Prozessautomatisierung

Karl Friedrich Gebhardt

Auflage vom 10. September 2022

Prof. Dr. K. F. Gebhardt

Tel: 0711-184945-11(16)(15)(12)
Fax: 0711-184945-10
email: kfg@lehre.dhbw-stuttgart.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

Aspekte:

- Prozess*daten*verarbeitung – PDV: prozessgekoppelte EDV, Messdatenerfassung, Messdatenauswertung, Datenreduktion, Kurvenfit, Graphik, Statistik, Datenverwaltung, DDC ***(direct digital control)*** und höhere Regelverfahren, Modell-Anpassung, Big Data / Small oder Smart Data
- Prozess*leit*system – PLS: Einflussnahme auf den Prozess, Steuerung, fahrbare Maschine, Kraftwerk, Flugzeug, Auto, Look-and-Feel-Systeme, Produktionsleittechnik
- Prozess*automat*isierungssystem – PAS: Vollautomatisierung, "Arbeit sich selbst erledigen lassen."

In der Fertigungstechnik hatte man es mit schnellen Prozessen, aber primitiven Daten (Bit-Daten oder boolschen Daten) zu tun, so dass dort Prozessrechner zunächst nicht eingesetzt wurden, da sie einerseits zu langsam waren, andererseits nicht die Notwendigkeit komplexer Datenverarbeitung bestand.

Prozessrechner wurden aber schon frühzeitig in der chemischen Verfahrenstechnik eingesetzt, da es sich hier i.A. um relativ langsame Prozesse mit komplexen Datenverknüpfungen handelt (Analogdaten, Regelalgorithmen).

Die chemische Industrie spricht von **Verfahrensleittechnik oder Prozessleittechnik**, Maschinenbau- und Elektroindustrie von **Fertigungsleittechnik**. Wegen der ungeheuren Leistungsfähigkeit moderner Prozessrechensysteme sehen wir heutzutage hier keine wesentlichen Unterschiede und betrachten grundsätzlich beide Bereiche unter den Begriffen **Produktionsleittechnik *(MES Manufacturing Execution System)*** oder noch allgemeiner **Automatisierungs-System**.

Im Englischen umfassen die Begriffe ***System Control*** oder ***Automation*** in eleganter Weise beide Bereiche. Der Aspekt der Software dieses ganzen Bereichs wird auch mit **Realzeit-System (Echtzeit-System)** ***(Real-Time System, RTS)*** oder ***Embedded System (ES, EmS)*** bezeichnet.

Die Unterteilung des Gesamtsystems in einen zu steuernden Prozess P und das Automatisierungs-System AS ist ein nützliches Modell:

P $\Longleftrightarrow$ **AS**

Das *Ziel* des Einsatzes von Prozessrechensystemen ist letzten Endes die Vollautomatisierung mit folgenden Vor- und Nachteilen:

- Höhere Arbeitskultur durch Aufhebung der räumlichen und zeitlichen Bindung des Menschen an den Produktionsprozess. (Fließband, Nähmaschine,

  gesundheitsgefährdende, stereotype Routinearbeiten werden dem Menschen abgenommen, Freisetzung des Menschen für *schöpferische Aufgaben*)

- Erhöhung der Leistungsfähigkeit eines Menschen durch Ergänzung seiner sensorischen, aktorischen und kommunikativen Fähigkeiten (Chauffeure, Soldaten).

- Höhere Sicherheit der Anlagen. Automatisierungssysteme machen weniger Fehler, sind aber "dümmer" und "sehen, hören und fühlen" weniger.

- Höhere Wirtschaftlichkeit

  - Arbeitsproduktivität, Geschwindigkeit, Effektivität (Kraft greift dort an, wo sie am wirksamsten ist, d.h. die geleistete Arbeit maximal ist.)
  - Qualität, Genauigkeit (Automatisierung der Automobilfertigung hat sicherlich zu qualitativ besseren Fahrzeugen geführt.)
  - Einsparung von Energie und Rohstoff (Optimales Ausnutzen von Energie und Einsatzstoffen erfordert aufwendige Optimierungsrechnungen.)

- Komplizierte Prozessverflechtungen werden möglich, die bei menschlicher Steuerung nicht oder nur mit besonderen Spezialisten möglich sind. (Bei einem Crash-Test fallen etwa eine Million Messwerte pro Sekunde an. Signalanlagen: Versuchen Sie eine Modellbahnanlage mit mehr als vier gleichzeitig fahrenden Zügen zu steuern. Umwelt-Messnetze: An sehr vielen Messstellen müssen Konzentrationen bestimmter Gase in der Luft gemessen und auf Grenzwerte überprüft werden. Die entstehende Datenflut ist nur mit Prozessrechensystemen beherrschbar. Lichtsteuerung in einem Theater.)

- Zunehmende Abhängigkeit von Technologie. (Eventuell geht überhaupt nichts mehr, Buchungen, Fahrkartenverkauf, Hochregallager.) Ein schwäbischer Philosoph hat sich intensiv mit der Dialektik von Herrschaft und Knechtschaft auseinandergesetzt, Georg Wilhelm Friedrich Hegel (1770-1831). Bei ihm geht die Argumentation schließlich soweit, dass der Herr der eigentliche Knecht ist. Wie kann man sich gegen diese Herrschaft der Sklaven wehren? Indem man die Systeme möglichst gut kennenlernt.

  Zu dem Thema lohnt sich auch – weniger anspruchsvoll – der Science-Fiction- Roman "Die Maschine stirbt" von Asimov.

Man unterscheidet drei verschiedene Größenordnungen von Prozess-Automatisierungssystemen:

- ***(deeply) embedded,* integrierte Systeme:** Embedded Systeme gibt es überall (Waschmaschine, Herd, Fotoapparat, Telefon, Kopiergerät, Drucker, Auto, Rakete, Smart-Card, Telefon, Handy, Unterhaltungselektronik, Medizintechnik). Ohne es zu wissen, besitzt man mehrere solcher Systeme und benutzt sie täglich.

An solch einem System kann man nicht mehr programmieren. Das wird einmal programmiert und dann in einer "fest verdrahteten" Version massenweise hergestellt. Das wird sich aber ändern. (Die update-bare Waschmachine oder die Waschmaschine, die sich für jeden Waschvorgang das Programm aktuell aus dem Strom-Netz zieht, ist in Sicht. Das Smartphone, das jederzeit eine neue Applikation laden kann.) Ein embedded System kann typischerweise nicht als Rechner genutzt werden.

Die zentrale Einheit bildet ein **Mikroprozessor** *(microprocessor, microcontroller, embedded microcontroller, network processor, digital signal processor)* und besteht aus CPU, RAM, ROM, I/O-Ports und Timern *(ROMable* oder *scalable)*.

- **Prozessrechner** ***(single board computer)*** : Roboter, Flugzeug, Computertomograph, Produktionszelle. Programmieren ist möglich.
- **Netzwerk** ***(network)*** : Fabrik, Signalanlage, Ampelsteuerung, Buchungssystem. Besteht aus vielen Rechnern unterschiedlicher Größenordnung, die Daten austauschen können.

Um die Flexibilität all dieser Systeme zu erhöhen, wird eine Tendenz der Reduktion der Hardware zu Gunsten der Software beobachtet ***(Tradeoff HW − SW)***. Das erhöht die Komplexität der Software.

Einsatzbeispiele von Automatisierungssystemen

Computers Everywhere: Diese Vision sieht den Einsatz von Computern überall. Man wird sie nicht nur in technischen Geräten, sondern auch in Wohnungen, in der Kleidung, im menschlichen Körper finden.

Steuerung von Walzwerken: Der Prozessrechner misst die Stärke, die Temperatur und die Geschwindigkeit des Walzgutes und überwacht dessen jeweilige Lage. Er optimiert die Einstellung der Walzen sowie der Walzgeschwindigkeit in Abhängigkeit von Temperatur, Anforderungen an die Qualität und Walzgutstärke. Hinzu kommt die Störungsüberwachung der ganzen Anlage.

Automatisierung im Zementwerk: Der Prozessrechner wird eingesetzt, um ein optimales Mischungsverhältnis der einzelnen Komponenten des Zements zu erreichen. Aufgrund von Analysen des Zementgemischs kann der Zufluss der einzelnen Komponenten gesteuert werden.

Chemische Verfahrenstechnik: Prozessrechner dienen hier zur Überwachung der Produktionsanlagen sowie zur Optimierung des Prozess-Zustands. Oft werden Verfahren der direkten digitalen Regelung (DDC) eingesetzt, da es sich häufig um sehr viele, langsame Regelstrecken mit allerdings recht komplexem Regelverhalten handelt (Große Regelstreckenzeitkonstanten, große Totzeiten).

Steuerung von Hochregallagern: Steuerung und Überwachung der Förderzeuge. Daneben müssen die Wege für das Ein- und Auslagern der Waren optimiert werden. Verwaltung des Lagerinhalts in einer Datenbank.

Steuerung von Textilmaschinen: Textilmaschinen gehören zu den frühesten automatisch gesteuerten Maschinen. Moderne Textilmaschinen, etwa Rundstrickmaschinen, werden heute

mit Prozessrechnern gesteuert, wobei bereits eine direkte Umsetzung von Mustervorlagen in gestrickte Textilwaren möglich wird (optisches Abtasten des Musters, Umsetzen in Maschinenanweisungen, Steuerung der Maschine).

Fertigungstechnik, Roboter: Zellenrechner steuern und überwachen die Fertigung und Montage von Teilen. Roboter werden mit intelligenten Sichtsystemen zur Werkstückanalyse ausgerüstet, auf deren Resultaten die Bahn- und Greifersteuerung beruht.

Betriebsdatenerfassung: Einzelne Güter werden durch die gesamte Produktionsanlage bis zur Fertigstellung verfolgt. Schadhafte Güter können auf ihrem Produktionsweg zurückverfolgt werden und damit auf Schwachstellen innerhalb der Produktionsanlage hinweisen. Statistische Methoden werden hier eingesetzt.

Qualitätskontrolle: Durch automatisierte Überwachungs- oder Prüfeinrichtungen werden Produkte kontrolliert.

Energietechnik: Prozessrechner werden bei der Energieerzeugung, bei der Energieverteilung und bei der optimalen Nutzung der Energie eingesetzt.

Verkehrstechnik: Ampelsteuerungen, Reservierungen, automatische Zuglaufsysteme (Flughafen Atlanta), Eisenbahn-Technik

Automotive Systems: Einsatz von Prozessrechnern in Automobilen

Avionics: Einsatz von Prozessrechnern in Flugzeugen

Raumfahrt: Einsatz von Prozessrechnern in Raumfahrzeugen und Satelliten

Laborautomatisierung: Viele physikalische und chemische Laborgeräte stellen erhebliche (sehr unterschiedliche, extreme) Anforderungen an die Leistung der Datenerfassung und Datenauswertung. Aufgaben sind Detektion von Gipfelwerten (Massenspektrometrie), Integration von Flächen (Gaschromatographie), Eichung, Driftkontrollen und -korrekturen, statistische und graphische Aufbereitung, komplexe Umrechnung gemessener Werte in andere Werte (Entfaltung des Signals mit der Gerätefunktion) und Steuerung der Geräte und Experimente.

Computer-Tomographie: CT war erst durch den Einsatz von Rechnern möglich. Nuklearmedizinische Bilder, Ultraschallbilder, Röntgenbilder.

Verarbeitung von Biosignalen: Elektro-Kardiogramm, Elektro-Enzephalo-gramm, Elektro-Myogramm, Elektro-Retinogramm.

Patienten-Intensivüberwachung: Zur Erhöhung der Zuverlässigkeit werden Mehrrechnersysteme eingesetzt.

Militärische Anwendungen: Strategische Entscheidungen (Künstliche Intelligenz), Projektilsteuerung, Aufklärung.

Nach DIN 66201 ist ein Prozessrechensystem eine Funktionseinheit zur prozessgekoppelten Verarbeitung von Prozessdaten, nämlich zur Durchführung boolescher, arithmetischer, vergleichender, umformender, übertragender und speichernder Operationen.

Das Wesentliche daran ist die Kopplung zwischen Rechner und einem technischen Prozess.

Automatisierungssystem:

- **Benutzeroberfläche:** Bedienungsfreundliche Schnittstelle zwischen Mensch und Prozess, d.h. der Prozessbediener muss keine Programmierkenntnisse und im Extremfall keine Fachkenntnisse haben.
- **Sensorik:** Prozessdatenerfassung durch sensorische Glieder. Ein physikalisch-chemischer Effekt (meist Spannung oder Strom) wird in eine Bitfolge umgewandelt.
- **Intelligenz:** Datenverarbeitung. Prozessdaten werden in andere Datenformen umgerechnet. Daten werden evtl. reduziert. Intelligente Regelung: Aktionen hängen prinzipiell von allen Daten ab (Modellierung, Expertensysteme).
- **Persistenz:** Datenverwaltung. Daten werden in einer Datenbank gespeichert. Datendokumentation und Berichterstellung.
- **Aktorik:** Prozesssteuerung. Über aktorische Glieder wird in den Prozess programmgesteuert eingegriffen. Eine Bitfolge wird in einen physikalisch-chemischen Effekt (meist motorische Bewegung) umgewandelt.
- **Vernetzung:** Kommunikation mit anderen Prozessen, Rechnersystemen, Produktionsebenen.

Prozessrechner: Das Herz eines Automatisierungssystems ist der Prozessrechner und seine Software. Charakteristisch für einen Prozessrechner und sein Betriebssystem ist **Realzeitfähigkeit**. Er muss in der Lage sein, zu bestimmten, festgelegten Zeiten Daten aufzunehmen und auszugeben und auf Prozesszustände zu reagieren ***(specified time constraints)***.

Der Prozessor ist umgeben von ungewöhnlicher Peripherie (Sensoren, Aktoren, Kommunikationsnetzen). Die "Dringlichkeit" von Ereignissen spielt eine besondere Rolle.

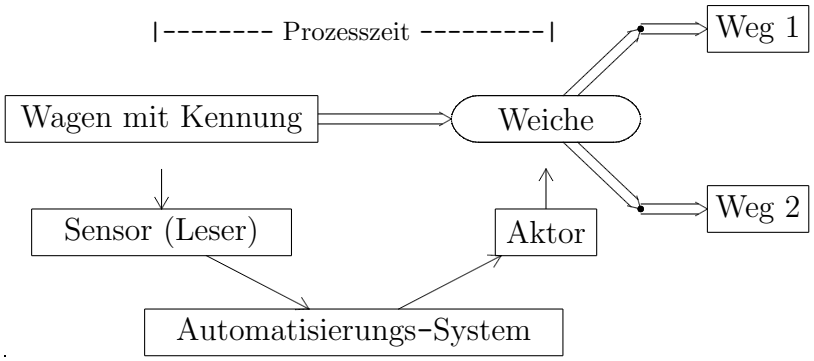
Nach DIN 44300 ist ein Realzeitsystem definiert als der Betrieb eines Rechensystems, bei dem Programme zur Bearbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne (Antwortzeit) verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu bestimmten Zeitpunkten anfallen.

Die für die Aufgaben

- Alarmanalyse (1 μs bis 1 ms),
- Messen (Messinstrument) (1 ns bis mehrere Tage),
- Erfassen (Messdaten vom Messinstrument über I/O-Schnittstelle in Prozessrechner) (1 bis 5 μs),
- Verarbeiten (Berechnung von Steuergrößen),
- Ausgeben (Steuergröße vom Prozessrechner über I/O-Schnittstelle an Aktor) (1 bis 5 μs),
- Steuern (Aktion, Motorische Bewegung) (1 ms bis Sekunden)

benötigte Zeit muss deutlich kleiner sein als die Regelstrecken-Zeitkonstante (Prozesszeitkonstante). Besonders zeitkritisch sind z.B. die Prozesse des zeitgenauen Einspritzens und Zündens bei modernen Kraftfahrzeugmotoren.

Im unten gezeigten Beispiel soll ein Wagen je nach Kennung auf den Weg 1 oder den Weg 2 geschickt werden:



Ferner muss der Prozessrechner multitasking-fähig sein, d.h. er muss in der Lage sein, mehrere solcher Prozesse quasi-parallel bedienen zu können. Ein realer technischer Prozess wird am einfachsten als eine Summe parallel ablaufender Prozesse modelliert. Daher wird das Automatisierungssystem naturgemäß ein **nebenläufiges** ***(concurrent)*** Realzeit-System sein (im Unterschied zu sogenannten **sequentiellen** Systemen).

Im Allgemeinen müssen viele solcher Realzeitbedingungen an unterschiedlichen Orten oder von unterschiedlichen Prozessrechenern gleichzeitig eingehalten werden. Ein Realzeit-System wird also durch drei Merkmale charakterisiert. Es ist:

- nebenläufig *(concurrent)*
- verteilt *(distributed)*
- echtzeitabhängig *(real time dependent)*

Mit der steigenden Leistungsfähigkeit von Prozessrechnern haben sich die Einsatzschwerpunkte von "schnellen Reaktionen" zu gehobenen Aufgaben verschoben:

- Prozessoptimierung
- verbesserte Verfahrensführung, z.B. durch modellgestützte Regelung oder modellgestützte Alarmierung
- Kenngrößenberechnung
- gehobene Auswertung, Protokollierung und Archivierung
- Auswahl, Korrektur und Kommentierung von Daten für die Weiterleitung an die nächst höhere Ebene in der Produktionshierarchie

- Entwicklung von Herstellungsanleitungen und Rezepturen
- gehobene Produktionssteuerung (PPS)
- Prozessvisualisierung auf Farbbildschirmen
- Integration von Expertensystem-Software (XPS)

Zur Abgrenzung gegen den Prozessrechner seien einige andere Arten der digitalen Steuerung definiert:

VPS verbindungsprogrammierte Steuerung, wo das Programm durch eine feste Verdrahtung als Anlagenstruktur verschaltet ist. Solche Systeme sind relativ inflexibel. Einsatz für einfache Überwachungs- und Regelaufgaben. Man unterscheidet

- festprogrammiert: Änderungen sind nur durch Umlöten möglich.
- umprogrammierbar durch Umstecken von Leitungen, Auswechseln von Diodenmatrizen, Austausch von Baugruppen
- Durch den Einsatz moderner Gate-Arrays sind diese Steuerungen wesentlich flexibler programmierbar.

Komponenten: Relais, Elektronikkarten mit verknüpften Gattern, Gate-Arrays, GAL, PAL
Programmspeicher: Verdrahtung, Wahlschalter
Abarbeitung: parallel

An dieser Stelle sind auch CPLDs, FPGAs und DSPs zu erwähnen, mit denen wir uns nicht beschäftigen werden, sofern darauf nicht ein Realzeitbetriebssystem mit einer höheren Programmiersprache (C/C++, Java, `C#`) läuft.

SPS speicherprogrammierte Steuerung, wo das Programm in einem Speicher abgelegt ist ***(PLC programmable logic controller)***. Diese Systeme sind wesentlich flexibler als VPS und anspruchsvoller programmierbar.

Man unterscheidet

- austauschprogrammierbar: Nur-Lese-Speicher (ROM, Festwertspeicher). Programmänderung ist nur durch mechanischen Austausch von Speicherbauteilen möglich.
- frei programmierbar: Schreib-Lese-Speicher (RAM). Inhalt kann ohne mechanischen Eingriff geändert werden.

Komponenten: IC, μP
Programmspeicher: RAM (Random Access Memory, elektrisch-l, elektrisch-p, flüchtig), ROM (Read Only Memory, bei Herstellung-p, nicht flüchtig), PROM (Programmable ROM, elektrisch-p, nicht flüchtig), EPROM (Erasable PROM, UV-Licht-l, elektrisch-p, nicht flüchtig), RPROM (Reprogrammable ROM), EEROM (Electrically Erasable ROM, elektrisch-l, elektrisch-p, nicht flüchtig), EAROM (Electrically Alterable ROM) (l: löschbar, p: programmierbar)

Abarbeitung: zyklisch, nur quasi-parallel, 1000 Anweisungen in 1 bis $10ms$. In einem Zyklus wird ein Prozessabbild (PAE) verarbeitet.

Eine moderne SPS hat z.B.

- 496 Eingänge
- 240 Ausgänge
- 2117 Merker
- 192 Zeiten $1ms$ bis 999 Tage
- 18 Befehle
- 32000 Speicherplätze für Befehle
- Verarbeitungsgeschwindigkeit: $1\mu s$ pro Befehl

Zur Programmierung ist ein relativ geringes elektronisches Basiswissen erforderlich. Schnelle Anpassungen und Veränderungen sind im Feld durch den Automatisierungstechniker möglich.

Programmiert wird entweder in AWL (Anweisungsliste nach DIN 19239), KOP (Kontaktplan, Elektriker) oder FUP (Funktionsplan, Digitaltechnik).

Die SPS war früher ein typischer 1-Bit-Rechner ohne Verarbeitung von Analogwerten. Heutzutage gibt es komfortable Prozessleitsysteme auf SPS-Basis, die sich bezüglich Rechnerhardware kaum von Prozessrechnern unterscheiden und sich möglicherweise nur nach außen wie eine SPS verhalten, um den Programmiergewohnheiten von Automatisierungstechnikern gerecht zu werden.

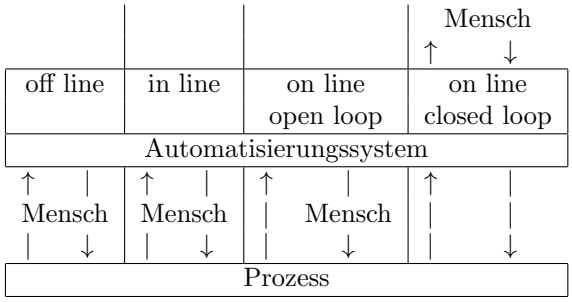
Kopplung Prozess und Automatisierungssystem – Begriffe:

off line (ohne Zeitbindung, indirekt) Mensch ist Verbindung zwischen Prozess und Rechner (Bänder, Lochstreifen, Bediener liest Werte am Instrument ab, Handnotizen werden hin- und hergetragen)

in line (mit Zeitbindung, indirekt) Mensch ist Verbindung zwischen Prozess und Rechner. Es muss auf zeitgenaue Eingabe geachtet werden.

on line, open loop (mit Zeitbindung, direkt) Mensch ist nur noch für eine Richtung die Verbindung zwischen Prozess und Rechner.

on line, closed loop (mit Zeitbindung, direkt) Prozess und Rechner sind ohne Mensch als Glied miteinander gekoppelt.



Übung:

Diskutieren Sie die Verhältnisse bei einer Motorsteuerung.

Kapitel 2

Prozesslehre

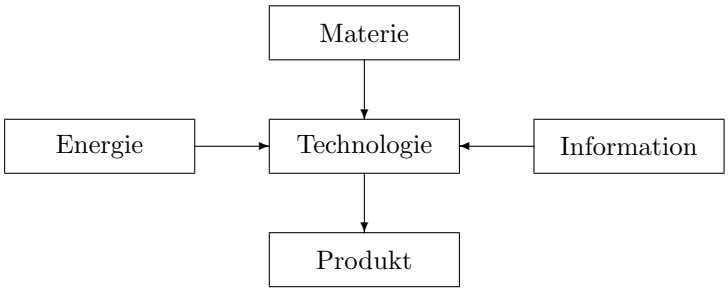
Ein **Prozess** ist

- jeder Vorgang in der Zeit.
- ein Ablauf, Verlauf.
- der Übergang eines Systems von einem Zustand in einen anderen.
- die zeitliche Veränderung eines Systems.
- das dynamische Verhalten eines Systems, bei dem eine **Umformung**, **Transport** und/oder **Speicherung** von **Materie**, **Energie** und **Information** stattfindet (DIN 66 201, DIN 19 222).

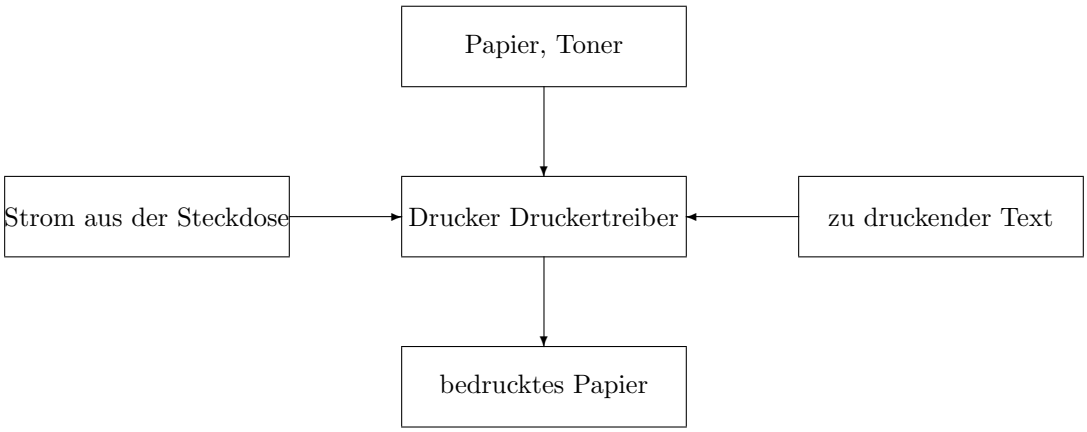
Ein **technischer Prozess** (DIN 66 201) ist ein Prozess, dessen physikalische Größen mit technischen Mitteln erfasst und beeinflusst werden.

Zur Messwerterfassung dienen **Sensoren** und zur Beeinflussung des Prozesses dienen **Aktoren**.

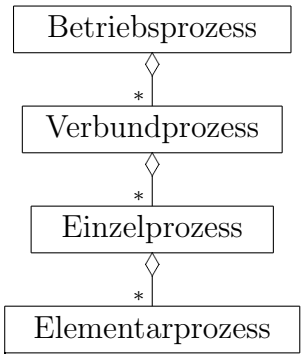
In der Prozessautomatisierung haben wir es i.A. mit technischen Prozessen zu tun, mit denen gewünschte Ergebnisgrößen (Produkte) erzielt werden sollen. Der Prozess beruht dann im wesentlichen auf einer **Technologie** (**Rezeptur, Herstellungsanleitung**).



Als Beispiel schauen wir uns einen Drucker an.

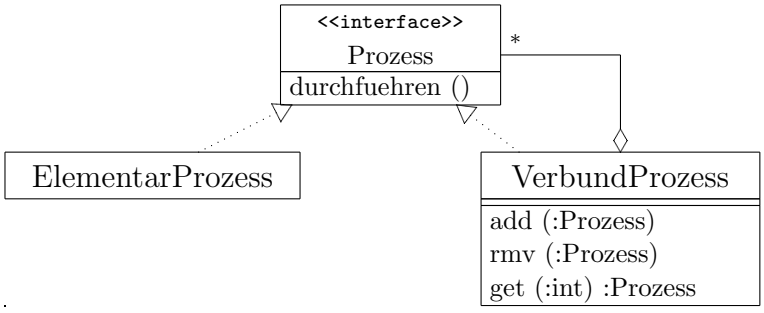


Die kleinste Teilaufgabe in einem Prozess wird als **Elementarprozess** bezeichnet. Mehrere Elementarprozesse in einem geschlossenen Wirkungskreis bilden einen **Einzelprozess**. Viele Einzelprozesse ergeben einen **Verbundprozess**, der eine Aufgabe bearbeitet. Schließlich können verschiedene Verbundprozesse zu einem **Betriebsprozess** zusammengefasst werden.



Beispiel: In einer Großküche ist das Heizen eines Kessels ein Elementarprozess. Die Erfassung der Temperatur ist ebenfalls ein Elementarprozess. Beide zusammen und eventuell weitere Elementarprozesse bilden einen Einzelprozess. Dessen Aufgabe ist es, die Temperatur des Kessels auf einem bestimmten Wert zu halten. Die verschiedenen Kessel und Küchenmaschinen, die benötigt werden, um ein Gericht herzustellen bilden einen Verbundprozess. Die Planung, Einkauf der Lebensmittel, Zubereitung und Verteilung der Gerichte einer Großküche bilden einen Betriebsprozess.

Allgemeiner anwendbar ist eine Composite-Pattern-Struktur, indem wir unter einem Verbundprozess jede Art von Prozess verstehen, der aus Elementarprozessen und/oder anderen Verbundprozessen besteht.



Die Schnittstelle `Prozess` kann natürlich auch eine abstrakte Klasse sein, die dann Datenelemente und eine Prozess-Durchführlogik enthalten könnte.

2.1 Formale Beschreibung von Prozessen

Ein Prozess ist ein **unendlicher Mealy-Automat** mit folgender algebraischer Struktur:

$$P = (U, X, Y, f, g)$$

Die drei Mengen und zwei Abbildungen sind:

U**:** Menge der Eingabewerte, Wertevorrat der Eingangsgrößen, Wertevorrat der Eingangssignale, Eingangsalphabet

X**:** Menge der Zustände (und ihrer zeitlichen Ableitungen) oder Zustandswerte, Wertevorrat der Zustandsgrößen, Wertevorrat der Zustandssignale, Zustandsalphabet

Y**:** Menge der Ausgabewerte, Wertevorrat der Ausgangsgrößen, Wertevorrat der Ausgangssignale, Ausgangsalphabet

$f : X \times U \to X$**:** Überführungsfunktion

$g : X \times U \to Y$**:** Ausgabefunktion

Eingangsgrößen: $u = (u_1 \dots u_m) \in U$

- steuerbare Eingangsgrößen (Stell- oder Steuergrößen)
- nicht steuerbare Eingangsgrößen (messbare und nicht messbare Störgrößen)

Zustandsgrößen: $x = (x_1 \dots x_m) \in X$

Wichtige Zwischengrößen, die zum Teil nicht messbar sind. Ihre rechnerische Ermittlung hat in der Prozessdatenverarbeitung eine große Bedeutung. Sie beschreiben den Zustand des Systems eindeutig. Die Zeit t ist speziell eine Komponente von x. (Philosophische Randbemerkung: Die Zeit ist nur dann eine Zustandsgröße des Prozesses, wenn wir eine Uhr dazustellen.) Außerdem ist es ganz nützlich, auch die zeitlichen Ableitungen von x, nämlich

$$\dot{x} = (\dot{x}_1 \dots \dot{x}_m) \in X$$

als Element von X zu sehen.

Ausgangsgrößen: $y = (y_1 \dots y_m) \in Y$
Gesteuerte Größen (Ergebnisgrößen).

Wenn die Mengen U, X, Y endlich sind, dann ist der Prozess ein **endlicher Mealy-Automat**, ansonsten ein unendlicher Mealy-Automat.

Es gibt drei wichtige Standardformen für die Prozessmodellierung:

1. diskrete, diskontinuierliche Systeme:

   $x^{(k+1)} = f(x^{(k)}, u^{(k)})$ Überführungsfunktion
   $y^{(k)} = g(x^{(k)}, u^{(k)})$ Ausgabefunktion
   $k = \text{trunc}(\frac{t}{T_A})$ Takt- oder Abtastzeitpunkte mit tastperiode T_A

2. analoge, kontinuierliche Systeme:

   $\dot{x} = f(x, u)$ Überführungsfunktion
   $y = g(x, u)$ Ausgabefunktion

   Bemerkung:

   $\frac{dx}{dt} = f(x, u)$
   $x(t + dt) = x(t) + dx = f(x(t), u(t))\, dt + x(t) = \varphi(x(t), u(t))$

3. analoge, kontinuierliche lineare Systeme:

   $\dot{x}(t) = \frac{dx}{dt} = A\, x(t) + B\, u(t)$ Überführungsfunktion
   $y(t) = C\, x(t) + D\, u(t)$ Ausgabefunktion

   Wenn u, x, y Vektoren der Länge m, n, r sind, dann haben wir folgende Matrizen:

   A: $n \times n$-Systemmatrix, Dynamik des ungestörten Systems
   B: $n \times m$-Steuermatrix (Eingangsmatrix)
   C: $r \times n$-Beobachtungsmatrix (Ausgangsmatrix)
   D: $r \times m$-Durchgangsmatrix, bestimmt die unmittelbare Wirkung des Eingangsvektors auf den Ausgangsvektor, meistens Null

2.2 Prozessarten

Eine Klassifizierung technischer Prozesse ist sinnvoll, da man artverwandte Prozesse häufig mit ähnlichen Verfahren beschreiben und automatisieren kann.

2.2.1 Art des Mediums oder des Verarbeitungsmaterials

Einteilung der Prozesse nach der Art des Mediums, das transportiert oder umgeformt wird:

1. **Materialprozesse:**

   (a) **Fertigungsprozesse:** Herstellung von Teilen exakt definierter geometrischer Form, Festigkeit und Oberflächenbeschaffenheit. Die Formgebung steht im Vordergrund. Beispiele: Prozesse der Fertigungsindustrien, wie Maschinenbau, Feinmechanik, Optik, Elektrotechnik.

   (b) **Verarbeitungsprozesse:** Verarbeitung vorwiegend nichtmetallischer Werkstoffe. Die Stoffveränderungen haben keine primäre Bedeutung. Beispiele: Prozesse der verarbeitenden Industrie für Papier, Textil, Leder, Holz, Druck.

   (c) **Verfahrensprozesse:** Stoffänderungen und Stoffumwandlungen. Die geometrische Form und Formänderungen haben keine primäre Bedeutung. Beispiel: Prozesse der chemischen Industrie, Metallurgie, Biotechnologie.

2. **Energieprozesse:** Umwandlung und Weiterleitung aller Energieformen zum Zwecke der technischen Nutzung. Beispiel: Prozesse der Energieindustrie, wie Wärme- und Kernenergieerzeugung, Verteilung und Übertragung elektrischer Energie.

3. **Informationsprozesse:** Sammeln, Ordnen, Verdichten und Verteilen von Informationen. Beispiel: Prozesse der Mess- und Prüftechnik, Nachrichtenwesen, Funk und Fernsehen, Datenverarbeitung, Nachrichtendienste.

2.2.2 Technologisches Verfahren oder Verarbeitungsart

Einteilung der Prozesse nach dem technologischen Verfahren:

1. **Bearbeitungs-** und **Erzeugungsprozesse** der Fertigungs-, Verarbeitungs- und Verfahrenstechnik: Zustandsänderungen physikalischer und chemischer Größen, wie Geometrie, Stoffzusammensetzung, Stoffeigenschaften. Beispiel: Gießen, Drehen, Fräsen, Spritzgießen, Destillieren, Polymerisieren, Vulkanisieren, Raffinerie, chemischer Reaktor, Walzwerk, Rechnen in EDV.

2. **Verteilungsprozesse:** räumliche und zeitliche Verteilung von Stoffen, Energien und Informationen. Beispiele: Briefzustellung, Werkstücktransport, Montagestraße, Pipelinetransport, elektrische Energieversorgungsnetze, Telefonvermittlung, Berichtswesen, Netze in EDV.

3. **Ordnungs-** und **Aufbewahrungsprozesse:** räumliche und zeitliche Ordnung beliebiger Mengen. Beispiele: sortenreine Schüttgutbunkerung, Lagerung, Steuerung von Hochregallagern, Fahrplansteuerungen, Terminisierung und Lenkung von Fertigungsaufträgen, Datenbanken in EDV.

2.2.3 Form des Stoffes oder Zustand des Verarbeitungsmaterials

Einteilung der Prozesse nach der Form des im Prozess beeinflussten Stoffes:

1. **Fließgutprozesse (Mengenprozesse):** Zuführung, Dosierung oder Bemessung eines stetigen Material- bzw. Stoffstroms von Feststoffen, Flüssigkeiten und/oder Gasen. Es werden Fluide, Massen, Strang- und Schüttgüter verarbeitet. Beispiele: Strangguss, Drahtziehen, Bandwalzwerk, Materialtransport über Förderbänder, Bearbeiten von Textil-, Plast- oder Papierbahnen, Bändern und Fäden.
2. **Stückgutprozesse:** Verarbeitung oder Transport von identifizierbaren Einzelstücken. Beispiel: Block-Brammenwalzwerke, Teiletransport und - lagerung, Montageprozesse, Verpackungsvorgänge.

2.2.4 Zeitlicher Ablauf

Einteilung der Prozesse nach dem zeitlichen Ablauf:

1. **Kontinuierliche Prozesse (Fließprozesse):** ständiger Zu- und Abfluss von Material, Energie oder Information. Um den Fließprozess zu stabilisieren, muss ein voreingestellter (möglichst optimaler) Prozesszustand durch eine Steuerung aufrechterhalten werden. Die dominierende Aufgabe bei kontinuierlichen Prozessen ist daher das Regeln. Das System verhält sich im Arbeitspunkt meistens linear. Beispiel: Kraftwerke, Betrieb von Versorgungsnetzen für Elektrizität, Gas, Wasser und Öl; viele Prozesse der Verarbeitungs- und Verfahrenstechnik (Papier, Plast, Textil, Zement, Chemie, Raffinerie).
2. **Diskontinuierliche Prozesse:**
   (a) **Chargenprozesse:** Der Einsatz von Material- bzw. Stoffmengen (ohne wesentliche Geometrie, vgl. Fließgutprozesse) erfolgt zu unterschiedlichen, diskreten Zeitpunkten (portionsweise, hoch nichtlinearer Arbeitspunkt); die Verarbeitung selbst erfolgt in einem kontinuierlichen Prozess. Chargenprozesse bestehen z.B. aus den
   Phasen: Beladen, Aufheizen, Reaktion, Abkühlen, Entladen, Reinigen.
   Die dominierende Leitaufgabe bei Chargenprozessen ist das Steuern in Form von Verknüpfungs- und **Ablaufsteuerungen**.
   Beispiele: Hochofenprozess, Plaststoff-Polymerisation, viele chemische und biotechnische Prozesse.
   (b) **Stückprozesse (Folgeprozesse):** Verarbeitung und Transport von Folgen identifizierbarer Einzelmengen (vgl. Stückgutprozesse) von Stoffen, Materialien oder Informationen zu unterschiedlichen, diskreten Zeitpunkten. Beispiel: Rangiervorgänge von Güterwagen, Ein- und Auslagern in Hochregallagern, Übertragung einzelner Datenblöcke über Signalleitungen, Geräteprüfung, Folgesteuerungen (Aufzüge, Zentrifugen), Fertigungsprozesse.

2.3 Prozesszustände

Der zeitliche Verlauf ist für die Automatisierung eine wichtige Charakterisierung eines Prozesses. Der Prozess durchläuft eine Reihe von Prozesszuständen. Folgende Zustände oder Operations-

moden dürften für die meisten Prozesse typisch sein:

- Anfahren
- Betrieb
- Absetzen, Abfahren
- Wartung
- Gefahren- oder Fehlerzustände

Hieran ist wichtig, dass es außer "Betrieb" noch vier andere Prozesszustände gibt, die auch berücksichtigt werden müssen, aber zunächst gern vernachlässigt werden.

2.3.1 Stationäre Prozesse

Das System oder der Prozess ist in einem **stationären** Zustand, wenn Ein- und Ausgänge und die Systemvariablen keine zeitliche oder höchstens periodische Veränderung zeigen (z.B. die Bewegung eines Motors).

2.3.2 Zyklische Prozesse

Der technische Prozess und das Automatisierungssystem interagieren in regelmäßigen (konstanten) Zeitabständen, wobei entweder der Prozess durch Alarme oder das Automatisierungssystem den Anstoß dazu gibt.

Bei nicht-zyklischen Prozessen erfolgt die Interaktion aufgrund zufälliger Ereignisse.

2.4 Allgemeine Automatisierungsaufgaben

- Steuerung und Überwachung von verfahrenstechnischen Prozessen: Raffinerie, Papiermühle, Schokoladenfabrik
- Datenerfassung: Datenerfassung bei einer chemischen Reaktion, Pipelinedaten,
- Qualitätskontrolle
- Kommunikation: Steuerung von Satelliten, Telefonsysteme
- Transaktionsorientierte Prozesse: Reservierungssysteme, Bankbuchungen, Aktienmarkt
- Flugsimulation und -kontrolle: Autopilot, Raketensteuerung, Fahrsimulator
- Fabrikautomatisierung: Materialverfolgung, Teileproduktion, Bandsteuerung, Robotik
- Transport: Zuglaufsystem, Ampelsysteme, Luftverkehrssteuerung
- Graphik und Mustererkennung: Bildverarbeitung, Videospiele, CAD-Solid.Modelling
- Detektionssysteme: Radarsysteme, Alarmsysteme
- Strategische (wirtschaftliche und militärische) Systeme

2.5 Einzelne Automatisierungsaufgaben

Folgende Begriffe sind DIN-Begriffe und sollten daher hiermit bekannt gemacht werden.

- **Datenerfassung:** Durch Messen, Zählen oder Dateneingabe werden analoge oder digitale Daten gewonnen.
- **Messen** ist der experimentelle Vorgang, durch den ein spezieller Wert einer physikalischen Größe als Vielfaches einer Einheit oder eines Bezugswertes ermittelt wird. Das Auswerten von Messwerten (z.B. Korrigieren, Linearisieren) bis zum Messergebnis gehört zum Begriff Messen.
- **Zählen** ist das Ermitteln der Anzahl von jeweils in bestimmter Hinsicht gleichartigen Elementen oder Ereignissen, die bei dem zu untersuchenden Vorgang in Erscheinung treten.
- **Dateneingabe** durch die Hand, Disketten, Bänder oder Netzverbindungen.
- **Datenverarbeitung:** Analoge oder digitale Daten werden mit Hilfe eines Programms in andere Daten umgeformt.
- **Stellen** ist das Verändern von Material-, Energie- oder Informationsflüssen.
- **Regeln** ist ein Vorgang, bei dem fortlaufend eine Größe – die Regelgröße – erfasst, mit einer anderen Größe – der Führungsgröße – verglichen und abhängig vom Ergebnis dieses Vergleichs im Sinne einer Angleichung an die Führungsgröße beeinflusst wird (geschlossener Wirkungsablauf).
- **Steuern** ist der Vorgang in einem System, bei dem eine oder mehrere Größen als Eingangsgrößen andere Größen als Ausgangsgrößen aufgrund der dem System eigentümlichen Gesetzmäßigkeiten beeinflussen (offener Wirkungsablauf).
- **Melden** ist das Weitergeben zu überwachender binärer Größen und das Anzeigen in besonders auffälliger Form.
- **Anzeigen** ist das wahrnehmbare Darstellen von Größen und Schaltzuständen.
- Unter **Eingreifen (Bedienen)** versteht man das Einwirken des Menschen auf die Leiteinrichtung oder auf die Stellglieder des Prozesses.
- Unter **Überwachen** versteht man das Überprüfen ausgewählter Größen auf Einhaltung vorgegebener Werte, Wertbereiche oder Schaltzustände.
- Beim **Schützen** wird mit den Mitteln der Leittechnik aufgrund von Überwachungsvorgängen in der Weise auf den Prozess eingewirkt, dass er keinen den Menschen gefährdenden, die Anlagen, das Produkt oder die Unwelt schädigenden Zustand annimmt oder die Anlage vor schädigenden Auswirkungen von Störungen bewahrt.
- Beim **Auswerten** werden aus erfassten Größen durch Berechnen oder auch Sortieren Kenngrößen des Prozesses ermittelt.
- **Aufzeichnen (Registrieren, Protokollieren)** ist das Festhalten und Darstellen von Größen in einer für den Menschen leicht auswertbaren Form.

- **Datenübertragen** ist ein Vorgang, durch den Daten zwischen voneinander getrennten Einrichtungen transportiert werden.

- **Datenausgeben** ist ein Vorgang, bei dem analoge oder digitale Daten aufgezeichnet, angezeigt oder an den Prozess gegeben werden.

- Unter **Optimieren** versteht man Maßnahmen zur Erzeugung einer solchen Wirkungsweise des Systems, dass unter den gegebenen Nebenbedingungen und Beschränkungen das **Gütekriterium** einen entweder möglichst großen oder möglichst kleinen Wert annimmt. Optimierung heißt **statisch**, wenn das Gütekriterium nur zeitlich konstante Zustände des Systems bewertet (stationäre Zustände), **dynamisch**, wenn das Gütekriterium den Übergang von einem Anfangszustand in einen Endzustand bewertet (Anfahren, Abfahren, Änderung des Lastzustands). Gütekriterium ist eine Bewertungsvorschrift (Kosten, Führungsverhalten, Störverhalten, Wirkungsgrad, Durchsatz).

2.6 Struktur von Automatisierungssystemen

Da das eigentliche Ziel die Vollautomatisierung der ganzen Produktion ist, wollen wir hier alle Ebenen betrachten. Die Bezeichnung der Ebenen ist abhängig von der jeweiligen Branche.

