CTL Model Checking
Hauptseminar Modellüberprüfung
WS 2001/2002

Uwe Bubeck

13. Februar 2002

Zusammenfassung

Der bemerkenswerte Durchbruch der Modellüberprüfung hat vor allem eine Beschreibungssprache ins Zentrum des Interesses gerückt: die Computation Tree Logic, kurz CTL. Die vorliegende Arbeit gibt eine Einführung in den Prozeß der Modellüberprüfung mit CTL. Nach der Vorstellung von Syntax und Semantik betrachten wir die praktische Modellierung an einem bekannten Beispiel, dem wechselseitigen Ausschluß (mutual exclusion). Dabei wird auch auf Fragen der Ausdruckskraft eingegangen, bevor zum Abschluß der grundlegende Algorithmus zur Modellüberprüfung mit CTL vorgestellt wird.

Inhaltsverzeichnis

1 Einleitung und Motivation 2

2 Die Spezifikationssprache CTL 3

2.1 Historische Entwicklung 3

2.2 Syntax 3

2.3 Semantik 4

2.4 Äquivalenzen und reduzierte Operatormengen 8

3 Anwendung und Systemmodellierung **8**

3.1 Wichtige Spezifikationsmuster 8

3.2 Beispiel: Wechselseitiger Ausschluß (Mutual Exclusion) 9

3.2.1 Problembeschreibung 9

3.2.2 Modellierung 9

3.2.3 Formulierung der Anforderungen in CTL 10

3.2.4 Korrektes Modell 10

3.2.5 Fairneß-Bedingungen 11

4 Ausdruckskraft **11**

4.1 Definition von $\mathbf{CTL}^*$ 11

4.2 LTL und CTL als Teilsprachen von $\mathbf{CTL}^*$ 12

5 Grundlegender Algorithmus zur Modellüberprüfung **13**

5.1 Beschreibung mit Pseudo-Code 13

5.2 Beispiel 15

5.3 Abschließende Bemerkungen zur Komplexität 15

1 Einleitung und Motivation

„Zeit“ ist ein schwierig zu formalisierendes Phänomen, das wissen wir nicht erst seit Einsteins Relativitätstheorie. In der Logik spiegelt sich dies wider durch die Verwendung unterschiedlicher Zeitbegriffe. Je nach Zeitauffassung erhält man somit unterschiedliche temporale Logiken, welche sich in der Praxis in zwei Klassen einteilen lassen[1]:

- Logiken mit *linearer Zeit* (linear time)
- Logiken mit *verzweigter Zeit* (branching time)

Beiden gemein ist die Diskretisierung der eigentlich kontinuierlichen Größe Zeit, gemäß der intendierten Beschreibung von und Verarbeitung mit digital arbeitenden Computersystemen. Die Unterschiede liegen in der Frage, ob es für einen Zeitpunkt *genau einen* oder *mehrere mögliche* Folgezustände gibt.

[1] Eine (vor allem historische) Bedeutung besitzen außerdem Logiken mit *zirkulärer Zeit* (circular time)

In dieser Arbeit konzentrieren wir uns mit CTL auf einen Vertreter der zweiten Klasse. Da hier für jeden Zeitpunkt alternative Folgewelten spezifiziert werden können, eignen sich Logiken mit verzweigter Zeit besonders zur Beschreibung nichtdeterministischer Zustandsübergangssysteme, beispielsweise dem in dieser Arbeit vorgestellten wechselseitigen Ausschluß. Wie sich diese Fähigkeit in der Ausdruckskraft von CTL im Vergleich zu einer Logik mit linearer Zeit niederschlägt, werden wir in Kapitel 4 genauer untersuchen.

2 Die Spezifikationssprache CTL

2.1 Historische Entwicklung

Von wiederkehrenden Naturereignissen wie der Abfolge von Tag und Nacht oder dem Zyklus der Jahreszeiten geprägt, herrschte in der Philosophie bis in die Antike ein zyklischer Zeitbegriff vor. Dieser wurde ab dem 5. Jahrhundert durch Augustinus (354-430) aufgrund von theologischen Überlegungen durch einen linearen Zeitbegriff verdrängt. Erst gegen Ende des Mittelalters wurden die Grundlagen für eine Logik mit verzweigter Zeit gelegt von Wilhelm von Ockham (1285-1349). Nach einer langen Epoche zeitloser Wahrheitsbegriffe in der Logik, unter anderem geprägt von Leibniz (1646-1716), wurde die Idee der Logik mit verzweigter Zeit im 20. Jahrhundert wieder aufgegriffen und bald darauf für das (axiomatische) Beweisen von Eigenschaften insbesondere nebenläufiger Computersysteme herangezogen.

Die hier behandelte Computation Tree Logic, kurz CTL, geht zurück auf Edmund M. Clarke und E. Allen Emerson. Mit ihren Algorithmen zur Modellüberprüfung mit CTL legten sie Anfang der 80er Jahre den Grundstein zur effizienten automatisierten Verifikation von Zustandsübergangssystemen durch Modellüberprüfung.

