2 Verifkation eingebetteter Systeme

Zur Vorlesung

Embedded Systems

WS 14/15

Reiner Kolla

2.1 Verifkation – Motivation und Techniken

Zur Vorlesung

Embedded Systems

WS 14/15

Reiner Kolla

Wozu Verifikation

Verifikation ist vor allem bei sicherheitskritischen Eingebetteten Systemen unverzichtbar. Fehlerhafte Abweichungen von der Spezifikation haben in der Vergangenheit oft verheerenden Schaden angerichtet:

Beispiele:

- Strahlenüberdosis bei der Therac-25

  Mindestens 6 Patienten erhielten eine 100fache Überdosis

  3 Krebspatienten starben daran
- AT&T Telefon Totalausfall 1990 Kosten mehrere 100 M$
- Crash der Ariane 5 (Konversionsfehler) 500 M$
- Pentium Bug (1 von 9 Milliarden zufällig gewählten Divisionen erzeugte Fehler) Kosten 500 M$

Verifikation versus Validierung

Verification = check that we are building the thing right

Validation = check that we are building the right thing

Die **Verifikation** soll den Nachweis liefern, dass das System die Anforderungen immer erfüllt (Korrektheit der Implementierung im Bezug auf die Spezifikation).

Die **Validierung** entspricht eher dem kritischen Überprüfen der Spezifikation anhand eines ersten Designs auf Richtigkeit und Vollständigkeit.

Testen (mit adequaten Werkzeugen zum Ermitteln von Qualitätsmetriken) ist ein Mittel für beides und weit verbreitet. Verifikation lässt sich damit nicht bewerkstelligen, allenfalls Falsifikation. Daher vertiefen wir diese primitiven aber aufwändigen Techniken hier nicht weiter.

Populäre Techniken der Verifikation

Peer Reviewing

Statische Methode der manuellen Durchsicht und Prüfung des Codes durch einen Experten. Im Mittel bringt dies 60% der Fehler, subtile Fehler werden meist nicht gefunden.

Testen

Dynamische Methode, häufig unterstützt durch Qualitätsmetriken (Code Coverage). 30% bis 50% der Gesamtkosten gehen auf das Konto von Testen, bei Hardware sind oft 70% des entwickelten Codes Testbenches.

Man kann mit Tests die Anwesenheit von Fehlern nachweisen, nicht jedoch deren Abwesenheit!

Formale Methoden

Formale Methoden basieren auf formalen Sprachen zur Spezifikation von Eigenschaften wie auch zur Konstruktion des Systems. Gebräuchliche Verifikationstechniken mit formalen Methoden sind:

Deduktive Methoden

Hier liefert man einen mathematischen Beweis, dass die Implementierung die Spezifikation erfüllt. Meist werden Theorembeweiser oder Beweisprüfer als Werkzeuge benutzt. Problem: Sehr aufwändig, das System muss die Form einer Mathematischen Theorie haben.

Model Checking

Systematische und erschöpfende Überprüfung der Spezifikation auf allen erreichbaren Systemzuständen. Meist vollautomatisch durch sogenannte Modellchecker. Problem: Zustandsraumexplosion

Modelbasierte Simulation und Test

Exploration möglicher Verhalten mit Überprüfung der Spec.

Formale Methoden - Meilensteine

Mathematische Korrektheit von Programmen

(Turing 1949)

Syntax basierte Beweistechnik für sequentielle Programme

(Hoare 1969)

Erzeugt ein sequ. Programm auf eine gegeben Eingabe die richtige Ausgabe? Basierend auf Beweisregeln in Prädikatenlogik.

Syntax basierte Beweistechnik für nebenläufige Programme

(Pnuelli 1977)

Behandelt Eigenschaften im Bezug auf die während der Berechnung erreichten Zustände. Beweisregeln in temporaler Logik.

Automatische Verifikation nebenläufiger Programme

Modellbasiert statt auf Beweisregeln bauend, dafür aber automatisch. Liefert Gegenbeispiel wenn das System nicht die Spec erfüllt.

Beispiele für Beweisregeln

Backward axiom

$$\frac{}{\{A[e/x]\}\ x:=e\ \{A\}}$$

Invariant rule

$$\frac{\{I \wedge b\}\ P\ \{I\}}{\{I\}\ \text{while } b \text{ do } P\ \{I \wedge \neg b\}}$$

Cut rule

$$\frac{\{A\}\ P\ \{B\} \quad \{B\}\ Q\ \{C\}}{\{A\}\ P;Q\ \{C\}}$$

Logical Rule

$$\frac{A \Rightarrow A' \quad \{A'\}\ P\ \{B'\} \quad B' \Rightarrow B}{\{A\}\ P\ \{B\}}$$

Was bedeutet Model-Checking?

