Eingebettete Systeme

A. Löffler, C. Hempfling

31. Januar 2015

1 Kapitel

Chapter 1.1

Definition Eingebettetes System:

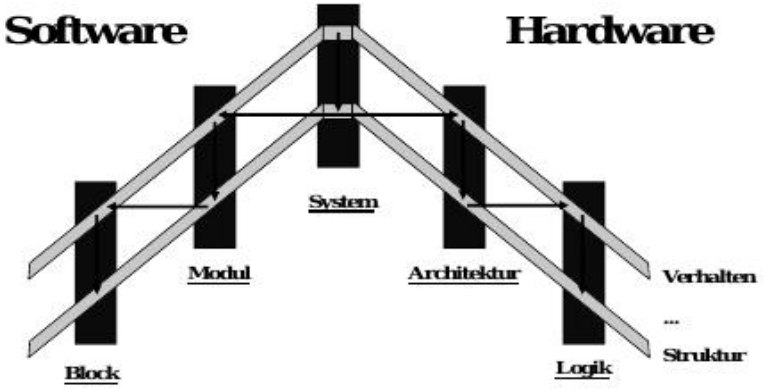
Ein eingebettetes System ist ein informationsverarbeitendes System bestehend aus Hardware und Software, das in einem technischen Gesamtsystem eingebettet ist, dessen primärer Zweck nicht das Verarbeiten und Aufbereiten von Information ist.

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

	Eingebettetes System	**konventioneller Rechner**
Funktion	*feste Funktion*: zum Zeitpunkt des Entwurfs ist die Funktion, die das System übernehmen soll, bekannt	*universell und offen*: zum Zeitpunkt des Entwurfs sind die Funktionen, die das System übernehmen soll, weitgehend offen
Verhalten	*reaktives Verhalten*: das System muss auf Aktionen der Umgebung unter festen zeitlichen Bedingungen reagieren	*interaktives Verhalten*: das System sollte mit der Umgebung unter sehr losen zeitlichen Bedingungen interagieren
Design	*Co-Design*: Hardware und Software werden aufeinander abgestimmt, entwickelt und optimiert	*unabhängiges Design*: 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 Echtzeitbedinungen
  2. gleichzeitige Entwicklung von Hardware und Software
- die Entscheidung, welche Funktionen in Hardware und welche in Software auf einem oder mehreren Mikrokontrollern 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

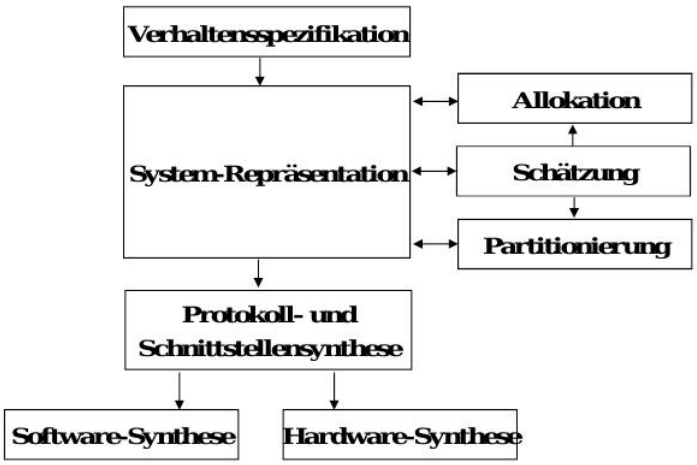
- **zustandsorientierten** (z.B. auf endlichen Automaten basierende)
- **aktivitätsorientierten** (z.B. auf Datenflussgraphen basierende)
- **strukturorientierten** (z.B. auf Blockschaltbildern basierende)
- **datenorientierten** (z.B. als Kollektion von Datenobjekten mit Relationen)

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. ⇒ 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

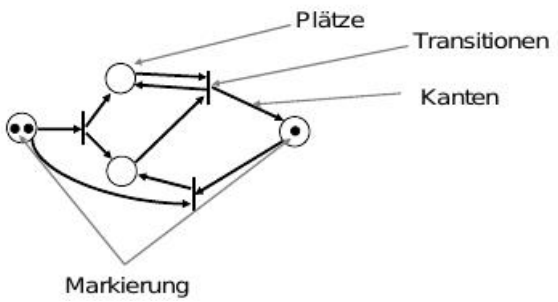
Formale Methoden zur Spezifikation von Systemen sind notwendig, um Realisierungen mit korrektem Verhalten synthetisieren zu können. Spezifizieren mit formalen Methoden ist nicht trivial... aber der einzige gehbare Weg!