2.2 Syntax

In Backus-Naur-Form (BNF) werden CTL-Formeln wie folgt induktiv definiert:

$$
\begin{aligned}
\phi \quad ::= \quad & \bot \mid \top \mid p \mid \\
& (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \Rightarrow \phi) \mid \\
& \mathbf{AX}\,\phi \mid \mathbf{EX}\,\phi \mid \\
& \mathbf{AF}\,\phi \mid \mathbf{EF}\,\phi \mid \\
& \mathbf{AG}\,\phi \mid \mathbf{EG}\,\phi \mid \\
& \mathbf{A}[\phi\,\mathbf{U}\,\phi] \mid \mathbf{E}[\phi\,\mathbf{U}\,\phi]
\end{aligned}
$$

Dabei stehen $\bot$ und $\top$ für die Wahrheitswerte *falsch* und *wahr*, und p repräsentiert eine atomare Formel. Die Verknüpfungen in der zweiten Zeile sind die üblichen aussagenlogischen Junktoren. **AF, EF**, usw. werden *temporale Verknüpfungen* genannt. Sie sind jeweils aus zwei Komponenten zusammengesetzt:

- dem *Pfadquantor*:
  - **A**: „entlang aller Pfade" (*Always)*
  - **E**: „entlang (mindestens) eines Pfades" (*Exists*)
- dem *temporalen Operator*:
  - **X**: „nächster Zustand" (*neXt*)
  - **F**: „schließlich" oder „in einem zukünftigen Zustand" (*Future*)
  - **G**: „immer" oder „in allen zukünftigen Zuständen" (*Globally*)
  - **U**: „bis" (*Until*)

Dabei gelten folgende Bindungsprioritäten:

- Die unären Verknüpfungen $\neg$ sowie $\mathbf{AX}$, $\mathbf{EX}$, $\mathbf{AF}$, $\mathbf{EF}$, $\mathbf{AG}$ und $\mathbf{EG}$ binden am stärksten.
- Es folgen die binären Operatoren $\wedge$ und $\vee$.
- Danach kommen $\Rightarrow$ sowie $\mathbf{AU}$ und $\mathbf{EU}$.

Beispiele für wohlgeformte CTL-Formeln sind etwa: $\mathbf{EF}\,\mathbf{E}[r\,\mathbf{U}\,q]$, $\mathbf{AG}\,\mathbf{AF}\,r$ oder $\mathbf{A}[p_1\,\mathbf{U}\,\mathbf{A}[p_2\,\mathbf{U}\,p_3]]$. $\mathbf{FG}\,r$ oder $\mathbf{EF}[r\,\mathbf{U}\,q]$ sind jedoch nicht wohlgeformt, weil in CTL keine temporalen Verknüpfungen mit zwei temporalen Operatoren ($\mathbf{FG}$) oder gar drei Komponenten ($\mathbf{EFU}$) erlaubt sind.

2.3 Semantik

Voraussetzung für die formale Überprüfung eines Systems auf bestimmte Eigenschaften ist eine genaue Beschreibung sowohl des zu überprüfenden Systems selbst als auch der geforderten Eigenschaften. Ein Verifikationsprozeß umfaßt damit typischerweise drei Phasen:

1. Systemmodellierung
2. Spezifikation der Eigenschaften
3. Überprüfung der Eigenschaften gegen das Modell

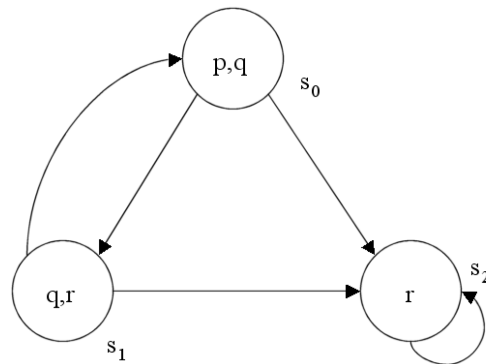

Abbildung 1: Modell mit drei Zuständen

Wir betrachten zunächst die Systemmodellierung. Dabei ist es wichtig, sich auf einem vernünftigen Abstraktionsniveau zu bewegen: Es dürfen einerseits keine wichtigen Systemeigenschaften weggelassen werden, andererseits sollte man die Verifikation nicht durch zuviele Details unnötig erschweren. Bei einem digitalen Schaltkreis könnte man sich etwa auf Gatter und Wahrheitswerte beschränken und von Spannungspotentialen abstrahieren. Wenn man dieses Prinzip konsequent fortführt, dann kann man das Verhalten eines Systems beschreiben durch *Zustände*, in denen man die Werte aller relevanten Systemvariablen zu einem bestimmten Zeitpunkt zusammenfaßt, sowie die *Übergänge* zwischen diesen. Ein solches *Zustandsübergangssystem* beschreiben wir formal durch eine *Kripke-Struktur* $\mathcal{M}$ und nennen diese ein CTL-Modell:

Eine Kripke-Struktur $\mathcal{M}$ über einer Menge A von Atomen ist ein Tupel

$$\mathcal{M} = (S, \rightarrow, L)$$

Dabei ist S eine Menge von Zuständen mit einer (links-)totalen Zustandsübergangsrelation $\rightarrow \subseteq S \times S$, d.h. für jedes $s \in S$ gibt es ein $s' \in S$ mit $s \rightarrow s'$. $L : S \rightarrow \mathcal{P}(A)$ wird Markierungsfunktion genannt.

CTL-Modelle können als gerichtete Graphen veranschaulicht werden. Jeder Knoten entspricht dabei einem Zustand und ist mit den ihm zugeordneten wahren Atomen beschriftet. Die Kanten geben dann die Zustandsübergänge an. Ein Beispiel für ein solches Zustandsübergangssystem gibt Abbildung 1.

Der vorgestellte Modellbegriff erlaubt eine adäquate Beschreibung der Gesamtstruktur eines Zustandsübergangssystems. Um jedoch mittels CTL das Verhalten des Systems in einem Zustand s beschreiben zu können, ist noch ein weiterer Transformationsschritt notwendig: aus dem Graphen ergeben sich die von s ausgehenden *Berechnungspfade*. Formal sind diese definiert als unendliche Folgen von Zuständen $(s_0, s_1, ...)$ mit $s_0 = s$ und der Eigenschaft, daß $(s_i, s_{i+1}) \in \rightarrow \ \forall i$. Durch die Totalität von $\rightarrow$ ist sichergestellt, daß jeweils ein Nachfolgezustand existiert. Notfalls können wir dies durch Hinzufügen eines Endzustands mit Schleife erreichen. Für das obige Beispielmodell sind die in s_0 beginnenden Berechnungspfade in Abbildung 2 skizziert.