Ebene (BASF)	Ebene (Siemens)	Funktionen der Ebene
Produktionsleitebene	Betriebsleitebene CIM	Produktionsplanung Beschaffung von Einsatzprodukten (Aufträgen) Bestandsführung Terminüberwachung Langzeitsicherung von Daten Mengenabrechnungen Kostenanalysen Darstellungs- Protokollier- und Bedienfunktionen
Betriebsleitebene	Produktionsleitebene CIM / PLS	Bilanzrechnungen Ausbeute-, Rationalitätsberechnungen Langzeitspeicherung von Daten Statistische Auswertungen Qualitätskontrolle Überwachen von Wartungsintervallen Verwaltung von Rezepturen und Herstellungsanleitungen Planung des Personaleinsatzes Anzeige-, Bedien-, Melde- und Protokollierfunktionen
Prozessleitebene	–	
–	Prozessführungsebene Gehobene Leitfunktionen und bedienernahe Funktionen PLS / FLS	Optimieren Störungsbearbeitung Erweiterte Protokollierung Bearbeitung von Rezepturen und Herstellungsanleitungen Datensammeln
–	Prozesssteuerungsebene Grundfunktionen und prozessnahe Funktionen PLS / FLS	Regeln Steuern Zählen Überwachen Auswerten Aufzeichnen Protokollieren Schützen Eingreifen Energieversorgung Datenerfassen, -eingeben, -ausgeben, -übertragen, -verarbeiten
Feldebene	Prozessebene PLT / SPS / FLT	Messen Zählen Stellen Melden Anzeigen Bedienen Verteilen von Energie Datenerfassen, -ausgeben

CIM: Computer Integrated Manufacturing
PLS: Prozessleitsystem
FLS: Fertigungsleitsystem
FLT: Fertigungsleittechnik
PLT: Prozessleittechnik
SPS: Speicherprogrammierbare Steuerung

Begriff **Strukturierte Automation**: hierarchische Struktur der Automationsaufgaben.

Prinzipien der Gliederung in Ebenen:
Alle Funktionen, die untereinander einen intensiven Datenaustausch erfordern, sind in einer Ebene zusammengefasst. Damit werden die Ebenen weitgehend autark; der Datenaustausch zwischen den Ebenen wird minimiert und zeitunkritisch.

Diejenigen Funktionen, für deren technische Realisierung die gleiche Gerätetechnik und die gleichen Programme verwendet werden können, sind in einer Ebene zusammengefasst.

In Wirklichkeit gibt es meistens mehr Ebenen als oben dargestellt wurde. Das Prinzip jeder einzelnen Ebene sollte sein, möglichst viel Routinearbeit auf eine untere Ebene zu verschieben (Delegieren). Damit ergibt sich die Bildung von Spezialisten in unteren Ebenen. (Wenn wir den Begriff "Ebene" durch den Begriff "Objekt" ersetzen, dann folgen wir hier einem wichtigen Prinzip objekt-orientierter Software-Entwicklung.)

Wir werden uns im Folgenden im wesentlichen mit den beiden unteren Ebenen beschäftigen, da dort die Automatisierung schon am weitesten fortgeschritten ist. Wir werden uns mit dem beschäftigen, was in den Bereich der Realzeit-Datenverarbeitung fällt.

Kapitel 3

Zustandsgraphen

Um die oft schwer verständlichen zeitlichen Abfolgen zu verstehen, entwickeln wir das **dynamisches Modell** ***(dynamic model)*** als **Zustandsgraphen (*statechart diagram,* Zustandsmaschine, Automat, Zustands-Übergangs-Automat, Zustands-Diagramm, ASM-Diagramm, *Algorithmic State Machine Diagram, state machine)***.

Für die Analyse und den Entwurf von kleinen Echtzeitsystemen oder Automaten haben sich Zustandsgraphen seit langem bewährt.

Ein Zustandsgraph beschreibt das dynamische Verhalten eines Objekts, d.h. sein zeitliches Verhalten. Das Objekt erkennt **Ereignisse** ***(event)***, die ihm von anderen Objekten geschickt werden, und reagiert darauf, indem es seinen **Zustand** ***(state)*** ändert oder auch beibehält.

Prinzipiell ist für jedes Objekt ein eigener Zustandsgraph zu erstellen. Die Menge der Zustandsgraphen bildet das dynamische Modell.

Ereignisse beschreiben Zeitpunkte, Zustände beschreiben Zeitintervalle zwischen zwei Ereignissen. Ein Zustand hat eine Dauer und beschreibt oft eine kontinuierliche Aktivität des Objekts. Ereignisse und Zustände sind *dual* zueinander.

3.1 Zustände

Ein Zustand ist eine Abstraktion der aktuellen Werte und Beziehungen eines Objekts. Die aktuellen Werte der Attribute eines Objekts bestimmen den Zustand eines Objekts, allerdings möglicherweise nicht vollständig oder nicht eindeutig, da der Zustand von der Geschichte des Objekts abhängen kann. (Die Geschichte ***(history)*** eines Objekts wird oft nicht durch seine Datenelemente verwaltet.) Ein Zustand kann entweder durch einen Wertesatz oder durch eine Menge von Wertesätzen oder Wertebereiche charakterisiert werden. Ein Zustand spezifiziert qualitativ die Antwort des Objekts auf ein ihm gesendetes Ereignis. Die Antwort kann quantitativ vom Wertesatz abhängen.

Ein Zustand ist ein Objekt einer bestimmten Klasse.

Die Antwort des Objekts kann eine Aktion, Aktivität und/oder eine Zustandsänderung sein.

Bei der Definition von Zuständen eines Objekts werden – der Einfachheit halber – Attribute ignoriert, die insofern keinen Einfluss auf das Verhalten des Objekts haben, als sie die Zustandsfolge nicht qualitativ beeinflussen.

Die **Granularität** von Zuständen und Ereignissen hängt ab vom Abstraktionsniveau. Z.B. kann auf einem hohen Abstraktionsniveau ein Flug von Stuttgart nach München als ein Zustand mit zwei Ereignissen (Start und Landung) charakterisiert werden, obwohl der Flug verschiedenste Zustände (Starten, Steigflug, Flug auf verschiedenen geographischen Abschnitten, Sinkflug, Landen) durchläuft.

Ein Zustand führt **Aktivitäten** ***(activity)*** und **Aktionen** ***(action)*** durch. Aktivitäten haben eine Dauer und können eventuell unterbrochen werden. Aktionen haben – für die Zwecke der Modellierung – keine Dauer und können auch nicht unterbrochen werden.

Zur Beschreibung eines Zustands gehören folgende Informationen:

Name des Zustands. Sollte mindestens innerhalb eines Zustandsgraphen eindeutig und aussagekräftig gewählt werden.

Beschreibung oder Charakterisierung des Zustands.

Definition von drei Arten von Aktionen und Aktivitäten:

entry: Beschreibung von Aktionen oder Aktivitäten, die bei Eintritt in den Zustand auf jeden Fall durchgeführt werden. Diese Phase wird nicht durch empfangene Ereignisse unterbrochen. Es werden alle Ereignisse aufgeführt, deren Information im Zustand verarbeitet wird. Ereignisse, die zwar in den Zustand führen, aber deren Informationsgehalt für den Zustand bedeutungslos ist, sollten nicht berücksichtigt werden. Dadurch wird der Zustand und seine Beschreibung unabhängiger von den Ereignissen oder anderen Zuständen.

Im Diagramm werden diese Aktionen/Aktivitäten durch das Wort ***entry/*** gekennzeichnet.

do: Beschreibung der im Zustand durchgeführten einmaligen oder kontinuierlichen Tätigkeiten (Aktivitäten und auch Aktionen). Diese Tätigkeiten können typischerweise durch Ereignisse unterbrochen werden. Eine wichtige Tätigkeit ist die Erfassung von für den Zustand bestimmten Ereignissen ***(event monitor)***. Alle empfangbaren Ereignisse müssen hier aufgelistet werden. Ereignisse, die der Zustand an andere Objekte sendet, werden typischerweise hier beschrieben.

Im Diagramm werden diese Aktionen/Aktivitäten durch das Wort ***do/*** gekennzeichnet.

exit: Hier werden Tätigkeiten beschrieben, die vor Verlassen des Zustands auf jeden Fall zu erledigen sind. Hier können auch empfangene Ereignisse verarbeitet werden.

Im Diagramm werden diese Aktionen/Aktivitäten durch das Wort ***exit/*** gekennzeichnet.

default: Oft werden diese drei Fälle nicht unterschieden. Dann wird das als **do** interpretiert. Die Aktionen werden sofort bei Betreten des Zustands ausgeführt. Die Aktivitäten sind durch Ereignisse unter- und abbrechbar.

3.2 Ereignisse

Ein Ereignis passiert zu einer bestimmten Zeit. Seine Dauer ist vernachlässigbar bezüglich der betrachteten Zeitskala.

Ein Ereignis kann logisch vor oder nach einem anderen Ereignis stattfinden. Zwischen solchen Ereignissen besteht ein Kausalzusammenhang. Bei Ereignissen, die *nicht* logisch oder kausal voneinander abhängen, oder die zeitlich ohne Beziehung sind, heißen **nebenläufige (parallele, *concurrent)*** Ereignisse. Insbesondere, wenn die Signallaufzeit zwischen den Orten zweier Ereignisse größer ist als die Zeitdifferenz zwischen den Ereignissen, dann müssen die Ereignisse unabhängig sein, da sie sich nicht beeinflussen können. Bei nebenläufigen Ereignissen werden in der Modellierung keine Annahmen über die Reihenfolge getroffen.

Ein Ereignis ist die "gerichtete" Übertragung von Information von einem Objekt zum anderen. Eine mögliche Antwort ist ein weiteres Ereignis. Alle Objekte existieren in der realen Welt parallel.

Jedes Ereignis ist ein Objekt. Diese Objekte werden nach ihrer Struktur und ihrem Verhalten in Objektklassen gruppiert, wobei auch Vererbungshierarchien gebildet werden können. Ereignisse können einfache Signale sein oder eine Struktur haben, mit der zusätzliche Information übermittelt werden kann. Insbesondere die Zeit, zu der ein Ereignis stattfindet, ist häufig eine implizit oder explizit mitgelieferte Information. Den Begriff "Ereignis" verwenden wir sowohl für Ereignis-Objekte (Instanzen) alsauch für Ereignis-Klassen.

Wenn sich Ereignisse in einer Klassen-Hierarchie strukturieren lassen, dann wird eine Transition auch von Ereignissen vom Typ der Subklassen der Klasse des eigentlich spezifizierten Ereignisses gefeuert, sofern eventuell angegebene Bedingungen erfüllt sind.

Die Beschreibung eines Ereignisses enthält

- seine Datenstruktur,
- sein Verhalten,
- *nicht* aber sendende oder empfangende Objekte bzw. Zustände.

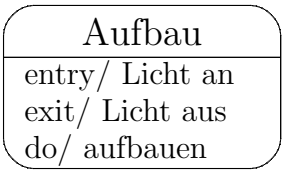
3.3 Notation

Die Automatentheorie ermöglicht eine formale Beweisführung von Eigenschaften wie Vollständigkeit, Vernetztheit und Determinismus von Zustandsgraphen [15] oder [30].

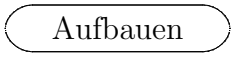
Die UML-Regeln zur Erstellung von Zustandsgraphen orientieren sich an Harel [13]. Das Objekt, für das wir den Graphen erstellen, nennen wir "Automat".

- Jeder stabile Zustand oder Operationsmode des Automaten wird durch ein Oval oder einen Kreis repräsentiert. Er hat i.A. ähnlich einer Klasse zwei Bereiche:
  - Der Name oder eine Nummer des Zustands steht im ersten Bereich. Häufig wird der Name weggelassen, wenn die Aktionen oder Aktivitäten des Zustands hinreichend aussagekräftig sind.

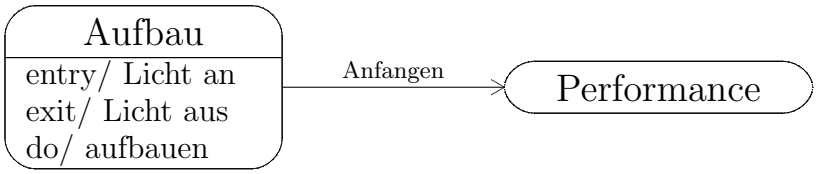
  - Im zweiten Bereich stehen die ***entry***, ***exit*** und ***do*** Prozeduren, d.h. **Aktionen *(action)*** und **Aktivitäten *(activity)*** (abgekürzt Akt., Operationen, Werte von Ausgangssignalen, Outputs, Aktuatorsignale) des Automaten, die in diesem Zustand ausgeführt werden. Die Angabe der Werte von Ausgangssignalen (Aktuatorsignalen) steht für die Aktion des Setzens dieser Werte. Aktionen haben keine Zeitdauer, d.h. sind nicht unterbrechbar. Aktivitäten haben eine Dauer und sind, falls das erlaubt wird, unterbrechbar.



  - Zustände können auch nur einen Bereich haben, entweder für den Namen oder eine aussagekräftige Aktion oder Aktivität.



- **Ereignisse *(events,*** Eingangssignale, Inputs, Sensorsignale, oft logische Verknüpfungen der Eingangssignale), die der Automat empfängt und die den nächsten Zustand bestimmen, werden auf Pfeile geschrieben, die zum nächsten Zustand führen. Pfeile beschreiben also **Zustandsübergänge *(transition)***. Ereignisse können auch durch eine Aktivität des Zustands – etwa als Resultat einer Berechnung – generiert werden.



  Ein Ereignis kann parametrisiert werden, d.h. bei einem Ereignis können Werte in runden Klammern `()` angegeben werden.

  Wenn ein Ereignis nur unter einer **Bedingung *(condition)*** in den nächsten Zustand führt, dann wird diese Bedingung in eckigen Klammern `[]` hinter das Ereignis auf den Pfeil geschrieben.

  Aktionen, die direkt auf ein Ereignis folgen, können durch `/` getrennt direkt hinter das Ereignis auf den Pfeil geschrieben werden. Man kann damit Zustände sparen.

  Zusammenfassend kann auf einem Transitions-Pfeil folgendes stehen:

```
Objekt.Ereignis (Attributwerte) [Bedingungen] / Aktionen
```

  Dabei sollte der Plural möglichst vermieden werden. "Objekt" ist hier das Ereignis sendende Objekt. (Diese Syntax entspricht leider nicht der Syntax gängiger objekt-orientierter Sprachen.)

  Beispiel:

```
Maus.gedrückt (mittlerer Knopf) [wenn im roten Feld] / lösche Feld
```

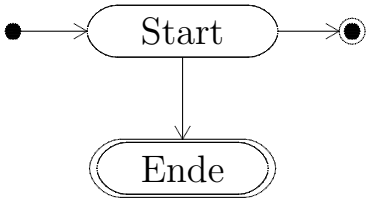
- Unbedingtes Fortschreiten (Pfeil ohne Ereignis) in den nächsten Zustand ist möglich.

- Rückführung in denselben Zustand ist möglich. Ob ein Ereignis **ignoriert** wird oder ob es in denselben Zustand zurückführt, ist i.A. ein Unterschied, da im letzteren Fall der Zustand neu betreten wird, was die Durchführung von `exit`-, `entry`- und `do`-Aktionen/Aktivitäten zur Folge hat. Daher sollten nicht-ignorierte Ereignisse auch modelliert werden. Ignorierte Ereignisse tauchen in der Modellierung normalerweise nicht auf.

- Der **Startzustand** ***(initial state)*** wird durch einen dicken Punkt zum Startzustand dargestellt. Er ist ein Pseudozustand und geht sofort in den eigentlichen Startzustand über. Es kann in einem Zustandsgraphen nur einen Startzustand geben.

- **Terminale** oder **finale** Zustände ***(terminal, final state)*** werden durch Doppelkreis oder -oval dargestellt. Es kann mehrere terminale Zustände geben. Ein terminaler Zustand ist dadurch gekennzeichnet, dass das System nicht mehr auf Ereignisse reagiert. Aus diesen Zuständen führt kein Pfeil heraus.

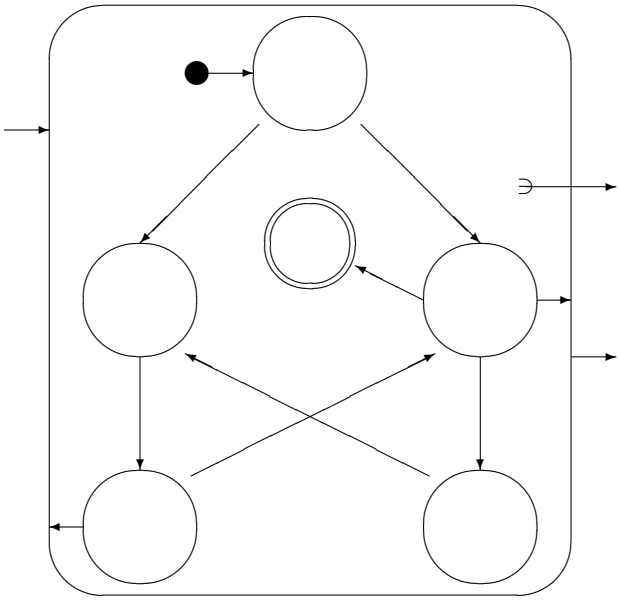
  Es gibt auch einen terminalen Pseudozustand dargestellt durch einen dicken Punkt in einem Kreis.



- Hierarchie von Zustandsgraphen ist möglich: Ein Zustandsgraph X kann eine Aktivität des Zustands $Y.A$ eines übergeordneten Zustandsgraphen Y sein. D.h., wenn im Zustandsgraph Y der Zustand $Y.A$ erreicht wird, dann wird der Startzustand des Zustandsgraphen X – gekennzeichnet durch ($\bullet\!\!\longrightarrow$) – betreten. Die Einbettung von X in $Y.A$ wird dargestellt, indem der Zustandsgraph X durch ein großes Oval, das den Zustand $Y.A$ darstellt, umrahmt wird. Der Zustandsgraph X wird durch Pfeile verlassen, die aus einem oder mehreren Zuständen von X an die Umrahmung führen. Ein terminaler Zustand in X führt auch in den übergeordneten Zustand $Y.A$.

  Pfeile der Art ($\ni\!\!\longrightarrow$), die im übergeordneten Zustand $Y.A$ beginnen und aus dem Zustand herausführen, bedeuten, dass jedes auf diesem Pfeil angegebene Ereignis zur Folge hat, dass der Zustandsgraph X aus jedem seiner Zustände heraus verlassen wird.

  Die Notation $\ni\!\!\longrightarrow$ eignet sich auch, wenn ein Ereignis auf alle Zustände (ausgenommen terminale Zustände) wirkt.



- Parallelität: Mehrere Automaten können gleichzeitig laufen. Das wird dadurch dargestellt, dass die entsprechenden Zustandsgraphen irgendwie nebeneinander oder übereinander, getrennt durch gestrichelte Linien plaziert werden.

  Es bleibt zu klären, wie die Interaktion zwischen parallel laufenden Zustandsgraphen darzustellen ist. Das Senden eines Ereignisses ist eine Aktion, die im sendenden Zustand ausgeführt wird und in einen Text der Art ***send event ... to ...*** eingebettet wird. Dabei kann das ***to ...*** möglicherweise weggelassen werden, da das Ereignis auf einem oder mehreren Übergangspfeilen zu finden ist.

  Um die Herkunft eines Ereignisses zu zeigen, kann der Name des Ereignisses auf dem Übergangs-Pfeil durch den Namen des sendenden Objekts qualifiziert werden ***(object.event)***.

  Eine graphische Darstellung durch Pfeile würde das Diagramm zu unübersichtlich machen.

3.4 Zustandsübergangsmatrix

Die **Zustandsübergangsmatrix** ***(transition matrix)*** ist eine nicht-graphische Darstellung von Zustandsgraphen. In der Matrix werden Zustände $Z_1, Z_2 \cdots Z_n$ gegen Ereignisse $E_1, E_2 \cdots E_m$ aufgetragen. Die Matrixzellen enthalten den Folge-Zustand, d.h. den Zustand, der betreten wird, wenn im Zustand Z_i (Zeilenkopf) das Ereignis E_j (Spaltenkopf) empfangen wird.

	E_1	E_2	$\cdots$	E_m
Z_1	Z_4	Z_6	$\cdots$	Z_1
Z_2	Z_3	Z_2	$\cdots$	Z_7
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
Z_n	Z_8	Z_1	$\cdots$	Z_5

Wenn ein Ereignis ignoriert wird, dann bleibt die Zelle leer.

3.5 Formale Beschreibungssprache

Im ISO-Standard 8807 wurde die formale Beschreibungssprache LOTOS *(Language of Temporal Ordering Specification)* festgelegt. Sie wurde zur Beschreibung von verteilten Systemen, insbesondere von Kommunikationssystemen entworfen. Ein kommerzielles Produkt für die Entwicklung von Zustandsgraphen ist STATEMATE von der Firma i-Logix.

3.6 Strukturierung von Zustandsgraphen

Für Ereignisse können Erweiterungs-Hierarchien (Vererbungs-Hierarchien) gebaut werden.

Die Strukturierung von Zustandsgraphen erfolgt analog zur Strukturierung von Objekten, wo wir hauptsächlich die Begriffe *Erweiterung* und *Aggregation* verwendet haben.

Der Erweiterung von Objekten entspricht die Expansion der Aktivität eines Zustands als eigenen Zustandsgraphen (Hierarchie von Zustandsgraphen).

Der Aggregation entspricht die Aufteilung einer Zustandsaktivität in parallele Aktivitäten.

Bei nebenläufigen Zustandsgraphen ist zu bemerken, dass diese Graphen sich durch das Senden von Ereignissen beeinflussen können, wobei die Übergänge auch von Zuständen anderer Zustandsgraphen abhängen können.

3.7 Entwicklung von Zustandsgraphen

Ein erster Schritt zur Erstellung des dynamischen Modells ist ein **Szenario** ***(scenario)***. Das ist die Auflistung aller Vorgänge in *einer* möglichen Reihenfolge ohne Berücksichtigung der sendenden oder empfangenden Objekte. Bei komplizierten Problemen müssen eventuell mehrere oder besonders lange Szenarien erstellt werden.

Beispiel eines Szenarios an einem Kofferband:

1. Band hält an.
2. Koffer wird in der Mitte des Bandes aufgelegt.
3. Koffer wird links aufgelegt.
4. Band fährt nach rechts.
5. Koffer verlässt rechts das Band.
6. Koffer wird links aufgelegt.
7. Koffer wird in der Mitte vom Band genommen.
8. Koffer wird links aufgelegt.
9. Koffer wird rechts aufgelegt.
10. Koffer verlässt rechts das Band.

11. Koffer verlässt rechts das Band.
12. Koffer verlässt rechts das Band.
13. Band hält an.

Der nächste Schritt ist die Identifikation von Sendern und Empfängern. Das Resultat kann in einem Sequenz-Diagramm dargestellt werden.

Mit einem Szenario und/oder Event-trace können nur einzelne von möglicherweise Tausenden von Ereignisfolgen dargestellt werden. Mit einem Zustandsgraphen dagegen können alle Ereignis- und Zustandsfolgen dargestellt werden.

3.8 Übungen

3.8.1 Kofferband

Erstellen Sie einen Zustandsgraphen für ein Kofferband, das von beiden Seiten benutzbar ist und auf beiden Seiten jeweils genau eine Lichtschranke hat. Die Koffer sollen nicht vom Band geworfen werden. Das Band läuft eine gewisse Zeit (z.B. 8 Sekunden) nach.

Varianten:

1. Die Koffer werden vom Band geworfen.
2. Die Koffer werden nach einer gewissen Zeit (z.B. 1 Minute) vom Band geworfen.
3. Überall entlang des Bandes gibt es Lichtschranken.
4. Der Bandmotor hat eine Totmann-Schaltung.

3.8.2 Fahrstuhl

Das Automatisierungssystem für die Steuerung eines Fahrstuhls mit drei Stockwerken habe folgende

digitale Eingänge:

$s1$**,** $s2$**,** $s3$**:** Sind `true`, wenn der Haltebereich eines Stockwerks erreicht ist, sonst `false`. Wenn der Haltebereich erreicht ist, kann der Motor des Fahrstuhls abgeschaltet werden und der Fahrstuhl wird richtig an dem betreffenden Stockwerk halten.

1, 2, 3: Sind `true`, wenn im Fahrstuhl ein Stockwerksziel gedrückt wurde. Dabei geht auch das entsprechende Lämpchen an, ohne dass der Programmierer sich darum kümmern müsste.

$1h$**,** $2r$**,** $2h$**,** $3r$**:** Sind `true`, wenn auf einem Stockwerk eine Anforderungstaste gedrückt wurde.

digitale Ausgänge:

M0: Fahrstuhl-Motor aus

M1: Fahrstuhl-Motor ein

Rr: Fahrstuhl-Richtung runter

Rh: Fahrstuhl-Richtung hoch

zero1: Lösche Lampe 1, Signal 1 ist dann `false`.

zero$1h$**:** Lösche Lampe $1h$, Signal $1h$ ist dann `false`.

entsprechend: für die Signale 2, 3, $2r$, $2h$, $3r$.

Alle Knöpfe können gleichzeitig gedrückt werden.

Aufgabe: Erstelle einen Zustandsgraphen ohne Start- und Fehlerzustände.

3.9 Zustandsmanager

Zustandsgraphen können mit einem einfachen Zustandsmanager implementiert werden. Der Zustandsmanager ist eine Funktion oder Methode, die folgende generelle Struktur hat:

```
public void manageZustaende ()
   {
   // Für jeden Zustand wird eine ganzzahlige Variable definiert:
   final int   START_ZUSTAND = 1;
   final int   ZUSTAND_A = 2;
   final int   ZUSTAND_B = 3;
   // ...
   final int   END_ZUSTAND = 0;

   // Definition einer Zustandsvariablen und Initialisierung
   // mit dem Startzustand:
   int   next = START_ZUSTAND;

   // Definition einer Abbruchvariablen:
   boolean  weiter = true;

   // do-while-switch-case Konstrukt, der von Zustand zu Zustand führt:
   while (weiter)
      {
      switch (next)
         {
         case START_ZUSTAND:
            // ...
            break;
         case IRGEND_EIN_ZUSTAND:
            // entry-Aktionen
            // entry-Aktivitäten (nicht unterbrechbar)
            // do-Aktivitäten (unterbrechbar)
            // Warte auf ein Ereignis oder polle Ereignisse
            // Analysiere das Ereignis
            next = // nächster Zustand
            // Breche gestartete do-Aktivitäten ab.
            // exit-Aktionen und -Aktivitäten
            break;
         case END_ZUSTAND:
            weiter = false;
            // ...
            break;
         }  // end switch
      }  // end while
   }  // end manageZustaende
```

Die Struktur eines Zustands sei hier noch einmal hervorgehoben (Der Anfänger sollte dabei

unbedingt die Reihenfolge einhalten!):

1. Führe entry-Aktionen aus.
2. Führe entry-Aktivitäten (nicht unterbrechbar) aus.
3. Starte do-Aktivitäten (unterbrechbare).
4. Warte auf Ereignisse.
5. Analysiere das Ereignis.
6. Bestimme den nächsten Zustand.
7. Breche gestartete do-Aktivitäten ab.
8. Führe exit-Aktionen und exit-Aktivitäten aus.

Dieser Ansatz ist ziemlich einfach und sicher zu programmieren, obwohl es kompliziert werden kann, wenn wir unterbrechbare Aktivitäten haben.

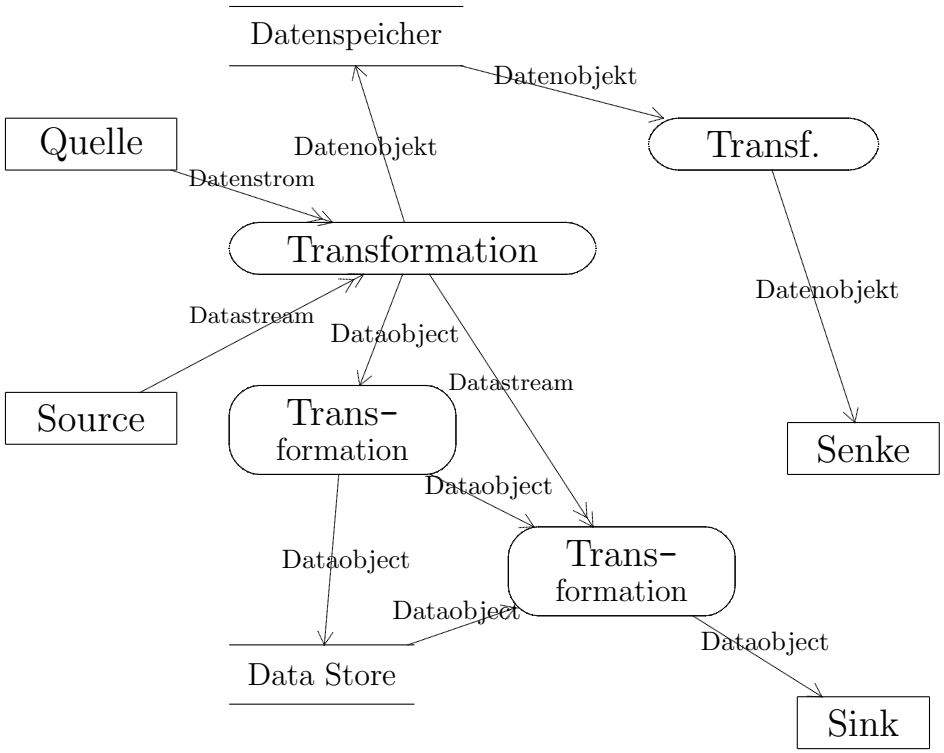
Der nächste Schritt wäre die Verpackung der einzelnen `case`-Konstrukte in Methoden, die den nächsten Zustand zurückgeben. Diese Methoden müssten aber irgendwie den Zustandsgraphen kennen. Daher wäre ein weiterer Schritt, dass man diese Methoden so programmiert, dass sie nur Ergebnisse liefern, mit denen über eine Transitionsmatrix der nächste Zustand bestimmt wird.

Zur Realisierung eines Ereignis-Monitors eignet sich möglicherweise das Observer-Pattern.

Bezüglich einer durchgängig objekt-orientierten – aber für die meisten Fälle wohl zu aufwendigen – Programmierung sei auf das Dokument `www.ba-stuttgart.de/~kfg/pdv/zstdgrph.pdf` verwiesen.

Kapitel 4

Funktionales Modell



Das funktionale Modell beschreibt die Berechnungen im System. Es zeigt, wie **Eingangsdaten** ***(Input)*** in **Ausgangsdaten** ***(Output)*** transformiert werden. Es besteht aus einem vernetzten **Datenflussdiagramm** ***(data flow diagram, DFD)***, das den Fluss der Daten von externen Eingängen durch **Transformationen** (oder Operationen oder Prozesse) ***(transformations****, operations, processes)* und durch **Datenspeicher** ***(data store)*** zu externen Ausgängen zeigt.

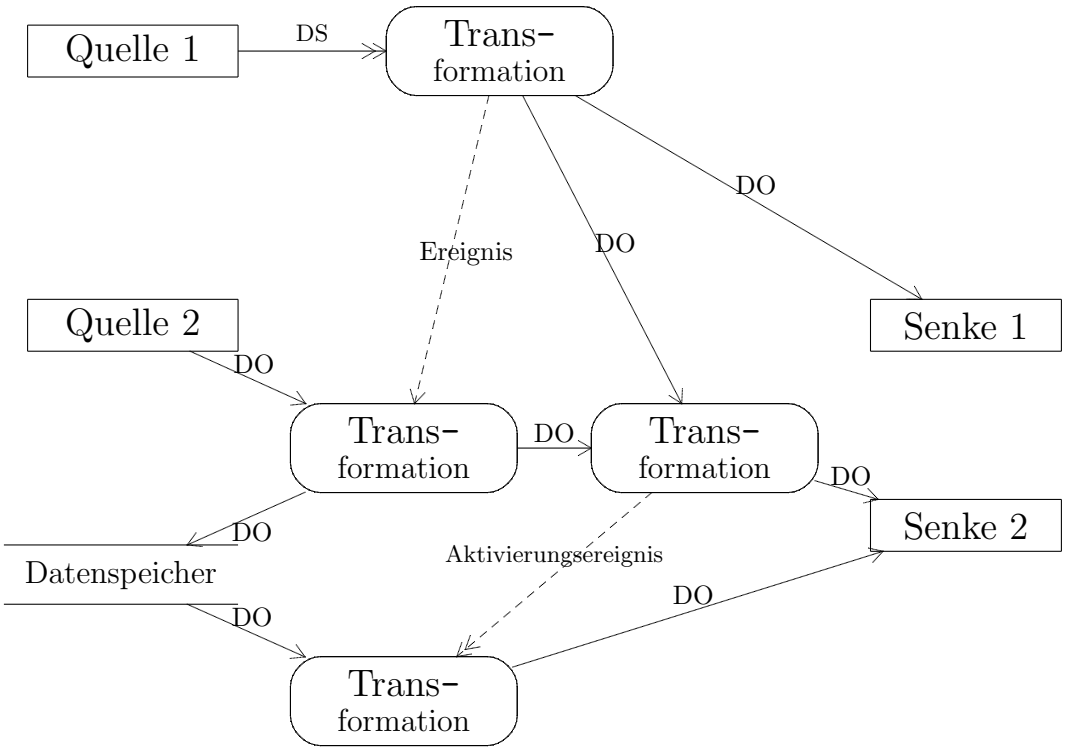
Die **Quellen** ***(source)*** und **Senken** ***(sink, destination)*** der Daten sind Objekte und werden **Aktoren** ***(actor, terminator)*** genannt. Sie heißen auch **Datenerzeuger** ***(data producer)***

und **Datenverbraucher *(data consumer)***. Transformationen und Datenspeicher können auch Erzeuger und Verbraucher sein.

Die Aktoren werden durch Rechtecke dargestellt, die Datenspeicher durch parallele Linien und die Transformationen durch Ovale. Die Datenflüsse werden durch Pfeile vom Erzeuger zum Verbraucher dargestellt, die mit den transportierten Daten bezeichnet sind. Diese Daten sind Objekte. Transformationen sind Methoden von Objektklassen. Das kann die Klasse eines der Eingangs- oder Ausgangsobjekte sein oder auch die Klasse eines Aktors.

Bei diskreten Datenflüssen sind die Pfeile *eine* Pfeilspitze, bei kontinuierlichen Datenflüssen haben die Pfeile *zwei* Pfeilspitzen hintereinder.

Kontrollfluss-Diagramme (CFD): Es gibt auch die Möglichkeit, **Kontrollflüsse** (Ereignisse) darzustellen. Das sind dann gestrichelte Pfeile. Entsprechend werden **Kontrollfluss-Transformationen** und **Ereignisspeicher** gestrichelt dargestellt. Eine **Prozessaktivierung** ist ein Kontrollfluss mit doppelter Pfeilspitze.



Die DFDs können hierarchisch gegliedert werden, indem jede Transformation in ein eigenes DFD expandiert werden kann, bis schließlich nur **atomare *(atomic)*** Transformationen übrig bleiben.

Atomare Transformationen können dann durch Zustandsgraphen oder Entscheidungstabellen beschrieben werden.

Die oberste DFD-Ebene (größte Abstraktions-Ebene) könnte nur *eine* Transformation und ihre Verbindungen zu den externen Datenquellen und -senken enthalten. Solch ein Diagramm heißt dann **Kontextdiagramm**.

DFDs sind die Grundlage für die **Strukturierte Analyse** ***(structured analysis, SA, structured analysis design technique, SADT) [5]*** .

Kapitel 5

Nebenläufigkeit

Dieses Kapitel behandelt eine Abstraktion der Probleme bei Betriebs- und Realzeit-Systemen. Reale (technische) Systeme bestehen aus nebenläufig oder parallel ablaufenden Prozessen. Um für solche Systeme einen Ausschnitt der realen Welt bequem modellieren zu können, muss das Software-Entwicklungs-System nebenläufige Prozesse zur Verfügung stellen.

Bei Realzeit-Systemen spricht man von **Nebenläufigkeit, *Concurrency*** (Gehani): *A concurrent program specifies two or more sequential programs that may be executed concurrently as parallel process.*

Oft wird in diesem Zusammenhang der Begriff "Parallelität" bzw. "parallel" synonym für "Nebenläufigleit" bzw. "nebenläufig" verwendet.

Wir machen hier folgende Unterscheidung: Nebenläufige Prozesse sind dann parallele Prozesse, wenn sie wirklich gleichzeitig, also etwa auf zwei oder mehreren Prozessoren ablaufen. "Nebenläufigkeit" ist der allgemeinere Begriff und umfasst auch die "Quasi-Parallelität", die uns ein Multitasking-Betriebssystem auf einem oder wenigen Prozessoren vorspiegelt.

Das Gegenteil von "nebenläufig" ist "sequentiell" oder "seriell".

Abhängig vom Betriebssystem oder der Programmiersprache spricht man von "Prozessen", "Threads" oder "Tasks". Wir werden hier als übergeordneten Begriff **"Task"** verwenden.

Betrachten wir als Beispiel ein Realzeit-System, das einen Druck überwacht, verschiedene andere Signale (1 bis 4) verarbeitet und schließlich eine Berechnung durchführt. Das könnte man folgendermaßen programmieren, wobei der Code zyklisch durchlaufen wird:

```
Druck
Signal1
Signal2
Signal3
Signal4
Berechnung
```

Wenn der Druck sehr wichtig ist, dann könnte das folgendermaßen programmiert werden:

```
Druck
Signal1
Druck
Signal2
Druck
Signal3
Druck
Signal4
Druck
Berechnung
```

Wenn nun `Signal3` eine mittlere Wichtigkeit hätte, wie würde dies dann aussehen? Es würde sehr unübersichtlich werden. Solch ein System ist also sehr schwer wart- und erweiterbar, etwa um weitere Signale. Daher wünscht man sich hier ein Betriebssystem, das die nebenläufige Kontrolle von Automatisierungsaufgaben und die Vergabe von Prioritäten erlaubt. Jeder Signal- oder Ereignisquelle wird eine **Task** zugeordnet. Jede Task bekommt eine bestimmte Priorität. Dann hätten wir ein aus mehreren nebenläufigen Programmen (Tasks) bestehendes Programm. Alle Programme werden nebenläufig mit unterschiedlicher Priorität als Task zyklisch durchgeführt:

```
Task D Prio 1     Task 1 Prio 2     Task 2 Prio 2     ...   Task B Prio 3

   Druck             Signal1           Signal2        ...      Berechnung
```

Solch ein System ist relativ leicht zu warten und zu erweitern.

Die Parallelität ist bei einem Einprozessorsystem nur eine **Quasiparallelität** oder **Pseudoparallelität**, d.h. der Prozessor bearbeitet zu einem Zeitpunkt immer nur genau eine Task. Der **Scheduler** ist die Komponente eines **Multitasking-**Betriebssystems, die den Tasks die Kontrolle über die CPU zuteilt. Er lässt das System nach außen hin parallel erscheinen.

Bemerkungen:

1. Aus Gründen der Performanz und/oder Einfachheit des Systems werden Systeme *ohne* Nebenläufigkeit bei Embedded Systemen und in der Mikroprozessorprogrammierung durchaus noch verwendet.

2. Normalerweise ist ein gut entworfenes nebenläufges Programm trotz des Taskwechsel-Overheads performanter als ein rein sequentielles Programm.

3. Die Verwendung von Mehrprozessor-Systemen ***(multi core)*** wird zur Zeit sehr diskutiert. Teilweise werden sie aus Gründen der Performanz schon eingesetzt, teilweise sind sie aus Gründen der Sicherheit (noch?) regelrecht verboten.

5.1 Arten, Vorteile, Notwendigkeit von Nebenläufigkeit

5.1.1 Parallele Algorithmen

Bei einem parallelen Algorithmus kann eine Berechnungsaufgabe so in Teilaufgaben zerlegt werden, dass die Teilaufgaben parallel abgearbeitet werden können. Beispiele sind Quicksort oder Fast-Fourier-Transformation. Eigentlich sind das sogenannte *binäre* Algorithmen. Sie werden normalerweise sequentiell durchgeführt, bringen aber einen deutlichen Zeitgewinn aufgrund des parallelen bzw. binären Algorithmus.

5.1.2 Konzeptionelle Vereinfachung

Betriebssystem-, Realzeit-System- und Datenbankprogramme werden durch Nebenläufigkeit konzeptionell vereinfacht, indem die Datenverarbeitung als ein (unendlicher) Datenfluss aufgefasst wird, der durch verschiedene *nebenläufig laufende* Unterprogramme bearbeitet und verändert wird:

Durch eine solche Aufteilung kann ein sehr verschachtelter Code in eine sequentielle Form gebracht werden, die leichter zu schreiben, zu lesen und zu warten ist.

Berühmt ist die Aufgabe von Conway: Lies einen Textfile mit Zeilen der Länge 40. Ersetze jede Folge von n = 1...10 Nullen durch eine Null gefolgt von n mod 10. Gib aus als Textfile mit Zeilen der Länge 75.

1. Erstellung **eines** ("monolithischen") Programms `leprschr`:

   ```
   $ leprschr < inFile > outFile
   ```

2. Erstellung von **drei** Programmen `lese`, `presse` und `schreibe`, die dann unter Verwendung des Pipe-Mechanismus nebenläufig ausgeführt werden:

   ```
   $ lese < inFile | presse | schreibe > outFile
   ```

Es zeigt sich, dass Aufgabe 2.) wegen der Aufteilung in drei einfache Aufgaben wesentlich einfacher als Aufgabe 1.) ist.

5.1.3 Multicomputer-Architekturen

Durch Parallelität können Multicomputer-Architekturen effizient ausgenutzt werden. Wir unterscheiden da:

- Transputer (Parallelität auf einem Chip)
- Multiprozessor-Architektur (Parallelität auf einem Board)
- Vernetzte Computer ***(Grid-Computing, Cloud-Computing)***

5.1.4 Ausnützen von Wartezeiten

Bei Uniprozessoren werden durch Nebenläufigkeit Wartezeiten effizient ausgenützt, um andere Benutzer zu bedienen. (Das dürfte die ursprüngliche Motivation zur Entwicklung von Multitasking/Multiuser-Betriebssystemen gewesen sein.) Ferner kann der Prozessor allen Benutzern gerecht zugeteilt werden.

5.1.5 Nebenläufigkeit als Modell

Die Systementwicklungsaufgabe erfordert die Nebenläufigkeit als Modell. Die zu modellierende reale Welt besteht aus parallel ablaufenden Prozessen. Die Überwachung von mehreren Alarmen oder die Steuerung eines Roboterarms mit mehreren Achsen würde sequentiell zu stark verzahnten oder verschachtelten Programmen führen.

Diese Art von Nebenläufigkeit ist das Thema der Vorlesung.

5.2 Konkurrenz und Kooperation

Die wichtigste Aufgabe bei der nebenläufigen Programmierung ist die **Synchronisation** und der **Datenaustausch** zwischen den Tasks. Die Tasks konkurrieren ***(Competition)*** um Betriebsmittel, müssen aber in einem technisch interessanten Programm auch zusammenarbeiten ***(Cooperation)***.

Für die konzeptionelle und programmiertechnische Vereinfachung durch den Einsatz von nebenläufigen Programmen muss ein Preis entrichtet werden: Die Anzahl der möglichen Verschachtelungen von nebenläufigen Tasks ist für praktische Zwecke unendlich. Außerdem ist es selten möglich, eine gewünschte Verschachtelung "einzustellen".

Nebenläufige Programme sind daher nicht erschöpfend zu testen.

Sie laufen i.A. **nicht deterministisch** ab – oft wegen Input von außen (Reaktion auf Signale) zu **unbekannten** Zeiten. (Per definitionem: Interrupts passieren zu unvorhersehbaren Zeitpunkten.) Das Resultat ist eine willkürliche **Verschachtelung** der Programme. Wenn ein Programm "mal richtig, mal falsch" läuft, ist das meistens ein Zeitproblem.

Schauen wir uns als Beispiel folgenden Pseudocode in Pascal-S an:

```
program increment;
   const m = 8;
   var n :integer;

   procedure incr;
      var i :integer;
      begin
         for i:= 1 to m do n := n + 1;
      end;

   begin (* Hauptprogramm *)
```

```
   n := 0;
   cobegin
      incr;   incr;
   coend;
   writeln ('Die Summe ist : ', n);
end.
```

Was ist `n` ?

Bemerkungen:

1. Praktisch gibt es "unendlich" viele Verschachtelungen (Szenarien *interleaving code*). Berechnung? Siehe Übung.
2. Jede Verschachtelung ist gleich wahrscheinlich.
3. Aber wieviele Verschachtelungen ein bestimmtes Resultat liefern, ist sehr unterschiedlich. Das Resultat 10 ist z.B. sehr wahrscheinlich, während 7 schon relativ selten ist. Das Resultat 2 wird man wohl nicht erleben.

Da es praktisch unendlich viele Möglichkeiten der Verschachtelung gibt, können Fehler durch Testen normalerweise nicht ausgeschlossen werden. Fehler zeigen sich bei nebenläufigen Programmen nicht vorhersehbar. Wenn sie schließlich in Erscheinung treten, dann geschieht das oft im ungünstigsten Moment: Das System ist in Produktion und läuft unter hoher Last.

Da man nicht erschöpfend testen kann, sind eigentlich theoretische Beweise notwendig. Diese sind aber häufig zu schwierig oder zu aufwendig. In der Praxis wird man daher nach extremen oder schwierigen **Szenarien** suchen, mit denen ein Programm "theoretisch" getestet wird. Dabei dürfen keine Annahmen über das Zeitverhalten gemacht werden. Man muss **asnychron** denken.