Petri-Netze

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

- $P = \{p_1, \ldots, p_m\}$ ... die Menge der Plätze oder Stellen.
- $T = \{t_1, \ldots, t_n\}$ ... die Menge der Transitionen.
- $P \cap T = \varnothing$. Beide Mengen zusammen bilden die Menge der Knoten.
- $F \subseteq (P \times T) \cup (T \times P)$ ... die Menge der Kanten. Auch Flussrelation genannt.
- $K : P \to N \cup \{\infty\}$, die jedem Platz eine Kapazität zuordnet
- $W : F \to N$, die jeder Kante ein Kantengewicht zuordnet.
- $M_0 : P \to N_0$ 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:

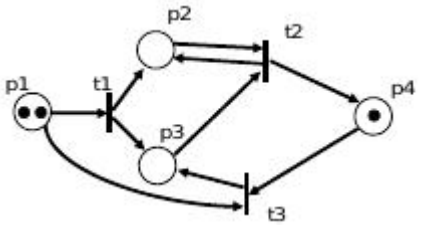


Abbildung 1.1: Transitionen t1 und t3 sind schaltbereit

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

$$M[t > M'$$

Aktivierbare und lebendige Transitionen

Definition: *Ist $M : P \to N_0$ 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 $B : P \to N_0 \cup \{\infty\}$ 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.

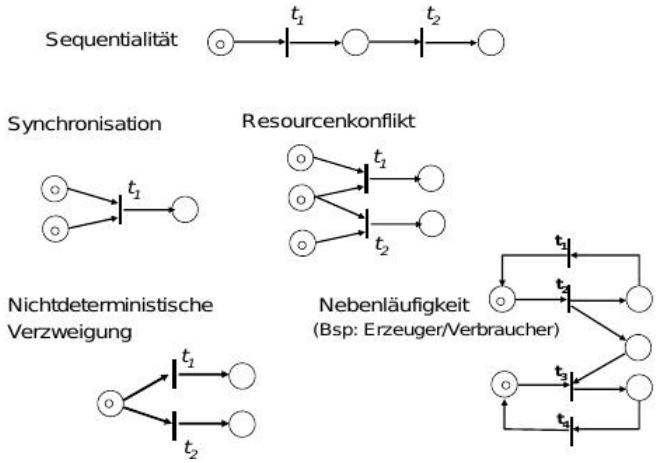


Abbildung 1.2: Ausdruckskraft von Petri-Netzen

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 $M : P \to N_0 \cup (\omega)$

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 $M : P \to N_0 \cup \{\omega\}$
- 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

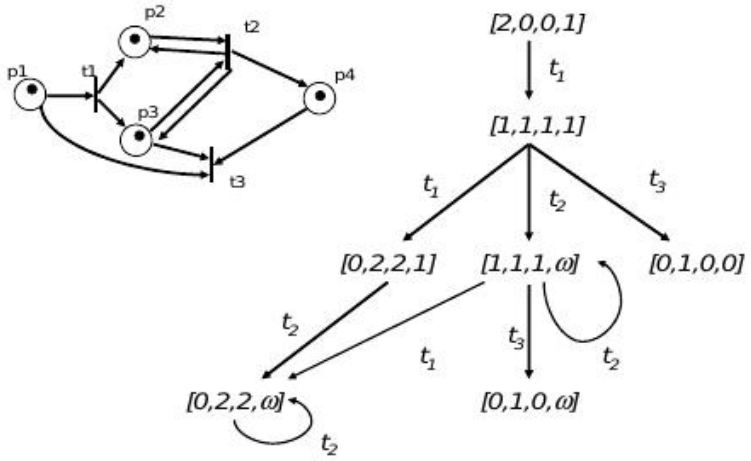


Abbildung 1.3: Beispiel für Überdeckungsgraph

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**
$\delta : S \times I \to S$ heißt **Übergangsfunktion**
$\lambda : S \times I \to O$ 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. ⇒ 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. ⇒ Notiere neben die Übergangsprädikate Aktionen.