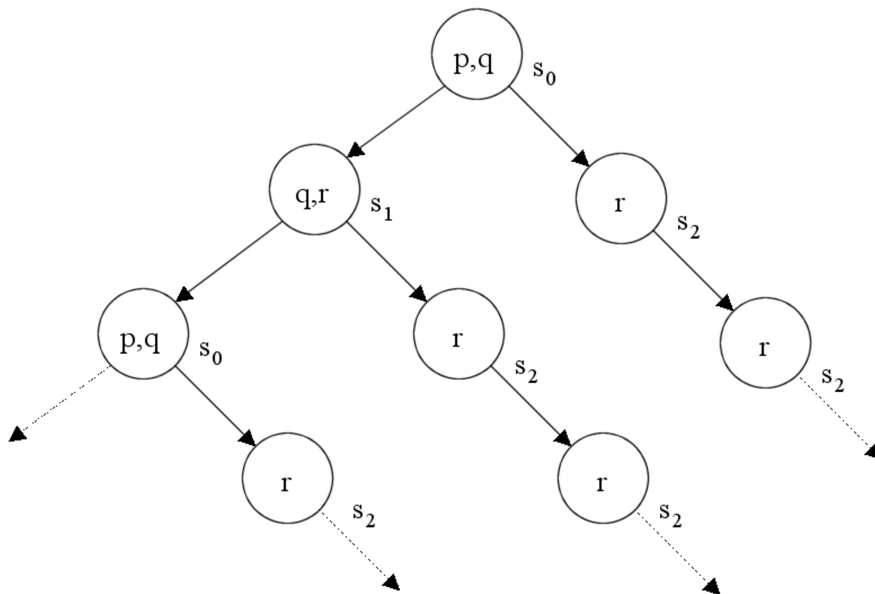

Abbildung 2: Berechnungspfade für das Beispielmodell

Auf der Grundlage einer solchen Systembeschreibung in Form einer Kripke-Struktur sollen nun bestimmte Eigenschaften geprüft werden. Im Rahmen der Modellüberprüfung mit CTL spezifiziert man jede Eigenschaft als CTL-Formel und überprüft, ob diese im Startzustand der Kripke-Struktur erfüllt wird. Dann ist die Eigenschaft verifiziert, und wir nennen die Kripke-Struktur ein Modell der Formel. Genauer: Wenn eine Formel ϕ von einem Modell $\mathcal{M} = (S, \rightarrow, L)$ im Zustand s erfüllt wird, dann schreiben wir $\mathcal{M}, s \models \phi$.

Die Modellbeziehung $\models$ definieren wir mit obigen Vorüberlegungen wie folgt durch strukturelle Induktion über den Aufbau der CTL-Formeln:

- $\mathcal{M}, s \models \top$ und $\mathcal{M}, s \not\models \bot$ für alle $s \in S$.
- $\mathcal{M}, s \models p$ genau dann, wenn $p \in L(s)$.
- $\mathcal{M}, s \models \neg\phi$ genau dann, wenn $\mathcal{M}, s \not\models \phi$.
- $\mathcal{M}, s \models \phi_1 \wedge \phi_2$ genau dann, wenn $\mathcal{M}, s \models \phi_1$ und $\mathcal{M}, s \models \phi_2$.
  Die Semantik von Disjunktion und Implikation ist analog definiert.
- $\mathcal{M}, s \models \mathbf{AX}\,\phi$ genau dann, wenn für jeden Nachfolgezustand s' (d.h. für jedes s' mit $s \rightarrow s'$) gilt $\mathcal{M}, s' \models \phi$.
- $\mathcal{M}, s \models \mathbf{EX}\,\phi$ genau dann, wenn es einen Nachfolgezustand s' (d.h. ein s' mit $s \rightarrow s'$) gibt mit $\mathcal{M}, s' \models \phi$.
- $\mathcal{M}, s \models \mathbf{AG}\,\phi$ genau dann, wenn für jeden von s ausgehenden Berechnungspfad $(s = s_0, s_1, ...)$ und alle s_i entlang des Pfades gilt $\mathcal{M}, s_i \models \phi$.

- $\mathcal{M}, s \models \mathbf{EG}\, \phi$ genau dann, wenn es einen von s ausgehenden Berechnungspfad $(s = s_0, s_1, ...)$ gibt, in dem für alle s_i entlang des Pfades gilt $\mathcal{M}, s_i \models \phi$.
- $\mathcal{M}, s \models \mathbf{AF}\, \phi$ genau dann, wenn es für jeden von s ausgehenden Berechnungspfad $(s = s_0, s_1, ...)$ ein s_i entlang des Pfades gibt mit $\mathcal{M}, s_i \models \phi$.
- $\mathcal{M}, s \models \mathbf{EF}\, \phi$ genau dann, wenn es einen von s ausgehenden Berechnungspfad $(s = s_0, s_1, ...)$ gibt, in dem für ein s_i entlang des Pfades gilt $\mathcal{M}, s_i \models \phi$.
- $\mathcal{M}, s \models \mathbf{A}[\phi_1\, \mathbf{U}\, \phi_2]$ genau dann, wenn alle von s ausgehenden Berechnungspfade $(s = s_0, s_1, ...)$ $\phi_1\, \mathbf{U}\, \phi_2$ erfüllen, d.h. wenn es ein s_i entlang des Pfades gibt mit $\mathcal{M}, s_i \models \phi_2$ und $\mathcal{M}, s_j \models \phi_1$ für alle $j < i$.
- $\mathcal{M}, s \models \mathbf{E}[\phi_1\, \mathbf{U}\, \phi_2]$ genau dann, wenn ein von s ausgehender Berechnungspfad $(s = s_0, s_1, ...)$ $\phi_1\, \mathbf{U}\, \phi_2$ erfüllt, d.h. wenn es ein s_i entlang des Pfades gibt mit $\mathcal{M}, s_i \models \phi_2$ und $\mathcal{M}, s_j \models \phi_1$ für alle $j < i$.

