auf synchrone Schaltkreise beschränkt haben, lässt sich der "Cone of Influence" einfach induktiv definieren. Denn sei V die Menge der Variablen in einem gegebenen Schaltkreis. Dann kann der Schaltkreis durch eine Menge von Gleichungen der Form

$$v_i\prime = f_i(V) \text{ für alle i } \in V, \text{ wobei } f_i \text{ eine boolsche Funktion ist}$$

beschrieben werden. Dies folgt, da man offensichtlich jeden synchronen Schaltkreis in dieser Form beschreiben kann (siehe [1, Kapitel 2]). Dann gilt für die formale Definition des "Cone of Influence":

Definition 1 ("Cone of Influence"). *Der "Cone of Influence" C von V' ist die minimale Menge von Variablen, so daß gilt:*

- $V' \subseteq C$
- *wenn für $v_l \in C$ die entsprechende Funktion f_l von v_j abhängt, dann $v_j \in C$*

Der Cone of Influence umfasst also alle Variablen, deren Änderung Einfluss auf die Variablen in der Spezifikation haben können.

Beispiel 1. Einfach veranschaulichen lässt sich dies am Beispiel eines synchronen Modulo 2 Zählers. In Abbildung 1 ist ein entsprechender Schaltkreis dargestellt. Die Gleichungen für die Variablen v_i' lauten nun (wie man am Schaltplan ablesen kann):

$$\begin{aligned} v_0' &= \neg v_0 \\ v_1' &= v_0 \otimes v_1 \\ v_2' &= (v_0 \wedge v_1) \otimes v_2 \end{aligned}$$

Daher gilt für $V = \{v_0\}$, da f_0 nur von v_0 abhängt, $C = \{v_0\}$. Für $V = \{v_1\}$ gilt $C = \{v_0, v_1\}$, da f_1 von v_0 und v_1 abhängen. Aber $v_2 \notin C$ da keine Variable, von der f_1 abhängt, die also automatisch in C ist, von v_2 abhängt. Ausserdem gilt offensichtlich für $V = \{v_3\}$, $C = \{v_0, v_1, v_3\}$. Wir sehen also, daß für die Überprüfung der Korrektheit von CTL* Formeln, in denen nur v_0 vorkommt, auch nur der Wert von v_0 im Startzustand benötigt wird, was auch offensichtlich ist.

1.1 Reduktion der Kripke Struktur bezüglich des "Cone of Influence"

Definition 2. *Sei $M = (S, S_0, R, L)$ eine Kripke Struktur für einen synchronen Schaltkreis mit den bekannten Elementen*

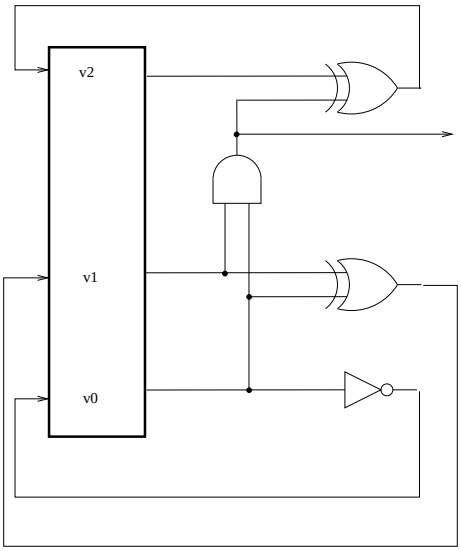


Abb. 1. Schaltplan eines Modulo 2 Zählers

1. *S ist eine endliche Menge von Zuständen*
2. *$S_0 \subseteq S$ eine Menge von Anfangszuständen*
3. *$R \subseteq S \times S$ ist eine totale Übergangsrelation gegeben als eine Konjunktion von Funktionen f_i.*
   Also $R = \bigwedge_{i=1}^{n} [v_i' = f_i(V)]$.
4. *$L : S \to \mathcal{P}(AP)$ ist eine Beschriftungsfunktion, die jedem Zustand die atomaren Aussagen, die in diesem Zustand gelten, zuordnet.*

Dann muss das Ziel sein, für die reduzierte Kripkestruktur $\hat{M} = (\hat{S}, \hat{R}, \hat{S_0}, \hat{L})$ das folgende Theorem zu zeigen:

Theorem 1. *Sei f eine CTL* Formel mit atomaren Aussagen über C. Dann $M \models f \iff \hat{M} \models f$*

Beweis. Für den Beweis wird zuerst eine geeignete Reduktionsmethode angegeben. Geeignet heißt, das man das obige Ergebnis beweisen können muss und daß die Reduktion schnell erfolgen kann. Schnelligkeit ist kein Problem, da die Reduktion vor allem durch das Weglassen von Elementen erfolgt, die Variablen enthalten, die nicht in C enthalten sind.

Es wird gezeigt, daß $M \equiv \hat{M}$ gilt, also daß M und $\hat{M}$ Bisimulation-äquivalent sind. Daraus folgt sofort das Ergebnis (wie in [1, Kapitel 11] gezeigt wird).

Definition 3 (Die reduzierte Kripkestruktur $\hat{M}$). *Angenommen die Kripkestruktur M (siehe Definition [1]) soll bezüglich eines "Cone of Influence" $C = \{v_1, \ldots, v_k\}$ für ein $k \leq n$ reduziert werden. Dann wird die reduzierte Kripkestruktur $\hat{M} = (\hat{S}, \hat{R}, \hat{S_0}, \hat{L})$ definiert durch:*

1. $\hat{S} = \{0,1\}^k$ *ist die Menge aller Belegungen von* $\{v_1, \ldots, v_k\}$
2. $\hat{R} = \bigwedge_{i=1}^{k}[v_i' = f_i(V)]$.
3. $\hat{L}(\hat{s}) = \{v_i | \hat{s}(v_i) = 1$ *für* $1 \leq i \leq k\}$.
4. $\hat{S_0} = \{(\hat{d_1}, \ldots, \hat{d_k}) |$ *es gibt einen Zustand* $(d_1, \ldots, d_n \in S_0$ *, so daß* $\hat{d_1} = d_1 \wedge \ldots \wedge \hat{d_k} = d_k\}$.

Die reduzierte Kripkestruktur entsteht also dadurch, daß nur Zustände, die sich durch Variablen, die sich im "Cone of Influence" befinden, unterscheiden, betrachtet werden. Da sich die Übergangsrelation R schon in der Form $\bigwedge_{i=1}^{n}[v_i' = f_i(V)]$ *befindet, reduziert man einfach dadurch, daß die konjugierten Ausdrücke weggelassen werden, die Zustandsübergänge von Variablen beschreiben, die nicht im "Cone of Influence" sind. Nach der Definition des "Cone of Influence" hängen die Ausdrücke der Form* $v_i' = f_i(V)$ *für* $i \leq k$ *nicht mehr von* $v_i, i > k$*, ab.*

Definition 4 (Die Bisimulationsrelation B). *Sei* $B \subseteq S \times \hat{S}$ *eine Relation zwischen den Zuständen von* M *und* $\hat{M}$ *die definiert ist durch:*

$$((d_1, \ldots, d_n), (\hat{d_1}, \ldots, \hat{d_k})) \in B \iff d_i = \hat{d_i} \text{für alle } i \leq i \leq k$$

Es wird jetzt gezeigt, daß B aus Definition 4 eine Bisimulationsrelation zwischen M und $\hat{M}$ ist und daß gilt: $\forall s \in S_0 \exists \hat{s} \in \hat{S_0} : B(s, \hat{s})$. Für jeden Zustand s in S_0 gibt es also einen Zustand in $\hat{S_0}$, mit dem s in der Relation B steht. Ferner gilt auch das Umgekehrte, d.h. für jeden Zustand $\hat{s}$ aus $\hat{S_0}$ gibt es einen Zustand aus S_0 mit dem $\hat{s}$ in der Relation B steht. Diese Stukturen sind also Bisimulation-äquivalent (für die Definition von Bisimulationsäquivalenz, siehe [1, Kapitel 11]).[1]

1. Aus der Definition von $\hat{M}$ folgt sofort, daß wenn $(d_1, \ldots, d_k) \in \hat{S_0}$ dann gibt es einen Zustand $(d_1, \ldots, d_n) \in S_0$ mit $B((d_1, \ldots, d_n), (\hat{d_1}, \ldots, \hat{d_k}))$ und vice versa.

2. Jezt muss noch gezeigt werden, daß B eine Bisimulationsrelation ist. Dazu zeigen wir im Folgenden die dazu nötigen Eigenschaften:
   (a) Es muss gelten $L(s) = \hat{L}(s)$. Da wir die Äquivalenz bezüglich Zuständen, die durch $(v_1, \ldots, v_k)$ beschrieben werden, zeigen wollen, muss gelten: $L(s) \cap C = \hat{L}(s)$. Dies folgt wiederum direkt aus der Definition von $\hat{L}$.

[1] Im Unterschied zur Bisimulationsäquivalenz, wie sie allgemein in vorherigen Beiträgen definiert worden ist, wird hier die Äquivalenz zwischen zwei Strukturen über verschiedene Mengen von Aussagenvariablen gezeigt. Dies ist aber kein Problem, da $C \subseteq \{v_1, \ldots, v_n\}$ und z.B. L in offensichtlicher Weise auf die Variablen in C eingeschränkt werden kann. Dies ist auch sinnvoll, da wir zeigen wollen, daß die beiden Strukturen M und $\hat{M}$ dieselben CTL* Formeln über Variablen in C erfüllen. Dies wird hier nur der Vollständigkeit halber erwähnt, da die Definition sonst gleichbleibt und nur darauf geachtet werden muss, daß man M in eindeutiger Weise auf die Variablen in C einschränkt.

(b) Es muss noch gezeigt werden, daß für jeden Übergang $s \to t$ in M es einen Zustand $\hat{t}$ aus $\hat{M}$ gibt, für den gilt: $B(t, \hat{t})$.
Also formal: $\forall t \in S : \big(R(s,t) \Rightarrow \exists \hat{s} \in \hat{S} : \hat{R}(\hat{s}, \hat{t}) \wedge B(t, \hat{t})\big)$
Dazu bezeichne $t = (e_1, \ldots, e_n)$. Aus der Definition von R folgt, daß alle Zustandsvariablen v_i durch $v_i' = f_i(V)$ für $1 \leq i \leq n$ beschrieben werden können. Für $1 \leq i \leq k$ hängen die v_i aber nur von Variablen in C ab, dann gilt also: $v_i' = f_i(C)$. Dann impliziert $(s, \hat{s}) \in B$, daß gilt: $e_i = f_i(d_1, \ldots, d_k) = f_i(\hat{d_1}, \ldots, \hat{d_k})$ für alle $1 \leq i \leq k$, da $\bigwedge_{i=1}^{k}(d_i = \hat{d_i})$. Sei nun $\hat{t} = (e_1, \ldots, e_k)$, dann $\hat{s} \to \hat{t}$ und $(t, \hat{t}) \in B$ wie gefordert.

(c) Umgekehrt muss gelten: $\forall \hat{t} \in \hat{S} : \big(\hat{R}(\hat{s}, \hat{t}) \Rightarrow \exists s \in S : R(s,t) \wedge B(t, \hat{t})\big)$.
Dazu: Sei $\hat{s} \to \hat{t}$ ein Übergang in $\hat{R}$. Sei $\hat{t} = (\hat{e_1}, \ldots, \hat{e_k})$. Dann gilt $\hat{e_i} = f(\hat{d_1}, \ldots, \hat{d_k})$. Außerdem gibt es $t \in S$ mit $t = (e_1, \ldots, e_k, e_{k+1}, \ldots, e_n)$ mit $\bigwedge_{i=1}^{k}(\hat{e_i} = e_i)$ (folgt aus der Reduktion). Und es gilt insgesamt:
$\hat{e_i} = f_i(\hat{d_1}, \ldots, \hat{d_k}) = f_i(d_1, \ldots, d_k) = f_i(d_1, \ldots, d_k, d_{k+1}, \ldots d_n)$. Daher ist $(t, \hat{t}) \in B$ und es gibt ein $s \in S$ (da es d_i gibt mit $f_i(d_1, \ldots, d_k) = e_i$, $\bigwedge_{i=1}^{k}(d_i = \hat{d_i})$

(d) Insgesamt haben wir nun gezeigt, daß B eine Bisimulationsrelation ist.

Mit dem Vorherigen ist der Beweis komplett. Wir haben gezeigt, daß $M \equiv \hat{M}$ und daß also für eine beliebige CTL* Formel f über C gilt:

$$M \models f \iff \hat{M} \models f$$

Die "Cone of Influence" Reduktion ist sehr wichtig bei der Verifikation von Hardware. Aus der Beschreibung der zu testenden Hardware kann relativ einfach eine effiziente Reduktion abgeleitet werden. Besonders bei komplexen und integrierten Systemen, bei denen Komponenten, also Zustandsvariablen, unabhängig voneinander sind, ist diese Art der Reduktion wichtig (siehe auch [1, Kapitel 12]). Insbesondere wenn die Zerlegung in Komponenten nicht eindeutig aus der Spezifikation ableitbar ist.

2 Daten-Abstraktion

2.1 Einführung

Systeme die komplexe Datenstrukturen beeinhalten sind schwierig mit den bisher gezeigten Techniken zu verifizieren, wenn der exakte Inhalt der komplexen Datenstruktur sichtbar ist, das heißt in dem jeweiligen Zustand kodiert wird. Dadurch wächst der Zustandsraum ja gerade exponentiell in der Anzahl der Datenfelder. Auch werden andere Arten der Reduktion des Zustandsraumes schwieriger. Systeme bei denen diese Probleme vor allem auftreten sind zum Beispiel Schaltungen, die Datenpfade enthalten und Software. Gerade die Verifikation von Software ist eine Problemstellung, die an Wichtigkeit gewonnen hat und ohne die Technik der Abstraktion unmöglich zu bewältigen wäre. Denkt man zum Beispiel an arithmetische Operationen, dann würde ein einzelnes Datenfeld mit einer 64-Bit Zahl im schlimmsten Fall zu einem 2^{64} mal so großen Zustandsraum

führen. Man beobachtet aber, daß meistens nur einfache Verhältnisse zwischen den Daten verifiziert werden sollen. Zum Beispiel kann es sein, daß nur das Vorzeichen des Datenfeldes von Interesse ist, oder aber die Größenordnung der Zahl.

Dies führt zur Idee der *Daten-Abstraktion*. Die komplexe Datenstruktur wird auf eine kleine Menge von abstrakten Werten abgebildet, die gerade die Eigenschaften, die bei der Verifikation von Interesse sind, erfassen sollen.

Dabei gehen wir von einer Beschreibung des Systems mit den Formeln $\mathcal{S}$, und $\mathcal{R}$ aus, dies sind Aussagenlogische Formeln erster Ordnung. Hier hat eine atomare Aussage die Form $x = d$ für eine Systemvariable x, die über einem Wertebereich D_x definiert ist, und einen Wert $d \in D_x$. Die Werte in D_x werden dann auf einen abstrakten Wertebereich A_x abgebildet. Das abstrahierte System wird dann durch Systemvariablen $\hat{x}$ über A_x beschrieben. Die Abbildungsfunktion nenen wir $h_x : D_x \to A_x$. h_x sei *surjektiv*.
Sei zum Beispiel D_x die Menge aller natürlichen Zahlen, wir wollen aber nur eine Eigenschaft ausdrücken, für die das Vorzeichen von x benötigt wird ($A = \{a_0, a_+, a_-\}$), dann kann die folgende Abbildung sinnvoll sein:

$$h_x(d) = \begin{cases} a_0 & \text{wenn } d = 0 \\ a_+ & \text{wenn } d > 0, \text{ und} \\ a_- & \text{wenn } d < 0. \end{cases}$$

Abstrakte Zustände werden dann mit den Aussagen über den neuen Wertebereich A_x beschriftet, die in diesem Zustand gelten. Dadurch werden also Zustände verschmolzen, die in dem abstrakten System M_r demselben Zustand entsprechen. Zustände, die im Originalsystem durch Kanten verbunden waren, werden im abstrahierten System auch durch Kanten verbunden, und die Startzustände S_0^r von M_r sind die Bilder der Startzustände des Originalsystems M. Die Konstruktion erfolgt also so, daß $M \preceq M_r$, M_r also M simuliert. Dadurch gelten dann ACTL* Formeln f, die für M_r gelten auch für M, aber nicht umgekehrt (siehe [1, Kapitel 11]). Dadurch ensteht dann das Problem der "spurious counterexamples", also das Problem, daß Formeln f', die man für M zeigen will, und die womöglich auch für M wahr sind, nicht in M_r gelten, also ein Gegenbeispiel gefunden wird. Findet man ein solches Gegenbeispiel, folgt daraus also nicht, daß f' für M nicht gilt, dazu müsste die Negation gebildet und in M_r überprüft werden. Eines der Hauptprobleme bei der Abstraktion ist also eine Abbildung auf einen abstrakten Wertebereich zu finden, der eingeschränkt genug ist, den Zustandsraum substantiell zu verkleinern, aber nicht so eingeschränkt, daß die Eigenschaften, die man für M zeigen will, nicht für M_r gelten.

Aus Effizienzgründen muss es auch möglich sein, die Abstraktion zu bilden und M_r zu berechnen, ohne vorher M explizit berechnet zu haben. Denn dies ist unter Umständen nicht möglich, etwa aus Speicherplatzgründen. Auserdem würde keine Speicherplatzersparnis das Resultat der Abstraktion sein, wenn vorher genügend Speicher gebraucht werden würde, um M zu speichern. Deshalb wird später auch noch auf die Implementation eingegangen und auf die Repräsentation als OBDD.

2.2 Konstruktion der abstrakten Struktur

Sei $M = (S, R, S_0, L)$ die Originalstruktur, M_r die reduzierte oder abstrakte Struktur. Seien die atomaren Aussagen von M über der Menge D formuliert und $h : D \to A$ eine Abbildung, die Werte in D auf den abstrakten Wertebereich A abbildet. Dann ist $L : S \to A$ die Beschriftungsfunktion, die jedem Zustand aus S die abstrakten atomaren Aussagen zuordnet, die in diesem Zustand wahr sind (im obigen Beispiel etwa $\hat{x} = a_{-}$). [2] Der leitende Gedanke bei der Konstruktion der abstrakten Struktur ist, daß die Relation $H = \{(s, s_r)|s_r = L(s))$ als Simulationsrelation genutzt werden kann. Der erste Schritt zur Konstruktion der reduzierten Struktur wurde also schon getan: Die Beschriftung der Zustände mittels der neuen Beschriftungsfunktion L, die durch h festgelegt wird.[3] Dann wird $M_r = (S_r, R_r, S_0^r, L_r)$ wie folgt festgelegt:

1. $S_r = \{L(s)|s \in S\}$. Die Menge der Zustände in M_r ist also die Menge aller Beschriftungen der Struktur M, die ja mit den abstrakten Aussagen beschriftet sind, die in dem jeweiligen Zustand wahr sind.
2. $s_r \in S_0^r$ genau dann, wenn es ein s gibt mit $s_r = L(s)$ und $s \in S_0$.
3. $L_r(s_r) = s_r$, da jeder Zustand s_r schon eine Menge von atomaren Aussagen ist.
4. $R_r(s_r, t_r)$ genau dann, wenn es Zustände s und t gibt, so daß $s_r = L(s)$, $t_r = L_(t)$, und $R(s, t)$.

Die Konstruktion ist analog zur Definition einer Simulationsrelation zwischen zwei Strukturen. Es ist also klar, daß M_r **simuliert** M **(**$M \preceq M_r$**)**. Ein Beweis dafür, der induktiv über die Struktur der ACTL* Formeln geführt wird, bringen Clarke, Grumberg und Long in [2, Theorem 5.6]. Dadurch gelten die bekannten Beziehungen zwischen M und M_r, insbesondere die Behauptung, daß für eine beliebige ACTL* Formel f über den Variablen $\hat{x_i}$ aus dem Wertebereich A gilt:

$$M_r \models f \Rightarrow M \models f$$

Man beachte, daß durch die Wahl von h bereits die Struktur M_r festgelegt ist, ausserdem, daß die Struktur M im Unterschied zur Originalstruktur schon mit der Menge der abstrakten Aussagen, die in dem jeweiligen Zustand wahr sind, beschriftet ist.

Zur Konstruktion aber zuerst noch ein Beispiel:

Beispiel 2. Betrachten wir eine Ampel an einer Kreuzung, so kann der Zustand der Ampel durch die Farbe, die sie gerade anzeigt, beschrieben werden. Das System der Ampel wird also durch die Variable *color* beschrieben. Diese kann Werte aus $D = \{green, yellow, red\}$ annehmen. Die Struktur ist in Abbildung 2 dargestellt.

Zur Modellierung am Beispiel des Verkehrsflusses kann es aber sein, daß nur von Interesse ist, ob in einem bestimmtem Zustand Autos erlaubt ist, die

[2] Die Menge der abstrakten atomaren Aussagen wird mit $AP = AP_r$ bezeichnet
[3] O.B.d.A. gehen wir von nur einer Abbildung h aus

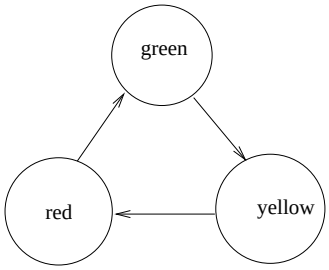


Abb. 2. Originalstruktur

Kreuzung zu befahren (Zustand *go*) oder nicht (Zustand *stop*). Die Menge der abstrakten Werte ist also $A = \{go, stop\}$ und die Abbildung h wird vollständig durch:

$$h(red) = stop; \qquad h(yellow) = stop; \qquad h(red) = stop$$

beschrieben, und die Menge der atomaren Aussagen durch:

$$AP = \{'co\hat{l}or = stop', 'co\hat{l}or = go'\}$$

Nach dem die Beschriftung der Originalstruktur nun wie beschrieben geändert wurde, sieht die Struktur also aus wie in Abbildung 3.

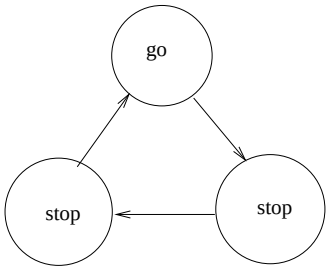


Abb. 3. Struktur M

Die Übergänge sind noch dieselben, aber man sieht schon, daß der Zustand, der mit *stop* beschriftet ist, zweimal vorhanden ist, diese also verschmolzen werden können. Um die Simulationsbeziehung zu erhalten, muss allerdings noch eine Kante von dem mit *stop* beschrifteten Zustand auf sich selbst hinzugefügt werden, da es in der Struktur M möglich war, von einem Zustand *stop* zu einem anderem, in dem dieselben Eigenschaften gelten, zu kommen. In der abstrakten Struktur (siehe Abbildung 4) sind diese Zustände nicht mehr unterscheidbar, wohl aber in der Originalstruktur.

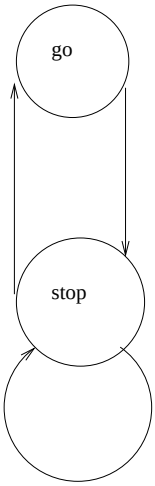


Abb. 4. Struktur M_r

Hier kann man auch dann auch das Problem der "Spurious Counterexamples", also der falschen Gegenbeispiele, illustrieren. Will man zum Beispiel zeigen, daß nach spätestens zwei Zykeln der Zustand, der mit *go* beschriftet ist, erreicht wird, so stimmt diese Aussage für die Struktur M (also die Originalstruktur, bei der die Werte der Variablen auf den abstrakten Wertebereich abgebildet wurden). In der abstrakten Struktur M_r ist es allerdings möglich, unendlich lange in dem Zustand, der mit *stop* beschriftet ist, zu bleiben.

2.3 Approximationen

Im vorherigen Beispiel wurde M explizit konstruiert, um M_r zu konstruieren. Dies ist aber nicht immer möglich. Statt dessen können die aussagenlogischen Formeln erster Ordnung $\mathcal{R}$ und $\mathcal{S}\prime$ ausgewertet werden, um daraus die Startzustände und die Übergangsrelation der abstrakten Struktur $\hat{\mathcal{R}}$ und $\hat{\mathcal{S}}\prime$ zu berechnen. Dies kann aber immer noch zu komplex sein. Stattdessen kann eine Approximation M_a an die Struktur M_r berechnet werden. Dazu werden $\hat{\mathcal{R}}$ und $\hat{\mathcal{S}}\prime$ so vereinfacht (approximiert), daß es möglich ist die Formeln einfach auszuwerten. Das Ziel bei dieser Approximation ist es, daß M_a "nah" genug an M_r ist, um immer noch interessante Eigenschaften zu verifizieren. Wir werden zeigen, daß M_a M_r simuliert, also die Ordnung $M \preceq M_r \preceq M_a$ gilt und wir, wie im vorherigen Abschnitt mit M_r, M_a zur Verifikation von ACTL* Formeln über die abstrakten Variablen $\hat{x_i}$ benutzen können.