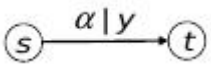
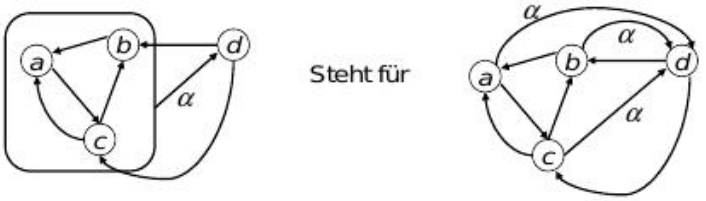
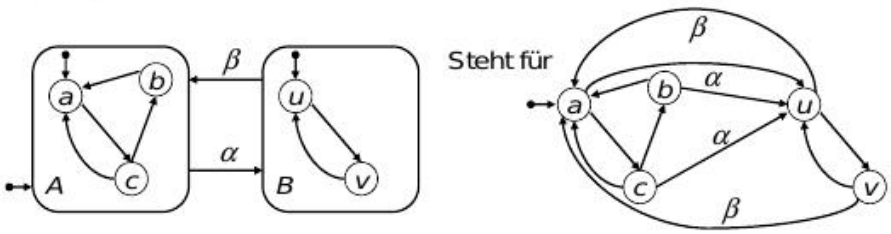


Abbildung 1.4: Unter allen Eingaben x mit $\alpha(x)$ gehe mit Ausgabe y nach t

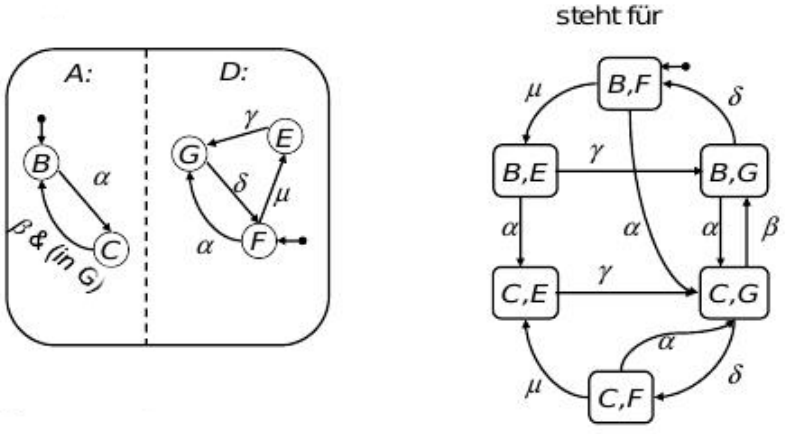
Statecharts – Gruppierung:
Gruppiere Zustände durch Einkreisen und erlaube Übergänge aus Zustandsgruppen. Ein Übergang aus einer Zustandsgruppe steht für Übergänge von jedem Zustand der Gruppe aus zu dem gegebenen Zielzustand.



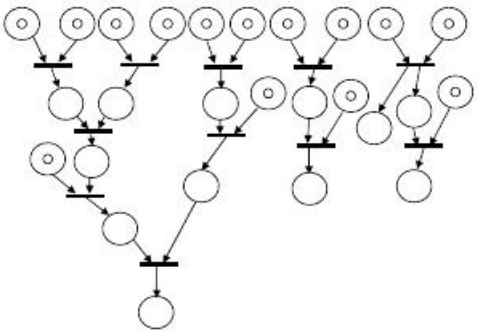
Statecharts – Hierarchie:
Gruppiere Zustände durch Einkreisen, benenne sie und markiere einen Startzustand.



