Formale Techniken der Software-Entwicklung

Matthias Hölzl, Christian Kroiß

30. Juni 2014

Logik

- Aussagen über „unveränderliche Welten“:

$$\Phi \models \phi$$

  in jeder „Welt“ (jedem Modell) $\mathcal{M}$ in dem alle Aussagen aus Φ gelten, gilt auch ϕ
- Das ermöglicht Aussagen über Programme, die terminieren und ein Ergebnis zurückgeben:

```
sum(n): RECURSIVE nat =
    IF n = 0 THEN 0 ELSE n + sum(n - 1) ENDIF
    MEASURE (LAMBDA n: n)
closedFormOfSum: THEOREM
    sum(n) = (n * (n + 1))/2
```

- Aber: was ist mit *reaktiven Systemen*, d.h., mit Systemen, die kontinuierlich mit ihrer Umwelt interagieren?

Beispiel: Virtuelle Maschine

- In der letzten Vorlesung haben wir mit der virtuellen Maschine ein Beispiel gesehen, wie man ein derartiges System modellieren kann:

```
step(state: vm_state): vm_state =
    ...
```

- Die VM wird durch eine Funktion repräsentiert, die einen Zustand auf einen neuen Zustand abbildet (wir sehen dabei im Moment von Komplikationen wie dem Rückgabetyp `lift[vm_state]` ab)

Beispiel: Virtuelle Maschine

- Die VM führt diese `step`-Funktion in einer Schleife aus und liefert damit eine Folge von Zuständen

$$s_0 \to s_a \to \cdots \to s_n$$

oder

$$s_0 \to s_1 \to s_2 \to \ldots$$

- Damit wir endliche und unendliche Abläufe formal gleich behandeln können, setzen wir endliche Abläufe mit einem Pseudozustand s_t fort, schreiben also für die erste Folge:

$$s_0 \to s_a \to \cdots \to s_n \to s_t \to s_t \ldots$$

Modellierung von Zustandsübergängen der VM

- Bei der VM gibt es unendlich viele mögliche Zustände, denn ein Zustand enthält
  - ganze und natürliche Zahlen als Elemente
  - einen Stack unbegrenzter Größe
  - eine Funktion `[string -> vm_data]` für die globalen Variablen
- Der Übergang von einem Zustand zu seinem Nachfolger ist eine Funktion, und damit eindeutig

Abstraktere Modellierung von Zustandsübergängen der VM

- Wenn wir die aktuelle Instruktion der VM nicht im Zustand mitführen, so erhalten wir eine Funktion

  ```
  step(inst: vm_inst, state: vm_state): vm_state =
      ...
  ```

- Wenn wir in `step` von der aktuellen Instruktion abstrahieren bekommen wir eine Funktion

  ```
  next_states(state: vm_state): set[vm_state] =
      ...
  ```

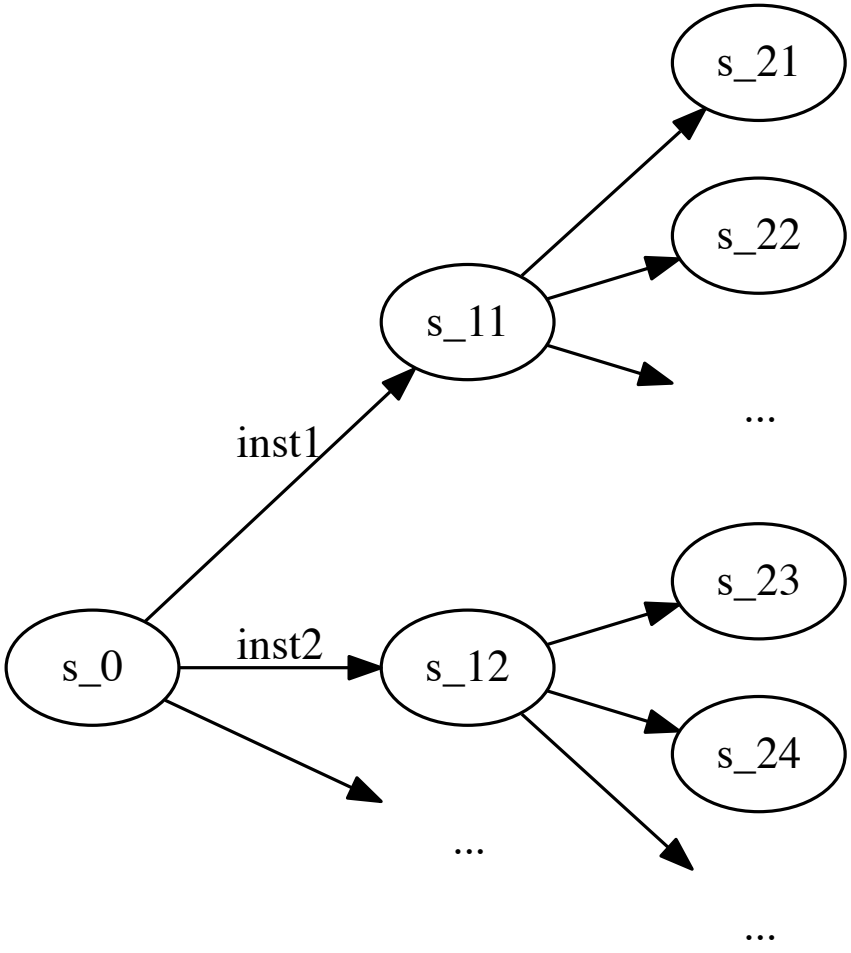
  die alle möglichen Nachfolgezustände von `state` zurückgibt (`next_states` ist möglicherweise auf ein bestimmtes Programm spezialisiert)