Wir nehmen zur Vereinfachung an, daß es nur eine Abstraktionsfunktion h gibt und daß alle Variablen $x_1, x_2, \ldots$ über demselben Wertebereich D definiert sind. Dies ist aber keine Einschränkung, da verschiedene Wertebereiche zu einem vereinigt werden könnten. Die abstrakten Werte $\hat{x_1}, \hat{x_2}, \ldots$ sind über dem abstrakten Wertebereich A definiert und $\hat{x_i}$ repräsentiert den abstrakten Wert

von x_i. Dann ist der Ausgangspunkt die Kripkestruktur $M = (S, R, S_0, L)$, wie im letzten Abschnitt definiert. Die Zustandsmenge ist $S = D \times D \times \ldots \times D$ und da wir von einer Beschreibung der Struktur mit den Aussagenlogischen Formeln erster Ordnung $\mathcal{S}\prime$ und $\mathcal{R}$ ausgehen, erhalten wir S_0 als die Zustände, die $\mathcal{S}\prime$ erfüllen. R erhalten wir analog dazu aus $\mathcal{R}$ [4]. Die Definition von L folgt der aus dem letztem Abschnitt: Wenn $s = (d_1, \ldots, d_n)$ ein Zustand ist, in dem die Variable x_i den Wert d_i hat und $a_i = h(d_i)$, dann führen wir die atomare Aussage $'\hat{x_i} = a_i'$ ein. Dies heißt, daß x_i den abstrakten Wert a_i hat. Jetzt wird $L(s) = \{'\hat{x_1} = a_1', \ldots, '\hat{x_n} = a_n'\}$ definiert.
Das Ziel ist es, Formeln aufzustellen, die es uns erlauben M_r zu erzeugen. M_r ist natürlich über der abstrakten Zustandsmenge $A \times \ldots \times A$ definiert und die Formeln $\hat{S_0}$ und $\hat{R}$ über den Variablen $\hat{x_1}, \ldots, \hat{x_n}$ und $\hat{x_1}', \ldots, \hat{x_n}'$. Konkret kann man $\mathcal{S}\prime$ und $\mathcal{R}$ wie folgt aufstellen:

$$\hat{S_0} = \exists x_1 \ldots \exists x_n (h(x_n) = \hat{x_1} \wedge \ldots \wedge h(x_n) = \hat{x_n} \wedge \mathcal{S}_0(x_1, \ldots, x_n)) \tag{1}$$

und

$$\begin{aligned} \hat{R} = \exists x_1 \ldots \exists x_n \exists x_1' \ldots \exists x_n' (h(x_n) = \hat{x_1} \wedge \ldots \wedge h(x_n) = \hat{x_n} \\ \wedge h(x_1') = \hat{x_1'} \wedge \ldots \wedge h(x_n') = \hat{x_n'} \wedge \mathcal{R}(x_1, \ldots, x_n, x_1', \ldots, x_n')) \end{aligned} \tag{2}$$

Man kann also die abstrakte Kripke-Struktur M_r aufstellen, indem man die obigen Formeln auswertet. Man beachte, daß die freien Variablen $\hat{x_i}$ und $\hat{x_i}'$ sind und daß Zustände, die $\hat{\mathcal{S}\prime}$ lösen in $\hat{S_0}$ sind, und daß Übergänge zwischen Zuständen $s = (\hat{x_1}, \ldots, \hat{x_n})$ und $s' = (\hat{x_1}', \ldots, \hat{x_n}')$ existieren, wenn diese eine Lösung von $\hat{\mathcal{R}}$ sind. Die existentiellen Abstraktionen in den Formeln ist ein wesentliches Merkmal. Die Auswertung der Formeln 1 und 2 kann sehr komplex sein. Der wesentliche Grund ist, daß die Operation der existentiellen Abstraktion auf der äußersten Ebene der Verschachtelung der Formel steht. Unser Ziel ist es nun, die Formeln so zu vereinfachen, daß eine daraus konstruierte Struktur M_a immer noch M_r simuliert, aber die Auswertung mit einem sinnvollen Aufwand erfolgen kann. Diese Aufgabenstellung findet man auch in anderen Bereichen der Informatik [5], und eine Lösung ist es, die existentielle Abstraktion "nach innen" zu ziehen. Wenn die $\exists$-Operatoren direkt vor den atomaren Aussagen stehen, auf die sie sich beziehen, können erfüllende Belegungen mit relativ geringem Aufwand gefunden werden.
Zur Abkürzung führen wir zuerst den Operator $[\cdot]$ ein. Seien im folgenden Φ, Φ_1 und Φ_2 aussagenlogische Formeln erster Ordnung, die über den atomaren Teilformeln des Systems formuliert sind. Hänge Φ also von $x_1, \ldots, x_m$ ab, dann definiere:

$$[\Phi](\hat{x}_1, \ldots \hat{x}_m) = \exists x_1 \ldots \exists x_m (h(x_1) = \hat{x}_1 \wedge \ldots \wedge h(x_m) = \hat{x}_m \wedge \Phi(x_1, \ldots, x_m)) \tag{3}$$

[4] Zur Definition von $\mathcal{S}\prime$ und $\mathcal{R}$ siehe [1, Kapitel 2]
[5] z.B. Planning in der künstlichen Intelligenz

Man beachte wiederum, daß die freien Variablen von $[\Phi](\hat{x}_1, \ldots \hat{x}_m)$ die $\hat{x_i}$ sind, während die freien Variablen von Φ die x_i sind. Wie man sieht gilt also: $\hat{\mathcal{S}}\prime = [\mathcal{S}_0]$ und $\hat{\mathcal{R}} = [\mathcal{R}]$.
Da es wie gesagt zu aufwendig ist, S_0^r und R_r aus $[\mathcal{S}\prime]$ und $[\mathcal{R}]$ zu berechnen, führen wir stattdessen eine Transformation $\mathcal{A}$ ein , die auf die Formel Φ wirkt. Diese verwirklicht die oben beschriebene Idee, das "nach innen drücken" der existentiellen Abstraktion zur vereinfachten Auswertung. Der $[\cdot]$ Operator wird nur noch auf primitive Ausdrücke angewandt. Obwohl es oft nicht möglich ist $[\mathcal{S}\prime]$ und $[\mathcal{R}]$ auszuwerten, ist es doch meistens möglich $\mathcal{A}(\mathcal{S}\prime)$ und $\mathcal{A}(\mathcal{R})$ auszuwerten und durch das Ersetzen des $[\cdot]$ Operators durch die Transformation $\mathcal{A}$ die approximierende Struktur M_a zu konstruieren. Zur Definition:

Definition 5 (Die Transformation $\mathcal{A}$).

1. $\mathcal{A}(P(x_1, \ldots, x_m)) = [P](\hat{x_1}, \ldots, \hat{x_m})$ *wenn* P *atomare Teilformel ist genauso:* $\mathcal{A}(\neg P(x_1, \ldots, x_m)) = [\neg P](\hat{x_1}, \ldots, \hat{x_m})$
2. $\mathcal{A}(\Phi_1 \wedge \Phi_2) = \mathcal{A}(\Phi_1) \wedge \mathcal{A}(\Phi_2)$
3. $\mathcal{A}(\Phi_1 \vee \Phi_2) = \mathcal{A}(\Phi_1) \vee \mathcal{A}(\Phi_2)$
4. $\mathcal{A}(\exists x \Phi_1) = \exists \hat{x} \mathcal{A}(\Phi)$
5. $\mathcal{A}(\forall x \Phi_1) = \forall \hat{x} \mathcal{A}(\Phi)$

Damit dieses Vorgehen sinnvoll ist, müssen wir noch zeigen, daß M_a M simuliert, daß also eine Simulationsrelation zwischen den Zuständen in M und M_a besteht. Dazu das folgende Theorem:

Theorem 2. *M_a simuliert M also:*

$$M \preceq M_a$$

Beweis. Zuerst erinnern wir uns daran, daß die Struktur M schon mit den abstrakten Aussagen der Form $'\hat{x_i} = a_i'$ beschriftet ist, also die Beschriftungsfunktion L der Originalstruktur ersetzt wurde. Wir müssen jetzt eine Simulationsrelation nennen und die entsprechenden Eigenschaften zeigen (Details, siehe [1, Kapitel 2]).

- Zwei Zustände, die zueinander in Relation stehen, müssen die gleiche Beschriftung haben
- Übergänge zwischen zwei Zuständen in einer Struktur M müssen auch zwischen den Bildern der Zustände in der simulierenden Struktur M_a erhalten bleiben M_a
- Das Bild eines Zustandes aus der Menge der Startzustände einer Struktur M muss in der Menge der Startzustände der simuliernden Struktur M_a liegen.

Definiere die Simulationsrelation H so, daß $H(s, s_a)$ genau dann, wenn für alle i gilt: $h(d_i) = a_i$. Dabei sei $s = (d_1, \ldots, d_n)$ und $s_a = (a_1, \ldots, a_n)$ Es gelte nun $S(s, s_a)$. Dann gilt auch mit $s = (d_1, \ldots, d_n)$ nach der Definition von H: $s_a = (h(d_1), \ldots, h(d_n))$. Zuerst zeigen wir, daß die Beschriftung der beiden Zustände s und s_a gleich ist: Die Beschriftung von s ist nämlich die Menge

von atomaren Aussagen über den abstrakten Variablen mit Werten aus dem abstrakten Wertebereich A der Form $'\hat{x}_i = a'_i$. s hat diese Beschriftung genau dann, wenn $h(d_i) = a_i$. s_a erhält die Beschriftung $'\hat{x}_i = a'_i$ genau dann, wenn die i-te Komponente von s_a gleich a_i ist. Die i-te Komponente von s_a ist aber, nach Definition von H, $h(d_i)$ was dasselbe ist wie a_i □

.

Um nun zu zeigen, daß Übergänge aus M in M_a erhalten bleiben, betrachten wir einen Übergang $R(s,t)$ mit $t = (e_1, \ldots, e_n)$. Definiere $t_a = (h(e_1), \ldots, h(e_n))$. Wir müssen also zeigen, daß $R(s_a, t_a)$. Nach der Definition von R wissen wir, daß Übergänge genau dann in R enthalten sind, wenn die entsprechenden Zustände $\mathcal{R}$ erfüllen. Also gilt $\mathcal{R}(s,t)$. Wir zeigen zunächst $[\mathcal{R}(s_a, t_a)]$. Nach Einsetzen in die Definition von $[\cdot]$ gilt $[\mathcal{R}](s_a, t_a)$ genau dann, wenn

$$\exists x_1 \ldots \exists x_n \exists x'_1 \ldots \exists x'_n (\bigwedge_{i=1}^{n} (h(x_i) = h(d_i) \wedge h(x'_i) = h(e_i)) \wedge \mathcal{R}(x_1, \ldots, x_n, x'_1, \ldots, x'_n)).$$

Die obige Formel ist wahr, wie man leicht einsieht, wenn man die d_i als "Zeugen" für die x_i, und die e_i als "Zeugen" für die x'_i nimmt. $[\mathcal{R}](s_a, t_a)$ impliziert $\mathcal{A}(\mathcal{R})(s_a, t_a)$, wie wir später zeigen werden (siehe Theorem 3). Da aber $\mathcal{A}(\mathcal{R})$ R_a definiert, also Übergänge zwischen Zuständen, die $\mathcal{A}(\mathcal{R})$ erfüllen, existieren, ist H eine Simulationsrelation. Analog sieht man, daß aus $s \in S$ folgt $s_a \in S_0^a$. Also hat jeder Startzustand von M einen korrespondierenden Startzustand von M_a. M_a simuliert also M, $M \preceq M_a$. □

Zum obigen Beweis fehlt also noch das folgende Theorem:

Theorem 3. *Aus* $[\Phi]$ *folgt* $\mathcal{A}(\Phi)$. *Insbesondere gilt:* $[S_0] \Rightarrow \mathcal{A}(S_0)$ *und* $[\mathcal{R}] \Rightarrow \mathcal{A}(\mathcal{R})$.

Beweis. Der Beweis erfolgt durch Induktion über die Struktur der Formel Φ.

1. Wenn $\Phi = P(x_1, \ldots, x_m)$ oder $\Phi = /P(x_1, \ldots, x_m)$ wobei P eine atomare Teilformel ist, dann ist $[\Phi] = \mathcal{A}(\Phi)$ und die Aussage gilt.
2. Sei $\Phi(x_1, \ldots, x_m) = \Phi_1 \wedge \Phi_2$, dann ist $[\Phi_1 \wedge \Phi_2]$ nach der Definition von $[\cdot]$ identisch zu der Formel

$$\exists x_1 \ldots \exists x_m (\bigwedge_i h(x_i) = \hat{x_i} \wedge \Phi_1 \wedge \Phi_2). \tag{4}$$

Aus dieser Formel folgt:

$$\exists x_1 \ldots \exists x_m (\bigwedge_i h(x_i) = \hat{x_i} \wedge \Phi_1) \wedge \exists x_1 \ldots \exists x_m (\bigwedge_i h(x_i) = \hat{x_i} \wedge \Phi_2) \tag{5}$$

da x_i, die als "Zeugen" für den Ausdruck 4 genommen werden können, Φ_1 und Φ_2 erfüllen müssen und gleichzeitig $\bigwedge_i h(x_i) = \hat{x_i}$ erfüllen. Diese x_i müssen also auch Ausdruck 5 erfüllen, wo zusätzlich die Belegungen, die Φ_1 und Φ_2 erfüllen, verschieden seien können.

Da aber der Ausdruck 5 gerade $[\Phi_1] \wedge [\Phi_2]$ ist, und nach Definition von $\mathcal{A}$ $\mathcal{A}(\Phi_1 \wedge \Phi_2) = \mathcal{A}(\Phi_1) \wedge \mathcal{A}(\Phi_2)$ ist, folgt aus der Induktionsvoraussetzung ($[\Phi] \Rightarrow \mathcal{A}(\Phi)$). Aus $[\Phi_1 \wedge \Phi_2]$ folgt $[\Phi_1] \wedge [\Phi_2]$ und aus $[\Phi_1]$ folgt $\mathcal{A}(\Phi_1)$ und aus $[\Phi_2]$ folgt $\mathcal{A}(\Phi_2)$, also folgt aus $[\Phi_1 \wedge \Phi_2]$ $\mathcal{A}(\Phi_1) \wedge \mathcal{A}(\Phi_2) = \mathcal{A}(\Phi_1 \wedge \Phi_2)$. □

3. Der Fall $\Phi = \Phi_1 \vee \Phi_2$ ist analog zum vorherigen Fall.
4. Sei $\Phi = \forall x \Phi_1$, dann ist $[\forall x \Phi_1]$

$$\exists x_1 \ldots \exists x_m (\bigwedge_i h(x_i) = \hat{x_i} \wedge \forall x \Phi_1(x, x_1, \ldots, x_m))$$

Ohne Beschränkung der Allgemeinheit sei die gebundene Variable x verschieden von den x_i und $\hat{x_i}$ (andernfalls wird sie umbenannt). Dann ist der Ausdruck äquivalent zu

$$\exists x_1 \ldots \exists x_m \forall x (\bigwedge_i h(x_i) = \hat{x_i} \wedge \Phi(x, x_1, \ldots, x_m)),$$

da man den $\forall$-Operator dann herausziehen kann. Daraus folgt

$$\forall x \exists x_1 \ldots x_m (\bigwedge h(x_i) = \hat{x_i} \wedge \Phi_1(x, x_1, \ldots, x_m)). \tag{6}$$

Bis jetzt haben wir nur logische Umformungen benutzt, die unabhängig von der Problemstellung waren. Jetzt benutzen wir zusätzlich noch die Tatsache, daß die Abbildung h eine Surjektion ist. Das heißt, daß es zu jedem Element $\hat{x} \in A$ ein Element $x \in D$ gibt, so daß $h(x) = \hat{x}$. Also: $\forall \hat{x} \exists x (h(x) = \hat{x})$. Damit folgt aus dem Ausdruck 6:

$$\forall \hat{x} \exists x [\exists x_1 \ldots \exists x_m (h(x) = \hat{x} \wedge \bigwedge_i h(x_i) = \hat{x_i} \wedge \Phi_1(x, x_1, \ldots, x_m))]. \tag{7}$$

Dies sieht man leicht ein, da nur die Surjektivität von h ausgedrückt wurde und der Quantor $\forall x$ durch $\forall \hat{x} \exists x$ ersetzt wurde. Da aber ein Ausdruck, der für alle Werte von x gilt, erst recht für den Wert gilt, für den $h(x) = \hat{x}$ ist. Ausdruck 7 ist aber genau $\forall \hat{x} [\Phi_1]$. Da aber, nach Induktionsanfang, wiederum aus $[\Phi_1]$ $\mathcal{A}(\Phi_1)$ folgt, folgt aus $\forall \hat{x} [\Phi_1]$ $\forall \hat{x} \mathcal{A}(\Phi_1)$. Aber dies ist gerade $\mathcal{A}(\forall x \Phi_1)$.

5. Der Fall $\Phi = \exists x \Phi_1$ wird analog zum vorherigen behandelt.

□

2.4 Zusammenfassung

Komplexe Datenstrukturen in dem zu verifizierndem System führen zu einem stark vergrößertem Zustandsraum. Dies ist zum Beispiel bei Softwaresystemen der Fall, oder Schaltkreisen, die Datenpfade (Data paths) enthalten. Dadurch wird Model Checking sehr erschwert oder sogar unmöglich gemacht, die schiere Größe gestattet es nicht mehr in vertretbarem Aufwand das Model Checking Problem zu lösen. Zusätzlich wird durch das Vorhanden-Sein und Sichtbar-Sein

komplexer Datenstrukturen auch die Anwendung anderer Techniken zur Zustandsraumverkleinerung erschwert. In diesem Teil wurden Techniken zur Abstraktion von komplexen Datenstrukturen vorgestellt, die es ermöglichen die Gültigkeit von Formeln über die abstrakten Daten effizient zu verifizieren. Beschrieben wurde die Technik der idealen Abstraktion und der Approximation an diese. Grob gesagt sind Approximationen einfacher zu berechnen, sind aber ungenauer als ideale Abstraktionen. Auserdem wird im nchsten Abschnitt der Spezialfall der exakten Approximation beschrieben.

Unter bestimmten Bedingungen kann man nmlich zeigen, daß $M_a \equiv M$ gilt, was uns die Möglichkeit gibt, beliebige CTL* Ausdrücke zu verifizieren. M_a heißt dann "exakte" Approximation, da $[\Phi] \iff \mathcal{A}(\Phi)$ gelten muss.

3 Exakte Approximation.

Im vorherigen Abschnitt wurde gezeigt, daß $M \preceq M_a$, daß also beliebige ACTL* Formeln über den abstrakten Wertebereich A, die für M_a gezeigt werden können, auch für M gelten. Ein Ziel ist es natürlich zu zeigen, daß $M \equiv M_a$ ist, also daß M_a und M Bisimulation-äquivalent sind. Dies würde uns ermöglichen, beliebige CTL* Ausdrücke zu verifizieren und auch das Problem der "spurious counterexamples", also der falschen Gegenbeispiele, zu beseitigen. Diese können nur deshalb auftreten, da ein Gegenbeispiel für einen Ausdruck über A in der Struktur M_a nicht unbedingt bedeutet, daß in M ein Gegenbeispiel existiert. Bei Bisimulation-Äquivalenz würde aber in M ein solches existieren. Intuitiv muss zumindest $[\Phi] \iff \mathcal{A}(\Phi)$ gelten, damit eine Approximation exakt ist. Andernfalls würden Übergänge in M_a nicht unbedingt zwischen korrespondierenden Zuständen in M existieren. Dann gilt nämlich insbesondere $[\mathcal{S}_l]$ und $[\mathcal{R}]$ sind äquivalent zu $\mathcal{A}(\mathcal{S}_l)$ beziehungsweise $\mathcal{A}(\mathcal{R})$. Clarke, Grumberg und Long beweisen in [2], daß dann eine Bisimulationsrelation zwischen M_a und M existiert und daß M_a und M auch Bisimulation-äquivalent sind.
In diesem Abschnitt werden nur die Ergebnisse genannt, für die Beweise und exakte Ausarbeitung sei auf die Literatur verwiesen.

Die Forderung, die an M und die h_x gestellt wird (wir erinnern uns daran, daß M_a vollständig durch die h_x festgelegt wird) ist, daß die Äquivalenzrelationen $\sim_x$, die von jeder Abstraktionsabbildung h_x induziert wird, Kongruenzen in Bezug auf die atomaren Aussagen P sind.
Jede Abstraktionsabbildung h_x für die Variable x induziert auf folgende Weise eindeutig eine Äquivalenzrelation:
Seien d_1 und d_2 in D_x. Dann $d_1 \sim_x d_2$ genau dann, wenn $h_x(d_1) = h_x(d_2)$. Diese müssen jetzt kongruent in Bezug auf die atomaren Aussagen $P(x_1, \ldots, x_n)$ des Systems sein. Dazu die Definition von kongruent:
Sei $P(x_1, \ldots, x_n)$ eine Aussage wobei x_i Werte in D_{x_i} hat. Dann ist die Äqui-

valenzrelation $\sim_x$ eine Kongruenz in Bezug auf P genau dann, wenn

$$\forall d_1 \ldots d_m \forall e_1 \ldots \forall e_m (\bigwedge_{i=1}^{m} d_i \sim_{x_i} e_i \Rightarrow (P(d_1, \ldots, d_m) \iff P(e_1, \ldots, e_m))). \tag{8}$$

Wie erwähnt basiert der Beweis für die Exaktheit der Approximation wenn die $\sim_{x_i}$ Kongruenzen in Bezug auf die atomaren Aussagen des Systems sind, auf dem folgenden Theorem:

Theorem 4. *Wenn die $\sim_{x_i}$ Kongruenzen in Bezug auf die atomaren Aussagen sind und Φ ist eine Formel, die über diesen exakten Aussagen definiert ist, dann $[\Phi] \iff \mathcal{A}(\Phi)$. Insbesondere sind $[\mathcal{S}_\prime]$ und $[\mathcal{R}]$ äquivalent zu $\mathcal{A}(\mathcal{S}_\prime)$ beziehungsweise $\mathcal{A}(\mathcal{R})$.*

Mit diesem Theorem kann das folgende und entscheidende bewiesen werden:

Theorem 5. *Wenn die $\sim_{x_i}$ Kongruenzen in Bezug auf die atomaren Aussagen sind, dann $M \equiv M_a$.*

Für Beweise bzw Ausarbeitungen, siehe [2].

4 Praktische Überlegungen und Beispielabstraktionen

Clarke stellt in seinem Text ein Umgebung vor, die die vorhergehenden Überlegungen implementiert. An dieser Stelle werden die wichtigsten Punkte zusammengefasst und Ergebnisse für reale Probleme genannt, die Clarke untersucht hat. Das Ziel ist es, eine qualitative Vorstellung von der Zeit- und Platzersparniss durch ein solches System zu bekommen.

Wie bereits bei den Ausführungen zu den Abstraktionsmethoden (siehe Abschnitt 2.1 auf Seite 102) erläutert wurde, ist ein wichtiger Aspekt, daß die Strukturen nicht explizit generiert werden, sondern OBDD Beschreibungen der Strukturen generiert werden. Grundlage ist die Darstellung in Form von $\mathcal{S}_\prime$ und $\mathcal{R}$. Clarke stellt ein System vor, bestehend aus einer einfachen Hardware-Beschreibungssprache (HDL) und einem Compiler, der eine Darstellung des Systems als OBDD direkt generiert. Diese kann dann als Eingabe für einen Model-Checker benutzt werden. Der Benutzer kann Abstraktionen spezifizieren und das System generiert dann die entsprechende Darstellung der abstrakten Struktur direkt, ohne zuerst die Originalstruktur zu instanziieren. Dies, sowie die durchgehende Darstellung als OBDD, ist sehr wichtig, da wie beschrieben die Instanziierung der Originalstruktur aus Platzgründen unmöglich sein kann. Die Darstellung als OBDD erfolgt auch aus Effizienzgründen. Der Benutzer kann neue Abstraktionen spezifizieren, indem er direkt oder indirekt die OBDDs, die diese Abstraktion repräsentieren definiert. Dies ist ein wichtiger Punkt, und wird symbolische Abstraktion genannt.

4.1 Symbolische Abstraktion

Die Benutzung von einem OBDD basierten Compiler mit einem OBDD basierten Model-Checker, der die vom Compiler generierte Struktur als Eingabe hat macht es möglich, Abstraktionen zu definieren, die von symbolischen Werten abhängen. Was dies heißt und was das für Möglichkeiten eröffnet, machen wir uns an einem einfachem Beispiel klar: Das Listing in Abbildung 4.1 stellt darüber hinaus ein Beispiel für die Sprache dar, die Clarke für sein System benutzt. Wichtige Eigenschaften sind:

- Sie ist prozedural und enthält viele Elemente der strukturierten Programmierung, wie zum Beispiel **while**-Schleifen.
- Der Zustandsraum der Programme ist endlich, das heißt, dass für Variablen die Bitbreite angegeben werden muss. Im Beispiel geschieht dies z.B. in dem Ausdruck **input** $a : 8;$, wo eine acht Bit breite Eingabevariable a definiert wird.
- Berechnungen erfolgen synchron. Am Anfang jedes Berechnungsschrittes werden die Eingaben von der Umgebung abgefragt, dann erfolgt die Berechnung. Diese erfolgt sofort, das heißt braucht keine Zeit. Die Ausgaben werden bei Erreichen eines **wait**-Ausdruckes in die mit **output** gekennzeichneten Variablen geschrieben. Die Anzahl der erreichten **wait**-Ausdrücke gibt also die Anzahl der Zeitschritte an, die das Programm schon gelaufen ist.

Diese grobe Beschreibung sollte reichen, um das folgende Beispiel und die Ausführungen dazu zu verstehen. Die Elemente der Sprache sind ansonsten intuitiv zu verstehen.

```
input a : 8;
output b : 8 := 0;

loop
      b:=a;
      wait;
end loop
```

Abb. 5. Beispiellisting

Abbildung 4.1 stellt also ein Programm dar, das den Wert a von der Umgebung übernimmt und ihn in die Variable b, den Ausgang, kopiert.