Durch die Quasiparallelität von Einprozessor-Systemen scheinen falsche Programme oft richtig zu laufen, weil es nicht oder nur extrem selten zu einer wirklich willkürlichen Verschachtelung kommt. Man muss aber immer damit rechnen, dass z.B. ein Mehrprozessorsystem verwendet wird und es dadurch zu einer **echten Parallelität** kommt! Daher muss jede beliebige Verschachtelung in Betracht gezogen werden!

Die verschiedenen Szenarien haben normalerweise ganz unterschiedliche Wahrscheinlichkeiten für ihr Auftreten. Extreme Szenarien haben oft um viele Zehnerpotenzen kleinere Wahrscheinlichkeiten als "normale" Szenarien und sind daher experimentell kaum sichtbar zu machen.

Wir fassen zusammen:

- Fehler treten auf wegen besonderer zeitlicher Verschachtelungen.
- Die Fehler passieren sehr selten.
- Die Fehler sind schwer zu reproduzieren und zu finden.

Was bedeutet **Korrektheit**? Bei sequentiellen Programmen genügt meistens die richtige Antwort in Grenzfällen und bekannten Fällen.

Bei nebenläufigen Programmen muss darüber hinaus noch die **Sicherheit** und die **Lebendigkeit** gegeben sein.

- **Sicherheit** ***(safety)***: **Kritische Abschnitte** (oft Zugriff auf eine Ressource) ***(critical sections, "read-modify-write")*** müssen sich gegenseitig ausschließen **(Problem des gegenseitigen Ausschlusses,** ***mutual exclusion, "nothing bad ever happens to an object"[16])***. Wenn der Zustand eines Objekts verändert wird, muss dafür gesorgt werden, dass das Objekt immer von einem **konsistenten** Zustand in einen anderen konsistenten Zustand überführt wird. Dabei können durchaus **inkonsistente** Zustände **vorübergehend** (transient) auftreten. Das kann allerdings zu Problemen führen, wenn mehrere Tasks das Objekt gleichzeitig verändern.

  Da das Ergebnis von der Reihenfolge abhängen kann, in welcher die Tasks zum Zuge kommen – "den Wettlauf um die Ressource gewinnen", spricht man hier auch von sogenannten ***race conditions***.

  Daher müssen solche Zustandsänderungen unter gegenseitigem Ausschluss durchgeführt werden.

- **Lebendigkeit** ***(liveliness or liveness[16], "something good eventually happens")***: Jede Aktivität muss irgendwie "vorankommen". Lebendigkeit wird durch folgende Szenarien gefährdet:

  - **Verklemmung** ***(deadlock, deadly embrace)***: Das System steht, weil die Tasks gegenseitig aufeinander warten.
  - **Aussperrung** ***(starvation)***: Einzelne Tasks kommen nicht mehr zum Zug.
  - **Verschwörung** ***(conspiracy, livelock)***: Unwahrscheinliche Synchronisations-Szenarien machen die Leistung des Systems unvorhersehbar. D.h. die Tasks verschwören sich gegen die/den EntwicklerIn zu einem – normalerweise unwahrscheinlichen – blockierenden Zeitverhalten.

  Lebendigkeit kann man auch graduell verstehen. D.h. ein System kann mehr oder weniger performant oder reaktionsfähig oder nebenläufig sein. Sicherheit und Lebendigkeit sind oft gegenläufig. Eine erhöhte Sicherheit kann die Lebendigkeit einschränken und umgekehrt.

Um Lebendigkeit zu gewährleisten gibt es verschiedene Strategien:

- **Entdeckung und Behebung** von Verklemmungen bzw. Aussperrungen ***(detection and recovery)***
- **Verhinderung** ***(prevention)***: Es gibt keine Zustände, die die Lebendigkeit beeinträchtigen.
- **Vermeidung** ***(avoidance)***: Zustände, die die Lebendigkeit beeinträchtigen, werden vermieden.

Bei der Behandlung dieser Probleme durch Synchronisationsmechanismen spielen folgende Begriffe eine wichtige Rolle:

- ***Lock:*** Ein Lock ist ein Konstrukt, um den Zugang zu einem kritischen Bereich zu kontrollieren. Nur **eine** Task kann ein Lock für einen kritischen Bereich bekommen. Der Zugang ist exklusiv.

- ***Permit:*** Ein Permit ist ebenfalls ein Konstrukt, der den Zugang zu einem kritischen Bereich kontrolliert. Allerdings können **mehrere** Tasks ein Permit bekommen (bis zu einer anwendungsbedingten, festgelegten Obergrenze).
- Lock und Permit schließen sich gegenseitig aus. D.h. wenn eine Task ein Lock auf ein Objekt hat, kann keine andere Task ein Permit oder Lock auf dasselben Objekt bekommen. Wenn eine Task ein Permit auf ein Objekt hat, können zwar andere Tasks ebenfalls ein Permit auf dasselbe Objekt bekommen, aber keine andere Task kann ein Lock auf dieses Objekt bekommen.
- ***Block:*** Die genannten Konstrukte werden so eingesetzt, dass jede Task vor Betreten eines kritischen Bereichs versucht, ein Lock oder ein Permit zu bekommen. Ein Block zeigt an, dass die Task auf die Zuteilung eines Locks oder eines Permits wartet. (Es gibt aber auch die Möglichkeit, dass die Task nicht wartet, sondern ohne Betreten des kritischen Bereichs anderen (unkritischen) Code durchführt.)

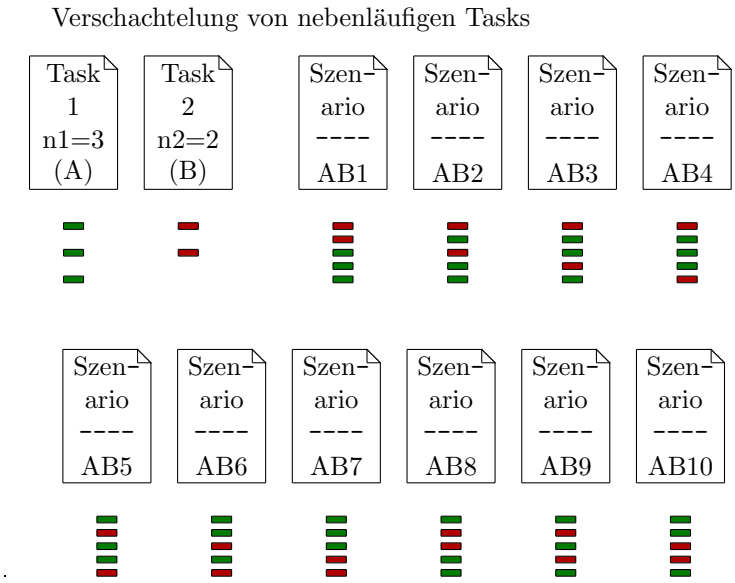
Nach verlassen des kritischen Bereichs wird das Lock bzw. das Permit wieder freigegeben.

5.3 Übungen

5.3.1 Anzahl Szenarien

Wie groß ist die Anzahl der Szenarien (Verschachtelungen) v bei t Tasks $i = 1 \dots t$, mit jeweils n_i atomaren Anweisungen?

Beispiel:



Kapitel 6

Synchronisationsmechanismen

Ein Echtzeit-System (Automatisierungssystem, AS) muss sich mit den Zuständen eines (externen technischen) Prozesses (P) synchronsieren, d.h. muss versuchen, gewisse *interne* Zustände zu erreichen, bevor oder nachdem gewisse externe Prozess-Zustände erreicht werden bzw. wurden. Es wäre kein Problem, einen externen Prozess zu steuern, wenn dessen Fortschreiten von Zustand zu Zustand beliebig durch das Automatisierungssystem verlangsamt oder beschleunigt werden könnte. Aber i.A. gehen die meisten realen Prozesse relativ unkontrollierbar von Zustand zu Zustand.

In diesem Kapitel werden wir uns mit folgenden Synchronisationsmechanismen beschäftigen:

- Zeit
- **Semaphor**
- Bolt-Variable
- **Monitor**
- Rendezvous
- **Channels (Kommunikationskanäle)**

Semaphor und Monitor als die am häufigsten eingesetzten Mechanismen werden wir eingehender behandeln.

6.1 Zeit

Die Verwendung der Zeit als Synchronisations-Mittel bedeutet, dass man Zeit- oder Stundenpläne für die einzelnen Tasks aufstellt. Ein Zeitplan gibt an, wann eine Task was wielange tun soll. Solche Zeitpläne wären ein nützliches Synchronisations-Mittel, wenn reale Prozesse deterministisch ablaufen würden. Das ist aber nicht der Fall. Daher ist die Zeit kein geeignetes Synchronisationsmittel. Im Gegenteil, sie stellt in der Form von einzuhaltenden Zeitplänen eine wichtige Aufgabe der Automatisierung dar, die es zu lösen gilt.

6.2 Semaphore

Für die Synchronisation von Tasks und die Kommunikation zwischen Tasks über Shared Memory hat Dijkstra **Semaphore** (Zeichenträger) eingeführt.
(Der Begriff "Semaphor" ist Neutrum oder Masculinum, also "das" oder "der" Semaphor.)

Semaphore sind ein Synchronisationsmittel, mit dem sogenannte **kritische Abschnitte**, Bereiche oder Gebiete ***(critical sections)*** geschützt werden können. Das sind Codestücke, die nur unter gegenseitigem Ausschluss oder nur in gewisser Reihenfolge betreten werden dürfen.

Semaphore sind folgendermaßen definiert:

- Es gibt eine ganzzahlige Semaphorvariable `s`, die nur **einmal** initialisiert werden kann und für die die folgenden beiden **atomaren (unteilbaren)** und **sich gegenseitig ausschließenden** Systemaufrufe zur Verfügung stehen:
- `p (s) :`
  aufrufende Task wartet bis `s > 0`
  `s = s - 1`
- `v (s) :`
  `s = s + 1`

Bemerkungen:

1. Eine Task wird (eventuell) durch ein Semaphor "gesperrt".
2. Durch ein `v (s)` wird maximal *eine* Task "freigegeben".
3. Das Abfragen des Wertes eines Semaphors ist nicht erlaubt.
4. Ein Semaphor verwaltet typischerweise ein Betriebsmittel, das gesperrt oder freigegeben wird. Es löst das Problem des gegenseitigen Ausschlusses, ohne dass es zu Aussperrung und Verklemmung kommt. Als Beispiel betrachten wir einen Drucker, der nur exklusiv benutzt werden kann.

   `init s = 1`

Task A	Task B	Task C
↓	↓	↓
`p (s)`	`p (s)`	`p (s)`
`drucken`	`drucken`	`drucken`
`v (s)`	`v (s)`	`v (s)`
↓	↓	↓

5. Bezeichnungen: Betriebssysteme und Echtzeit-Sprachen verwenden unterschiedliche Namen für die Semaphor-Operationen. Im Folgenden sind ein paar Beispiele aufgeführt:

`p (s)`	`v (s)`	Dijkstra
passeren	verlaten	Dijkstra
sperren	freigeben	
`s.p ()`	`s.v ()`	objekt-orientiert
`request (s)`	`release (s)`	
`request (s)`	`free (s)`	
`wait (s)`	`signal (s)`	
`take (s)`	`give (s)`	
`semTake (s)`	`semGive (s)`	VxWorks
`down (s)`	`up (s)`	
`pend (s)`	`post (s)`	
`s.acquire ()`	`s.release ()`	Java
Ereignis abwarten	Ereignis senden	Ereignissicht

6. Wesentlich ist, dass die Semaphoroperationen **atomar** ***(atomic, primitive,*** **unteilbar)** sind. Sie laufen im nicht unterbrechbaren Teil eines Betriebssystems.

7. Ohne Semaphore heißt die Lösung "Dekkers Algorithmus" mit den folgenden Nachteilen:

   - Aktives Warten der Tasks.
   - Bei mehr als zwei Tasks wird die Programmierung sehr unübersichtlich.

8. **Binäre Semaphore:** Bisher konnte die Semaphorvariable jeden Wert ≥ 0 annehmen. Daher heißen diese Semaphore auch **allgemeine** oder ***counting*** **Semaphore**. Wenn wir nur die Werte 0 und 1 zulassen, dann sprechen wir von einem binären Semaphor. Im Algorithmus für `v (s)` muss dann `s = s + 1` durch

   ```
   s = 1
   ```

   ersetzt werden.

9. **Allgemeine Semaphore:** Wenn die Semaphor-Variable mit k initialisiert wird, dann kann das entprechende Betriebsmittel von bis zu k Tasks gleichzeitig benutzt werden. Als Beispiel könnte man sich k Drucker vorstellen, die als *ein* Betriebsmittel gesehen werden.

10. Außer der Initialisierung sind keine Zuweisungen an die Semaphorvariable erlaubt.

11. Für die Warteschlange wird keine FIFO-Strategie gefordert, aber auch nicht verboten. Jedenfalls dürfen Programme nicht abhängig von einer Warteschlangen-Strategie sein.

12. Es gibt zahllose Varianten. Zum Beispiel:

    - Atomares Sperren von mehreren Semaphoren, also etwa `p (s1, s2, s3)`.
    - Manche Betriebssyteme oder Sprachen bieten Semaphoroperationen mit Zeitschranken *(timeout)* an.
    - **Additive Semaphore:** Mit den Semaphoroperationen kann das Semaphor um ein Delta verändert werden. Wenn das Semaphor dadurch kleiner Null würde, wird die Task gesperrt.
    - **Semaphorgruppen:** Mit einer Semaphoroperation können für eine Anzahl von Semaphoren Delta-Werte übergeben werden. Falls dabei ein Semaphor kleiner Null würde, wird die Task gesperrt.

Es lohnt sich hier nicht, auf diese Varianten im Detail einzugehen. Das wird erst wichtig, wenn man mit einem konkreten Echtzeit-Betriebssystem oder -Sprache arbeitet. Da allerdings sollte man die Dokumentation sehr genau lesen.

13. **Semaphorproblematik:** Das Semaphor-Konzept bietet Lösungen für zwei Problembereiche, nämlich das

    Problem des **gegenseitigen Ausschlusses** *(competition)*

    und das

    Synchronisationsproblem **"Warten auf ein Ereignis"** und **"Senden eines Ereignisses"** (Ereignissicht, *cooperation)*.

    Beide Problembereiche werden vermischt und das führt zu konzeptionellen Verständnisproblemen. Die bisher definierten Semaphore sind eigentlich nur für die Ereignissicht vernünftig einsetzbar. Für den gegenseitigen Ausschluss sollten die unten definierten MUTEX-Semaphore verwendet werden. Manche Betriebssysteme (z.B. OSEK) kennen daher keine Semaphore, sondern nur MUTEXe (Ressourcen) und Ereignisse (Botschaften).

6.2.1 Synchronisationsbeispiel Schweiß-Roboter

Übung.

6.2.2 MUTEX-Semaphore

Mutual-Exclusion-Semaphore sind eine Erweiterung der binären Semaphore zur Realisierung des gegenseitigen Ausschlusses.

- Verschachtelter Zugriff durch dieselbe Task auf kritische Bereiche ist möglich ***(re-entrant)***. Eine Task, die ein MUTEX-Semaphor belegt, wird **Inhaber** ***(owner)*** des Semaphors. Ein Inhaber eines Semaphors kann das Semaphor **mehrfach** belegen ***(reentrancy, concept of ownership)***. Damit können Programme wesentlich modularer gestaltet werden. Ein "verschachtelter" Zugriff auf kritische Ressourcen ist möglich. Also bei folgendem Code wird sich eine Task nicht selbst aufhängen:

  ```
  init MUTEX-Semaphor  m = freigegeben;

  p (m);
     tu was Kritisches;
     p (m);
        tu noch mehr Kritisches;
     v (m);
  v (m);
  ```

  Bemerkungen:

  – MUTEX-Semaphore werden automatisch mit `1` initialisiert.

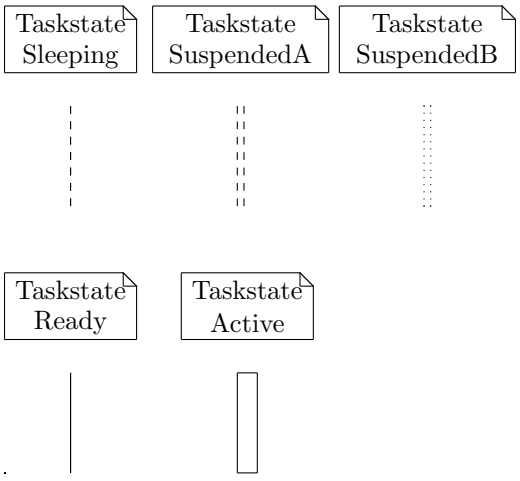
  - Die Inhaber*in eines MUTEX-Semphors muss den Semaphor so oft wieder freigeben, wie sie ihn belegt hat, damit der Semaphor für andere Tasks freigegeben ist.
  - Nur die Inhaber*in eines MUTEX-Semaphors kann den Semaphor freigeben.
  - Eine Task, die einen MUTEX-Semaphor besitzt, kann nicht durch eine andere Task gelöscht werden.
  - Im Allgemeinen kann ein MUTEX-Semaphor nicht von einer Interruptserviceroutine belegt oder freigegeben werden.
  - Unter Go sind MUTEXe **nicht** re-entrant!

- Die Verwaltung der MUTEX-Semaphore verursacht einen größeren internen Zeitaufwand als die Verwaltung binärer Semaphore.

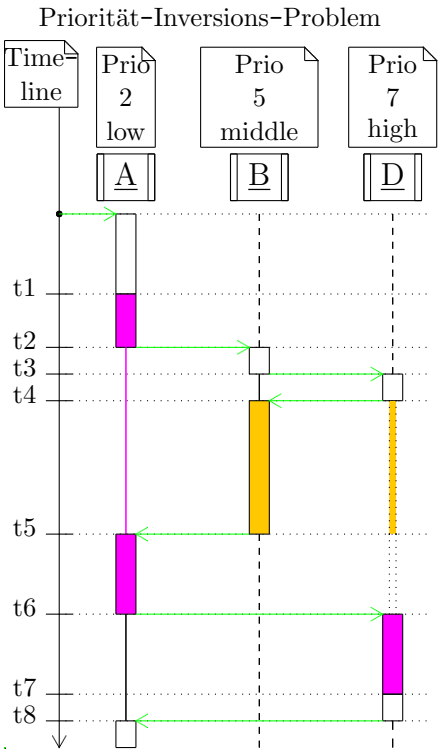
- **Prioritätsumkehr** wird verhindert.

Prioritäts-Inversion entsteht, wenn eine hochpriore Task D eine Ressource benötigt, die gerade eine niedrigpriore Task A besitzt, und eine mittelpriore Task B verhindert, dass Task A läuft, um die Ressource für D freizugeben.

In den folgenden Time-Line-Diagrammen werden Szenarien für das Phänomen (Problem) der Prioritäts-Inversion und möglicher Lösungen dargestellt. Die Notation ist folgende:



Das folgende Timeline-Diagramm zeigt ein Szenario mit Prioritäts-Inversions-Problem (PIP):



Das PIP ensteht, wenn eine Task mit niedriger Priorität (Task A) eine Ressource (einen kritischen Bereich, ein Betriebsmittel) zum Zeitpunkt `t1` exklusiv benutzen will (Magenta-farbiger Bereich).

Eine zunächst untätige, aber höherpriore Task B möchte zum Zeitpunkt `t2` laufen und bekommt wegen ihrer höheren Priorität den Prozessor. (Wir setzen immer ein ***priority based preemptive scheduling*** voraus, wie es für Echtzeit-Systeme typisch ist.) Task A wird in den Zustand Ready versetzt, hat aber immer noch die Ressource exklusiv.

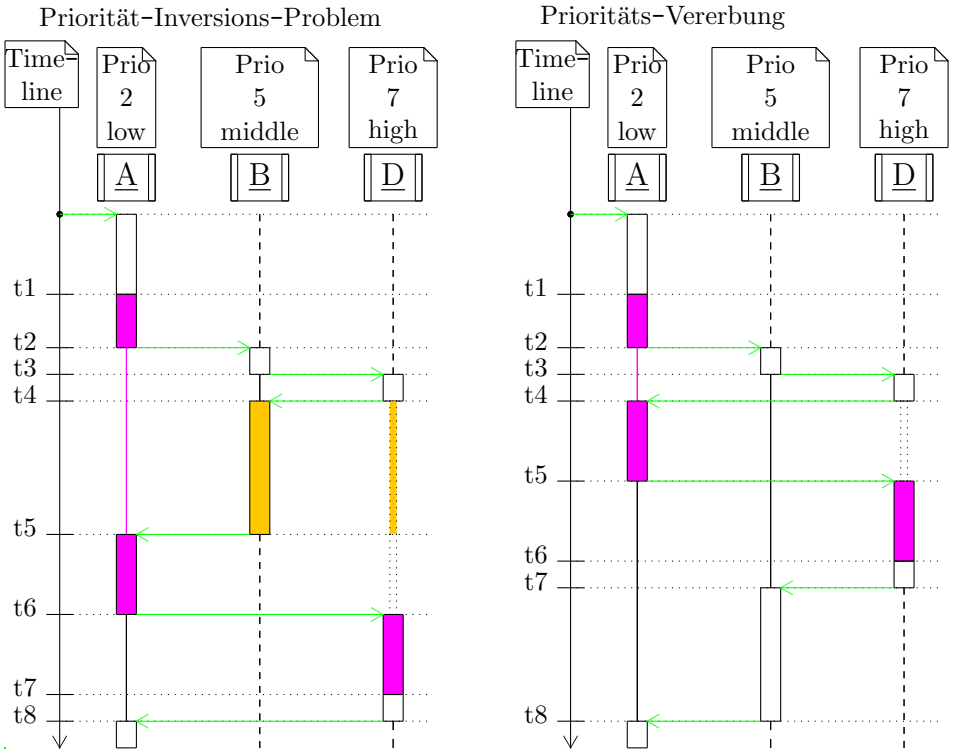
Eine dritte, auch zunächst untätige Task D mit noch höherer Priorität will zum Zeitpunkt `t3` ebenfalls laufen und bekommt den Prozessor wegen ihrer höheren Priorität. Task B wird in den Zustand Ready versetzt.

Zum Zeitpunkt `t4` möchte Task D diesselbe Ressource verwenden, die Task A gerade hat. Eine exklusiv genutzte Ressource darf aber niemals durch eine andere Task "gestohlen" werden! Also wird Task D suspendiert. Task B bekommt den Prozessor.

Ab Zeitpunkt `t4` bis Zeitpunkt `t5` (oranger Bereich) haben wir das Phänomen PIP, weil Task B die höherpriore Task D mit ihrer Aktivität beliebig lange "zappeln" lassen kann. Erst wenn sie sich zum Zeitpunkt `t5` entschließt, den Prozessor abzugeben, dann kann zwar Task D immer noch nicht laufen, weil sie warten muss, bis Task A zum Zeitpunkt `t6` die Ressource freigegeben hat. Zum Zeitpunkt `t6` bekommt dann Task D die Ressource und kann mit ihr arbeiten.

Es gibt zwei Möglichkeiten, das Problem der Prioritäts-Inversion zu lösen:

Vererbung von Prioritäten: Wenn z.B. Task D die Ressource zum Zeitpunkt `t4` anfordert, dann erbt Task A die Priorität von Task D, falls diese höher ist, bis Task A die Ressource wieder frei gibt.



Zum Zeitpunkt `t4` "erbt" Task A die Priorität von Task D und kann damit ungestört von B die Ressource verwenden und wieder für Task D zum Zeitpunkt `t5` freigeben.

Dieser Mechanismus ist häufig an spezielle Semaphore für den gegenseitigen Ausschluss gekoppelt (MUTEX–Semaphore, `synchronized` in Java).

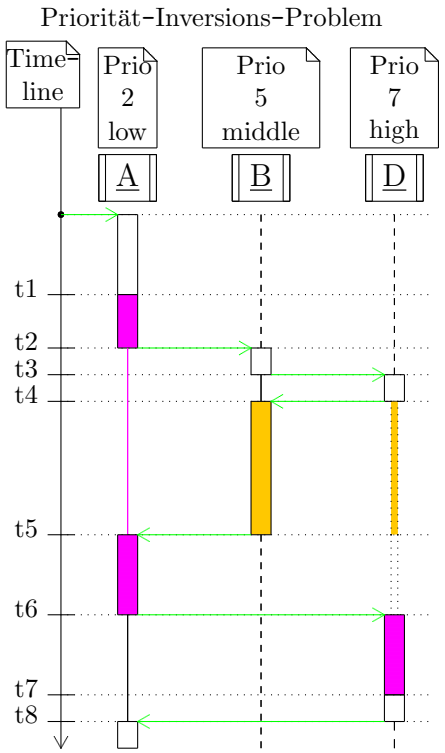
Priority Ceiling Protocol (PCP): Jede Ressource, die unter gegenseitigem Ausschluss benutzt wird, bekommt eine "höchste" Priorität ***(ceiling priority)*** (P_c). Diese Priorität P_c sollte mindestens so hoch sein wie die höchste Priorität (P_1) der Tasks, die die Ressource verwenden. Sie sollte aber kleiner sein als die kleinste Priorität (P_2) der Tasks, die die Ressource (BM) *nicht* benutzen und die höhere Priorität haben als P_1:

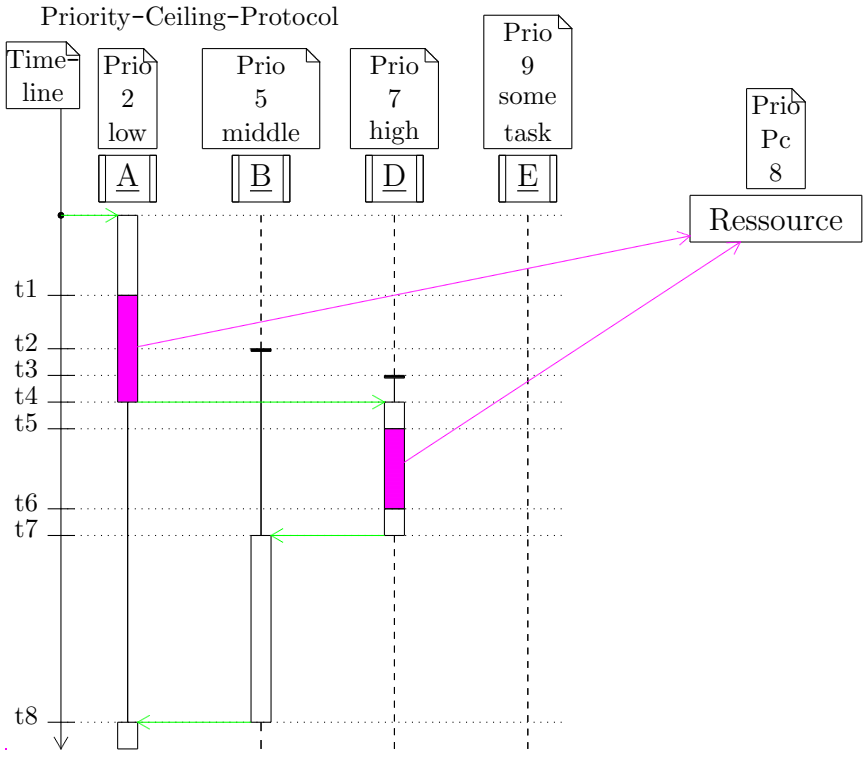
$P_1 = \max(P)$ über alle Tasks, die BM benutzen
$P_2 = \min(P)$ über alle Tasks mit $P > P_1$, die BM nicht benutzen
$P_1 \leq P_c < P_2$ (logisch)

Wenn eine Task eine Ressource benutzt, dann wird ihre Priorität während dieser Zeit auf die P_c der Ressource gesetzt, falls ihre eigene Priorität kleiner war.

Bemerkung: Hier beschrieben wurde das auch sogenannte "Immediate Ceiling Priority Protocol". Davor gab es noch das "Original Ceiling Priority Protocol", wobei die Priorität erst dann auf die Ceiling Priority einer Ressource angehoben wird, wenn eine andere Task auf diese Ressource zugreifen möchte.

Damit das Beispiel anschaulicher wird, wurde noch eine sehr hochpriore Task E hinzugefügt.

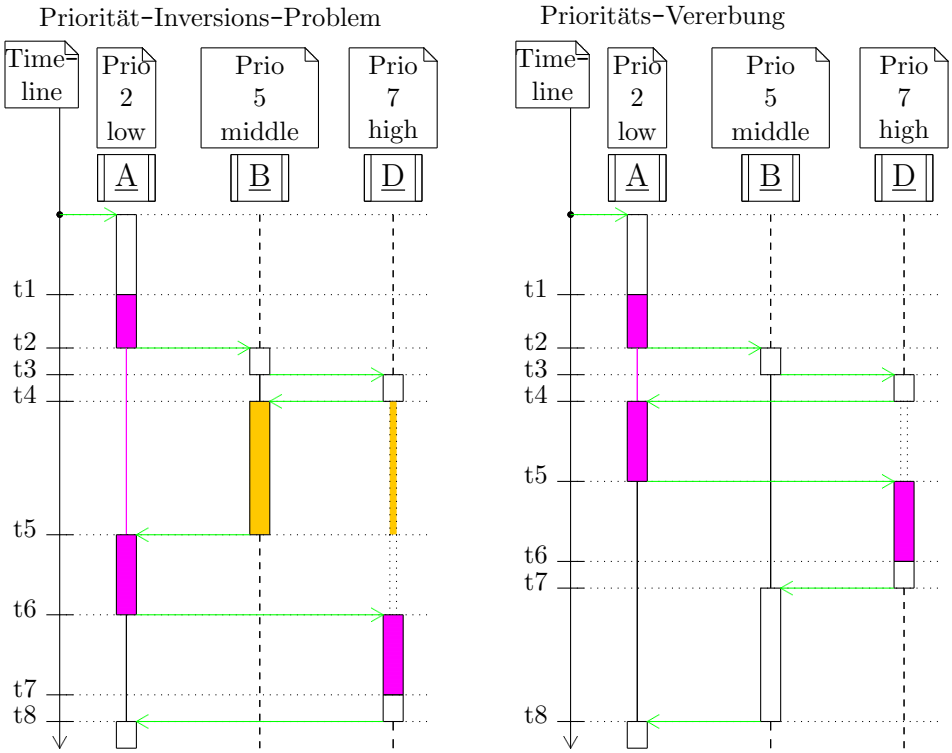

Priorität-Inversions-Problem
Time line
Prio 2 low
Prio 5 middle
Prio 7 high
A
B
D
t1
t2
t3
t4
t5
t6
t7
t8


Priority-Ceiling-Protocol
Time line
Prio 2 low
Prio 5 middle
Prio 7 high
Prio 9 some task
Prio Pc 8
Ressource
A
B
D
E
t1
t2
t3
t4
t5
t6
t7
t8

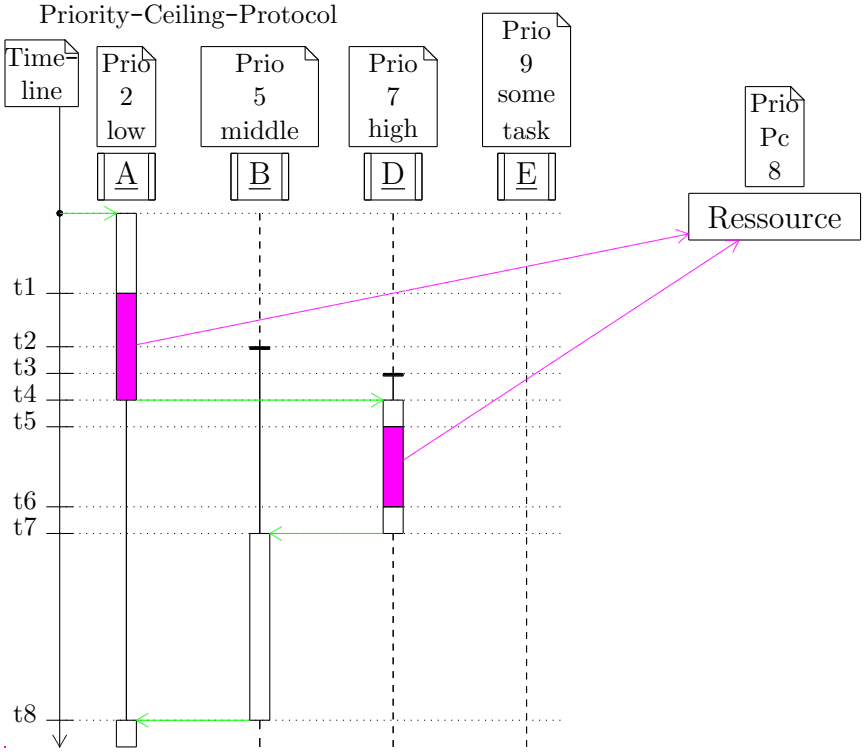
Für dieses Beispiel erhalten wir wir $P_1 = 7$ und $P_2 = 9$. Damit dürfen wir P_c auf 7 oder 8 setzen. Wir haben 8 gewählt.

Dieses Schema hat den Vorteil, dass eine niedrigpriore Task bei Benutzung der Ressource sofort mit höherer Priorität läuft und dadurch die Ressource wieder früher freigibt.
In der Regel sollen ja exklusiv genutzte Ressourcen möglichst schnell wieder freigegeben werden!
Es hat den Nachteil, dass es weniger dynamisch ist.

Das letzte Bild zeigt noch den Vergleich. PCP hat tendenziell weniger Taskwechsel.


Priorität-Inversions-Problem
Time line
Prio 2 low
Prio 5 middle
Prio 7 high
A
B
D
t1
t2
t3
t4
t5
t6
t7
t8

Prioritäts-Vererbung
Time line
Prio 2 low
Prio 5 middle
Prio 7 high
A
B
D
t1
t2
t3
t4
t5
t6
t7
t8


Priority-Ceiling-Protocol
Time line
Prio 2 low
Prio 5 middle
Prio 7 high
Prio 9 some task
Prio Pc 8
Ressource
A
B
D
E
t1
t2
t3
t4
t5
t6
t7
t8

Verklemmung: Wenn mehr als eine Ressource exklusiv beantragt wird, dann kann es zu Verklemmungen ***(deadlock)*** kommen.

Wir nehmen an, dass wir einen Drucker und einen Plotter haben, die gemeinsam exklusiv benutzt und daher durch die MUTEX-Semaphore `md` und `mp` geschützt werden sollen:

```
init md = 1
init mp = 1
```

Task A	Task B
↓	↓
`p (md)`	`p (mp)`
`p (mp)`	`p (md)`
`drucken`	`drucken`
`plotten`	`plotten`
`v (md)`	`v (mp)`
`v (mp)`	`v (md)`
↓	↓

Wenn Task A nach seinem erfolgreichen `p (md)` suspendiert wird, und dann Task B ein erfolgreiches `p (mp)` machen kann, werden beide durch die nächste Semaphoroperation gesperrt.

Das kann nur vermieden werden, wenn man fordert, dass immer in der gleichen Reihenfolge gesperrt wird:

```
init md = 1
init mp = 1
```

Task A	Task B
↓	↓
`p (md)`	`p (md)`
`p (mp)`	`p (mp)`
`drucken`	`drucken`
`plotten`	`plotten`
`v (md)`	`v (mp)`
`v (mp)`	`v (md)`
↓	↓

Deswegen bieten manche Systeme das atomare Sperren von mehr als einem Semaphor an (`p (md, mp)`).

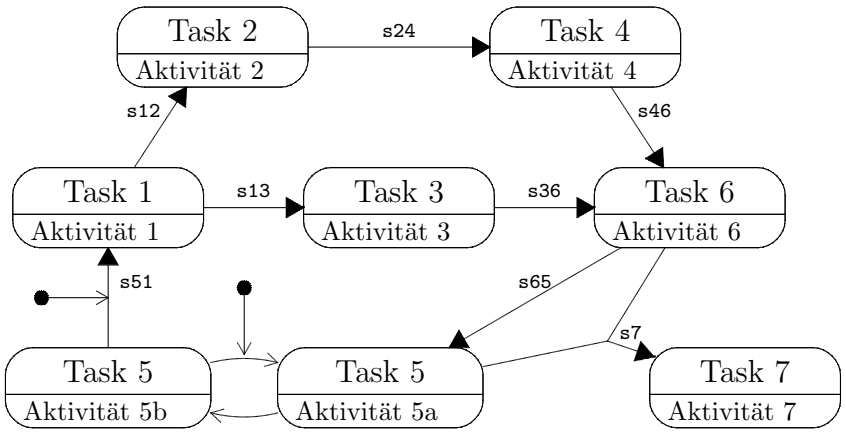
6.2.3 Herstellung von Ausführungsreihenfolgen

Für die Ereignis-Sicht (Warten-auf-und-Senden-von-Ereignissen) werden die Allgemeinen (oder *counting*) Semaphore verwendet. Damit werden meistens Abläufe gesteuert.

Wenn Aktivitäten von verschiedenen Tasks durchgeführt werden, dann muss man eventuell die Reihenfolge dieser Aktivitäten festlegen, um sogenannte ***race conditions*** zu vermeiden. Eine

Race-Condition liegt vor, wenn das Ergebnis davon abhängt, welche Task das Rennen bei nebenläufigen Aktivitäten gewinnt (z.B. beim Zugriff auf gemeinsamen Speicher).

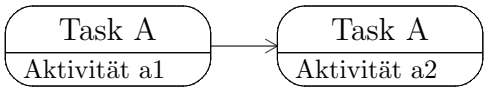
Im Folgenden **Ablauf-Diagramm** wird der Ablauf durch Ovale und Pfeile definiert.



Grafische Notation:

- Die Ovale enthalten einen Tasknamen und eine Aktivität.
- Das Ablauf-Diagramm ist so zu verstehen, dass alle Tasks gleichzeitig loslaufen, aber einige eventuell durch Synchronisationsmechanismen gesperrt werden.

  Typischerweise laufen die Tasks in einer Schleife, ohne dass das notwendigerweise im Diagramm angezeigt wird.
- Die Ovale werden durch einen oder mehrere Pfeile verbunden.
- Wenn ein Pfeil Ovale **ein- und derselben** Task verbindet (Beispiel Task 5), dann wird er mit einfacher Spitze notiert.



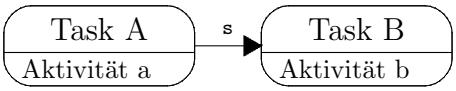
Im Code der Task A werden die beiden Aktivitäten in der durch den Pfeil angedeuteten Reihenfolge kodiert:

```
Task A:
// ...
Aktivität a1
Aktivität a2
// ...
```

- Wenn ein Pfeil zwei Ovale **unterschiedlicher** Tasks verbindet, wird ein Pfeil mit "gefüllter" Pfeilspitze verwendet. Er wird durch einen

  Warte-Auf-Sende-Ereignis-Mechanismus

  implementiert. Dazu können wir Semaphore oder Monitore verwenden. Eine Bezeichnung für das Synchronisationsmittel kann auf dem Pfeil angebracht werden.



  Diese Notation bedeutet, dass die Task B ihre `Aktivität b` erst nach Beendigung der `Aktivität a` in Task A beginnen darf.

  Im Folgenden nehmen wir allgemeine Semaphore als Synchronisationsmittel:

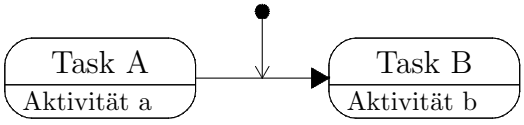
  > Jeder Pfeil zwischen unterschiedlichen Tasks wird mit einem allgemeinen Semaphor implementiert.

  Die Pfeilspitze bedeutet eine `p`-Operation ("warte auf Ereignis") am Anfang der Aktivität der Zieltask. Die Aktivität der Ausgangstask eines Pfeiles macht am Ende eine `v`-Operation ("sende Ereignis").

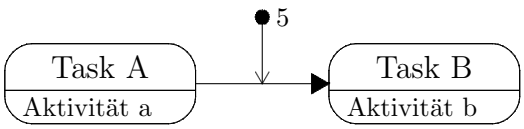
```
Task A:              Task B:
// ...               // ...
Aktivität a          p (s)
v (s)                Aktivität b
// ...               // ...
```

  Die Semaphore werden normalerweise mit 0 initialisiert. Das bedeutet, dass Task B vor Aktivität b gesperrt wird.

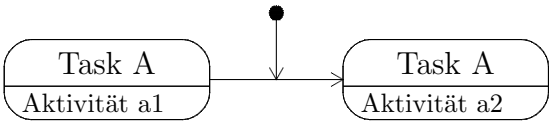
- Damit es überhaupt "losgeht", müssen eventuell eine oder mehrere Start-Aktivitäten definiert werden. Im Beispiel oben ist das "Aktivität a" der Task A. Das bedeutet, dass der oder die Semaphore, die den Eintritt in die Start-Aktivität kontrollieren, mit 1 initialisiert werden. Grafisch wird das mit einem "Pseudozustand" angegeben:



  Wenn ein Semaphor mit `>1` initialisiert wird, dann schreiben wir den Wert an den Pseudozustand:

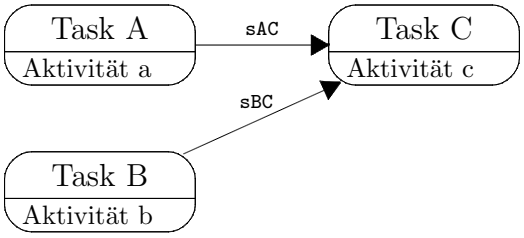


Diese Notation wird auch verwendet, um anzugeben, mit welcher Aktivität innerhalb *einer* Task begonnen wird (Beispiel Task 5).



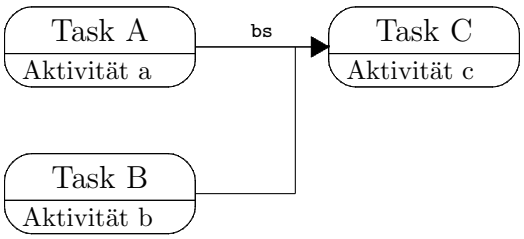
Hier wäre mit `Aktivität a2` zu beginnen.

- Wenn mehrere Pfeile in ein Oval gehen, dann bedeutet das eine "UND"-Verknüpfung. Im oben gezeigten Beispiel kann die `Aktivität 6` der Task 6 erst beginnen, wenn sowohl `Aktivität 4` als auch `Aktivität 3` fertig sind.




```
Task A:              Task B:              Task C:
// ...               // ...               // ...
Aktivität a          Aktivität b          p (sAC)
v (sAC)              v (sBC)              p (sBC)
// ...               // ...               Aktivität c
// ...               // ...               // ...
```

- Bei einer "ODER"-Verknüpfung vereinigen wir die beiden Pfeile vorher (Beispiel Task 7) und realisieren diese Pfeile – im einfachsten Fall! – durch einen gemeinsamen Binären Semaphor.




```
Task A:              Task B:              Task C:
// ...               // ...               // ...
Aktivität a          Aktivität b          p (bs)
```

```
v (bs)          v (bs)          Aktivität c
// ...          // ...          // ...
```

Die "ODER"-Verknüpfung ist eher untypisch. Die oft sehr komplexe Semantik lässt sich grafisch nicht vernünftig darstellen. Abhängig vom gewünschten Verhalten der beteiligten Tasks kann die Realisierung sehr kompliziert werden. Eventuell muss eine **Erzeuger-Verbraucher**-Semantik (siehe folgenden Abschnitt) definiert und implementiert werden. Oder man muss zusätzliche Tasks definieren, um das gewünschte Ergebnis zu erhalten.

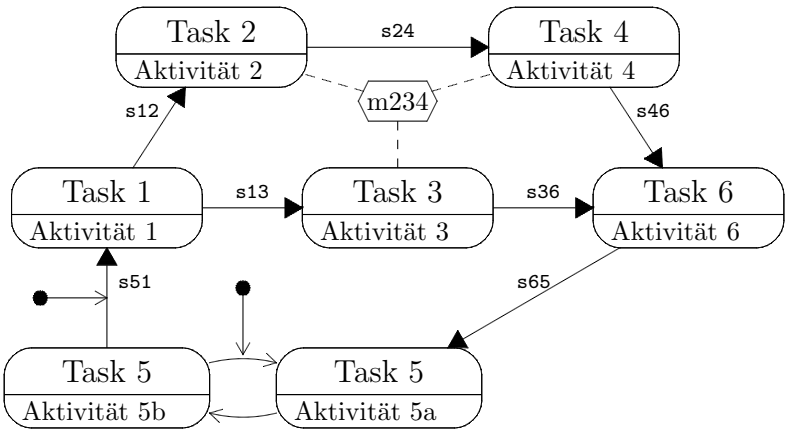
Im oben angegebenen Beispiel würden die Semaphore `s12`, `s13`, `s24`, `s36`, `s46`, `s65`, `s51` und `s7` definiert werden. Der Semaphor `s51` müsste mit 1, alle andern mit 0 initialisiert werden.