- Damit gleichwertig ist eine Relation

  ```
  next_states_rel(state, succ_state: vm_state): bool =
      ...
  ```

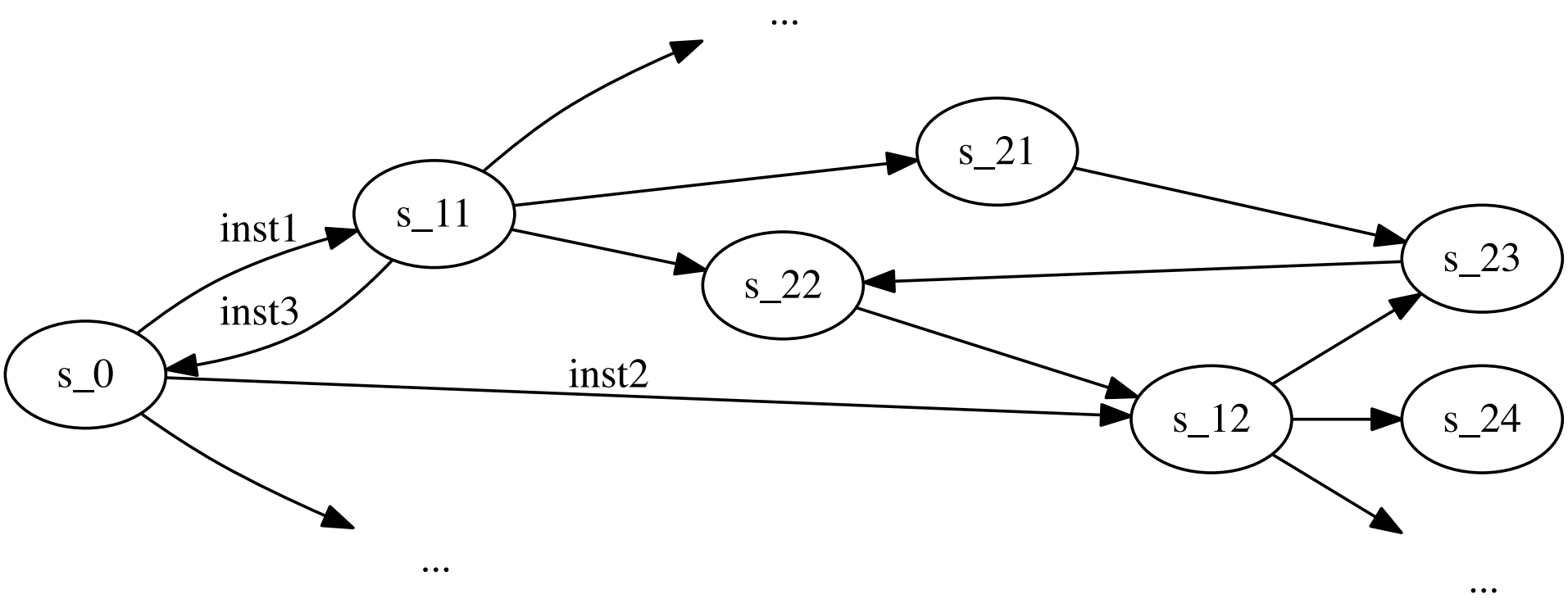
Abstraktere Modellierung von Zustandsübergängen der VM

`next_states`:



Abstraktere Modellierung von Zustandsübergängen der VM

Für manche Folgen von Instruktionen führt `next_states` zu identischen Zuständen. Wenn wir jeden Zustand nur einmal darstellen wollen erhalten wir einen (unendlichen) Graphen, keinen Baum:



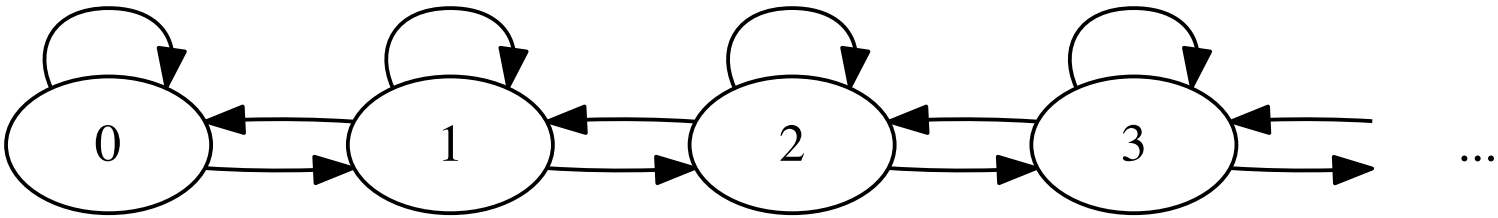
Abstraktere Modellierung von Zustandsübergängen der VM

- Bei der Verifikation von Eigenschaften eines Systems sind wir oft nur an gewissen Aspekten des Zustandes interessiert.
- Wenn wir z.B. die VM in Hardware implementieren, wollen wir evtl. nur eine begrenze Stacktiefe erlauben. Dann können wir alle Zustände mit der gleichen Stacktiefe identifizieren:

  ```
  stack_depth(state: vm_state): nat
  ```

