

Eingebettete Systeme

A. Löffler, C. Hempfling

28. Januar 2015

1 Kapitel

Chapter 1.1

Definition Eingebettetes System:

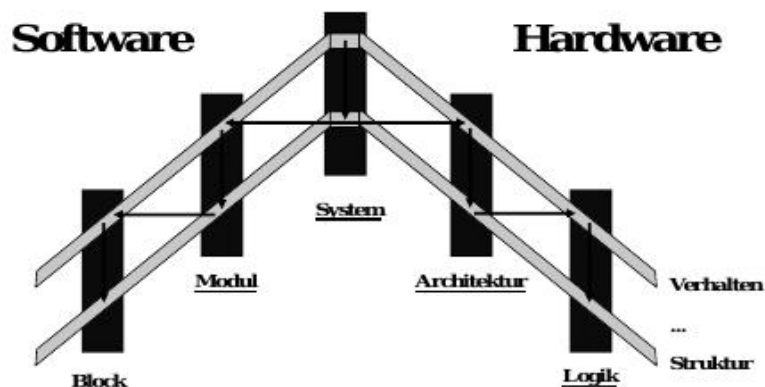
Ein eingebettetes System ist ein informationsverarbeitendes System bestehend aus Hardware und Software, das in einem technischen Gesamtsystem eingebettet ist, dessen primärer Zweck nicht das Verarbeiten und Aufbereiten von Information ist.

Durch die zunehmende Komplexität hat sich der Entwurf eingebetteter Systeme von analogen Schaltungen und Assemblerprogrammen auf einfachen Mikrocontrollern weg zum Problem hin entwickelt, komplexe Hardware/Software Systeme mittels leistungsfähiger Werkzeuge zu entwickeln und zu optimieren.

	Eingebettetes System	konventioneller Rechner
Funktion	<i>feste Funktion:</i> zum Zeitpunkt des Entwurfs ist die Funktion, die das System übernehmen soll, bekannt	<i>universell und offen:</i> zum Zeitpunkt des Entwurfs sind die Funktionen, die das System übernehmen soll, weitgehend offen
Verhalten	<i>reaktives Verhalten:</i> das System muss auf Aktionen der Umgebung unter festen zeitlichen Bedingungen reagieren	<i>interaktives Verhalten:</i> das System sollte mit der Umgebung unter sehr losen zeitlichen Bedingungen interagieren
Design	<i>Co-Design:</i> Hardware und Software werden aufeinander abgestimmt, entwickelt und optimiert	<i>unabhängiges Design:</i> Hardware und Software werden getrennt voneinander entwickelt und unabhängig voneinander optimiert

- aber auch konventionelle Computer haben eingebettete Systeme (z.B. Grafik-, Netzwerkkarte)
- die beiden wichtigsten Merkmale für eingebettete Systeme sind:
 1. reaktives Verhalten unter Echtzeitbedingungen
 2. gleichzeitige Entwicklung von Hardware und Software
- die Entscheidung, welche Funktionen in Hardware und welche in Software auf einem oder mehreren Mikrocontrollern realisiert werden, fällt oft sogar während der Entwicklung und Optimierung des Entwurfs

Abstraktionsebenen:



Abstraktionsebenen – Software-Bereich:

- *Modul*: die Modulebene gehört zum Softwarebereich. Die Modelle der Modulebene beschreiben komplexe Funktionen und ihre Interaktionen.
- *Block*: die Blockebene gehört ebenfalls zum Softwarebereich. Die entsprechenden Modelle beschreiben Programme bis hin zu Instruktionen, die auf der zugrundeliegenden Rechnerarchitektur elementare Operationen ausführen.

Abstraktionsebenen – Hardware-Bereich:

- **Architektur**: die Architekturebene gehört zum Hardwarebereich. Die Modelle dieser Ebene beschreiben kommunizierende Blöcke, die alle nebenläufig komplexe Operationen ausführen.
- **Logik**: die Logikebene gehört ebenfalls zum Hardwarebereich. Die Modelle dieser Ebene beschreiben verbundene Gatter und Register, die Boolesche Funktionen berechnen.

Modell:

Unter einem Modell versteht man die formale Beschreibung eines Systems (oder Teilsystems). Wie genau ein Modell ein System beschreibt, hängt von der entsprechenden Abstraktionsebene ab. Man unterscheidet i.d.R. zwischen

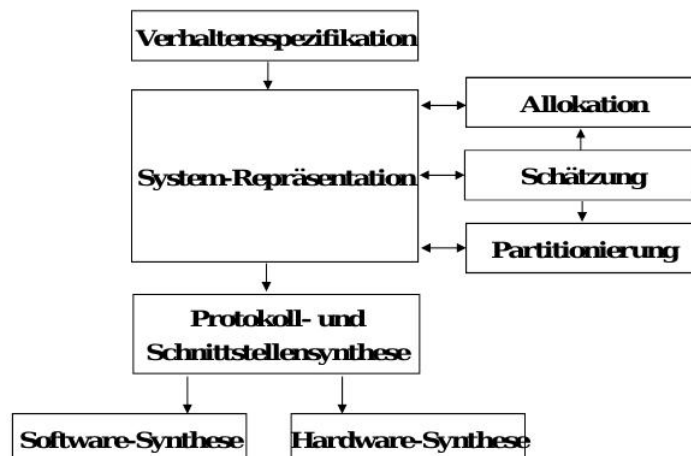
- **zustandsorientierten** (z.B. auf endlichen Automaten basierende)
- **aktivitätsorientierten** (z.B. auf Datenflussgraphen basierende)
- **strukturorientierten** (z.B. auf Blockschaltbildern basierende)
- **datenorientierten** (z.B. als Kollektion von Datenobjekten mit Relationen)

Modellen.

Aufgabe der (automatischen) Systemsynthese:

- Festlegung der Komponententypen, die in der Implementierung verwendet werden (Allokation), z.B. Mikroprozessor, ASIC, Speicherbausteine...
- Festlegung der Anzahl der jeweiligen Komponenten, Auswahl und Dimensionierung der Verbindungsstruktur (Allokation)
- Zuordnung der Variablen zu Speicherbausteinen, Operationen zu Funktionsbausteinen und Kommunikationen zu Bussen (Binding). Hierbei sind Realisierung in Hardware und in Software gegeneinander abzuwägen (Hardware/Software-Partitionierung).
- Erstellung eines Ablaufplans: Wann wird welche Aufgabe durch seine Ressource ausgeführt?
- Schätzung von Systemeigenschaften, um eine vernünftige Exploration des Entwurfsraumes zu ermöglichen. Gerade auf den obersten Entwurfsebenen werden grundlegende Entwurfsentscheidungen getroffen, die die Leistungsfähigkeit und die Kosten des ganzen Systems bestimmen.