Die Tasks müssten dann folgendermaßen programmiert werden:

```
Task 1:       Task 2:       Task 3:       Task 4:       Task 5:
// ...        // ...        // ...        // ...        // ...
p (s51)       p (s12)       p (s13)       p (s24)       p (s65)
Aktivität 1   Aktivität 2   Aktivität 3   Aktivität 4   Aktivität 5a
v (s12)       v (s24)       v (s36)       v (s46)       v (s7)
v (s13)       // ...        // ...        // ...        Aktivität 5b
// ...        // ...        // ...        // ...        v (s51)
// ...        // ...        // ...        // ...        // ...
```

```
Task 6:       Task 7:
// ...        // ...
p (s36)       p (s7)
p (s46)       Aktivität 7
Aktivität 6   // ...
v (s65)       // ...
v (s7)        // ...
// ...        // ...
```

Bei den Ablauf-Diagrammen werden Bereiche des gegenseitigen Ausschlusses normalerweise nicht berücksichtigt. Wenn man das ebenfalls graphisch darstellen möchte, dann kann man die sich gegenseitig ausschließenden Bereiche gestrichelt mit der betreffenden Kontrollinstanz (etwa `m234`) verbinden. Das kann ein (MUTEX-)Semaphor oder ein Monitor oder sonst ein Lock sein. Im folgenden Beispiel würden sich die Aktivitäten 2, 3 und 4 gegenseitig ausschließen.



Dabei würden die drei Aktivitäten 2, 3 und 4 jeweils in MUTEX-Operationen verpackt werden:

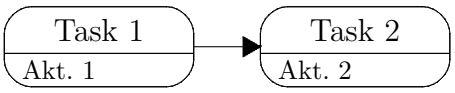
```
p (m234)
Aktivität x
v (m234)
```

Bemerkungen:

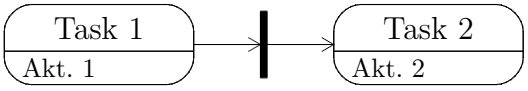
MUTEXe: Da der Einsatz von MUTEXen immer die Gefahr eines Deadlocks birgt, wird neuerdings eher davon abgeraten, diese auf einem niedrigen Code-Niveau einzusetzen. Deswegen werden sie auch schon beim Design selten berücksichtigt.

Task-Rahmen: Beispielprogramme zum Testen solcher Abläufe lassen sich relativ einfach in Java schreiben unter Verwendung eines Task-Rahmens (`www.dhbw-stuttgart.de/~kfg/pdv/taskRahmen.zip`).

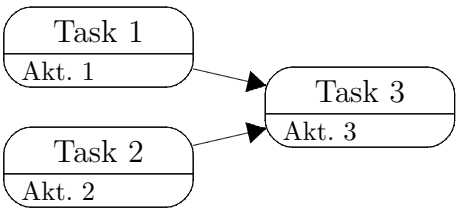
UML-Notation: Als UML-Notation müsste das Aktivitäts-Diagramm verwendet werden:



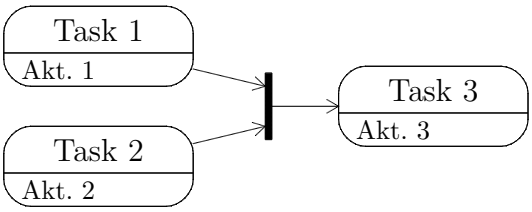
UML-Aktivitäts-Diagramm:



Oder:



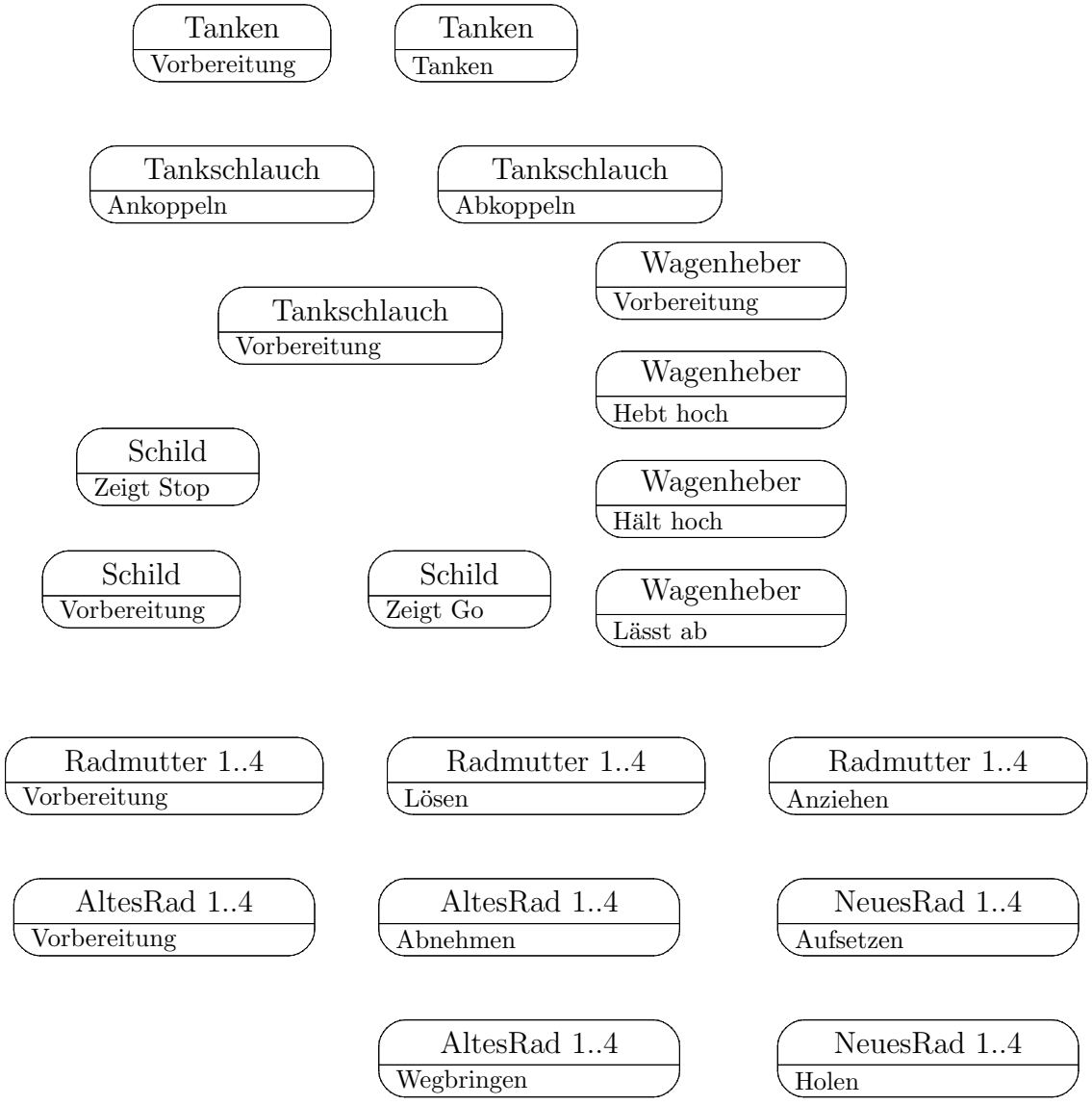
UML–Aktivitäts–Diagramm:



Allerdings werden durch die UML-Notation manche Diagramme ziemlich überladen.

Übung Boxenstop

Vervollständigen Sie das folgende Ablauf-Diagramm durch Pfeile.



Übung Holzhacken

Roboter hacken Holz. Folgende Aktivitäten kann man die Roboter ausführen lassen:

`Holen` (Roboter holt ein zu zerhackendes Holzstück.)
`Vorlegen` (Roboter legt das Holzstück auf dem Hackblock vor.)

`Ausholen` (Roboter holt mit der Axt aus.)
`Hacken` (Roboter schlägt mit der Axt zu.)
`Freimachen` (Roboter macht den Hackblock frei von Holzscheiden.)
`Schichten` (Roboter schichtet Holzscheide auf.)

In jedem Fall gibt es nur **einen** Hackblock!

Entwerfen Sie Ablaufdiagramme für

einen
zwei
drei

Roboter, d.h. Tasks.

6.2.4 Erzeuger-Verbraucher-Problem

Ein wichtiges Element der Kooperation zwischen Tasks ist der Austausch von Daten. Zur Verdeutlichung der Problematik betrachten wir folgendes Beispiel:

Die Task `E` schickt der Task `V` die Koordinatenwerte (x, y, z) der Zielposition eines Roboterarms, indem drei gemeinsame Speicherplätze verwendet werden. Ein Szenario ist:

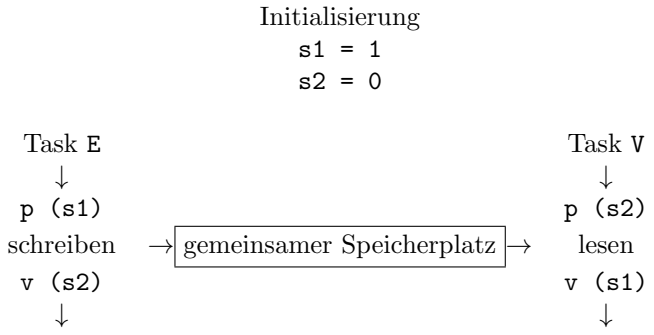
`E` schreibt x_1, y_1, z_1.
`V` liest x_1, y_1.
`V` wird suspendiert.
`E` schreibt neue Werte x_2, y_2, z_2.
`V` liest weiter z_2. $\Longrightarrow$ `V` hat inkonsistenten Datensatz x_1, y_1, z_2.

Vor einer Lösung des Problems muss die Art des Datenaustauschs geklärt werden:

- a) Soll jeder geschriebene (erzeugte) Datensatz auch gelesen (verbraucht) werden?
- b) Sollen dabei Schreiben und Lesen zeitlich entkoppelt werden?
- c) Soll nur der jeweils aktuelle Datensatz (eventuell öfter) gelesen werden?
- d) Soll der jeweils aktuelle Datensatz nur gelesen werden, wenn geschrieben wurde?
- e) Soll das Reader-Writer-Problem gelöst werden (siehe unten)?

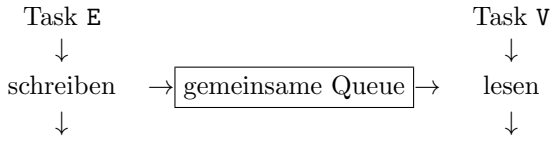
Lösungen mit Semaphoren:

- zu a) Jeder geschriebene (erzeugte) Datensatz soll auch gelesen (verbraucht) werden.



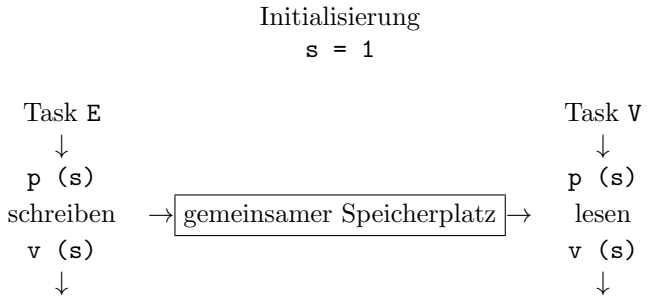
Die beiden Tasks sind zeitlich eng gekoppelt.

- zu b) Schreiben und Lesen sollen zeitlich entkoppelt werden. Zeitliche Entkopplung ist nur möglich über eine Queue:

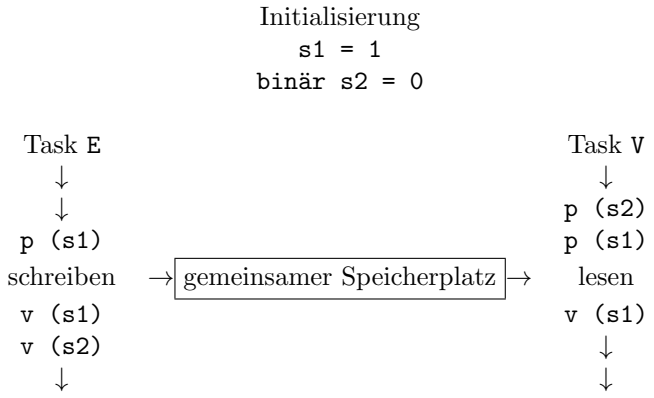


Dazu mehr in einem späteren Abschnitt.

- zu c) Nur der jeweils aktuelle Datensatz soll (eventuell öfter) gelesen werden.



- zu d) Nur der jeweils aktuelle Datensatz soll gelesen werden, wenn geschrieben wurde.

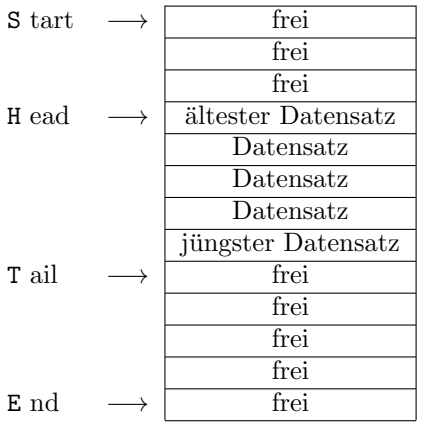


- zu e) Das Reader-Writer-Problem wird in einem späteren Abschnitt behandelt.

Queue

Eine Queue (Ringpuffer, Zirkularpuffer, Pipe, Kanal) ist ein Speicherbereich mit `L` Speicherplätzen `0 ... L-1` für `L` Datensätze und zwei Methoden: Eine Methode `enqueue` (lege ab, schreiben), um Datensätze in die Queue zu schreiben, und eine Methode `dequeue` (entnehme, lesen), um Datensätze aus der Queue zu entnehmen. Die Entnahme-Strategie ist FIFO. Der Zustand der Queue wird durch folgende Größen beschrieben:

`L` : Anzahl der Speicherplätze `0 ... L-1`
`N` : Anzahl der belegten Speicherplätze
`T` : Nummer des ersten freien Speicherplatzes, auf den ein Datensatz abgelegt werden kann ***(Tail***, Schwanz).
`H` : Nummer des ersten belegten Speicherplatzes, aus dem ein Datensatz entnommen werden kann ***(Head***, Kopf).
`S` : (`= 0`) Anfang (Start) der Queue
`E` : (`= L-1`) Ende der Queue



Initialisierung der Queue:

```
H = T = S
N = 0
```

Methode `enqueue (D)` (`D` sei ein Datensatz.):

```
void  enqueue (D)
   {
   Warte bis N < L;
   Schreibe nach T;
   if T == E then T = S;
   else T = T ⊕ 1;
```

```
   N = N + 1;
   }
```

(Mit "$\oplus$" meinen wir eben "zur nächsten Adresse".)

Methode `dequeue ()`:

```
D  dequeue ()
   {
   Warte bis N > 0;
   Lies von H;
   if H == E then H = S;
   else H = H ⊕ 1;
   N = N - 1;
   }
```

Nun wollen wir versuchen, das "Warten" mit Semaphoren zu realisieren. Außerdem muss erreicht werden, dass die Verwaltung von `N` nicht durcheinander kommt, wenn wir nebenläufige Schreiber und Leser haben.

1. Versuch

```
init  binärer Semaphor s = 1;

void  enqueue (D)
   {
   while N == L do nothing;
   Schreibe nach T;
   if T == E then T = S;
   else T = T ⊕ 1;
   p (s);
   N = N + 1;
   v (s);
   }

D  dequeue ()
   {
   while N == 0 do nothing;
   Lies von H;
   if H == E then H = S;
   else H = H ⊕ 1;
   p (s);
   N = N - 1;
   v (s);
```

```
   }
```

`N` ist zwar geschützt, aber 1.) wird aktiv gewartet und 2.) käme die Verwaltung von `H` und `T` durcheinander, wenn wir mehrere Schreiber bzw. Leser hätten.

2. Versuch

```
init  binärer Semaphor s = 1;

void  enqueue (D)
   {
   while N == L do nothing;
   p (s);
   Schreibe nach T;
   if T == E then T = S;
   else T = T ⊕ 1;
   N = N + 1;
   v (s);
   }

D  dequeue ()
   {
   while N == 0 do nothing;
   p (s);
   Lies von H;
   if H == E then H = S;
   else H = H ⊕ 1;
   N = N - 1;
   v (s);
   }
```

Auch hier wird aktiv gewartet. Ferner könnte es sein, dass z.B. bei `N == 1` zwei Leser über die Warte-Anweisung kommen und dann versuchen zwei Datensätze zu entnehmen, obwohl nur einer vorhanden ist. (Das Problem gibt es natürlich auch schon bei Versuch 1.)

3. Versuch

```
init  binärer Semaphor s = 1;

void  enqueue (D)
   {
   p (s);
   while N == L do nothing;
   Schreibe nach T;
   if T == E then T = S;
   else T = T ⊕ 1;
   N = N + 1;
   v (s);
   }
```

```
D  dequeue ()
   {
   p (s);
   while N == 0 do nothing;
   Lies von H;
   if H == E then H = S;
   else H = H ⊕ 1;
   N = N - 1;
   v (s);
   }
```

Das löst zwar alle Probleme des gegenseitigen Ausschlusses. Aber das System ist tot. Denn wenn eine Task mal in einer Warteschleife ist, dann wird jede andere Task durch `p (s)` gesperrt. Wir müssen also immer irgendwie mal eine andere Task in den kritischen Bereich lassen.

4. Versuch

```
init  binärer Semaphor s = 1;

void  enqueue (D)
   {
   p (s);
   while N == L { v (s); do nothing; p (s); }
   Schreibe nach T;
   if T == E then T = S;
   else T = T ⊕ 1;
   N = N + 1;
   v (s);
   }

D  dequeue ()
   {
   p (s);
   while N == 0 { v (s); do nothing; p (s); }
   Lies von H;
   if H == E then H = S;
   else H = H ⊕ 1;
   N = N - 1;
   v (s);
   }
```

Das müsste einigermaßen funktionieren. Allerdings haben wir immer noch eine aktive Warteschleife. Um das zu lösen, führen wir noch zwei weitere binäre Semaphore ein:

5. Versuch

```
init  binärer Semaphor s = 1;
```

```
init  binärer Semaphor s1 = 0;
init  binärer Semaphor s2 = 0;

void  enqueue (D)
   {
   p (s);
   while N == L { v (s); p (s1); p (s); }
   Schreibe nach T;
   if T == E then T = S;
   else T = T ⊕ 1;
   N = N + 1;
   v (s2);
   v (s);
   }

D  dequeue ()
   {
   p (s);
   while N == 0 { v (s); p (s2); p (s); }
   Lies von H;
   if H == E then H = S;
   else H = H ⊕ 1;
   N = N - 1;
   v (s1);
   v (s);
   }
```

Es ist sehr schwer und umständlich zu beweisen, dass dieser Algorithmus tatsächlich funktioniert.

Derartige Probleme können einfacher und sicherer mit dem Monitor-Konzept programmiert werden.

Übung: Welcher Semaphor kann/sollte in den oben genannten Beispielen ein MUTEX sein?

6. Versuch

Wir versuchen hier nochmal eine elegante Lösung mit zwei allgemeinen Semaphoren, die die Verwaltung von `N` übernehmen, und zwei binären Semaphoren, die die Queue-Zeiger-Verwaltung schützen:

```
init  allgemeiner Semaphor sl = L;
   // zählt die leeren Plätze

init  allgemeiner Semaphor sn = 0;
   // zählt die vollen Plätze

init  Binärer Semaphor sh = 1;
init  Binärer Semaphor st = 1;

void  enqueue (D)
```

```
    {
    p (sl);
    p (sh);
    Schreibe nach H;
    if H == E then H = S;
    else H = H ⊕ 1;
    v (sn);
    v (sh);
    }

D   dequeue ()
    {
    p (sn);
    p (st);
    Lies von T;
    if T == E then T = S;
    else T = T ⊕ 1;
    v (sl);
    v (st);
    }
```

6.2.5 Emulation von Semaphoren in Java

Siehe Skriptum Java bzw. Kapitel Threads.

6.3 Bolt-Variable — Reader-Writer-Problem

Reader-Writer-Problem: Es gibt einen gemeinsamen Speicherbereich, für den folgende Regeln gelten sollen:

1. Schreiber haben exklusiven Zugriff. D.h. höchstens ein Schreiber darf schreiben. Wenn ein Schreiber schreibt, darf kein Leser lesen.
2. Viele Leser können gleichzeitig lesen. Wenn Leser lesen, darf kein Schreiber schreiben.
3. Solange Leser lesen, werden neue Leser zum Lesen zugelassen.

Dieses Problem ist mit Semaphoren sehr umständlich zu lösen. Die Sprache PEARL hat deswegen die Bolt-Variable erfunden. Datenbanksyteme kennen X- und S-Locks für die Lösung dieses Problems.

Eine Bolt-Variable `B` hat drei Zustände

```
reserved    (gesperrt)
free        (Sperre möglich)
```

```
entered     (Sperre nicht möglich)
```

und einen Zähler `Z` für die gerade mitbenutzenden Tasks und vier **atomare** und **sich gegenseitig ausschließende** Anweisungen:

```
reserve (B):   aufrufende Task wartet bis B == free
               B = reserved

free (B):      B = free

enter (B):     aufrufende Task wartet bis B != reserved
               B = entered
               Z = Z + 1

leave (B):     Z = Z - 1
               if (Z == 0) B = free
```

Schreiber schützen ihre Schreib-Operation mit der Kombination:

```
reserve (B);
   Schreiben;
free (B);
```

Leser schützen ihre Lese-Operation mit der Kombination:

```
enter (B);
   Lesen;
leave (B);
```

6.3.1 Implementierung mit dem Java-Monitor

Übung Bolt-Variable:

```
import java.io.*;
public interface  BoltVariable
   {
   void  reserve ();
   void  free ();
   void  enter ();
   void  leave ();
   }
```

Realisieren Sie diese Schnittstelle in einer Klasse `BoltVariableI`.

6.3.2 Reader-Writer-Problem mit Writer-Vorrang

Eine meist sinnvolle Variante des Reader-Writer-Problems ist es, den Schreibern Vorrang zu geben. D.h., wenn ein Schreiber schreiben will, wird kein weiterer Leser zugelassen.

Die Lösung nur mit Semaphoren wird hier schon recht kompliziert.

Wenn man allerdings die Bolt-Variable verwendet, dann ist man auch noch auf Semaphore angewiesen und es wird eigentlich zu komplex.

Insgesamt sind Bolt-Variable kein nützliches Konzept und wurden hier eigentlich nur behandelt, um das Reader-Writer-Problem vorzustellen, das ein interessantes Beispiel für den Einsatz von Semaphoren, Monitoren und anderen Synchronisationsmechanismen ist.

6.4 Monitore

Da Semaphoroperationen entfernte Wirkungen haben können, widerspricht dies dem Konzept der Strukturierung. Räumlich und zeitlich im Entwicklungsprozess eines Programms voneinander entfernte Prozeduren können Semaphoroperationen benützen. Das Semaphor gilt als das "Goto" der nebenläufigen Programmierung.

Der Monitor dagegen ist ein *strukturiertes* Synchronisationswerkzeug, mit dem ein Programmbereich definiert wird, der zu einer Zeit nur von *einer* Task oder *einem* Prozess betreten bzw. benutzt werden darf. Damit kann z. B. gegenseitiger Ausschluss implementiert werden.

Bei einem *monolithischen* Monitor werden alle kritischen Phasen in *einem* Monitor zentralisiert. Bei *dezentralisierten* Monitoren hat jeder Monitor eine spezielle Aufgabe mit geschützten Daten und Funktionen.

Ein Monitor hat folgende Struktur:

```
Monitor
   Folge von Datenvariablen
   Folge von Prozeduren
   Körper zur Initialisierung
```

Auf die im Monitor definierten Daten kann nur über die Prozeduren des Monitors zugegriffen werden. Der Körper wird bei Start des Programms einmal durchlaufen. Jedem Monitor sind in Pascal-S ein oder mehrere Bedingungsvariablen `c` zugeordnet, auf die mit den Proceduren `wait (c)`, `signal (c)` und `nonempty (c)` zugegriffen werden kann:

`wait (c)` : Der aufrufende Prozess wird blockiert und in eine Warteschlange (FIFO-Strategie) von Prozessen eingeordnet, die ebenfalls wegen `c` blockiert sind. Die Ausführung von `wait (c)` gibt den Eintritt in den Monitor wieder frei.

`signal (c)` : Falls die Warteschlange von `c` nicht leer ist, wird der erste Prozess in der Warteschlange von `c` erweckt. Danach muss der Monitor sofort verlassen werden. "Ich signalisiere, dass `c` eingetreten ist."

`nonempty (c)`: Ist `true`, falls die Warteschlange nicht leer ist.

Da Monitore das Synchronisationskonzept in Java sind, findet sich eine weitergehende Diskussion mit Beispielen im Java-Skriptum bzw. im Kapitel Threads.

6.5 Rendezvous

Das Rendezvous basiert auf Ideen von Hoare[14] und ist der Synchronisationsmechanismus, den die Echtzeitsprache **Ada** zur Verfügung stellt.

Ada entstand als ein Abkömmling von Pascal im Auftrag des Department of Defense der USA zwischen 1975 und 1980 in Konkurrenz zwischen mehreren Firmen. Durchgesetzt hat sich die Gruppe um Jean Ichbiah von CII Honeywell Bull in Frankreich. Der erste Standard war ANSI MIL-STD-1815. Benannt ist die Sprache nach Augusta Ada Byron, Countess of Lovelace, die als die welterste ProgrammiererIn gilt und mit Charles Babbage zusammenarbeitete.

Das Rendezvous-Konzept wird unter der Bezeichnung ***Transaction*** auch in der von Gehani entwickelten nebenläufigen Erweiterung von C/C++ ConcurrentC/C++ (CCC) als Synchronisationsmechanismus verwendet. Wir verwenden einen Pseudocode, der an die Syntax und Semantik von CCC angelehnt ist, um die grundlegenden Mechanismen des Rendezvous vorzustellen.

Das Rendezvous ist entwickelt worden für eine typische Client/Server-Situation (im generischen Sinn). Dazu ein Beispiel:

Ein Friseur bietet den Dienst `Haarschneiden` an. Er ist allein und kann daher nur einen Kunden gleichzeitig bedienen. Wenn es mehrere Kunden gibt, dann müssen diese warten. Andererseits, wenn es keine Kunden gibt, dann muss er warten. Mit Rendezvous bzw. Transaction würden wir das folgendermaßen programmieren:

```
process body Friseur ()
   {
   while (true)
      {
      // Bereitet sich auf den nächsten Kunden vor

      accept Haarschneiden (Wunsch)
         {
         // Versucht den Wunsch zu erfüllen.
         // Überlegt sich den Preis, den er verlangen kann.
         Preis = "14,50";
         treturn Preis;
         }  // end accept

      // Kehrt die Haare zusammen.
      }  // end while
   }  // end process

process body Kunde (Friseur friseur)
   {
   char* Wunsch;
   char* Preis;
```

```
   // Überlegt sich, welche Haartracht er wünscht.
   Wunsch = "top modern";

   Preis = friseur.Haarschneiden (Wunsch);

   // Verarbeitet den Preis und Haarschnitt.
   }
```

`Friseur` und `Kunde` sind beide nebenläufig laufende Tasks (in CCC `process` genannt). Das Rendezvous ist ein Mechanismus, bei dem die beiden Tasks an der sogenannten `accept`-Prozedur – hier Haarschneiden – aufeinander warten. Charakteristisch ist die **Unsymmetrie**. Der Kunde muss bei `Haarschneiden` warten, bis der Friseur an das `accept Haarschneiden` gekommen ist. Umgekehrt muss der Friseur am `accept Haarschneiden` warten, bis der Kunde `Haarschneiden` aufruft. Für den Kunden verhält sich das Rendezvous ähnlich wie ein Funktionsaufruf. Er wird allerdings währenddessen (d.h. während er auf die Durchführung der Prozedur warten muss und während der Durchführung selbst) suspendiert. Der Friseur macht die eigentliche Arbeit in seiner Umgebung und in seinem Task-Kontext.

Die `accept`-Prozedur kann Argumente und einen Rückgabewert haben.

Mit dem einfachen Rendezvous kann schon das Problem des gegenseitigen Ausschlusses realisiert werden. Typischerweise gibt es viele Kunden, für die das Haarschneiden unter gegenseitigem Ausschluss durchgeführt wird.

Ein Server kann durchaus mehrere `accept`-Routinen anbieten:

```
process body Friseur ()
   {
   while (true)
      {
      // Bereitet sich auf den nächsten Kunden vor

      accept Haarschneiden (Wunsch)
         {
         treturn Preis;
         }  // end accept

      // Kehrt die Haare zusammen.

      accept Rasieren ()
         {
         treturn Preis;
         }  // end accept

      // Reinigt das Waschbecken.
      }  // end while
   }  // end process
```

Man kann damit gewisse Reihenfolgen von Operationen, die unter gegenseitigem Ausschluss

durchzuführen sind, fest vorgeben. In unserem Beispiel ist das nicht besonders sinnvoll, da der Friseur unbedingt zwischen einem Haarschneide-Kunden und einem Rasier-Kunden abwechseln müsste. Aber in dem Beispiel des Schweiss-Roboters wäre das durchaus sinnvoll, wenn man das Positionieren und Schweissen in einer Server-Task als `accept`-Routinen unterbringen würde. Dann wäre damit die Reihenfolge und der gegenseitige Ausschluss garantiert.

Aber unser Friseur hätte gern die Möglichkeit, unter den `accept`s auswählen zu können. Dafür gibt es den `select`-Mechanismus:

```
process body Friseur ()
   {
   while (true)
      {
      // Bereitet sich auf den nächsten Kunden vor

      select
         {
         accept Haarschneiden (Wunsch) {treturn Preis;}
         or accept Rasieren () {treturn Preis;}
         or accept Bartschneiden () {treturn Preis;}
         or delay 1 Minute ; LiestKurzZeitung ();
         }

      // Räumt auf.
      }  // end while
   }  // end process
```

Der Friseur hat jetzt seinen Synchronisationspunkt am `select`. Dort schaut er, ob es einen Kunden in der Warteschlange gibt, der einen Dienst aufgerufen hat. Wenn es einen solchen Kunden gibt, dann wird er diesen Kunden bedienen. Oder er wartet höchstens eine Minute, ob innerhalb dieser Minute ein Kunde kommt. Wenn kein Kunde kommt, geht er in die `delay`-Alternative und liest etwas Zeitung, ehe er dann aufräumt und sich wieder auf den nächsten Kunden vorbereitet, was beides ja dann recht kurz sein kann oder entfällt.

Das Rendezvous in CCC bietet noch mehr Möglichkeiten, die ohne weiteren Kommentar im Folgenden zusammengefasst sind.

Accept-Statement :

`accept` Transaktionsname`(`Argumente$_{opt}$`)`
`suchthat(`Bedingungsausdruck`)`$_{opt}$
`by (`Prioritätsausdruck`)`$_{opt}$
Block (Körper des Accept)

Ohne `suchthat` und `by` wird eine FIFO-Warteschlange der wartenden Transaktionen gebildet.

Mit `by` werden die Prioritätsausdrücke für alle wartenden Transaktionen ausgewertet. Die Transaktion mit dem *Minimum* wird angenommen.

Mit `suchthat` werden nur solche Transaktionen berücksichtigt, deren Bedingungsausdruck ungleich Null ergibt.

Select-Statement :

```
select
   {
      (Bedingung-1):opt     Alternative-1
   or (Bedingung-2):opt     Alternative-2
   ...
   or (Bedingung-n):opt     Alternative-n
   }
```

Die Alternativen sind eine Folge von Statements. Das erste Statement einer Alternative bestimmt die Art der Alternative. Es werden vier Arten von Alternativen unterschieden:

- `accept`-Alternative: `accept`-Statement eventuell gefolgt von anderen Statements.
- `delay`-Alternative: `delay`-Statement eventuell gefolgt von anderen Statements.
- `terminate`-Alternative: `terminate;`
- `immediate`-Alternative: alle anderen Statements

Die Bedingungen entscheiden, ob eine Alternative offen oder geschlossen ist. Mindestens eine Bedingung muss offen sein. Die Reihenfolge der Alternativen spielt keine Rolle.

Select-Mechanismus:

1: *if* es eine offene `accept`-Alternative gibt *and* ein Transaktionsaufruf dafür vorliegt *and* eine mögliche `suchthat`-Bedingung erfüllt ist, *then* nimm diese Alternative.

2: *else if* es eine offene `immediate`-Alternative gibt, *then* nimm diese.

3: *else if* es keine offenen `delay`- oder `terminate`-Alternativen gibt, *then* warte auf Transaktionsaufrufe für eine der offenen `accept`-Alternativen.

4: *else if* es eine offene `delay`-Alternative (z.B. `delay d`), aber keine offene `terminate`-Alternative gibt, *then* wenn eine akzeptable Transaktion innerhalb von `d` Sekunden kommt, nimm die entsprechende `accept`-Alternative, sonst nimm `delay`-Alternative. (Bei mehereren `delay`-Alternativen nimm die mit minimalem `d`.)

5: *else if* es eine offene `terminate`-Alternative, aber keine offene `delay`-Alternative gibt, *then* warte auf das erste der folgenden Ereignisse:

- **5a:** akzeptable Transaktion. Nimm dann entsprechende `accept`-Alternative.
- **5b:** Das Programm gerät in einen Zustand, wo alle Prozesse entweder *completed* sind oder an einem Select-Statement mit offener `terminate`-Alternative warten. In diesem Fall beende das ganze Programm.

6: *else* (es gibt offene `delay`- und offene `terminate`-Alternativen.) sind zwei Implementierungsstrategien möglich:

- **6a:** ignoriere offenes `terminate` und verfahre wie unter **4**.
- **6b:** Warte auf das erste der folgenden Ereignisse:
  - **6b1:** akzeptable Transaktion. Nimm dann entsprechende `accept`-Alternative.
  - **6b2:** Es passiert **5b**. Verfahre dann wie unter **5b**.
  - **6b3:** Zeitlimit läuft aus. Nimm `delay`-Alternative.

6.6 Channels

Kommunikationskanäle – ***Channels*** – können unter den meisten Sprachen und Betriebssystemen als Kommunikations-Mittel zwischen Tasks eingesetzt werden, da die meisten Sprachen oder Betriebssyteme Queues anbieten. Auch das Rendezvous basiert letzten Endes auf Channels.

Die Sprache Go bietet Channels als schlechthin **die** (einzige) Synchronisations-Möglichkeit an, und die Mechanismen sind besonders bequem und expressiv.

Diese Mechanismen müssen meistens **kooperativ** eingesetzt werden.

Das Ada-Rendezvous basiert ja auf dem Konzept des Austauschs von Botschaften (Hoare[14]). In Go kann dieses Konzept mit Channels wesentlich "leichtgewichtiger" und klarer realisiert werden.

Die Zugriffe (Schreiben, Lesen) auf Channels sind immer **thread-sicher**. Es kann nicht zu Race-Conditions kommen.

In Channel Schreiben: `x ->kanal`
Aus Channel Lesen: `x <-kanal`

Ist ein Channel voll, muss das Schreiben warten. Ist ein Channel leer muss das Lesen warten.

Entscheidend ist der `select`-Mechanismus, der als "Mehrweg-Nebenläufigkeits-Schalter" (***multi way concurrent control switch)*** verwendet werden kann.

```
select {
   case y ->kanal0: // mach was
   case <-kanal1: // mach was
   case x <- kanal2: // mach was mit x
   case <-time.After(3.5*time.Second): // mach was
   default: // mach was
}
```

Die Programm-Ausführung wartet am `select` bis einer der Fälle zutrifft. Wenn der Fall `default` dabei ist, trifft dieser immer zu und es wird bei `select` nicht gewartet. Allerdings haben alle anderen Fälle Priorität, d.h. wenn ein `case` zutrifft, also ein Channel beschreibbar oder lesbar oder eine Zeitdauer abgelaufen ist, dann wird dieser Fall ausgeführt.

`case <-kanal`-Alternativen können geschlossen werden, indem die Channel-Variable `kanal` auf `nil` gesetzt wird.

Golang bietet auch MUTEX-Semaphore an, die allerdings **kein *Owner-ship***-Konzept kennen. Das bedeutet, dass eine Task, die ein Semaphor beantragt und erhalten hat, es nicht ein weiteres Mal bekommen kann. Eine weitere Beantragung für also unweigerlich zum Deadlock.

Die MUTEX-Lock-Operation ist also unter Golang nicht *re-entrant* bzw. nicht *recursive*.

Obwohl die Szenarien, die zeigen würden, dass MUTEX-Reentrancy zu Problemen führen, sehr kompliziert sind, bedeutet das doch, dass man MUTEXe **re-entrant** verwenden sollte !

In folgendem Szenario verlässt sich die Funktion `G()` eventuell auf gewisse Invarianten, die aber von Funktion `F()` in ihrem ersten Codestück vorübergehend verletzt wurden, im zweiten Codenstück natürlich wieder hergestellt würden.

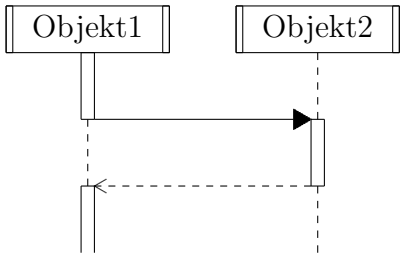
```
func F() {
   mu.Lock()
   ... do some stuff ...      // erstes Codestück
   G()
   ... do some more stuff ... // zweites Codestück
   mu.Unlock()
}

func G() {
   mu.Lock()
   ... do some stuff ...
      // verlässt sich auf Invarianten,
      // deren Invarianz von F() temporär
      // verletzt wurde.
   mu.Unlock()
}
```

6.7 Message Passing

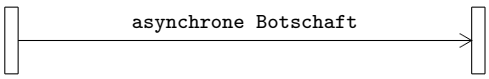
Das Rendezvous in Ada oder die Transaction in Concurrent C beruhen auf einem ***Message Passing*** Konzept. Daher werden bei dieser Gelegenheit im Folgenden die beiden Message Passing Konzepte **Synchrones** und **Asynchrones** Message Passing (sync-MP und async-MP) diskutiert.

Synchrone (blocking) Message Passing :



Objekt 1 schickt eine Botschaft an Objekt 2 und wartet auf eine Antwort.

Asynchrone (non-blocking) Message Passing :



Objekt 1 schickt eine Botschaft an Objekt 2 und wartet **nicht** auf eine Antwort, sondern macht sofort weiter. Eine eventuelle Antwort wäre dann wieder eine Asynchrone Botschaft von Objekt 2 an Objekt 1.

Synchrone (blocking) Message Passing : (Vorteile, Merkmale)

- Bidirektionaler Botschaftenaustausch.
- Verständlicher als async-MP. Async-MP verhält sich oft tückisch wegen unübersichtlicher Zeitabhängigkeiten. Sync-MP verhält sich dagegen wie ein Funktionsaufruf.
- Keine implementationsabhängige Semantik, d.h. synchron ist immer synchron, während asynchron auch manchmal synchron wird, wenn etwa die Botschaften-Queue voll ist.
- Fehlermeldungen sind ganz natürlich.
- Timeouts sind leicht implementierbar.
- Benutzbar als Synchronisationsmechanismus. Bei async-MP ist Polling nötig.
- Programmierfehler sind leichter zu finden. Bei async-MP treten Fehler häufig nur unter Stress auf.

Asynchrone (non-blocking) Message Passing **oder** *Message Queue* : (Vorteile, Merkmale)

- Maximiert Nebenläufigkeit (Concurrency).
- Botschaftenkanal ***(Message Queue)*** mit der Möglichkeit der Auswahl von Botschaften zwecks Optimierung. Oft ist es möglich, den Botschaften Prioritäten ("NORMAL" und "URGENT") zuzuweisen und damit das einfache FIFO-Verhalten zu unterlaufen.
- Expressiveness: z.B. Server kann erst andere Klienten bedienen, wenn für einen Klienten nicht alle benötigten Resourcen zur Verfügung stehen. Bei sync-MP gibt es die sogenannte "early reply"-Technik (Wäsche-Bon, laundry ticket).
- Das synchrone Protokoll kann leicht simuliert werden. Simulation von async-MP mit sync-MP ist umständlicher: man benötigt einen Buffer-Prozess (für die Queue).
- Eine Message Queue transportiert individuelle Botschaften. Der Begriff ***Pipe*** steht für einen Zeichenstrom zwischen zwei Tasks. Eine Pipe wird wie ein I/O-Device behandelt und stellt `read/write`-Methoden zur Verfügung. Für Pipes ist eine Select-Anweisung realisierbar. D.h. eine Task kann alternativ auf eine Menge von I/O-Devices warten.

Ungefähre Leistungsdaten:

Relative Zeiten	Uni-prozessor	Multi-prozessor
Synchron	1	100
Asynchron	10	30
Sim.Async	2	110

6.8 Verteilter gegenseitiger Ausschluss

Man unterscheidet drei Algorithmen:

- **Zentrale Verwaltung:** Ein Knoten übernimmt die Verwaltung eines Semaphors. Er heißt **Koordinator**. Bei ihm müssen alle Tasks, die einen kritischen Bereich betreten wollen, anfragen, ob das möglich ist.

Konzeptionell sind zentrale Konzepte oder zentrale Koordination bei verteilten Systemen verpönt. Aber obwohl die nächsten beiden Konzepte als nicht "zentral" gelten, kommen sie ohne eine gewisse zentrale Koordination mindestens in der Initial-Phase nicht aus.

- **Token-Ring:** Bei der Initialisierung bekommt eine Task das Token, d.h. die Erlaubnis, einen kritischen Bereich zu betreten. Außerdem muss jede Task die im Token-Ring nachfolgende Task kennen.

  Wenn eine Task das Token hat, dann kann sie entweder den kritischen Bereich betreten oder sie muss das Token an die folgende Task weitergeben.

- **Eintrittskarte:** Hierbei sendet eine Task, die einen kritischen Bereich betreten will, an alle Konkurrenten einen Betretungswunsch und wartet dann, bis alle Konkurrenten diesen Wunsch erlauben. Dann kann sie den kritischen Bereich betreten. Nach Verlassen des kritischen Bereichs wird an alle Tasks, die ebenfalls warten, eine Erlaubnis geschickt. Algorithmus:

  1. Betretungswunsch an alle Konkurrenten senden.
  2. Warten bis alle Konkurrenten die Erlaubnis erteilen. (Wenn dies mit einer Task realisiert wird, dann heißt diese Task **Reply-Task**.) Währenddessen muss auf Betretungswünsche anderer Prozesse reagiert werden. Das wird mit einer weiteren Task **(Request-Task)** realisiert, die diese Prozesse als "verschoben" registriert, aber die sonst (d.h. wenn man selbst nicht wartet) diesen Prozessen eine Erlaubnis erteilt. D.h. die Request-Task muss immer laufen.
  3. Wenn alle anderen interessierten Tasks entweder eine Erlaubnis erteilt haben oder "verschoben" sind, dann kann der kritische Bereich betreten werden.
  4. Kritischen Bereich verlassen und dann alle "verschobenen" Konkurrenten informieren, d.h. eine Erlaubnis schicken.

  Das Problem hier ist, dass die Intertask-Kommunikation sehr hoch ist. Außerdem muss zentral geregelt werden, wer zu den Konkurrenten gehört.

6.9 Intertask-Kommunikation

Zur Intertask- oder Interprozess-Kommunikation werden ***Message Queues*** und ***Pipes*** eingesetzt.

Message Queue: Das ist eine Queue, deren Speicherplätze durch ganze Objekte, nämlich Botschaften belegt werden. Es werden typischerweise die Methoden

```
send (:Message)
```
und
```
receive () :Message
```

bzw. nach modernem Standard

```
enqueue (:Message)
```
und
```
dequeue () :Message
```

zur Verfügung gestellt.

Pipe: Das ist eine Queue, deren Speicherplätze Bytes oder Chars sind. Eine Pipe repräsentiert einen Datenstrom. Es werden typischerweise die Methoden

```
write (anzBytes, :byte[])
```
und
```
read (anzBytes, :byte[]) :int
```

zur Verfügung gestellt. Die `read`-Methode gibt die tatsächlich gelesenen Bytes zurück. D.h. es können durchaus weniger als `anzBytes` gelesen werden. Die Methode blockiert nur, wenn der Puffer ganz leer ist.

Abhängig vom System werden zusätzliche Methoden oder unterschiedliches Verhalten angeboten.

6.10 Diskussion

Semaphore und Monitore sind elementare Konzepte. Semaphore sind weiter verbreitet.

Monitore sind strukturiert und damit wesentlich sicherer. Semaphore sind das "Goto" der nebenläufigen Programmierung. Entfernte Prozeduren können Semaphore benützen. Das widerspricht dem Konzept der Strukturierung.

Dijkstra: "Was ist Software-Engineering? How to program if you can't."

Kapitel 7

Petri-Netze

Obwohl wir Petri-Netze als nicht besonders nützlich für die Entwicklung von Echtzeitsystemen erachten, seien sie hier der Vollständigkeit halber vorgestellt (und auch weil sie ein nettes Spielzeug sind).

Carl Adam Petri hat 1962 in einer Veröffentlichung die dann nach ihm benannten Petri-Netze vorgestellt. Seither wurden verschiedene Varianten entwickelt, von denen hier eine vorgestellt wird[33].

Petri-Netze sind ein Mittel zur Modellierung paralleler Abläufe.