Abstraktere Modellierung von Zustandsübergängen der VM

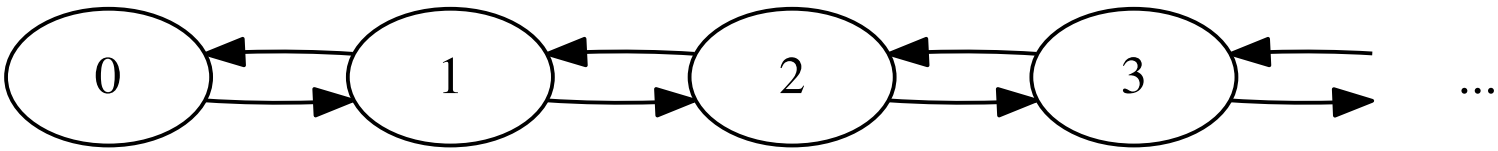
Dann erhalten wir ein wesentlich einfacheres Transitionssystem:



(In den Zuständen ist die Tiefe des Stacks in Frames angegeben, d.h. in Zustand *i* gilt die PVS-Formel `stack_depth(state) = i`.)

Abstraktere Modellierung von Zustandsübergängen der VM

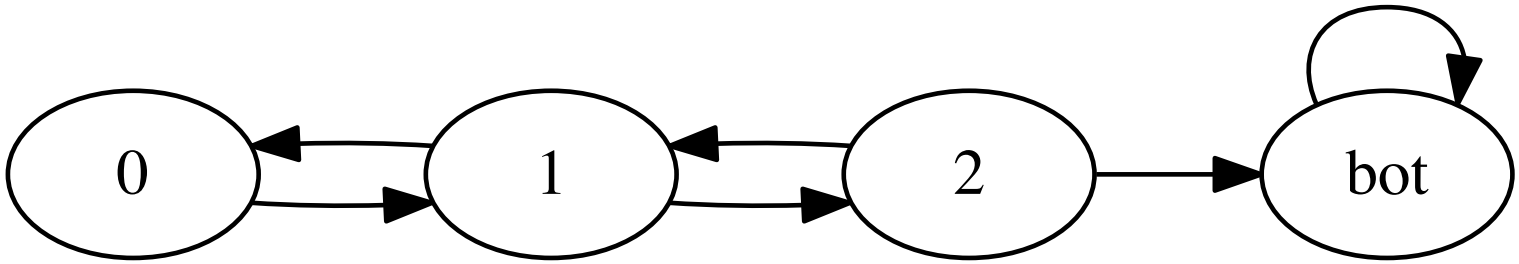
Wir können dann weiter abstrahieren und nur Transitionen betrachten, die die Tiefe des Stacks verändern:



Dieses Transitionssystem ist immer noch unendlich.

Abstraktere Modellierung von Zustandsübergängen der VM

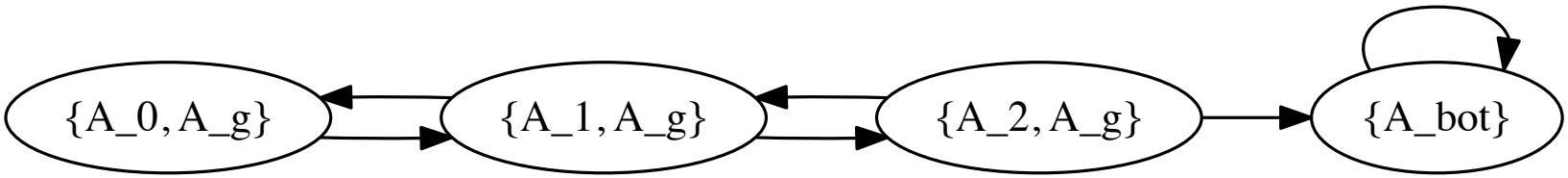
Wenn wir in der Hardware nur Speicherplatz für zwei Stackframes bereitstellen, können wir das Modell nochmals vereinfachen:



Dieses Transitionssystem hat nur noch endlich viele Zustände.

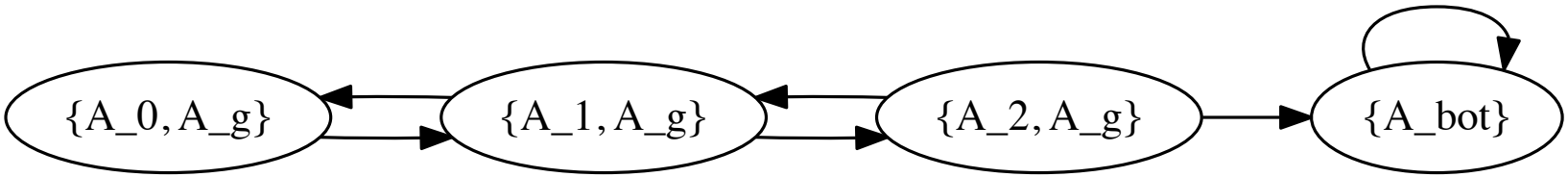
Abstraktere Modellierung von Zustandsübergängen der VM

Wenn wir die Aussagenlogischen Formeln A_0 (A_1, A_2) als „Die Tiefe des Stacks ist 0“ (1, 2) und $A_\perp$ als „Die zulässige Stacktiefe wurde überschritten“ definieren, dann können wir jedem Zustand (genau) eine dieser Formeln zuordnen. Wenn wir noch andere Eigenschaften betrachten (z.B., A_g = „Das Programm hat die gültige Stacktiefe nicht überschritten“) können wir jedem Zustand eines endlichen Transitionssystems eine Menge von atomaren Formeln zuordnen, die in diesem Zustand gilt:



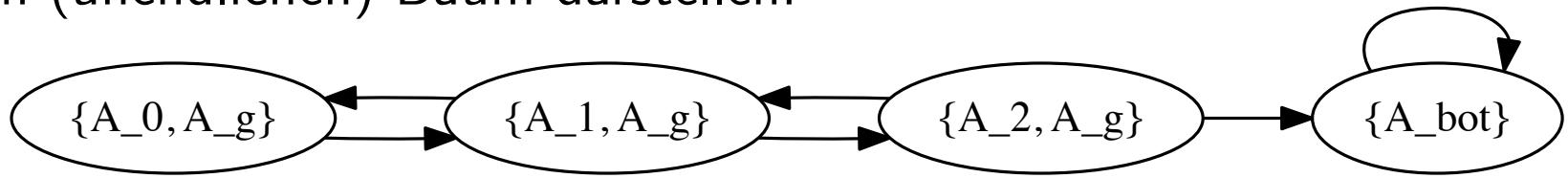
Abstraktere Modellierung von Zustandsübergängen der VM

Jedem Zustand entspricht also eine aussagenlogisches Modell (d.h. eine Variablenbelegung), die Transitionen entsprechen Übergängen zwischen Modellen.

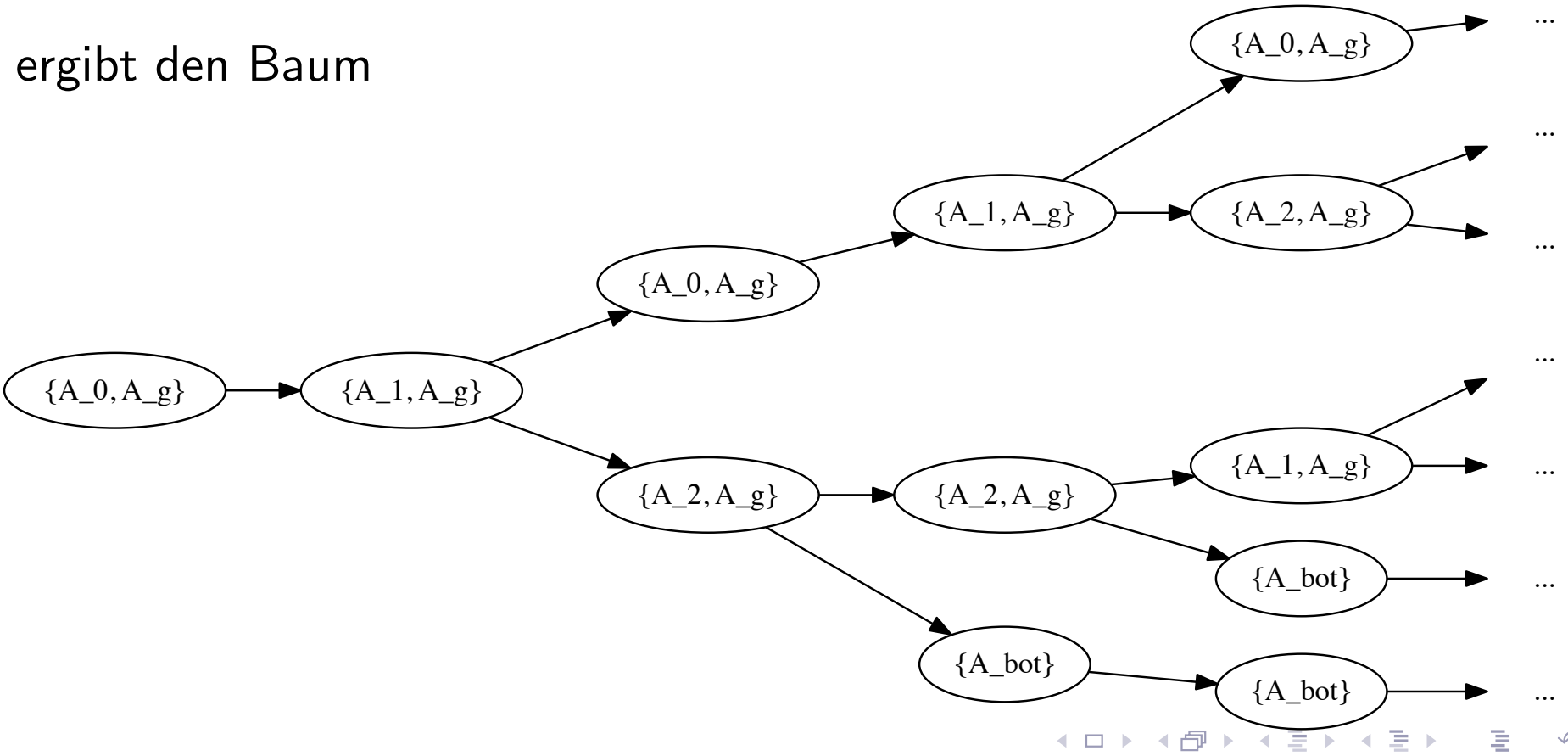


Auffalten eines Transitionssystems

Alle möglichen Durchläufe durch ein Transitionssystem lassen sich in einem (unendlichen) Baum darstellen:



ergibt den Baum



Auffalten eines Transitionssystems

Jeder Ast in einem derartigen Baum entspricht einem Durchlauf durch das Transitionssystem. Wenn wir einen Knoten in diesem Baum auswählen (der ja einen Zustand s im zugrundeliegenden Transitionssystem entspricht), können wir z.B. fragen:

- Gilt eine Aussage A in s?
- Gilt eine Aussage A im unmittelbar auf s folgenden Zustand?
- Gilt A in allen Durchläufen, die mit s beginnen?
- Gibt es einen mit s beginnenden Durchlauf π, so dass A in allen Knoten von π gilt?
- Gibt es einen mit s beginnenden Durchlauf, in dem A irgendwann gilt?