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:

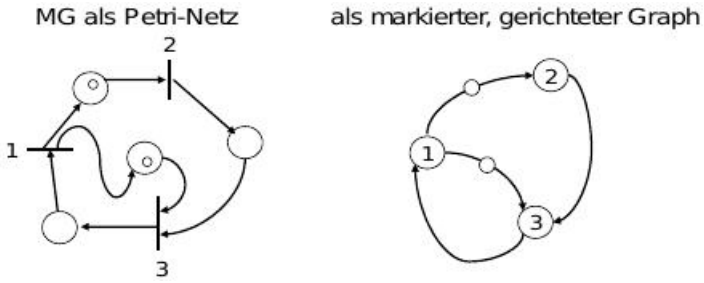


Jeder Platz hat genau eine Nachfolge- und eine Vorgängertransition, denn jede Datenabhängigkeit hat genau einen Produzenten und genau einen Konsumenten ⇒ markierte Graphen

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

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

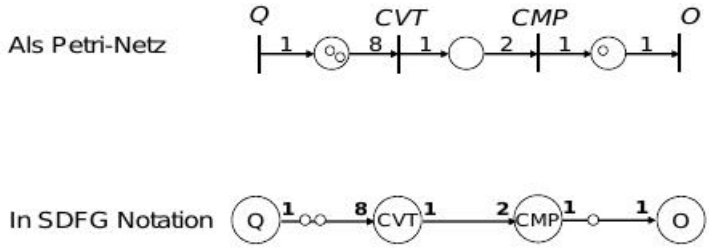
Marken jeder Stelle p werden in der Reihenfolge entnommen, in der sie dort entstehen (FIFO) ⇒ keine Konflikte auf den Transitionen. Da in MG die Plätze stets genau einen Vorgänger und einen Nachfolger haben, kann man die Plätze auch als Knoten im Netz weglassen und direkt die Kanten zwischen Transitionen 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 $M_0 : E \to N_0$ 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*

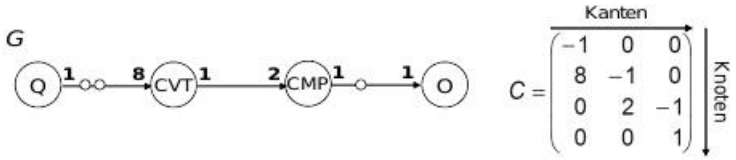
- V ist die Menge der Knoten – Knoten stehen für Operationen
- $E \subseteq V \times V$ ist die Menge der Kanten – aus Kanten können mehrere Daten liegen. Es gilt FIFO.
- $cons : E \to N$ gibt die Anzahl der beim Feuern des Zielknotens konsumierten Marken an – sie stehen am Pfeilende
- $prod : E \to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

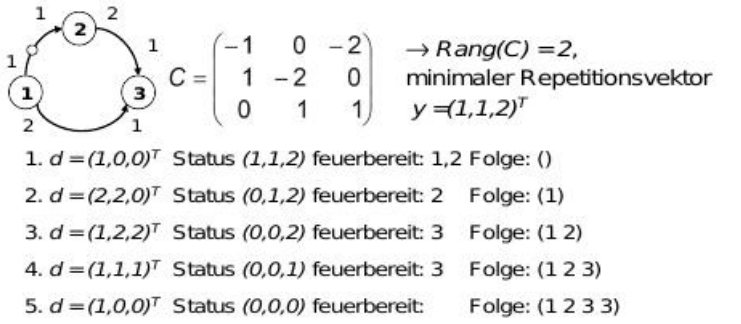
Satz: *Ist G zusammenhängend und konsistent, dann gilt $Rang(C) = |V| - 1$*

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 Repititionsvektors 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.

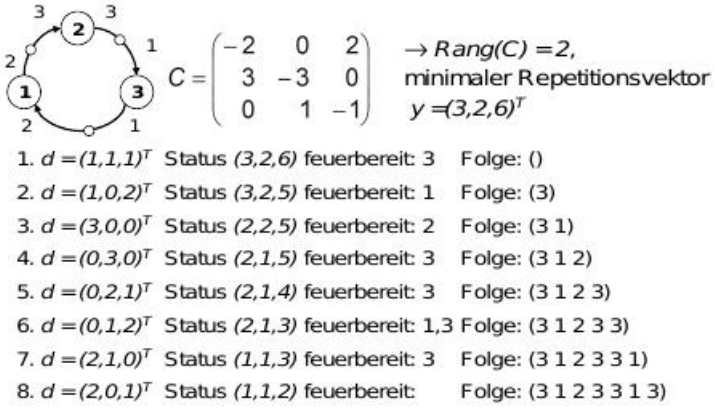
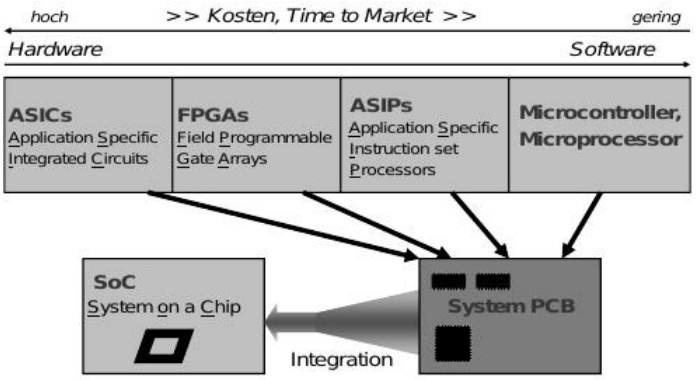