Wir wollen zeigen, daß der nächste Zustand von b immer den aktuellen Wert von a annimmt. Dies kann für eine feste Zahl, z.B. 42 durch die folgende Formel ausgedrückt werden:

$$\mathbf{AG}(a = 42 \Rightarrow \mathbf{AX}\ b = 42).$$

Will man nur diese Eigenschaft verifizieren bietet sich die folgende Abstraktion für a und b an:

$$h(d) = \begin{cases} 0 \text{ wenn } i = 42 \\ 1 \text{ sonst} \end{cases}$$

Wenn diese Abstraktion angewandt wird und das Programm kompiliert wird, wird die entsprechende Übergangsrelation $\hat{R}$ generiert, die durch $\hat{b'} = \hat{a}$ definiert wird. Die gestrichenen Variablen bezeichnen den Folgezustand. Die abstrakten Variablen $\hat{a}$und $\hat{b}$ haben den Wertebereich $\{0, 1\}$ und auf der abstrakten Ebene wird die folgende Formel verifiziert:

$$\mathbf{AG}(\hat{a} = 0 \Rightarrow \mathbf{AX}\ \hat{b} = 0).$$

Diese Formel wird von dem Programm natürlich erfüllt. Allerdings muss zur Verifikation des Programmes diese Formel für alle möglichen Werte in der Abstraktionsabbildung überprüft werden. Der Benutzer müsste also den ganzen Vorgang für alle möglichen Werte wiederholen. Offensichtlich ist das nicht was wir wollen und die Lösung ist es, den Parameter (42 in diesem Fall) durch einen symbolischen Parameter zu ersetzen. Die entstehende Abstraktionsfunktion sieht dann folgendermaßen aus:

$$h_c(d) = \begin{cases} 0 \text{ wenn } i = c \\ 1 \text{ sonst} \end{cases}$$

Wird das Programm nun mit dieser **symbolischen Abstraktion** kompiliert, erhalten wir eine Übergangsrelation $\hat{R_c}$, die durch den Parameter c indiziert wird. Setzen wir $c = 42$ fest, was dem Setzen der entsprechenden Variablen im OBDD, der die Abstraktionsabbildung repräsentiert, entspricht, erhalten wir wieder $\hat{R}$, wie vorher beschrieben. Die Spezifikation über die abstrakten Variablen $\hat{a}$ und $\hat{b}$

$$\mathbf{AG}(\hat{a} = 0 \Rightarrow \mathbf{AX}\ \hat{b} = 0)$$

ist praktisch gleichbedeutend zu

$$\mathbf{AG}(a = c \Rightarrow \mathbf{AX}\ b = c).$$

Wenn diese Formel für alle möglichen Belegungen von c wahr ist, dann haben wir die gewünschte Spezifikation erfüllt. c wird im OBDD der Abstraktionsabbildung h_c durch 8 zusätzliche Felder kodiert, da es ein 8-Bit Wert ist. Es zeigt sich nun, daß es möglich ist, diese Spezifikation zusammen mit der als OBDD dargestellten Abstraktion zu kompilieren und dann als Eingabe zu einem Model-Checker zu benutzen, wie er in vorherigen Beiträgen definiert wurde. Die Ausgabe des Model-Checkers sind dann ein parametrisierter OBDD, der die Zustände repräsentiert, für die die Spezifikation gilt. Es ist also möglich, und Clarke hat diese Technik benutzt um den Schaltkreis aus Abbildung 6 zu verifizieren,

1. h_c durch einen OBDD zu repräsentieren (dies ist die Benutzereingabe);
2. mit h_c zu kompilieren, um einen OBDD der $\hat{R}(\hat{a}, \hat{a'}, \hat{b}, \hat{b'}, c)$ zu produzieren (dieser Schritt wird automatisch vom Compiler ausgeführt)

3. das Model-Checking durchzuführen, um einen OBDD zu erhalten, der die parametrisierte Zustandsmenge repräsentiert, die die Spezifikation erfüllt (dies wird automatisch vom Model-Checker, der c einfach als konstante, zusätzlichen Komponenten des Zustandes sieht, gemacht)
4. wenn nötig ein spezielles c auszuwählen, um ein Gegenbeispiel zu generieren (dies wird auch vom Model-Checker übernommen)

Die Benutzung von symbolischen Abstraktionsabbildungen, wie h_c eine ist, macht die Prozesse der Abstraktion, Kompilation und Model-Checking durch die Benutzung von OBDDs die gemeinsam genutzt werden, sehr viel effizienter. Dazu das folgende Beispiel:

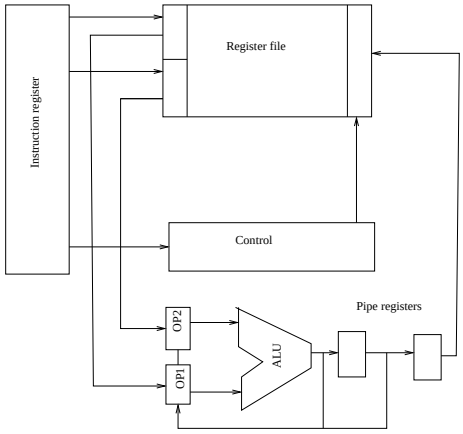


Abb. 6. Pipeline Schaltkreis - Blockdiagramm

Beispiel 3. Clarke hat die Methode der symbolischen Abstraktion benutzt, um einen einfachen Pipeline-Schaltkreis, wie er in Abbildung 6 dargestellt ist, zu verifizieren (für eine detailierte Beschreibung der Funktion des Schaltkreises siehe [1, Kapitel 6]). Dieser Schaltkreis führt 3-Adressen arithmetische und logische Operationen an Operanden durch, die in einem Register-File gespeichert werden. Es wurden zwei unabhängige Abstraktionen benutzt. Zuerst wurden die Registeradressen abstrahiert, so daß nur zwischen drei symbolischen Konstanten ra, rb und rc sowie einer sonstigen Adresse unterschieden wird. Dies macht es möglich, das ganze Register-File auf nur drei Register, eines für jede Konstante, zu reduzieren. Die zweite symbolische Abstraktion betrifft den Inhalt der einzelnen Register. Für jede Operation werden symbolische Konstanten definiert (ca und cb). Nach der Abstraktion kann dann jedes Register entweder den Wert ca, cb, $ca + cb$ oder einen sonstigen Wert enthalten. Also wieder in Anlehnung an das obige Beispiel. Das heißt, daß z.B. die Additionsfunktion durch die folgende Formel verifiziert werden kann:

$$\mathbf{AG}((srcaddr1 = ra) \wedge (srcaddr2 = rb) \wedge (destaddr = rc) \wedge \neg stall$$
$$\Rightarrow \mathbf{AG}\ \ \mathbf{AX}((regra = ca) \wedge (regrb = cb) \Rightarrow \mathbf{AX}(regrc = ca + cb))).$$

Diese Formel besagt, daß wenn die Quelladressregister ra und rb sind, und das Zieladressregister rc ist, der Inhalt des Registers rc nach drei Zyklen gleich der Summe der Werte in den Registern ra und rb ist. Die drei Zyklen Verzögerung enstehen durch die Pipeline. Voraussetzung ist natürlich, daß die Pipeline frei ist ($\neg stall$). Die Verifikation erfolgt dabei unter Benutzung der symbolischen Abstraktion, wie sie vorher erläutert worden ist. Dies ist der wichtige Punkt, der es möglich macht, größere Schaltkreise zu verifizieren.
Die größte Pipeline, die mit den oben genannten Techniken verizifizert wurde, hatte 64 Register im Register-File und jedes Register war 64 Bit breit. Der Schaltkreis hatte mehr als 4000 Bits die den jeweiligen Zustand kodierten, also fast 10^{1300} erreichbare Zustände. Clarke gibt an, daß die Verifikation ungefähr sechseinhalb Stunden CPU-Zeit brauchte. Ein wichtiger Punkt ist auch noch, daß diese Zeit linear in der Anzahl der Register und der Breite der Register ist. Zum Vergleich: Der größte Schaltkreis, der ohne Abstraktion verifiziert werden konnte, war eine Pipeline mit 8 Registern von 32-Bit Breite. Der Verifikationsprozess brauchte dann schon ungefähr viereinhalb Stunden CPU-Zeit auf einer Sun 4. Vor allem wuchs diese Zeit quadratisch in der Breite der Register und kubisch in der Anzahl der Register.

4.2 Beispielabstraktionen

Wie wir im Beispiel 3 gesehen haben ist es ein wichtiger Punkt, im Verifikationsprozess geeignete Abstraktionen zu definieren. Die Wahl der Abstraktion ist ein Balanceakt zwischen zu grober Abstraktion, die nicht genug mit dem Verifikationsproblem zu tun hat, und ungenügender Abstraktion, die es nicht ermöglicht, das Problem genügend zu verkleinern und damit praktisch zu bewältigen. Es gibt Bemühungen, die Wahl geigneter Abstraktionen iterativ und automatisch zu gestalten.
In diesem Abschnitt werden aber einige Abstraktionen, also Abbildungen h für spezielle Problemstellungen vorgestellt, die auch in der Praxis benutzt werden. Es soll dabei keine mathematisch genaue Beschreibung gegeben werden, da diese auch von den spezifischen Anforderungen abhängt, sondern vielmehr ein Gefühl dafür gegeben werden, wie man das Werkzeug der Abstraktion anwenden kann.

Kongruenz modulo einer ganzen Zahl Sei m eine ganze Zahl. Dann bietet sich zur Verifikation von arithmetischen Funktionen oft die Abstraktionsabbildung

$$h(i) = i \bmod m$$

an, da die Eigenschaften der Arithmetik von Zahlen modulo m,

$$((i \bmod m) + (j \bmod m)) \bmod m \equiv i + j (\bmod m)$$
$$((i \bmod m) - (j \bmod m)) \bmod m \equiv i - j (\bmod m)$$
$$((i \bmod m)(j \bmod m)) \bmod m \equiv ij (\bmod m),$$

denen der Arithmetik mit ganzen Zahlen i und j entsprechen. Dadurch wird es möglich den Wertebereich der Zustandsvariablen einzugrenzen und damit die Verifikation von Schaltkreisen, wie Multiplizierer, zu ermöglichen. Um nun einen Schaltkreis exakt zu verifizieren, also nicht nur modulo einer ganzen Zahl, kann das folgende Theorem benutzt werden:

Theorem 6 (Chinesischer Restklassensatz). *Seien $m_1, m_2, \ldots, m_n$ positive ganze Zahlen, die paarweise relativ prim zueinander sind. Definiere $m = m_1 m_2 \ldots m_n$ und seien $b, i_1, i_2, \ldots, i_n$ ganze Zahlen. Dann gibt es genau eine ganze Zahl i, so daß*

$$b \leq i < b + m \text{ und } i \equiv i_j (mod\ m_j) \text{ für } 1 \leq j \leq n.$$

Kann man jetzt verifizieren, daß der Wert einer Variable x in einem Programm gleich $i_j \ mod \ m_j$ für alle m_j, die relativ zueinande prim sind, dann ist der Wert der Variablen exakt definiert. Um alle möglichen Werte für die i_j zu überprüfen, kann man sich wiederum der symbolischen Abstraktion bedienen.

4.3 Repräsentation durch den Logarithmus

In manchen Fragestellungen ist nur die Größenordnung einer Variable von Interesse. Dann bietet sich die Abstraktionsabbildung

$$h(i) = lg\ (i)$$

an, wobei $lg\ (i)$ durch

$$lg\ (i) = \lceil log_2(i+1) \rceil$$

definiert ist. Also ist $lg(i)$ von 0 gleich 0. Für $i > 0$ ist $lg\ (i)$ die kleinste Anzahl Bits, die benötigt wird, um i als Binärzahl darzustellen. Dadurch ist es zum Beispiel möglich, in arithmetischen Schaltkreisen Überläufe festzustellen.

4.4 Abstraktion durch Repräsentation durch einzelne Bits

Bei bitweisen logischen Operationen kann es sinnvoll sein, die Werte durch die Abstraktionsabbildung

$$h(i) = \text{j-tes Bit von i}$$

darzustellen. Reicht das nicht um die gewünschten Eigenschaften zu verifizieren, dann kann man

$$h(i) = (h_1(i), h_2(i))$$

als Abstraktionsabbildung benutzen. h_1 und h_2 sind dabei auch Abstraktionsabbildungen. [6]

[6] Ein Beispiel findet sich z.B. in [2]

4.5 Ausblick

Zum Schluss wollen wir noch kurz einige Punkte hervorheben, die besonders in Hinblick auf die praktische Anwendung von Bedeutung sind und deren weitere Entwicklung noch aktiv betrieben wird.

- Einer der wichtigsten Punkte ist die vollautomatische Verifikation. Wie an mehreren Stellen erwähnt, ist die Auswahl der geeigneten Abstraktionsabbildung ein kritischer Punkt. Dies ist ein Balanceakt zwischen zu grober Abstraktion und ungenügender Abstraktion. Die Erstere würde zu viele "spurious counterexamples" liefern oder schlicht nicht mehr viel mit der zu verifizierenden Eigenschaft zu tun haben während die Letztere den Zustandsraum nicht genügend verkleinern würde. Das Ziel ist in diesem Zusammenhang die vollautomatische Auswahl der Abstraktionsabbildungen. Hierzu gibt es inkrementelle Ansätze, die mit einer groben Abstraktion anfangen und diese solange verfeinern, bis die gewünschte Eigenschaft verifiziert werden kann.
- Die Verifikation von Software nimmt mitlerweile einen wichtigen Platz ein. Denn durch die verbesserten Verifikationstechniken wird es möglich, Systeme mit vielen Zustandsvariablen zu verifizieren. Z.B. werden Model Checking Techniken bei Microsoft angewandt, um Hardware-Treiber zu verifizieren. Dieses Einsatzgebiet bietet sich an, da korrekte Funktion hier sehr wichtig ist, aber es sich um relativ kleine Systeme handelt. Zur Wichtigkeit dieses wachsenden Gebietes auch für kommerzielle Anwendungen Bill Gates: "Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability" (Keynote address bei der WinHec 2002: [4]) Bei Microsoft wird in Zusammenarbeit mit Universitäten mit dem SLAM Projekt (http://research.microsoft.com/slam/) dieses Ziel verfolgt.
- Für alle Bereiche ist es wichtig, praktisch relevante Abstraktionen zu finden und zu charakterisieren (siehe z.B. [3]). Dazu gehört auch, den Begriff der Genauigkeit der Abstraktionen und Approximationen zu formalisieren.

Literatur

1. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999.
2. E.M. Clarke, O. Grumberg and D. E. Long. Model checking and abstraction. In *ACM Transactions on ProgrammingLanguages and Systems 16(5)*, pages 1512-1542.
3. S. Bensalem, A. Bouajjani, C. Loiseaux und J. Sifakis. Property Preserving Simulations. In *Workshop on Computer-Aided Verification , Fourth International Workshop CAV'92. Proceedings, LNCS 663*, Springer, 1992
4. http://www.microsoft.com/billgates/speeches/2002/04-18winhec.asp

Symmetrie

Michael Drescher

Institut für Informatik
Albert-Ludwigs-Universtität Freiburg
mdresche@informatik.uni-freiburg.de

Zusammenfassung. Ein Hauptproblem beim Model Checking ist die Größe des zu durchsuchenden Zustandsraums. Das Ausnutzen von Symmetrie in Systemen mithilfe gruppentheoretischer Methoden ist eine Möglichkeit zur Verkleinerung des Zustandsraums. Probleme treten dabei insbesondere bei den BDD-basierten symbolischen Techniken auf.

1 Einleitung

Viele für Model-Checking geeignete Systeme weisen einen hohen Grad an Symmetrie auf. Seien das Speicher, bei denen es nicht wichtig ist, in welcher der Speicherzellen ein Wert abgespeichert ist, oder gleichartige Prozessoren, die an einen gemeinsamen Bus angeschlossen sind oder sich in einer regelmäßigen Struktur (etwa einem Ring) untereinander ein Token herumreichen (siehe Abb. 1). Auch in allgemeineren Kommunikationsprotokollen und Softwaresystemen, die aus identischen Prozessen bestehen, tritt Symmetrie auf.

Diese Symmetrie spiegelt sich dann auch in den zugehörigen Kripke-Strukturen wieder, in denen sie sich in Form sogenannter Automorphismen niederschlägt. Wir werden sehen, daß diese Automorphismen Anlaß zu einer Äquivalenzrelation auf der Zustandsmenge bieten. Verschmilzt man grob gesagt äquivalente Zustände (bildet also den sogenannten Quotienten nach dieser Relation), so erhält man eine kleinere Struktur, die sogenannte Quotientenstruktur. Diese ist unter bestimmten Voraussetzungen bisimulationsäquivalent zur Originalstruktur und erfüllt damit dieselben CTL^*-Formeln. Fordert man, daß nur die Gültigkeit einer bestimmten gegebenen Formel erhalten bleibt, kann man häufig noch kleinere Quotientenstrukturen erhalten. Quotientenstrukturen können exponentiell kleiner sein als ihre Originalstrukturen.

Diese Reduktion ist sehr wichtig, da die Anwendbarkeit von Model-Checking meist abhängig ist von der Größe der Kripke-Struktur. Diese ist nicht nur der zeitlich bestimmende Faktor, häufig ist auch der wegen ihr während des Model-Checkings benötigte Speicherplatz zu groß.

Für die explizite Konstruktion der kleineren Struktur gibt es relativ einfache Algorithmen, bei der symbolischen kommt es zumindest beim hier gewählten Ansatz jedoch zu Problemen.

Parallel führten 1993 Clarke et al. und Emerson/Sistla das Ausnutzen von Symmetrie im Model-Checking ein ([3], [7]). Dieser Text orientiert sich im wesentlichen an [4, Kapitel 14], das wiederum auf [3] basiert, das den allgemeineren

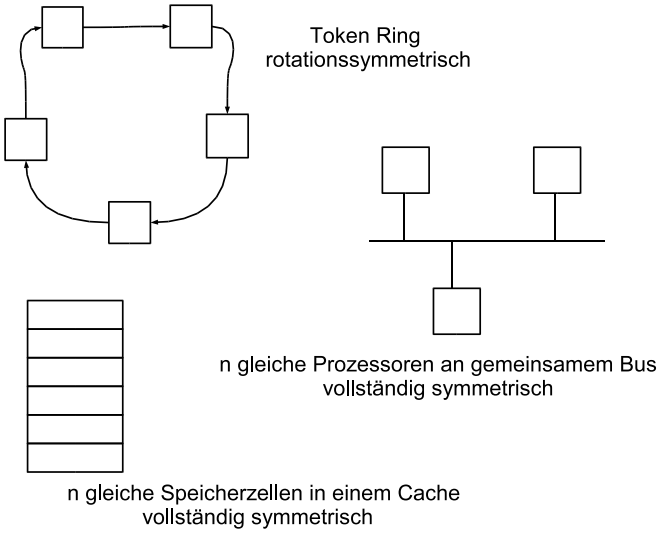


Abb. 1. Beispiele für Symmetrie in Systemen

der beiden Ansätze beschreibt. Emerson und Sistla beschränken sich in [7] auf Indexpermutationen (siehe unten).

Wir werden zunächst in Abschnitt 2 die benötigten mathematischen Grundlagen über Gruppen, Permutationen und Automorphismen kennenlernen. Darauf aufbauend folgt in Abschnitt 3 die Definition der Quotientenstrukturen, deren Nutzen für das Model-Checking wir in Abschnitt 4 sehen werden. Möglichkeiten zur Konstruktion dieser Strukturen sowohl für den expliziten als auch den BDD-basierten symbolischen Fall werden in Abschnitt 5 behandelt, deren Probleme in Abschnitt 6 diskutiert werden. Gegen Ende in Abschnitt 7 folgt dann noch ein kurzer Ausblick auf hier nicht besprochene Probleme und Fragestellungen.

2 Mathematische Grundlagen

Zur Reduktion des Zustandsraums wird die sogenannte Automorphismengruppe einer Kripke-Struktur das zentrale Hilfsmittel sein. Diese besteht aus bestimmten Permutationen des Zustandsraums.
Daher zunächst eine Einführung in Gruppen und Permutationen:

Definition 1. *Sei G eine Menge und $\circ : G \times G \to G$.*
$(G, \circ)$ heißt Gruppe $:\Longleftrightarrow \exists e \in G$

1. $\forall a, b, c \in G : a \circ (b \circ c) = (a \circ b) \circ c$ *(Assoziativität)*
2. $\forall a \in G : a \circ e = e \circ a = a$ *(Existenz eines neutralen Elements)*
3. $\forall a \in G \exists a^{-1} \in G : a \circ a^{-1} = a^{-1} \circ a = e$ *(Existenz inverser Elemente)*

$(H, \circ^H)$ *heißt* Untergruppe von $(G, \circ^G) : \iff H \subseteq G$, $\circ^H = \circ^G|_{H \times H}$ *und* $(H, \circ^H)$ *ist Gruppe*
Sei G *Gruppe,* $g_1, \ldots, g_k \in G$. *Dann existiert die kleinste Untergruppe von* G, *die* $g_1, \ldots, g_k$ *enthält*[1]. *Diese heißt* von $\{g_1, \ldots, g_k\}$ erzeugte Untergruppe *(kurz:* $\langle g_1, \ldots, g_k \rangle$ *)*

Anmerkung 1. Statt $(G, \circ^G)$ schreiben wir auch einfach G, wenn die Verknüpfung aus dem Zusammenhang klar ist.

Definition 2. *Sei* A *eine endliche Menge. Dann heißt* $\sigma : A \to A$ Permutation auf A, *wenn* σ *bijektiv ist.* $Sym\ A := \{\sigma | \sigma$ *Permutation auf* $A\}$

Beispiel 1. Eine Permutation der Form $x_{i_1} \mapsto x_{i_2},\ x_{i_2} \mapsto x_{i_3},\ \ldots, x_{i_k - 1} \mapsto x_{i_k},\ x_{i_k} \mapsto x_{i_1}$ für paarweise verschiedene x_j, sowie $x_m \mapsto x_m$ für $m \notin \{i_1, \ldots, i_k\}$ heißt *k-Zykel* und wird mit $(x_{i_1}\ x_{i_2}\ \ldots\ x_{i_k})$ abgekürzt. Ein 2-Zykel heißt auch *Transposition* (Vertauschung).
Jede Permutation läßt sich als Produkt von Transpositionen schreiben (Beweis siehe etwa [10]).

Beispiel 2. $Sym\ A := \{\sigma | \sigma$ Permutation auf $A\}$ bildet mit Hintereinanderausführung von Abbildungen eine Gruppe, die sogenannte *Symmetrische Gruppe auf* A. $Sym\ \{1, 2, \ldots, n\}$ kürzen wir im folgenden mit S_n ab.
Ist G eine Untergruppe von $Sym\ A$, so nennt man G eine *Permutationsgruppe auf* A.

Nun ist die Grundlage geschaffen für den zentralen Begriff eines Automorphismus:

Definition 3. *Sei* $M = (S, R, L)$ *Kripke-Struktur*[2] *und* G *eine Permutationsgruppe auf* S. *Dann heißt* $\sigma \in G$ Automorphismus von M, *wenn*
$\forall s_1 \in S\ \forall s_2 \in S : (s_1, s_2) \in R \Rightarrow (\sigma(s_1), \sigma(s_2)) \in R$[3]
G *heißt* Automorphismengruppe, *wenn* $\forall \sigma \in G : \sigma$ *Automorphismus*

Anmerkung 2. Die Identität *id* ist ein Automorphismus. Ist σ ein Automorphismus, so auch σ^{-1} (denn jedes $(s, t) \in R$ läßt sich auch als $(\sigma(s_1), \sigma(s_2))$ darstellen, da σ eine Permutation ist). Sind σ_1, σ_2 Automorphismen, so auch $\sigma_1 \circ \sigma_2$. Also bilden die Automorphismen einer Kripke-Struktur M eine Gruppe, die sogenannte *Automorphismengruppe von M.* Wir werden auch Untergruppen dieser Gruppe so bezeichnen.
Vereinfacht kann man sagen:
Je größer die Automorphismengruppe einer Struktur ist, desto symmetrischer ist diese Struktur.

[1] Da der Schnitt von Gruppen wiederum eine Gruppe ist
[2] Das bedeutet: S ist eine Menge von sogenannten Zuständen, $R \subseteq S \times S$ die Übergangsrelation und $L : S \to 2^{AP}$ eine Beschriftung der Zustände mit sogenannten atomaren Aussagen (atomic propositions).
[3] Es handelt sich hier also einfach um Automorphismen des Graphen der Kripke-Struktur

Anmerkung 3. Sind $g_1, g_2, \ldots, g_k$ Automorphismen, so ist $\langle g_1, g_2, \ldots, g_k \rangle$ eine Automorphismengruppe (wegen der gleichen Argumente wie in der vorigen Anmerkung).

Beispiel 3. Sei eine Kripke-Struktur gegeben, wie in Abb. 2 links. Dabei interpretiere man die ungerichteten Kanten als gerichtet in beiden Richtungen, Beschriftungen (L) sind hierfür nicht wichtig und daher weggelassen.
Sei $s = (1\ 2\ 3\ 4\ 5)$ eine Permutation der Zustandsmenge. Dann ist diese Permutation *kein* Automorphismus, da z.B. $(2, 4) \in R$, aber $(s(2), s(4)) \notin R$. Andererseits ist die Permutation $(1\ 4)(2\ 3)$ sehr wohl ein Automorphismus.

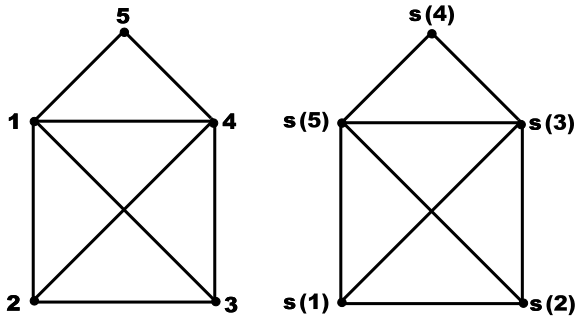


Abb. 2. Zu Beispiel 3

Anmerkung 4. Häufig läßt sich eine Kripke-Struktur in der Form schreiben:
$S \subseteq D^n$
$d_i \in L((x_1, \ldots, x_n)) \Leftrightarrow x_i = d$ (für ein $d \in D$)
Dann induziert eine Permutation $\sigma \in S_n$ eine Permutation σ' auf der Zustandsmenge S via

$$\sigma' : (x_1, \ldots, x_n) \mapsto (x_{\sigma(1)}, \ldots, x_{\sigma(n)})$$

Leicht sieht man, daß so von einer Permutationsgruppe auf der Indexmenge eine Permutationsgruppe auf der Zustandsmenge induziert wird.[4]
Auf solche Weise erzeugte Gruppen sind ein sehr wichtiger Spezialfall, viele Arbeiten auf diesem Gebiet beschränken sich auf derartige Symmetrien (etwa [7]). Wegen Anmerkung 3 reicht es, um eine solche durch Indexpermutationen induzierte Gruppe als Automorphismengruppe zu identifizieren, zu zeigen, daß die von den Erzeugenden induzierten Permutationen Automorphismen sind.