7.1 Netzregeln und Notation

Ein Petri-Netz besteht aus

- **Stellen** (Kreise, Ovale) und
- **Transitionen** (Rechtecke) und
- **Pfeilen**, die
  - entweder von einer Stelle zu einer Transition gehen
  - oder von einer Transition zu einer Stelle gehen.
- Stellen können **Marken** (fette Punkte) besitzen.
- Zustandsänderungen erfolgen durch das **Schalten** von Transitionen. Eine Transition t kann schalten, wenn sich in allen Stellen, von denen ein Pfeil nach t geht, mindestens eine Marke befindet. Durch das Schalten wird aus all diesen Stellen eine Marke entfernt und eine Marke den Stellen hinzugefügt, zu denen ein Pfeil von t aus geht. (Es kann immer nur eine Transition schalten.)
- Zwei Transitionen stehen im **Konflikt**, wenn sie mindestens eine eingehende Stelle teilen und beide schalten können.

- Es kann Transitionen geben, die keine **ein**gehenden Pfeile haben. Diese Transitionen können immer schalten. Sie sind sogenannte Markengeneratoren.

- Es kann Transitionen geben, die keine **aus**gehenden Pfeile haben. Wenn diese Transitionen schalten, dann vernichten sie Marken.

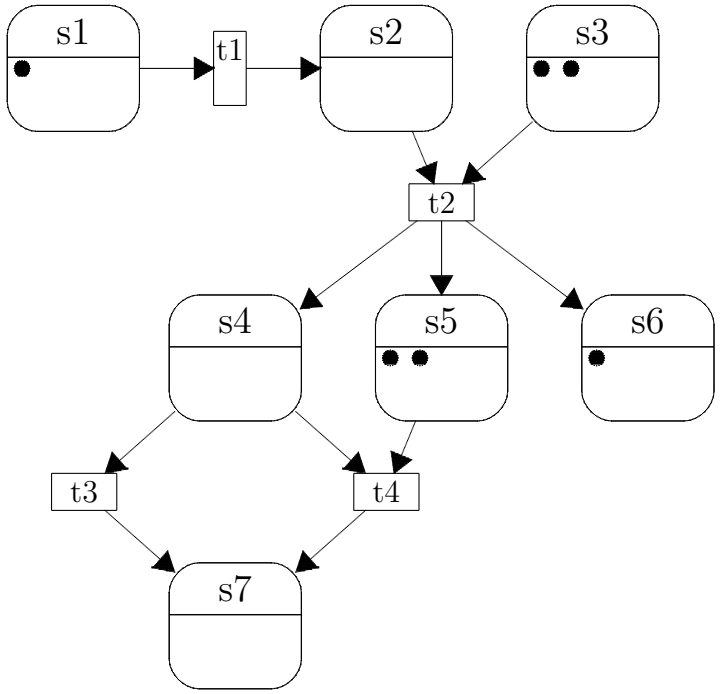
- Die Reihenfolge, in der Transitionen, die schalten können, schalten, ist nicht festgelegt. Damit ist ein Petri-Netz nicht deterministisch. D.h. es gibt eben verschiedene Szenarien.

  Konflikte müssen irgendwie behandelt werden. Z.B. kann man bei Konflikten Prioritäten vergeben.

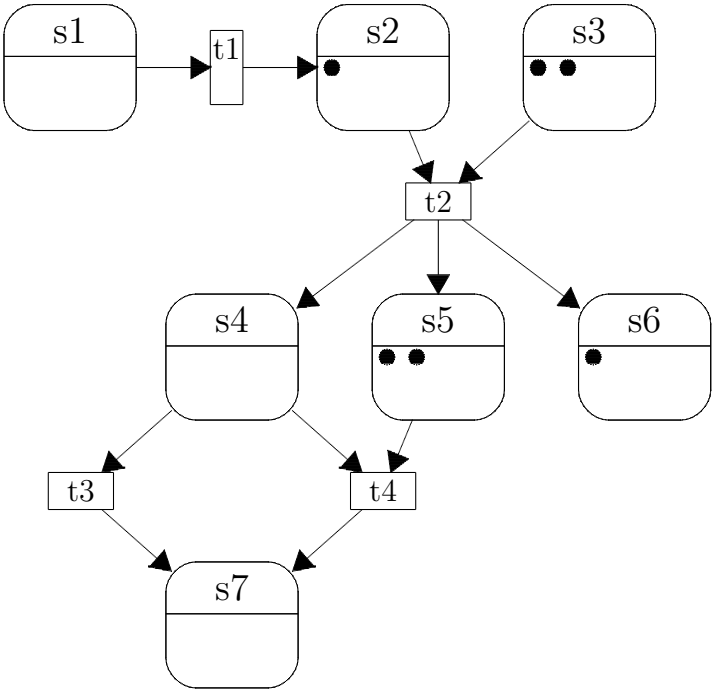
- Variante: Die Kapazität (maximale Anzahl von Marken) einer Stelle kann beschränkt sein. Eine Transition, die solch einer Stelle eine Marke zuführen würde, kann nicht schalten.

- Variante: Ein Pfeil kann eventuell mehr als eine Marke transportieren.

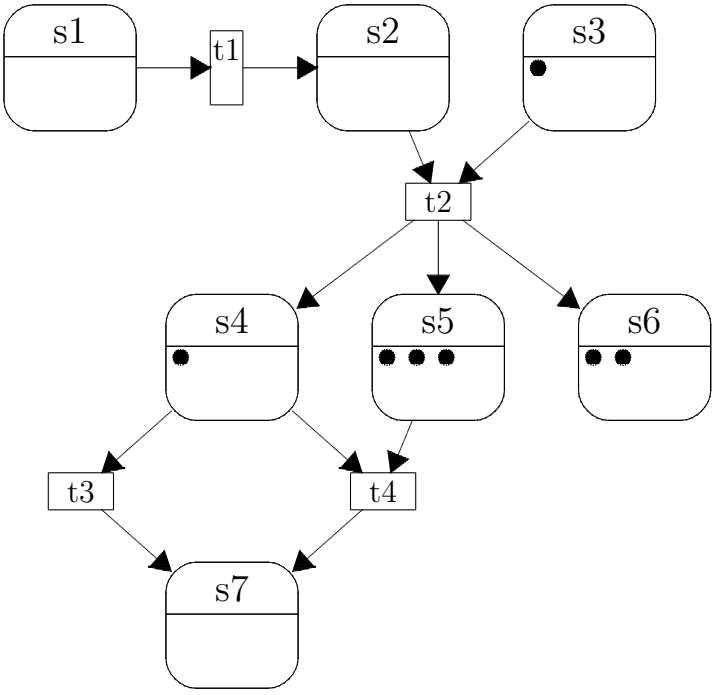
Als Beispiel betrachten wir folgendes Petri-Netz:



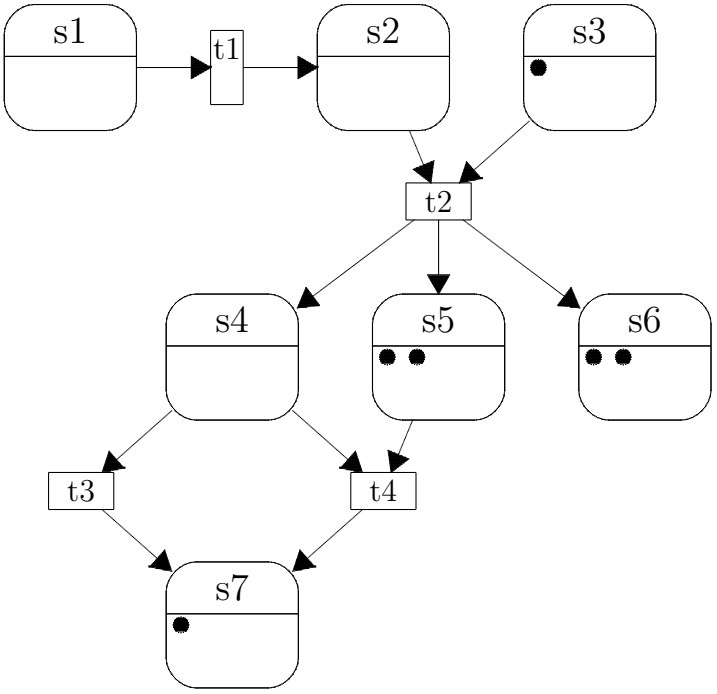
Als einzige Transition kann t_1 schalten. Das ergibt:



Jetzt kann Transition t_2 schalten. Das ergibt:



Jetzt können t_3 und t_4 schalten. Es kommt zum Konflikt. Wir entscheiden uns für t_4. Das ergibt:



Wir bilden Code folgendermaßen auf Stellen und Transitionen ab:

- Stellen sind Punkte vor Code-Stücken oder Prozedur-Aufrufen, die alle mit Marken belegt sein müssen, damit die Code-Stücke bzw. Prozeduren ausgeführt werden können. Wenn z.B. vor Betreten eines Code-Stücks ein oder mehrere Bedingungen erfüllt sein müssen, auf die gewartet werden muss, dann lässt sich das durch ein oder mehrere Stellen realisieren.

  Stellen sind Wartebedingungen. Es wird an der Stelle gewartet, bis die Stelle mindestens eine Marke hat. (Eventuell muss auf mehr als eine Stelle gewartet werden.)

- Transitionen sind Code-Stücke oder Prozeduren. Es werden Pfeile von den vor der Transition liegenden Stellen zu der betreffenden Transition gezogen.

- Wenn nach einer Transition eine Wartebedingung aufgehoben wird, dann wird ein Pfeil von dieser Transition zur entsprechenden Stelle gezogen.

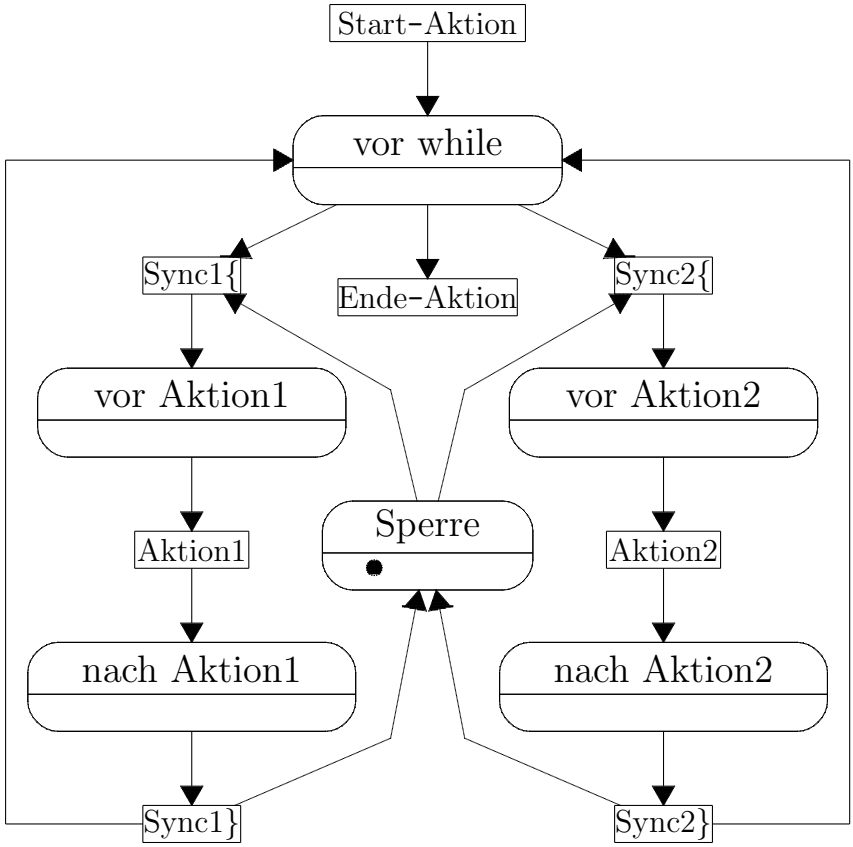
7.2 Beispiel Java-Monitor

Als Beispiel zeigen wir die Realisierung des gegenseitigen Ausschlusses mit dem Monitor-Konzept von Java. Der Java-Code hat folgende Form:

```
public class   Task extends Thread
   {
   private static Object   sync = new Object ();
      // Wird zum synchronisieren verwendet.
   private static java.util.Random  zufall = new java.util.Random ();
   public void run ()
```

```
    {
    System.out.println ("Start-Aktion");         // Transition: "Start-Aktion"
    boolean weiter = true;
    while (weiter)                               // Stelle: "vor while"
       {
       switch (zufall.nextInt (2))
          {
          case 0:
             synchronized (sync)                 // Transition: "Sync0{"
                {                                // Stelle: "vor Aktion0"
                System.out.println ("Aktion0");  // Transition: "Aktion0"
                weiter = false;                  // Stelle: "nach Aktion0"
                }                                // Transition: "Sync0}"
          case 1:
             synchronized (sync)                 // Transition: "Sync1{"
                {                                // Stelle: "vor Aktion1"
                System.out.println ("Aktion1");  // Transition: "Aktion1"
                                                 // Stelle: "nach Aktion1"
                }                                // Transition: "Sync1}"
          case 2:
             synchronized (sync)                 // Transition: "Sync2{"
                {                                // Stelle: "vor Aktion2"
                System.out.println ("Aktion2");  // Transition: "Aktion2"
                                                 // Stelle: "nach Aktion2"
                }                                // Transition: "Sync2}"
          }
       }
    System.out.println ("Ende-Aktion");          // Transition: "Ende-Aktion"
    }
 public static void   main (String[] arg)
    {
    for (int i = 0; i < 10; i++) new Task ().start ();
    }
 }
```

Dieses Programm kann mit folgendem Petri-Netz modelliert werden: (Der ”0”-Zweig wurde der Übersichtlichkeit halber weggelassen, sieht aber wie die anderen Zweige aus.)



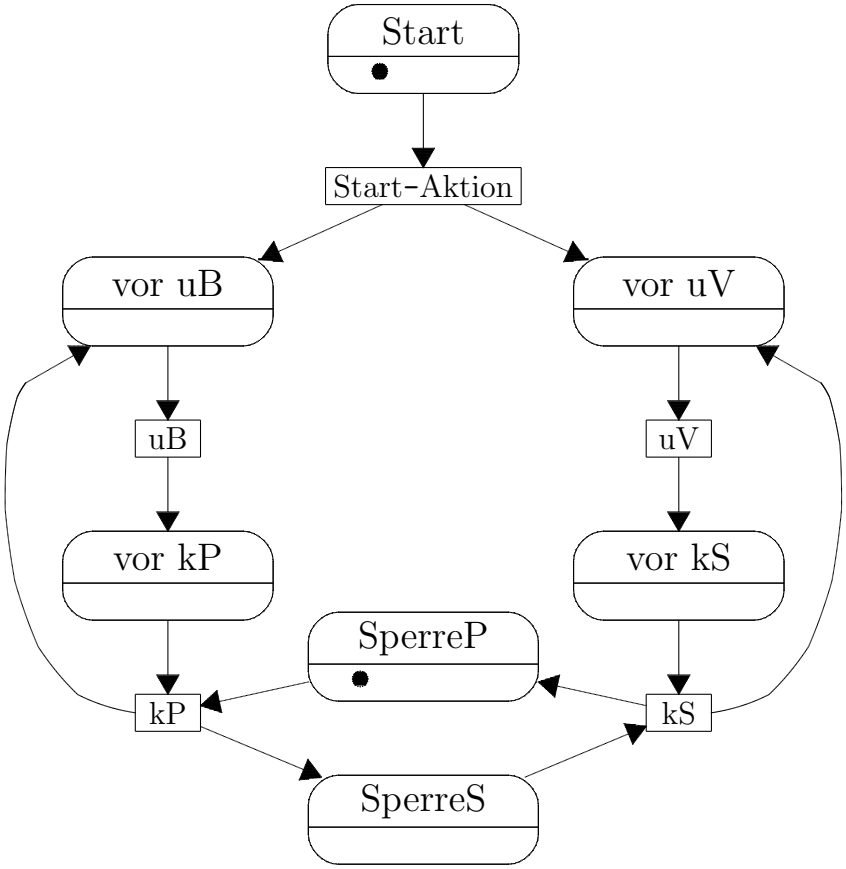
Die einzige Transition, die in diesem Zustand schalten kann, ist "Start-Aktion". Sie kann immer schalten. Dies entspricht einer Generierung von Threads. Sie liefert dabei Marken in die Stelle "vor while". Dadurch kann dann die Transition "Sync{" schalten, wenn die Stelle "Sperre" mit einer Marke belegt ist.

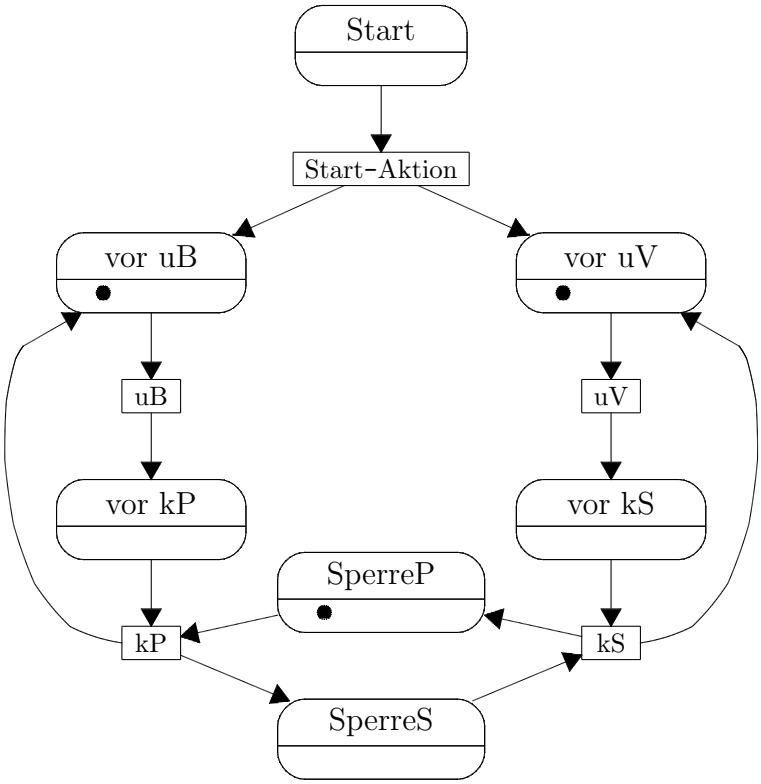
Dass die Sperre mit einer Marke initialisiert ist, bedeutet, dass die eine Task den geschützten Bereich betreten kann (Schalten von Transition "SyncX"). Dabei verliert die Sperre ihre Marke. Die nächste Task muss warten, bis eine Transition "SyncX" geschaltet hat und dadurch der Sperre wieder eine Marke gegeben hat.

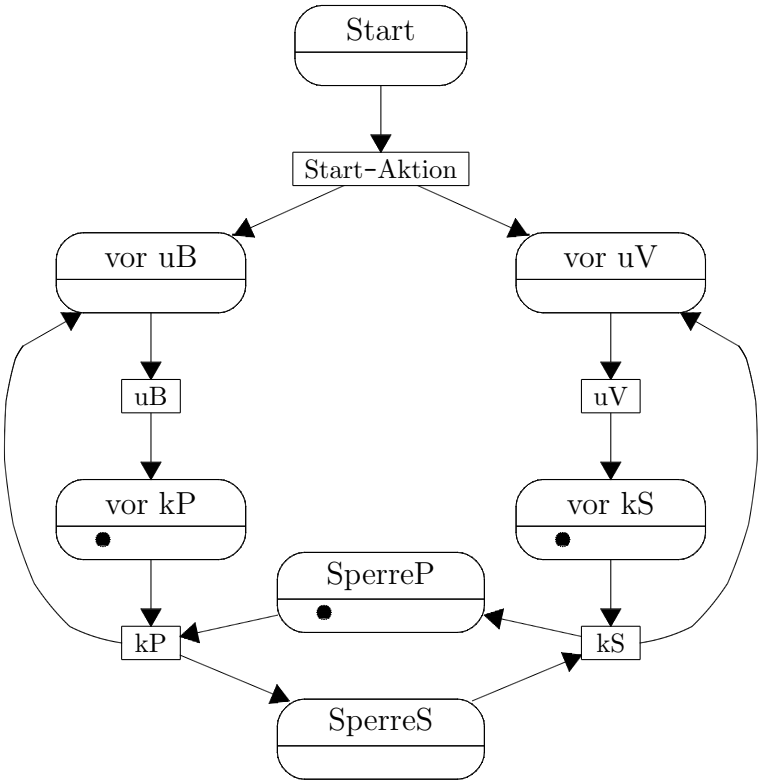
Es ist sehr umständlich, diese Petri-Netze von Hand durchzuschalten. Ein Simulator ist hier unerlässlich.

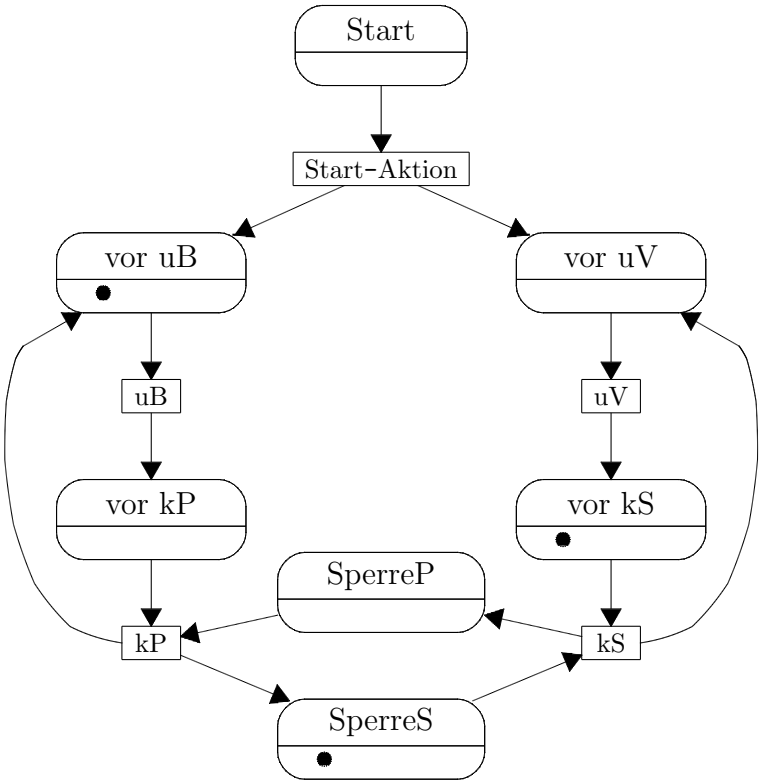
7.3 Beispiel Schweissroboter

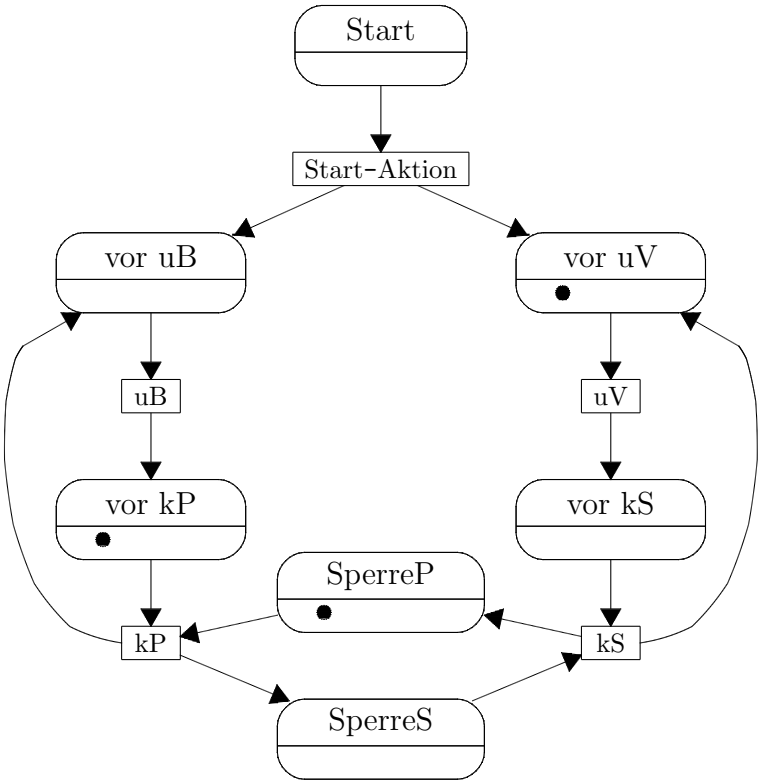
Als weiteres Beispiel stellen wir den Schweiss-Roboter aus dem Kapitel Synchronisationsmechanismen vor:


Start
Start-Aktion
vor uB
vor uV
uB
uV
vor kP
vor kS
SperreP
kP
kS
SperreS


Start
Start-Aktion
vor uB
vor uV
uB
uV
vor kP
vor kS
SperreP
kP
kS
SperreS


Start
Start-Aktion
vor uB
vor uV
uB
uV
vor kP
vor kS
SperreP
kP
kS
SperreS


Start
Start-Aktion
vor uB
vor uV
uB
uV
vor kP
vor kS
SperreP
kP
kS
SperreS



7.4 Übung

Erstellen Sie Regeln für die Implementation mit Semaphoren.

Kapitel 8

Schritthaltende Verarbeitung

Schritthaltende Verarbeitung, Realzeit-Betrieb, Echtzeit-Betrieb oder **Realtime-Betrieb** bedeutet, dass das einen **technischen Prozess (P)** steuernde **Automatisierungssystem (AS)** auf die Anforderungen des Prozesses innerhalb vorgegebener Zeiten reagiert. (Das gesamte System bezeichnen wir als ein **Prozess-Automatisierungs-System (PAS)**.)

Zusätzlich zur **Richtigkeit** eines Ergebnisses spielt bei der Realzeit noch die **Rechtzeitigkeit** des Ergebnisses eine wichtige Rolle.

Diese Zeitschranken finden ihren Ursprung in den Regelkreisen, die zur Steuerung von verfahrenstechnischen Anlagen, Maschinen, Robotern, Fahrzeugen (Fahrdynamik, Antrieb) benötigt werden.

8.1 Prozessbedingte Zeiten

8.1.1 Programm- oder Prozessaktivierung

Ein Programm des AS kann

- **zyklisch** (Alarme des technischen Prozesses und die Reaktion des Automatisierungssystems erfolgen in zeitlich konstantem Abstand.) oder
- **nicht zyklisch** (Alarme und Reaktion erfolgen zu willkürlichen Zeitpunkten.)

arbeiten. In jedem Fall kann das Programm des AS **ereignisgesteuert** durch den realen Prozess und/oder durch die **eigene festgelegte Programmausführung** aktiviert werden.

8.1.2 Spezifikation über Zeitdauern

Die prozessbedingte **maximal zulässige Reaktionszeit** oder **spezifizierte Reaktionszeit** bezeichnen wir mit t_z.

Die Ereignisse des zu steuernden Prozesses P *(events,* Alarme, Anforderungen), auf die das AS zu reagieren hat, treten i.A. zyklisch im zeitlichen Abstand t_p auf. t_p heißt **Prozesszeit (PZ)** oder im Zusammenhang mit Direct Digital Control (DDC) Abtastperiode oder Taktzeit. Die Prozesszeit ist nicht zu verwechseln mit der **Prozess-Zeitkonstante** oder genauer **Prozess-Regelstrecken-Zeitkonstante (PRZ)**, die durch die Natur des Prozesses (δ-Sprungverhalten, Einschwingverhalten usw.) vorgegeben ist und nicht durch das AS beeinflusst werden kann. Die Prozesszeit ist eine Design-Größe in Abhängigkeit von der Prozess-Zeitkonstanten. Bei DDC ist es z.B. sinnvoll, die Prozesszeit um eine oder mehrere Größenordnungen kleiner zu wählen als die Prozess-Zeitkonstante.

Die **Prozesszeit** t_p ist entweder eine Größe, die der Prozessbetreiber unter Berücksichtigung der physikalisch-chemischen Eigenschaften des Prozesses festlegt. Dann sind das häufig konstante Zeitintervalle. Oder die Prozesszeit hängt von mehr oder weniger zufälligen Ereignissen oder Prozesszuständen ab. In dem Fall können die Prozesszeiten sehr unterschiedlich sein. Damit aber überhaupt ein Systementwurf möglich ist, muss auf jeden Fall für die Prozesszeit eine untere Grenze $t_{p_{min}}$ angebbar sein:

$$t_{p_{min}} \leq t_p$$

Alle folgenden Überlegungen bleiben auch für den Fall der zufällig verteilten Prozesszeiten richtig, wenn man als Prozesszeit t_p den Wert $t_{p_{min}}$ verwendet *(Worst Case Design).*

Die **maximal zulässige Reaktionszeit** t_z muss so spezifiziert werden, dass

$$t_z \leq t_p$$

gilt. Wenn t_z nicht explizit angegeben wird, wird $t_z = t_p$ angenommen.

Damit Echtzeit-Betrieb gewährleistet ist, muss die **Reaktionszeit** t_r des Prozessautomatisierungssystems kleiner als die spezifizierte Reaktionszeit t_z sein, d.h. die **Echtzeitbedingung (EZB)** ***(real-time condition, RTC)***

$$t_r \leq t_z$$

muss gelten.

Durch welche Maßnahmen kann man nun die Erfüllung der Echtzeitbedingung erreichen?

Zunächst ist zu prüfen, ob die Erfassungs-, Rechen- und Ausgabekapazitäten überhaupt ausreichen, um die geforderten Aufgaben in den spezifizierten Zeitintervallen durchführen zu können. Darüberhinaus gibt es für den Programmierer eines AS nur noch zwei Entwurfs-Möglichkeiten: **Tasking** und **Vergabe von Prioritäten**.

Tasking bedeutet Aufteilung der ganzen Steuerungsaufgabe in Teilaufgaben, die (quasi-)parallel bearbeitet werden. Ein technischer Prozess kann i.A. so modelliert werden, dass er aus mehreren Teilprozessen i mit den zugehörigen Prozesszeiten t_{p_i} und maximal zulässigen Reaktionszeiten t_{z_i} besteht. Für alle Teilprozesse muss die Echtzeitbedingung erfüllt sein:

$$t_{r_i} \leq t_{z_i} \quad (\leq t_{p_i}) \qquad \text{für alle Teilprozesse} \quad i$$

- Jedem Teilprozess eine ihn steuernde Task zuzuordnen, ist ein bewährtes **Design-Prinzip** des AS.

8.1.3 Spezifikation über Zeitpunkte

Wir haben die Echtzeitbedingungen über Intervalle definiert. Man kann das auch über Zeitpunkte tun:

- R: frühester Startzeitpunkt *(release time)*
- D: spätester Endzeitpunkt *(deadline)*
- S: tatsächlicher Startzeitpunkt *(start time)*
- E: tatsächlicher Endzeitpunkt *(completion time)*

Es wird dann häufig noch die Laufzeit *(computation time)* spezifiziert:

$$C = E - S$$

Übung 1: Stellen Sie den Zusammenhang mit den Intervalldefinitionen her.

8.1.4 Übung 2: Motorsteuerung

Diskutieren Sie die Verhältnisse bei einer Motorsteuerung.

8.2 Task

Der Begriff **Task** bezieht sich auf die *Ausführung* eines Programms oder Programmstücks. Verschiedene Tasks können sich auf dasselbe Programmstück beziehen.

Anstatt Task werden auch die Begriffe **Prozess** (hier nicht als "zu steuernder Prozess", sondern als "Rechenprozess") oder **Thread** verwendet. Für uns ist Task der übergeordnete Begriff.

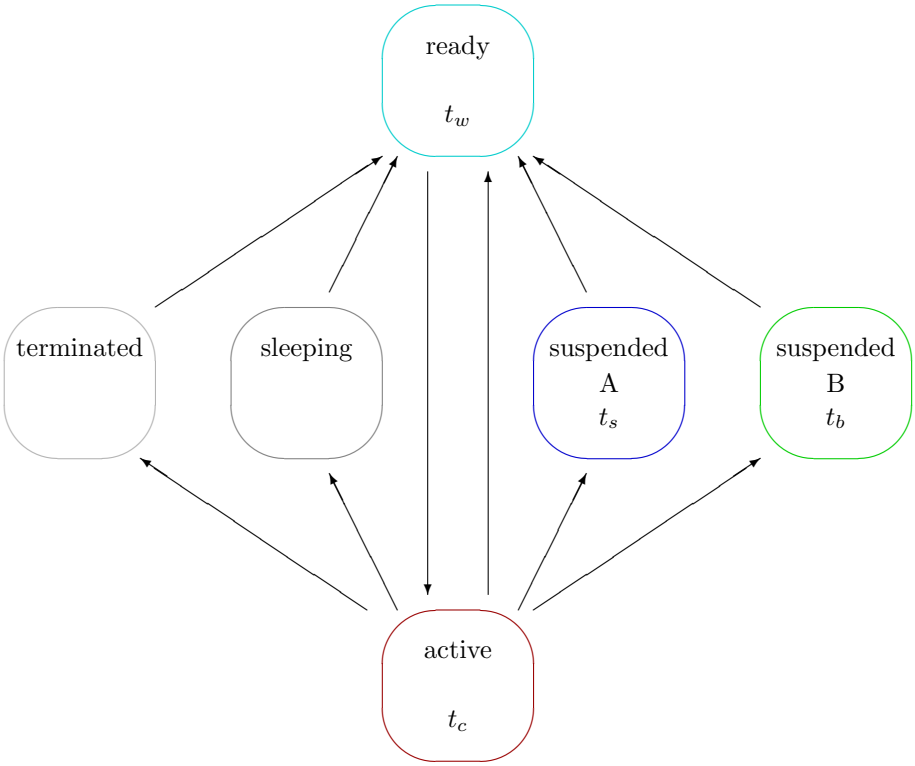
Ein (Rechen-)Prozess hat einen eigenen Adressraum. Diese Prozesse können nur über Betriebssystem-Mittel kommunizieren. Threads dagegen sind Leichtgewichtsprozesse ohne eigenen Adressraum. Sie können auch über globale Variablen kommunizieren.

Ein **Programm** ist eine statische Aufschreibung einer Folge von Anweisungen, während die Task als Programm*ablauf* zu verstehen ist.

Objekt-orientiert ist eine Task ein Objekt einer Klasse, die eine Methode hat, deren Code als Prozess oder Thread gestartet werden kann.

Das Verhalten einer Task wird für die hier nötigen zeitlichen Betrachtungen mit sechs Zuständen beschrieben.

Zustandsgraph einer Task:



Die bei den Zuständen angegebenen Zeiten sind die jeweiligen (kumulierten) Gesamtzeiten, die eine Task in einem Zustand während der Reaktionszeit t_r (näheres siehe unten) verbringt.

Die Bezeichnungsweisen sind sehr uneinheitlich. Daher sind hier einige der häufiger verwendeten Bezeichnungen alternativ angegeben.

Zustände:

terminated, beendet, neu: Die Task ist wurde noch nicht gestartet. Durch das Betriebssystem kann sie gestartet werden. Dann kommt sie in den Zustand *ready*. Wenn eine Task fertig ist, dann betritt sie ebenfalls diesen Zustand als *terminated* und kann dann eventuell neu gestartet werden.

sleeping, dormant, ruhend: Die Task ruht solange, bis sie auf Grund eines Ereignisses (Alarm) geweckt wird. Dann kommt sie in den Zustand *ready*.

ready, lauffähig, bereit, rechenbereit: Alle Voraussetzungen dafür, dass die Task laufen (d.h. die CPU benutzen) kann, sind erfüllt. Es ist nur noch eine Frage der Priorität, ob die Task läuft. Wenn sie die höchste Priorität bekommt, dann geht die Task in den Zustand *active* über.

active, running, busy, laufend, rechnend: Die Task läuft, d.h. wird vom Prozessor bearbeitet. Bei einem Einprozessorsystem kann nur eine Task in diesem Zustand sein. (Alle anderen Zustände können von mehreren Tasks eingenommen werden.)

Der Zustand *active* wird verlassen,

- wenn die Task die höchste Priorität verloren hat (zurück in Zustand *ready)*,
- oder wenn die Task an ihr natürliches Programmende gelaufen ist (dann in Zustand *terminated)*,
- oder wenn die Task sich schlafen legt oder auf Alarm/Ereignis wartet (dann in Zustand *sleeping)*,
- oder wenn eine Warte-Bedingung eingetreten ist (dann in Zustand *suspended* A oder B).

suspended, pended, blockiert, wartend: Die Task kann nicht laufen, da für sie eine Wartebedingung gilt, d.h. sie muss auf ein Ereignis warten, z.B. darauf, dass ein Betriebsmittel frei wird, oder auf I/O, auf einen Interrupt oder darauf, dass ein Zeitintervall abgelaufen ist. Nach Eintreten des Ereignisses geht die Task in den Zustand *ready*.

Abweichend von sonst üblichen Darstellungen spalten wir diesen Zustand in zwei Zustände auf:

- `A`: Die Task wartet auf ein Ereignis, das nur von ihrem zu steuernden Teilprozess abhängig ist (z.B. ein vom Teilprozess benutzter A/D-Wandler).
- `B`: Die Task wartet auf die Zuteilung eines – von anderen Tasks benutzten – Betriebsmittels (z.B. gemeinsam benutzte I/O-Karten oder Speicherplätze) oder kritischen Bereichs.

Randbemerkung: Wenn die CPU gar keine Task zu bearbeiten hat, dann ist sie im sogenannten ***idle state***. (Das ist aber ein Zustand der CPU, nicht der einer Task.)

8.3 Reaktionszeit

Um Aussagen über die Echtzeitbedingung machen zu können, müssen wir uns genauer mit der Reaktionszeit t_r des AS beschäftigen. Innerhalb der Reaktionszeit muss das AS typischerweise auf eine Anforderung – ausgelöst durch ein Prozessereignis (spezieller Prozesszustand, Interrupt oder Uhrzeit) – reagieren mit

- Analyse der Anforderung
- Messen mit einem Messinstrument
- Erfassen des Messwerts über eine I/O-Schnittstelle
- Verarbeiten des Messwerts und Berechnung einer Ausgabegröße
- Ausgeben der Ausgabegröße über eine I/O-Schnittstelle
- Steuern mittels eines Aktors

Zur Abschätzung von t_r muss man grundsätzlich den ***Worst-Case*** betrachten **(WCET, *worst-case execution time*)**.

Die Zeit, die für diese Tätigkeiten – **ohne** Berücksichtigung anderer Tasks – benötigt wird, nennen wir **Verarbeitungszeit** t_v. Sie setzt sich zusammen aus der **Prozessorzeit** t_c (Zeit im Zustand *active*) und der Zeit im Zustand *suspended (A)* t_s, weil die Task etwa auf einen A/D-Wandler, ein Messgerät oder ein anderes Ereignis **ihres zu steuernden** Prozesses während der Reaktionszeit warten muss.

$$t_v = t_c + t_s$$

Die Verarbeitungszeit kann man für eine isolierte Task relativ gut abschätzen oder messen. Die Reaktionszeit t_r dagegen ist die Verarbeitungszeit t_v plus die Zeit t_w, die die Task im Zustand *ready* auf die Prozessorzuteilung warten muss, plus die Zeit im Zustand *suspended (B)* t_b, wo die Task auf ein gemeinsam benutztes Betriebsmittel warten muss.

$$t_r = t_v + t_w + t_b$$

Die Zeit t_w ist abhängig von der Anzahl und den Prioritäten anderer Tasks und kann daher nur sehr schwer für eine einzelne Task bestimmt werden. Wir geben hier eine Abschätzung für die Task i:

$$t_{w_i} \leq \sum_{\text{alle } j \neq i} t_{c_j} \cdot \left\lceil \frac{t_{z_i}}{t_{p_j}} \right\rceil \qquad \text{Prio}(j)(\text{logisch} \geq)\text{Prio}(i)$$

Hierbei ist über alle Tasks j zu summieren, die höhere oder gleiche Priorität haben als die Task i. Die Task i wird in der Summe nicht berücksichtigt. Der gerundete Term ($\lceil x \rceil$ = ceil(x) = nächst höhere ganze Zahl) gibt an, wie oft der Prozess j innerhalb der maximal zulässigen Reaktionszeit t_{z_i} der Task i seine Prozessorzeit t_{c_j} benötigt. Für ein Worst-Case-Design sollte das Gleichheitszeichen genommen werden.

Wie kann man t_{b_i} abschätzen? Unter der Voraussetzung, dass das unten diskutierte Problem der Prioritätsinversion gelöst ist, muss für jedes von der Task i benutzte Betriebsmittel BM für alle Tasks $j \neq i$ die maximale Benutzungsdauer $t_{BM_{max_{j,i}}}$ innerhalb von $\min(t_{z_j}, t_{z_i})$ bestimmt werden.

$$t_{alleBM_{max_{j,i}}} = \sum_{alleBM} t_{BM_{max_{j,i}}}$$

Hierbei wird über alle Betriebsmittel summiert, die sich Task i und Task j während t_{r_i} teilen.

Dann ergibt sich als Abschätzung für t_{b_i}

$$t_{b_i} \leq \sum_{\text{alle } j \neq i} t_{alleBM_{max_{j,i}}} \cdot \left\lceil \frac{t_{z_i}}{t_{p_j}} \right\rceil$$

Hier muss – unabhängig von der Priorität – über alle Tasks summiert werden, wobei natürlich für alle Tasks, die mit der Task i kein Betriebsmittel teilen, der Term $t_{alleBM_{max_{j,i}}}$ Null ist. Für ein Worst-Case-Design sollte das Gleichheitszeichen genommen werden.

Wartezeiten, die während des Gebrauchs eines Betriebsmittels durch die Task i auftreten, wurden schon bei t_{s_i} berücksichtigt.

Bemerkungen:

- Die Bestimmung der Terme $t_{BM_{j,i}}$ ist oft nicht einfach. Als Beispiel betrachten wir zwei Tasks A und B, die über einen Ringpuffer kommunizieren. Der Zugriff auf den Ringpuffer geschieht unter gegenseitigem Ausschluss und dauert $10\mu s$. Task A schreibt alle $t_{z_A} = 100\mu s$ Daten in den Ringpuffer. Task B liest diese Daten aus dem Ringpuffer im Schnitt mit gleicher Geschwindigkeit oder etwas schneller, aber ziemlich unregelmäßig innerhalb ihres Zyklus von $t_{z_B} = 10\,000\mu s$.
  $t_{BM_{max_{B,A}}} = 10\mu s \quad \text{bis} \quad 100\mu s$
  $t_{BM_{max_{A,B}}} = 10\mu s$
  Bei $t_{BM_{max_{B,A}}}$ kann der Wert 100 auftreten, wenn Task B lange nicht gelesen hat, und auf einmal 10 mal oder noch öfter liest. Das würde bedeuten, dass Task A ihre Echtzeitbedingung nicht erfüllen könnte. Allerdings tritt der Wert 100 nicht auf, wenn wir den Zugriff auf den Ringpuffer für Task A priorisieren. Wenn dann Task A schreiben will, muss sie höchstens einen Lesezugriff von $10\mu s$ abwarten. Für die Zeiten t_b ergibt sich:
  $t_{b_A} = 10 \cdot \lceil \frac{100}{10\,000} \rceil = 10\mu s$
  $t_{b_B} = 10 \cdot \lceil \frac{10\,000}{100} \rceil = 1000\mu s$
  Eine Tabelle der folgenden Form mag hilfreich sein:

Task X → wartet $t_{alleBM_{max_{Y,X}}}$ auf Task Y ↓	A	B	C	...
A	0	10 + 50 = 60ms	100 + 40 = 140ms	...
B	10 + 60 = 70ms	0	80ms	...
C	120ms	80ms	0	...
...	...	...	...	0
t_{b_X}	190ms	140ms	220ms	...

- Die vorgestellten Abschätzungen orientieren sich an der ***rate monotonic analysis (RMA)*** bzw. ***rate monotonic scheduling (RMS)***[24][23].

- Der System-Overhead, d.h. die Prozessorzeit, die die Systemtask(s) benötigen (Taskwechselzeiten), ist hier mit zu berücksichtigen. Das ist im Detail sehr schwer durchzuführen, kann aber eventuell pauschal gemacht werden. Wenn z.B. der Systemoverhead erfahrungsgemäß etwa 10% beträgt, könnte die oben angegebene Summe mit dem Faktor 1.1 multipliziert werden.

  Allgemein ist das schwierig zu berücksichtigen. Innerhalb eines t_c kann es durchaus mehrere Taskwechsel geben. Oft sind die Taskwechselzeiten größer als die netto Rechenzeiten.

  Eine vernünftige Näherung dürfte sein, wenn man anstatt von t_c

  $$t_c + 2 \cdot \text{Taskwechselzeit}$$

  verwendet.

8.4 Relative Gesamtbelastung

Eine notwendige Voraussetzung für das Einhalten der Echtzeitbedingung ist, dass die relative Gesamtbelastung des Prozessors kleiner 1 bzw. kleiner 100% ist.

$$\text{Relative Gesamtbelastung} \quad = \sum_{\text{alle Tasks } i} \frac{t_{c_i}}{t_{p_i}} \qquad \leq 1$$

Wenn die Tasks diese Bedingung erfüllen, dann heißen sie **zeitlich verwaltbar**.