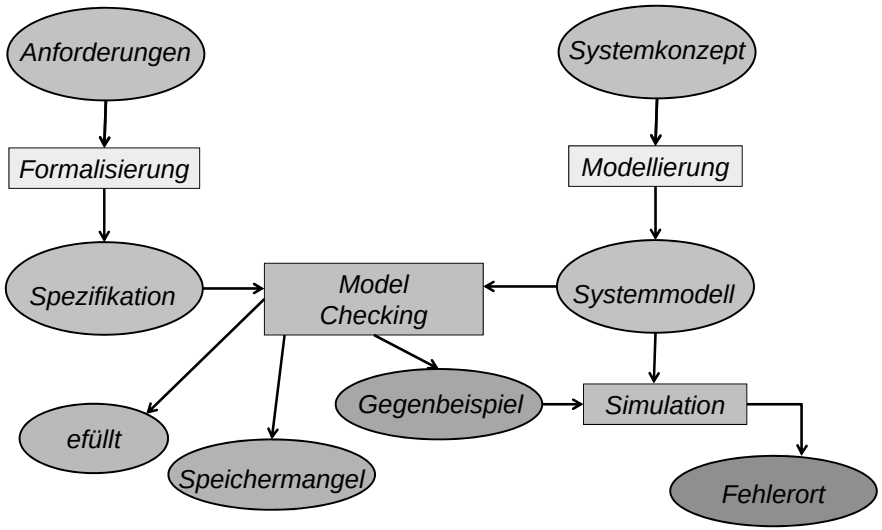
Man betrachtet

Spezifikation: Formale Definition der Eigenschaften des gewünschten Systems in Form von „Sätzen einer Mathematischen Theorie“ (Aussagen in einer temporalen Logik)

Implementierung: Formale Definition der konkreten Konstruktion des Systems (als diskretes Transistionssystem)

Model Checking: Ist die Implementierung ein Model für die Theorie. (Gelten die Sätze für die Implementierung)

Modelchecking - das große Bild



Transitionssysteme

Wir haben im Prinzip Modelle als Transitionssysteme kennengelernt, wobei Petrinetze sowohl endliche als auch unendliche Zustandübergangssysteme modellieren können.

Rechner sind nichts anderes als gigantische Transitionssysteme, Programme beschreiben i.d.R. nichts anderes als gigantische Transitionssysteme, parallel arbeitende Rechner, verteilte Programme, es sind im Prinzip alles einfache Transitionssysteme oder als solche abstrakt modellierbar.

Es ist also legitim sich im Zusammenhang mit der Verifikation für solche Transitionssysteme zu interessieren.

Ein Tupel $T = (S,(Act,)\rightarrow,St)$ heißt **Transitionssystem**, wobei

- S eine Menge von Zuständen
- $\rightarrow \subseteq S \times S$ die Transitionsrelation ($\rightarrow \subseteq S \times Act \times S$)
- Act eine Menge von Aktionen
- $ST \subseteq S$ eine Menge von Startzuständen ist.

Transitionssysteme

Auf die Startzustände kann man auch verzichten oder auf genau einen gehen, der dann vermöge → nichtdeterministisch in *St* verzweigt.

Auch die Aktionen sind Komfort, sie machen es leichter, mehrere kooperierende Transitionssysteme zu definieren und Synchronisationsbedingungen über Aktionen zu formulieren. Wenn wir die Zustandsmenge endlich machen und die Aktionen als Ein/Ausgabemarkierungen auffassen, sind wir wieder bei unseren vertrauten Automaten.

Bei endlicher Zustandsmenge ist auch klar, dass man die Übergangsrelation als gerichteten Graphen darstellen kann.

Die grundsätzliche Frage ist, wie können wir Zustände eines Systems bestimmen oder voneinander unterscheiden? Sie ist auch eng verbunden mit der Frage, wie man Eigenschaften eines Transistionssystems beschreiben und letztlich dann auch beweisen kann.

Beispiel: einfaches 2 Prozess System

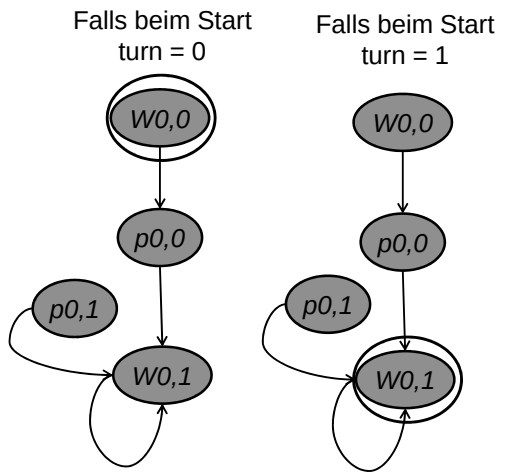
P = **start**;
while true do
*w*0: **wait** (turn = 0);
*p*0: /* Produziere */
turn := 1;
od;
end