Entwurfsablauf auf Systemebene



Architektursynthese

Aufgabe der Architektursynthese: eine strukturelle Sicht aus einer Verhaltensbeschreibung (z.B. VHDL) zu generieren. \Rightarrow Grad der Parallelität

wesentlichste Aufgaben:

- Identifikation von Hardware-Elementen, die die spezifizierten Operationen ausführen können (Allokation)
- Ablaufplanung zur Bestimmung der Zeitpunkte, an denen die Operationen ausgeführt werden
- Binding, d.h. Zuordnung von
 - Variablen zu Speichern
 - Operationen zu funktionalen Einheiten
 - Kommunikationskanäle zu Bussen

Modulsynthese, Logiksynthese, Blocksynthese: werden in anderen Vorlesungen genauer behandelt

Chapter 1.2

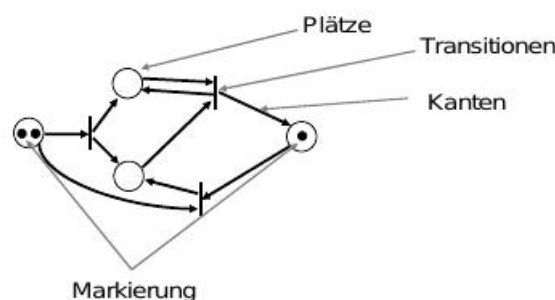
Formale Methoden zur Spezifikation von Systemen sind notwendig, um Realisierungen mit korrektem Verhalten synthetisieren zu können. Spezifizieren mit formalen Methoden ist nicht trivial... aber der einzige gehbare Weg!

Petri-Netze

Definition: Ein Petri-Netz ist ein 6-Tupel $G = (P, T, F, K, W, M_0)$ mit

- $P = \{p_1, \dots, p_m\}$... die Menge der Plätze oder Stellen.
- $T = \{t_1, \dots, t_n\}$... die Menge der Transitionen.
- $P \cap T = \emptyset$. Beide Mengen zusammen bilden die Menge der Knoten.
- $F \subseteq (P \times T) \cup (T \times P)$... die Menge der Kanten. Auch Flussrelation genannt.
- $K : P \rightarrow \mathbb{N} \cup \{\infty\}$, die jedem Platz eine Kapazität zuordnet
- $W : F \rightarrow \mathbb{N}$, die jeder Kante ein Kantengewicht zuordnet.
- $M_0 : P \rightarrow \mathbb{N}_0$ mit $\forall p \in P : M_0(p) \leq K(p)$, die Anfangsmarkierung

Petri-Netze sind also gerichtete bipartite Graphen, denen man eine Interpretation im Sinne erreichbarer Markierungen gibt. Gängige Notation:



Schreibweise:

1. $\forall t \in T : \bullet t = \{p; (p, t) \in F\}$: Vorbereich von t
2. $\forall p \in P : \bullet p = \{t; (t, p) \in F\}$: Vorbereich von p
3. $\forall t \in T : t \bullet = \{p; (p, t) \in F\}$: Nachbereich von t
4. $\forall p \in P : p \bullet = \{t; (t, p) \in F\}$: Nachbereich von p

Schaltbereitschaft von Transitionen:

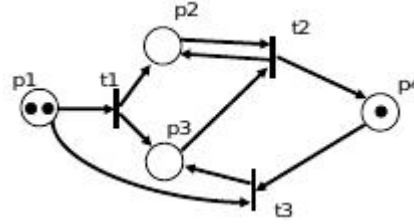


Abbildung 1.1: Transitionen t1 und t3 sind schaltbereit

Eine Transition $t \in T$ eines Petri-Netzes schaltet von der Markierung M auf M' :

$$M[t > M']$$

Aktivierbare und lebendige Transitionen

Definition: Ist $M : P \rightarrow N_0$ eine Markierung eines Petri-Netzes, so heißt eine Markierung M' des Petri-Netzes **erreichbar**, wenn

- $\exists t \in T : M[t > M'$
- $\exists t \in T : M''[t > M'$ und M'' ist von M aus erreichbar

$[M >$ bezeichne die Menge der von M aus erreichbaren Markierungen.

Eine Transition t eines Petri-Netzes heißt bzgl. einer Markierung M **tot**, wenn sie unter keiner von M aus erreichbaren Markierung schaltbereit ist, d.h. $\forall M' \in [M > : \neg(M'[t >)$. Ansonsten heißt sie **aktivierbar**.

Eine Transition t eines Petri-Netzes heißt bzgl. einer Markierung M **lebendig**, wenn sie bzgl. jeder von M aus erreichbaren Markierung aktivierbar ist.

Ein Petri-Netz heißt **deadlockfrei** oder **schwach lebendig**, wenn es zu jeder von M_0 aus erreichbaren Markierung M' eine Transition $t \in T$ gibt, die schaltbereit ist.

Ein Petri-Netz heißt **lebendig** oder **stark lebendig**, wenn jede seiner Transitionen bzgl. M_0 lebendig ist.

Sei G ein Petri-Netz und $B : P \rightarrow N_0 \cup \{\infty\}$ eine Abbildung, die jeder Stelle eine kritische Markenzahl zuordnet ($B(p) \leq K(p)$). Das Petri-Netz G heißt **B-sicher** oder **B-beschränkt**, wenn $\forall M \in [M_0 > \forall p \in P : M(p) \leq B(p)$.

Das Petri-Netz heißt **sicher** oder **beschränkt**, wenn es eine natürliche Zahl b gibt, für die G b -beschränkt ist.

Ein Petri-Netz ist genau dann beschränkt, wenn seine Erreichbarkeitsmenge $[M_0 >$ endlich ist.

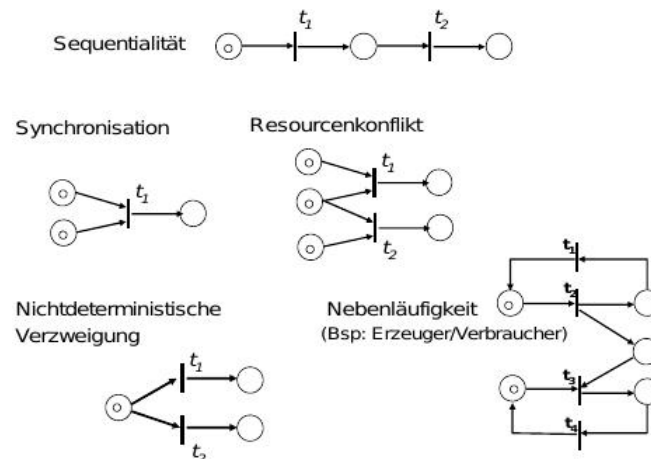


Abbildung 1.2: Ausdruckskraft von Petri-Netzen

Überdeckung von Markierungen:

Annahme: Die Kapazität eines Platzes ist unendlich.

Seien M und M' Markierungen. Dann ist $M+M'$ die Markierung, die man durch $M+M'(p) := M(p)+M'(p)$ erhält. Entsprechend sei “-” auf Markierungen definiert.