[4] Allgemeiner kann man hier von einer sogenannten Aktion einer beliebigen Gruppe auf der Zustandsmenge der Kripke-Struktur sprechen. Auch die später auftretenden Begriffe einer Bahn und transitiver Gruppen(-aktionen) sind diesem Gebiet entlehnt. Ich verzichte hier hier auf eine formale Einführung und verweise den interessierten Leser auf [2], Kap. 5.2

Sind die Strukturen durch OBDDs derart kodiert, daß jede Komponente genau von einer gewissen Menge von Zustandsbits repräsentiert wird, so läßt sich für eine Indexpermutation relativ leicht überprüfen, ob sie zu einem Automorphismus führt:
Man benennt in dem OBDD für die Übergangsrelation die Variablen entsprechend der Indexpermutation um, bringt den entstehenden BDD auf dieselbe Variablenordnung wie den ursprünglichen und prüft auf Gleichheit.

3 Die Quotientenstruktur

Im folgenden sei stets eine feste Kripke-Struktur $M = (S, R, L)$ über den atomaren Aussagen AP gegeben.

Die kleinere Struktur erhalten wir im wesentlichen dadurch, daß wir Zustände verschmelzen, die bezüglich der folgenden Relation äquivalent sind.

Definition 4. *Sei G Permutationsgruppe auf S. $\Theta \subseteq S \times S$ mit $\Theta(s,t) :\Leftrightarrow \exists \sigma \in G : \sigma(s) = t$ heißt* Orbit-Relation.

Anmerkung 5. Die Orbit-Relation ist eine Äquivalenzrelation, denn:
Reflexivität: $id \in G$
Symmetrie: $\Theta(s,t)$ per $\sigma \Rightarrow \Theta(t,s)$ per σ^{-1}
Transitivität: $\Theta(s,t)$ per σ_1 und $\Theta(t,u)$ per $\sigma_2 \Rightarrow \Theta(s,u)$ per $\sigma_2 \circ \sigma_1$
Die Äquivalenzklasse von $s \in S$ heißt *Orbit* (auch *Bahn*) *von* s und wird mit $\theta(s)$ abgekürzt (also $\theta(s) := \{t | \exists \sigma \in G : \sigma(s) = t\}$).
Ein Repräsentantensystem einer Äquivalenzrelation ist eine Menge bestehend aus je einem Element aus jeder Äquivalenzklasse. Hier bezeichnen wir den Repräsentanten der Äquivalenzklasse von s mit $rep(\theta(s))$.

Definition 5. *Sei G eine Automorphismengruppe von M und ein festes Repräsentantensystem gegeben.*
Dann ist die Quotientenstruktur[5] *$M_G = (S_G, R_G, L_G)$ wie folgt definiert:*

1. $S_G := \{\theta(s) | s \in S\}$
2. $R_G := \{(\theta(s), \theta(t)) |$ *ex.* $s_1, t_1 \in S$ *mit* $\theta(s_1) = \theta(s)$, $\theta(t_1) = \theta(t)$ *und* $(s_1, t_1) \in R\}$
3. $L_G(\theta(s)) := L(rep(\theta(s)))$

Die Definition der Beschriftung L_G ist offensichtlich sehr stark abhängig vom gewählten Repräsentantensystem. Um die Gültigkeit von beliebigen CTL^*-Formeln zu erhalten, reicht diese ebenfalls beliebige Beschriftung im allgemeinen natürlich nicht aus. Wir werden uns daher zunächst auf eine spezielle Klasse von Automorphismengruppen beschränken:

[5] Es handelt sich hierbei um den sogenannten Quotienten nach der Äquivalenzrelation Θ, daher der Name.

Definition 6. *Eine Automorphismengruppe G heißt* Invarianzgruppe für $p \in AP :\Longleftrightarrow \forall \sigma \in G\ \forall s \in S : p \in L(s) \ \Leftrightarrow\ p \in L(\sigma(s))$.
Sei f eine aussagenlogische Formel über AP. Dann heißt eine Automorphismengruppe G Invarianzgruppe für $f :\Leftrightarrow \forall \sigma \in G\ \forall s \in S : L(s) \models f \Longleftrightarrow\ L(\sigma(s)) \models f$[6].

Für Invarianz unter einer Automorphismengruppe genügt Invarianz unter den Erzeugenden (denn jedes Element läßt sich als endliche Kombination der erzeugenden Elemente darstellen, siehe dazu bei Bedarf den Beginn des Beweises von Theorem (3)).

Beispiel 4. In Abb. 3 ist links ein sehr vereinfachtes Modell einer Token-Ring-Struktur angegeben. Die 3 Variablen stehen für 3 Prozesse, die sich jeweils im Zustand „hat Token“(t), „hat Token nicht“(n) oder „hat Token und nutzt kritische Ressource“(c) befinden können.
Betrachtet man analog Anm. 4 eine Permutation $\sigma = (1\ 2\ 3)$ auf der Indexmenge, so induziert die von dieser erzeugte Gruppe $\langle\sigma\rangle = \{id,\ (1\ 2\ 3), (3\ 1\ 2)\}$ eine Automorphismengruppe auf der Token-Ring-Struktur. Eine mögliche Quotientenstruktur ist rechts daneben zu sehen. Betrachtet man die offensichtliche Verallgemeinerung der Originalstruktur auf n parallele Prozesse, so sieht die Quotientenstruktur übrigens unabhängig von n so aus (abgesehen von der Beschriftung), bietet also eine große Kompression der Originalstruktur.

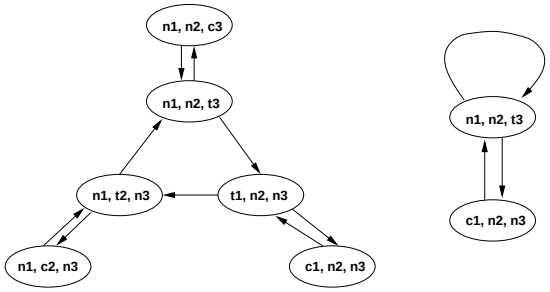


Abb. 3. Aus Beispiel 4: Links die originale Kripke-Struktur, rechts eine mögliche Quotientenstruktur.

4 Ausnutzen der Quotientenstruktur für Model Checking

Wir werden nun zunächst die sehr einschränkende Voraussetzung machen, daß die Automorphismengruppe alle atomaren Aussagen invariant läßt, und zeigen,

[6] Hierbei soll die Belegung von $p \in AP$ genau dann unter $L(s)$ wahr sein, wenn $p \in L(s)$.

daß dann in der Quotientenstruktur dieselben CTL^*-Formeln erfüllt sind wie in der Originalstruktur. Schrittweise wird diese Voraussetzung dann abgeschwächt, wobei man für eine gegebene Formel dann nur noch fordert, daß die Automorphismengruppe bestimmte größere Teilformeln dieser Formel invariant läßt [7].

Lemma 1. *Sei G Invarianzgruppe für alle $p \in AP$. Sei $B \subseteq S \times S_G : \ B(s,\vartheta) \ :\Longleftrightarrow \vartheta = \theta(s)$. Dann ist B eine Bisimulation zwischen M und M_G.*

Beweis. zu zeigen:
(1) $L(s) = L_G(\theta(s))$
(2) $\forall t : \ R(s,t) \ \Rightarrow \ \exists \vartheta : \ R_G(\theta(s),\vartheta)$ und $B(t,\vartheta)$
(3) $\forall \vartheta : \ R_G(\theta(s),\vartheta) \Rightarrow \exists t : \ R(s,t)$ und $B(t,\vartheta)$
zu (1): Klar, da G Invarianzgruppe.
zu (2): Wähle $\vartheta := \theta(t)$. Dann klar mit den Definitionen der Quotientenstruktur und von B.
zu (3): Gelte $R_G(\theta(s),\vartheta)$. Dann existieren nach Definition der Quotientenstruktur $s_1, t_1 \in S$ mit $R(s_1,t_1)$ und $\theta(s_1) = \theta(s), \ \theta(t_1) = \vartheta$.
Nun gilt wegen $\theta(s_1) = \theta(s)$, daß ein $\sigma \in G$ exisiert mit $\sigma(s_1) = s$; und für ein solches gilt, da es sich um eine Automorphismengruppe handelt, $R(s, \sigma(t_1))$.
Da $\sigma(t_1) \in \vartheta$, gilt nach Definition $B(\sigma(t_1),\vartheta)$. □

Mit den Resultaten des Beitrags über Äquivalenzen und Anordnungen von Strukturen folgt:

Korollar 1. *Sei G Invarianzgruppe für alle $p \in AP$, f CTL^*-Formel über AP. Dann folgt: $\forall s \in S : M, s \models f \ \Leftrightarrow \ M_G, \theta(s) \models f$*

Intuitiv dürfte einleuchten, daß es nur auf die in einer Formel tatsächlich vorkommenden atomaren Aussagen ankommt. Dies kann man auch formal beweisen:

Theorem 1. *Sei G Invarianzgruppe für alle $p \in AP' \subseteq AP$. f CTL^*-Formel über AP'. Dann folgt $\forall s \in S : M, s \models f \ \Leftrightarrow \ M_G, \theta(s) \models f$*

Beweis. Sei M' wie M mit $L' := L \cap AP'$, die sogenannte Restriktion von M auf AP'.
Klar:

$$M, s \models f \iff M', s \models f \tag{1}$$

Nun ist auch ${M'}_G$ Restriktion von M_G auf AP', da die Definition von S_G, R_G unabhängig von L ist.
Also:

$$\forall \vartheta \in S_G : M_G, \vartheta \models f \iff {M'}_G, \vartheta \models f \tag{2}$$

Also: $\forall s \in S : M, s \models f \iff$ (wegen (1)) $M', s \models f \iff$ (wegen 1) ${M'}_G, \theta(s) \models f \iff$ (wegen (2)) $M_G, \theta(s) \models f$ □

[7] Natürlich wird dann auch nur noch die Gültigkeit bestimmter Formeln erhalten

Man kann sich sogar auf die in einer Formel vorkommenden maximalen aussagenlogischen Teilformeln beschränken. Aussagenlogische Teilformeln sind Teilformeln, die aus aussagenlogischen Verknüpfungen von atomaren Aussagen bestehen. Sie sind maximal, wenn sie an einer Stelle nicht als Teilformel einer anderen aussagenlogischen Teilformel auftreten.

Beispiel 5. Betrachten wir die Formel

$$AG(c_1 \wedge c_2 \wedge E((a_1 \rightarrow b_1 \wedge a_2 \rightarrow b_2 \wedge a_3 \rightarrow b_3)U(c_1 \wedge c_2 \wedge c_3)$$

über den Variablen $\{a_1, a_2, a_3, b_1, b_2, b_3, c_1, c_2, c_3\}$ und Permutationen der Indexmenge. Dann sind die maximalen aussagenlogischen Teilformeln dieser Formel $a_1 \rightarrow b_1 \wedge a_2 \rightarrow b_2 \wedge a_3 \rightarrow b_3$ (invariant unter Rotationen), $c_1 \wedge c_2 \wedge c_3$ (invariant unter allen Permutationen) und $c_1 \wedge c_2$ (nicht invariant unter Rotationen).

In [7] wird folgender Satz nur für den Spezialfall von Indexpermutationen bewiesen. Ich glaube, daß sich das Resultat analog zum vorhergehenden Satz auf die hier vorliegende Situation verallgemeinern läßt, werde mir einen Beweisversuch jedoch wegen mangelnder Risikobereitschaft sparen.[8]

Theorem 2. *Sei f CTL^*-Formel über $AP' \subseteq AP$ und G eine (durch Indexpermutationen induzierte) Invarianzgruppe* für alle maximalen aussagenlogischen Teilformeln in *f:*
$\Rightarrow \forall s \in S : M, s \models f \;\Leftrightarrow\; M_G, \theta(s) \models f$

Es reicht sogar eine noch etwas schwächere Bedingung, was die Invarianz für eine gegebene Formel betrifft (siehe [7]). Diese ist allerdings komplizierter zu formulieren, weswegen ich sie hier auslasse.

Man mag sich fragen, ob es nicht reicht, die ganze Formel invariant zu lassen, ohne Rücksicht auf die Invarianz von Teilformeln. Die Antwort hierauf ist ein klares Nein:

Beispiel 6. Betrachte etwa die Struktur aus Bsp. 4 bzw. Abb. 3 und die Formel $AG(EFc_1 \wedge EFc_2 \wedge EFc_3)$. Diese Formel gilt in der Originalstruktur, allerdings unabhängig vom gewählten Repräsentantensystem in keiner der Quotientenstrukturen.

5 Konstruktion der Quotientenstruktur

Wir haben gesehen, daß eine Quotientenstruktur unter Umständen großen Nutzen bringen kann. Allerdings ist noch nicht klar, wie man diese effektiv konstruiert. Die Originalstruktur dafür explizit angeben und vollständig durchlaufen zu müssen, wäre sicher nicht hilfreich. Schließlich laufen zum einen die meisten

[8] Beweisidee: Man führe neue aussagenlogische Variablen für die maximalen aussagenlogischen Teilformeln ein, betrachte die Formel f als Formel über diesen Variablen und schließe dann analog zu Theorem (1)

Model-Checking-Algorithmen (insbesondere für CTL und LTL) in linearer Zeit bezogen auf die Größe der Struktur (man könnte also genausogut direkt auf der Originalstruktur Model-Checking betreiben), und zum anderen ist ja gerade der Platzbedarf der Originalstruktur ein wesentlicher Grund dafür, eine kleinere Struktur konstruieren zu wollen.

In diesem Abschnitt stellen wir Verfahren zur Konstruktion vor, im nächsten sehen wir dann gewisse Probleme mit deren Laufzeit und Speicherplatzbedarf. Zuerst eine explizite Konstruktion, dann mithilfe von BDDs.

5.1 Explizite Konstruktion

Ein einfacher Algorithmus zur expliziten Konstruktion einer Quotientenstruktur ist in Abb. 4 angegeben (angelehnt an [16]).

```
S_G := ∅
for all s ∈ S_0 do
   if not ∃u ∈ S_G mit Θ(s,u)
      S_G := S_G ∪ {s}
endfor
unexplored := S_G
While unexplored ≠ ∅ do
   entferne ein s aus unexplored
   for all t Nachfolgerzustände von s do
      if ∃u ∈ S_G mit Θ(u,t)
         R_G := R_G ∪ {(s,u)}
      else
         S_G := S_G ∪ {t}
         R_G := R_G ∪ {(s,t)}
         unexplored := unexplored ∪ {t}
   endfor
endwhile
```

Abb. 4. Ein Algorithmus zur expliziten Konstruktion einer Quotientenstruktur

Dieser Algorithmus basiert auf einem einfachen Durchlauf des Originalgraphen, der dafür jedoch auch in symbolischer oder anderer Form gegeben sein kann. Wichtig ist in der Repräsentation des Originals zunächst, zu einem gegebenen Knoten die Nachfolger bestimmen zu können.

S_0 steht hier für die Anfangszustände, von deren Existenz ausgegangen wird.

Wichtig zu bemerken ist, daß nur jeweils die Nachfolger eines einzigen Knotens pro Äquivalenzklasse betrachtet werden müssen. Dies führt meist zu einer deutlichen Laufzeitreduktion.

Es werden dabei tatsächlich alle in der Originalstruktur erreichbaren Klassen erreicht, da man wie im Beweis von Lemma 1 gesehen eine von einer Klasse

erreichbare andere Klasse auch schon von jedem Zustand der ersteren erreichen kann.

Bezeichnet $|M_G|$ die Größe der Quotientenstruktur (Anzahl Kanten plus Anzahl Knoten), d den maximalen Ausgangsgrad eines Knoten in der Originalstruktur und t die Zeit, um für zwei Knoten zu bestimmen, ob sie in der gleichen Äquivalenzklasse liegen, so liegt die Laufzeit bei $O(|M_G| * d * t)$, berechnet man Θ vorher in Zeit T, so bei $O(|M_G| * d + T)$ (ungefähr). Wichtiger ist, dass der Speicherplatz im wesentlichen durch die Größe der Quotientenstruktur beschränkt ist (abgesehen von der Orbit-Relation, falls man diese vorher berechnet und abspeichert, dazu siehe auch Abschnitt 6).

Dieser Algorithmus (bzw. eine Variante davon) kann beim LTL-Model-Checking mittels Büchi-Automaten auch mit der Konstruktion des Formel-Automaten verzahnt werden, sodaß On-the-Fly-Model-Checking möglich wird (weitere Speicherplatzeinsparung).

5.2 Symbolische Konstruktion

Ist die Originalstruktur als OBDD(s) gegeben und soll die Quotientenstruktur ebenfalls mit OBDDs dargestellt werden, so verkompliziert es sich ein wenig.

Wir werden ab jetzt immer annehmen, daß die Automorphismengruppe nur durch sie erzeugende Elemente gegeben ist ($G = \langle g_1, \ldots, g_k \rangle$).

Definiert man $\zeta : s \mapsto rep(\theta(s))$, so läßt sich die Übergangsrelation der kleineren Struktur z.B. über

$$R_G(x, y) = \exists x_1 \exists y_1 (R(x_1, y_1) \wedge \zeta(x_1) = x \wedge \zeta(y_1) = y)$$

per BDDs berechnen[9].

ζ wiederum kann aus einem BDD für Θ effizient bestimmt werden (siehe [14]).

Bleibt die Frage: Wie bestimmt man Θ?

Theorem 3. *Sei $M = (S, R, L)$ eine Kripke-Struktur und $G = \langle g_1, \ldots, g_k \rangle$ eine Automorphismengruppe von M. Dann ist die Orbit-Relation $\Theta (\subseteq S \times S)$ der kleinste Fixpunkt der folgenden Gleichung:*

$$Y(x, y) = (x = y) \vee \exists z (Y(x, z) \wedge \bigvee_i y = g_i(z)) \tag{3}$$

Beweis.
Vorbemerkung:

G ist als Untergruppe von $Sym\ S$ endlich.

Daher $\forall i \exists j, j' : j < j' \wedge g_i^j = g_i^{j'}$

also $g_i^{j'-j} = id$ und $g_i^{j'-j-1} = g_i^{-1}$ (für ein $j' - j - 1 \geq 0$)

[9] Man kodiert hier also zunächst die Zustände der Quotientenstruktur auf dieselbe Weise wie die der Originalstruktur. Falls man das nicht umgeht, benötigt man somit außerdem einen nichttrivialen BDD für die neue Zustandsmenge.

$\Rightarrow \forall \sigma \in G$ ex. $g_{\sigma_k}, \ldots, g_{\sigma_1} \in G : \sigma = g_{\sigma_k} \cdots g_{\sigma_1}$.
D.h. jedes Element läßt sich als Kombination endlich vieler erzeugender Elemente schreiben, denn daß es sich als endliche Kombination von Erzeugenden und deren Inversen schreiben läßt, ist klar. Wir zeigen nun:
1. Θ ist Fixpunkt
2. Jeder Fixpunkt von Gleichung (3) enthält Θ
zu 1.:
$\supseteq$:
Fall 1: $x = y$
$\Rightarrow \Theta(x, y)$
Fall 2: $\exists z(\Theta(x, z) \wedge \bigvee_i y = g_i(z))$
$\Rightarrow \exists z : \Theta(x, z) \wedge \Theta(z, y)$ (da $g_i \in G$)
$\Rightarrow \Theta(x, y)$ (wegen der Transitivität, da Äquivalenzrelation)
$\subseteq$:
Gelte $\Theta(x, y)$, etwa $\sigma(x) = y$
Fall 1: $x = y$: klar
Fall 2: $x \neq y$:
etwa $\sigma = g_\sigma \sigma'$ (für ein $g_\sigma \in \{g_1, \ldots, g_k\}$ wie in Vorbemerkung),
also für $z := \sigma'(x) : \Theta(x, z)$ und $y = g_\sigma(z) \Rightarrow$ Beh.
zu 2.:
Sei T Fixpunkt von Gleichung (3). Gelte $\Theta(x, z)$.
Dann existiert $\sigma = g_{i_m} \ldots g_{i_2} g_{i_1} : \sigma(x) = z$.
Es gilt $T(x, x)$ (wegen $x = x$) und mit $g_{i_1}(x) =: y$ deshalb auch $T(x, y)$
Induktiv folgt: $T(x, z)$.
Also $\Theta \subseteq T$.

□

Dieser Fixpunkt kann leicht mit BDDs berechnet werden.

6 Probleme der Konstruktionsansätze

Repräsentiert man wie im vorigen Ansatz Θ über einen BDD, so wird dieser leider häufig sehr groß.

Denn oft besteht die Struktur aus N identischen Komponenten bzw. Prozessen mit je k lokalen Zustandsvariablen (-bits) und die Automorphismengruppe ist induziert von einer Gruppe G, die die Rotationsgruppe auf $\{1, 2, \ldots, N\}$ oder die S_N ist (siehe etwa einleitende Beispiele in Abschnitt 1 oder das Ende von Abschnitt 2).

Rotationsgruppen und die S_N sind spezielle sogenannte *transitive* Gruppen, wobei eine Permutationsgruppe G auf $\{1, \ldots, N\}$ transitiv heißt, wenn $\forall i, j \in \{1, \ldots, N\} \exists \sigma \in G : \sigma(i) = j$.[10]

[10] Achtung: Eine transitive Gruppe auf S würde zu nur einem Orbit führen. Die hier gemeinten Gruppen agieren allerdings auf der Indexmenge der Komponenten.

Theorem 4. *Sei $S = B^{Nk}$. Sei G eine transitive Gruppe auf $\{1, \ldots, N\}$. Dann gilt für einen BDD B, der die induzierte Orbit-Relation Θ repräsentiert: $|B| > 2^K$ mit $K = min(\sqrt{N}, 2^{k-1} - 1)$.*

Beweis. Gruppentheoretische Vorbereitungen:
$G_{ij} := \{g \in G | g(i) = j\}$
Seien g_{ij} Elemente aus G mit $g_{ij}(i) = j$ (existieren, da Gruppe transitiv). $G_{ij} = g_{i1} G_{11} g_{j1} \Rightarrow |G_{ij}| = |G_{11}|$ (denn Links- und Rechtstranslation (d.h. -multiplikation) mit einem festen Element sind jeweils bijektive Abbildungen, dafür beachte man die Existenz von Inversen)

Nun zum eigentlichen Beweis:
Sei B ein BDD wie in der Behauptung über Variablen $x_{i,j}, x'_{i,j} (1 \leq i \leq N, 1 \leq j \leq k)$. Wir beachten nur die erste Zustandsvariable jeder Komponente, d.h. die $x_{i,1}$ bzw. die $x'_{i,1}$, und durchlaufen nun die Variablenordnung von Beginn an, bis wir K Variablen der Form $x_{i,1}$ oder K Variablen der Form $x'_{i,1}$ haben.
Seien o.B.d.A. $I = \{i_1, \ldots, i_K\}$ die Menge der Indizes der ungestrichenen Variablen bis hier und J die Menge der gestrichenen bis hier (also $|J| < K$).

$$\begin{aligned} \text{Nun } |\textstyle\bigcup_{i \in I, j \in J} G_{ij}| &\leq |I||J||G_{11}| \text{ (nach Vorbemerkung)} \\ &< K^2 |G_{11}| \text{ (wegen } |I| = K \text{ und } |J| < K) \\ &\leq N |G_{11}| \text{ (wegen } K \leq \sqrt{N}) \\ &= |G| \end{aligned}$$

Also existiert ein $g \in G$ mit $g \notin \bigcup_{i \in I, j \in J} G_{ij}$, d.h. $\forall i \in I : g(i) \notin J$.
Wähle ein solches $g \in G$:
Instanziiere nun für alle $i_j \in I$ die Tupel $(x_{i_j,2}, \ldots, x_{i_j,N})$ und $(x'_{g(i_j),2}, \ldots, x'_{g(i_j),N})$ mit der Binärdarstellung von j (dies ist möglich, da $K \leq 2^{k-1} - 1$) und $x_{i,j}, x'_{g(i),j}$ für $i \notin I$ mit 0.
Der resultierende BDD B' ist kleiner als der ursprüngliche BDD B, hat an freien Variablen nur noch $x_{i,1}, x'_{g(i),1}$ für $i \in I$ und alle ungestrichenen Variablen sind in der Variablenordnung vor den gestrichenen.
Wegen der gewählten Instanziierung ist B' nun BDD für $\bigwedge_{i \in I} (x_{i,1} = x'_{g(i),1})$ und damit $|B'| \geq 2^K$, denn je zwei unterschiedliche Belegungen der ungestrichenen Variablen müssen zu verschiedenen (inneren) Knoten des BDD führen (und von diesen Belegungen gibt es 2^K). □

Der resultierende BDD ist also exponentiell in der Anzahl der Komponenten oder doppelt exponentiell in der Anzahl der Zustandsvariablen einer Komponente. Insbesondere ist der Algorithmus nicht skalierbar.
Damit ist der Umweg über einen BDD für die Orbit-Relation nur für Strukturen mit wenigen Komponenten oder Komponenten mit wenigen Zuständen sinnvoll.

Stattdessen kann man auch (gerade im expliziten Fall macht dies Sinn) Θ direkt berechnen, sobald es gebraucht wird. Dies geht zumindest im allgemeinen leider auch nicht sonderlich schnell.

Definition 7. *Das* Orbit-Problem *ist das folgende Entscheidungsproblem:*
Gegeben: $g_1, \ldots, g_k \in Sym\ \{1, \ldots, n\},\ x, y \in B^n$
Frage: Existiert $\sigma \in \langle g_1, \ldots, g_k \rangle$ mit $\sigma(x) = y$?

Anmerkung 6. Kann man ζ effizient berechnen, so auch dies. („ja“ $\Leftrightarrow \zeta(x) = \zeta(y)$)

Definition 8. *Das* Graph-Isomorphismus-Problem *(kurz: GI) ist das folgende Entscheidungsproblem:*
Gegeben: Zwei Graphen $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ mit Knotenmenge der gleichen Mächtigkeit, d.h. $|V_1| = |V_2|$.
Frage: Sind diese Graphen isomorph, d.h. existiert ein bijektives $\sigma : V_1 \to V_2$ mit $(v_1, v_2) \in E_1$ gdw. $(\sigma(v_1), \sigma(v_2)) \in E_2$?