Umgangssprachlich kann man die Bedeutung der temporalen Verknüpfungen wie folgt zusammenfassen:

- **AX**: „in *jedem* direkten Nachfolgezustand von s gilt ...“
- **EX**: „in *einem* direkten Nachfolgezustand von s gilt ...“
- **AG**: „für *alle* Pfade ausgehend von s und *jeden* Zustand im Pfad gilt ...“
- **EG**: „es gibt *einen* Pfad ausgehend von s, in dem für *jeden* Zustand gilt ...“
- **AF**: „für *alle* Pfade ausgehend von s gibt es *einen* Zustand, in dem gilt ...“
- **EF**: „es gibt *einen* Pfad ausgehend von s, in dem für *einen* Zustand gilt ...“
- **AU**: „für *alle* Pfade ausgehend von s gilt ϕ_1 *bis* ϕ_2“
- **EU**: „es gibt *einen* Pfad ausgehend von s, in dem ϕ_1 *bis* ϕ_2 gilt“

Im Beispielmodell der Abbildungen 1 und 2 gelten beispielsweise:

- $\mathcal{M}, s_0 \models \mathbf{AF}\, r$, da in den Nachfolgezuständen s_1 und s_2 jeweils r gilt.
- $\mathcal{M}, s_0 \models \mathbf{EG}\, q$, da auf dem Pfad $s_0, s_1, s_0, s_1, ...$ stets q gilt.
- $\mathcal{M}, s_0 \models \mathbf{A}[p \wedge q\, \mathbf{U}\, r]$, da $p \wedge q$ in s_0 gilt und r in den Nachfolgezuständen.

2.4 Äquivalenzen und reduzierte Operatormengen

Zwei CTL-Formeln ϕ und ψ heißen semantisch äquivalent, in Zeichen $\phi \equiv \psi$, wenn für beliebiges Modell $\mathcal{M}$ und einen beliebigen darin enthaltenen Zustand s gilt:
$\mathcal{M}, s \models \phi$ genau dann, wenn $\mathcal{M}, s \models \psi$.

Damit ergeben sich folgende wichtige Äquivalenzen:

- Aussagenlogische Äquivalenzen gelten ebenfalls in CTL. Dies gilt insbesondere auch dann, wenn die Teilformeln echte CTL-Formeln darstellen. Beispielsweise ist $(\mathbf{AX}\, p) \vee \neg(\mathbf{AX}\, p) \equiv \top$.
- **A** und **E** bzw. **G** und **F** können als universelle und existentielle Quantoren über Pfade bzw. Zustände auf einem bestimmten Pfad aufgefaßt werden. Man erhält dann die den Gesetzen von de Morgan vergleichbaren Äquivalenzen:
  - $\neg\mathbf{AF}\, \phi \equiv \mathbf{EG}\, \neg\phi$
  - $\neg\mathbf{EF}\, \phi \equiv \mathbf{AG}\, \neg\phi$
  - $\neg\mathbf{AX}\, \phi \equiv \mathbf{EX}\, \neg\phi$
- Außerdem können wir **AF** und **EF** mit dem Until-Operator ausdrücken:
  - $\mathbf{AF}\, \phi \equiv \mathbf{A}[\top\, \mathbf{U}\, \phi]$
  - $\mathbf{EF}\, \phi \equiv \mathbf{E}[\top\, \mathbf{U}\, \phi]$
- $\mathbf{AG}\, \phi \equiv \phi \wedge \mathbf{AX}\, \mathbf{AG}\, \phi$, die sogenannte Fixpunkt-Charakterisierung

Unter Ausnutzung von Äquivalenzen können wir uns bei der Definition wohlgeformter CTL-Formeln auf eine reduzierte Operatormenge beschränken und die restlichen Verknüpfungen daraus ableiten - eine Vorgehensweise, die uns bereits aus der Aussagenlogik bekannt ist. Für den in Kapitel 5 angegebenen Algorithmus wählen wir die Operatoren $\neg$ und $\vee$ sowie die temporalen Verknüpfungen **EX**, **EU** und **EG**.

3 Anwendung und Systemmodellierung

3.1 Wichtige Spezifikationsmuster

Niemand erfindet gerne ein zweites Mal das Rad. In der Praxis greift man deshalb bei der Beschreibung von Systemeigenschaften häufig auf gängige Spezifikationsmuster zurück, zum Beispiel:

- $\mathbf{AG}$ (Anforderung $\Rightarrow$ $\mathbf{AF}$ Bereitstellung)
  In jedem Zustand folgt auf die Anforderung einer Ressource irgendwann schließlich deren Bereitstellung.

- $\mathbf{AG}\,(\mathbf{AF}$ aktiviert$)$
  Ein Prozeß wird auf jedem Berechnungspfad unendlich oft aktiviert.
- $\mathbf{AF}\,(\mathbf{AG}$ beendet$)$
  Auf jedem Berechnungspfad wird schließlich ein Endzustand erreicht und nicht mehr verlassen.