Eine Markierung M **überdeckt** M' ($M \geq M'$) genau dann, wenn für alle Plätze p gilt: $M(p) \geq M'(p)$

ω -Erweiterungen

Eine ω -erweiterte Markierung ist gegeben durch $M : P \rightarrow N_0(\omega)$

Überdeckungsgraph

Man kann nun zu jedem Petri-Netz wie folgt einen Überdeckungsgraphen von M_0 (ist ein Knoten des Graphen) aus konstruieren:

- Knoten: ω -erweiterte Markierungen $M : P \rightarrow N_0 \cup \{\omega\}$
- Kanten: Ist M ein Knoten, dann ist für jede feuerbereite Transition $M[t > M'$
 - M' Nachfolgeknoten, falls kein Vorgänger M'' von M existiert mit $M' \geq M''$
 - $\Omega(M'', M')$ Nachfolgeknoten, falls M'' Vorgänger von M mit $M' \geq M''$
- Der Überdeckungsgraph ist stets endlich.
- Ein Petri-Netz ist beschränkt dann und nur dann, wenn bei der Konstruktion des Überdeckungsgraphen keine ω -erweiterten Knoten entstehen
- Petri-Netze können den gleichen Überdeckungsgraphen haben und doch ungleiches Verhalten
- Bei beschränkten Petri-Netzen ist das Verhalten genau dann gleich, wenn die Überdeckungsgraphen isomorph sind

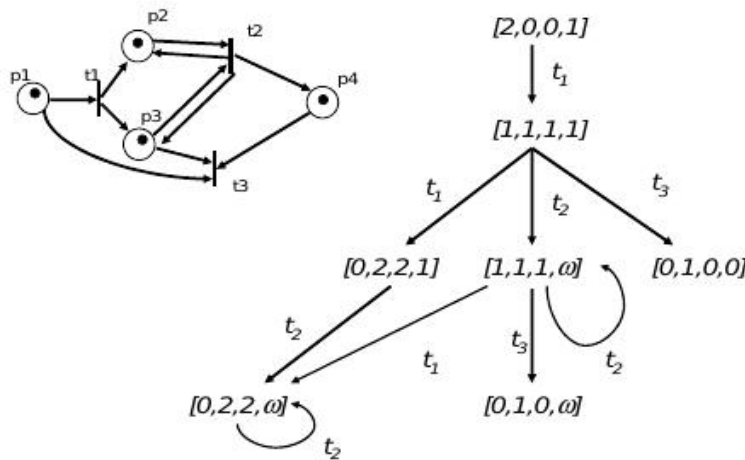


Abbildung 1.3: Beispiel für Überdeckungsgraph

Zustandsorientierte Modelle

Definition:

Ein **endlicher Automat** ist gegeben durch ein 6-Tupel $M := (I, O, S, R, \delta, \lambda)$.

Dabei sind S, I, O endliche Mengen.

S heißt **Zustandsmenge**, $R \subseteq S$ die Menge der **Startzustände**

I heißt **Eingabemenge**

O heißt **Ausgabemenge**

$\delta : S \times I \rightarrow S$ heißt **Übergangsfunktion**

$\lambda : S \times I \rightarrow O$ heißt **Ausgabefunktion**

sind λ, δ “echte” Relationen, nennt man den Automaten auch **nichtdeterministisch**. Sind δ, λ partiell, d.h. nicht für alle Paare (s, x) definiert, nennt man den Automaten auch unvollständig.

Es muss dann aber gelten, dass $\lambda(s, x)$ definiert $\Rightarrow \delta(s, x)$ definiert.

Endliche Automaten können als Spezialfall interpretierter Petri-Netze aufgefasst werden. Ein nichtdeterministischer endlicher Automat ist ein Petri-Netz $G = (P, T, F, K, W, M_0)$ mit

- $\forall t \in T : |\bullet t| = |t \bullet| = 1$
- $\exists p_0 \in P : M_0(p_0) = 1$ und $\forall p \neq p_0 : M_0(p) = 0$
- $K = W = 1$
- jeder Transition ist ein Prädikat (Eingabeereignis) als zusätzliche Schaltbedingung und ein Ausgabeereignis als Aktion zugewiesen

Statecharts

Die Eingabesymbole endlicher Automaten sind in der Regel Sensorsignale oder Zustandssignale kooperierender Automaten. \Rightarrow Notiere die Übergänge als Kanten mit Prädikaten (Ereignisse), die auf genau den Eingaben erfüllt sind, unter denen der Übergang stattfindet.

Die Ausgabesymbole sind in der Regel Steuersignale für Aktoren. \Rightarrow Notiere neben die Übergangsprädikate Aktionen.

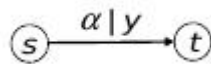


Abbildung 1.4: Unter allen Eingaben x mit $\alpha(x)$ gehe mit Ausgabe y nach t

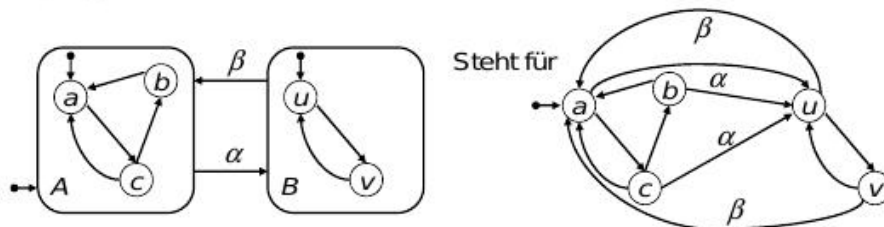
Statecharts – Gruppierung:

Gruppieren Zustände durch Einkreisen und erlaube Übergänge aus Zustandsgruppen. Ein Übergang aus einer Zustandsgruppe steht für Übergänge von jedem Zustand der Gruppe aus zu dem gegebenen Zielzustand.

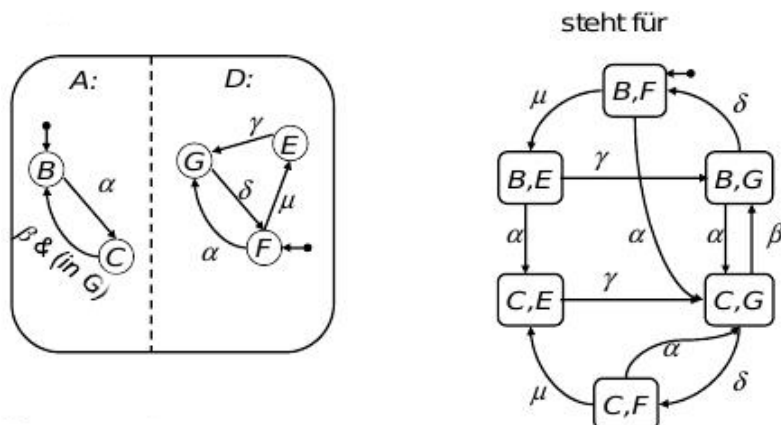


Statecharts – Hierarchie:

Gruppieren Zustände durch Einkreisen, benenne sie und markiere einen Startzustand.