Theorem 5. *Das Orbit-Problem ist (bzgl. Polynomialzeitreduktion) mindestens so schwer wie das Graph-Isomorphismus-Problem.*

Beweis. Seien $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ mit $|V_1| = |V_2| = n$.
Stelle die G_i durch ihre Adjazenzmatrizen $M_i \in \{0,1\}^{n \times n}$ dar. Die Graphen sind genau dann isomorph (kurz $G_1 \cong G_2$), wenn es eine Permutation der Knoten des einen Graphen gibt, sodaß die entstehende Adjazenzmatrix gleich der des anderen ist.
Schreibt man die Matrizen als 0-1-Vektoren B_i (etwa zeilenweise), so läßt sich eine simultane Zeilen-/Spaltenvertauschung der i-ten und j-ten Zeile/Spalte als Permutation $\sigma(i,j)$ eines solchen Vektors darstellen, etwa
$\sigma(i,j) = \sigma_{row}(i,j)\sigma_{col}(i,j)$ mit:
$\sigma_{row}(i,j) = (n(i-1)+1, n(j-1)+1) \ldots (n(i-1)+n, n(j-1)+n)$
$\sigma_{col}(i,j) = (i,j) \ldots ((n-1)n+i, (n-1)n+j)$
Jede Permutation der Knoten eines Graphen korrespondiert zu einer (nicht unbedingt eindeutig bestimmten) Folge solcher Vertauschungen.
Ferner gilt: $\sigma(i,j) = \sigma(1,i)\sigma(1,j)$.
Also: $G_1 \cong G_2 \iff$ ex. $\sigma \in \langle \sigma(1,2), \sigma(1,3), \ldots, \sigma(1,n) \rangle$ mit $\sigma(B_1) = B_2$ □

Es wird vermutet, daß GI nicht NP-hart ist, aber auch nicht in P.

Eine intuitive Beschreibung der Probleme, die zur Schwierigkeit führen, könnte lauten:

1. Die Größe einer Gruppe kann exponentiell sein in der Anzahl ihrer generierenden Elemente.
2. Man muß zur Lösung eines Problems häufig viele andere Orbit-Probleme nebenbei lösen.

Eine naheliegende Möglichkeit wäre es nun hier (und im symbolischen Fall), ζ direkt zu berechnen und etwa per BDDs abzuspeichern. Dies wird auf gewisse Weise in [1] angewandt. Im expliziten Fall macht es beim oben angegebenen Algorithmus nicht unbedingt Sinn, da man hoffen kann, die Orbit-Relation nur für vergleichsweise wenige Zustandspaare berechnen zu müssen[11], da man ja die Originalstruktur hoffentlich nicht ganz durchläuft.

Man sollte hier auch beachten, daß dies Resultat (d.h. Theorem (5)) natürlich nicht für jede gegebene Gruppe Auswirkungen haben muß. Insbesondere gibt

[11] zu deren Berechnung genügt ja ζ, siehe oben

es für die oben betrachteten Spezialfälle S_N bzw. Rotationsgruppe polynomielle Algorithmen zur Berechnung eines Repräsentanten (die Ideen hierzu und weitere findet man in [12]).

Symmetrische Gruppen Für die Zustandsäquivalenz ist nur wichtig, wieviele der Prozesse oder Komponenten in einem bestimmten lokalen Zustand sind. Bei geeigneter Kodierung sortiere man also die lokalen Zustände etwa nach absteigender Binärdarstellung und erhalte so einen kanonischen Repräsentanten in polynomieller Zeit.

Rotationsgruppen Eine Rotationsgruppe ist erzeugt von $(1\ 2\ 3\ \ldots N-1\ N)$, hat nur N Elemente, also hat auch jede Äquivalenzklasse höchstens N Elemente.

Das Berechnen von Θ für zwei Elemente ist daher in polynomieller Zeit möglich (erzeuge einfach alle Elemente derselben Äquivalenzklasse aus dem einen gegebenen und siehe nach, ob das andere dabei ist)[12].

7 Zusammenfassung und Ausblick

Zunächst haben wir, nach einer Einführung in die benötigten mathematischen Grundlagen, die durch eine Automorphismengruppe einer Kripke-Struktur induzierte Orbit-Relation und deren Quotientenstruktur kennengelernt. Danach Bedingungen, unter denen man diese Struktur zum Model Checking nutzen kann, die häufig eine erhebliche Reduktion des Zustandsraums erlaubt. Wir haben dann eine explizite und eine symbolische, BDD-basierte Konstruktionsmethode kennengelernt. Letztere Methode gilt gemeinhin als gescheitert, da sie im wichtigen Fall mehrerer identischer Komponenten stets exponentiellen Speicherplatzverbrauch hat. Wir haben auch gesehen, dass die Frage, ob sich zwei Zustände in derselben Äquivalenzklasse befinden, im allgemeinen schwierig ist, in wichtigen Spezialfällen jedoch nicht.

Interessant ist sicher noch die bisher nicht besprochene Frage, wie man Automorphismen einer Kripke-Struktur automatisch finden kann. [7] bietet immerhin Heuristiken, die in bestimmten Fällen nebenläufiger Programme mit identischen Prozessen ein schnelles Finden per Hand über den sogenannten Kommunikationsgraphen des Programms erleichtern. In [13] wird es durch einen speziellen Datentyp erleichtert, aber damit eigentlich wieder dem Benutzer zugeschoben. Ein allgemeinerer Ansatz wird in [15] verfolgt. Man sollte noch erwähnen, daß, sofern man sich auf Automorphismen der Struktur beschränkt und die Formel nicht beachtet, auch allgemeine Algorithmen zum Finden von Graphautomorphismen genügen.

Weitere Arbeiten wurden zum Beispiel publiziert zur Kombination von Symmetrie-Reduktion mit anderen Techniken wie Partial Order Reduction ([6]) oder Echtzeit ([9]). Auch spezielle Techniken für die Symmetrie-Reduktion beim Model Checking von Software wurden untersucht (etwa [11]). Alternative Methoden

[12] Die Länge der Darstellung eines Elements ist hier in Bits mindestens N, daher ist der Algorithmus polynomiell

für symbolisches Model-Checking (auch on-the-fly) finden sich in [1].
Zum Abschluß beispielhaft zwei ausgewählte weiterführende Betrachtungen:

7.1 Symmetrie und Fairness

Es sei zunächst an die Fairnessdefinition aus [4] erinnert:
Gegeben $\{P_1, \ldots, P_k\}$ mit $P_j \subseteq S$ $(\forall j)$, so heißt ein Pfad *fair*, wenn aus jedem P_i ein Zustand unendlich oft besucht wird.
Derartige Fairness wird erhalten, wenn die P_i invariant unter G sind, d.h. wenn $\forall \sigma \in G \; \forall j$ gilt: $s \in P_j \iff \sigma(s) \in P_j$, also wenn sich jedes P_i als Vereinigung von Äquivalenzklassen darstellen läßt.
Es gibt jedoch auch andere Fairness-Begriffe, etwa *strong fairness*: Steht ex_i für „Prozeß i wird ausgeführt“ und en_i für „Prozeß i ist ausführbar (enabled)“, so heißt ein Pfad *strongly fair*, wenn er $\bigvee_i (GFen_i \rightarrow GFex_i)$ erfüllt.
Dummerweise läßt von den Indexpermutationen nur die Identität die maximalen aussagenlogischen Teilformeln invariant, eine Reduktion der Struktur mit den kennengelernten Methoden ist somit nicht möglich[13]. In [8] wird als Lösung eine annotierte Quotientenstruktur eingeführt, die sich von der uns bekannten dadurch unterscheidet, daß sie mehrfache Kanten zuläßt, die dann mit bestimmten Permutationen beschriftet sind. Dadurch wird diese Struktur im allgemeinen erheblich größer als die uns bekannte, ist aber nicht mehr abhängig von der zu prüfenden Formel. Mithilfe von Büchi-Automaten können so auch andere Fairness-Notationen geprüft werden, im Rahmen von LTL beziehungsweise sogenannten *Fair-Indexed-CTL**-Formeln. Eine Erweiterung dieses Ansatzes wurde im Model-Checker SMC implementiert.

7.2 Virtual Symmetry Reduction

In [5] wird gezeigt, daß man sich nicht immer auf Automorphismengruppen beschränken muß, wenn man zusätzlich die Symmetrie einzelner Zustände betrachtet.
Formaler: Sei M eine Kripke-Struktur, G eine Permutationsgruppe auf deren Zustandsmenge.
Sei M^G wie M, aber mit $R_{M^G} = \{(g(s), g(t)); g \in G \text{ und } (s,t) \in R_M\}$ (man fügt Kanten hinzu, damit G auf M^G eine Automorphismengruppe ist).
Dann gilt:
$B = \{(s, \theta(s)) | s \in S\}$ ist Bisimulation (bis auf Kantenbeschriftung) zwischen M und $M_G \iff$ für alle $(s,t) \in R_{M^G}$ existiert ein $g \in G$, sodaß $(s, g(t)) \in R_M$
Insbesondere gelten dann die Sätze aus Abschnitt 4, wenn die Permutationsgruppe die geforderten Invarianzeigenschaften bezüglich der Beschriftung hat.
Als Anwendung wird etwa ein Leser/Schreiber-Problem mit Schreiber-Priorität genannt.

[13] Wenigstens nicht, wenn als Variablen nur die en_i und die ex_i auftauchen

Literatur

1. Barner, S., Grumberg, O.: Combining Symmetry-Reduction and Under-Approximation for Symbolic Model Checking. Proceedings of the 14th International Conference on Computer Aided Verification (CAV). LNCS 2404, Springer-Verlag (2002) 93-106
2. Bosch, S.: Algebra. (4.Auflage) Springer-Verlag (2001)
3. Clarke, E., Enders, R., Filkorn, T., Jha, S.: Exploiting Symmetry in Temporal Logic Model Checking. Formal Methods in System Design **9(1/2)** (1996) 77-104
4. Clarke, E., Grumberg, O., Peled, D.: Model Checking. The MIT Press (1999)
5. Emerson, E.A., Havlicek, J.W., Trefler, R.J.: Virtual Symmetry Reduction. 15th Annual IEEE Symposium on Logic in Computer Science (LICS), Santa Barbara, California (2000)
6. Emerson, E.A., Jha, S., Peled, D.: Combining partial order and symmetry reductions. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97). LNCS 1217, Springer-Verlag (1997) 19-34
7. Emerson, E.A., Sistla, A.P.: Symmetry and Model Checking. Formal Methods in System Design **9(1/2)** (1996) 105-130
8. Emerson, E.A., Sistla, A.P.: Using Symmetry When Model Checking under Fairness Assumptions: An Automata Theoretic Approach. ACM Transactions on Programming Languages **19(4)** (1997)
9. Emerson, E.A., Trefler, R.J.: Model checking real-time properties of symmetric systems. Proc. of the 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS) (1998) 427-436
10. Fischer, G.: Lineare Algebra. (11.Auflage) Verlag Vieweg (1997)
11. Godefroid, P.: Exploiting Symmetry when Model-Checking Software. Proceedings of FORTE/PSTV'99 (Formal Methods for Protocol Engineering and Distributed Systems), Peking (1999) 257-275
12. Hung, W., Aziz, A., McMillan, K.: Heuristic Symmetry Reduction for Invariant Verification. IEEE/ACM International Workshop on Logic Synthesis (IWLS), Tahoe City, California (1997)
13. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal Methods in System Design **9(1/2)** (1996) 41-75
14. Lin, B., Newton, R.: Efficient Symbolic Manipulation of Equivalence Relations and Classes. International Workshop on Formal Methods in VLSI Design, Miami, Florida (1991)
15. Manku, G.S., Hojati, R., Brayton, R.: Structural Symmetry and Model Checking. Proc. 10th International Conference on Computer-Aided Verification (CAV), Vancouver, Canada. LCNS 1427, Springer-Verlag (1998) 159-171
16. Sistla, A.P.: Symmetry Reductions Tutorial Slides (VMCAI03 Talk). http://www.cs.uic.edu/ sistla/symred.pdf (2003)

Diskrete Echtzeit

Jochen Eisinger

Institut für Informatik
Albert-Ludwigs-Universtität Freiburg
eisinger@informatik.uni-freiburg.de

Zusammenfassung. Echtzeit Systeme müssen korrekte Antworten innerhalb definierter Zeitfenster geben. Für ihre Verifikation muss nicht nur das korrekte Verhalten bewiesen werden, sondern auch das Einhalten der Zeitfenster. Kann das Echtzeit System mit diskreter Zeit beschrieben werde, so lassen sich herkömmliche Techniken zur Verifikation von Modellen anwenden.

Einleitung

Computer werden häufig für zeitkritische Anwendungen eingesetzt, bei denen nicht nur das korrekte Verhalten sondern auch das Einhalten definierter Zeitfenster wichtig ist. Solche Systeme werden als *Echtzeit Systeme* bezeichnet. Beispiele für solche Systeme sind Controller in Fahrzeugen, industriellen Anlagen und Robotern.

Kann ein solches System unter Voraussetzung diskreter Zeit beschrieben werden, so kann sein korrektes zeitliches Verhalten mit Hilfe herkömmlicher formalen Techniken verifiziert werden.

In dieser Seminararbeit werde ich den Begriff *diskretes Echtzeit System* definieren und Ansätze zur Verifikation deren zeitlichen Verhaltens aufzeigen. Außerdem werde ich die Logik *RT-CTL* einführen, mit der zeitliches Verhalten von Modellen beschrieben werden kann, sowie auf dieser Logik aufbauende Algorithmen.

1 Diskrete Echtzeit Systeme

1.1 Definition

Echtzeit Systeme unterscheiden sich von anderen Systemen im wesentlichen dadurch, dass sie periodisch Aufgaben ausführen, die innerhalb fester Zeitlimits beendet werden müssen.

Kann ein solches System unter der Voraussetzung diskreter Zeit beschrieben werden, so spricht man von einem *diskreten Echtzeit System*. Dies ist der Fall bei synchronen Systemen mit einem globalen Takt, wie zum Beispiel Controllern oder Protokollen.

Eine andere geläufige Definition von Echtzeit Systemen unterscheidet solche Systeme von anderen durch reellwertige Zeitangaben. Diese Definition ist für den

vorliegenden Fall zu allgemein. Auf solche Systeme wird im nächsten Kapitel näher eingegangen.

Definition 1. *Ein solches* diskretes Echtzeit System *ist durch folgende Angaben gegeben:*

- *Eine Anzahl n unabhängiger Aufgaben $\tau_1, \tau_2, \ldots \tau_n$*
- *Eine Periode T_i für jede Aufgabe $i = 1 \ldots n$. Die Rate der Aufgabe τ_i ist $1/T_i$*
- *Eine maximal benötigte Laufzeit C_i für jede Aufgabe $i = 1 \ldots n$*

Definition 2. *Weiterhin werde folgende Begriffe im Zusammenhang mit Echtzeit Systemen definiert:*

- *Das* Zeitlimit*, innerhalb dessen eine Aufgabe ausgeführt werden muss, ist der Zeitpunkt, zu dem diese Aufgabe das nächste Mal ausgeführt werden soll. Daraus folgt sofort, dass $C_i \leq T_i$ für alle i gelten muss, da sonst das Zeitlimit nie eingehalten werden kann.*
- *Man spricht von einem* Überlauf *zum Zeitpunkt t, wenn das Zeitlimit einer Aufgabe zum Zeitpunkt t überschritten wurde.*
- *Ist eine Menge von Aufgaben für ein Echtzeit System so gegeben, und ist es möglich, diese ohne Überläufe auszuführen, so nennt man diese Aufgaben* durchführbar*.*
- *Die* Antwortzeit *ist die Dauer, die von einer Anfrage für eine Aufgabe bis zu deren vollständiger Abarbeitung vergeht.*
- *Der Zeitpunkt t, zu dem die Antwortzeit für eine bestimmte Aufgabe am längsten ist, wenn sie zu diesem Zeitpunkt angefordert wird, wird* kritischer Zeitpunkt *für diese Aufgabe genannt.*

Mit diesen Definition kann bereits geschlossen werden:

Lemma 1. *Eine Menge von Aufgaben ist genau dann durchführbar, wenn alle Aufgaben zu ihren kritischen Zeitpunkten ihre jeweiligen Zeitlimits nicht überschreiten.*

Die Aufgabe des Echtzeit Systems ist es also, die zur Verfügung stehende Rechenzeit den einzelnen Aufgaben zuzuteilen, so dass kein Überlauf eintritt. Das Zuteilen der Rechenzeit übernimmt dabei der so genannte Scheduler.

Definition 3. *Ein* Scheduler *entscheidet, welche Aufgabe ausgeführt wird, falls mehrere Aufgaben anstehen. Die Entscheidung, welche Aufgabe ausgeführt werden soll, wird aufgrund der Priorität getroffen, die der Aufgabe zugeordnet ist. Dabei kann weiter unterschieden werden zwischen*

- nicht preemptiven Scheduler: *die Aufgabe mit der höchsten Priorität wird ausgeführt, sobald die Rechenzeit zur Verfügung steht.*
- preemptiven Scheduler: *steht eine Aufgabe höherer Priorität an, als die der Aufgabe, die augenblicklich ausgeführt wird, so wird deren Ausführung unterbrochen und die Aufgabe mit der höheren Priorität erhält die Rechenzeit.*

Mit dieser Definition lässt sich ein Scheduler auf den Algorithmus reduzieren, der den einzelnen Aufgaben die Prioritäten zuteilt. Diese Zuteilung kann entweder statisch erfolgen, wenn das Echtzeit System zusammengestellt wird, oder dynamisch während der Laufzeit.

Wird ein statischer Scheduler Algorithmus verwendet, ist die Verifikation einfach und es existieren optimale Algorithmen, die immer, falls eine Menge von Aufgaben durchführbar ist, eine entsprechende Prioritäten Verteilung finden. Allerdings kann auch gezeigt werden, dass die Auslastung der Rechenzeit durch einen statischen Scheduler Algorithmus suboptimal ist. Ein solcher optimaler Algorithmus wird im nächsten Kapitel vorgestellt.

Dynamische Scheduler erlauben es, die vorhandene Rechenzeit effizienter zu nutzen, die Verifikation ihres Laufzeitverhaltens ist aber mit herkömmlichen Analysemethoden wesentlich schwieriger bzw. nicht durchführbar, so dass sie bei dem häufig konservativen Design von Echtzeit System selten zum Einsatz kommen.

Es existieren auch gemischte Ansätze, bei denen ein Teil der Aufgaben statische und ein anderer Teil dynamische Prioritäten zugewiesen bekommt. Wie die dynamischen Scheduler Algorithmen werden auch diese Ansätze hier nicht weiter diskutiert.

Weiterhin existieren Scheduler für Mehrprozessor Systeme, auch diese werden hier nicht behandelt.

2 Rate Monotonic Scheduler

Definition 4. *Der so genannte* Rate Monotonic Scheduler *ist ein optimaler statischer Scheduler Algorithmus. Prioritäten werden nach der Rate der Anfragen der jeweiligen Aufgaben zugewiesen. Um so höher die Rate ist, um so höher ist die Priorität.*

Lemma 2. *Der Rate Monotonic Scheduler ist ein optimaler Algorithmus.*

Beweis. Seien τ_1, τ_2 mit $T_1 < T_2$ gegeben.

Angenommen, τ_1 hat eine höhere Priorität als τ_2. Dann muss wegen Lemma 1 gelten, dass

$$\lfloor T_2/T_1 \rfloor C_1 + C_2 \leq T_2. \tag{1}$$

(notwendige, aber nicht hinreichende Bedingung)

Angenommen, τ_2 hat die höhere Priorität, muss gelten, dass

$$C_1 + C_2 \leq T_1. \tag{2}$$

$$\begin{aligned}
(2) \quad &\Rightarrow \quad C_1 + C_2 \leq T_1 \\
&\Rightarrow \quad \lfloor T_2/T_1 \rfloor C_1 + \lfloor T_2/T_1 \rfloor C_2 \leq \lfloor T_2/T_1 \rfloor T_1 \\
&\Rightarrow \quad \lfloor T_2/T_1 \rfloor C_1 + \lfloor T_2/T_1 \rfloor C_2 \leq T_2 \\
&\Rightarrow \quad \lfloor T_2/T_1 \rfloor C_1 + C_2 \leq T_2 \quad (\text{da } T_2/T_1 \geq 1)
\end{aligned}$$

Ist also $T_1 < T_2$ und C_1, C_2 sind so gewählt, dass die Aufgaben bei τ_2 mit höherer Priorität durchführbar sind, so sind sie auch bei τ_1 mit höherer Priorität durchführbar.

Angenommen, der Rate Monotonic Scheduler findet keine Prioritäten, so dass eine gegebene Menge von Aufgaben durchführbar ist, es existiert aber eine solche Verteilung. Eine solche Verteilung kann man durch paarweises Vertauschen in eine zweite Verteilung überführen, deren Prioritäten nach Ausführungsrate geordnet sind, und mit der die Menge der Aufgaben auch durchführbar ist.

Dies ist ein Widerspruch zur Annahme, also gibt es eine solche Verteilung nicht, wenn der Rate Monotonic Scheduler keine Verteilung liefert.

2.1 Verifikation

Mit Hilfe des Rate Monotonic Schedulers kann eine gegebene Menge von Aufgaben auf Durchführbarkeit untersucht werden.

Um eine Menge von Aufgaben auf Durchführbarkeit zu untersuchen, werden die Aufgaben zunächst nach ihrer Anfrage Häufigkeit sortiert. Dann wird für jede Aufgabe τ_i überprüft, ob sie zu ihren kritischen Zeitpunkten noch innerhalb ihres Zeitlimits durchführbar ist, das heißt, wenn alle Aufgaben mit höherer Priorität gleichzeitig mit τ_i angefordert werden.

Im folgenden wird eine konkrete Formel hergeleitet, mit der die Durchführbarkeit einer gegebenen Menge von Aufgaben überprüft werden kann.

Definition 5. *Die* Auslastung U *des Rechners für eine gegebene Menge von* n *Aufgaben ist*

$$U := C_1/T_1 + C_2/T_2 + \cdots + C_n/T_n$$

Die Auslastung des Rechners kann also erhöht werden, indem entweder die Laufzeiten C_i erhöht oder die Zeitlimits T_i erniedrigt werden. Eine Menge von Aufgaben lastet den Rechner vollständig aus, wenn gilt:

1. Die Menge der Aufgaben ist durchführbar.
2. Jede Erhöhung der Laufzeit der Laufzeit oder Erniedrigung des Zeitlimits führt dazu, dass die Menge der Aufgaben nicht mehr durchführbar ist.

Die kleinste obere Grenze für die Auslastung ist das Minimum aller U von Aufgabenmengen, die den Rechner vollständig auslasten. Alle Mengen von Aufgaben, deren Auslastung unter diese Grenze fallen, sind mit dem Rate Monotonic Scheduler durchführbar.

Lemma 3. *Für zwei Aufgaben ist die kleinste obere Grenze für die Auslastung*

$$U = 2(2^{1/2} - 1)$$

Beweis. (Ansatz) Seien τ_1, τ_2 Aufgaben mit Zeitlimits T_1, T_2 und Laufzeiten C_1, C_2 mit $T_1 < T_2$. Entsprechend hat τ_1 eine höhere Priorität als τ_2.

Während des kritischen Zeitraumes von τ_2 wird τ_1 $\lceil T_2/T_1 \rceil$ Mal angefordert.

Wir erhöhen C_2 so lange, bis sämtlich zur Verfügung stehende Prozessorzeit während des kritischen Zeitraumes verwendet wird. Dann treten zwei Fälle auf:

1. C_1 ist kurz genug, so dass all Anfragen für τ_1 in dem kritischen Zeitraum von τ_2 beendet sind, bevor τ_2 das nächste Mal angefordert wird:

$$\begin{aligned} & C_1 \leq T_2 - T_1 \lfloor T_2/T_1 \rfloor \\ \Rightarrow \quad & C_2 = T_2 - C_1 \lceil T_2/T_1 \rceil \\ \Rightarrow \quad & U = 1 - C_1((1/T_2)\lceil T_2/T_1 \rceil - 1/T_1) \end{aligned}$$

   In diesem Fall fällt U mit steigendem C_1 monoton.
2. Die $\lceil T_2/T_1 \rceil$-te Anfrage für τ_1 überlappt die nächste Anforderung von τ_2:

$$\begin{aligned} & C_1 \geq T_2 - T_1 \lfloor T_2/T_1 \rfloor \\ \Rightarrow \quad & C_2 = (T_1 - C_1)\lfloor T_2/T_1 \rfloor \\ \Rightarrow \quad & U = (T_1/T_2)\lfloor T_2/T_1 \rfloor + C_1((1/T_1) - (1/T_2)\lfloor T_2/T_1 \rfloor) \end{aligned}$$

   In diesem Fall steigt U mit steigendem C_1 monoton.

Die Auslastung U wird also minimal bei $C_2 = T_2 - T_1 \lfloor T_2/T - 1 \rfloor$, also

$$\begin{aligned} U &= 1 - (T_1/T_2)(\lceil T_2/T_1 \rceil - (T_2/T_1))((T_2/T_1) - \lfloor T_2/T_1 \rfloor) \\ &= 1 - f(1-f)/(l+f) \\ &\text{mit} \\ & l = \lfloor T_2/T_1 \rfloor \\ & f = (T_2/T_1) - \lfloor T_2/T_1 \rfloor \end{aligned}$$

U steigt monoton mit l. Wegen $T_2 > T_1$ ist $l = 1$ das Minimum für l.

Damit wird U für $f = 2^{1/2} - 1$ minimal, es ist also

$$\min U = 2(2^{1/2} - 1)$$

(ohne Beweis) Für eine Menge von n Aufgaben ist die kleinste untere Grenze $U = n(2^{1/n} - 1)$. □

Mit Lemma 3 kann man nun eine Menge von Aufgaben auf Durchführbarkeit überprüfen: Eine Menge von n Aufgaben $\tau_1, \ldots, \tau_n$ mit Laufzeit $C_1, \ldots, C_n$ und Zeitlimits $T_1, \ldots, T_n$ ist dann durchführbar, wenn gilt

$$C_1/T_1 + C_2/T_2 + \cdots C_n/T_n \leq n(2^{1/n} - 1)$$

2.2 Diskussion

Der Rate Monotonic Scheduler ist ein optimaler statischer Scheduler. Obwohl er starke Restriktionen für die Definition von Aufgaben erfordert, wird er in der Praxis häufig eingesetzt. Da für Echtzeit Systeme das Einhalten der Zeitlimits von enormer Wichtigkeit ist, ist besonders die Möglichkeit, die Durchführbarkeit einer Menge von Aufgaben einfach abzuschätzen, ein Grund, dass der Rate Monotonic Scheduler eingesetzt wird.