3.2 Beispiel: Wechselseitiger Ausschluß (Mutual Exclusion)

3.2.1 Problembeschreibung

Aus der Betriebssystemtheorie kennen wir das Problem des *wechselseitigen Ausschlusses* (mutual exclusion): bestimmte kritische Ressourcen in einem nebenläufigen System müssen vor dem gleichzeitigen Zugriff durch mehrere Prozesse geschützt werden. Beispiele dafür sind etwa das Drucken oder der (schreibende) Dateizugriff. Eine mögliche Lösung des Problems kapselt die Zugriffe auf diese Ressourcen in *kritische Bereiche* (critical sections) innerhalb des jeweiligen Programms. Ein Prozeß muß die Erlaubnis für das Eintreten in einen solchen Bereich vom Betriebssystem anfordern und warten, bis diese vorliegt. Gesucht ist nun ein Protokoll, welches den Zugang zu kritischen Bereichen so regelt, daß folgende Bedingungen erfüllt werden:

- **Sicherheit**: zu jedem Zeitpunkt darf sich höchstens ein Prozeß in einem kritischen Bereich befinden.
- **Lebendigkeit**: die Anforderung eines Prozesses zum Eintritt in einen kritischen Bereich wird irgendwann schließlich erfüllt.
- **Blockierungsfreiheit**: diese Anforderung kann von einem Prozeß zu jeder beliebigen Zeit gestellt werden.
- **Keine strikte Sequenzierung**: ausgeschlossen sind einfache Protokolle, die den Prozessen abwechselnd Zugang gewähren.

3.2.2 Modellierung

Wir beschränken uns hier auf zwei Prozesse. In unserem Modell durchlaufen diese Zyklen der Form $n_i \to t_i \to c_i \to n_i \to t_i \to c_i \to \cdots$, wobei $i = 1, 2$ der Index des jeweiligen Prozesses ist und die Zustände folgende Bedeutung besitzen:

- n_i (non-critical): Prozeß befindet sich außerhalb eines kritischen Bereiches
- t_i (trying): Prozeß wartet auf Zugang zu einem kritischen Bereich
- c_i (critical): Prozeß durchläuft einen kritischen Bereich

Wir nehmen ferner an, daß beide Prozesse unabhängig voneinander, jedoch nicht gleichzeitig, einen Zustandsübergang ausführen (sogenanntes *asynchrones Interleaving*).

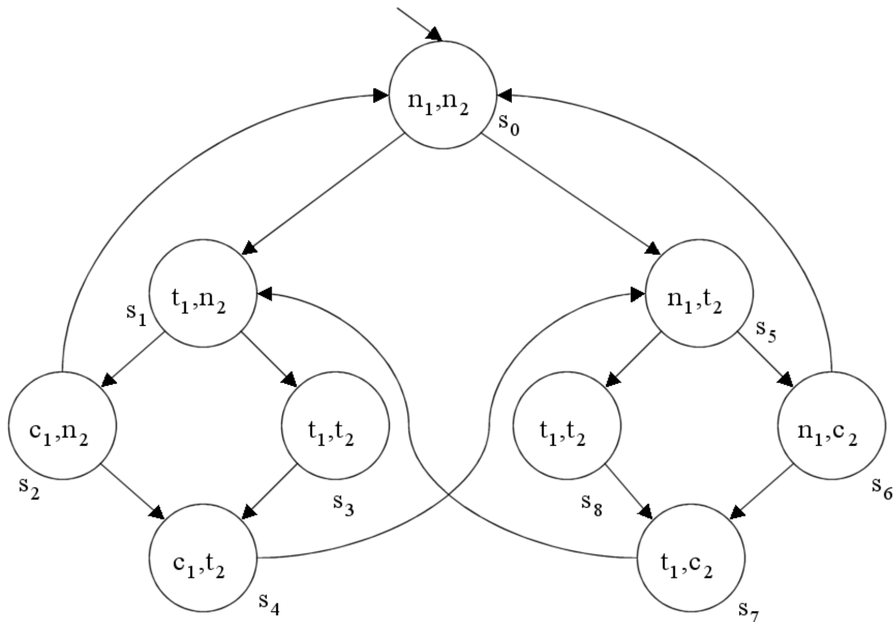

Abbildung 3: Modell für den wechselseitigen Ausschluß

3.2.3 Formulierung der Anforderungen in CTL

In CTL können wir die Anforderungen an ein korrektes Protokoll aus Sicht des ersten Prozesses wie folgt formalisieren:

- **Sicherheit**: $\mathbf{AG}\,\neg(c_1 \wedge c_2)$
- **Lebendigkeit**: $\mathbf{AG}\,(t_1 \Rightarrow \mathbf{AF}\,c_1)$
- **Blockierungsfreiheit**: $\mathbf{AG}\,(n_1 \Rightarrow \mathbf{EX}\,t_1)$
- **Keine strikte Sequenzierung**: $\mathbf{EF}\,(c_1 \wedge \mathbf{E}[c_1\,\mathbf{U}\,(\neg c_1 \wedge \mathbf{E}[\neg c_2\,\mathbf{U}\,c_1])]$.
  Umgangssprachlich: es kommt vor, daß der erste Prozeß zweimal in Folge den kritischen Bereich betreten und wieder verlassen kann.

3.2.4 Korrektes Modell

In Abbildung 3 ist ein Modell angegeben, welches im Startzustand s_0 alle vier Eigenschaften erfüllt:

- **Sicherheit**: $\neg(c_1 \wedge c_2)$ gilt in jedem Zustand, und somit auch $\mathbf{AG}\,\neg(c_1 \wedge c_2)$.
- **Lebendigkeit**: ist erfüllt, da man von den Zuständen, in denen t_1 gilt (nämlich s_1, s_3, s_7 und s_8) unweigerlich nach s_2 oder s_4 gelangt, wo c_1 gilt.

- **Blockierungsfreiheit**: ist erfüllt, da die Zustände, in denen n_1 gilt (nämlich s_0, s_5 und s_6) einen Nachfolgezustand besitzen, in dem t_1 gilt.
- **Keine strikte Sequenzierung**: wird beispielsweise durch den Pfad $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_0 \rightarrow \cdots$ erfüllt.