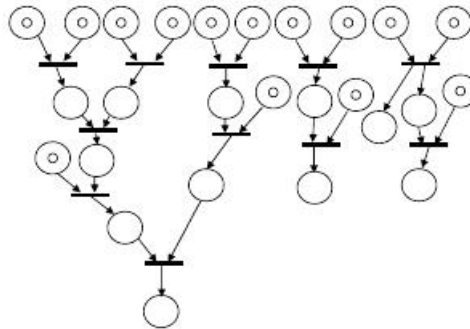


Statecharts – Nebenläufigkeit:



Datenflussgraphen - DFG

DFGs sind gerichtete Graphen, die die durchzuführenden Berechnungen allein über die Verfügbarkeit der dazu notwendigen Daten definieren. Knoten entsprechen Operationen (Aktoren), Kanten entsprechen Datenabhängigkeiten. DFGs sind ein Sonderfall von Petri-Netzen: ersetze die Operationsknoten durch Transitionen und die Kanten durch Plätze:



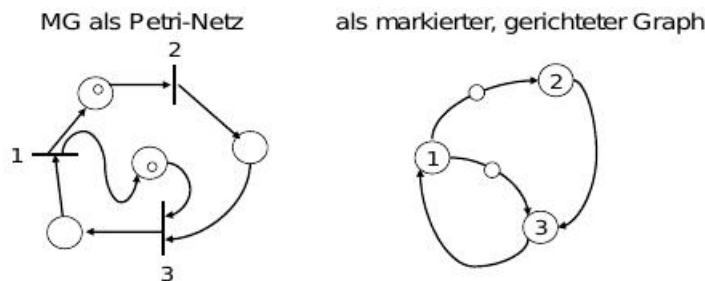
Jeder Platz hat genau eine Nachfolge- und eine Vorgängertransition, denn jede Datenabhängigkeit hat genau einen Produzenten und genau einen Konsumenten \Rightarrow markierte Graphen

Markierte Graphen

Definition: Ein markierter Graph (MG) ist ein Petri-Netz $G = (P, T, F, K, W, M_0)$ mit folgenden Eigenschaften:

- $\forall p \in P : |\bullet p| = |p\bullet| = 1$
- $W = 1$
- $\forall p \in P : K(p) = \infty$

Marken jeder Stelle p werden in der Reihenfolge entnommen, in der sie dort entstehen (FIFO) \Rightarrow keine Konflikte auf den Transitionen. Da in MG die Plätze stets genau einen Vorgänger und einen Nachfolger haben, kann man die Plätze auch als Knoten im Netz weglassen und direkt die Kanten zwischen Transitionen betrachten und markieren.



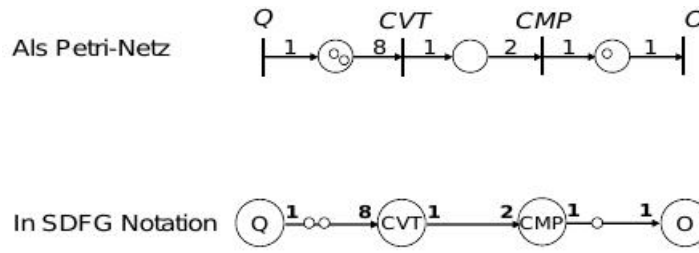
Ein Knoten kann feuern, wenn alle einlaufenden Kanten markiert sind. Er nimmt von jeder eingehenden Kante eine Marke und legt auf jede ausgehende Kante eine Marke.

Definition: Ein markierter Graph ist ein gerichteter Graph $G = (V, E, M_0)$, wobei $M_0 : E \rightarrow N_0$ die Anfangsmarkierungen der Kanten ist.

Synchrone Datenflussgraphen

Definition: Ein synchroner Datenflussgraph (SDFG) ist ein Petri-Netz $G = (P, T, F, K, W, M_0)$ mit folgenden Eigenschaften:

- $\forall p \in P : |\bullet p| = |p\bullet| = 1$
- $\forall p \in P : K(p) = \infty$



Marken jeder Stelle p werden in der Reihenfolge entnommen, in der sie dort entstehen (FIFO). Im Unterschied zu markierten Graphen erhalten die Kanten echte Gewichte.

Definition: Ein SDFG G ist ein 5-Tupel $G = (V, E, cons, prod, d)$ mit

- V ist die Menge der Knoten – Knoten stehen für Operationen
- $E \subseteq V \times V$ ist die Menge der Kanten – aus Kanten können mehrere Daten liegen. Es gilt FIFO.
- $cons : E \rightarrow N$ gibt die Anzahl der beim Feuern des Zielknotens konsumierten Marken an – sie stehen am Pfeilende
- $prod : E \rightarrow N$ gibt die Anzahl der beim Feuern des Quellknotens produzierten Marken an – sie stehen am Pfeilanfang
- $d : E \rightarrow N_0$ gibt die Anfangsmarkierung der Kanten an, d.h. die Anzahl der anfangs auf jeder Kante verfügbaren Daten.

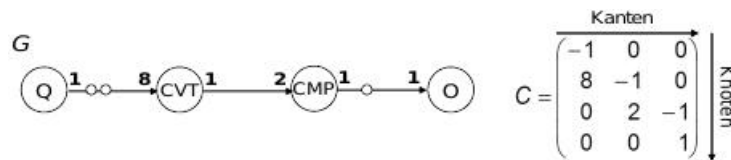
Topologiematrix des SDFG

Definition: Sei $G = (V, E, cons, prod, d)$ ein SDFG. Die zu G zugehörige Topologiematrix ist ein Matrix C aus $Z^{|V| \times |E|}$, deren Zeilen den Knoten und deren Spalten den Kanten von G entsprechen. Verläuft die Kante e_j vom Knoten v_i zum Knoten v_k , so gilt

$$C[i, j] := -prod(v_i, v_k)$$

$$C[k, j] := cons(v_i, v_k)$$

$$C[l, k] = 0 \text{ für } l \notin \{i, k\}$$



Ist d_1 die momentane Kantenmarkierung und feuert Knoten v_i jeweils y_i -mal, so erhält man als resultierende Knotenmarkierung

$$d_2 = d_1 - C^T y$$

Definition: Ein zusammenhängender SDFG G mit Topologiematrix C aus $Z^{|V| \times |E|}$ heißt konsistent, wenn die Startmarkierung in endlich vielen Schritten wieder angenommen werden kann, d.h. es ein periodisches Verhalten von G gibt.