Abbildung 1.5: Beispiel für Verklemmung in einem SDFG

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 Modulaufruf (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** $c : V_T \to N_0$, die die Kosten einer Instanz eines Ressourcentyps angibt
- **Ausführungszeiten** $w : E_R \to N_0$, 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* $\alpha : v_T \to N_0$, *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* $\tau : V_S \to N_0$, *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 Sequenzgraphens* $G_S = (V, E)$ *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 Sequenzgraphens* $G_S = (V, E)$ *bzgl. eines Ressourcengraphen* $G_R = (V_R, E_R)$ *und einer Allokation* $\alpha : V_T \to N_0$ *ist ein Paar von Funktionen* $\beta : V_S \to V_T$ *und* $\gamma : V_S \to N$ *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

Chapter 2.1

Verifikation: *check that we are building the thing right*

Soll den Nachweis liefern, dass das System die Anforderungen immer erfüllt (Korrektheit der Implementierung in Bezug auf die Spezifikation).

Validation: *check that we are building the right thing*

Entspricht eher dem kritischen Überprüfen der Spezifikation anhand eines ersten Designs auf Richtigkeit und Vollständigkeit.
Testen: *ist ein Mittel für beides und weit verbreitet.*

Verifikation lässt sich damit nicht bewerkstelligen, allenfalls Falsifikation.

Populäre Techniken der Verifikation:

- **Peer Reviewing**: Statische Methode der manuellen Durchsicht und Prüfung des Codes durch einen Experten. Im Mittel bringt dies 60% der Fehler, subtile Fehler werden meist nicht gefunden.
- **Testen**: Dynamische Methode, häufig unterstützt durch Qualitätsmetriken (Code Coverage). 30% bis 50% der Gesamtkosten gehen auf das Konto von Testen, bei Hardware sind oft 70% des entwickelten Codes Testbenches.

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

Formale Methoden:

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

- **Deduktive Methoden**: Hier liefert man einen mathematischen Beweis, dass die Implementierung die Spezifikation erfüllt. Meist werden Theorembeweiser oder Beweisprüfer als Werkzeuge benutzt. Problem: Sehr aufwändig, das System muss die Form einer Mathematischen Theorie haben.
- **Model Checking**: Systematische und erschöpfende Überprüfung der Spezifikation auf allen erreichbaren Systemzuständen. Meist vollautomatisch durch sogenannte Modellchecker. Problem: Zustandsraumexplosion
- **Modelbasierte Simulation und Test**: Exploration möglicher Verhalten mit Überprüfung der Spezifikationen

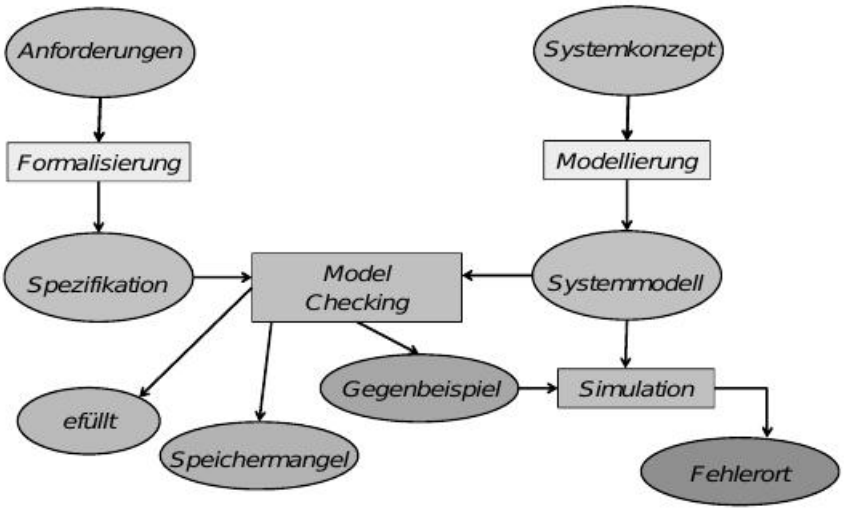
Model-Checking:

Man betrachtet

- **Spezifikation**: formale Definition der Eigenschaften des gewünschten Systems in Form von Sätzen einer mathematischen Theorie (Aussagen einer temporalen Logik)
- **Implementierung**: formale Definition der konkreten Konstruktion des Systems (als diskretes Transitionssystem)
- **Model Checking**: Ist die Implementierung ein Modell für die Theorie (gelten die Sätze für die Implementierung)

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

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



- $St \subseteq S$ eine Menge von Startzuständen ist

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

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

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

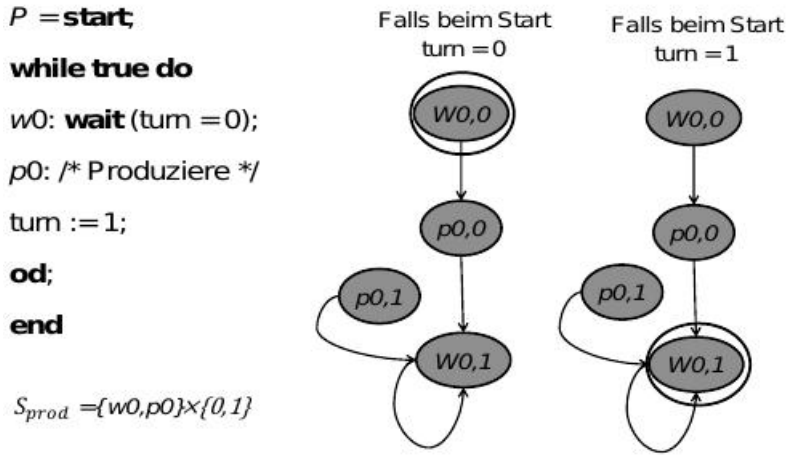


Abbildung 2.1: Producer als Transitionssystem

Kripke-Strukturen:

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

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

$$K = (S, (Act,) \rightarrow, St, AP, L)$$

Dabei kommen noch zwei Komponenten dazu:

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

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

$$AP = \{p, k\}$$

mit p: es wird gerade produziert
und k: es wird gerade konsumiert

Linear-Zeit vs. Baum-Zeit:

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

Die **Linear-Zeit Logik (linear time logic, LTL)** erlaubt Aussagen über Labelings aller (unendlichen) Berechnungsfolgen der Kripke-Struktur. LTL-Aussagen gelten entweder für alle Folgen oder nicht.

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

Chapter 2.2

Linear-Time Logic

Linear-Zeit Logik ist eine Logik, die auf Sequenzen von Belegungen mit atomaren Eigenschaften basiert. Die Zeit ist diskret und schreitet linear voran. Es gibt nur eine mögliche Zukunft.

Sei AP die Menge der atomaren Aussagen. Dann korrespondiert das Labeling $X \subseteq AP$ zu der Tatsache, dass alle Eigenschaften $x \in X$ in dem Zustand gelten und alle $y \in AP \setminus X$ in diesem Zustand nicht gelten. LTL Formeln werden nun über Mengen von unendlichen Folgen von Labelings interpretiert. D.h. wir betrachten Sequenzen $\sigma \in (2^{AP})^\omega$, wofür ein Alphabet Σ, Σ^ω die Menge aller unendlichen Folgen über Σ sei.
Für $\sigma \in (2^{AP})^\omega$ sei

- $\sigma(i) \subseteq AP$ das i-te Element der Folge
- $\sigma^i \in (2^{AP})^\omega$ die unendliche Folge $\sigma(i)\sigma(i+1)\sigma(i+2)\ldots$

Syntax von LTL:

Sei AP eine Menge von atomaren Aussagen. Dann ist die Menge der LTL Formeln über AP wie folgt definiert:

1. jedes $p \in AP$ ist eine LTL Formel
2. sind Φ_1 und Φ_2 Formeln, dann auch $\neg\Phi_1$, $\Phi_1 \vee \Phi_2$, $\mathbf{X}\Phi_1$, $\Phi_1\mathbf{U}\Phi_2$
3. nichts sonst ist eine LTL Formel

Dies ist eine sehr minimalistische Definition. **X** (next) und **U** (until) sind temporale Operatoren.