3.2.5 Fairneß-Bedingungen

Bei genauerer Betrachtung von Abbildung 3 fällt auf, daß ein Prozeß nicht über längere Zeit in einem kritischen Bereich verweilen kann. Möchte man diese Vereinfachung beseitigen und einem Prozeß durch das Hinzufügen einer Kante von den Zuständen s_2 und s_4 bzw. s_6 und s_7 auf sich selbst erlauben, so lange wie nötig im kritischen Bereich zu bleiben, stößt man auf ein Problem: Die Lebendigkeitsbedingung wird nicht mehr erfüllt, da einer der Prozesse dann unendlich lange im kritischen Bereich verweilen und den anderen Prozeß somit lahmlegen könnte. Einen solch unschönen Pfad, der endlos im Zustand s_2 verbleibt, würden wir gerne ausschließen mit der Bedingung, daß entlang jedes zulässigen Pfades das Atom n_1 unendlich oft wahr sein muß.

Leider läßt sich die Forderung, daß eine Eigenschaft ϕ auf allen fairen Pfaden unendlich oft gelten soll, nicht direkt in CTL ausdrücken[2]. Den recht umfangreichen Beweis dafür findet der interessierte Leser im Anhang von [1]. Offensichtlich sind wir damit an die Grenzen der Ausdruckskraft von CTL gestoßen. Im nächsten Kapitel werden wir deshalb die Ausdruckskraft dieser Sprache noch genauer untersuchen und eine Verallgemeinerung vorstellen, mit der sich auch Fairneß-Bedingungen direkt ausdrücken lassen.

4 Ausdruckskraft

4.1 Definition von CTL^*

Warum sind die temporalen Verknüpfungen immer aus zwei Komponenten zusammengesetzt? Bei der Behandlung der Syntax von CTL im Abschnitt 2.2 könnte man auf die Idee kommen, Pfadquantoren und temporale Operatoren auch einzeln und unabhängig voneinander zu benutzen und damit Formeln wie diese aufzustellen:

- $\mathbf{A}[\mathbf{X}p \vee \mathbf{XX}p]$: entlang aller Pfade ist p entweder im nächsten oder im übernächsten Zustand wahr.
- $\mathbf{E}[\mathbf{GF}p]$: es gibt einen Pfad, auf dem p unendlich oft wahr ist.

[2] Die Betonung liegt hier auf „direkt“, denn man kann natürlich (und tut dies auch in der Praxis) die Semantik von CTL so verändern, daß nur noch über faire Pfade quantifiziert wird. Eine Erweiterung der Syntax kann man dadurch vermeiden.

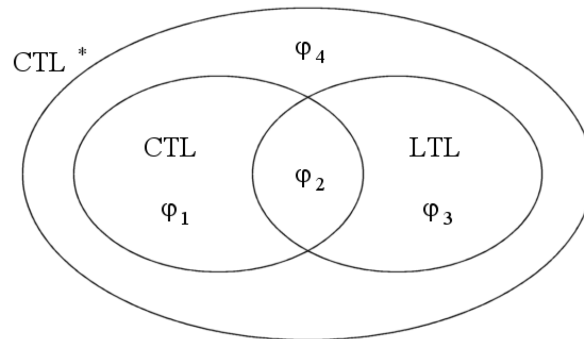

Abbildung 4: Ausdruckskraft von CTL^*, CTL und LTL im Vergleich

Diese Formeln besitzen kein Äquivalent in CTL. Für die zweite Formel haben wir dies bereits im letzten Abschnitt über Fairneß-Bedingungen angesprochen. Wir nennen die erweiterte Sprache CTL* und definieren diese formal, indem wir die wohlgeformten Formeln in zwei Klassen aufteilen:

- *Zustandsformeln*: $\phi ::= p \mid \top \mid (\neg\phi) \mid (\phi \wedge \phi) \mid \mathbf{A}[\alpha] \mid \mathbf{E}[\alpha]$,
  wobei p für eine atomare Formel steht, α für eine Pfadformel.
- *Pfadformeln*: $\alpha ::= \phi \mid (\neg\alpha) \mid (\alpha \wedge \alpha) \mid (\alpha\, \mathbf{U}\, \alpha) \mid (\mathbf{G}\, \alpha) \mid (\mathbf{F}\, \alpha) \mid (\mathbf{X}\, \alpha)$,
  wobei ϕ für eine beliebige Zustandsformel steht.

Diese verschränkt-rekursive Definition erlaubt eine gute Unterscheidung, welche Formeln in einem Zustand und welche entlang von Pfaden ausgewertet werden.

4.2 LTL und CTL als Teilsprachen von CTL^*

Linear Temporal Logic, kurz LTL, ist - der Name legt es bereits nahe - ein Vertreter der Logiken mit linearer Zeit. Folglich gibt es in LTL keine Pfadquantoren. Wir können eine Formel in LTL aber als implizit allquantifiziert betrachten und damit semantisch äquivalent zu $\mathbf{A}[\alpha]$ in CTL^*. Damit wird LTL zu einer Teilsprache von CTL^*.

Bei CTL ist diese Tatsache offensichtlich, denn die Sprache CTL erhalten wir durch Restriktion der Pfadformeln auf $\alpha ::= (\phi\, \mathbf{U}\, \phi) \mid (\mathbf{G}\, \phi) \mid (\mathbf{F}\, \phi) \mid (\mathbf{X}\, \phi)$.

Wie stehen diese Teilsprachen miteinander in Beziehung? Die Inklusionen von LTL in CTL* sowie CTL in CTL* sind jeweils echt, d.h. CTL* besitzt erwartungsgemäß eine größere Ausdruckskraft. Überraschend ist jedoch die Beziehung zwischen CTL und LTL. Letztere ist nicht etwa in CTL enthalten, wie man intuitiv vielleicht angenommen hätte, sondern sie überschneiden sich lediglich. Abbildung 4 verdeutlicht diese Erkenntnis.