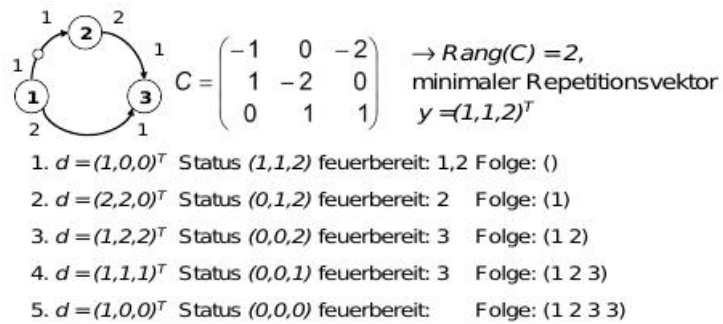
Satz: Ist G zusammenhängend und konsistent, dann gilt $Rang(C) = |V| - 1$

Minimaler Repititionsvektor

Definition: Sei G ein zusammenhängender, konsistenter SDFG. Dann heißt der kleinste positive Vektor $y \neq 0$ im Nullraum von C^T , d.h.

$$y \in \{x \in N^{|V|}; C^T x = 0\} \sum_i y_i \text{ minimal}$$

minimaler Repititionsvektor (von G).



Verklemmung in SDFG

Problem: mit der Existenz eines minimalen Repetitionsvektors y ist noch nicht sichergestellt, dass es zu diesem Vektor auch eine konkrete Folge schaltbereiter Knoten gibt, so dass jedes v_i genau y_i -mal feuert. Das Netz könnte verklemmen.

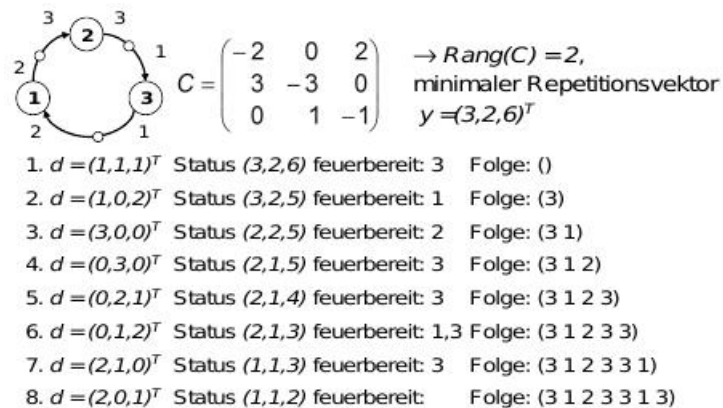


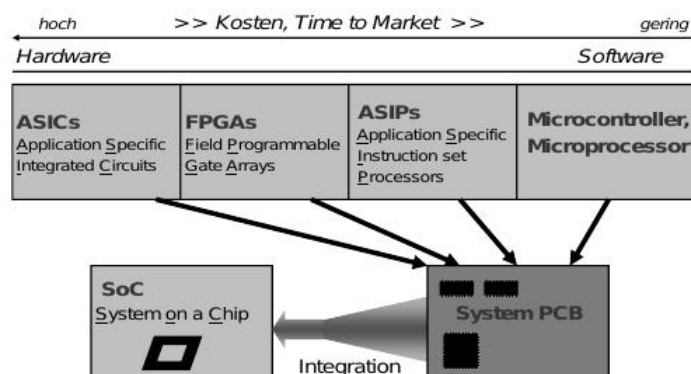
Abbildung 1.5: Beispiel für Verklemmung in einem SDFG

Sequenzgraphen

Definition: Ein Sequenzgraph ist ein azyklischer gerichteter Graph $G = (V, E)$ mit zwei ausgezeichneten Knoten, einem Start- und einem Endknoten. Zusätzlich besitzt G die folgenden Eigenschaften:

- die vom Start- und Endknoten verschiedenen Knoten lassen sich in die folgenden Klassen aufteilen:
 - Operationsknoten
 - Hierarchieknoten
- Hierarchieknoten sind Zeiger auf andere Sequenzgraphen. Man unterscheidet hierbei zwischen dem Modulaufwurf (CALL), der Verzweigung (BR) und der Iteration (LOOP)

Chapter 1.3



Allgemeines Synthesemodell

Gegeben ist:

- eine Menge von Aufgaben (Operationen, Tasks, ...), die untereinander Datenabhängigkeiten besitzen können
- eine Menge von Ressourcen (ALUs, CPUs, Operatoren, ...), auf denen jeweils gewisse Aufgaben ablaufen können

Gesucht ist:

- eine Festlegung der Anzahl von Ressourcen \Leftarrow **Allokation**
- eine Festlegung des zeitlichen Ablaufs der Bearbeitung der Aufgaben unter Berücksichtigung der Datenabhängigkeiten \Leftarrow **Ablaufplan (Schedule)**
- eine Zuordnung der Aufgaben zu den Ressourcen, so dass zu jeder Zeit jede Ressource höchstens eine Aufgabe bearbeitet \Leftarrow **Bindung (Binding)**

Je nach Implementierungstechnik werden die Aufgaben unterschiedlich ausgeführt:

1. *Allokation*: dieses Problem wird in der Regel beim Entwurf gelöst und hat entscheidenden Einfluss auf die Kosten
2. *Ablaufplan*: Ablaufpläne können beim Entwurf gelöst werden (statische Planung) oder zur Laufzeit (dynamische Planung), wobei dynamische Lösungen meist bei Softwareimplementierungen, statische eher bei Hardwareimplementierungen genutzt werden
3. *Bindung*: Bindungen können ebenso als statische wie dynamische Aufgabe auftreten und von trivial (eine CPU) bis sehr schwierig (mehrere Ressourcen gleichen Typs mit unterschiedlicher Leistung) rangieren

Definitionen

- **Sequenzgraph (Problemgraph, Taskgraph)** $G_S = (V, E)$. Sei ein gerichteter azyklischer Graph mit genau einem Start- und genau einem Endknoten. Im folgenden bezeichnen wir mit $V_S \subseteq V$ die Knotenmenge V ohne den Start- und Endknoten und mit $E_S \subseteq E$ die Kanten, die weder den Start- noch den Endknoten als Randknoten besitzen.
- bipartiter **Ressourcengraph** $G_R = (V_R, E_R)$ mit $V_R = V_S \cup V_T$ und $E_R \subseteq V_S \times V_T$. V_T ist die Menge der Ressourcentypen (z.B. Prozessor, ALU, Addierer, ...)
- **Kostenfunktion** $c : V_T \rightarrow N_0$, die die Kosten einer Instanz eines Ressourcentyps angibt
- **Ausführungszeiten** $w : E_R \rightarrow N_0$, die jeder Kante $(v_s, v_t) \in E_R$ die Ausführungszeit der Aufgabe $v_s \in V_S$ auf einer Instanz des Ressourcentyps $v_t \in V_T$ angibt
- ein **Syntheseproblem** P ist gegeben durch ein Tupel $P = (G_S, G_R)$

Definition – Allokation: Gegeben sei ein Syntheseproblem. Eine Allokation ist eine Funktion $\alpha : V_T \rightarrow N_0$, die jedem Ressourcentyp $v_t \in V_T$ eine Anzahl $\alpha(v_t)$ verfügbarer Instanz zuordnet.