Die relative Gesamtbelastung kann berechnet werden, ohne dass das Tasking klar ist, indem man für jede Aktivität a die benötigte CPU-Zeit t_{c_a} und ihre Häufigkeit, d.h. ihre Prozesszeit t_{p_a} bestimmt, und obige Formel in folgender Formulierung verwendet:

$$\text{Relative Gesamtbelastung} \quad = \sum_{\text{alle Aktivitäten } a} \frac{t_{c_a}}{t_{p_a}} \qquad \leq 1$$

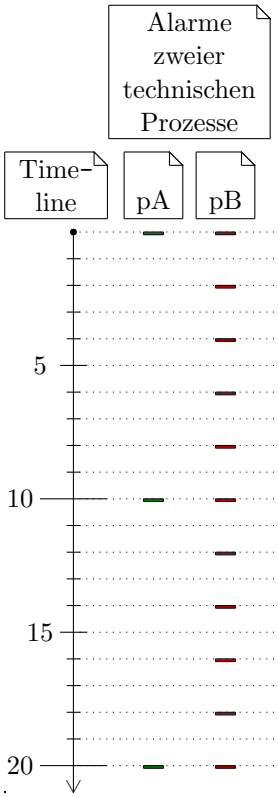
Nach dem Tasking kommen dann wahrscheinlich noch Terme für die Intertask-Kommunikation hinzu.

8.5 Priorität

Betrachten wir als Beispiel zwei Tasks, die jeweils einen realen Teilprozess steuern sollen, mit folgenden Zeiten:

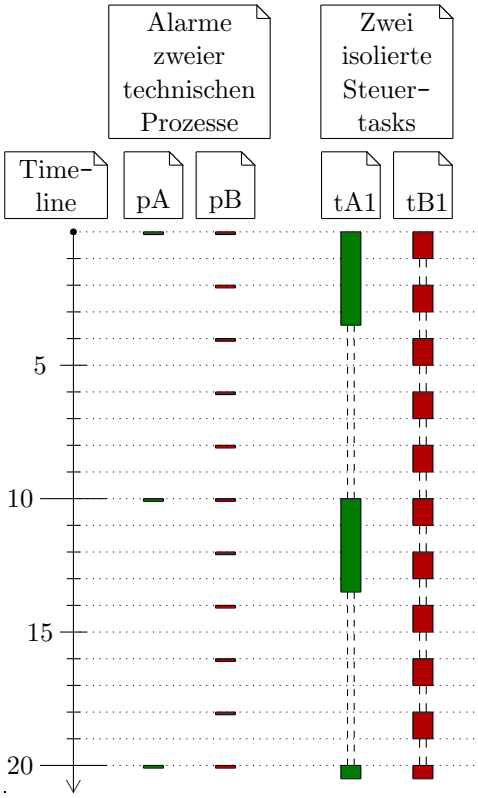
Prozess	$t_z = t_p$ ms	$t_c = t_v$ ms
A	10.0	3.5
B	2.0	1.0

Zwei technische Prozesse



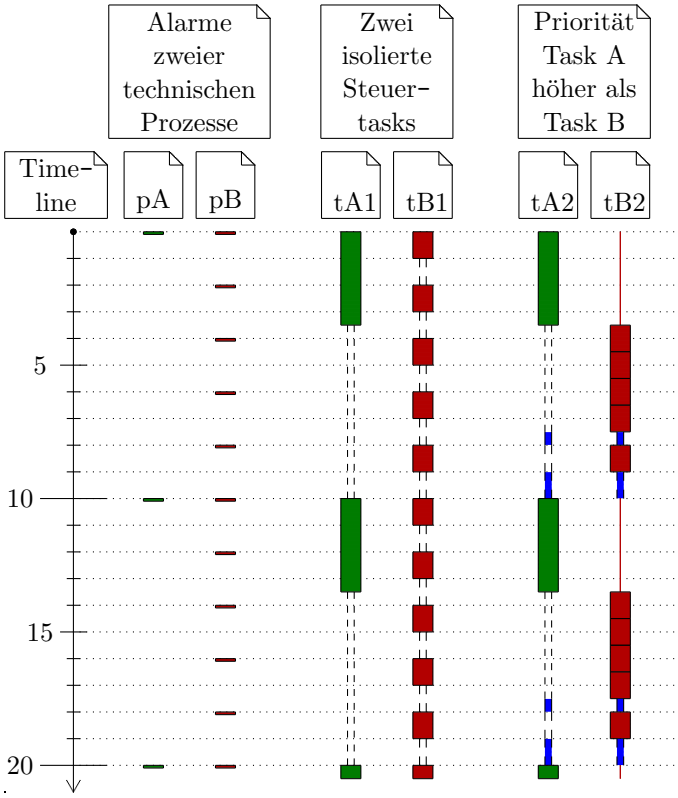
Wenn der Prozessor ausschließlich **einen** Teilprozess bedienen kann, dann sieht das für die beiden Teilprozesse folgendermaßen aus:

Zwei technische Prozesse und zwei Tasks



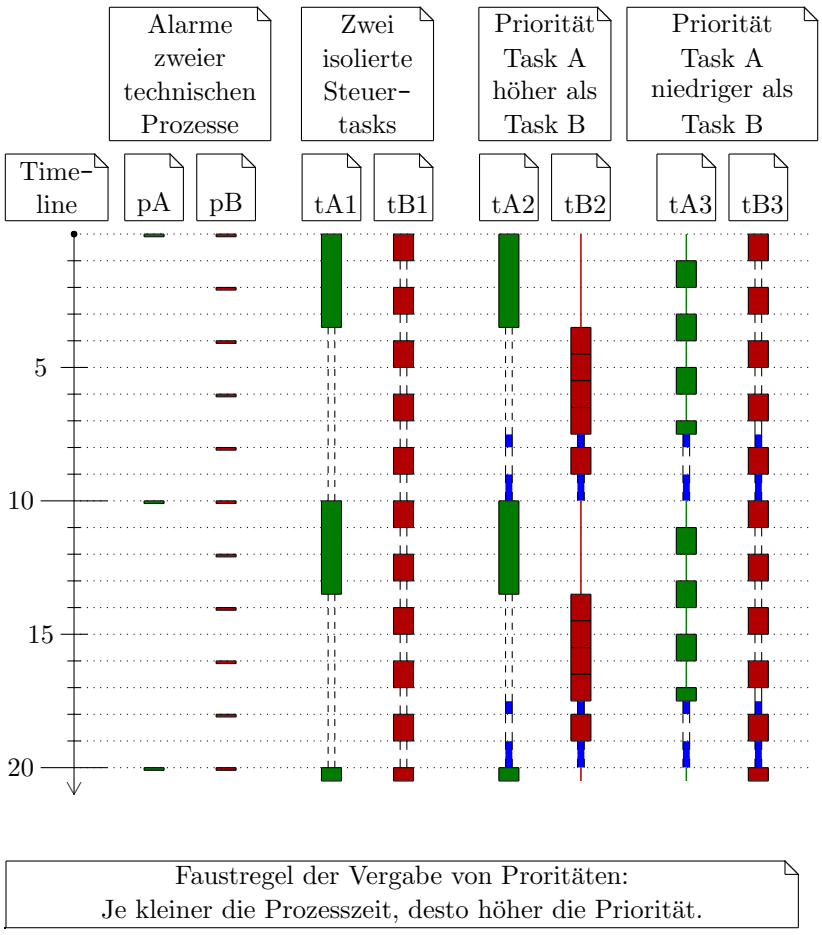
Wenn wir Task A, die Teilprozess A steuert, die höhere Priorität geben, wird die Echtzeitbedingung bei A eingehalten. Bei Teilprozess B kann die Echtzeitbedingung meistens nicht erfüllt werden, wie man an unten dargestelltem Szenario erkennt.

Zwei technische Prozesse und zwei Tasks



Wenn wir Task B, die Teilprozess B steuert, die höhere Priorität geben, wird die Echtzeitbedingung bei B und auch bei A eingehalten, wie man an unten dargestelltem Szenario erkennt. Allerdings haben wir hier noch vorausgesetzt, dass die höher priore Task die Task mit niedrigerer Priorität unterbrechen kann.

Zwei technische Prozesse und zwei Tasks



Faustregel der Vergabe von Proritäten:
Je kleiner die Prozesszeit, desto höher die Priorität.

In diesem Beispiel ergibt sich die relative Gesamtbelastung zu $\frac{3.5}{10.0} + \frac{1.0}{2.0} = 0.85$.

Je nach Prioritätsvergabe ergeben sich mit der oben angegebenen Abschätzung für die Wartezeiten t_w folgende Werte:

Prozess	$t_z = t_p$	$t_c = t_v$	t_v	A höhere Priorität		B höhere Priorität	
				t_w	t_r	t_w	t_r
A	10.0	3.5	3.5	0.0	3.5	5.0	8.5
B	2.0	1.0	1.0	3.5	4.5	0.0	1.0

Diese Werte bestätigen das Ergebnis des Szenarios.

Damit Prioritäten sinnvoll vergeben werden können, muss das Betriebssystem ein ***priority based preemptive scheduling*** zur Verfügung stellen. (Eine Task muss auf Grund von Priorität unterbrechbar sein.)

Als Faustregel ergibt sich:

Prioritäten werden umgekehrt propotional zu den Prozesszeiten vergeben.

Diese Faustregel entspricht der Scheduling-Strategie ***rate monotonic scheduling strategy (analysis) (RMS(A))***.

Liu und Layland [23] haben gezeigt, dass mit RMS die Realzeit-Bedingung für alle Tasks eingehalten werden kann, wenn die relative Gesamtbelastung folgende Bedingung erfüllt:

$$\text{Rel.Ges.} \quad = \quad \sum_{i=1}^{n} \frac{t_{c_i}}{t_{p_i}} \quad \leq \quad n(2^{\frac{1}{n}} - 1)$$

Hierbei sei n die Anzahl der Tasks.

$n =$	1	2	3	4	10	100	∞
$n(2^{\frac{1}{n}} - 1) =$	1,000	0,828	0,780	0,757	0,718	0,696	$\ln 2 =$ 0,693

Weitere Beispiele:

Prozess	$t_z = t_p$ ms	$t_c = t_v$ ms
A	10.0	1.0
B	2.0	1.5

Prozess	$t_z = t_p$	$t_c = t_v$
A	24 h	8 h
B	7 ms	1 ms
C	600 s	300 s

8.6 Scheduling-Strategien

Das Kapitel hat sich an der **RMA** ***(Rate Monotonic Analysis)*** oder auch der **RMS** ***(Rate Monotonic Scheduling-Strategy)*** orientiert. Als Ergänzung seien hier erwähnt:

EDF ***(Earliest Deadline First)*****:** Bei jedem möglichen Taskwechsel ermittelt der Scheduler die früheste Deadline.

Die Task mit der frühesten Deadline bekommt den Prozessor. Bei RMA haben niedrigpriore Tasks die Tendenz den zulässigen Zeitbereich fast ganz auszunützen, was u.U. sehr gefährlich aussieht. Bei EDF können Tasks mit großen Prozesszeiten auch früher mit ihrer Reaktion fertig werden.

Allerdings ist EDF ein dynamisches Verfahren, was manche Regelwerke verbieten.

PTS ***(Preemption-Threshold Scheduling)*****:** Zusätzlich zu ihrer RMA-Priorität P_1 hat eine Task eine zweite Priorität $P_2 \geq P_1$. Oft haben mehrere Tasks diesselbe P_2 ***(preemption threshold)***.

Das PTS lässt Unterbrechungen nur durch Tasks zu, deren Priorität höher ist als die Preemption-Threshold P_2.

Insgesamt hat das weniger Taskwechsel zur Folge, ohne dass die Echtzeit-Bedingung verletzt wird. (Das Realtime-BS ThreadX bietet PTS an. Siehe dort auch den User Guide.)

Übung 3:

Task	t_p	$t_v = t_c$
A	6	3
B	8	3
C	10	1

Berechnen Sie die relative Gesamtbelastung.

Vergeben Sie Prioritäten nach RMS und simulieren Sie graphisch.

Wenden Sie EDF an und simulieren Sie graphisch.

Übung 4:

Task	t_p	$t_v = t_c$
A	6	2
B	8	2
C	10	4

Berechnen Sie die relative Gesamtbelastung.

Vergeben Sie Prioritäten nach RMS und simulieren Sie graphisch.

Wenden Sie EDF an und simulieren Sie graphisch.

Wenden Sie PTS an, wobei P_2 gleich der Priorität von Task B sein soll, und simulieren Sie graphisch.

8.7 Harte und weiche Echtzeitbedingungen

Bei vielen Systemen wird eine gelegentliche Verletzung der Echtzeitbedingung nicht als Systemzusammenbruch gewertet. Stankovic hat daher die Begriffe ***Hard and Soft Real-Time Constraints*** – harte und weiche Echtzeitbedingungen – geprägt.

8.7.1 Harte Echtzeitbedingung

Eine "harte" Echtzeitbedingung ***(hard real time constraint)*** liegt vor, wenn bei Nichterfüllung der Bedingung das System ausfällt *(system fails, loss of control)*.

(Bild)

$h(t_r)$ ist die Häufigkeit, mit der eine Reaktionszeit t_r auftritt. Bei der harten Echtzeitbedingung gilt:

$$h(t_r) = 0 \quad \text{für alle} \quad t_r > t_z$$

t_z ist ein "harter" Termin.

Beispiele sind eine Paketsortieranlage, wenn bei nicht rechtzeitiger Weichenstellung Pakete falsch sortiert werden, oder eine Flugzeugsteuerung oder ein Kofferband.

8.7.2 Weiche Echtzeitbedingung

Stankovic spricht von einer weichen EZB, wenn durch die Nichterfüllung einer EZB zwar die Systemleistung beeinträchtigt ***(soft real time constraint)*** *(system degraded)* ist, das System aber noch nicht ausgefallen ist.

(Bild)

In dem Fall spricht man eventuell erst dann von einem Systemausfall, wenn der Schwerpunkt der Häufigkeitsverteilung in die Nähe von t_z wandert.

$$\text{Schwerpunkt} = \frac{\int_0^\infty t_r \, h(t_r) \, dt_r}{\int_0^\infty h(t_r) \, dt_r} \; < \; t_z \cdot \text{Faktor}$$

Die Spezifikation des Faktors ist z.B. Gegenstand der System-Anforderungen. Beispiele für weiche Echtzeitbedingungen sind Buchungssysteme oder interaktive Systeme, wo das System normalerweise innerhalb einer Sekunde reagieren sollte, wo aber auch 10 Sekunden gelegentlich hingenommen werden.

8.8 Entwurfsregeln und Bemerkungen

Echtzeitbedingungen:

- $t_{v_i} \leq t_{z_i}$ für alle Tasks i ist eine *notwendige* Echtzeitbedingung.
- $\sum_i \frac{t_{c_i}}{t_{p_i}} \quad \leq 1$ ist auch nur eine *notwendige* Echtzeitbedingung.
- $t_{r_i} \leq t_{z_i}$ für alle Tasks i ist eine *hinreichende* Echtzeitbedingung oder *die Echtzeitbedingung* schlechthin.
- Besonders kritische Zustände (Alarmzustände) dürfen u.U. die Echtzeitbedingungen verletzen.

Tasking: Die gesamte Automatisierungsaufgabe muss so in parallele Teilaufgaben zerlegt werden, dass die Echtzeitbedingungen eingehalten werden. Hierbei sollte man sich an den Prozesszeiten orientieren.

Faustregel: Pro Prozesszeit wird eine steuernde Task definiert.

Scheduling-Strategie: RMS, EDF oder PTS? RMS lässt sich bei fast allen Systemen anwenden. D.h. die Priorität wird umgekehrt proportional zu t_z vergeben. Da hier die Prioritäten vorab vergeben werden ("Vorabprioritäten", ***static priority***), ist das ein **statisches** Verfahren. Es gibt Regelwerke, wonach Prioritäten zur Laufzeit niemals verändert werden dürfen ("dynamische Prioritäten-Änderung", ***dynamic priority***).

Wie wir in der Übung 3 gesehen haben, lässt sich mit EDF die Realzeitbedingung bei mehr Systemen erfüllen als mit RMS. Allerdings ist EDF ein dynamisches Verfahren und lässt sich in vielen Systemen nicht realisieren. Ferner sollte die relative Gesamtbelastung 50 % nicht übersteigen, so dass RMS immer möglich ist.

PTS ist wieder ein statisches Verfahren, das allerdings nur von wenigen Systemen angeboten wird.

Relative Gesamtbelastung: Die relative Gesamtbelastung $\sum_i \frac{t_{c_i}}{t_{p_i}}$ sollte 0.3 bis 0.5 , d.h. 30% bis 50% nicht übersteigen. Die 85% des oben genannten Beispiels gelten als sehr gefährlich.

Zeitscheibenverfahren: Wenn das Betriebssystem ein Zeitscheibenverfahren ohne Möglichkeit der Prioritätsvergabe hat, dann ist unter folgenden Bedingungen immer noch ein Echtzeitbetrieb möglich: n sei die maximale Anzahl der Tasks. τ sei die Dauer einer Zeitscheibe *(time slice, time quantum,* Zeit, die einer Task zugeteilt wird). Dann gilt:

$$t_{r_i} \leq n \cdot \tau \cdot \left\lceil \frac{t_{c_i}}{\tau} \right\rceil + t_{s_i} + t_{b_i}$$

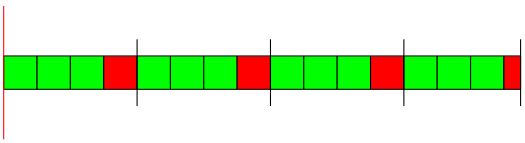
Denn pro Durchlaufzeit $n\tau$ durch alle Tasks wird an der Task i nur τ-lang gearbeitet. Um auf die Prozessorzeit t_{c_i} zu kommen, sind daher höchstens $\lceil \frac{t_{c_i}}{\tau} \rceil$ Durchläufe bzw. höchstens die Zeit $n \cdot \tau \cdot \lceil \frac{t_{c_i}}{\tau} \rceil$ notwendig.

Die Echtzeitbedingung $t_{r_i} \leq t_{z_i}$ wird zu

$$n \cdot \tau \cdot \left\lceil \frac{t_{c_i}}{\tau} \right\rceil + t_{s_i} + t_{b_i} \quad \leq \quad t_{z_i}$$

Wenn diese Bedingung erfüllt werden kann und wenn die Anzahl der Tasks den Wert n nie überschreitet, dann ist das eine ziemlich gute Möglichkeit ein Echtzeitsystem zu realisieren.

Beispiel Zeitscheibenverfahren:



$$n = 4; \qquad \tau = 1; \qquad t_v = 3.5; \qquad t_c = t_v;$$

$$t_r \leq n \cdot \tau \cdot \left\lceil \frac{t_c}{\tau} \right\rceil = 4 \cdot 1 \cdot \left\lceil \frac{3.5}{1} \right\rceil = 16$$

Kritische Bereiche: Kritische Bereiche sind Bereiche, die von allen oder manchen Tasks ausschließlich benutzt werden (exklusive Benutzung von Betriebsmitteln oder Ressourcen).

Kritische Bereiche: Kritische Bereiche sind Bereiche, die von allen oder manchen Tasks ausschließlich benutzt werden (exklusive Benutzung von Betriebsmitteln oder Ressourcen). Da diese Bereiche das Prioritätsschema stören können, weil die Tasks im kritischen Bereich eventuell mit wesentlich höherer Priorität läuft, sollten kritische Bereiche so kurz wie möglich gehalten werden.

In den Formeln sind diese Bereiche zwar durch t_{b_i} berücksichtigt, aber nicht bei Fällen, wo die Tasks sich keinen kritischen Bereich teilen. In diesen Fällen sollte man die Abschätzung für t_w modifizieren und CPU-Zeiten in kritischen Bereichen für alle Tasks hinzunehmen, die dann mit höherer Priorität laufen.

Ablaufsteuerung: Die Vergabe von Prioritäten darf auf keinen Fall zur Ablaufsteuerung oder zum Schützen kritischer Bereiche verwendet werden, um etwa zu erreichen, dass eine Task ein Programmstück *vor* einer anderen Task ausführt, oder etwa nach dem Muster: Gib der Task, die einen kritischen Bereich betritt, für die Zeit im kritischen Bereich eine höhere Priorität als ihren Konkurrenten. Das ist deswegen sehr gefährlich, weil anzunehmen ist, dass im kritischen Bereich Wartebedingungen (z.B. auf I/O) eintreten, wodurch die Task suspendiert würde und damit eine niedrigerpriore Task Gelegenheit bekäme, ihren Code durchzuführen, eventuell auch einen nicht durch ein Lock geschützten kritischen Bereich zu betreten.

(Wenn durch glückliche Umstände eine Ablaufsteuerung durch Priorität auf einem Ein-Prozessor-System vielleicht noch funktioniert hat, dann wird es auf einem Mehr-Prozessor-System nicht mehr funktionieren, weil wir da eventuell echte Parallelität haben und Tasks unterschiedlicher Priorität gleich schnell laufen können.)

Die Priorität ist *kein* Mittel zur Synchronisation oder Ablaufsteuerung. Semaphore, Boltvariable, Monitore, Rendezvous und Transactions sind Mittel für die Ablaufsteuerung.

Für das Schützen kritischer Bereiche sollten Lock-Mechanismen (z.B. MUTEXe) verwendet werden.

Jedesmal, wenn eine Wartezeit verwendet wird (um etwa andere Tasks zum Zuge kommen zu lassen), dann ist das ein Zeichen von schlampiger, meist auch gefährlicher Programmierung. Die Verwendung einer Wartezeit ist nur dann möglicherweise gerechtfertigt und eventuell unvermeidbar, wenn sie mit dem zu steuernden Prozess zu tun hat, d.h. wenn das Zeitverhalten die einzige Information ist, die über das Verhalten des realen Prozesses zur Verfügung steht.

Hard-Soft Real-Time Constraints: Auch technische Systeme haben bei genauer Betrachtung häufig weiche Echtzeitbedingungen. Oft ist es daher sinnvoll, eine harte *und* eine weiche Echtzeitbedingung zu spezifizieren. Bei einer Überdruckventilsteuerung z.B. sollte normalerweise innerhalb einer Sekunde reagiert werden, aber Explosionsgefahr besteht erst, wenn einige Minuten lang nicht reagiert wird. Dann wären etwa folgende Spezifikationen sinnvoll:

$$\begin{aligned} t_{z_{\text{soft}}} &= 1 \text{ Sekunde mit Faktor} = 2 \\ t_{z_{\text{hard}}} &= 60 \text{ Sekunden} \end{aligned}$$

Deterministisches Verhalten: Ein realer technischer Prozess verhält sich nie **deterministisch**. Denn sonst müsste man ihn nicht steuern. Im Gegenteil, wir müssen davon ausgehen, dass sich der technische Prozess relativ willkürlich verhält.

Allerdings muss sich das AS insofern deterministisch verhalten, als es auf alle möglichen Zustände des technischen Prozesses P in einer vorhersehbaren Weise reagiert. Ziel des AS ist es, deterministisches Verhalten des Gesamtsystems (Technischer Prozess P und AS = PAS) insofern zu garantieren, als dass gewisse spezifizierte Grenzen nicht überschritten werden.

Determinismus *(determinism)* bedeutet, dass zu jedem Systemzustand und jeder dann möglichen Eingabemenge die Ausgabemenge eindeutig ist.

Diese Definition betrifft alle Datenverarbeitungssysteme. Für Echtzeit-Systeme fordern wir zusätzlich ein Zeitverhalten:

Zeitlicher Determinismus *(temporal determinism)* bedeutet, dass die für eine Reaktion benötigte Zeit endlich und in Grenzen vorhersagbar ist.

Vorgehensweise:

- Ermittlung bzw. Definition von Prozesszeiten und zulässigen Reaktionszeiten.
- Tasking: Definition von Teilaufgaben bezüglich der Prozesszeiten mit konzeptioneller Vereinfachung durch Parallelisierung.
- Abschätzung oder Messung der Verarbeitungszeiten und Vergleich mit den zulässigen Reaktionszeiten.
- Berechnung der Relativen Gesamtbelastung.
- Vergabe von Prioritäten.
- Abschätzung der Reaktionszeiten und Vergleich mit den zulässigen Reaktionszeiten.

8.9 Prozess- bzw. Taskbindung

In diesem Abschnitt werden lediglich einige Begriffe vorgestellt, mit denen die Interaktion verschiedener Aktionen, Aktivitäten und Tasks beschrieben werden und die möglicherweise Hinweise für das Tasking liefern.

Zeitliche Bindung: Aktionen und Aktivitäten, die stets in einem definierten Zeitintervall und Reihenfolge ausgeführt werden, werden in einer Task zusammengefasst. Z.B. kann das eine Initialisierungstask sein.

Sequentielle Bindung: Aktivitäten sind so hintereinandergeschaltet, dass die Ausgabe der einen Aktivität die Eingabe der nächsten Aktivität ist (Datenfluss). Wenn die Aktivitäten nebenläufig sein sollen, werden sie durch eine Queue oder Pipe zeitlich entkoppelt.

Bindung durch Kommunikation: Aktivitäten hängen durch intensive Kommunikation so eng zusammen, dass die Nebenläufigkeit stark eingeschränkt ist. Hier bietet sich die Zusammenfassung in einer Task an.

Aktionsbindung: Einzelne Aktionen und Aktivitäten einer Task arbeiten an einer abgeschlossenen Aufgabe.

Informale Bindung: Verwendet eine Task Betriebsmittel exklusiv, die von anderen Tasks auch nur exklusiv verwendet werden können, so liegt bei diesen Tasks eine informale Bindung vor.

Harte Echtzeitbedingung: Eine Aktivität, die harten Echtzeitbedingungen unterliegt, wird innerhalb einer Task möglichst ohne Bindung an andere Tasks durchgeführt.

8.10 Beispiel Numerische Bahnsteuerung

Jede Millisekunde soll eine Koordinate x an eine Positioniereinrichtung gegeben werden. Die Steuerdaten liegen als Stützstellen $x(t_i)$ alle 10 ms vor: $t_{i+1} = t_i + 10$ ms. Die dazwischenliegenden Werte (Zwischenstützwerte) werden durch ein Polynom zweiten Grades interpoliert. Zur Bestimmung eines Zwischenstützwerts benötigt der Rechner $t_c = 160$ μs. Die Ausgabe an die Positioniereinrichtung dauert $t_v = 30$ μs, wovon 10 μs CPU-Zeit sind. Für die Feststellung, ob ein Wert ein Zwischenstützwert ist, werden $t_c = 10$ μs benötigt.

Alle 10 ms sind jedoch neue Polynomkoeffizienten zu berechnen. Dies dauert $t_c = 2$ ms.

(Bild)

Welche Prozesszeiten gibt es?

Welche maximal zulässigen Reaktionszeiten gibt es?

Wie groß ist die Relative Gesamtbelastung?

Eine Lösung mit einer Task ist ohne große Umstände offenbar nicht möglich, wie unten stehende Abbildung zeigt:

(Bild)

Wie sieht die Lösung mit mehreren Tasks aus? Dabei kann angenommen werden, dass eine eventuell nötige Intertaskkommunikation in "Schreibrichtung" 100 μs, in "Leserichtung" 50 μs kostet.

(Bild)

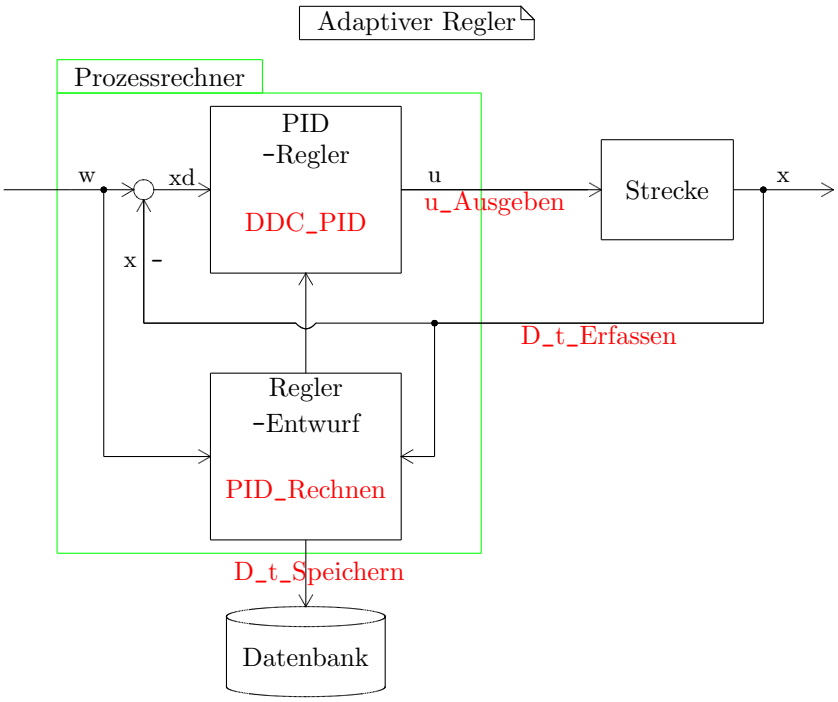
Müssen Prioritäten vergeben werden und wie?

Weisen Sie nach, dass die Echtzeitbedingung bei richtigem Tasking und Prioritätsvergabe erfüllt ist.

Beschreiben Sie näher die Intertaskkommunikation. D.h. welches Schreib- und Leseverhalten muss man fordern? Welche Art von Puffer ist geeignet und welchen Einfluss hat der Puffer auf das Verhalten der Tasks?

8.11 Übungen

8.11.1 Beispiel Durchflussregelung



Aufgabe Echtzeitbedingung bei Durchflussregelung

Der Durchfluss `D` und die Temperatur `T` eines Rohres sollen mit DDC geregelt werden. Der Durchfluss wird mit einem PID-Algorithmus, die Temperatur mit einem einfachen schaltenden Zweipunktregler geregelt. Dazu steht ein Einprozessor-Rechnersystem zur Verfügung.

Die Temperaturüberwachung soll jede 6 Millisekunden (6 ms) erfolgen. Bei der Durchflussregelung soll die Abtastperiode 60 ms betragen. Alle gemessenen Durchfluss-Zeit-Wertepaare sollen in eine Datenbank geschrieben werden. Ferner sollen neue Regelparameter aus jeweils 100 Durchfluss-Zeit-Wertepaaren berechnet werden und der Regelung zur Verfügung gestellt werden. Folgende Funktionen stehen für diese Aufgaben zur Verfügung:

`T_Erfassen` A/D-Wandlung der Temperatur: Dauert insgesamt 300 μs (μs = Mikrosekunden), wobei die CPU immer beschäftigt ist ($t_v = t_c = 300$ μs).

`T_Vergleich` Die Temperatur wird mit einem oberen und unteren Temperaturwert verglichen ($t_v = t_c = 100$ μs).

`H_Ausgeben` Ausgabe über eine Digital-I/O-Karte an eine Heizung, die je nach gemessener Temperatur an- oder abgeschaltet wird ($t_v = t_c = 200$ μs).

`D_t_Erfassen` A/D-Wandlung des Durchflusses `D` und Bestimmung der Zeit `t`, zu der der Durchflusswert anfällt. Der A/D-Wandler ist sehr langsam, daher ($t_v = 20$ ms $\quad t_c = 490$ μs).

`DDC_PID` Berechnung der Ventil-Steuergröße `u` mit Hilfe des DDC-PID-Algorithmus unter Verwendung der gerade aktuellen Regelparameter ($t_v = t_c = 4$ ms).

`u_Ausgeben` Ausgabe der Ventil-Steuergröße `u` über eine D/A-Karte an das Rohrventil ($t_v = 10$ ms $\quad t_c = 490$ μs).

`PID_Rechnen` Berechnung neuer PID-Regelparameter aus 100 Durchfluss-Zeit-Wertepaaren ($t_v = t_c = 1500$ ms).

`D_t_Schreiben` Ein Durchfluss-Zeit-Wertepaar wird auf einen gemeinsamen Speicherbereich geschrieben ($t_v = t_c = 10$ μs).

`D_t_Lesen` Ein Durchfluss-Zeit-Wertepaar wird von einem gemeinsamen Speicherbereich gelesen ($t_v = t_c = 10$ μs).

`PID_Schreiben` Ein Satz PID-Parameter wird auf einen gemeinsamen Speicherbereich geschrieben ($t_v = t_c = 10$ μs).

`PID_Lesen` Ein Satz PID-Parameter wird von einem gemeinsamen Speicherbereich gelesen ($t_v = t_c = 10$ μs).

`D_t_Speichern` Schreiben von 100 Durchfluss-Zeit-Wertepaaren in eine Datenbank. Zeitdauer insgesamt $t_v = 1$ s bis 2 s, CPU-Zeit $t_c = 10$ ms.

Das Schreiben auf und Lesen von einem gemeinsamen Speicherbereich erfolgt immer unter gegenseitigem Ausschluss. Das betrifft also die Methodenpaare `D_t_Schreiben`/`D_t_Lesen` und `PID_Schreiben`/`PID_Lesen`.

Aufgaben:
a) Welche Prozesszeiten t_p gibt es in diesem Prozess? Geben Sie die Werte an.

b) Wie kann durch geeignetes Tasking und Verteilung von Prioritäten die Echtzeitbedingung möglicherweise erfüllt werden? (Wenn Sie mit Prioritätszahlen arbeiten, geben Sie deutlich an, was bei Ihnen hohe und niedrige Priorität ist.)
(Antwort z.B. in folgender Art (der Form nach, Inhalt ist falsch) möglich:

```
TASK P Prio12       TASK T Prio20       TASK Prio33
jede 5 ms           jede 20 ms          Forever
T_Erfassen          D_t_Erfassen        for i=1 to 100
D_t_Erfassen        T_Erfassen             DDC_PID
T_Vergleich         D_t_Schreiben          D_t_Lesen
u_Ausgeben          PID_Schreiben       PID_Lesen            )
```

c) Berechnen Sie die relative Gesamtbelastung der CPU :

d) Offensichtlich kann die Echtzeitbedingung bei sequentieller Programmierung nicht eingehalten werden. Besteht auf Grund des errechneten Wertes für die relative Gesamtbelastung Hoffnung, dass die Echtzeitbedingung durch geeignetes Tasking eingehalten werden kann?
Antwort (ja/nein) mit kurzer Begründung:

e) Ist Ihr Wert für die relative Gesamtbelastung eher als kritisch zu betrachten?
Antwort (ja/nein) mit kurzer Begründung:

f) Welch ein gemeinsamer Speicherbereich würde sich für die Durchfluss-Zeit-Wertepaare eignen, d.h. welche Schreib- bzw. Lese-Eigenschaften muss der Speicher haben und wie groß müsste dieser mindestens gewählt werden?

g) Welch ein gemeinsamer Speicherbereich würde sich für die PID-Regelparameter eignen, d.h. welche Schreib- bzw. Lese-Eigenschaften muss der Speicher haben und wie groß müsste dieser mindestens gewählt werden?

h) Ist die Echtzeitbedingung erfüllt? Zur Beantwortung dieser Frage füllen Sie bitte unten stehende Tabelle aus. Die Tabelle ist für 5 Tasks ausgelegt. Wahrscheinlich haben Sie weniger Tasks verwendet. Schreiben Sie komplexere Berechnungsausdrücke für die Zahlen der Tabelle *unter* die Tabelle.

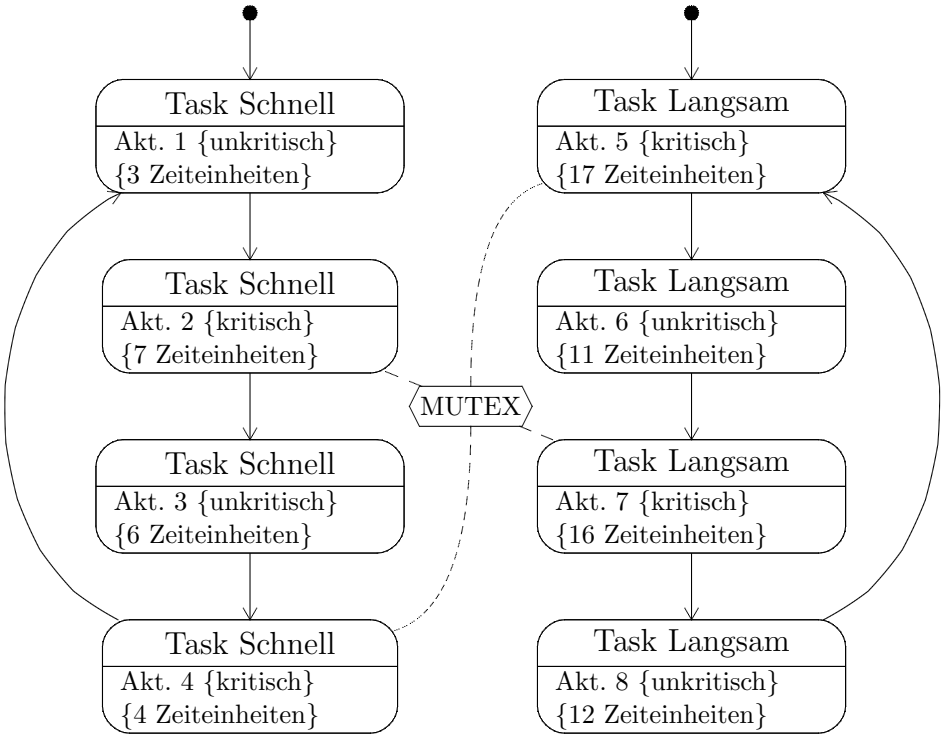
Task					
Zeit-einheit					
t_p					
t_z					
t_c					
t_s					
t_v					
t_w					
t_b					
t_r					
EZB?					

i) Nun machen wir noch eine zusätzliche Annahme: Die Temperaturregelung und die Durchflussregelung greifen über dieselbe serielle Schnittstelle auf den Prozess zu. Das betrifft die Methoden `T_Erfassen`, `H_Ausgeben`, `D_t_Erfassen` und `u_Ausgeben`. Die Zugriffe auf die Schnittstelle erfolgen unter gegenseitigem Ausschluss. Jeder Zugriff dauert 1 ms. `D_t_Erfassen` und `u_Ausgeben` benötigen mehrere solcher Zugriffe, höchstens aber 5. (Am Anfang und dann wird alle 5 ms abgefragt.)
Schätzen Sie die Zeiten t_{b_i} ab.

j) Diskutieren Sie an diesem Beispiel die "harte" und "weiche" Echtzeitbedingung.

8.11.2 Gegenseitiger Ausschluss

In folgendem Beispiel haben eine schnelle Task (`Schnell`) und eine langsame Task (`Langsam`) kritische und unkritische Aktivitäten. Die Aktivitäten verbrauchen keine CPU-Zeit, haben aber eine – hier in Ticks (irgendwelche Zeiteinheiten) angegebene – Dauer. Einige Aktivitäten sind "kritische" Aktivitäten, die nur unter gegenseitigem Ausschluss durchgeführt werden können und durch einen MUTEX geschützt werden.



Aufgabe 1: Machen Sie sich mit Hilfe eines Timeline- oder Sequenz-Diagramms klar, wann welche Aktivitäten laufen.

Aufgabe 2: Schätzen Sie auf Grund des Diagramms ab, was die jeweilige Worst-Case-Zyklusdauer ist.

Aufgabe 3: Schätzen Sie die jeweilige Worst-Case-Zyklusdauer unter Verwendung der Formeln im Abschnitt "Reaktionszeit" ab.

Aufgabe 4: Programmieren Sie eine Simulation.

Kapitel 9

Echtzeitsystem-Entwicklung

Dieses Kapitel diskutiert Aspekte des Software-Engineerings, die speziell bei Echtzeitsystemen (eingebettete Systeme) oder generell **nebenläufigen *(concurrent)*** Systemen eine Rolle spielen und über die bisher genannten hinausgehen.

Echtzeitsysteme sind Automatisierungssysteme, die einen technischen Prozess steuern oder automatisieren. Diese Prozesse sind charakterisiert durch:

- Technische Prozesse bestehen aus vielen (Teil-)Prozessen.
- Die Prozesse laufen parallel **(Nebenläufigkeit)**.
- Die Prozesse unterliegen gewissen **Zeitbedingungen**.
- Prozesse sind Einzelkomponenten **(Task)** des Entwurfs.
- Prozesse steuernde Programme sind sequentielle Programme. Für ihre interne Strukturierung können Entwicklungsprinzipien des sequentiellen Entwurfs verwendet werden.
- Bei technischen Prozessen müssen **Sicherheits-Aspekte** berücksichtigt werden.

Die besondere Herausforderung bei der Entwicklung eines Echtzeit-Systems liegt darin, dass in der Regel sehr eng mit anderen Disziplinen – ("Hardware") **mechanisches Engineering** oder **elektrisches Engineering** – zusammengearbeitet werden muss. Dabei zeigt sich als Tendenz, dass immer mehr Funktionen der Hardware-Disziplinen durch Software realisiert werden.

Moderne Echtzeit-Software wird objekt-orientiert entwickelt. Dazu wird UML eingesetzt. Bezüglich UML gehen wir hier nur auf die speziellen Anforderungen von Echtzeitsystemen und deren Modellierung ein. Ein wichtiger Grund für den Erfolg des objektorientierten Ansatzes ist eine klare Unterscheidung von

> "**was** ein Objekt tut" (Abstraktion, Schnittstelle eines Objekts)
> und
> "**wie** ein Objekt etwas tut" (Realisierung, Implementierung, Repräsentation eines Objekts).

Dieses Prinzip wird unterstützt durch die konsequente Verwendung von Schnittstellen.

Echtzeitsysteme sind von Natur aus **nebenläufig, asynchron** und **verteilt** ***(concurrent, asynchronous, distributed)***. Die Steuerungsrechner von Robotern an einer Bandstraße laufen parallel nebeneinander. Die Abarbeitung erfolgt asynchron, d.h. es gibt i.A. keine feste Reihenfolge der Ereignisse verschiedener Roboter. Allerdings werden sich die Roboter gelegentlich synchronisieren, d.h. eventuell aufeinander warten. Ferner sind die Roboter räumlich verteilt.

Bei einem mechanischen technischen Prozess hat der mechanische Teil typischerweise eine Lebenszeit zwischen 5 und 20 Jahren, während die verwendete Hard- und Software-Technologie nur eine Lebenszeit zwischen 3 und 5 Jahren hat. Daher spielt der Aspekt der Portierung von Software von einem veralteten HW-SW-System zu einem moderneren System ein große Rolle.

9.1 Benutzerschnittstelle

Bei Echtzeitsystemen kann die Benutzerschnittstelle sehr unterschiedlich sein. Die Wechselwirkung mit einem menschlichen Bediener reicht vom Betätigen eines Schalters bis zu einer voll-animierten graphischen Oberfläche oder extrem spezialisierten Schnittstellen.

In der Luftfahrtindustrie ist das Prinzip ***fly-by-wire*** schon verwirklicht. Die Piloten haben keine direkte Verbindung mehr zum mechanischen System des Flugzeugs. Alles läuft über einen oder mehrere Computer. In der Automobilindustrie entwickelt man dieses Prinzip ***(drive-by-wire)*** gerade.

Wenn die Schnittstelle Mensch-Prozess gut gemacht ist, dann erleichtert sie ganz wesentlich die Prozessbedienung. Wenn sie aber schlecht und fehlerhaft ist, dann führt das zu Sicherheitsproblemen, insbesondere in Stress-Situationen oder bei ungewöhnlichen Zuständen.

9.2 Analyse

Bei Echtzeitsystemen hat sich der Begriff **Essentielles Modell** mit den Teilen **Umgebungs-Modell** und **Verhaltens-Modell** etabliert.

Das Umgebungs-Modell ist die Beschreibung des zu automatisierenden technischen Prozesses. Dieser bildet die "Umgebung" des zu entwickelnden Automatisierungssystems. Z.B. könnte hier eine Liste der externen Ereignisse (vom technischen Prozess kommend) und der externen Aktionen (auf den technischen Prozess wirkend) erstellt werden.

Das Verhaltens-Modell beschreibt, wie das Automatisierungs-System auf die Ereignisse zu reagieren hat.

Oft ist ein Datenflussdiagramm bei Echtzeitsystemen sehr nützlich (siehe Kapitel "Funktionales Modell").

9.3 Testen

Testen ist bei Echtzeitsystemen besonders schwierig, da der reale Prozess i.A. nicht unter beliebigen Szenarien gefahren werden kann. Der reale Prozess kann nicht bis zu seiner Zerstörung

gefahren werden. Daher muss oft simuliert werden, was die Erstellung eines Simulationsmodells erfordert, das aber immer auch eine gewisse "Entfernung" vom realen Prozess hat.

9.4 Simulation

Man scheint sich darüber einig zu sein, dass die Einführung eines Simulationsschritts insgesamt Zeit spart.

Die Entwicklung eines Simulationsmodells wird häufig als überflüssige Projektverzögerung gesehen. Denn es ist sehr aufwendig, ein solches Modell zu entwickeln. Aber alle Probleme, die in einer "freundlichen" Simulationsumgebung erkannt werden, können dort viel ökonomischer gelöst werden.