Temporallogik

Eine Temporallogik ist eine Logik, in der man Operatoren hat, die derartige „zeitlichen“ Zusammenhänge beschreiben. Dabei unterscheidet man Logiken, mit

- linearer Zeit: diese Temporallogiken beschränken sich auf Fragen, die einen zukünftigen Ablauf, bzw. alle in einem Knoten beginnenden Abläufe betreffen, und solche mit
- verzweigter Zeit: in diesen Logiken kann man über zukünftige Abläufe existenziell und universell quantifizieren.

Wir werden im Folgenden die Logik CTL* (Computation Tree Logic*) betrachten, der ein verzweigtes Modell der Zeit zugrundeliegt.

Syntax von CTL*

In CTL* werden die Aussagenlogischen Formeln erweitert um

- Die Pfadquantoren **A** (für alle Pfade) und **E** (es gibt einen Pfad)
- Die Temporaloperatoren
  - $\mathbf{X}\,\phi$ (neXt time): die Eigenschaft ϕ gilt im nächsten Zustand des Pfades
  - $\mathbf{F}\,\phi$ (Future): die Eigenschaft ϕ wird auf einem Zustand des Pfades gelten
  - $\mathbf{G}\,\phi$ (Globally): die Eigenschaft ϕ gilt für jeden Zustand des Pfades
  - $\phi\,\mathbf{U}\,\psi$ (Until): gilt, wenn es einen Zustand auf dem Pfad gibt, für den ψ gilt, und wenn in allen vorhergehenden Zuständen ϕ gilt
  - $\phi\,\mathbf{R}\,\psi$ (Release): gilt, wenn ψ bis zum ersten Zustand gilt, in dem ϕ gilt (einschließlich dieses Zustandes). ϕ muss aber nicht gelten.

Beispiele

- Es ist unmöglich in einen Zustand zu kommen, in dem `started` gilt, aber nicht `ready`: $\mathbf{G}\,\neg(\texttt{started} \wedge \texttt{ready})$
- Jede Anfrage wird (irgendwann) beantwortet:
  $\mathbf{G}(\texttt{requested} \Rightarrow \mathbf{F}\,\texttt{acknowledged})$
- Ein Prozess wird in jedem Pfad unendlich oft freigeschaltet:
  $\mathbf{A}\,\mathbf{G}\,\mathbf{F}\,\texttt{enabled}$
- Es gibt einen Pfad, auf dem der Prozess deadlockt: $\mathbf{E}\,\mathbf{F}\,\mathbf{G}\,\texttt{deadlock}$
- Wenn ein Prozess unendlich oft aktiviert wird, dann startet er unendlich oft: $\mathbf{G}\,\mathbf{F}\,\texttt{enabled} \Rightarrow \mathbf{G}\,\mathbf{F}\,\texttt{start}$
- Ein Lift, der vom 2. Stock aus nach oben in den 5. Stock fährt ändert seine Richtung nicht:
  $\mathbf{G}(\texttt{floor}(2) \wedge \texttt{direction}(\texttt{up}) \wedge \texttt{buttonPressed}(\texttt{up}) \Rightarrow (\texttt{direction}(\texttt{up})\,\mathbf{U}\,\texttt{floor}(5)))$
- Ein System kann von jedem Zustand aus neu gestartet werden:
  $\mathbf{G}\,\mathbf{E}\,\texttt{restart}$

Syntax von CTL*

In CTL* gibt es *Zustandsformeln (state formulas)* und *Pfadformeln (path formulas)*. Sei AP die Menge der atomaren Aussagen.

- *Zustandsformeln* sind folgendermaßen rekursiv definiert:
  - Ist $A \in AP$, dann ist A eine Zustandsformel
  - Sind ϕ und ψ Zustandsformeln, so sind auch $\neg\phi$, $\phi \wedge \psi$ und $\phi \vee \psi$ Zustandsformeln
  - Ist ϕ eine Pfadformel, dann sind $ctlA\phi$ und $\mathbf{E}\,\phi$ Zustandsformeln
- *Pfadformeln* sind folgendermaßen rekursiv definiert:
  - Ist ϕ eine Zustandsformel, dann ist ϕ auch eine Pfadformel
  - Sind ϕ und ψ Pfadformeln, dann sind auch $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\mathbf{X}\,\phi$, $\mathbf{F}\,\phi$, $\mathbf{G}\,\phi$, $\phi\,\mathbf{U}\,\psi$ und $\phi\,\mathbf{R}\,\psi$ Pfadformeln

  CTL* ist die Menge der durch diese Regeln generierten Zustandsformeln. Wir schreiben $ZF(\phi)$ wenn ϕ eine Zustandsformel ist und $PF(\phi)$ wenn ϕ eine Pfadformel ist.

Transitionssysteme und Kripke-Strukturen

Sei $\mathcal{L}$ eine logische Sprache mit Atomen AP.

Definition

Ein Transitionssystem $\mathcal{M} = (S, \to, L)$ besteht aus einer Menge von Zuständen S, einer binären Relation $\to$ auf S, so dass es für jedes $s \in S$ ein Element $s' \in S$ gibt, für das $s \to s'$ gilt, und einer Funktion $L : S \to \mathfrak{P}(A)$, die jedem Zustand $s \in S$ die Menge der Atome zuordnet, die in s gelten.