Definition – Ablaufplan: Gegeben sei ein Syntheseproblem. Ein Ablaufplan (Schedule) eines Sequenzgraphen $G_S = (V, E)$ ist eine Funktion $\tau : V_S \rightarrow N_0$, die jedem Knoten $v_s \in V_S$ die Startzeit $\tau(v_s)$ zuordnet und für jede Kante $(v_s, v_t) \in E_S$ die Bedingung

$$\tau(v_t) - \tau(v_s) \geq w(v_s)$$

erfüllt. Hierbei gibt $w(v_s)$ die Laufzeit der Aufgabe v_s auf ihrer Ressource an.

Definition – Latenz: Die Latenz eines Ablaufplans τ eines Sequenzgraphen $G_S = (V, E)$ ist definiert als

$$L(\tau) = \text{späteste Beendungszeit} - \text{früheste Startzeit} = \max\{\tau(v_i) + w(v_i); v_i \in V_S\} - \min\{\tau(v_j); v_j \in V_S\}$$

Definition – Bindung: Eine Bindung eines Sequenzgraphen $G_S = (V, E)$ bzgl. eines Ressourcengraphen $G_R = (V_R, E_R)$ und einer Allokation $\alpha : V_T \rightarrow N_0$ ist ein Paar von Funktionen $\beta : V_S \rightarrow V_T$ und $\gamma : V_S \rightarrow N$ mit

- $\forall v_s \in V_S : (v_s, \beta(v_s)) \in E_R$
- $\forall v_s \in V_S : \gamma(v_s) \leq \alpha(\beta(v_s))$

d.h. die Bindung ordnet jeder Aufgabe eine verfügbare Instanz einer Ressource zu, die diese Aufgabe ausführen kann. Liegt ein Ablaufplan vor, so muss ferner gelten, dass zu jedem Zeitpunkt t jeder Ressource nur eine Aufgabe zugeordnet ist.

2 Kapitel

3 Ablaufplanung

Unterscheidungskriterien

- statische vs. dynamische – statische Ablaufpläne treten auf, wenn zur Übersetzungs-/Synthesezeit ein Ablaufplan erstellt werden muss, vor allem bei hardwarenahen Implementationstechniken. Dynamische Ablaufpläne treten zur Laufzeit des ES auf, meist bei Softwareimplementierungen.
- präemptiv vs nicht-präemptiv – bei präemptiven Plänen geht man davon aus, dass die Laufzeit eines Jobs viel länger ist als die Dauer, ihn zu pausieren, meist Software. Bei kurzen oder nicht-unterbrechbaren Jobs verwendet man nicht-präemptive Pläne.
- Abhängigkeitsbedingungen – bei Datenabhängigkeiten unterliegt der Plan zeitlichen Einschränkungen, gegeben durch DAG. Kommen sehr häufig vor.
- Ressourcenbeschränkungen – Hardware (PUs, Busse, Speicher) als Nebenbedingung, Probleme werden schwerer.
- Zeitbeschränkungen –
 - absolute: Deadlines & Release-Termine
 - relative: zeitliche Bedingungen zwischen den einzelnen Jobs (Start- & Endzeitpunkte)
- periodisch vs. aperiodisch – periodische Probleme kommen bei Iterationen vor, periodische Einplanung. Findet Verwendung bei Schleifenoptimierungen oder DSP.

3.1 Statische Ablaufplanung

3.1.1 Statische Ablaufplanung ohne Ressourcenbeschränkungen

ist einfach und kann von gängigen Graph-Algorithmen effizient gelöst werden.

der **As Soon as Possible-Algorithmus (ASAP)** (Komplexität $O(|V| + |E|)$)

- berechne topologische Sortierung
- verplane jeden Knoten in Reihenfolge der topologischen Sortierung vom frühest möglichen Zeitpunkt (wenn die entsprechende Ressource wieder frei ist)

Dieser Algorithmus liefert einen Ablaufplan minimaler Latenz \rightarrow untere Schranke für Pläne mit Zeit- & Ressourcenbeschränkungen. (Beweisbar über Induktion über die Startzeitpunkte.)

der **As Late as Possible-Algorithmus (ALAP)** (Komplexität $O(|V| + |E|)$)

- berechne umgekehrte topologische Sortierung
- verplane jeden Knoten in Reihenfolge der umgekehrten Sortierung vom spätest möglichen Zeitpunkt (beachte kritische Pfade)

Dieser Algorithmus findet Anwendung, wenn eine Latenzschranke vorgegeben ist und die Jobs möglichst spät gestartet werden sollen.

3.1.2 Ablaufplanung mit Zeitbeschränkungen

absolute Zeitbeschränkungen \Leftrightarrow relative Zeitbeschränkungen zum Startknoten \Rightarrow wir betrachten nur relative Zeitbeschränkungen.

Der **Constraintgraph** ist ein gewichteter, gerichteter Graph mit Gewichtsfunktion, die sich aus dem Sequenzgraphen ergibt. Er enthält alle Knoten & Kanten des Sequenzgraphen sowie für jede Zeitbeschränkung eine weitere Kante, deren Gewicht der Zeitbeschränkung entspricht.

Die Existenz eines gültigen Ablaufplans kann mit Zeit- aber ohne Ressourcenbeschränkungen in polynomieller Zeit entschieden und ein latenzminimaler, gültiger Ablaufplan τ – falls existent – gefunden werden. (Komplexität $O(|V_S| \cdot |E_C|)$ mittels Bellman-Ford)

3.1.3 Ablaufplanung mit Ressourcenbeschränkungen

Optimierungsprobleme

- Latenzminimierung mit Ressourcenbeschränkungen – bei gegebener Allokation sind Bindung und Ablaufplan mit minimaler Latenz gesucht
- Kostenminimierung unter Latenzbeschränkung – bei gegebener Latenzschranke sind Allokation, Bindung und Ablaufplan mit minimalen Kosten gesucht
- Zulässiges Ablaufproblem – gesucht ist eine Implementierung bei gegebener Latenzschranke und Allokation
- Gewichtete Minimierung von Latenz und Kosten – gesucht ist eine Implementierung, die bei gegebener Latenzschranke und Allokation eine Kostenfunktion minimiert

List Scheduling ist eine Weiterentwicklung von ASAP. Es werden Listen geführt für alle abgearbeiteten Jobs, alle offenen Jobs und alle Ressourcen mit ihren darauf laufenden Jobs. Wähle eine maximale Teilmenge aller Jobs je Ressourcentyp mit absteigender Priorität. Plane diese Jobs sinnvoll ein. Die Qualität des Scheduling hängt von der Wahl der Knoten ab. Gebräuchliche Prioritätsfunktionen:

- Anzahl der Nachfolgaufträge je Job
- Länge des längsten Pfades vom Job zum Ende
- Mobilität der Aufgaben (Differenz zwischen ASAP und ALAP Startzeiten)

List Scheduling ist nur eine Heuristik!