Ohne Beweise geben wir im folgenden einige Beispiele für die unterschiedliche Ausdruckskraft:

- In CTL, aber nicht in LTL: $\phi_1 = \mathbf{AG}\,\mathbf{EF}\,p$
- In LTL und CTL: $\phi_2 = \mathbf{AG}(p \Rightarrow \mathbf{AF}\,q)$
- In LTL, aber nicht in CTL: $\phi_3 = \mathbf{A}[\mathbf{GF}\,p \Rightarrow \mathbf{F}\,q]$
- In CTL*, aber weder in CTL noch in LTL: $\phi_4 = \mathbf{E}[\mathbf{GF}\,p]$

5 Grundlegender Algorithmus zur Modellüberprüfung

5.1 Beschreibung mit Pseudo-Code

In diesem Kapitel stellen wir einen grundlegenden Algorithmus vor, der für eine gegebene Kripke-Struktur $\mathcal{M} = (S, \rightarrow, L)$ und eine CTL-Formel ϕ alle Zustände $s \in S$ zurückliefert, die ϕ erfüllen, d.h. für die gilt $\mathcal{M}, s \models \phi$.

Bei dem vorgestellten Algorithmus handelt es sich um einen Markierungsalgorithmus, der den induktiven Formelaufbau in CTL ausnutzt. Der Algorithmus arbeitet sich im Syntaxbaum von ϕ von den elementaren Teilformeln aufwärts. Für jede Teilformel werden alle Zustände im Graphen markiert, die diese erfüllen. Bei diesem Entscheidungsprozeß wird wieder auf die bereits gesetzten Marken zurückgegriffen. Zu Beginn sind alle Knoten genau mit den dort geltenden Atomen beschriftet. Am Ende kommt man bei der Gesamtformel ϕ an und kann schließlich alle Zustände markieren, die diese erfüllen.

Betrachten wir den Markierungsalgorithmus nun im Detail: Dabei sei ψ die aktuell betrachtete Teilformel von ϕ, und der Graph sei bereits für die Teilformeln von ψ markiert. Um aus diesen auf die notwendigen Markierungen für ψ zu schließen, benötigen wir eine Fallunterscheidung. Dabei können wir uns auf sechs Fälle beschränken, denn wie bereits in Abschnitt 2.4 erwähnt, setzen wir voraus, daß in ϕ nur noch die Verknüpfungen $\neg$ und $\vee$ sowie **EX**, **EU** und **EG** vorkommen. Für eine beliebige Formel können wir diese Bedingung zunächst durch Umformen herstellen.

Die Fälle sind:

- $\psi = p$: markiere all diejenigen Zustände $s \in S$ mit p, für die gilt $p \in L(s)$.
- $\psi = \neg\psi_1$: markiere all diejenigen Zustände $s \in S$ mit $\neg\psi_1$, die noch nicht mit ψ_1 markiert sind.
- $\psi = \psi_1 \vee \psi_2$: markiere all diejenigen Zustände $s \in S$ mit $\psi_1 \vee \psi_2$, die bereits mit ψ_1 oder ψ_2 markiert sind.
- $\psi = \mathbf{EX}\,\psi_1$: markiere all diejenigen Zustände $s \in S$ mit $\mathbf{EX}\,\psi_1$, die einen mit ψ_1 markierten Nachfolger besitzen.

- $\psi = \mathbf{E}[\psi_1\,\mathbf{U}\,\psi_2]$: In diesem Fall bestimmen wir zunächst die mit ψ_2 markierten Zustände und gehen dann rückwärts zu allen Zuständen, die von dort mit der invertierten Übergangsrelation in einem Pfad erreicht werden können, in welchem alle Zustände mit ψ_1 markiert sind. Wir markieren alle so erreichbaren Zustände mit $\mathbf{E}[\psi_1\,\mathbf{U}\,\psi_2]$.

  Eine entsprechende Prozedur in Pseudo-Code ist in Algorithmus 1 angegeben.

Algorithmus 1 Prozedur zur Behandlung von $\mathbf{E}[\psi_1\,\mathbf{U}\,\psi_2]$

```
procedure LabelEU(ψ1, ψ2)
  T := {s ∈ S | ψ2 ∈ label(s)};
  for all s ∈ T do label(s) := label(s) ∪ {E[ψ1 U ψ2]};
  while T ≠ ∅ do
    choose s ∈ T;
    T := T \ {s};
    for all t such that t → s do
      if E[ψ1 U ψ2] ∉ label(t) and ψ1 ∈ label(t) then
        label(t) := label(t) ∪ {E[ψ1 U ψ2]};
        T := T ∪ {t};
      end;
    end;
  end;
end;
```

- $\psi = \mathbf{EG}\,\psi_1$: In diesem Fall markieren wir alle bereits mit ψ_1 beschrifteten Zustände auch mit $\mathbf{EG}\,\psi_1$.

  Jetzt wiederholen wir den folgenden Schritt, bis keine Änderung mehr eintritt: lösche die Markierung $\mathbf{EG}\,\psi_1$ aller Zustände, die keinen mit $\mathbf{EG}\,\psi_1$ markierten Nachfolger besitzen.

  Eine entsprechende Prozedur in Pseudo-Code ist in Algorithmus 2 angegeben.