Nachteil dieser Methode ist vor allem die restriktive Definition eines diskreten Echtzeit Systems und die Einschränkung auf statische Scheduler Algorithmen. Weiterhin liefert die Analyse nur eine Aussage, ob die gesamte Menge der Aufgaben sicher durchführbar ist. Auch wenn die Analyse ein negatives Ergebnis liefert, ist es möglich, dass eine gegebene Menge von Aufgaben durchführbar ist.

Auch welche Aufgabe wann ihr Zeitlimit um wie viele Takte überschreitet, oder, falls die Menge der Aufgaben durchführbar ist, wie viel Toleranz noch vorhanden ist, wird bei dieser Methode nicht bestimmt.

Neuere Arbeiten[3] heben einige Restriktionen für die Aufgaben, die modelliert werden können, auf: Eigenschaften, die nicht über ihre Laufzeit modelliert werden können, oder verteilte Systeme, bzw. Systeme, die kein periodisches Verhalten haben.

3 Verwandte Methoden

Um die Einschränkungen des Rate Monotonic Schedulers zu umgehen, wurden verschiedene andere Methoden entwickelt. Diese Methoden basieren jeweils auf anderen Techniken aus dem Bereich Verifikation, die für diese Aufgabe angepasst wurden.

3.1 Erreichbarkeitsanalyse

Wenn das Echtzeit System als endlicher Automat modelliert werden kann, so kann die Durchführbarkeit mit einer *Erreichbarkeitsanalyse* überprüft werden. Dafür wird dem Automat ein neuer Zustand hinzugefügt und der Automat wird so modifiziert, dass im Falle der Überschreitung eines Zeitlimits der Automat in diesen neuen Zustand geht.

Ist dieser Zustand nicht erreichbar, wird folglich kein Zeitlimit überschritten, und das Echtzeit System, dass von diesem Automat modelliert wurde, ist durchführbar.

Vorteil dieser Methode ist, dass alle Restriktionen für modellierbare Echtzeit Systeme wegfallen. Alle Echtzeit Systeme, die als endlicher Automat darstellbar sind, können so untersucht werden.

Aber auch diese Methode liefert keine quantitativen Aussagen, sondern nur boolesche Ergebnisse. Eigenschaften, die nicht mit Hilfe von Zuständen modelliert werden können, können mit der Erreichbarkeitsanalyse nicht verifiziert werden.

Obwohl viele Eigenschaften so modelliert werden können, ist der gesamte Vorgang schwierig und fehleranfällig.

3.2 Model Checking

Es ist möglich, Eigenschaften diskreter Echtzeit Systeme mit Hilfe von symbolischen Model Checking Techniken zu verifizieren.

Temporale Logiken scheinen auf den ersten Blick geeignet, um Eigenschaften diskreter Echtzeit System zu beschreiben. Jedoch gibt es keinen einfach und effizienten Weg, Eigenschaften diskreter Echtzeit Systeme durch Ausdrücke temporaler Logiken zu beschreiben.

Symbolisches Model Checking kann aber, wenn auch nicht elegant, verschiedene Eigenschaften von diskreten Echtzeit Systemen verifizieren. Es ist aber schwierig, komplexe Eigenschaften von Zeitverhalten auszudrücken. Es ist möglich auszudrücken, dass ein Ereignis in der Zukunft eintreten wird, aber es ist nicht ohne weiteres möglich, eine Eigenschaft zu formulieren, dass ein Ereignis nach maximal n Schritten eintreten wird.

Im weiteren Verlauf dieser Ausarbeitung werden Techniken diskutiert, mit deren Hilfe durch symbolisches Model Checking Eigenschaften diskreter Echtzeit Systeme leichter verifiziert werden können, und eingeschränkte quantitative Analyse durchgeführt werden können.

Diese Techniken sind auch in frei verfügbaren Programmen realisiert, mit deren Hilfe im Anschluss die Verifikation eines einfachen diskreten Systems demonstriert wird.

4 RT-CTL Model Checking

In diesem Kapitel werden Methoden entwickelt, um diskrete Echtzeit Systeme zu spezifizieren und zu verifizieren, die auf herkömmlichen Methoden des symbolischen Model Checking aufbauen. Weiterhin werden Algorithmen vorgestellt, mit denen quantitative Informationen über das Modell ermittelt werden können.

Ein wichtiger Vorteil dieses Ansatzes ist es, dass der Designer anhand der quantitativen Informationen überprüfen kann, ob das Modell verschiedene Zeitbedingungen einhält: Durchführbarkeit kann überprüft werden, indem die maximale Antwortzeit für die verschiedenen Aufgaben ermittelt wird, Reaktionszeiten und andere Parameter können bestimmt werden. Diese Informationen können dazu genutzt werden, Stellen des Designs zu finden, die weiter optimiert werden können. Da diese Analysen bereits durchgeführt werden können, bevor das System produziert wird, lassen sich dadurch Entwicklungskosten sparen.

4.1 RT-CTL

Der oben beschriebene Ansatz basiert grundlegend darauf, dass die Anzahl der Berechnungsschritte zwischen zwei Ereignissen gezählt werden kann. Später werden wir sehen, dass das ein Spezialfall davon ist, die Häufigkeit des Auftretens eines dritten Ereignissen zwischen zwei Ereignissen zu ermitteln.

Diese Methode eignet sich in erster Linie für synchrone Designs und Protokolle. Asynchrone Designs erfordern andere Verifikationsmethoden, wie sie im Kapitel über kontinuierliche Echtzeit Systeme vorgestellt werden. Viele Designs sind aber synchron und werden daher von dieser Methodik erfasst. Ein synchrones Design wird für Echtzeit Systeme oft angestrebt, da ist die Vorhersagbarkeit

des zeitlichen Verhaltens des Systems erhöht und es damit erleichtert, die engen Zeitlimits, die das Echtzeit System einhalten muss, zu realisieren.

Ein einfacher Weg, dies zu erreichen, ist, die Logik CTL um temporale Operatoren zu erweitern. Die erweiterte Logik wird *RT-CTL* genannt. Durch die Erweiterung ändert sich die Ausdrucksstärke von CTL nicht, d.h. die beiden Logiken sind äquivalent. RT-CTL erlaubt es, zeitlich begrenzte Ausdrücke zu formulieren. Diese könnten auch durch eine Schachtelung von EX oder AX Operatoren erreicht werden, die Schreibweise in RT-CTL ist aber wesentlich kompakter und verständlicher.

Die Logik RT-CTL ist eine Erweiterung von CTL um zeitlich begrenzte Operatoren. Der grundlegende zeitlich begrenzte Operator von RT-CTL ist der begrenzte *until* Operator. Der begrenzte *until* Operator hat die Form $\mathbf{U}_{[a,b]}$, wobei $[a, b]$ den Zeitintervall definiert, innerhalb dessen die beschriebene Eigenschaft gelten muss:

$f\mathbf{U}_{[a,b]}g$ ist wahr für einen Pfad $\pi = s_0, s_1, \ldots$ when g in irgendeinem Zustand s in der Zukunft wahr ist, f in allen Zuständen von s_0 bis s wahr ist, und die Entfernung von s_0 zu s im Intervall $[a, b]$ liegt.

Der begrenzte *global* Operator kann genauso definiert werden.

Definition 6. *Die Logik* RT-CTL *erweitert CTL also um zeitliche begrenzte Version von* EU *und* EG*, indem die folgenden semantischen Konstrukte eingeführt werden*

- $s \models \mathbf{E}[f\mathbf{U}_{[a,b]}g]$ *gdw. es existiert ein Pfad* $\pi = s_0 s_1 s_2 \ldots$ *der bei* $s = s_0$ *anfängt und es existiert ein* i *mit* $a \leq i \leq b$*, so dass* $s_i \models g$ *und für alle* $s_j \models f$ *für alle* $j < i$*.*
- $s \models \mathbf{EG}_{[a,b]}f$ *gdw. es existiert ein Pfad* $\pi = s_0 s_1 s_2 \ldots$ *der bei* $s = s_0$ *anfängt, so dass* $s_i \models f$ *für alle* $a \leq i \leq b$*.*

Mit dieser Definition kann zum Beispiel die Eigenschaft „Es ist immer wahr, dass auf p innerhalb von drei Takten q folgt“ folgendermaßen ausgedrückt werden: $\mathbf{AG}(p \rightarrow \mathbf{EF}_{[0,3]}q)$. Der begrenzte *finally* Operator kann von dem begrenzten *until* Operator genauso wie im unbegrenzten Fall abgeleitet werden, also

$$\mathbf{EF}_{[a,b]}f \equiv [true\mathbf{U}_{[a,b]}f].$$

4.2 Fixpunkt Berechnung für RT-CTL Formeln

Zur Verifikation können RT-CTL ähnlich wie CTL Formeln mit Hilfe von Fixpunkt Iteration ausgewertet werden, die auch für CTL zum Einsatz kommt.

Der Operator $\mathbf{E}[f\mathbf{U}_{[a,b]}g]$ wird folgendermaßen berechnet:

$$\mathbf{E}[f\mathbf{U}_{[a,b]}g] = \begin{cases} f \wedge \mathbf{EX}\ \mathbf{E}[f\mathbf{U}_{[a-1,b-1]}g] & \text{wenn } a > 0 \text{ und } b > 0 \\ g \vee (f \wedge \mathbf{EX}\ \mathbf{E}[f\mathbf{U}_{[0,b-1]}g]) & \text{sonst, wenn } b > 0 \\ g & \text{sonst} \end{cases}$$

Der Operator $\mathbf{EG}_{[a,b]}f$ wird folgendermaßen berechnet:

$$\mathbf{EG}_{[a,b]}f = \begin{cases} \mathbf{EX}\ \mathbf{EG}_{[a-1,b-1]}f & \text{wenn } a > 0 \text{ und } b > 0 \\ f \vee \mathbf{EX}\ \mathbf{EG}_{[0,b-1]}f & \text{sonst, wenn } b > 0 \\ \mathit{true} & \text{sonst} \end{cases}$$

4.3 Quantitative Analyse

Traditionelle temporale Verifikation arbeitet so, dass vom Designer explizit vorgegebene Eigenschaften über das zeitliche Verhalten überprüft werden. Hierbei wird keine Information darüber gewonnen, wie weit das System von den vorgegebenen Grenzen entfernt ist. Trotzdem werden diese Angaben benötigt, um das Echtzeit System so weit wie möglich zu optimieren.

In diesem Abschnitt werden Algorithmen vorgestellt, mit denen die minimale und maximale Anzahl von Berechnungsschritten zwischen zwei Ereignissen berechnet werden können, also zum Beispiel der Anfrage einer Aufgabe und deren vollständiger Durchführung.

Die Algorithmen sind darauf ausgelegt, in einen symbolischen Model Checker integriert zu werden, so dass sie mit Hilfe von BDDs effizient zu implementieren sind.

4.4 Minimum Delay Algorithmus

Der Minimum Delay Algorithmus ermittelt die kleinste Anzahl von Schritten in einem gegebenen Modell, also Transitionen zwischen Zuständen, die benötigt werden, um von einem bestimmten Zustand in einen anderen zu gelangen. Dies kann zum Beispiel die minimale Antwortzeit seit, als die Zeit, die zwischen der Anfrage und der Bearbeitung einer bestimmten Aufgabe vergeht.

In Abbildung 1 ist der Minimum Delay Algorithmus definiert. Als Eingabe nimmt der Algorithmus eine Kripke Struktur $M = (S, R, L)$ und zwei Zustandsmengen *start* und *final*. Das Ergebnis ist die minimale Länge eines Pfades von einem Zustand aus *start* bis zu einem Zustand in *final*. Wenn kein solcher Pfad existiert wird unendlich zurückgeliefert.

In der Definition des Algorithmus liefert die Funktion $T(S)$ die Menge der Nachfolgezustände aller Zustände in S zurück, also $T(S) = \{s' | \exists s \in S : R(s, s')\}$. Die Variablen Z und Z' bezeichnen dabei Zustandsmengen.

Der Algorithmus selber ist relativ einfach aufgebaut. Ausgehend von der Zustandsmenge *start* werden solange die Folgezustände hinzugenommen, bis der Schnitt mit der Zustandsmenge *final* nicht mehr leer ist, dann wurde ein minimaler Pfad gefunden, oder die Zustandsmenge sich nicht mehr ändert, dann existiert kein Pfad von Zuständen aus *start* zu Zuständen aus *final*.

4.5 Maximum Delay Algorithmus

Der Maximum Delay Algorithmus ermittelt die Anzahl der Übergänge in der Kripke Struktur, die auf dem längsten Pfad zwischen zwei Ereignissen benötigt

```
procedure min(start, final)
    i := 0;
    Z := start;
    Z' := T(Z) ∪ Z;
    while ((Z' ≠ Z) ∧ (Z∩ final) = ∅) do
        i := i + 1;
        Z := Z';
        Z' := T(Z') ∪ Z';
    end while;
    if (Z∩ final ≠ 0) then
        return i;
    else return ∞;
    end if;
end procedure
```

Abb. 1. Der *Minimum Delay* Algorithmus

werden. Diese Angabe liefert Informationen zum Beispiel darüber, ob die maximale Verzögerung von einer Anfrage bis zu deren Bearbeitung das Zeitlimit übersteigen kann oder nicht. Außerdem ist daraus ersichtlich, wie nah die maximale Ausführungszeit an das Zeitlimit herankommt.

```
procedure max(start, final)
    i := 0;
    Z := ∅;
    Z' := ¬ final;
    while ((Z' ≠ Z) ∧ (Z'∩ start ≠ ∅)) do
        i := i + 1;
        Z := Z';
        Z' := T^-1(Z') ∩ ¬ final;
    end while;
    if (Z = Z') then
        return ∞;
    else return i;
    end if;
end procedure
```

Abb. 2. Der *Maximum Delay* Algorithmus

In Abbildung 2 ist der Maximum Delay Algorithmus definiert. Wie der Minimum Delay Algorithmus nimmt er eine Kripke Struktur $M = (S, R, L)$ und zwei Zustandsmengen *start* und *final*. Die Funktion $T^{-1}(S)$ ist die Umkehrung von $T(S)$, sie bildet eine Menge von Zuständen S auf die Zustände ab, deren

Folgezustände in S liegen, also $T^{-1}(S) = \{s | \exists s' \in S : R(s, s')\}$. Der Ausdruck $\neg S$ bezeichnet die Menge aller Zustände, die nicht in S liegen.

Der Maximum Delay Algorithmus ist von der Struktur her ähnlich aufgebaut wie der Minimum Delay Algorithmus. Der Algorithmus liefert die Anzahl Zustände auf dem längsten Pfad von einem Zustand aus *start* zu einem Zustand aus *final*. Wenn so ein Pfad nicht existiert, man also von jedem Zustand aus *start* aus die Zustände in *final* immer vermeiden kann, liefert der Algorithmus unendlich zurück.

Zur Berechnung des längsten Pfades wird in der Variable Z Schritt i die Zustände gespeichert, von denen aus in i Schritten *final* nicht zu erreichen ist. Im $i + 1$ Schritt wird Z durch die Vorgänger von Z ersetzt, das sind also die Zustände, aus denen in $i+1$ Schritten *final* nicht zu erreichen ist. Der Algorithmus terminiert, wenn Z geschnitten mit *start* der leeren Menge entspricht, also in *start* keine Zustände mehr sind, von denen aus man $i + 1$ Schritte Zustände aus *final* vermeiden kann. Der andere Fall, in dem der Algorithmus terminiert, ist wenn Z einen Fixpunkt erreicht, dann gibt es Zustände aus *start*, von denen aus *final* immer vermieden werden kann. In diesem Fall liefert der Algorithmus unendlich zurück.

Der Algorithmus terminiert immer. Angenommen, die Bedingung $Z' \cap$ *start* $\neq \emptyset$ wird nie verletzt. Angenommen, ein Zustand ist in der i-ten Iteration in Z enthalten, dann ist er auch in der $(i - 1)$-ten Iteration enthalten. Am Anfang sind alle Zustände, die nicht in *final* enthalten sind, in Z enthalten und bei jedem Schleifendurchlauf werden nur Zustände entfernt, nie neue hinzugenommen. Da die Anzahl der Zustände der Kripke Struktur endlich sind, und, nach Annahme, immer Zustände aus *start* in Z enthalten sind, gibt es ein k, so dass nach k Schritten ein Fixpunkt erreicht ist und die Schleife abbricht. Bildlich gesprochen erreicht jeder Pfad, der in $i + 1$ Schritten die Zustände aus *final* nicht erreicht, die Zustände aus *final* auch nicht in i Schritten. Angenommen, die Bedingung $Z' \neq Z$ wird nie verletzt. Da, wie oben hergeleitet, Z endlich ist und mit jeder Iteration kleiner wird, gibt es ein k, so dass nach k Iterationen $Z' \cap$ *start* $= \emptyset$ ist, womit der Algorithmus auch terminiert. Trivial ist klar, dass es immer ein k gibt, so dass nach k Schritten eine der beiden Abbruchbedingungen erfüllt ist, der Algorithmus terminiert also immer.

Im Fall des Minimum Delay Algorithmus werden die berechneten Pfad Schritt für Schritt verlängert, während bei dem Maximum Delay Algorithmus die Anfangspunkte der Pfade in jedem Schritt neu gewählt werden.

4.6 Condition Counting Algorithmus

In vielen Situation ist man nicht nur an der Anzahl der Transitionen zwischen zwei Ereignissen interessiert, sondern allgemeiner, wie oft ein bestimmtes Ereignis zwischen zwei anderen Ereignissen auftritt. Man möchte also die minimale und maximale Anzahl von Zuständen zwischen zwei Ereignissen ermitteln, die eine bestimmte Bedingung erfüllen.

Der Minimum Condition Counting Algorithmus, der in Abbildung 3 definiert ist, arbeitet vom Prinzip her genau so wie der Minimum Delay Algorithmus. Es

```
procedure min-cond (start, final, cond)
    Z := ∅;
    Z' :=start ∩ ¬ cond;
    while (Z ≠ Z') do
        while ((Z ≠ Z') ∧ (Z'∩ final = ∅)) do
            Z := Z';
            Z' := (T(Z') ∩ ¬ cond) ∪ Z';
        end while;
        if (Z'∩ final ≠ ∅) then
            return i;
        end if;
        Z' := (T(Z') ∩ cond) ∪ Z';
        if (i = 0) then
            Z' := Z' ∪ (start ∩ cond);
        end if;
        i := i + 1;
    end while;
    return ∞;
end procedure
```

Abb. 3. Der *Minimum Condition Counting* Algorithmus

werden aber nicht alle mögliche Transitionen sofort durchgeführt, sondern immer nur die, die Zustände aus *cond* vermeiden. Erst wenn keine anderen Transitionen möglich sind, also ein Fixpunkt erreicht wurde, werden solche Transitionen für einen Schritt zugelassen und der Zähler i erhöht. Dadurch werden für den i-ten Durchlauf der äußeren Schleife alle Pfade in Erwägung gezogen, die maximal i Zustände enthält aus *cond*.

Genauso wie der Minimum Condition Counting Algorithmus ist auch der Maximum Condition Counting Algorithmus dem Maximum Delay Algorithmus sehr ähnlich.

Der Maximum Condition Counting Algorithmus, wie er in Abbildung 4 definiert ist, arbeitet ebenfalls von *final* aus rückwärts. Dabei ist zu beachten, dass alle betrachteten Pfade immer gleich viele Zustände aus *cond* enthalten. Es werden also alle Pfade solange es geht um Zustände, die nicht in *cond* liegen erweitert. Erst, wenn ein Fixpunkt erreicht ist, werden alle Pfade um einen Zustand aus *cond* erweitert. Dass der Algorithmus terminiert kann parallel zu dem Beweis für den Maximum Delay Algorithmus hergeleitet werden.

Mit Hilfe der Condition Counting Algorithmen kann zum Beispiel festgestellt werden, wie häufig eine Prioritäten-Inversion vorkommen kann, d.h. ein Prozess mit höherer Priorität muss warten, da noch ein Prozess niedriger Inversion den Rechner blockiert.

```
procedure max-cond(start, final, cond)
    i := 1;
    Z := ∅;
    Z' := cond;
    Z'' := ∅;
    while (Z' ≠ Z'') do
        Z'' := Z';
        while (Z ≠ Z') do
            Z := Z';
            Z' := ((T^-1(Z') ∩ ¬ final) ∩¬ cond) ∪ Z';
        end while;
        if (Z'∩ start= ∅) then
            return i;
        end if;
        Z' := (T^-1(Z') ∩ ¬ final)∩ cond;
        i := i + 1;
    end while;
    return ∞;
end procedure
```

Abb. 4. Der *Maximum Condition Counting* Algorithmus

4.7 Implementationen

Die hier beschriebene Erweiterung RT-CTL ist in dem Model Checker *VERUS* implementiert. Modelle können in einer an C angelehnten Sprache definiert werden. Auf diesen Modellen können CTL und RT-CTL Formeln verifiziert werden. Außerdem können die Minimum und Maximum Delay bzw. Condition Counting Algorithmen ausgeführt werden.

Ein Anwendungsbeispiel mit VERUS wird im nächsten Kapitel gegeben.

Ähnlich wie RT-CTL definiert wurde, kann auch RT-LTL definiert werden. Das Softwarepaket *RT-Spin*, eine Erweiterung von Spin enthält diese Erweiterung von LTL

4.8 Grenzen von Model Checking

Die größte Einschränkung, die gemacht werden muss, um ein Echtzeit System mit den vorgestellten Techniken zu verifizieren, ist die Beschränkung auf diskrete Zeit. Dadurch wird das Anwendungsgebiet von RT-CTL auf synchrone Designs oder Protokolle beschränkt. Dies deckt aber bereits einen großen Teil von Echtzeit System ab, wie zum Beispiel Controller.

Ein weiteres Problem ist, dass es durch die Diskretisierung zu einer Zustandsexplosion kommen kann, und somit das resultierend Modell nicht mehr handhabbar ist. Weiterhin muss die kleinste Zeiteinheit von vornherein bekannt sein. Wird eine beliebig kleine Zeiteinheit für die Diskretisierung gewählt, so sind RT-CTL Formeln auf diesem Modell unentscheidbar.

Auf der anderen Seite können wesentlich größere Modelle mit Hilfe diskreter Zeit verifiziert werden, als es für kontinuierliche Modelle möglich ist. Weiterhin ist die quantitative Information, die mit Methoden für diskrete Echtzeit Systeme gewonnen werden kann, sehr nützlich um die entsprechenden Modelle zu optimieren.

5 Anwendungsbeispiel

Die oben vorgestellten Methoden sollen in diesem Abschnitt an einem kleinen Beispiel zum besseren Verständnis demonstriert werden.

5.1 Traffic Light Controller

Als Beispiel dient ein so genannter Traffic Light Controller. Dieser Controller steuert eine Verkehrsampel, die an einer Kreuzung steht. Aufgabe des Controllers ist es, dafür zu sorgen, dass die beiden sich kreuzenden Straßen nie gleichzeitig grün bekommen, jedoch jede Straße immer wieder grün bekommt.

Anhand dieses Modells soll der Maximum Delay Algorithmus demonstriert werden, um die Frage zu beantworten, wie lange es maximal dauert, bis eine Straße grün bekommt, wenn gerade die andere Straße grün hat.

5.2 TLC Modell in VERUS

Der Traffic Light Controller, wie er in Abbildung 5 dargestellt ist, muss zunächst in die Beschreibungssprache von VERUS übersetzt werden. Die Beschreibungssprache ist an C angelehnt. Funktionen entsprechen dabei parallel laufenden Prozessen, Zeitverhalten muss explizit über `wait()` Aufrufe kodiert werden. Die Bezeichnungen „EW“ und „NS“ stehen dabei für „east-west„ und „north-south“ und geben die Richtung der jeweiligen Straße an.

Die Beschreibung für VERUS ist in Abbildung 6 gegeben.

5.3 Verifikation

Das Programm VERUS erlaubt es sowohl CTL als auch RT-CTL Formeln auf einem gegebenen Modell zu verifizieren. Wir werden in diesem Beispiel überprüfen, dass das Modell die oben formulierten Eigenschaften hat, also nie beide Straßen gleichzeitig grün bekommen, aber jede Straße immer wieder grün bekommt:

$$\mathbf{AG}\ \mathbf{EF}(tl\_ns = 2)$$
$$\mathbf{AG}\ \mathbf{EF}(tl\_ew = 2)$$
$$\mathbf{AG}\ \neg(tl\_ns = 2 \wedge tl\_ew = 2)$$

Außerdem ermitteln wir, wie lange man auf der „EW“ Straße maximal warten muss, bis es grün wird:

$$\mathbf{MAX}(\neg(tl\_ew = 2), tl\_ew = 2)$$

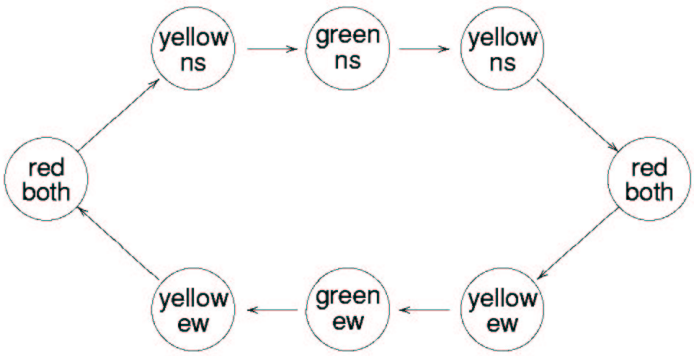


Abb. 5. Modell des Traffic Light Controllers

```
int tl_ns, tl_ew;

main()
{
    /* {0, 1, 2} = {red, yellow, green} */
    tl_ns = 0; tl_ew = 0;
    while (true) {
        tl_ns = 1; wait(1);
        tl_ns = 2; wait(5);
        tl_ns = 1; wait(1);
        tl_ns = 0; wait(2);
        tl_ew = 1; wait(1);
        tl_ew = 2; wait(5);
        tl_ew = 1; wait(1);
        tl_ew = 0; wait(2);
    }
}
```

Abb. 6. Der Traffic Light Controller als VERUS Modell

```
*** VERUS ***

Using BDD library version 1.0.
................................
Compiled function main.

Time to construct the model: User :  1.9e-05 s  System   :    1e-06 s

Spec.  is true    : AG EF(tl_ns == 2);
Spec.  is true    : AG EF(tl_ew == 2);
Spec.  is true    : AG !((tl_ns == 2) && (tl_ew == 2));
Result is      13 : MAX (tl_ns != 2, tl_ns == 2);

Execution information:

Time              - User               :  2.2e-05 s  System   :    2e-06 s
BDD nodes used    - Transition relation:        651  Total    :       7134
Bytes allocated   -                          462004
Boolean variables -                              24
States            - Total              : 1.17965e+06  Reachable:         18
```

Abb. 7. Ausgabe von VERUS

Die Ausgabe von VERUS ist ein Abbildung 7 dargestellt.