Force Directed Scheduling benutzt ein Kräftemodell, um die Jobs an geeignete Ausführungszeitpunkte zu schieben/ziehen. Dynamische Änderung des Modells nach Planung von Aufgaben, Berücksichtigung direkter Nachbarschaften im Graphen, Updates des Kräftemodells nach jedem Schritt. Man beachtet dabei folgende Kräfte:

- Selbstkraft – Ausführung am besten zu Zeiten geringer Auslastung
- Nachbarschaftskräfte – Einfluss durch Vorgänger- & Nachfolger-Jobs. Planung nach sinkender mittlerer Auslastung der Ressourcen

Benutze diese Kräfte als Priorisierung für das List Scheduling

- wähle die höchsten Kräfte für minimale Latenz unter Ressourcenbeschränkungen
- benutze Kräfte, um Job für Job Zeitpunkten zuzuordnen für minimalen Ressourcenbedarf unter Latenzschranke (heuristische Minimierung der Auslastung)

3.1.4 Ablaufpläne mittels ILP

ILPs sind ganzzahlige lineare Programme. Sollte ein Ablaufplan mit Bindung existieren, so wird dieser auch vom ILP gefunden. (Lösen mittels Branch&Bound, Relaxierung findet untere Schranken)

3.2 Dynamische Ablaufplanung

Es liegt eine iterative Definition der Berechnungen vor, also immer die gleichen Jobs abhängig vom Zeitindex n .

Ein **iterativer Problemgraph** ist ein Netzwerk, dessen Kantengewichte die Indexverschiebungen darstellen – dieser kann insbesondere Zyklen beinhalten. Ein periodischer Ablaufplan ordnet jedem Job eine Menge von Startzeitpunkten zu, die die Indexverschiebungen berücksichtigen. Das Iterationsintervall ist die Menge aller Zeitschritte zwischen Start des ersten Jobs und Ende des letzten Jobs einer Iteration. Die Art der **Parallelisierung** hat starken Einfluss auf die Datenrate, mit der die Jobs verarbeitet werden können.

- sequenzielle Abarbeitung – erforderlich, wenn alle Daten einer Iteration mit einem Takt synchronisiert werden müssen
- nichtüberlappende Abarbeitung – erforderlich, wenn sich alle Jobs nach einer Iteration synchronisieren müssen. Äquivalent zur sequenziellen Abarbeitung bei geeignetem Retiming (Indexverschiebung)

- überlappende Abarbeitung – ermöglicht viel kürzere Perioden bei geeigneter Planung
- vollstatische Bindung – jeder Job läuft in jeder Iteration stets auf der gleichen Ressourceninstanz
- zyklstatische Bindung – jeder Job läuft nach jeder k -ten Iteration wieder auf der gleichen Ressourceninstanz

3.2.1 Sequenzielle periodische Ablaufplanung

Satz: Es gibt einen gültigen, sequentiellen, periodischen Ablaufplan ohne Ressourcenbeschränkung mit Iterationsintervalllänge L für einen Problemgraphen $G = (V, E, s)$, genau dann, wenn

$$\forall u, v \in V : W(u, v) > L \Rightarrow S(u, v) \geq 1$$

Dabei ist $S(u, v)$ die Menge der Längen der kürzesten Pfade von u nach v .

3.2.2 Retiming

Wir versuchen durch Indexverschiebungen die Länge des Iterationsintervalls zu minimieren \rightarrow Neufestlegung der Iterationsindizes. Ein Retiming $r(u)$ ändert im Problemgraphen nur die Größen $S(u, v)$ – und auch nur in Abhängigkeit von r – jedoch nicht die Größe $W(u, v)$. Die Lösung eines gültigen Retimings ist also eigentlich ein single-source-longest-path Problem in einem besonderen Graphen $G_{r,L}$. Gibt es in diesem Graphen einen positiven Zyklus, so gibt es kein Retiming der Länge L .

Retiming-Algorithmus

- Berechne zu einem Problemgraphen die Größen $S(\cdot)$ und $W(\cdot)$ durch Lösung eines all pair shortest path Problems $O(|V||E|\log|V|)$
- Sortiere die Menge $\{W(u, v) | u, v \in V\}$ $O(|V|2\log|V|)$
- Bestimme durch Binärsuche das kleinste $L = W(u, v)$ aus obiger Menge, für das single source longest path Problem in $G_{r,L}$ eine Lösung r hat. Wähle r als Retiming. $O(|V|3\log|V|)$

3.2.3 Überlappende periodische Ablaufplanung

Satz: gegeben sei ein iterativer Problemgraph $G(V, E, s)$. Für jede Kante $e \in E$ sei $w(q(e))$ die Berechnungszeit ihres Quellknotens. Dann ist die Periode nach unten beschränkt durch

$$P_{min} = \max \left\{ \left\lceil \frac{\sum_{e \in Z} w(q(e))}{\sum_{e \in Z} s(e)} \right\rceil \mid Z \text{ ist Zyklus in } G \right\}$$

Die Periodenschranke kann in $O(|V| \cdot |E| \cdot \log \sum_{v \in V} w(v))$ berechnet werden.

Ein Problemgraph hat **perfekte Rate** wenn es für jeden einfachen Zyklus Z die Summe $\sum_{e \in Z} s(e) = 1$ ist. Hat ein Graph perfekte Rate, so gibt es zu einem Ablaufplan minimaler Periode auch eine statische Bindung.

Da eine zyklstatische Bindung eine Periodizität K hat, kann man sie im Grunde gleichsetzen mit einer statischen Bindung bei einem K -fach abgerollten oder entfalteten Problemgraphen (zu jedem Knoten gibt es nun K -viele Instanzen usw.).

3.2.4 Periodische Ablaufplanung unter Ressourcenbeschränkungen

Diese können wieder mit ILPs gelöst werden.

Satz – Processor Bound: Gegeben sei ein iterativer Problemgraph $G = (V, E, s)$ und ein einziger Ressourcotyp (Prozessor) auf dem jeder Knoten v_i in Zeit $w(v_i)$ ausgeführt werden kann. Ferner sei eine Periode P für den Ablaufplan gegeben. Dann gilt für die minimale Zahl von benötigten Ressourcen α_{min} :

$$\alpha_{min} = \left\lceil \frac{\sum_i w(v_i)}{P} \right\rceil$$

Das zugehörige ILP in Worten:

- eine binäre Variable $x_{i,t}$ drückt den Ablaufplan aus. Sie ist genau dann 1, wenn die Aufgabe v_i zum Zeitpunkt $t + nP$ gestartet wird, also $\tau(v_i) = t + nP$.
- die Aufgabe v_i darf nur genau ein mal pro Periode geplant werden.