Ein Transitionssystem $\mathcal{M}$ zusammen mit einer Menge $S_0 \subseteq S$ nennt man eine *Kripke-Struktur*.

Ein *Pfad* in $\mathcal{M}$ ist eine unendliche Folge von Zuständen, $\pi = s_0, s_1, \ldots$, so dass für alle i gilt $s_i \to s_{i+1}$. (Ein Pfad ist somit ein unendlicher Ast in dem vom Transitionssystem erzeugten Baum.)

Transitionssysteme und Kripke-Strukturen

- Wir schreiben π^{s_i} oder π^i für den Suffix von π, der mit dem Zustand s_i beginnt; wir schreiben auch π^s um einen Pfad zu benennen, der mit dem Zustand s beginnt
- Für eine Zustandsformel ϕ schreiben wir $\mathcal{M}, s \models \phi$ wenn ϕ für den Zustand s im Transitionssystem $\mathcal{M}$ gilt
- Für eine Pfadformel ϕ schreiben wir $M, \pi \models \phi$, wenn ϕ entlang des Pfades π im Transitionssystem $\mathcal{M}$ gilt
- Üblicherweise schreiben wir $\mathcal{M}$ nicht explizit hin, da das Transitionssystem meist aus dem Kontext klar ist.

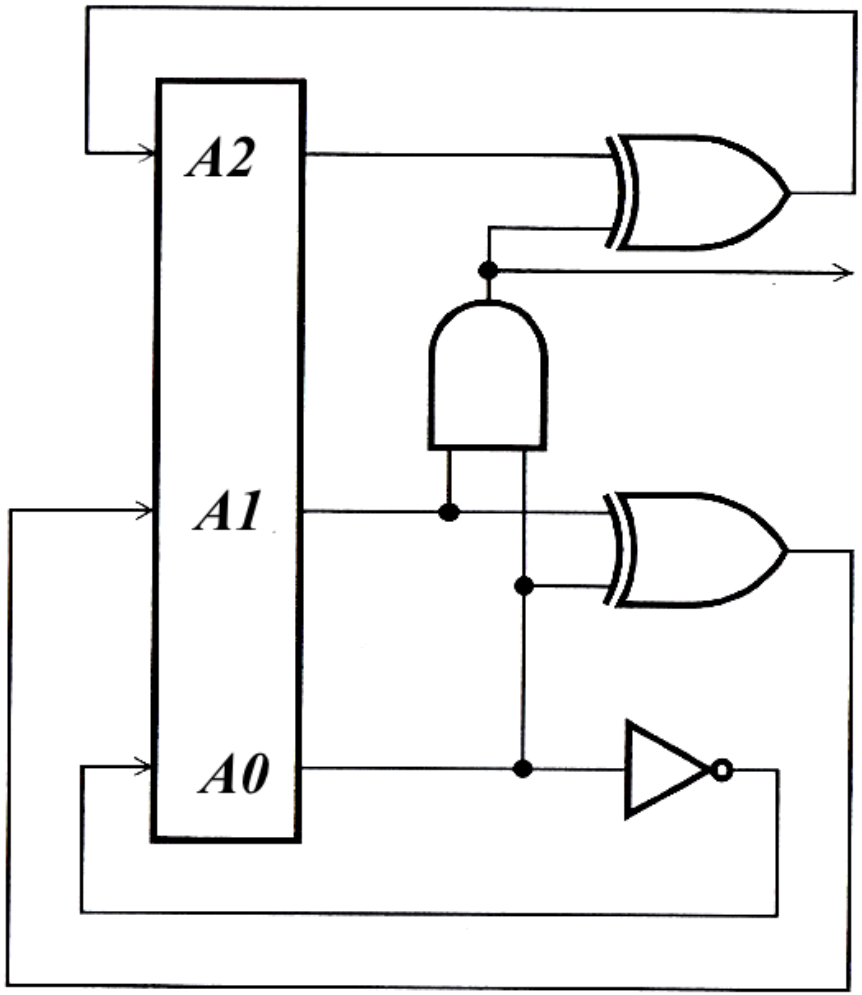
Transitionssysteme und Kripke-Strukturen

Wenn AP eine Teilmenge der aussagenlogischen Variablen (oder eine Menge bestehend aus prädikatenlogischen Grundatomen) ist, dann identifiziert man s wie üblich mit der Menge der Atome, für die s wahr ist

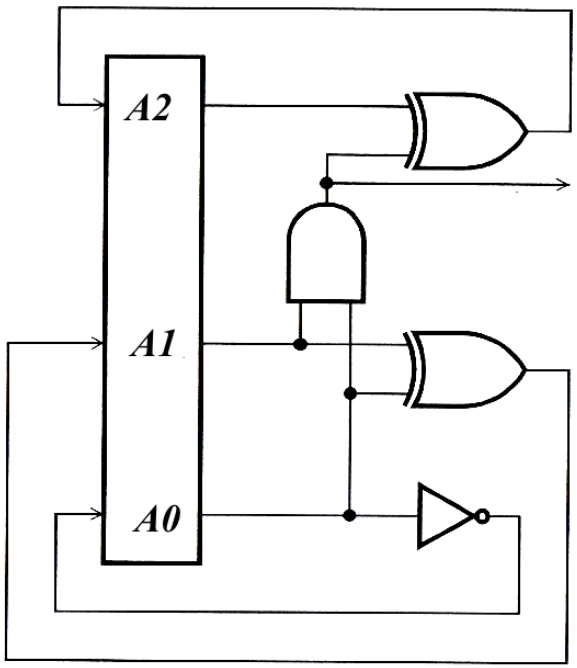
$$s \sim \{A_0, \ldots, A_n\} \iff s(A_0) = \cdots = s(A_n) = \text{wahr und}$$
$$s(B) = \text{falsch sonst.}$$

Ist $\to$ eine Funktion, dann schreibt dann oft $A_i' = f_i(A_1, \ldots, A_n)$ um auszudrücken, dass $s \to s'$ genau dann gilt, wenn $s'(A_i) = [\![A_i]\!]s' = [\![f(A_1, \ldots, A_n)]\!]s$ ist.