Algorithmus 2 Prozedur zur Behandlung von $\mathbf{EG}\,\psi_1$

```
procedure LabelEG(ψ1)
  X := ∅;
  Y := {s ∈ S | ψ1 ∈ label(s)};
  while X ≠ Y do
    X := Y;
    Y := Y ∩ {s ∈ Y | ∃s' ∈ Y : s → s'};
  end;
  for all s ∈ Y do label(s) := label(s) ∪ {EG ψ1};
end;
```

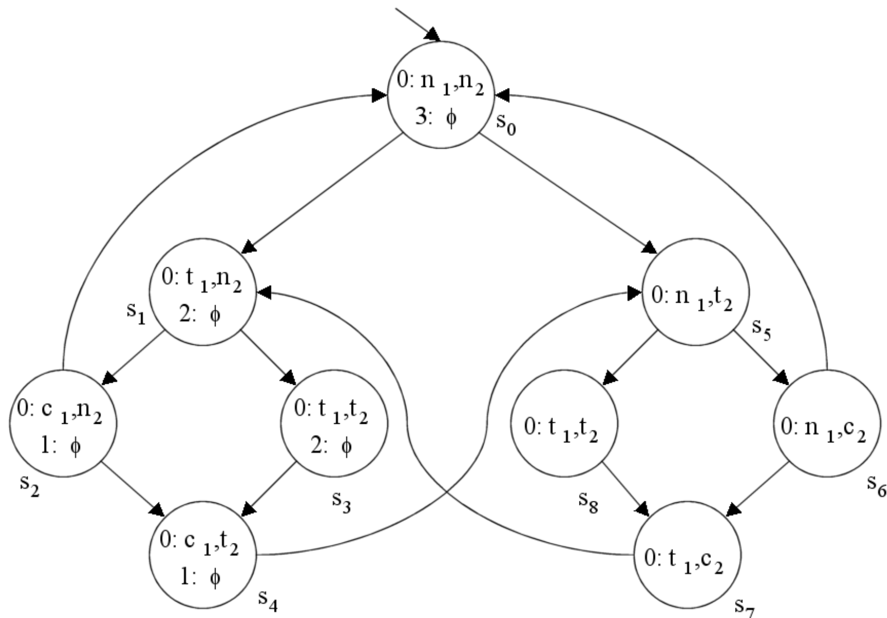

Abbildung 5: Beispieldurchlauf für den wechselseitigen Ausschluß

5.2 Beispiel

Zur Verdeutlichung wenden wir den Algorithmus für die Formel $\phi = \mathbf{E}[\neg c_2 \, \mathbf{U} \, c_1]$ auf das Modell für den wechselseitigen Ausschluß aus Abbildung 3 an. Wie in der Abbildung 5 dargestellt, sind zunächst (Phase 0) die Zustände lediglich mit den dort gültigen Atomen markiert. In Phase 1 werden alle Zustände, die c_1 erfüllen, mit ϕ markiert. Dies sind s_2 und s_4. In Phase 2 folgen alle Zustände, die nicht c_2 erfüllen und einen bereits mit ϕ markierten Nachfolgezustand besitzen. Dies sind s_1 und s_3. Schließlich wird s_0 markiert, da dieser Zustand nicht c_2 erfüllt und auf den bereits markierten Zustand s_1 zeigt. Der Algorithmus terminiert jetzt, weil keine weiteren Knoten markiert werden können.

5.3 Abschließende Bemerkungen zur Komplexität

Bei der bisherigen Behandlung von **EG** ist der Aufwand für die Suche nach den Nachbarzuständen der in der Menge Y enthaltenen Zustände quadratisch zur Größe des Zustandsraums. Man kann dies noch effizienter gestalten, wenn man sich eines graphentheoretischen Hilfsmittels bedient: den starken Zusammenhangskomponenten. Damit kann man im Zustandsraum Bereiche größtmöglicher Ausdehnung finden, in denen jeder Zustand mit jedem anderen verbunden ist. Das entspricht genau der Bedeutung des temporalen Operators **G** (globally), wenn der Graph zuvor auf die Zustände restringiert wird, die ϕ erfüllen.

Mit dieser Verbesserung ergibt sich folgendes Resultat:

Der Test, ob eine CTL-Formel ϕ in einem Zustand s einer Kripke-Struktur $\mathcal{M} = (S, \to, L)$ erfüllt ist, kann von einem Algorithmus mit Laufzeit $O(|\phi| \cdot (|S| + | \to |))$ ausgeführt werden.

Der Aufwand ist damit linear sowohl in der Länge der Formel als auch der Größe des Modells. Für CTL^* hingegen ist das Problem der Modellüberprüfung PSPACE-vollständig. Damit stellt CTL trotz Beschränkungen in der Aussagekraft auch für größere Systeme eine gute Wahl dar.

Allerdings bekommt man mit wachsender Systemgröße Probleme mit einem starken Anwachsen des Zustandsraumes, der sogenannten *Zustandsexplosion*. Dies liegt darin begründet, daß die Größe des Modells exponentiell wächst in der Anzahl der Variablen und der Anzahl der parallel arbeitenden Komponenten.

Literatur

[1] E. Allen Emerson, Joseph Y. Halpern:
„Sometimes“ and „Not Never“ Revisited: On Branching versus Linear Time Temporal Logic, Journal of the ACM 33(1): 151-178, 1986

[2] Emund M. Clarke, E. Allen Emerson:
Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, ACM Transactions on Programming Languages and Systems 8(2): 244-263, 1986

[3] Michael R. A. Huth und Mark D. Ryan:
Logic in Computer Science: Modelling and reasoning about systems, Cambridge University Press 2001

[4] Edmund M. Clarke, Orna Grumberg, Doron A. Peled:
Model Checking, MIT Press 2000

[5] Friedrich von Henke:
Maschinelles Beweisen, Vorlesungsunterlagen WS 2000/2001

[6] Fabio A. Schreiber:
Is Time a Real Time? An Overview of Time Ontology in Informatics,
in: W. A. Halang, A. D. Stoyenko: Real Time Computing, Springer 1994