- Es gilt $\sum_{t=l_i}^{h_i} t \cdot x_{i,t} = \tau(v_i)$ und somit sagt die Bedingung aus, dass Aufgabe v_j frühestens $w_i - s_{i,j}P$ Zeitschritte später als Aufgabe v_i geplant werden darf, wenn es eine Datenabhängigkeit mit Indexverschiebung $s_{i,j}$ zwischen Aufgabe v_i und Aufgabe v_j gibt.
- Ressourcenbeschränkung: Man überlege sich, dass die Ressource $\beta(v_i)$ zum Zeitpunkt $t + nP$ durch v_i nur belegt sein kann, wenn

$$\tau(v_i) \leq t \leq \tau(v_i) + w_i - 1 \text{ oder } \tau(v_i) - P \leq t \leq \tau(v_i) + w_i - 1 - P$$

Gründe für dynamische Ablaufplanung

- unbekannte oder datenabhängig stark variierende Länge der Bearbeitungszeiten
- Zahl/Art der Jobs vorher nicht bekannt
- Datenabhängigkeiten teilweise unbekannt oder variabel
- $t_r(v)$ – Releasezeit, der frühest möglichen Startzeitpunkt eines Jobs.
- $t_d(v)$ – Deadline, der spätest mögliche Endzeitpunkt eines Jobs.

Mit **Dispatchlatenz** L_D bezeichnet man die Zeitspanne, die benötigt wird, um nach Stoppen eines Jobs den nächsten zu starten.

Die **Ressourcenauslastung** U für eine CPU und einen Ablaufplan mit Latenz L ist gegeben durch

$$U = \frac{\sum_{v \in V} w(v)}{L} \cdot 100$$

Die **Flusszeit** einer Aufgabe v ist gegeben durch: $t_F(v) = \tau_e(v) - t_r(v)$

Die **Wartezeit** einer Aufgabe v ist gegeben durch: $t_W(v) = t_F(v) - w(v)$, also die Zeit vom frühest möglichen Startzeitpunkt bis zu ihrem effektiven Ende.

Die **Lateness** eines Jobs v ist gegeben durch: $t_L(v) = \tau_e(v) - t_d(v)$. Sie kann sowohl positiv als auch negativ sein.

Die **Tardiness** eines Jobs v ist gegeben durch: $t_T(v) = \max\{t_L(v), 0\}$. Diese verwendet man, wenn man nur an echten Zeitverletzungen interessiert ist.

Man kennt **time sharing Systeme** vor allem bei Mehrbenutzersystemen (Betriebssysteme) – man möchte die Ressourcenauslastung maximieren und die Fluss- bzw. Wartezeiten minimieren.

Bei **real time operating systems** herrschen harte Zeitbedingungen, daher sind die primären Ziele:

- minimieren der Lateness,
- minimieren der mittleren Tardiness,
- minimieren der Anzahl von Aufgaben mit Tardiness > 0

3.2.5 nichtpräemptive dynamische Systeme

also von Systemen, die ohne das Unterbrechen von Aufgaben auskommen.

Bei **first come first serve (FCFS)** handelt es sich um eine Heuristik, die die Aufgaben in der Reihenfolge einplant, in der sie auftreten. Dabei hängt die mittlere Wartezeit stark von der tatsächlichen Reihenfolge ab.

Bei **shortest job first (SJF)** handelt es sich um eine Heuristik, die die Reihenfolge nach der gewichteten Bearbeitungszeit bildet. $w(v)$ Bearbeitungszeit, $c(v)$ Gewicht \Rightarrow Reihenfolge aus $\frac{w(v)}{c(v)}$. Ohne Datenabhängigkeiten liefert dieses Verfahren eine minimale mittlere Wartezeit (Satz von Smith). Dabei bleibt die Priorität der Aufgaben statisch, ändert sich also nicht während der Berechnung. Für Releasezeiten $t_r(v) \neq 0$ oder mit Datenabhängigkeiten allerdings dieses Problem NP-hart.

Bei **earliest deadline first (EDF)** handelt es sich um eine Heuristik, die die Jobs anhand ihrer Deadlines einplant. Sie minimiert die maximale Lateness, wenn alle Jobs Releasezeit $t_r(v) = 0$ haben (Jackson Regel). Diese hat ebenfalls statische Prioritäten. Auch bei Releasezeiten $t_r(v) \neq 0$ oder mit Datenabhängigkeiten wird dieses Problem NP-hart.

3.2.6 Präemptive dynamische Planung

Alle Jobs können nun in ihrer Bearbeitung unterbrochen werden.

Bei **weighted round robin (WRR)** handelt es sich um eine Heuristik, die die Jobs in eine FIFO Warteschlange einreihet. Jeder Job dieser Schlange wird $c(v) \cdot Q$ Zeiteinheiten lang bearbeitet und dann pausiert und es kommt der nächste Job der Schlange dran. Dabei kommt es nicht zum Aushungern von Jobs, alle sind mal dran. WRR ist einfach zu implementieren, liefert jedoch eine hohe mittlere Wartezeit und kann nicht für Echtzeitbedingungen garantieren.

Auch **EDF** lässt sich gut auf präemptive Systeme anwenden. Wenn ein neuer Job bereit wird überprüft man dessen Priorität und unterbricht dann evtl. den aktuell laufenden Job, falls dieser eine niedrigere Priorität aufweist. Falls ein Ablaufplan ohne Datenabhängigkeiten existiert, der alle Deadlines einhält, so existiert auch ein Plan nach EDF, der alle Deadlines einhält (stärker als Jackson, da keine Releasezeiten gefordert).

Die Vorhersagbarkeit des Verhaltens ist ein Problem der präemptiven Planungssysteme, weswegen sie kaum in Echtzeitsystemen mit sicherheitskritischen Deadlines eingesetzt werden. Es ist nicht einmal möglich, stets best- und worst-case Verhalten solcher System anzugeben.

3.2.7 Präemptive periodische Planung

Für dynamische Planungsverfahren, bei denen alle Aufgaben unterbrochen werden können.

Periodische Tasks Es wird ein System periodisch auftretender Jobs mit relativen Deadlines $t_d^*(v)$ und relativen Releasezeiten $t_r^*(v)$ definiert, die mit folgenden Releasezeiten und Deadlines assoziiert werden:

- $t_r(v, n) = t_r^*(v) + n \cdot P(v)$
- $t_d(v, n) = t_d^*(v) + t_r^*(v) + n \cdot P(v)$

Die **ratenmonotone Planung (RM-scheduling)** ist ein solches Verfahren. Es ist prioritätsbasiert und ordnet Tasks ihrer Rate $\frac{1}{P(v)}$.

RM Schedule bei einer CPU:



4 Kapitel

5 Kapitel