Beispiel: Synchroner Zähler Modulo 8



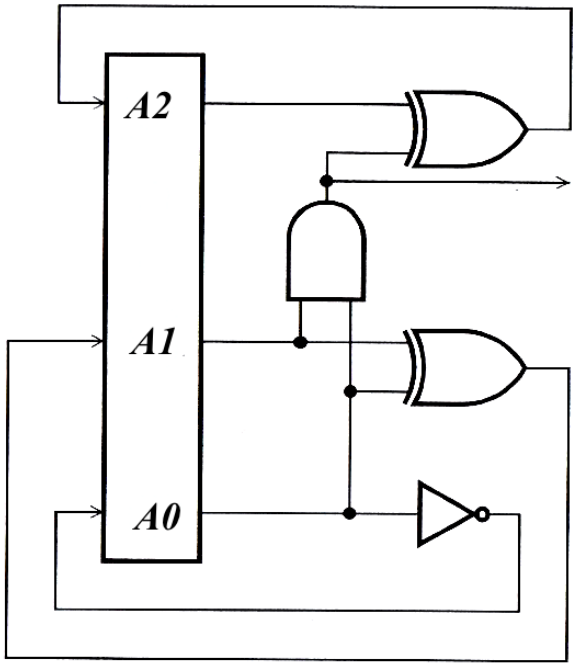
Beispiel: Synchroner Zähler Modulo 8



Zustände: Belegungen für A_0, A_1, A_2

$$A_0' = \neg A_0$$
$$A_1' = A_0 \oplus A_1$$
$$A_2' = (A_0 \wedge A_1) \oplus A_2$$

Beispiel: Synchroner Zähler Modulo 8



Zustände: Belegungen für A_0, A_1, A_2

$$A_0' = \neg A_0$$
$$A_1' = A_0 \oplus A_1$$
$$A_2' = (A_0 \wedge A_1) \oplus A_2$$

$s \to s'$ genau dann, wenn

$$s'(A_0) = [\![\neg A_0]\!]s$$
$$s'(A_1) = [\![A_0 \oplus A_1]\!]s$$
$$s'(A_2) = [\![(A_0 \wedge A_1) \oplus A_2]\!]s$$

Semantik von CTL*

$$
\begin{aligned}
\mathcal{M}, s \models A &\iff p \in L(s) \\
\mathcal{M}, s \models \neg\phi &\iff \mathcal{M}, s \not\models \phi \\
\mathcal{M}, s \models \phi \vee \psi &\iff \mathcal{M}, s \models \phi \text{ oder } \mathcal{M}, s \models \psi \\
\mathcal{M}, s \models \phi \wedge \psi &\iff \mathcal{M}, s \models \phi \text{ und } \mathcal{M}, s \models \psi \\
\mathcal{M}, s \models \mathbf{E}\,\phi &\iff \text{Es gibt einen Pfad } \pi^s \text{ mit } \mathcal{M}, \pi \models \phi \\
\mathcal{M}, s \models \mathbf{A}\,\phi &\iff \text{Für alle Pfade } \pi^s \text{ gilt } \mathcal{M}, \pi \models \phi
\end{aligned}
$$

Semantik von CTL*

$$\mathcal{M}, \pi \models \phi \iff ZF(\phi),\ \pi \text{ beginnt mit } s \text{ und } \mathcal{M}, s \models \phi$$

$$\mathcal{M}, \pi \models \neg\phi \iff \mathcal{M}, \pi \not\models \phi$$

$$\mathcal{M}, \pi \models \phi \vee \psi \iff \mathcal{M}, \pi \models \phi \text{ oder } \mathcal{M}, \pi \models \psi$$

$$\mathcal{M}, \pi \models \phi \wedge \psi \iff \mathcal{M}, \pi \models \phi \text{ und } \mathcal{M}, \pi \models \psi$$

$$\mathcal{M}, \pi \models \mathbf{X}\,\phi \iff \mathcal{M}, \pi^1 \models \phi$$

$$\mathcal{M}, \pi \models \mathbf{F}\,\phi \iff \text{es gibt } k \text{ mit } \mathcal{M}, \pi^k \models \phi$$

$$\mathcal{M}, \pi \models \mathbf{G}\,\phi \iff \text{für alle } i \geq 0 \text{ gilt } \mathcal{M}, \pi^i \models \phi$$

$$\mathcal{M}, \pi \models \phi\,\mathbf{U}\,\psi \iff \exists k \geq 0. \mathcal{M}\pi^k \models \psi \wedge \forall 0 \leq i < k. \mathcal{M}, \pi^i \models \phi$$

$$\mathcal{M}, \pi \models \phi\,\mathbf{R}\,\psi \iff \forall j \geq 0.\ \text{If } \forall i < j. \mathcal{M}, \pi^i \not\models \phi \text{ then } \mathcal{M}, \pi^j \models \psi$$