Die Ausführung des Maximum Delay Algorithmus hat uns den Wert 13 geliefert. Um zu verstehen, wie dieser Wert zustande gekommen ist, gehen wir im folgenden die einzelnen Schritte des Algorithmus durch.

Der Algorithmus wird mit den zwei Zustandsmengen {*yellow ew'*, *red both*, *yellow ns*, *green ns*, *yellow ns'*, *red both'*, *yellow ew*} als *start* und {*green ew*} als *final* aufgerufen. Die Menge *start* wird nun solange durch die Menge von Zuständen ersetzt, deren Folgezuständen in *start* liegen, die aber selber nicht in *final* liegen, bis entweder ein Fixpunkt erreicht wird, oder kein Zustand aus *start* mehr in der neuen Menge ist. Zunächst wird also der Zustand *yellow ew* entfernt, da er keine Folgezustände hat, die in *start* liegen. Im nächsten Schritt wird *red both'* entfernt, da nun der Folgezustand *yellow ew* nicht mehr in der Zustandsmenge ist. Dies geht solange weiter, bis nur noch *yellow ew* in der Zustandsmenge ist. Im darauf folgenden Schritt wird auch dieser Zustand entfernt, also $Z = \{yellow\ ew\}$ und $Z' = \emptyset$. Dann gilt zwar $Z \neq Z'$ aber $Z \cap start \neq \emptyset$ ist verletzt, der Algorithmus terminiert also, und liefert die Anzahl der Zustände in *start* zurück, da pro Iteration jeweils ein Zustand entfernt wurde. Da manche Zustände im VERUS Modell mehr als einen Takt dauern, ist die Summe also 13.

6 Zusammenfassung

In diesem Kapitel wurde der Begriff diskretes Echtzeit System definiert. Mit dieser Definition konnte die Verifikationsaufgabe als Überprüfung der Durchführbarkeit festgelegt werden.

Zunächst wurde die herkömmliche und etablierte Methode zur Überprüfung der Durchführbarkeit mit Hilfe des Rate Monotonic Scheduler eingeführt und deren Korrektheit gezeigt. Es stellte sich heraus, dass der Rate Monotonic Scheduler große Einschränkungen macht, welche Designs überprüft werden können, aber trotz diese Nachteile einfach handzuhaben ist, und deshalb häufig eingesetzt wird.

Anschließend wurde kurz auf alternative Methoden mit Hilfe der Erreichbarkeitsanalyse eingegangen, die viele der Einschränkungen des Rate Monotonic Scheduler nicht haben, dafür aber schwerer handzuhaben sind und auch keine quantitative Informationen über das zu untersuchende Design liefern.

Als Möglichkeit, all diese Beschränkungen zu umgehen, wurde die RT-CTL Logik eingeführt. Sie erlaubt es, eine große Anzahl von diskreten Echtzeit Systeme zu verifizieren und liefert quantitative Informationen über das Modell. Diese Informationen werden mit den Minimum und Maximum Delay Algorithmen bzw. den Minimum und Maximum Condition Counting Algorithmen gewonnen.

Obwohl die vorgestellten Methoden viele Echtzeit System erfassen, können nicht alle Echtzeit Systeme unter Verwendung diskreter Zeit betrachtet werden. Es können nur solche Echtzeit System verifiziert werden, die mit einer a priori bekannten kleinsten Zeiteinheit diskretisiert werden können, also zum Beispiel Controller oder Protokolle. Echtzeit Systeme, die nicht diskretisierbar sind, oder deren Zustandsraum bei der Diskretisierung explodieren würde, müssen mit Methoden für kontinuierliche Echtzeit Systeme verifiziert werden.

Die vorgestellte Logik RT-CTL wurde zum Schluss anhand eines Anwendungsbeispiels mit der Software VERUS demonstriert.

Literatur

1. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, 1999.
2. Thomas Kropf. *Introduction to Formal Hardware Verification.* Springer, 1999.
3. M. G. Harbour, M. H. Klein, and J. P. Lehoczky. *Timing analysis for fixed-priority scheduling of hard real-time systems.* IEEE Transactions on Software Engineering, 20(1): 13-28.
4. Stewart, David and Michael Barr. "Rate Monotonic Scheduling" *Embedded Systems Programming.* March 2002, pp. 79-80.
5. Stavros Tripakis and Costas Courcoubetis. *Extending Promela and Spin for Real Time* Proceedings of TACAS'96, LNCS 1044.
6. S. Campos and E. Clarke. *Real-time symbolic model checking for discrete time models.* In AMAST Series in Computing: Theories and Experiences for Real-Time System Development. World Scientific Publishing Company, 1995.

7. S. Campos, E. Clarke, W. Marrero and M. Minea. *Verus: a tool for quantitative analysis of finite-state real-time systems.* In Workshop on Languages, Compilers and Tools for Real-Time Systems, 1995.

Kontinuierliche Echtzeit

Markus Knapp

Institut für Informatik
Albert-Ludwigs-Universtität Freiburg
knapp@informatik.uni-freiburg.de

Zusammenfassung. Durch die Einführung asynchroner Systeme und die Erweiterung der Übergangssysteme um zeitliche Aspekte wie die Verweildauer in Zuständen und Verzögerungs- und Ausführundszeiten für Aktionen wird in dieser Seminararbeit das Arbeiten mit kontinuierlichen Echtzeitsystemen ermöglicht.

Einleitung

In den vorherigen Arbeiten war der zugrundeliegende Zeitbegriff ausschließlich diskret, daß heißt, es waren ausschließlich synchrone Systeme mit einem gemeinsamen Takt. Die möglichen Uhrenwerte waren nichtnegative, ganze Zahlen und Ereignisse waren nur zu ganzzahligen Uhrenwerten möglich.

Um auch über asynchrone Systeme mit kontinuierlicher Zeit Aussagen treffen zu können, wird das bisher bekannte Übergangssystem um zeitliche Aspekte wie die Verweildauer in den Zuständen und der Verzögerungs- und Ausführungszeit für Aktionen erweitert. Dies geschieht durch die Einführung des Echtzeitautomaten, einer endlichen Beschreibung eines kontinuierlichen Prozesses.

Die Seminararbeit ist in vier Abschnitte eingeteilt.

Im ersten Abschnitt wird der Echtzeitautomat beschrieben, anhand eines Beispiels vorgeführt und anschließend definiert. Daraufhin wird beschrieben, wie man aus einem aus Teilsystemen bestehendes System den kompletten Echtzeitautomaten des ganzen Systems macht und abschließend noch ein größeres System ansatzweise modelliert.

Im zweiten Abschnitt befassen wir uns damit, wie man aus dem in Abschnitt eins vorgestellten Übergangsgraphen ein endliches Modell machen kann. Dazu werden sogenannte Uhrenregionen eingeführt und der Regionengraph vorgestellt, anhand dessen man dann gewisse Eigenschaften eines Systems mittels Model Checking überprüfen kann.

Abschnitt drei befasst sich mit Uhrenzonen und einer Darstellungsform für diese: den Difference Bound Matrizen, mit denen wiederum Eigenschaften von Systemen mittels Model Checking berprft werden knnen.

Im vierten Abschnitt wird schließlich noch kurz der Cottbus Timed Automata vorgestellt, der den Echtzeitautomaten um das Prinzip der Modularität erweitert.

1 Echtzeitautomat

1.1 Einführung

Ein Echtzeitautomat (englisch: timed automata) [5, 6] ist eine endliche Beschreibung eines zeitkontinuierlichen Prozesses. Die dabei verwendeten Zeitschranken werden mit Hilfe von Uhren beschrieben. Die Menge der Uhren wird mit X bezeichnet. Diese Uhren können als spezielle Variablen mit Wertebereich $\mathbb{R}^+$ angesehen werden. Die einzigen zulässigen Zugriffe auf Uhren sind zum einen der lesende Zugriff, also die Abfrage des aktuellen Wertes einer Uhr oder der Vergleich von Werten von verschiedenen Uhren und zum anderen der schreibende Zugriff, der nur in Form von Zurücksetzung des Wertes der Uhr auf 0 mit $Reset(x)$ geschehen darf. Die Werte der Uhren ändern sich dynamisch mit der Zeit, das heißt, dass wenn in δ Zeiteinheiten kein $Reset(x)$ ausgeführt wurde, sich die Werte aller Uhren um δ erhöhen. Der Zeitbegriff ist global, das heißt, das allen Uhren derselbe Zeittakt zugrunde liegt. Es wird festgelegt, dass Übergänge im Echtzeitautomaten augenblicklich ausgeführt werden, also keine Zeit dabei verstreicht. In einem Zustand des Echtzeitautomaten läuft die Zeit weiter.

Sei nun X eine Menge von Uhren. $C(X)$ bezeichnet die Menge aller Uhrenbedingungen. Alle Ungleichungen der Form $x \prec c$ oder $c \prec x$ mit $\prec$ ist entweder $<$ oder $\leq$ und c ist eine nichtnegative rationale Zahl, sind Uhrenbedingungen. Wenn φ_1 und φ_2 Uhrenbedingungen sind, so auch $\varphi_1 \wedge \varphi_2$. Jedem Übergang werden Uhrenbedingungen zugeordnet. Der Übergang kann nur vollzogen werden, wenn die dazugehörenden Uhrenbedingungen zutreffen. Außerdem sind jedem Zustand Uhrenbedingungen zugeordnet. Diese werden Invarianten genannt und bestimmen, wie lange ein System in diesem Zustand verweilen darf.

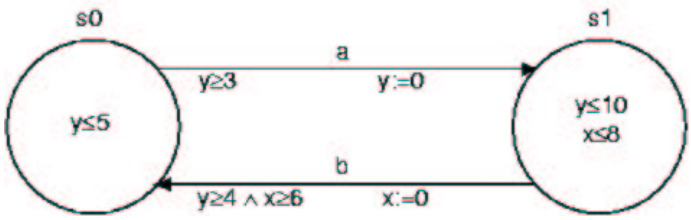


Abb. 1. Beispiel für einen Echtzeitautomaten

Beispiel 1. In Abbildung 1 ist ein Echzeitautomat dargestellt. Der Automat besteht aus zwei Zuständen $s0$ und $s1$, der Startzustand ist $s0$. Es gibt zwei Uhren x und y und zwei Übergänge a von $s0$ nach $s1$ und b von $s1$ nach $s0$. Der Automat kann im Zustand $s0$ bleiben, solange der Wert der Uhr y kleiner oder gleich 5 ist. Wird der Wert der Uhr y gleich oder größer als 3, so kann der Übergang a nach $s1$ ausgeführt werden. Dabei wird ein $Reset(y)$ ausgeführt, der Wert der Uhr y also auf 0 gesetzt. Im Zustand $s1$ kann der Automat bleiben, solange y

kleiner oder gleich 10 und x ist kleiner oder gleich 8. Wenn x mindestens 6 und y mindestens 4 kann der Automat den Übergang b ausführen wobei dabei ein $Reset(x)$ ausgeführt wird, also der Wert der Uhr y auf 0 gesetzt wird.

1.2 Definition

Definition 1. *Ein Echtzeitautomat ist ein Tupel*

$$A = (\Sigma, S, S_0, X, I, T)$$

mit

- Σ *ist ein endliches Alphabet.*
- S *ist eine endliche Menge von Zuständen.*
- $S_0 \subseteq S$ *ist die Menge der Anfangszustände.*
- X *ist die Menge aller Uhren*
- $I : S \to C(X)$ *ist eine Abbildung, die jedem Zustand eine Uhrenbedingung zuordnet.*
- $T \subseteq S \times \Sigma \times C(X) \times 2^x \times S$ *ist die Übergangsmenge.*

Definition 2. *Ein Übergang ist ein Tupel*

$$\langle s, a, \varphi, \lambda, s' \rangle$$

mit

- s *ist der Zustand am Anfang des Übergangs.*
- s' *ist der Zustand am Ende des Übergangs.*
- a *ist die Kennzeichnung des Übergangs.*
- φ *ist die Uhrenbedingung.*
- λ *ist die Menge von Uhren, die auf* 0 *zurückgesetzt werden, wenn der Übergang ausgeführt wird.*

Es wird vorausgesetzt, dass die Zeit bis ins Unendliche läuft, das heißt, dass in jedem Zustand der obere Grenzwert einer Uhr entweder unendlich ist oder kleiner als der maximale Wert der zu dieser Uhr gehörenden Uhrenbedingungen in diesem Zustand oder den zu diesem Zustand gehörenden ausgehenden Übergängen.

Definition 3. *Sei $A = (\Sigma, S, S_0, X, I, T)$ ein Echtzeitautomat. Ein Übergangsgraph ist ein Tupel*

$$T(A) = (\Sigma, Q, Q_0, R)$$

mit

- Σ *ist ein Alphabet.*
- Q *ist die Menge der Konfigurationen (s, v) mit*
  - $s \in S$ *ist ein Zustand.*
  - $v : X \to \mathbb{R}^+$ *ist eine Uhrenbelegung*

- Q_0 *ist die Menge der Anfangskonfigurationen* $\{(s_0, v)|s_0 \in S_0 \wedge \forall x \in X[v(x) = 0]\}$
- R *ist die Übergangsrelation*

Da die Anzahl der Uhrenbelegungen $v : X \to \mathbb{R}^+$ unendlich ist, ist auch das resultierende Übergangssystem im allgemeinen unendlich. Ein Echtzeitautomat ist also eine endliche Beschreibung eines unendlichen Übergangssystems.

Um ein Übergangsgraph beschreiben zu können, bedarf es weiterer Notationen:

- $v[\lambda = 0]$ ist die Uhrenbelegung, die die Uhren in λ auf 0 setzt.
- $v \pm d$ mit $d \in \mathbb{R}$ ordnet jedem $x \in X$ den Wert $v(x) \pm d$ zu.

Ein Übergangsgraph hat zwei Typen von Übergängen:

- Verzögerungsübergänge: Dies ist die Verweildauer des Systems in einem Zustand $(s, v) \to^d (s, v + d)$, wobei $d \in \mathbb{R}^+$ und für alle $0 \leq e \leq d$ die Invarianzbedingung $I(s)$ für $v + e$ erfüllt ist.
- Aktionsübergänge: Ausführung eines Übergangs aus T $(s, v) \to^a (s', v')$, wobei $a \in \Sigma$ und es gibt einen Übergang $\langle s, a, \varphi, \lambda, s' \rangle$, so dass v die Uhrenbedingung φ erfüllt und $v' = v[\lambda = 0]$.

Die Übergangsrelation R von $T(A)$ erhält man durch das Kombinieren der Verzögerungsübergänge und der Aktionsübergänge. Man schreibt $(s, v)R(s, v)$ oder $(s, v) \Rightarrow^a (s, v)$, falls ein s und ein v existiert, so dass für ein $d \in \mathbb{R}(s, v) \to^d (s, v) \to^a (s, v)$ gilt.

1.3 Parallelkomposition

Um auch komplexe Systeme modellieren zu können, bedarf es der Aufteilung des Systems in Teilprozesse. Aus diesen Teilprozessen werden dann Echtzeitautomaten modelliert und diese dann geeignet zusammengesetzt. Im folgenden wird nun beschrieben, wie das Zusammensetzen funktioniert.

Die Komposition für zwei Echtzeitautomaten $A_1 = (\Sigma_1, S_1, S_0^1, X_1, I_1, T_1)$ und $A_2 = (\Sigma_2, S_2, S_0^2, X_2, I_2, T_2)$ mit disjunkten Uhrenmengen X_1 und X_2 ist:

$$A_1 \| A_2 = (\Sigma_1 \cup \Sigma_2, S_1 \times S_2, S_0^1 \times S_0^2, X_1 \cup X_2, I, T)$$

wobei $I(s_1, s_2) = I_1(s_1) \wedge I_2(s_2)$.

Die Übergangsmenge T ergibt sich durch folgende Regeln:

- Für $a \in \Sigma_1 \cup \Sigma_2$, falls $\langle s_1, a, \varphi_1, \lambda_1, s_1' \rangle \in T_1$ und $\langle s_2, a, \varphi_2, \lambda_2, s_2' \rangle \in T_2$, dann enthält T den Übergang $\langle (s_1, s_2), a, \varphi_1 \wedge \varphi_2, \lambda_1 \cup \lambda_2, (s_1', s_2') \rangle$
- Für $a \in \Sigma_1 - \Sigma_2$, falls $\langle s, a, \varphi, \lambda, s \rangle \in T1$ und $t \in S_2$, dann enthält T den Übergang $\langle (s, t), a, \varphi, \lambda, (s, t) \rangle$.
- Für $a \in \Sigma_2 \cup \Sigma_1$, falls $\langle s, a, \varphi, \lambda, s \rangle \in T2$ und $t \in S_1$, dann enthält T den Übergang $\langle (t, s), a, \varphi, \lambda, (t, s) \rangle$.

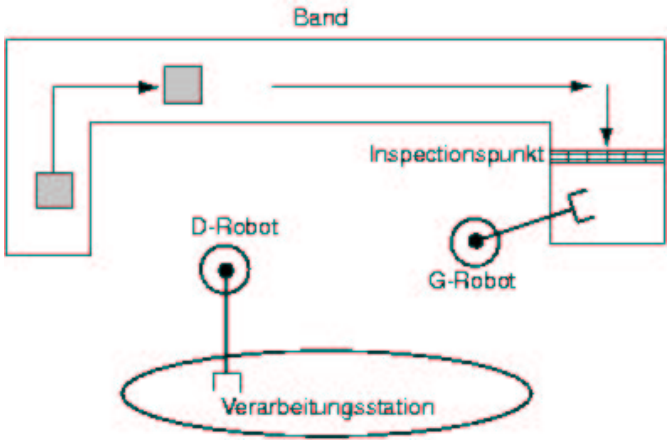


Abb. 2. Die Produktionsanlage

1.4 Beispiel einer Modellierung

Um zu zeigen, wie Echtzeitautomaten benutzt werden können, um Echtzeitsysteme zu modellieren, wird eine einfache Produktionsanlage (vorgestellt in [7]) genommen. Die Anlage besteht aus einem Förderband, das von links nach rechts läuft, einer Verarbeitungsstation und zwei Robotern, die Boxen zwischen der Station und dem Förderband hin und her transportieren (siehe Abbildung 2). Der erste Roboter (genannt D-Robot) nimmt eine Box von der Station und setzt sie links auf das Band. Der zweite Roboter (genannt G-Robot) nimmt eine Box rechts vom Band und transportiert sie zur Station.

Der Echtzeitautomat für den D-Robot ist in Abbildung 3 dargestellt. Der Roboter wartet bei der Station im Zustand D-Wait bis eine Box bereit steht (Übergang s-ready). Dann nimmt er die Box (D-Pick), dreht sich rechts (D-Turn-R) und setzt die Box auf das Förderband (D-Put). Er dreht sich links (D-Turn-L) und kehrt in den Anfangszustand zurück. Eine Box zu nehmen oder abzustellen dauert eine bis zwei Sekunden, das Links- oder Rechtsdrehen dauert zwischen fünf und sechs Sekunden.

Der Echtzeitautomat die den G-Robot ist in Abbildung 4 dargestellt. Dieser Roboter wartet am Inspektionspunkt (G-Inspect) am rechten Ende des Förderbandes bis eine Box den Inspektionspunkt passiert. Der Roboter muss diese Box nehmen (G-Pick) bevor sie am Ende des bandes herunterfällt. Dann dreht er sich nach rechts (G-Turn-R), wartet an der Verarbeitungsstation bis diese mit der Verarbeitung der vorherigen Box fertig ist (G-Wait) und legt dann die Box in die Station (G-Put). Zum Schluss dreht er sich wieder nach links zum Inspektionspunkt (G-Turn-L). Es dauert zwischen drei und acht Sekunden, eine Box aufzunehmen und zwischen sechs und zehn Sekunden, um nach rechts zu drehen. Es dauert zwischen einer und zwei Sekunden, die Box in die Station zu legen und zwischen acht und zehn Sekunden, zum Inspektionspunkt zurückzukehren.

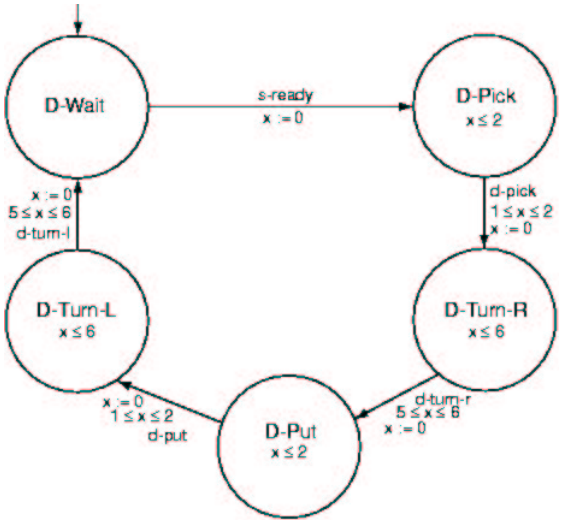


Abb. 3. Echtzeitautomat für den D-Robot

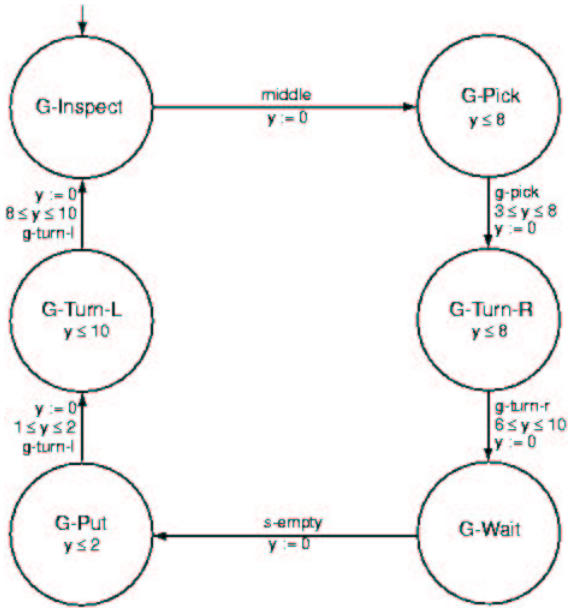


Abb. 4. Echtzeitautomat für den G-Robot

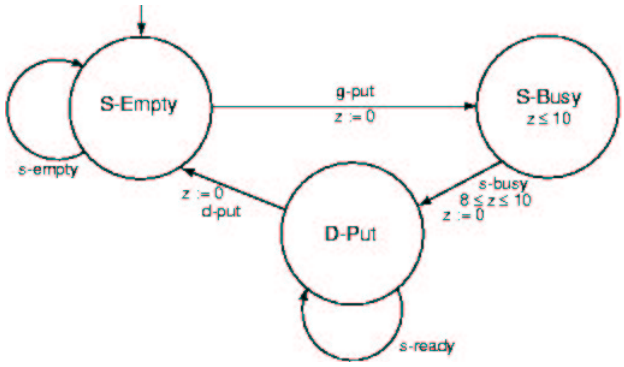


Abb. 5. Echtzeitautomat für die Verarbeitungsstation

Der Echtzeitautomat für die Verarbeitungsstation ist in Abbildung 5 dargestellt. Die Station ist am Anfang leer (S-Empty). Wenn eine Box ankommt, dauert es zwischen acht und zehn Sekunden, um sie zu verarbeiten. Die Box ist dann bereit, vom G-Robot abgeholt zu werden.

Der Echtzeitautomat für eine Box ist in Abbildung 6 dargestellt. Am Anfang liegt die Box auf dem Förderband und bewegt sich von links nach rechts (B-Mov) zum Inspektionspunkt. Wenn sie den Inspektionspunkt passiert hat (B-Inspect), fällt sie vom Band (B-Fall), außer der G-Robot hat sie aufgenommen (B-on-G). Im zweiten Fall wird die Box in die Station gelegt (B-in-S), wird vom D-Robot aufgenommen (B-on-D) und zurück auf das Band gelegt. Es dauert zwischen 133 und 134 Sekunden, bis die Box über das Band gelaufen ist und den Inspektionspunkt erreicht. Die Box wird vom Band fallen, wenn sie nicht innerhalb von einundzwandzig bis zweiundzwanzig Sekunden nachdem sie den Inspektionspunkt passiert hat, aufgenommen wurde.

Der Echtzeitautomat für das System ist die Parallelkomposition der vier einzelnen oben beschriebenen Echtzeitautomaten.