K = **start**;
while true do
*w*1: **wait** (turn = 1);
*k*1: /* Konsumiere */
turn := 0;
od;
end

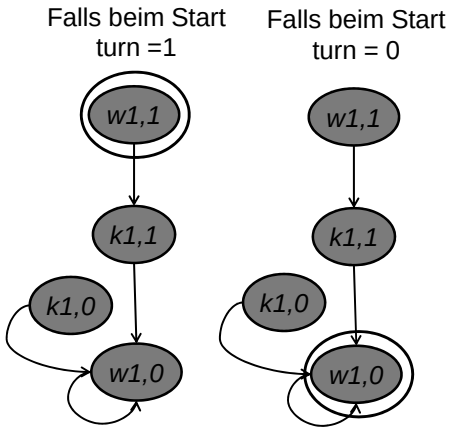
Beispiel: Producer als TS

P = **start**;
while true do
*w*0: **wait** (turn = 0);
*p*0: /* Produziere */
turn := 1;
od;
end

S_{prod} ={ w0,p0}×{0,1}



Beispiel: Konsumer als TS



K = **start**;
while true do
*w*1: **wait** (turn = 1);
*k*1: /* Konsumiere */
turn := 0;
od;
end

S_{cons} ={ w1,k1}×{0,1}

Beispiel: das Gesamtsystem als TS

Das Gesamtsystem erhält man nun, indem man das Produktsystem unter Identifikation der gemeinsamen Variablen bildet, d.h. wie betrachten alle Zustände aus

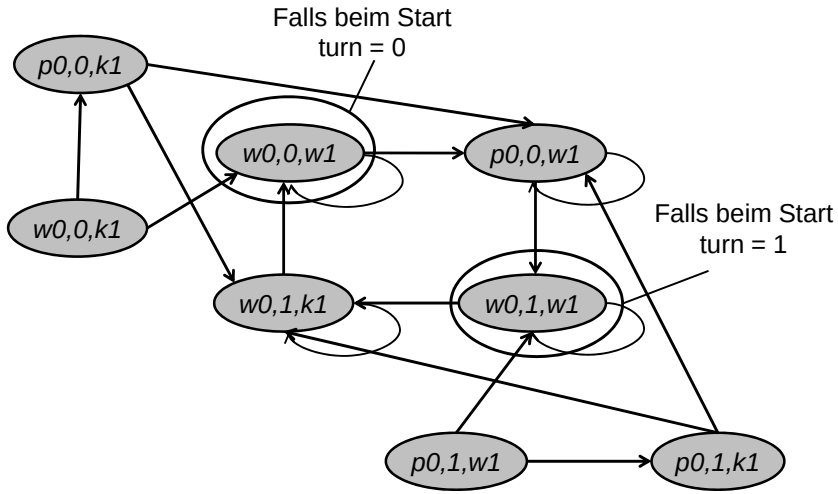
$$((x,t),(y,t')) \in S_{prod} \times S_{cons}$$

mit gleichem Wert der zur Variablen turn gehörigen Komponente, d.h. (t=t').

Da wir die gemeinsamen Variablen identifizieren reichen also für das Produktsystem Tripel wobei

$(x,t,y) \rightarrow (x',t',y')$ gdw. $(x,t) \rightarrow_{prod} (x',t')$ und $(y,t) \rightarrow_{cons} (y',t')$

Beispiel: Produktsystem



Kripke Strukturen

Praktisch stellt man Eigenschaften eines Systems durch Messung fest, d.h. Eigenschaften sind im einfachsten Falle binär, sie gelten oder gelten nicht.

Alles was wir über das Verhalten eines System wissen oder erfahren können sollte sich also grundsätzlich gesehen auf endlich viele (wir können nur endlich viel messen) **atomare Aussagen** zurückführen lassen. Wir erweitern ein Transistionssystem zu einer **Kripke Struktur**

K= (S,(Act,)→,St, AP,L)

dabei kommen noch zwei Komponenten hinzu:

- *AP* eine (endliche) Menge von atomare Aussagen (atomic propositions).
- $L: S \rightarrow 2^{AP}$ eine Markierung der Zustände mit den durch sie erfüllten atomaren Aussagen (Labeling)

Bemerkung: Isoliert betrachtet sind Zustände *s,s'* mit *L(s) = L(s')* nicht unterscheidbar, aber sie können zu unterschiedlichen Folgezuständen führen.