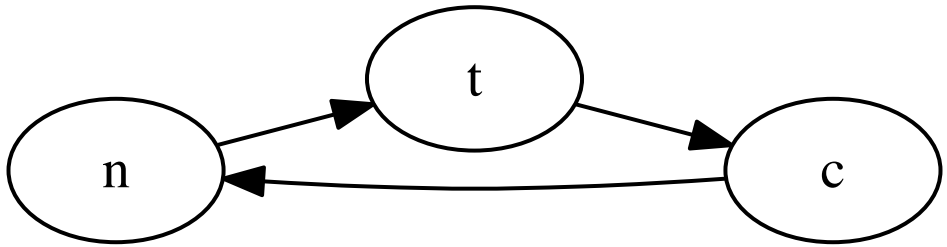
Beispiel: Wechselseitiger Ausschluss

Mehrere Prozesse greifen auf eine gemeinsame Ressource zu. Dabei sollen folgende Eigenschaften garantiert werden:

- *Sicherheit (Safety):* Nur ein Prozess ist im kritischen Bereich
- *Liveness:* Wenn ein Prozess verlangt in den kritischen Bereich zu kommen, kann er das irgendwann
- *Nicht-blockierend:* Ein Prozess kann immer verlangen in den kritischen Bereich zu kommen
- *Nicht-strikt-sequenziell:* Die Prozesse müssen nicht streng abwechselnd in den kritischen Bereich

Beispiel: Wechselseitiger Aussschluss

Wir modellieren jeden Prozess durch folgendes Diagramm:



Beispiel: Wechselseitiger Ausschluss

- *Safety:* Nur ein Prozess ist im kritischen Bereich: $\mathbf{A}\,\mathbf{G}\,\neg(c_1 \wedge c_2)$
- *Liveness:* Wenn ein Prozess verlangt in den kritischen Bereich zu kommen, kann er das irgendwann: $\mathbf{A}\,\mathbf{G}(t_i \Rightarrow \mathbf{F}\,c_i)$
- *Nicht-blockierend:* Ein Prozess kann immer verlangen in den kritischen Bereich zu kommen: $\mathbf{A}\,\mathbf{G}(n_i \Rightarrow \mathbf{E}\,\mathbf{X}\,t_i)$
- *Nicht-strikt sequenziell* Die Prozesse müssen nicht streng abwechselnd in den kritischen Bereich: $\mathbf{E}\,\mathbf{F}(c_1 \wedge \mathbf{E}(c_1\,\mathbf{U}(\neg c_1 \wedge \mathbf{E}(\neg c_2\,\mathbf{U}\,c_1))))$

Einschränkungen: LTL und CTL

- CTL (Computation-Tree Logic): Jeder der Operatoren **X**, **F**, **G**, **U** und **R** muss unmittelbar nach einem Pfadquantor stehen. CTL ist eine Sprache, die nur verzweigte Zeit beinhaltet
- LTL (Linear-Time Logic): Nur Formeln der Form $\mathbf{A}\,\phi$ mit einer Pfadformel ϕ, in der alle Zustands-Teilformeln atomar sind. (Typischerweise wird der Allquantor weggelassen.) LTL ist eine Sprache, die nur lineare Zeit beinhaltet

Beispiel: Wechselseitiger Aussschluss

- *Safety:* Nur ein Prozess ist im kritischen Bereich: $\mathbf{A}\,\mathbf{G}\,\neg(c_1 \wedge c_2)$ (LTL, CTL)
- *Liveness:* Wenn ein Prozess verlangt in den kritischen Bereich zu kommen, kann er das irgendwann: $\mathbf{A}\,\mathbf{G}(t_i \Rightarrow \mathbf{F}\,c_i)$ (LTL)
- *Nicht-blockierend:* Ein Prozess kann immer verlangen in den kritischen Bereich zu kommen: $\mathbf{A}\,\mathbf{G}(n_i \Rightarrow \mathbf{E}\,\mathbf{X}\,t_i)$ (CTL)
- *Nicht-strikt sequenziell* Die Prozesse müssen nicht streng abwechselnd in den kritischen Bereich: $\mathbf{E}\,\mathbf{F}(c_1 \wedge \mathbf{E}(c_1\,\mathbf{U}(\neg c_1 \wedge \mathbf{E}(\neg c_2\,\mathbf{U}\,c_1))))$ (CTL)

Model Checking

Sei $\mathcal{M}$ eine Kripke-Struktur, die ein (nebenläufiges System) mit endlich vielen Zuständen beschreibt, und sei ϕ eine temporallogische Formel, die eine gewünschte Eigenschaft des Systems beschreibt.

- Finde alle Zustände für die gilt $\mathcal{M}, s \models \phi$, also $MS = \{s \in S \mid M, s \models \phi\}$
- Damit die Eigenschaft für das System erfüllt ist, muss gelten $S_0 \subseteq MS$
- Wenn die Eigenschaft nicht erfüllt ist kann ein Pfad angegeben werden, der die Eigenschaft nicht erfüllt.

Vorteile/Nachteile des Model Checkings

- Model-Checking ist weitgehend automatisch
- Model-Checking kann zur Verifikation einzelner Eigenschaften hergenommen werden; ein vollständiges Systemmodell ist nicht notwendig
- Model-Checking eignet sich für nebenläufige und reaktive Systeme
- Gilt eine Eigenschaft nicht, so können Model-Checker typischerweise Fehler-Traces ausgeben
- Problem: Explosion des Zustandsraums *(State Explosion)*
- Model-Checking funktioniert nicht für datenabhängige Systeme