2 Uhrenregionen

2.1 Uhrenregionen

Um ein endliches Modell des Übergangsgraphen $T(A)$ zu bekommen, definieren wir Uhrenregionen [4, 5] als Mengen von Uhrenbelegungen. Der Vorkommateil eines Uhrenwertes bestimmt dabei, ob eine Uhrenbedingung erfüllt ist und der Grad des Nachkommateils bestimmt, welche Uhr ihren Vorkommateil als nächstes ändern wird, da die Uhrenbedingungen nur natürliche Zahlen enthalten und sich alle Uhren um denselben Wert erhöhen. Der Uhrenwert kann beliebig groß werden, aber wird die Uhr nie mit einer Konstante größer als c verglichen, wird der Wert der Uhr kein Effekt bei der Berechnung von A haben, wenn er c überschreitet. Wenn nun 2 Konfigurationen, die zu demselben Zustand eines Echtzeitautomaten gehören, in den Vorkommateilen und im Grad

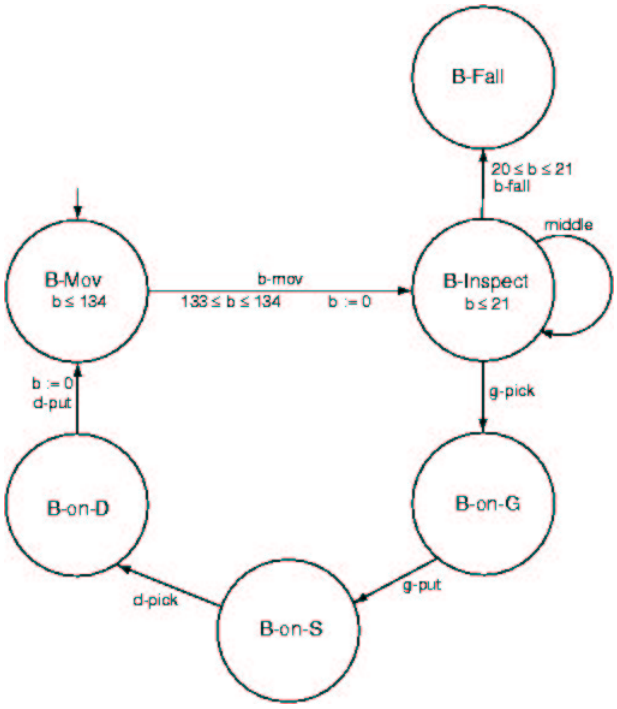


Abb. 6. Echtzeitautomat für eine Box

der Nachkommateile aller Uhren übereinstimmen, dann verhalten sich die beiden Konfigurationen aus zeitlicher Sicht gleich. Um dieses Verhalten des Echtzeitautomaten zu formalisieren, sei für jede Uhr $x \in X$, c_x die größte Konstante mit der x verglichen wird. Für $t \in \mathbb{R}^+$ ist $fr(t)$ der Nachkommateil von t und $\lfloor t \rfloor$ ist der Vorkommateil, so dass $t = \lfloor t \rfloor + fr(t)$. Wir definieren eine Äquivalenzrelation auf der Menge der möglichen Uhrenbelegungen: Seien v und v' zwei Uhrenbelegungen. Dann gilt $v \cong v'$ genau dann wenn die folgenden drei Bedingungen erfüllt sind:

1. Für alle $x \in X$ ist entweder $v(x) \geq c_x$ und $v'(x) \geq c_x$ oder $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$.
2. Für alle $x, y \in X$, so dass $v(x) \leq c_x$ und $v(y) \leq c_y$, $fr(v(x)) \leq fr(v(y))$ gdw. $fr(v'(x)) \leq fr(v'(y))$.
3. Für alle $x \in X$, so dass $v(x) \leq c_x$, $fr(v(x)) = 0$ gdw. $fr(v'(x)) = 0$.

Die Äqivalenzklassen von $\cong$ nennt man Regionen. $[v]$ bedeutet die Region, die die Uhrenbelegung v enthält:

– für jede Uhr $x \in X$

$$\{x = c \,|\, c = 0, ..., c_x\} \cup \{c - 1 < x < c \,|\, c = 1, ..., c_x\} \cup \{x > c_x\}$$

.

– für jedes Paar $x, y \in X$ und ganzzahligen p und q:
  – Die Gitterpunkte
  $$(q, p).$$
  – Die Gitterlinien
  $$\{(q, y) : p \leq y \leq p + 1\} \textit{ und } \{(x, p) : q \leq x \leq q + 1\}.$$
  – Das Innere der Quadrate
  $$\{(x, y) : q \leq x \leq q + 1; p \leq y \leq p + 1\}.$$

  bzw. deren weiterer Unterteilung:

  $$\{(x, y) : q \leq x \leq q + 1; p \leq y \leq p + 1, x - y < q - p\}.$$
  $$\{(x, y) : q \leq x \leq q + 1; p \leq y \leq p + 1, x - y = q - p\}.$$
  $$\{(x, y) : q \leq x \leq q + 1; p \leq y \leq p + 1, x - y > q - p\}.$$

Beispiel 2. Die Abbildung 7 zeigt einen Echtzeitautomaten mit zwei Uhren x, y mit $c_x = 2$ und $c_y = 1$. Insgesamt gibt es dann 28 Uhrenregionen, die sich wie folgt aufteilen:

– 6 Gitterpunkte (in der Abbildung mit 1 bis 6 nummeriert, beispielsweise die Nummer 2: $[(1, 0)]$).
– 14 Gitterliniern (in der Abbildung mit 7 bis 20 nummeriert, beispielsweise die Nummer 20: $[1 < x < 2 \wedge y = x - 1]$).

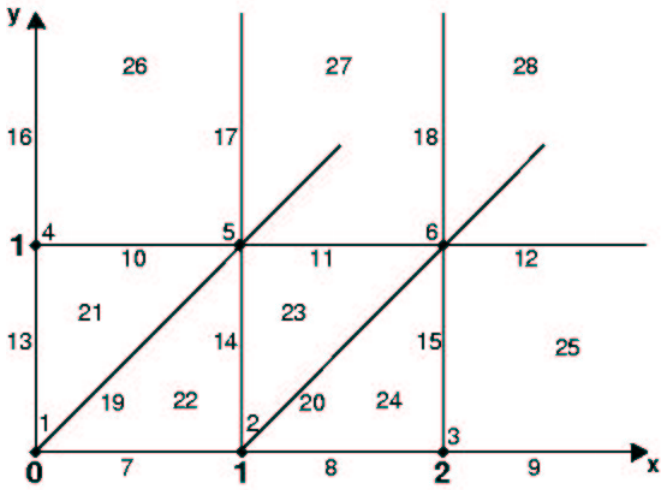


Abb. 7. Beispiel für Uhrenregionen

– 8 offene Regionen (in der Abbildung mit 21 bis 28 nummeriert, beispielsweise die Nummer 24: $[1 < x < 2 \wedge 0 < y < x - 1]$).

Lemma 1. *Die Anzahl der Äquivalenzklassen von* $\cong$ *ist beschränkt durch*

$$|X|! \cdot 2^{|X|} \cdot \prod_{x \in X} (2c_x + 2).$$

Beweis. Eine Äquivalenzklasse $[v]$ von $\cong$ kann als Tripel von Arrays $\langle \alpha, \beta, \gamma \rangle$ beschrieben werden: Für jede Uhr $x \in X$ sagt das Array α, welches der Intervalle

$$\{\, [0,0], (0,1), [1,1], ..., (c_x - 1, c_x), [c_x, c_x], (c_x, \infty) \,\}$$

den Wert $v(x)$ enthält. Das Array α repräsentiert also die Uhrenbelegung v gdw. für jede Uhr $x \in X$, $v(x) \in \alpha(x)$. Die Anzahl der Möglichkeiten, α zu wählen, ist

$$\prod_{x \in X} (2c_x + 2).$$

Sei nun X_α die Menge der Uhren x, so dass $\alpha(x)$ die Form $(i, i+1)$ für $i \leq c_x$. X_α ist also die Menge der Uhren mit einem Nachkommateil, der nicht 0 ist. Das Array $\beta : X_\alpha \to \{1, ..., |X_\alpha|\}$ ist eine Permutation von X_α, welches die Anordnung der Nachkommateile bezüglich $\leq$ wiedergibt. Das Array β repräsentiert also eine Uhrenbelegung v gdw. für alle Paare $x, y \in X_\alpha$, wenn $\beta(x) < \beta(y)$ dann $fr(v(x)) \leq fr(v(x))$. Für ein gegebenes α ist die Anzahl der Möglichkeiten, β zu wählen, gleich $|X_\alpha|!$ was durch $|X|!$ beschränkt wird.

Der dritte Teil γ ist ein boolsches Array, welches angibt, welche Uhren in X_α den selben Nachkommateil haben. Für jede Uhr x gibt $\gamma(x)$ an, ob der Nachkommateil von $v(x)$ gleich dem Nachkommateil des Vorgängers im Array β ist. Das Array γ repräsentiert also eine Uhrenbelegung v gdw. ür alle $x \in X_\alpha$ gilt $\gamma(x) = 0$ gdw. es gibt eine Uhr $y \in X_\alpha$ mit $\beta(y) = \beta(x) + 1$ und $fr(v(x)) = fr(v(y))$. Die Anzahl der Möglichkeiten, γ zu wählen, wird durch die Anzahl der boolschen Arrays über X_α beschränkt, die wiederum beschränkt ist durch $2^{|X|}$.

α repräsentiert also den Vorkommateil der Uhrenbelegung und β zusammen mit γ repräsentiert die Ordnung des Nachkommateils. Die Menge der Tripel $\langle \alpha, \beta, \gamma \rangle$ sind also Aquivalenzklassen von $\cong$ und jede Aquivalenzklasse wird durch ein Tripel repräsentiert. Die Schranke aus dem Lemma ist also das Produkt der Schranken von α, β und γ. □

Eigenschaften von $\cong$:

Lemma 2. *Seien v_1 and v_2 zwei Uhrenbelegungen, φ eine Uhrenbedingung und $\lambda \subseteq X$ eine Menge von Uhren.*

- *Wenn $v_1 \cong v_2$ und t eine nichtnegative ganze Zahl ist, dann gilt $v_1+t \cong v_2+t$.*
- *Wenn $v_1 \cong v_2$, dann $\forall t_1 \in \mathbb{R}^+ \exists t_2 \in \mathbb{R}^+ [v_1 + t_1 \cong v_2 + t_2]$.*
- *Wenn $v_1 \cong v_2$, dann erfüllt v_1 φ gdw. v_2 φ erfüllt.*
- *Wenn $v_1 \cong v_2$, dann ist $v_1[\lambda := 0] \cong v_2[\lambda := 0]$.*

Der Beweis dieses Lemmas ist recht einfach und wird deshalb weggelassen. Der Beweis für den 2. Punkt kann in [1] nachgelesen werden.

Die Äquivalenzrelation über die Uhrenbelegungen kann auch über den Konfigurationsraum von $T(A)$ erweitert werden. Die äquivalenten Konfigurationen haben dann identische Zustände und äquivalente Uhrenbelegungen:

$$(s, v) \cong (s', v')$$

gdw.

$$s = s' \qquad \text{und} \qquad v \cong v'.$$

Lemma 3. *Wenn $v_1 \cong v_2$ und $(s, v_1) \Rightarrow^a (s', v_1')$, dann gibt es eine Uhrenbelegung v_2', so dass $v_1' \cong v_2$ und $(s, v_2) \Rightarrow^a (s', v_2')$.*

Beweis. Angenommen $v_1 \cong v_2$ und $(s, v_1) \Rightarrow^a (s', v_1')$. Die Transition $(s, a, \varphi, \lambda, s')$ entspricht den zwei Transitionen in $T(A)$:

- der Verzögerungsübergang $(s, v) \rightarrow^d (s, v + d_1)$ für ein $d_1 \geq 0$
- der Aktionsübergang $(s, v_1 + d) \rightarrow^a (s', v_1')$, so dass v_1 φ erfüllt und $v_1' = (v_1 + d_1)[\lambda := 0]$.

Da $v_1 \cong v_2$ und v_1 $I(s)$ erfüllt, erfüllt v_2 auch $I(s)$. Zudem gibt es ein $d_2 \geq 0$ und es gilt $v_1 + d_1 \cong v_2 + d_2$, genau so erfüllen $v_1 + d_1$ und $v_2 + d_2$ auch die Invarianzbedingung $I(s)$. Da $I(s)$ konvex ist und von v_2 und $v_2 + d$ erfüllt wird, ist $I(s) = true$ für $v_2 + e$ für alle $0 \leq e \leq d_2$. Infolgedessen ist der Verzögerungsübergang $(s, v_1 + d) \rightarrow^d (s, v_1')$ gültig.

Wegen $v_1 + d_1 \cong v_2 + d_2$ erfüllen $v_1 + d_1$ und $v_2 + d_2$ die Uhrenbedingung φ. Somit ist die Transition $(s, a, \varphi, \lambda, s')$ aus $v_2 + d_2$ ausfüllbar. Sei $v_2' = (v_2 + d_2)[\lambda := 0]$. Dann gilt $v_1' \cong v_2'$. Daher gibt es einen Aktionsübergang $(s, v_2) \rightarrow^a (s'; v_2')$ und durch das Kombinieren der beiden Übergänge bekommt man den Übergang $(s, v_2) \Rightarrow^a (s, v_2')$. □

Lemma 3 zeigt, dass in regionenäquivalenten Zuständen genau dieselben Übergänge ausführbar sind und die Nachfolgezustände auch regionenäquivalent sind.

2.2 Regionengraph

Definition 4. *Der Regionengraph zu einem Echtzeitautomaten ist ein 3-Tupel*

$$R(A) = (S_R, S_0^R, T_R)$$

mit

- S_R *ist die Menge der Zustände* $(s, [v])$ *wobei* $s \in S$ *und* $[v]$ *ist eine Uhrenregion.*
- S_0^R *ist die Menge der Anfangszustände* $(s_0, [v])$ *wobei* $s_0 \in S_0$ *und* $v(x) = 0$ *für alle* $x \in X$.
- T_R *ist die Übergangsmenge* $((s, [v]), a, (s, [v])) \in T_R$ *gdw.* $(s, w) \rightarrow^a (s, w)$ *für ein* $w \in [v]$ *und* $w \in [v]$.

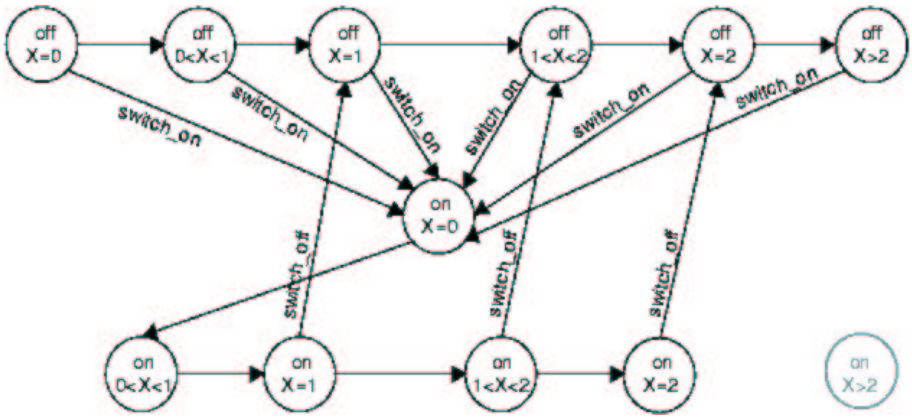


Abb. 8. Beispiel für einen Regionengraphen

Beispiel 3. In Abbildung 8 ist der Regionengraph für einen automatishen Lichtschalter dargestellt mit einer Uhr x, den Zuständen on und off und den Übergängen *switch_on* und *switch_off* dargestellt. Beim Einschalten des Lichtes durch den Benutzer wird die Uhr x auf 0 gesetzt. Der on-Zustand kann frühstens nach einer Minute durch den Benutzer und spätestens nach 2 Minuten automatisch wieder verlassen werden.

Lemma 4. *Der Übergangsgraph $T(A)$ und der Regionengraph $R(A)$ sind bisimulationsäquivalent.*

Beweis. Folgt aus Lemma 3 und ist in [1] ausgeführt.

3 Uhrenzonen und Difference Bound Matrizen

3.1 Uhrenzonen

Eine alternative Darstellung sind Uhrenzonen, die hier noch in Kürze vorgestellt werden. Dies sind Mengen von Uhrenbedingungen der Form

$$x \prec c,\ c \prec x,\ x - y \prec c$$

mit $\prec$ ist entweder $<$ oder $\leq$. Zusätzlich wird noch eine Uhr x_0 eingeführt, die immer 0 ist und Uhren werden jetzt mit $x_0, ..., x_n$ benannt. Damit lässt sich aus einer Uhrenbedingung

$$-c_{0,i} \prec x_i \prec c_{i,0}$$

mit nicht negativen $c_{0,i}$ und $c_{i,0}$ mit der Uhr x_0 die folgende Konjunktion machen:

$$x_0 - x_i \prec c_{0,i} \wedge x_i - x_0 \prec c_{i,0}$$

und dann auch die allgemeine Form

$$x_0 = 0 \wedge \bigwedge_{0 \leq i \neq j \leq n} x_i - x_j \prec c_{i,j}.$$

Folgende Operationen mit Uhrenzonen gibt es:

- Kreuzen: Wenn φ und ψ zwei Uhrenzonen sind, so auch $\varphi \wedge \psi$.
- Uhren Reset: Wenn φ eine Uhrenzone ist und λ ist eine Menge von Uhren, dann ist $\varphi[\lambda := 0]$ auch eine Uhrenzone.
- Verstrichene Zeit: Wenn φ eine Uhrenzone ist, dann ist eine Uhrenbedingung v ein Element der verstrichenen Zeit $\varphi^{\Uparrow}$, wenn v die Formel $\exists t \geq 0[(v-t) \in \varphi]$ oder $\exists t \geq 0[v \in (\varphi - t)]$ erfüllt. Dies ist auch eine Uhrenzone.

Mit diesen drei Operationen auf Uhrenzonen ist es nun möglich, eine endliche Repräsentation des Übergangsgraphen $T(A)$ zu konstruieren. Eine Möglichkeit wäre die in [3] vorgestellten sogenannten Zonen, die hier nicht näher beschrieben werden sollen.

3.2 Difference Bound Matrizen

Eine weitere Möglichkeit der Darstellung von Uhrenzonen sind die in [6] vorgestellten Difference Bound Matrizen. Die Matrix wird mit den in X vorhandenen Uhren inklusive der Uhr x_0 indiziert. Jeder Eintrag $D_{i,j}$ in der Difference Bound Matrix D hat die Form $(d_{i,j}, <)$ oder $(d_{i,j}, \leq)$ und repräsentiert eine Ungleichung $x_i - x_j < d_{i,j}$ oder $x_i - x_j \leq d_{i,j}$ oder $(\infty, <)$ wenn keine Ungleichung bekannt ist. $D_{j,0} = (d_{j,0}, <)$ meint die Bedingung $x_j < d_{j,0}$ und $D_{0,j} = (d_{0,j}, <)$ meint die Bedingung $0 - x_j < d_{0,j}$ bzw. $-d_{0,j} < x_j$

	0	1	2
0	$(0, \leq)$	$(-1, \leq)$	$(0, <)$
1	$(\infty, <)$	$(0, \leq)$	$(2, <)$
2	$(2, \leq)$	$(\infty, <)$	$(0, \leq)$

Abb. 9. Beispiel für ein Difference Bound Matrix

Beispiel 4. Abbildung 9 zeigt die Differenc Bound Matrix zu folgenden Uhrenzone:

$$x_1 - x_2 < 2 \wedge 0 < x_2 \leq 2 \wedge 1 \leq x_1$$

Difference Bound Matrizen sind nicht eindeutig. Mit $x_1 - x_2 < 2$ und $x_2 - x_0 \leq 2$ bekommt man $x_1 - x_0 < 4$ und mit $x_2 - x_0 \leq 2$ und $x_0 - x_1 \leq -1$ bekommt man $x_2 - x_1 \leq 1$ und damit die in Abbildung 10 dargestellte Matrix.

	0	1	2
0	$(0, \leq)$	$(-1, \leq)$	$(0, <)$
1	**$(4, <)$**	$(0, \leq)$	$(2, <)$
2	$(2, \leq)$	**$(1, \leq)$**	$(0, \leq)$

Abb. 10. Die umgewandelte Difference Bound Matrix von Abbildung 9

Aus $x_i - x_j$ und $x_j - x_k$ bekommt man also $x_i - x_k <'_{i,k} d'_{i,k}$ mit $d'_{i,k} = d_{i,j} + d_{j,k}$ und $<'_{i,k}$ ist $\leq$ wenn $<_{i,j}$ und $<_{j,\ k} \leq$ sind und $<$ sonst. Führt man alle derartigen Operationen durch, bis die Matrix sich nicht mehr ändert, hat man eine eindeutige Darstellung. In der Hauptdiagonalen steht immer $(0, \infty)$. Ist dies nicht der Fall, ist die Uhrenzone entweder leer oder nicht erfüllbar.

Analog zu den Operationen auf Uhrenzonen gibt es auch die entsprechenden Operationen auf Difference Bound Matrizen:

- Kreuzen: $D = D^1 \wedge D^2$ mit $D^1_{i,j} = (c_1, \prec_1)$ und $D^2_{i,j} = (c_2, \prec_2)$. Dann ist $D_{i,j} = (min(c_1, c_2), \prec)$ mit:
  - wenn $c_1 < c_2$, dann $\prec = \prec_1$
  - wenn $c_2 < c_1$, dann $\prec = \prec_2$
  - wenn $c_1 = c_2$ und $\prec_1 = \prec_2$, dann $\prec = \prec_1$
  - wenn $c_1 = c_2$ und $\prec_1 \neq \prec_2$, dann $\prec = <$
- Uhren Reset: $D' = D[\lambda := 0]$ mit $\lambda \subseteq X$ mit:
  - wenn $x_i, x_j \in \lambda$, dann $D'_{i,j} = (0, \leq)$
  - wenn $x_i \in \lambda$, $x_j \notin \lambda$, dann $D'_{i,j} = D_{0,j}$
  - wenn $x_j \in \lambda$, $x_i \notin \lambda$, dann $D'_{i,j} = D_{i,0}$
  - wenn $x_i, x_j \notin \lambda$, dann $D'_{i,j} = D_{i,j}$
- Verstreichen der Zeit: $D' = D^{\Uparrow}$
  - $D'_{i,0} = (\infty, <)$ für ein $i \neq 0$
  - $D'_{i,j} = D_{i,j}$ wenn $i = 0$ oder $j \neq 0$

Nach jeder Operation kann es nötig sein, die Difference Bound Matrix wieder in kanonische Form zu bringen.

Damit haben wir jetzt die Grundlagen zusammen, um aus einem Echtzeitautomaten mit Hilfe von Difference Bound Matrizen Aussagen über Systeme zu überprüfen. In [1] wird dies anhand des Echtzeitautomaten aus Abbildung 1 vorgeführt.

Im Regionengraph ist der Erreichbarkeitstest eines Zustandes exponentiell in der Anzahl der Uhren. Dies folgt aus Lemma 1. Difference Bound Matrizen können nur Konjunktionen von Uhrenbedingungen repräsentieren. Für Disjunktionen werden meist Listen von Difference Bound Matrizen genommen, was den Speicherbedarf erhöht und, da die Darstellung nicht mehr kanonisch ist, auch die Rechenzeit für den Vergleich von Difference Bound Matrizen.

4 Cottbus Timed Automata

Als kleiner Ausblick soll hier noch kurz der Cottbus Timed Automata [2] vorgestellt werden. Der Cottbus Timed Automata erweitert die bestehenden Konzepte des Echtzeitautomaten um Konzepte für die Modularität. Durch die Möglichkeit zur hierarchischen Strukturierung mit Modulen und durch einen Instanziierungsmechanismus zum mehrfachen Verwenden von Modulen sowie deren Austauschbarkeit können auch größere Systeme übersichtlich modelliert werden. Zur Verifikation von Cottbus Timed Automata-Modellen wird eine effiziente BDD-basierte Erreichbarkeitsanalyse verwendet. Mit Cottbus Timed Automatas lassen sich auch hybride Systeme (Systeme mit Uhren, Verzerrten Uhren (Echtzeitautomaten), Konstanten, diskreten Variablen, Stoppuhren (Stoppuhr-Automaten), abweichenden Uhren (Rectangular Automata), linearen Variablen (lineare hybride Automaten), nicht-linearen Variablen (allgemeine hybride Automaten)) darstellen.

Zusammenfassung

Zum Abschluß wollen wir nochmals zusammenfassen, was in den vier Kapiteln vorgestellt wurde.

Wir haben angefangen, den Echtzeitautomaten als eine endliche Beschreibung eines zeitkontinuierlichen Prozesses vorzustellen. Die dabei verwendeten Zeitschranken werden mit Hilfe von Uhren beschrieben. Im Echtzeitautomaten gibt es sowohl an den Zuständen als auch an den Übergängen Uhrenbedingungen, mit denen die Uhrenwerte verglichen werden. Anhand eines Beispiels wurde vorgeführt, wie dies anschaulich aussieht. Anschließend wurden der Echtzeitautomat und die Übergänge im Echtzeitautomaten formal definiert und der Übergangsgraph vorgestellt und definiert. Es wurden Techniken zur Erstellung von Echtzeitautomaten vorgestellt und anhand eines Beispiels vorgeführt.

Im nächsten Kapitel wurden dann die Uhrenregionen und der Regionengraph eingeführt. Diese dienen dazu, den Echtzeitautomaten in Aquivalenzklassen einzuteilen, über die man dann bestimmte Aussagen machen kann beziehungsweise

über die man dann Aussagen über das System überprüfen kann. Sowohl zu den Uhrenregionen, als auch zum Regionengraphen wurden Beispiele gegeben.

Als alternative Darstellung wurden dann die Uhrenzonen und Difference Bound Matrizen vorgestellt. Letzteres ist eine Darstellungsart für Uhrenregionen. Anhand der Difference Bound Matrizen wurde dann eine Aussage über einen schon vorher vorgestellten Echtzeitautomaten überprüft.

Als kleiner Ausblick wurde dann noch der Cottbus Timed Automata beschrieben, der Modularität in den Echtzeitautomaten einführt.

Literatur

1. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999.
2. D. Beyer, A. Noack. BDD-basierte Verifikation von Realzeit-Systemen. http://www-sst.informatik.tu-cottbus.de/∼wwwsst/LS-SST/Publications/2000/2000-BDD-basierte_Verifikation_von_Realzeit-Systemen.pdf
3. R. Alur. Timed Automata. NATO ASI Summer School on Verification of Digital and Hybrid Systems, 1998.(Available at http://www.cis.upenn.edu/alur/Nato97.ps.gz)
4. R. Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Stanfort University, 1991.
5. R. Alur, C. Courcoubetis, D. L. Dill. Model-checking for real-time systems. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pp 414–425. IEEE Computer Society Press, 1990.
6. David L. Dill. Timing assumptions and verification on finite-state concurrent systems. In J.Sifakis, ed., *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, LNCS 407, pp. 197–212. Springer, 1989.
7. C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proceedings of the 16th Real-Time Systems Symposium*, pp. 66–75. IEEE Computer Society Press, 1995.