Es kann allerdings sein, dass die Übertragung eines Systems von der Simulationsumgebung in die Realität nicht trivial ist.

Das Simulationsmodell sollte nicht mehr Detail enthalten als für den Test der Steuerungssoftware notwendig ist.

Ein Simulationsmodell fördert das Prozessverständnis.

9.5 Agenten-Modell

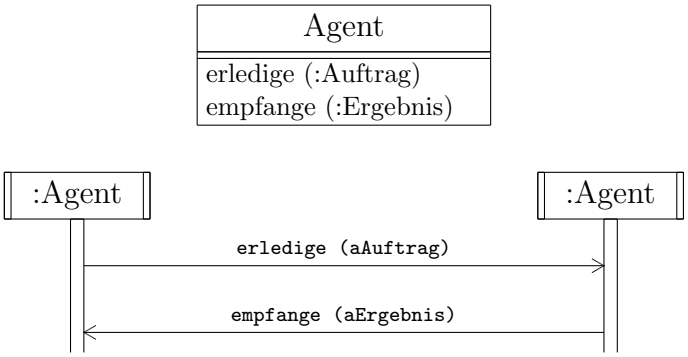
Wir modellieren Echtzeitsysteme mit **aktiven Objekten** (UML-Konstrukt) oder **Prozessen**. In der Echtzeitliteratur wird dafür oft der Begriff **Aktor** ***(actor)*** [32] verwendet, der in UML und auch sonst in der Prozessdatenverarbeitung allerdings eine andere Bedeutung hat. Wir entscheiden uns daher hier für den Begriff **Agent**. Der Agent ist ein aktives Objekt und der Begriff "Agent" unterstreicht die **Autonomie** des Objekts. Außerdem wird eine gewisse "Intelligenz" des Agenten suggeriert. Ein Agent kann eine oder mehrere Funktionen oder Methoden als **Task** durchführen, d.h. nebenläufig zu anderen Tasks sein.

Das Agenten-Modell ist sehr allgemein. Ein Agent kann z.B. einen Prozess, eine Steuereinheit, einen Rechner, einen Speicherbereich, einen Sensor oder Aktor repräsentieren.

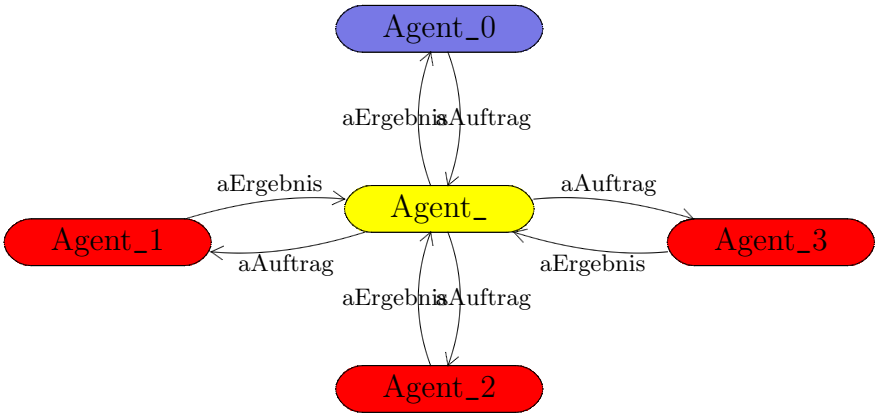
9.5.1 Spezifikation des Agenten-Modells

Agenten kommunizieren nur über **asynchrone Botschaften** ***(asynchronous message passing)***. Ein Agent muss also nicht auf eine Antwort seiner Botschaft an einen anderen Agenten warten, sondern kann seine Aktivitäten parallel weiter ausführen.

Typischerweise erhält der Agent als Botschaft **Aufträge** anderer Agenten, die dann asynchron ein **Ergebnis** von ihm erwarten. Der Agent kann den Auftrag entweder selbst komplett bearbeiten oder er beauftragt weitere Agenten.



Als Datenflussdiagramm (siehe Kapitel "Funktionales Modell") könnte das folgendermaßen aussehen:



Das Senden einer Botschaft an einen Agenten ist als **Ereignis** ***(event)*** definiert. Es gibt keine globale Reihenfolge von Ereignissen.

Regeln für den Botschaftenaustausch:

1. Botschaften werden ***point-to-point*** versendet, d.h. von Agent zu Agent.
2. Das Senden einer Botschaft ist eine ***non-blocking*** Operation. Der Sender wartet nicht auf eine Antwort, sondern kann seine Programmausführung sofort wieder aufnehmen.
3. Gesendete, aber noch nicht verarbeitete Botschaften werden gepuffert. Botschaften gehen nicht verloren. Sie werden beliebig lang gepuffert. (Im Gegensatz dazu werden **Signale** nicht gepuffert. Wenn Signale nicht empfangen werden, gehen sie verloren.)
4. Gesendete Botschaften erreichen garantiert ihr Ziel, aber möglicherweise verzögert durch den Kommunikationsweg. Eine Reihenfolge der Botschaften wird nicht garantiert.
5. Eine Botschaft enthält den Absender, damit irgendwann eine Antwort an den Absender geschickt werden kann.

9.5.2 Realisierung des Agenten-Modells

Zur Diskussion der Realisierungsmöglichkeiten des Agenten-Modells beschränken wir uns auf Java.

Die Aktivität des Agenten wird als Thread realisiert, der gegebenenfalls Botschaften an andere Agenten verschickt.

Eine Botschaft ist ein Objekt einer Klasse, die vorzugsweise eine entsprechende Schnittstelle, etwa die Schnittstelle `Botschaft` oder `Auftrag/Ergebnis` realisiert.

Botschaften werden versendet, indem eine Methode eines anderen Agenten mit dem Botschaftsobjekt als Argument aufgerufen wird. Diese Methoden könnten

```
empfange (Botschaft x)
```

oder spezifischer

```
erledige (Auftrag x)
empfange (Ergebnis x)
```

heißen.

Als Transport-Medium kann natürlich eine Message-Queue verwendet werden.

Bei verteilten Systemen kann dies über RMI oder – elementarer – über UDP-Botschaften realisiert werden.

Um Botschaften (Ergebnis oder Aufträge) zu empfangen, muss ein Agent die Methoden `empfange (Botschaft x)` (`empfange (Ergebnis x)` oder `erledige (Auftrag x)`) realisieren. Diese Methoden werden die Botschaft (das Ergebnis oder den Auftrag) nur in einen Puffer schreiben, typischerweise einen Ringpuffer, der prinzipiell unendlich groß werden kann.

Wie erfährt nun der Agent, d.h. der Thread des Agenten ("Agenten-Thread") von dem Eingang einer Botschaft? Wahrscheinlich ist es am elegantesten, wenn ein zweiter "Lese-Thread" kontinuierlich versucht, Botschaften aus dem Ringpuffer zu lesen. (Ein vernünftiger Ringpuffer wird diesen Thread suspendieren, wenn keine Botschaften zu lesen sind.) Der Lese-Thread kann dann den eigentlichen Agenten-Thread benachrichtigen und zur Behandlung des Ereignisses auffordern. Das erfolgt synchron, d.h. der Lese-Thread wird warten, bis das Ereignis behandelt ist, und dann die nächste Botschaft lesen. Der Lese-Thread kann eventuell auch entscheiden, ob eine Botschaft gerade relevant ist oder verworfen werden kann. (Ein Zurückstellen in den Puffer dürfte nicht sinnvoll sein, da dann der Lese-Thread dauernd mit Lesen und wieder Zurückstellen beschäftigt wäre.)

9.6 Beenden von Tasks

Es ist normalerweise nicht problematisch eine Task zu starten. Auch das Ende einer Task ist unproblematisch, wenn sie an ihr natürliches Ende läuft. Aber es gibt Fälle, wo Tasks von außen angehalten oder abgebrochen werden müssen. Die Schwierigkeit dieser Fälle wird i.A. unterschätzt. Die dafür in vielen Systemen angebotenen (auf Tasks anwendbare) Methoden `suspend/resume`

und `kill`, `cancel` oder `stop` sind gefährlich und führen zu Laufzeitproblemen, die kaum zu lösen sind. Bei diesen Methoden kann eine Task nicht die Aufräumarbeiten durchführen, die normalerweise vor einem Abbruch immer notwendig sind.

Anstatt eines "gewaltsamen" Eingriffs in den Ablauf der Task, sollte ein **kooperativer** Mechanismus gewählt werden, wobei der Task ein Interrupt geschickt wird, den sie selbst behandelt. Da dies das Design und die Implementierung verkompliziert, wird diese Problematik – Beendingung von Tasks ***(end-of-lifecycle issues, graceful shutdown)*** – gern vernachlässigt.

Der kooperative Mechanismus funktioniert i.A. folgendermaßen:

1. **Taskstruktur:** Die Task verwaltet eine ihr zugeordnete **Zustandvariable** mit folgenden Werten:
   (a) `laufend` ***(running)***
   (b) `beendet` ***(terminated)***
   (c) `suspendiert` ***(suspended)***

   Diese Zustandsvariable kann von außerhalb der Task *nur* gelesen werden.

   Ferner verwaltet die Task eine ihr zugeordnete **Zustandsänderungsvariable** mit folgenden Werten:

   (a) `beenden` ***(cancel)***
   (b) `suspendieren` ***(suspend)***
   (c) `weiter` ***(resume)***

   Die Zustandsänderungsvariable kann von außerhalb der Task gesetzt werden.

2. **Taskverhalten:** Die Task überprüft in passenden, aber möglichst kurzen Zeitabständen die Zustandsänderungsvariable. Entsprechend dem Wert dieser Variablen wird die Task entweder
   (a) einfach weiterlaufen (im Zustand `laufend` auf `weiter` gesetzt) oder
   (b) sich "vernünftig" beenden (auf `beenden` gesetzt) oder
   (c) "vernünftig" anhalten (auf `suspendieren` gesetzt) oder
   (d) aus einer Suspendierung weiterlaufen (im Zustand `suspendiert` auf `weiter` gesetzt).

   Die Task aktualisiert bei Erreichen eines neuen Zustands die Zustandsvariable.

   Die Task darf im Zustand `laufend` durchaus auch "schlafen" oder sich anderweitig suspendieren, wenn es dafür einen Interrupt-Mechanismus gibt.

   Nach einem Interrupt muss auf jeden Fall die Zustandänderungsvariable überprüft werden.

3. **Abbruch** ***(kill, stop, cancel)*** oder **Beenden** einer Task:
   (a) Die Zustandsänderungsvariable der Task wird auf `beenden` gesetzt.
   (b) Der Task wird ein Interrupt geschickt, um sie eventuell zu wecken.

4. **Suspendierung** ***(suspend)*** einer Task:
   (a) Die Zustandsänderungsvariable der Task wird auf `suspendieren` gesetzt.

   (b) Der Task wird ein Interrupt geschickt, um sie eventuell zu wecken.

5. **Weiterlaufenlassen *(resume)*** einer Task:

   (a) Die Zustandsänderungsvariable der Task wird auf `weiter` gesetzt.

   (b) Der Task wird ein Interrupt geschickt für den Fall, dass sie schläft, was sie in dem Fall ja auch tun sollte.

Bemerkungen:

Erzeuger-Verbraucher-Kommunikation: Oft ist es sehr schwierig eine Erzeuger-Verbraucher-Kommunikation über eine Queue sicher herunterzufahren, wenn die Schreib- und Lese-Methoden der Queue nicht unterbrechbar sind, was oft der Fall ist. Bevor man eigene Queues schreibt, wird man verschiedene Tricks versuchen. Oft wird eine sogenannte Giftpille ***(poison pill)*** verwendet, die der Erzeuger bei Abbruch der Kommunikation in die Queue schreibt. Der Verbraucher bricht dann ab, wenn er die Giftpille gelesen hat. Das Verfahren kann sich erheblich verkomplizieren, wenn man mehrere Erzeuger und Verbraucher hat (siehe [11] Seite 147).

Task Leakage: Man verliert eine Task, weil sie auf Grund eines Fehlers abgebrochen wird. Das wird oft nicht bemerkt, wenn diese Task eine von vielen gleichartigen Tasks ist, wenn z.B. ein Service mehrfach gestartet wird. Hier muss man eventuell Mechanismen einbauen, die das Verschwinden einer Task signalisieren.

Kapitel 10

Echtzeitsprachen

Wenn man ein Echtzeitbetriebssystem zur Verfügung hat, dann kann man in einer konventionellen Sprache wie Pascal, C, C++, FORTRAN Echtzeitsysteme programmieren, indem die Systemaufrufe des Echtzeitbetriebssystems verwendet werden. Heutzutage ist die Kombination mit C und in zunehmendem Maße mit C++ üblich.

Ohne Echtzeitbetriebssystem muss man eine *Echtzeitsprache* verwenden, die all die Mechanismen zur Verfügung stellt, die wir bei den Echtzeitbetriebssystemen diskutiert haben.

Der prominenteste Vertreter ist die Sprache "Ada", die sich dadurch auszeichnet, dass sie das Rendezvous als Synchronisationsmechanismus zur Verfügung stellt. Sie wird besonders im militärischen Bereich eingesetzt. Allerdings hört man von Ada nichts mehr.

"Concurrent C(++)" ist eine Echtzeiterweiterung von C++ und bietet ähnliche und mächtigere Mechanismen als Ada an, hat sich aber nicht durchgesetzt.
Seit 2011 gibt es aber C++11, das alle Multitasking-Mechanismen anbietet.

Die bundesdeutsche Entwicklung "PEARL" hat sich nicht durchgesetzt.

Zu nennen sind noch "FORCE" und die Pascal-Abkömmlinge "Modula" und "Oberon", die gelegentlich in der Prozessautomatisierung eingesetzt werden.

Viele andere Sprachen haben nur akademische Verbreitung gefunden.

Die objekt-orientierten Programmiersprachen **Java** und `C#` entwickeln sich zur Zeit rasant und bieten alle Mechanismen, die man zur Programmierung von Echtzeitsystemen benötigt. Zur Zeit mag es noch Geschwindigkeitsprobleme geben, die aber durch den Einsatz geeigneter Compiler oder Hardware in der nächsten Zukunft behoben werden. Nach meiner Einschätzung werden in naher Zukunft Echtzeit-Entwicklungen bevorzugt in diesen Sprachen erfolgen. Alle namhaften Echtzeit-Betriebssysteme unterstützen Java.

Kapitel 11

Echtzeitbetriebssystem

Technische Prozesse sind dadurch charakterisiert, dass viele Teilprozesse gleichzeitig ablaufen. Für das Prozessautomatisierungssystem bedeutet das, dass viele Mess-, Steuer- und Regelaufgaben parallel durchgeführt werden müssen. Der Begriff *Echtzeit* kommt daher, dass Reaktionen auf Prozess-Alarme innerhalb vorgegebener Zeiten eine häufig gestellte Anforderung an das Prozessautomatisierungssystem sind.

Daher werden die technischen Teilprozesse durch entsprechende (quasi-)parallel laufende Rechenprozesse oder *Tasks* modelliert. Die Programmieraufgaben, die bei der Verwaltung solcher Prozesse anfallen, sind äußerst schwierig, aber auch immer wieder dieselben Aufgaben. Daher wird die Lösung dieser Aufgaben üblicherweise in das Betriebssystem verlagert.

11.1 Anforderungen an Echtzeitbetriebssystemkerne

Das beste Modell für die interne Struktur eines Betriebssystems ist eine Schichtenhierarchie, auf deren niedrigster Ebene der *Kern* liegt. Der Kern umfasst die wichtigsten, sogenannten "low level" Funktionen des Betriebssystems. Der Kern bildet die Schnittstelle zwischen der Hardware des Rechners und der übrigen Software des Betriebssystems. Der Kern sollte einerseits die einzige Hardware-abhängige Komponente des Betriebssystems sein, andererseits sollte er nur einen minimalen Satz an Operationen zur Verfügung stellen, aus denen der Rest des Betriebssystems konstruiert werden kann.

Heutzutage sind wichtige Anforderungen **Safety** und **Security**, die im IT-Kontext nur als englische Begriffe verwendet werden und üblicherweise folgendermaßen unterschieden werden:

- **Safety:** Sicheres Funktionieren des Systems, insbesondere ohne Gefährdung von Menschen. Zuverlässigkeit. Robustheit. Datensicherheit. Datenkonsistenz.
- **Security:** Sicherheit des Systems vor Attacken von außen. System-Schutz. Datenschutz. Cyber-Security.

Betriebssystemkerne, die in Prozessrechnern eingesetzt werden, heißen *Echtzeitkerne* oder *Realzeitkerne* und sollten folgende Anforderungen erfüllen [25]:

Multitasking: Viele konkurrierende Tasks (laufende Programme, Rechenprozesse) müssen quasiparallel von der CPU abgearbeitet werden. Der *Scheduler* teilt die CPU den Tasks gemäß einer Strategie zu. Die Tasks dürfen sich einerseits nicht stören, andererseits müssen sie auch kommunizieren ***(competing and communicating processes)***. Zur Steuerung dieser Aktivitäten muss der Betriebssystemkern Mechanismen wie *Semaphore, Monitore oder Rendezvous* zur Verfügung stellen.

Priority Based Preemptive Scheduling: Die Ereignisse der realen Welt haben unterschiedliche Wichtigkeitsgrade oder *Prioritäten*. Diese Prioritäten müssen bei der Zuteilung der CPU berücksichtigt werden. Beim prioritätsbasierten unterbrechenden Scheduling wird der Task, die die höchste Priorität hat und die CPU benötigt, die CPU auch zugeteilt, wobei niedriger priorisierte Tasks unterbrochen werden können.

Preemption-Threshold ist eine Eigenart von ThreadX. Wenn z.B. eine Thread eine Priorität von 30 hat und eine Preemption-Threshold von 23, dann kann sie von allen Threads mit Prioritäten 0 bis 22 unterbrochen werden, nicht aber von Threads mit Prioritäten größer 23. Sie selbst kann nur Threads mit Prioritäten größer 30 unterbrechen.

Control Loop with Polling Scheduling: In einer festgelegten Reihenfolge fragt der RTOS-Kern die Tasks, ob sie einen Dienst benötigen. Wenn das der Fall ist, bekommt die Task den Prozessor zur Ausführung des Dienstes, führt den Dienst aus und gibt die Kontrolle zurück an den Kern. Das ist ein sehr kooperativer Ansatz und ist auch unter ***cooperative scheduling*** bekannt. Ein Variante davon ist das **Zeitscheibenverfahren** ***(time-slicing)***, wo einer Task die CPU nur für ein kurzes Zeitintervall zur Verfügung steht mit dem Nachteil, dass man sehr viele, oft unnötige Kontextwechsel hat.

Diese Scheduling-Varianten kommen oft zum Einsatz bei Tasks gleicher Priorität, wenn ansonsten ein Priority Based Preemptive Scheduling verwendet wird.

Intertaskkommunikation (Interprozesskommunikation): Da die Tasks Nachrichten austauschen, müssen dafür effektive Mechanismen wie z.B. Pipes, Queues ***(asynchronous or synchronous message passing)*** oder Shared Memory zur Verfügung gestellt werden.

Synchronisationsmechanismen: (Locks, Semaphore, Rendezvous oder Monitore) müssen für die gemeinsame bzw. ausschließliche Nutzung von kritischen Ressourcen zur Verfügung gestellt werden.

Leistungsfähigkeit: Ein Echtzeitkern muss auf den "Worst Case" und nicht auf maximalen Durchsatz optimiert werden. Ein System, das eine Funktion konsistent in 50 μs ausführt, ist für die Prozesssteuerung eher geeignet als ein System, das für diese Funktion durchschnittlich 20 μs, in Ausnahmefällen aber 80 μs benötigt.

Interrupt: Es muss möglich sein, die Abarbeitung eines Interrupts außerhalb der eigentlichen Unterbrechungsschicht (ISR, *interrupt service routine*) durchzuführen. Das System muss auf einen Interrupt möglichst schnell reagieren, die Abarbeitung aber an andere Tasks weiterreichen können, damit es auf weitere Interrupts ebenso schnell reagieren kann.

Task, Prozess, Thread: Es werden die Begriffe **Task**, **Prozess** (hier nicht als "zu steuernder Prozess", sondern als "Rechenprozess") und ***Thread*** verwendet. Für uns ist Task der übergeordnete Begriff.

Ein (Rechen-)Prozess hat einen eigenen Adressraum. Diese Prozesse können nur über Betriebssystem-Mittel kommunizieren. Prozesse werden vom Betriebssystem verwaltet.

Threads dagegen sind Leichtgewichtsprozesse ***(lightweight process)*** ohne eigenen Adressraum. Sie können auch über globale Variable kommunizieren. Mehrere Threads können zu einem Prozess gehören. Der **Context-Switch** auf Thread-Ebene ist wesentlich einfacher als auf Prozess-Ebene.

Ein *Programm* ist eine statische Aufschreibung einer Folge von Anweisungen, während die *Task* als Programm*ablauf* zu verstehen ist.

Objekt-orientiert ist eine Task ein Objekt einer Klasse, die eine Methode hat, deren Code als Thread oder Prozess gestartet werden kann.

11.2 Typische Werkzeuge von Echtzeitbetriebssystemen oder Sprachen

11.2.1 Pipes, Queues

(siehe Kapitel Concurrency Utlilities und Kapitel Threads Ringpuffer)

11.2.2 Watchdog Timer

Ein Watchdog Timer ermöglicht den Aufruf einer Routine nach einer vorgegebenen Zeitverzögerung d. Innerhalb des Zeitintervalls d kann der Aufruf der Routine zurückgenommen werden bzw. kann der Timer zurückgesetzt werden ("padding des Hundes").

I.A. ist das so realisiert, dass die Funktion normalerweise mit `true`, im Abbruchsfall aber mit `false` zurückkommt.

11.2.3 Timetables

Startzeiten von Tasks erfolgen nach einem festgelegten Zeitplan.

11.3 Vergleich von Echtzeitbetriebssystemen

Ein belgische Firma testet und vergleicht Betriebssysteme (`www.dedicated-systems.com`). Das dürfte bei einer Betriebssystem-Entscheidung sehr interessant sein.

Bei einem Vergleich spielen insbesondere zwei Zeiten eine Rolle:

- ISR (etwa 1μs): Zeit, bis erster Befehl in einer Interrupt Service Routine nach einem Interrupt durchgeführt wird.
- IST (etwa 10μs): Zeit, bis in einer Interrupt Service Routine nach einem Interrupt ein Thread gestartet wurde.

Grob unterscheiden wir drei Architekturen:

- ***flat architecture:*** Betriebssystemkern und Anwendung befinden sich auf derselben (der einzigen) Privilegienebene. Anwendung und Betriebssystemkern sind nicht klar unterscheidbar. Es gibt keinerlei Speicherschutz.
- ***monolithic architecture:*** (WinCE) Betriebssystemkern enthält alle Bibliotheken, auf die die Anwendung direkt zugreift. Es gibt mindestens zwei Privilegienebenen, Kern und User. Für die Kern-Prozesse gibt es keinen Speicherschutz.
- ***micro kernel architecture:*** (VxWorks, QNX) Betriebssystemkern enthält keine Bibliotheken. Diese befinden sich auf einer eigenen Ebene. Die Anwendung kann aber nur über den Kern auf die Bibliotheken zugreifen. Voller Speicherschutz.

Kapitel 12

Feldbusse

Der Feldbus verbindet einzelne Sensoren und Aktoren. Die einzelnen Knoten haben wenig Intelligenz. Ihre wesentliche Aufgabe ist die A/D- oder D/A-Wandlung.

Im automtive Bereich hat sich für normale Anwendungen der CAN-Bus durchgesetzt, für anspruchsvollere Anwendungen wird der FlexRay-Bus eingesetzt.

12.1 CAN-Bus

Kein Ring, Zweidrahtleitung 40 m, optische Kabel soll es auch bald geben, DeviceNet von Allan-Bradley

12.2 FlexRay-Bus

12.3 Lightbus

Ring, 45m zwischen den Knoten (Kunststoffkabel).

12.4 Interbus-S

Kapitel 13

E/A-Schnittstellen

Die Kopplung des Prozeßrechners an den Prozeß erfolgt über verschiedene Eingabe-/Ausgabe-Schnittstellen (E/A-Schnittstellen oder I/O-Schnittstellen), die über Register angesprochen werden. Wir wollen zwei Arten von Schnittstellen unterscheiden:

1. Schnittstellen, über die elektrische Prozeßsignale direkt ein- oder ausgegeben werden.
2. Schnittstellen, mit denen Meß- oder Stellgeräte des Prozesses angesprochen werden können.

Direkte Prozeßsignale sind

- entweder boolesch (digital): Nur zwei Spannungs- oder Stromniveaus können angenommen werden (0 und 5 V (TTL), 0 und 24 V, 0 bzw 4 und 20 mA). Aktorisch können z.B. Motoren, Heizungen, Lampen an- oder abgeschaltet werden, Ventile oder Relais geschlossen oder geöffnet werden. Sensorisch kann z.B. der Zustand von Lichtschranken oder Endschaltern erfaßt werden.

  Die digitalen Signale werden über digitale E/A-Kanäle ein- oder ausgegeben, wobei meistens 8, 16 oder 32 Kanäle parallel bedienbar sind. Spezialfälle sind die Centronics-Schnittstelle oder das GPIB-Interface. Es gibt für die verschiedenen Mikroprozessorsysteme ein breites Angebot an digitalen I/O-Karten.

- oder kontinuierlich (analog): Es werden Spannungen typischerweise zwischen 0 und 10 Volt oder zwischen -5 und +5 Volt mit einem A/D-Wandler erfaßt oder über einen D/A-Wandler ausgegeben. Ein noch häufiger angewandter Standard sind Ströme zwischen 0 bzw 4 und 20 mA, die allerdings vor der A/D-Wandlung durch einen geeigneten Widerstand in Spannungen umgewandelt werden. Ströme können über größere Entfernungen störungsfrei übertragen werden (Stromschleifen). Eine Unterbrechung des Stromkreises kann sofort durch Stromlosigkeit detektiert werden.

  Bei den genannten Spannungs- und Stromwerten handelt es sich um vorverarbeitete (meist verstärkte) Signale. Natürlich kommt es auch vor, daß die Meßeffekte direkt verarbeitet werden müssen.

  Als Analogwerte können z.B. Konzentrationen, Temperaturen, Drücke, Abstände, Kräfte erfaßt werden oder z.B. Führungsgrößen für Regler und kontinuierliche Stellgrößen ausgegeben werden.

- oder Spezielle Signale. Z.B. Videosignale, die von speziellen Karten verarbeitet werden.

"Intelligente" Meß- und Stelleinheiten können über folgende Schnittstellen angesprochen werden:

- Serielle RS232- oder V24-Schnittstelle
- Centronics-Schnittstelle
- SCSI-Schnittstelle
- Feldbus-Schnittstelle
  - GPIB- oder HPIB-Schnittstelle
  - VSI-Bus
  - CAN-Bus
  - BIT-Bus
  - Interbus-S
  - Lightbus
  - Profibus
  - Ethernet-TCP/IP
  - MAP
  - ZigBee

Dabei werden Feldbus-Systeme immer häufiger eingesetzt.

Kapitel 14

Aktorik

Aktuatoren ***(actuator)*** sind Stellglieder. Ihre Leistungsaufnahme kann mit einem Prozessrechensystem gesteuert werden. Dabei wird das Ausgangssignal des Prozessrechensystems i.a. verstärkt. Die elektrische Energie wird über einen Transducer in eine Prozessaktion umgewandelt.

Das Ausgangssignal kann digital (z.B. TTL-Niveau) oder analog (z.B. 0 bis 10 Volt) sein. Der Verstärker ist im digitalen Fall ein Schalter, im analogen Fall ein Analogverstärker. Der Transducer ist typischerweise ein Elektromotor.

Schalter

Bei den Schaltern unterscheiden wir drei Leistungsbereiche.

1. Niedrige Leistung (2 – 5V, 400 μA – 20 mA, $<$ 100 mW): Hier kann die digitale E/A-Schnittstelle ohne Verstärkung direkt als Schalter für den Transducer verwendet werden. Beispiele sind:

   - Leuchtdioden
   - Displays
   - kleinste Motoren
   - Reed-Relais

   Die Leistungsfähigkeit solch einer Schnittstelle hängt von der verwendeten Halbleitertechnik ab (TTL, CMOS, LSTTL (low power Schottky), HCCMOS (high speed)). Die Verdrahtung sollte so erfolgen, daß die Last zwischen Pluspol und E/A-Schnittstelle liegt. Als Stromsenke kann die Schnittstelle meist mehr Leistung vertragen. Im Einzelfall sollten die Datenblätter der Schnittstelle herangezogen werden.

2. Mittlere Leistung (bis etwa 150 Watt): Hier werden als Schalter

   - ICs (Transistoren) bis 80 V und 1,5 A
   - Leistungs-MOSFETs (metal oxide field effect transistor) bis 120 V bzw 10 A
   - Reed-Relais

verwendet.

3. Hohe Leistung (bis einige 100 KW): Hier werden als Schalter

   - Elektromotorische Relais, die ihrerseits nach 2) (typisch 12 V bei 0,5 A) angesteuert werden und Schaltzeiten im Millisekundenbereich haben,
   - Solid-State-Relais (Thyristoren), die typischerweise zur Schaltung von Wechselstrom (< 20 Adc, < 45 Aac) eingesetzt werden,

   verwendet.

Bemerkungen:

- Analoge Signale (Spannungsverläufe mit eventuell sehr niedrigen Spannungen) werden normalerweise durch elekromechanische Relais geschaltet, da Solid-State-Schalter i.a. bei niedrigen Spannungen nicht linear sind. Es gibt allerdings auch spezielle Analogschalter.
- Transiente Ströme (Einschaltströme) sind bei Lampen und Motoren zu beachten.
- Transiente Spannungen (induktive Abschaltspannungen) sind für die Funkenbildung bei elektromechanischen Relais verantwortlich.

Kontinuierliche Schalter (Analogverstärker)

Das Ausgangssignal eines D/A-Wandler liegt im mW-Bereich. Daher werden i.a. Leistungsverstärker benötigt:

1. bis einige 100 Watt Leistungsverstärker
2. Servoverstärker für Elektromotoren
3. Schrittmotorsteuerungen
4. programmierbare Netzgeräte (analog oder digital gesteuert, meist sehr langsam ≈ 1V/ms)
5. Puls-Breiten-Modulation: Hohe Leistung wird bei konstanter Frequenz, aber unterschiedlicher Pulsbreite (duty cycle) ein- und ausgeschaltet. Vorteil solch eines Verstärkers ist, daß er selbst kaum Leistung verbraucht.

Kapitel 15

Sensorik

Der in eine Spannung oder einen Strom gewandelte physikalisch-chemische Messeffekt muss i.a. unter Verwendung von Skalierfaktoren, Eichkonstanten, Kennlinien oder Modellen in eine physikalisch-chemische Messgröße transformiert werden. Bei Einsatz eines Prozessrechners kann dies sehr flexibel und sehr "intelligent" gehandhabt werden.

Beispiele für solche Transformationen sind:

- Linearisierung von Thermospannungen.
- Stark verrauschte Signale können durch (gleitende) Mittelwertbildung geglättet werden.
- Nicht direkt messbare Prozessgrößen können durch eine Modellanpassungsrechnung bestimmt werden (*modellgestützte Messung*). Einfachstes Beispiel war schon die Mittelwertbestimmung. Das nächst komplizierte Beispiel dürfte die Bestimmung einer Ausgleichsgerade sein, deren Steigung und Achsenabschnitt die gewünschten Messgrößen darstellen.

  Es ist allerdings zu bemerken, dass solche Berechnungen in immer stärkerem Maße von Mikroprozessoren durchgeführt werden, die in die Messgeräte integriert sind (*embedded systems*). Dadurch können die Messgeräte eine Vielzahl von Funktionsmöglichkeiten anbieten, die oft nur mit Hilfe des Prozessrechners sinnvoll eingesetzt werden können.

Beispiele:

1. Flüssigkeitsanalyse mit Ultraschall `www.sensaction.de` und `www.isat-coburg.de`

Kapitel 16

Prozess-Leitsysteme

Kapitel 17

Zuverlässigkeit

Kapitel 18

Bildverarbeitung

18.1 Einleitung

Die **Bildverarbeitung** ***(image processing, machine-vision-system)*** ist ein schwieriges Gebiet der Automatisierungstechnik mit riesigen Anwendungsmöglichkeiten. Größtes Problem der Roboter ist z.B., daß sie weitgehend blind sind. Das Gebiet ist schwierig, daß es über wesentliche Strecken ein Teilgebiet der Künstlichen Intelligenz ist.

Man schätzt, daß derzeit erst etwa 10% aller möglichen Anwendungen von Bildverarbeitungssystemen erschlossen sind. Anwendungsgebiete sind:

- **Fertigungskontrolle.** Beispiele: Identifikation, Vollständigkeitsprüfung, automatische Prüfung der Bedruckung und Bestückung von Platinen, Kontrolle der Verpackung, Prüfung von Schweißnähten.
- **Qualitätsprüfung.** Beispiele: Oberflächenkontrolle, Homogenitätsprüfung von Spinnvlies.
- **Optische Meßtechnik.** Überprüfung von Maßen und Formen. Beispiele: Vermessen von Autotüren in der Montage, Erfassen der Bandbreite von Walzblechen,Ermitteln der mittleren Korngröße von Schüttgütern.
- **Produktionsautomatisierung.** Beispiele: Positionserfassung und -regelung, Sortieren von Kleinteilen, Automatisches Auslagern von Motorblöcken, Führung von Schweißrobotern, Positionieren von Leiterplatten in einer Stanze.
- **Medizintechnik.** Beispiele: Automatische Auswertung von Röntgenaufnahmen.
- **Überwachung.** Beispiele: Zugangskontrolle für Personen oder Fahrzeuge, Authentifizierung am Rechner (Augenhintergrund als "Fingerabdruck").

Wesentliche Aufgabe der Bildverarbeitung oder besser des Bildverstehens ist die Transformation einer (großen) Menge von Lichtimpulsen in eine symbolische Darstellung **(Signal-zu-Symbol-Verarbeitung)**. Z.B. sollen die von einer Schraube reflektierten Lichtstrahlen in die Zeichenkette "Schraube, x=34, y=71, z=5" transformiert werden.

Bei schwierigen Problemen muß man alle Lösungsmöglichkeiten in Betracht ziehen. Insbesondere lohnt sich eine Orientierung an Mechanismen, die die Evolution der natur hervorgebracht hat. Man kann sich also zunächst mal fragen, wie sieht der Mensch?

Neugeborene Kinder können offenbar nicht sehen, obwohl sie Lichtimpulse wahrnehmen. Früher glaubte man, das läge daran, daß das Bild umgekehrt auf der Retina erscheint und das Kind eben erst lernen müsse, es umzudrehen. Heutzutage ist man eher der Auffassung, daß das Kleinkind zunächst fast garnichts sieht. Das **Gehirn ist ein Image-Processor**, der wahrscheinlich mit **spezieller Hardware** ausgestattet ist, die damit beginnt, daß die lichtempfindlichen Zellen auf der Retina nicht gleichmäßig verteilt sind, sondern **vorgeprägte Strukturen** aufweisen.

Wahrscheinlich ist das Sehvermögen ein **kreativer Prozeß**. D.h. einzelne Lichtimpulse werden zu immer komplexeren Einheiten zusammengefaßt. Erwartungen auf das zu Sehende spielen eine sehr große Rolle, die es erlauben mehrdeutige, verrauschte Bilder zu erkennen.

Mensch oder Tier sind keine passiven Beobachter. Das Sehen ist eine **aktive Tätigkeit**, bei der **Experimente** gemacht und **Hypothesen** erzeugt werden. Man bewegt sich oder die zu sehenden Gegenstände. Die Möglichkeit der physischen Wechselwirkung mit der Umwelt ist wichtig.

Den Prozeß des Bildverstehens kann man folgendermaßen gliedern:

- Sensoren liefern ein Signal.
- ***low level, early vision***:
  - Bestimmung lokaler Eigenschaften
  - Glättung
  - Schwellwertbestimmung ***(threshholding)***
  - Kantenelemente
  - Farbe (z.B. grau und rot)
  - Textur (z.B. Kopfsteinpflaster)
- ***intermediate Level***:
  - allgemeine Szenenattribute
  - Konturen
  - Regionen
  - Oberflächen (z.B. reflektierende)
  - Objekte (z.B. Räder)
- ***high level, late vision***:
  - symbolische Szenenbeschreibung (Sprache, Grammatik) (z.B. rotes Automobil ist auf einer Straße mit Kopfsteinpflaster)

Bild — Bildbearbeitung (Bildverarbeitung im engeren Sinne) → Bild — Mustererkennung → Symbolische Beschreibung — Graphik → Bild

Erläuterung der Begriffe:

- **Graphik** ***(graphics)***: Erzeugung von Bildern aus Nicht-Bild-Information (Input nichtbildlich, Output bildlich): CAD, Spiele, Szenen für Flugsimulatoren, Animation
- **Bildbearbeitung, Bildverarbeitung** ***(image processing)*** (Input und Output sind bildliche Information): Entfernung von Rauschen, Kontrastverstärkung, Datenreduktion
- **Mustererkennung** ***(pictorial pattern recognition)*** (Input bildlich, Output nichtbildlich): Beschreibung des Bildes, Zuordnung zu einer Klasse, OCR ***(optical character recognition)***, automatische medizinische Diagnose, Erkennung von Objekten

Bemerkungen:

- Haüfig gibt es zwischen Graphik und Mustererkennung eine **reziproke Problematik**. Z.B. ist bei der Graphik die Konturfüllung ein Problem (Beispiel einer Acht oder des offenen Rings), während bei der Mustererkennung die Konturzeichnung (Beispiel einer Acht oder des offenen Rings) ein besonderes Problem ist.
- Warum macht man Mustererkennung? Sie bringt eine gewaltige Datenreduktion. Z.B. kann ein handgeschriebenes Zeichen befriedigend in einer 20x15-Matrix mit 300 Bit/Zeichen dargestellt werden. Bildbearbeitung kann das um den Faktor 6 reduzieren auf 50 Bit/Zeichen. Die Mustererkennung erkennt schließlich ein Symbol, das mit 8 Bit/Zeichen dargestellt werden kann.

18.2 Klassifizierung von Bilddaten

18.3 Gerätesysteme

18.4 Histogramm-Gleichverteilung

Das Ziel der Histogramm-Gleichverteilung (histogram-equalization) ist Verbesserung der Bildqualität. Zunächst soll die Bezeichnung geklärt werden.

Das zu bearbeitende Bild liegt als Matrix von n Pixeln p vor, die durch die Matrixindizes y, x und einen Grau- bzw Farbwert f charakterisiert werden. Um anzudeuten, daß f von den Pixeln bzw von den Matrixindizes abhängt, schreiben wir auch $f(p)$, $f(y,x)$, f_p oder $f_{y,x}$. Bei den Matrixindizes ist zu beachten, daß x der Spaltenindex und y der Zeilenindex ist. Ferner beginnen wir an der linken oberen Ecke der Matrix mit den Indizes $(0,0)$ und enden in der rechten unteren Ecke mit den Maximalwerten der Indizes (x_{max}, y_{max}). Dies ist anders als in einer üblichen x, y-Koordinatendarstellung, wo der Punkt $(0,0)$ links unten zu vermuten wäre.

Die Anzahl der Pixel in der Matrix ist gegeben durch

$$n = (x_{max} + 1)(y_{max} + 1)$$

Die Grau- bzw Farb*werte* f können L verschiedene Werte annehmen, die wir mit den Grau*stufen* $z = 0 \ldots L-1$ durchnummerieren. Obwohl häufig – insbesondere bei Graubildern – f und z dieselben Werte haben, müssen wir diesen Unterschied zwischen Grau*wert* und Grau*stufe* machen, da die Grau- bzw Farb*werte* oft geräte- bzw graphiksystemabhängig sind. Z.B. kann man mit vielen Graphiksystemen nur 16 verschiedene Farben mit den Werten $0 \ldots 15$ darstellen, wobei die Farbzuordnung durchaus unterschiedlich sein kann. Wenn man von diesen 16 Farben nur die Schwarz-Weiß-Skala verwenden will, dann muß man sich z.B. auf die Farben Schwarz (0), Dunkelgrau (8), Hellgrau (7) und Weiß (15) beschränken. Hier korreliert der Farbwert z.B. nicht mit einem natürlichen Abstand der Farben. Dunkelgrau und Hellgrau sind sogar vertauscht. Für das Histogramm und andere Charakterisierungen von Bildern ist es aber sinnvoll, mit einer möglichst *linearen* Farbskala zu arbeiten, die durch die Grau- bzw Farb*stufen* z gegeben ist. In unserem Beispiel würde man folgende Zuordnung treffen:

$$\begin{array}{rcl} Graustufe & & Grauwert \\ L = 4 & & \\ z = 0 & \equiv & f = 0 \\ z = 1 & \equiv & f = 8 \\ z = 2 & \equiv & f = 7 \\ z = 3 & \equiv & f = 15 \end{array}$$

Die Funktionen zur Übersetzung von z nach f und umgekehrt nennen wir GW bzw GS.

$$\begin{aligned} f &= GW(z) \\ z &= GS(f) \end{aligned}$$

Die Histogramm-Gleichverteilung ist i.a. am sinnvollsten bei Graustufenbildern, auf die wir uns daher im folgenden beziehen werden. Bei Farbbildern kann dieses Verfahren auf jede einzelne RGB-Farbe angewendet werden.

Eine erste Charakterisierung eines Graustufenbildes ist der Mittelwert der Graustufen:

$$\bar{z} = \frac{1}{n} \sum_{alle\ p} z_p$$

Der Mittelwert gibt an, ob das Bild insgesamt hell oder dunkel bzw über- oder unterbelichtet ist.

Eine weitere Kenngröße ist die Standardabweichung.

$$\sigma_z = \sqrt{\frac{\sum (z_p - \bar{z})^2}{n-1}}$$

Sie gibt erste Hinweise, inwieweit der Graustufenbereich ausgenützt wird. Eine große Standardabweichung deutet auf ein kontrastreiches Bild hin.

Das Histogramm $H(z)$ gibt für jede Graustufe die Anzahl der Pixel mit dieser Graustufe an. Häufig wird das Histogramm auf Eins normiert:

$$h(z) = \frac{H(z)}{\sum_{\zeta=0}^{L-1} H(\zeta)} = \frac{H(z)}{n}$$

Der mittlere Informationsgehalt *(Entropie)* eines Bildes ist gegeben durch (Shannon)

$$S = -\sum_{z=0}^{L-1} h(z) \ \log_2 h(z)$$

und kann als die mittlere apriori-Unsicherheit pro Bildpunkt oder die mittlere Anzahl Bit pro Bildpunkt interpretiert werden. Ohne Informationsverlust muß das Bild mit mindestens S Bit pro Bildpunkt kodiert werden. Wenn die nächst höhere ganze Zahl von S kleiner als die Anzahl der zur Kodierung der Graustufen verwendeten Bit ist, dann lohnt es sich z.B. einen Komprimier-Algorithmus anzuwenden.

Beispiel: Ein Bild mit $n = 16$ Bildpunkten sei gleichmäßig mit den Graustufen $0, 1, 2, 3$ besetzt, sodaß

$$H(0) = H(1) = H(2) = H(3) = 4$$

Dann ist die Entropie gegeben durch:

$$S = -\left\{\frac{1}{4}(-2) + \frac{1}{4}(-2) + \frac{1}{4}(-2) + \frac{1}{4}(-2)\right\} = 2$$

Ist das Bild nur mit 1 und 3 besetzt, sodaß

$$H(0) = H(2) = 0 H(1) = H(3) = 8$$

Dann ist die Entropie gegeben durch:

$$S = -\left\{0(-\infty) + \frac{1}{2}(-1) + 0(-\infty) + \frac{1}{2}(-1)\right\} = 1$$

Der Anisotropiekoeffizient

$$\alpha = \frac{-\sum_{z=0}^{k} h(z)\ \log_2 h(z)}{S}$$
$$\text{wobei } k \text{ die kleinste Graustufe ist mit } \quad \sum_{z=0}^{k} h(z)\ \geq 0,5$$

ist ein Maß für die Symmetrie des Histogramms und bewegt sich zwischen 0 und 1, wobei ein kleiner Wert eine Betonung der niedrigen Graustufen, ein großer Wert die Betonung der hohen Graustufen bedeutet.

Grund für eine schlechte Bildqualität ist häufig eine nicht effektiv genutzte Graustufenskala. Ziel der Histogramm-Gleichverteilung ist es, eine Verbreiterung des dynamischen Bereichs durch Gleichverteilung der Graustufen auf die Pixel zu erreichen, d.h. ein möglichst flaches Histogramm zu erzeugen.

Der im folgenden vorgestellte Algorithmus bietet verschiedene Möglichkeiten oder Varianten (im Pseudocode als ***Regeln*** bezeichnet) der Manipulation von Graustufen, insbesondere auch die Anpassung an einen neuen Graustufenbereich.