LTL Formeln werden über (unendlichen) Folgen von Teilmengen atomarer Aussagen interpretiert. Die Semantik einer LTL-Formel Φ ist die Menge aller Folgen $\sigma \in (2^{AP})^\omega$, die Φ erfüllen, d.h.

$$[[\Phi]] = \{\sigma | \sigma \vDash \Phi\}$$

Erfüllung einer LTL-Formel:

Sei $\sigma \in (2^{AP})^\omega$ und sei Φ eine LTL Formel. Dann gilt $\sigma \vDash \Phi$ ("σ erfüllt Φ") nach folgender Fallunterscheidung über die Struktur von Φ:

- $\sigma \vDash p$ falls $p \in AP$ und $p \in \sigma(0)$
- $\sigma \vDash \neg\Phi$ falls $\sigma \nvDash \Phi$
- $\sigma \vDash \Phi_1 \vee \Phi_2$ falls $\sigma \vDash \Phi_1$ oder $\sigma \vDash \Phi_2$
- $\sigma \vDash \mathbf{X}\Phi$ falls $\sigma^1 \vDash \Phi$
- $\sigma \vDash \Phi_1\mathbf{U}\Phi_2$ falls $\exists i : \left(\sigma^i \vDash \Phi_2 \wedge \forall k < i : \sigma^k \vDash \Phi_1\right)$

Nützliche Abkürzungen für LTL-Formeln:

F ... finally ("irgendwann"), **G** ... globally ("immer"), **W** ... weak until, **R** ... releases

- $\Phi_1 \wedge \Phi_2 \equiv \neg(\neg\Phi_1 \vee \neg\Phi_2)$

- $\Phi_1 \rightarrow \Phi_2 \equiv \neg\Phi_1 \vee \Phi_2$
- $true \equiv a \vee \neg a$
- $false \equiv \neg true$
- $\mathbf{F}\Phi \equiv true\mathbf{U}\Phi$
- $\mathbf{G}\Phi \equiv \neg\mathbf{F}\neg\Phi$
- $\Phi_1\mathbf{W}\Phi_2 \equiv (\Phi_1\mathbf{U}\Phi_2) \vee \mathbf{G}\Phi_1$
- $\Phi_1\mathbf{R}\Phi_2 \equiv \neg(\neg\Phi_1\mathbf{U}\neg\Phi_2)$

Tautologie, Unerfüllbarkeit, Äquivalenz:

Tautologie: Jede Formel Φ mit $[[\Phi]] = \left(2^{AP}\right)^{\omega}$ heißt Tautologie

Unerfüllbarkeit: Jede Formel Φ mit $[[\Phi]] = \varnothing$ heißt unerfüllbar

Äquivalenz: zwei Formeln Φ_1 und Φ_2 heißen äquivalent, gdw. $[[\Phi_1]] = [[\Phi_2]]$. Wir schreiben dann auch $\Phi_1 \equiv \Phi_2$

Idempotenz und Rekursionsgesetze:

- $\mathbf{F}\Phi \equiv \mathbf{FF}\Phi$
- $\mathbf{G}\Phi \equiv \mathbf{GG}\Phi$
- $\Phi\mathbf{U}\Psi \equiv \Phi\mathbf{U}(\Phi\mathbf{U}\Psi)$
- Rekursionsgesetze:
  - $\mathbf{F}\Phi \equiv \Phi \vee \mathbf{XF}\Phi$
  - $\mathbf{G}\Phi \equiv \Phi \wedge \mathbf{XG}\Phi$
  - $\Phi\mathbf{U}\Psi \equiv \Psi \vee (\Phi \wedge \mathbf{X}(\Phi\mathbf{U}\Psi))$
  - $\Phi\mathbf{W}\Psi \equiv \Psi \vee (\Phi \wedge \mathbf{X}(\Phi\mathbf{W}\Psi))$

Sicherheitseigenschaften:

Eine Eigenschaft ist eine Sprache $L \subseteq \left(2^{AP}\right)^{\omega}$. Gibt es für alle $\sigma \in \left(2^{AP}\right)^{\omega}/L$ einen endlichen Präfix w, so dass für alle $\alpha \in \left(2^{AP}\right)^{\omega}$ schon gilt $w\alpha \notin L$, d.h. es gibt keine zulässige Erweiterung mehr, mit der man den Präfix w wieder nach L bringen kann, dann nennt man w einen **schlechten Präfix** und L eine **Sicherheitseigenschaft**.