Kripke Strukturen am Beispiel

Unser Transitionssystem zu den Producer/Consumer Prozessen könnte man durch folgende atomare Aussagen erweitern:

AP = {p,k} mit

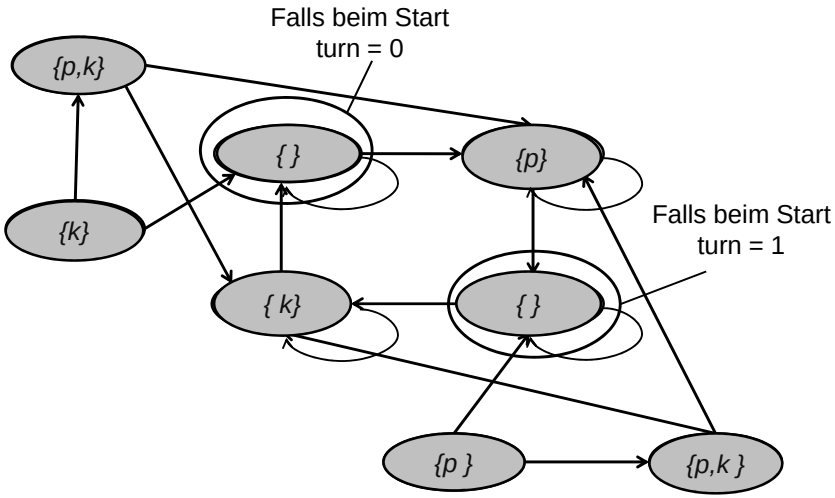
p: es wird gerade produziert

k: es wird gerade konsumiert.

Es ist klar, dass *p* wahr ist, wenn der Producer in Zustand *(p0,*)* ist und *k* genau dann wahr ist, wenn der Consumer im Zustand *(k1,*)* ist.

Für das Gesamtsystem ergibt sich also folgende Kripke Struktur:

Beispiel: Das Labeling



Linear-Zeit versus Baum-Zeit

Formal werden wir zu verifizierende Eigenschaften unserer Systeme in **temporaler Logik** ausdrücken.

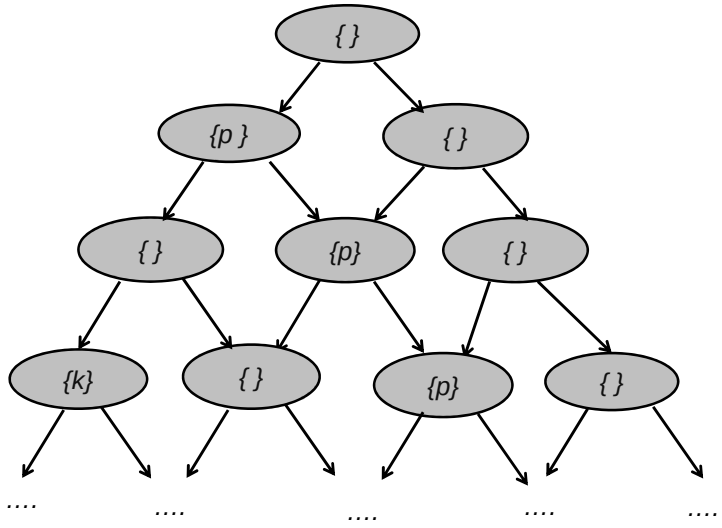
Die **Linear-Zeit Logik** (**LTL** – linear time logic) erlaubt Aussagen über Labelings **aller** (unendlichen) Berechnungsfolgen der Kripke Strukur. So sind z.B.

{} {p} {} {k} {} {p} {} {k} … oder auch *{} {p} {p} {p} {p} ....*

mögliche Folgen von labelings unserer Beispielstruktur. LTL Aussagen gelten entweder für alle Folgen oder nicht.

Die **Baum-Zeit Logik** (**CTL** – computation tree logic) erlaubt die Formulierung von Aussagen über Berechnungsbäume, d.h. man kann Existenz und Allquantoren für die Pfade in den Unterbäumen nutzen.

Beispiel: Berechnungs“baum“



Beispiel temporaler Eigenschaften

In unserem Bespiel kann man etwa folgende Eigenschaft ablesen:

„Es ist niemals möglich, dass zugleich p und k gilt“.

Dies ist eine Invariante. Sie gilt nicht in der ganzen Struktur, aber für alle von Start aus erreichbaren Zustände.

Eine andere Eigenschaft wäre etwa:

„Es wird stets konsumiert was produziert wurde.“

Dies ist eine Reaktionseigenschaft. Sie gilt nicht, weil z.B.

{} {p} {} {} {} ....

eine mögliche Folge von labelings ist. Unter einer Fairnessannahme (jeder Prozess kommt unendlich oft dran) gilt die Eigenschaft aber.