Algorithmus Histogram-Equalization:

$L-1$ sei die *neue* größte Graustufe (für den Fall einer Anpassung an einen neuen Graustufenbereich).

Wir benötigen folgende Hilfsfelder: $H, Links, Rechts, Neu : array\ [0 \ldots L-1]$ of integer

Schritt 1: Unterteile alten Graustufenbereich in L neue Bereiche, d.h. definiere eine Funktion F, die z_{alt} in das neue z transformiert, $z = F[z_{alt}]$ durch

a) entweder Graustufenbereich **Übernehmen**, d.h. nichts Tun

b) oder Graustufenbereich **Komprimieren** z.B. von $L_{alt} = 256$ nach $L = 4$, indem - gleichmäßig oder ungleichmäßig - Schwellwerte gesetzt werden. Wenn z.B. dunkle Regionen gut aufgelöst werden sollen, wird man die Graustufen eher bei den unteren (alten) Graustufen konzentrieren. Eine gleichmäßige Kompression ist z.B. gegeben durch die Funktion

$$z = F(z_{alt}) = \begin{cases} 0 & : & 0 & \leq z_{alt} \leq & 63 \\ 1 & : & 64 & \leq z_{alt} \leq & 127 \\ 2 & : & 128 & \leq z_{alt} \leq & 191 \\ 3 & : & 192 & \leq z_{alt} \leq & 255 \end{cases}$$

c) oder **Expandieren**, indem die alten Graustufen direkt den ersten neuen Graustufen zugeordnet werden und die übrigen neuen Graustufen unbesetzt bleiben. Z.B. bei der Expansion von $L_{alt} = 4$ nach $L = 256$ ergäbe sich folgende Zuordnung:

$$z = F(z_{alt}) = \begin{cases} 0 & : & z_{alt} = & 0 \\ 1 & : & z_{alt} = & 1 \\ 2 & : & z_{alt} = & 2 \\ 3 & : & z_{alt} = & 3 \\ 4 \ldots & : & & \text{werden noch nicht benötigt} \end{cases}$$

Schritt 1*: Im Fall einer Kompression kann Schritt 1 auch (sogar vorzugsweise) nach Schritt 5 durchgeführt werden.

Schritt 2: Bestimme das Histogramm:

for $z = 0$ **to** $L - 1$ **do** $H(z) = 0$

for all Pixel p **do begin**
  $z = F(GS(f(p)))$
  $f(p) = z$ (Alte Grauwerte werden überschrieben.)
  $H(z) = H(z) + 1$
  next p

Schritt 3: Bilde Mittelwert von $H(z)$:

$$H_{avg} = \frac{1}{L} \sum_{z=0}^{L-1} H(z) = \frac{n}{L}$$

Schritt 4: Belegung der Hilfsfelder $Links, Rechts, Neu$

$R = 0$ und $H_{integral} = 0$

for $z = 0$ **to** $L - 1$ **do begin**
  $Links(z) = R$
  $H_{integral} = H_{integral} + H(z)$
  while $H_{intgral} > H_{avg}$ **do begin**
    $H_{integral} = H_{integral} - H_{avg}$
    $R = R + 1$
    end while
  $Rechts(z) = R$
  case Regel **of**
    0 : Tue nichts (Bild bleibt bis auf Kompression das alte).
    1 : $Neu(z) = (Links(z) + Rechts(z))$ div 2
    2 : $Neu(z) = Rechts(z) - Links(z)$
    3 : $Neu(z)$ bleibt undefiniert
    4 : $Neu(z) =$ durch spezielle Anwendung bestimmt
    end case
  next z

Schritt 5: Belegung der Bildmatrix mit neuen Grauwerten

for all pixels p **do begin**
$z = f(p)$
if $Links(z) = Rechts(z)$ **then** $f(p) = GW(Links(z))$
else case Regel **of**
0 : Tue nichts (Bild bleibt bis auf Kompression das alte).
1 : $f(p) = GW(Neu(z))$ (Bewirkt nur ein Auseinanderrücken der stark besetzten Graustufen, keine Gleichverteilung.)
2 : Wähle zufälligen Wert W_{Zufall} zwischen 0 und $Neu(z)$.
$f(p) = GW(Links(z) + W_{Zufall})$
(Bewirkt Verlust an Kontrast, wenn H bimodal.
Zufälligkeit vermeidet systematische Fehler.)
3 : Bestimme Mittelwert M der Graustufen einer Nachbarschaft von p (4-, 8- oder 12-Nachbarschaft).
if $M > Rechts(z)$ **then** $f(p) = GW(Rechts(z))$
if $M < Links(z)$ **then** $f(p) = GW(Links(z))$
if $Links(z) \leq M \leq Rechts(z)$ **then** $f(p) = GW(M)$
(Bewirkt Kantenverschmierung. Erzwingt gewisse Kohärenz zwischen Pixeln.)
4 : Durch spezielle Anwendung angeregte Phantasie führt zu weiteren Verfahren.
end else case
next p

Schritt 6: Falls Schritt 1 noch nicht durchgeführt wurde, ist er jetzt durchzuführen.

Ende des Algorithmus

18.4.1 Anwendungen

18.5 Schwellwertbestimmung

18.6 Co-Occurence–Matrizen

18.7 Bildfilter

18.7.1 Lineare Filter

18.7.2 Rangordnungsoperationen

18.8 Segmentation

18.9 Objekt–Erkennung

(y, x) seien die Pixelkoordinaten (Spalten- und Zeilennummern) der Bildmatrix.

Bemerkung: Es wird vorausgesetzt, daß die Pixelabstände in x- und y-Richtung *gleich* sind. Wenn das nicht der Fall ist, dann muß eine entsprechende Transformation durchgeführt werden. Ferner wird es sinnvoll sein, nach Auswertung eines im Bild befindlichen Eichobjekts Weltkoordinaten zu verwenden.

$g(y, x)$ sei die aus der Segmentation resultierende Matrix, die zu jeder Pixelposition (y, x) die Nummer w des dazugehörigen Objekts enthält (Resultat des Segmentationsfilters).

Wir wählen nun ein Objekt mit der Nummer w und definieren dafür eine Gewichtsfunktion g_w :

$$g_w(y,x) = \begin{cases} 1 & : \text{ falls } \quad g(y,x) = w \\ 0 & : \text{ sonst} \end{cases}$$

Bemerkung: C-Jargon: $g_w \quad = \quad g(y,x) == w$

Für das Objekt w berechnen wir nun folgende charakteristische Größen, indem wir die technische Mechanik auf die Ebene anwenden:

1. Ausdehnung in x- und y-Richtung:

   $$\Delta x = x_{max}^{(w)} - x_{min}^{(w)}$$

   $$\Delta y = y_{max}^{(w)} - y_{min}^{(w)}$$

   Bemerkung: Könnte beim zweiten Durchgang der Segmentation mitberechnet werden.

2. Fläche:

   $$F = \sum_x \sum_y g_w$$

   Diese und die folgenden Summen stehen für

   $$\sum_x = \sum_{x=x_{min}^{(w)}}^{x=x_{max}^{(w)}} \qquad \text{und} \qquad \sum_y = \sum_{y=y_{min}^{(w)}}^{y=y_{max}^{(w)}}$$

3. Schwerpunkt:

   $$x_s = \frac{1}{F} \sum_x \sum_y x g_w$$

   $$y_s = \frac{1}{F} \sum_x \sum_y y g_w$$

4. Trägheitsmomente:
   Bezüglich einer Achse durch (y_s, x_s) parallel zur x-Achse:

   $$l_x = \sum_x \sum_y (y - y_s)^2 g_w$$

Bezüglich einer Achse durch (x_s, y_s) parallel zur y-Achse:

$$l_y = \sum_x \sum_y (x - x_s)^2 g_w$$

Kopplungsterm:

$$l_{yx} = 2 \sum_x \sum_y (x - x_s)(y - y_s) g_w$$

Bild einfügen.

5. Hauptträgheitsmomente:

$$l_1 = \frac{1}{2}(l_x + l_y) - \frac{1}{2}\sqrt{l_{yx}^2 + (l_y - l_x)^2}$$

$$l_2 = \frac{1}{2}(l_x + l_y) + \frac{1}{2}\sqrt{l_{yx}^2 + (l_y - l_x)^2}$$

6. Orientierung der Hauptachsen:

$$\tan(2\theta) = \frac{l_{yx}}{l_y - l_x}$$

θ ist der Winkel zwischen der Hauptträgheitsachse mit dem kleineren Trägheitsmoment l_1 und der x-Achse. Zur Berechnung des Winkels θ sollte eine zweistellige arctan-Funktion verwendet werden, damit der Quadrant richtig wird:

$$\theta = \frac{1}{2} \arctan(l_{yx}, \quad l_y - l_x)$$

Beim Vergleich mit dem Bild ist zu beachten, daß die linke obere Ecke der Ursprung ist und die y-Achse nach unten weggeht. Die ermittelten Winkel scheinen daher spiegelverkehrt bezüglich der x-Achse.

Häufig werden nur der Sinus bzw Cosinus des Winkels θ benötigt, die schneller zu erhalten sind:

$$\cos^2\theta = \frac{1}{2}\left(1 + \frac{l_y - l_x}{l_2 - l_1}\right)$$

$$\sin^2\theta = 1 - \cos_2\theta$$

$$\cos\theta = \sqrt{\cos_2\theta} \cdot \text{Vorzeichen}(l_y - l_x)$$

$$\sin\theta = \sqrt{\sin_2\theta} \cdot \text{Vorzeichen}(l_{yx})$$

7. Umfang: U
Anzahl der Objektpixel, die an den Hintergrund (oder ein anderes Objekt) grenzen. Wenn man einfach alle Übergänge $0 \leftrightarrow w$ zählt, wird der Umfang zwar überschätzt, was aber für die Praxis meist nicht schädlich ist.

8. Breite und Länge des Objekt:
Dazu müssen zunächst alle Konturpunkte bestimmt werden. Diese müssen dann um den Orientierungswinkel θ gedreht werden:

$$\begin{aligned} x' &= x\cos\theta + y\sin\theta \\ y' &= -x\sin\theta + y\cos\theta \end{aligned}$$

Durch Bestimmung der Extrema der gedrehten Koordinaten in x'-Richtung (x'_{max}, x'_{min}) und y'-Richtung (y'_{max}, y'_{min}) kann man schließlich das ausrechnen, was man gemeinhin unter Breite und Länge eines Objekts versteht:

$$\begin{aligned} \Delta x' &= x'_{max} - x'_{min} \\ \Delta y' &= y'_{max} - y'_{min} \end{aligned}$$

9. Durchmesser:
   Euklidische Distanz:

$$D = \max_{\text{alle Konturpaare } i,j} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   Schachbrett–Distanz:

$$D = \max_{\text{alle Konturpaare } i,j} [\max(|x_i - x_j|, |y_i - y_j|)]$$

10. Ausbreitung 1:

$$A_1 = \frac{l_2 + l_1}{F}$$

    Objekte etwa gleicher Größe (Durchmesser) kann man bezüglich der Frage unterscheiden, ob die Massen eher außen oder innen liegen. (Es macht hier übrigens keinen Sinn anstatt durch F durch F^2 zu teilen, um etwa auf dimensionslose Größen zu kommen. Das liegt daran, daß in Analogie zur Physik die Fläche die Dimension einer Masse hat. Die Trägheitsmomente haben die Dimension Weg * Weg * Masse.)

11. Ausbreitung 2:

$$A_2 = \frac{U^2}{F}$$

    Bilder einfügen.

12. Elongation:

$$E_l = \frac{l_2 - l_1}{l_2 + l_1}$$

    Die Elongation ist eine Zahl zwischen 0 ("Punkt") und 1 ("Strich").

13. Euler–Zahl:

    (a) bezogen auf das gesamte Bild mit allen Objekten:

$$E = \text{Anzahl der Objekte minus Anzahl der Löcher}$$

    (b) bezogen auf *ein* Objekt:

$$E = 1 \quad \text{minus Anzahl der Löcher}$$

    Bilder einfügen.

Mit der Eulerzahl kann man die Objekte bezüglich ihrer Topologie hart unterscheiden. Sie reagiert aber empfindlich auf Fehlstellen in der Pixelmatrix.
Algorithmus zur Bestimmung von E für ein Objekt w:

$$E = \text{Anzahl der Muster} \begin{bmatrix} 0 & 0 \\ 0 & w \end{bmatrix} \quad - \quad \text{Anzahl der Muster} \begin{bmatrix} 0 & w \\ w & ? \end{bmatrix}$$

Bilder und Beispiele einfügen.

Eine andere –eventuell elegantere – Möglichkeit zur Bestimmung der Eulerzahl ist die Anwendung der Segmentation auf das inverse Bild des Objekts und Bestimmung der Anzahl der Objekte n (, die ja dann die Löcher plus Hintergrund sind) und es gilt $E = 2 - n$.

Exkurs Topologie.

14. Spezielle Merkmale:
Problemabhängig wird es in der Praxis meistens notwendig sein, spezielle Objektmerkmale zu definieren und zu bestimmen, um Objekte zu erkennen, z.B. irgendwelche Winkel oder Abstände und Durchmesser von Bohrungen. Methoden der Schrifterkennung (OCR) arbeiten mit Projektionen auf verschiedene Achsen.

18.10 Neuronale Netze und Mustererkennung

Den Einsatz eines primitiven neuronalen Netzes wollen wir an der Erkennung von Zeichen zeigen. Die Klasse `OcrNeuronNetz` hat das im folgenden beschriebene neuronale Netz implementiert. Eine Benutzeroberfläche bietet die Klasse `OCRGUI`. Um auch diese Klassen mitzudokumentieren, werden dort verwendete Variablennamen hier mit aufgeführt.

Wir nehmen an, daß ein Zeichen graphisch mit einer binären Pixelmatrix (`pm`) der Größe

$$n = (y_{max} + 1)(x_{max} + 1)$$

dargestellt wird. ($y_{max} + 1$ entspricht `anzZeilen` und $x_{max} + 1$ entspricht `anzSpalten`.) Als Beispiel betrachten wir eine dreidimensionale Matrix, die den Buchstaben "T" darstellt:

$$T_1 = \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 0 & 1 & 0 \\ \hline 0 & 1 & 0 \\ \hline \end{array}$$

Die Werte von je m (`anzAdrPix`) Pixeln bilden einen Adressbereich (`adr`). Insgesamt haben wir $\frac{n}{m}$ (`anzAdrBer`) Adressbereiche. Üblicherweise sind die Pixel eines Adressbereichs willkürlich verstreut. Im dreidimensionalen Beispiel haben wir bei $m = 3$ auch nur $\frac{9}{3} = 3$ Adressbereiche, die wir a, b, c nennen und die folgendermaßen auf die Pixel der Matrix verteilt seien:

a_0	b_0	b_1
c_0	a_1	b_2
c_2	a_2	c_1

Jeder Adressbereich hat 2^m (`m2`) Speicherplätze (unser Beispiel $2^3 = 8$).

Für jedes Muster (`Zeichen`) (hier ein "T"), das man erkennen will, muß solch ein Satz von Adressbereichen zur Verfügung gestellt werden. (In der Klasse `OcrNeuronNetz` stellt das Feld `ab` die Beziehung zwischen den Pixeln und den Adressbereichen her. Dieses Feld wird zufällig erstellt. D.h. `ab` definiert, welche Pixel der Bildmatrix die Adresse eines Adressbereichs definieren.)

18.10.1 Lernen des Musters

Es werden verschiedene Beispiele des zu lernenden Musters präsentiert. Dabei werden die Inhalte der angesprochenen Adressen auf 1 gesetzt. Nicht angesprochene Adressen bleiben 0.

Also wenn wir T_1 präsentieren, dann werden

$$\begin{aligned} a_0 &= 1 \\ a_1 &= 1 \\ a_2 &= 1 \end{aligned}$$

und bilden die Adresse (111). Wir setzen den Inhalt dieser Adresse auf 1: $a(111) = 1$. Entsprechen machen wir es für b und c. Insgesamt erhalten wir:

$$\begin{aligned} a(111) &= 1 \\ b(110) &= 1 \\ c(000) &= 1 \end{aligned}$$

Als zweites Beispiel für das Muster "T" präsentieren wir ein "schiefes" T:

$$T_2 = \begin{array}{|c|c|c|} \hline 1 & 1 & 0 \\ \hline 0 & 1 & 1 \\ \hline 0 & 1 & 0 \\ \hline \end{array}$$

Das ergibt:

$$\begin{aligned} a(111) &= 1 \\ b(101) &= 1 \\ c(000) &= 1 \end{aligned}$$

Als letztes Beispiel für das Muster "T" präsentieren wir ein unvollständiges T:

$$T_3 = \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array}$$

Das ergibt:

$$\begin{aligned} a(110) &= 1 \\ b(110) &= 1 \\ c(000) &= 1 \end{aligned}$$

Der Inhalt des Adressbereichs für das Muster "T" ist dann:

Adressbereich	000	001	010	011	100	101	110	111
a	0	0	0	0	0	0	1	1
b	0	0	0	0	0	1	1	0
c	1	0	0	0	0	0	0	0

18.10.2 Testen des Netzes

Für eine präsentierte Pixelmatrix berechnen wir die Summe:

$$r = \sum_{x=a,b,c,\ldots} x(\alpha)$$

Dabei ist α die Adresse, die sich aus den Pixelwerten der zum Adressbereich x gehörigen Pixel ergeben.

Test mit T_1 ergibt wie zu erwarten $r = a(111) + b(110) + c(000) = 3$.

Test mit

$$T_4 = \begin{array}{|c|c|c|}\hline 1 & 1 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 1 & 0 \\ \hline \end{array}$$

ergibt $r = a(111) + b(100) + c(000) = 2$.

Test mit

$$T_5 = \begin{array}{|c|c|c|}\hline 1 & 0 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 0 & 1 \\ \hline \end{array}$$

ergibt $r = a(110) + b(011) + c(111) = 1$.

Je kleiner r ist, desto schlechter paßt das Testmuster auf das das zu erkennende Zeichen. Typischerweise definiert man einen Schwellwert, ab dem das Zeichen als erkannt gilt.

In der Praxis werden für Zeichen oft $9x7$-Matrizen mit $m = 4$ verwendet. $m = 5$ oder mehr bringt nicht viel mehr.

Kapitel 19

Optimierung

Prinzipiell lassen sich Optimierungsprobleme in drei Klassen einteilen:

Parameteroptinierung: Finde Werte für einen Satz von Parametern, so dass eine Zielgröße optimiert wird.

Subset-Selection: Finde aus 20 Parametern (Blutdruck, Puls, Blutwerte usw) die 5 Parameter, die am stärksten von einer Größe (Medikament) abhängig sind.

Kombinatorische Probleme: Travelling Salesman

19.1 Dynamische Programmierung

19.1.1 Deterministische Dynamische Programmierung

19.1.2 Probabilistische Dynamische Programmierung

Bei der deterministischen Dynamischen Programmierung wird der Zustand $x(i+1)$ der nächsten Stufe $i+1$ eindeutig durch die Entscheidung $u(i)$ und den Zustand $x(i)$ auf der Stufe §i§ bestimmt. Bei der *probabilistischen* Dynamischen Programmierung wird der Folgezustand $x(i+1)$ nur mit einer gewissen Wahrscheinlichkeit erreicht.

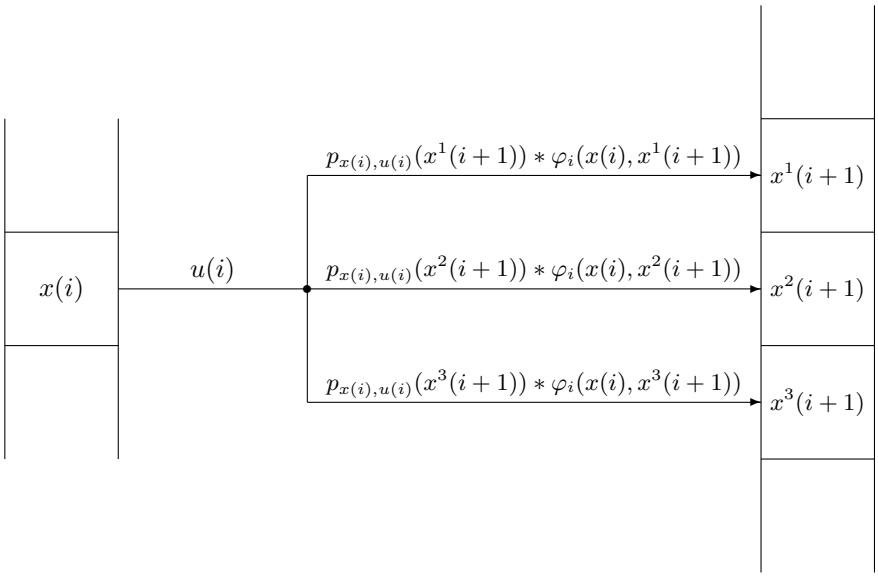
Für jede Entscheidung $u(i)$ im Zustand $x(i)$ gibt es eine Wahrscheinlichkeitsverteilung

$$p_{x(i),u(i)}(x(i+1)) \qquad ,$$

die angibt, mit welcher Wahrscheinlichkeit der Zustand $x(i+1)$ angenommen wird, wenn im Zustand $x(i)$ die Entscheidung $u(i)$ getroffen wurde. Es gilt

$$\sum_{j=1}^{J_{x(i),u(i)}} p_{x(i),u(i)}(x^j(i+1)) = 1 \qquad ,$$

wobei über alle $J_{x(i),u(i)}$ Zustände auf der Stufe $i+1$ summiert wurde, die vom Zustand $x(i)$ mit der Entscheidung $u(i)$ möglicherweise erreicht werden können. Die Summe der Wahrscheinlichkeiten ist 1; d.h. in irgendeinen Zustand der Stufe $i+1$ geht es auf jeden Fall.



Der wahrscheinliche Wert für die Gütefunktion für den Übergang aus dem Zustand $x(i)$ bei der Entscheidung $u(i)$ ist gegeben durch:

$$F_i(x(i), u(i)) = \sum_{j=1}^{J_{x(i),u(i)}} p_{x(i),u(i)}(x^j(i+1)) \cdot h\{\varphi_i(x(i), x^j(i+1)), F^*_{i+1}(x^j(i+1))\}$$

Der optimale Wert für die Gütefunktion ist gegeben durch Bildung des Extremums über alle Entscheidungen $u(i)$:

$$F^*_i(x(i)) = \text{Extremum}_{\text{über alle } u(i)} \; F_i(x(i), u(i))$$

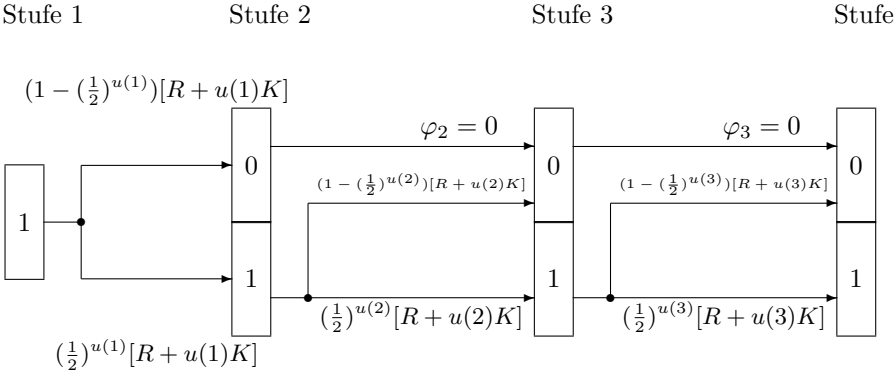
Beispiel:

Ein RAM-Hersteller produziert chargenweise RAM. Nur iedes zweite RAM ist brauchbar, d.h. die Wahrscheinlichkeit, daß ein RAM defekt oder in Ordnung ist, ist jeweils $\frac{1}{2}$. Die Rüstkosten betragen pro Charge $R = DM300$, die Produktionskosten pro RAM sind $K = DM100$. Falls ein RAM nicht rechtzeitig geliefert werden kann, beträgt die Konventionalstrafe $S = DM1600$. Die für einen Auftrag zur Verfügung stehende Zeit reicht gerade für drei Chargen.

Der Auftrag besteht darin, ein *einziges* funktionierendes RAM herzustellen.

Frage: Wieviel RAMs sollten in jeder Charge gefahren werden, damit die Produktionskosten wahrscheinlich minimal werden.

Lösung: Es gibt in diesem System zwei Zustände 0 und 1. Zustand 0 ist erreicht, wenn kein RAM mehr herzustellen ist. Zustand 1 besteht, wenn noch ein RAM zu fertigen ist. Die Stufen bilden die Zustände vor und nach jeder Charge. Auf jeder Stufe ist die Entscheidung $u(i)$ zu treffen, wieviel RAMs in der nächsten Charge zu fahren sind.



19.2 Nichtlineare Programmierung

19.2.1 Optimieralgorithmus EXTREM

Lösung eines Gleichungssystems

Wir haben n Gleichungen, die von m Parametern p_j abhängig sind. Die Gleichungen haben konstante "rechte" Seiten b_i.

$$\sum_{j=1}^{m} a_{ij} p_j = b_i \qquad (i = 1 \ldots n)$$

Dabei kann n kleiner als m sein. Allerdings haben wir für die m Parameter p_j Werte w_j vorgegeben, die möglichst angenommen werden sollten und die sich nur innerhalb gewisser Grenzen verändern dürfen.

Das System macht man zu einem überbestimmten System von $n + m$ Gleichungen, indem m Gleichungen

$$p_j = w_j \qquad (j = 1 \ldots m)$$

hinzugefügt werden.

Solch ein System sollte normiert werden: In den $n + m$ Gleichungen werden zunächst alle Parameter p_j durch $w_j p_j$ ersetzt. Dann werden die ersten n Gleichungen durch b_i dividiert:

$$\sum_{j=1}^{m} \frac{a_{ij} w_j p_j}{b_i} = 1 \qquad (i = 1 \ldots n)$$

Die weiteren m Parameter-Gleichungen werden jeweils durch w_j dividiert:

$$p_j = 1 \qquad (j = 1 \ldots m)$$

Das gesamte System von $n+m$ Gleichungen dürfte vernünftige Lösungen liefern, wenn die Summe der quadratischen Abweichungen als Gütefunktion genommen wird.

Ferner ist es möglich, die Gleichungen mit Gewichten

$$g_i \qquad (i = 1 \ldots n + m)$$

zu multiplizieren, wenn bestimmte Gleichungen möglichst gut erfüllt sein sollen.

19.2.2 Evolutonäre Methoden

Bei den evolutionären oder Simplexmethoden geht man von einem Simplex – i.a. einer k-dimensionalen Verallgemeinerung eines Dreiecks – im k-dimensionalen Raum der Vektoren c aus.

Bei $k = 1$ ist der Simplex eine Strecke, bei $k = 2$ ein Dreieck, bei $k = 3$ ein unregelmäßiger Tetraeder.

An jedem Punkt des Simplex wird die Gütefunktion F bestimmt. In einem Punkt oBdA $c^{(1)}$ ist F am schlechtesten. Dieser Punkt wird nun ausgetauscht gegen einen neuen Punkt c^{neu}, der dem Punkt $c^{(1)}$ gegenüberliegt.

Man nimmt an, daß sich auf diese Weise das System in eine günstige Richtung entwickelt. U.U wird der Simplex vergrößert oder verkleinert, wenn sich F zu wenig ändert bzw wenn der neue Punkt keine Besserung ergibt.

$$c^{neu} = -c^{(1)} + \frac{2}{k} \sum_{i=2}^{k+1} c^{(i)}$$

Diese Methode eignet sich insbesondere, wenn F experimentell bestimmt wird.

Der Start-Simplexe kann rekursiv folgendermaßen konstruiert werden. Bestimme den Mittelpunkt des $(k-1)$-dimensionalen Simplex

$$m = \frac{1}{k} \sum_{i=1}^{k} c^{(i)}$$

und erhöhe von dort aus die k-te Dimension c_k in $c^{(k+1)} = (m, c_k)$ auf z.B.

$$c_k = \sqrt{1 - \sum_{i=2}^{k} \frac{1}{i^2}}$$

bei einem gleichseitigen Simplex.

Die EVOP-Methode von Box beruht auf diesem Prinzip.

19.3 Evolutionäre Optimierung

19.3.1 Verwendung von Simplizes

Der Such- bzw Phasenraum der Parameter habe die Dimension n.

Der Parameter-Raum wird normiert. D.h. für jede Dimension $0, \ldots, n-1$ wird eine Anfangsschrittweite $d_0, \ldots, d_{n-1}$ festgelegt. Jede Dimension wird bezüglich ihrer Anfangsschrittweite normiert.

In diesem normierten Raum wird ein $(n+1)$-dimensionales Simplex mit Kantenlänge 1 erstellt (Strecke der Länge 1 ($n = 1$), gleichseitiges Dreieck mit Seitenlänge 1 ($n = 2$), gleichseitiger Tetraeder ($n = 3$) usw).

Wenn die Punkte $N_0, \ldots, N_n$ im normierten Raum bestimmt sind, dann wird jede Dimension mit ihrer Anfangsschrittweite multipliziert. Ferner wird jeder Punkt um einen vorgegebenen Startpunkt translatiert. Dadurch man erhält den Simplex in Weltkoordinaten.

Berechnung des Simplexes

Wir beginnen mit dem Ursprung und legen den auf Punkt N_n:

$$N_n = (c_0 = 0, c_1 = 0, \ldots, c_{n-1} = 0)$$

Mit $h_0 = 1$ ergibt sich der nächste Punkt zu:

$$N_0 = (h_0, 0, \ldots, 0)$$

$$N_0 = (1, 0, \ldots, 0)$$

Die Höhe im gleichseitigen Dreieck ist gegeben durch $h_1 = \sqrt{1 - (1/2h_0)^2} = \sqrt{1 - 1/4}$. Damit ergibt sich:

$$N_1 = (1/2, \sqrt{3/4}, 0, \ldots)$$

Für den Tetraeder erhalten wir die Höhe $h_2 = \sqrt{1 - (2/3h_1)^2} = \sqrt{1 - 1/3} = \sqrt{2/3}$:

$$N_2 = (1/2, 1/3\sqrt{3/4}, \sqrt{2/3}, 0, \ldots)$$

Entsprechend müßte sich ergeben: $h_3 = \sqrt{1 - (3/4h_2)^2} = \sqrt{1 - 3/8} = \sqrt{5/8}$:

$$N_3 = (1/2, 1/3\sqrt{3/4}, 1/4\sqrt{2/3}, \sqrt{5/8}, 0, \ldots)$$

Oder allgemeiner:

$$h_0 = 1$$

$$N_0 = (h_0, 0, \ldots, 0)$$

$$h_1 = \sqrt{1 - (1/2h_0)^2}$$

$$N_1 = (1/2h_0, h1, 0, \ldots)$$

$$h_2 = \sqrt{1 - (2/3h_1)^2}$$

$$N_2 = (1/2h_0, 1/3h_1, h_2, 0, \ldots)$$

$$h_3 = \sqrt{1 - (3/4h_2)^2}$$

$$N_3 = (1/2h_0, 1/3h_1, 1/4h_2, h_3, 0, \ldots)$$

$$h_i = \sqrt{1 - (\frac{i}{i+1}h_{i-1})^2}$$

$$N_i = (1/2h_0, \ldots, \frac{1}{i+1}h_{i-1}, h_i, 0, \ldots)$$

Die Transformation in Weltkoordinaten-Punkte P_i mit dem Anfangspunkt

$$C = (c_0, c_1, \ldots, c_{n-1})$$

und den Anfangsschrittweiten

$$D = (d_0, d_1, \ldots, d_{n-1})$$

wird nach folgendem Schema durchgeführt:

$$P_i = C + N_i * E * D$$

(Dabei ist E die Einheitsmatrix. D.h. N_i und D werden einfach komponentenweise miteinander multipliziert.)

Für jeden Punkt P_i wird nun die Zielfunktion bestimmt. Der schlechteste Punkt sei oEdA P_n und wird nun am Mittelwert M aller anderen Punkte gespiegelt und ergibt den neuen Punkt P'_n.

$$M = \frac{1}{n}\sum_{i=0}^{n-1} P_i$$

$$P'_n = 2M - P_n$$

Schrittweitensteuerung

"Oszillieren" bedeutet, dass immer zwischen denselben zwei Punkten hin- und hergespiegelt wird.

"Stagnieren" bedeutet, dass der Unterschied zwischen dem besten und dem schlechtesten Wert kleiner als ein vorgegebener Wert ist. Man sollte noch überlegen, ob man zwischen starkem und schwachem Stagnieren unterscheidet, indem man zwei Grenzwerte vorgibt.

- Wenn das System (schwach oder stark) stagniert und nicht oszilliert, dann verdopple die Schrittweite.
- Wenn das System (schwach oder) nicht stagniert, aber oszilliert, dann verkleinere die Schrittweite um die Hälfte.
- Wenn das System (stark) stagniert und oszilliert, dann ist das Optimum erreicht. Gehe dann immer auf den besten Punkt.
- Wenn das System nicht stagniert und nicht oszilliert, dann kann es weiterlaufen.

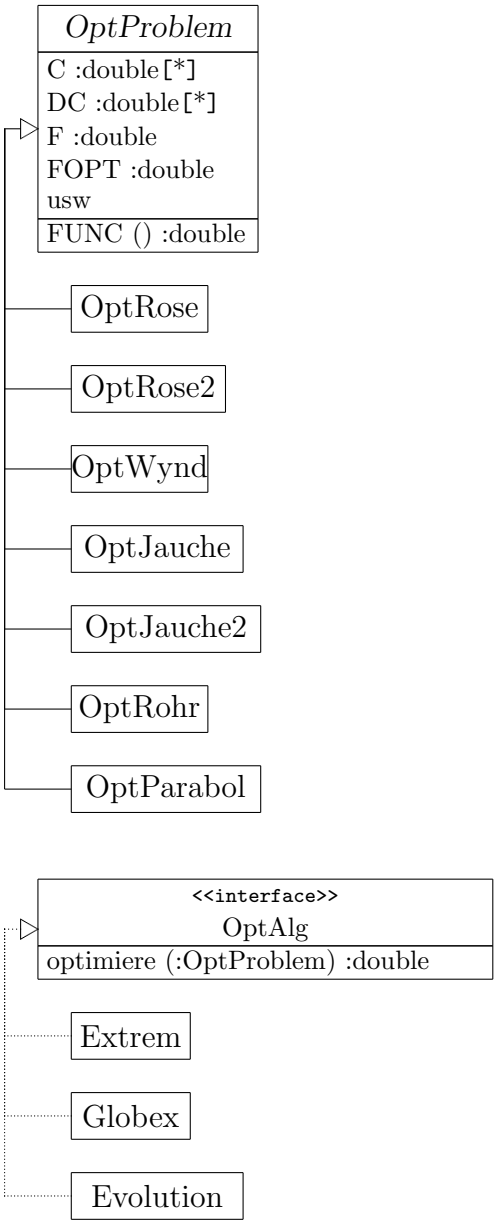
19.4 Software: Design und Implementation

19.4.1 Design

Es gibt schon zwei ältere Klassen `OptProblem1` (Vektor `C` der Steuergrößen beginnt bei "1".) und `OptProblem0` (Vektor `C` der Steuergrößen beginnt bei "0".), die wir als Referenzimplementation für den Algorithmus EXTREM zunächst mal weiter behalten. Verschiedene Optimierprobleme müssen von `OptProblem` erben und die Methode `FUNC` überschreiben. Das ist eigentlich solange in Ordnung, als es nur einen oder wenige Optimieralgorithmen gibt, nämlich EXTREM und GLOBEX. Wenn wir aber andere Algorithmen implementieren wollen, die eventuell andere Datenstrukturen benötigen, wird das sehr unübersichtlich.

Daher wird jetzt ein Strategy-Pattern verwendet, wobei `Context` zu `OptProblem` und `contextInterface ()` zu – im wesentlichen – `FUNC ()` wird. `Strategy` wird zu `OptAlg` und `algorithm ()` zu `optimiere (:OptProblem)`.

Im Moment ist dieses ganze Schema noch unter `lib/kj/opt/neu` zu finden.


OptProblem
C :double[*]
DC :double[*]
F :double
FOPT :double
usw
FUNC () :double
OptRose
OptRose2
OptWynd
OptJauche
OptJauche2
OptRohr
OptParabol
<<interface>>
OptAlg
optimiere (:OptProblem) :double
Extrem
Globex
Evolution

Kapitel 20

Identifikation

Beispiel Ausgleichsgerade

Zu fünf verschiedenen Zeiten t wurden die Ausgangswerte y_1 eines realen Prozesses bestimmt.

t	1	2	3	4	5
y_1	2	3	7	9	9

Als Prozeßmodell wird eine Gerade gewählt:

$y^M = g(p) = p_1 + p_2 t$

Als Startwerte für die Modellparameter p wird der Vektor

$$p^0 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

gewählt. Dann gilt:

$$g^0 = g(p^0) = 1 + 2t = \begin{pmatrix} 3 \\ 5 \\ 7 \\ 9 \\ 11 \end{pmatrix} \qquad y - g^0 = \begin{pmatrix} -1 \\ -2 \\ 0 \\ 0 \\ -2 \end{pmatrix}$$

$$\frac{\partial g}{\partial p} = \begin{pmatrix} 1 & t \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{pmatrix}$$

$$\left(\frac{\partial g^T}{\partial p}\frac{\partial g}{\partial p}\right) = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{pmatrix} = \begin{pmatrix} 5 & 15 \\ 15 & 55 \end{pmatrix}$$

$$\left(\frac{\partial g^T}{\partial p}\frac{\partial g}{\partial p}\right)^{-1} = \begin{pmatrix} 1.1 & -0.3 \\ -0.3 & 0.1 \end{pmatrix}$$

$$p = \begin{pmatrix} 1.1 & -0.3 \\ -0.3 & 0.1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix} \begin{pmatrix} -1 \\ -2 \\ 0 \\ 0 \\ -2 \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

$$p = \begin{pmatrix} 1.1 & -0.3 \\ -0.3 & 0.1 \end{pmatrix} \begin{pmatrix} -5 \\ -15 \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

Iteration (eigentlich nicht nötig, da Problem linear, aber um zu zeigen, daß alles in Ordnung ist):
Ergebnisse der ersten Iteration sind nun Startwerte:

$$p^0 = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

gewählt. Dann gilt:

$$g^0 = g(p^0) = 0 + 2t = \begin{pmatrix} 2 \\ 4 \\ 6 \\ 8 \\ 10 \end{pmatrix} \qquad y - g^0 = \begin{pmatrix} 2 \\ 3 \\ 7 \\ 9 \\ 9 \end{pmatrix} - \begin{pmatrix} 2 \\ 4 \\ 6 \\ 8 \\ 10 \end{pmatrix} = \begin{pmatrix} -0 \\ -1 \\ 1 \\ 1 \\ -1 \end{pmatrix}$$

$$p = \begin{pmatrix} 1.1 & -0.3 \\ -0.3 & 0.1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix} \begin{pmatrix} 0 \\ -1 \\ 1 \\ 1 \\ -1 \end{pmatrix} + \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

$$p = \begin{pmatrix} 1.1 & -0.3 \\ -0.3 & 0.1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

Nun beinahe dasselbe Beispiel, nur etwas pädagogischer:

Zu fünf verschiedenen Zeiten t wurden die Ausgangswerte y_1 eines realen Prozesses bestimmt.

t	1	2	3	4	5
y_1	3	4	7	9	10

Als Prozeßmodell wird eine Gerade gewählt:

$y^M = g(p) = p_1 + p_2 t$

Als Startwerte für die Modellparameter p wird der Vektor

$$p^0 = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

gewählt. Dann gilt:

$$g^0 = g(p^0) = 2 + 2t = \begin{pmatrix} 4 \\ 6 \\ 8 \\ 10 \\ 12 \end{pmatrix} \qquad y - g^0 = \begin{pmatrix} -1 \\ -2 \\ -1 \\ -1 \\ -2 \end{pmatrix}$$

$$\frac{\partial g}{\partial p} = \begin{pmatrix} 1 & t \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{pmatrix}$$

$$\left(\frac{\partial g^T}{\partial p}\frac{\partial g}{\partial p}\right) = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{pmatrix} = \begin{pmatrix} 5 & 15 \\ 15 & 55 \end{pmatrix}$$

$$\left(\frac{\partial g^T}{\partial p}\frac{\partial g}{\partial p}\right)^{-1} = \begin{pmatrix} 1.1 & -0.3 \\ -0.3 & 0.1 \end{pmatrix}$$

$$p = \begin{pmatrix} 1.1 & -0.3 \\ -0.3 & 0.1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix} \begin{pmatrix} -1 \\ -2 \\ -1 \\ -1 \\ -2 \end{pmatrix} + \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

$$p = \begin{pmatrix} 1.1 & -0.3 \\ -0.3 & 0.1 \end{pmatrix} \begin{pmatrix} -7 \\ -22 \end{pmatrix} + \begin{pmatrix} 2 \\ 2 \end{pmatrix} = \begin{pmatrix} -1.1 \\ -0.1 \end{pmatrix} + \begin{pmatrix} 2 \\ 2 \end{pmatrix} = \begin{pmatrix} 0.9 \\ 1.9 \end{pmatrix}$$

Kapitel 21

Grundlagen der Robotik

21.1 Roboter RV-M1

21.2 Roboter RV-1A

21.2.1 Ethernet-Schnittstelle

Einstellung der Adresse

Protokoll

21.3 Übungen

Kapitel 22

Prozess-Automatisierung

Literaturverzeichnis

[1] M. Ben-Ari, "Grundlagen der Parallel-Programmierung", Hanser 1985

[2] Joshua Bloch, "Effective Java Programming Language Guide", Addison-Wesley

[3] G. Bolch und M.-M. Seidel, "Prozeßautomatisierung", Teubner 1993

[4] Clay Breshears, "The Art of Concurrency", O'Reilly

[5] T. DeMarco, "Structured Analysis and System Specification", Yourdon Press 1979

[6] Bruce Powel Douglass, "Real-Time Design Patterns" Addison-Wesley

[7] Bruce Powel Douglass, "Real Time UML" Addison-Wesley

[8] G. Färber, "Prozeßrechentechnik", Springer 1994

[9] O. Föllinger, "Regelungstechnik", Hüthig Buch Verlag 1990

[10] Naran Gehani und William D. Roome, "The Concurrent C Programming Language", Silicon Press 1989

[11] Brian Goetz, "Java Concurrency in Practice", Addison-Wesley Longman

[12] Hassan Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison-Wesley

[13] David Harel, "Statecharts: a visual formalism for complex systems", Science of Computer Programming **8** (1987), 231 – 274

[14] C. A. R. Hoare, "Communicating Sequential Processes", Prentice Hall 1985

[15] John E. Hopcroft, Rajeev Motwani und J. D. Ullman, "Introduction to automata theory, languages, and computation", Addison-Wesley 2001

[16] Doug Lea, "Concurrent Programming in Java: Design Principles and Patterns", Addison-Wesley

[17] E. Kienzle und J. Friedrich, "Programmierung von Echtzeit-Systemen", Hanser

[18] H. Kopetz, "Real-Time Systems", Springer

[19] Sanjaya Kumar, James H. Aylor, Barry W. Johnson und Wm. A. Wulf "The Codesign of Embedded Systems – A Unified Hardware/Software Representation", Kluwer Academic Publishers 1996

[20] Edward. L. Lamie, "Real-Time Embedded Multithreading Using ThreadX and MIPS", Newnes

[21] P.D. Lawrence und K. Mauch, "Real-Time Microcomputer System Design", McGraw-Hill 1987

[22] Peter Liggesmeyer und Dieter Rombach (Hrsg.), "Software Engineering eingebetteter Systeme", Elsevier Spektrum Akademischer Verlag

[23] C. L. Liu und James W. Layland, "Scheduling Algorithms for Multiprogramming in Hard-Real-Time Environment", Journal of the ACM, **20**, 46 (1973)

[24] Wan-Chen Lu, Jen-Wei Hsieh und Wei-Kuan Shih, "A Precise Schedulability Test Algorithm for Scheduling Periodic Tasks in Real-Time Systems", Proceedings of the 2006 ACM symposium on Applied computing, 1451 – 1455 (2006)

[25] Christoph Marscholik, "Anforderungen an einen Echtzeitkern", Wind River Systems 1995

[26] Bertrand Meyer, "Object-oriented Software Construction", Prentice Hall 1988

[27] Scott Oaks und Henry Wong, "Java Threads", OŽReilly

[28] Lothar Piepmeyer, Vorlesungsskriptum "Java Threads", `lothar.piepmeyer@hs-furtwangen.de`

[29] Rainer Oechsle, "Parallele und verteilte Anwendungen in Java", Hanser 2007

[30] Dominique Perrin, "Finite Automata" in "Formal Models and Semantics" ed. Jan van Leeuwen, Elsevier 1992

[31] M. Polke, "Prozeßleittechnik", Oldenbourg 1992

[32] Shangping Ren und Gul A. Agha, "A Modular Approach for Programming Embedded Systems", Lecture Notes in Computer Science, 1996

[33] B. Rosenstengel und U. Winand, "Petri-Netze: Eine anwendungsorientierte Einführung", Vieweg