Lebendigkeitseigenschaften:

Es gilt $L \subseteq \left(2^{AP}\right)^{\omega}$. Gibt es für jedes $w \in \left(2^{AP}\right)^{*}$ ein $\alpha \in \left(2^{AP}\right)^{\omega}$ so dass wieder $w\alpha \in L$, d.h. es gibt stets eine zulässige Erweiterung, mit der man einen Präfix w nach L bringen kann, dann nennt man L eine **Lebendigkeitseigenschaft**.

Jede LTL-Eigenschaft lässt sich als Schnitt einer Sicherheitseigenschaft mit einer Lebendigkeitseigenschaft darstellen.

Interpretation von LTL über Kripke-Strukturen

Kripke Strukturen sind ja die formalen Modelle, die wir für Implementierungen von Systemen betrachten. Wenn wir nun wissen wollen, ob ein System in LTL formulierte Eigenschaften erfüllt, müssen wir nachprüfen, ob die Eigenschaft über den Menge aller (unendlichen) Ausführungsfolgen der Kripke Struktur gilt.

Sei $K = (S, \rightarrow, St, AP, L)$ eine Kripke-Struktur.

Eine Folge $\rho \in S^{\omega}$ mit $\rho(1) \in St$ und $\forall i : \rho(i) \rightarrow \rho(i+1)$ heißt Ausführung von K.

Für eine Ausführung ρ betrachten wir nun $L(\rho) \in \left(2^{AP}\right)^{\omega}$ mit $\forall i : L(\rho)(i) := L(\rho(i))$, d.h. die Labelingfolge zur Ausführung ρ.

Dann bezeichnen wir mit $[[K]]$ die Menge aller möglichen Labelingsequenzen zu Ausführungen von K.

$$[[K]] = \{L(\rho) | \rho \text{ ist Ausführung von } K\}$$

Das LTL-Model-Checking Problem

Das Problem ist es, zu entscheiden, ob eine Kripke Struktur, die wir aus der Implementierung abgeleitet haben, ein Modell für die Spezifikation ist.

Gegeben sei eine Kripke-Struktur $K = (S, \to, St, AP, L)$ und eine LTL-Formel Φ über AP. Entscheide, ob $[[K]] \subseteq [[\Phi]]$.

Es muss also für jede Ausführung ρ das Labeling $L(\rho)$ die Formel Φ erfüllen. Wir schreiben dann auch $K \vDash \Phi$. Vorsicht: es kann sowohl $K \vDash \Phi$ als auch $K \nvDash \neg\Phi$ gelten!

Fallstrick endliche Ausführungen:

Vorsicht: LTL Formeln werden nur über den unendlichen Ausführungen interpretiert.

Wenn die Kripke Struktur zum Beispiel Deadlocks enthält (Zustände ohne Nachfolger in der Transitionsrelation), dann werden alle Ausführungen, die auf einem solchen Zustand enden ignoriert, weil sie endliche Länge haben. Auf solchen Ausführungen wird die Formel nicht interpretiert.

Das kann im Extremfall unerwartete Konsequenzen haben: Angenommen in K führt jeder Weg in einen Deadlock. Dann ist $[[K]] = \varnothing$. Damit erfüllt K aber jede Formel!

LTL-Model-Checking

Wir wollen nun anschauen, wie man (im Prinzip, schnelle Heuristiken sind nach wie vor Gegenstand der aktuellen Forschung) Model Checking für LTL Formeln algorithmisch entscheidet:
Gegeben sei eine Kripke-Struktur $K = (S, \to, St, AP, L)$ und eine LTL-Formel Φ über AP. Entscheide, ob $[[K]] \subseteq [[\Phi]]$.
Im Prinzip gibt es zwei Möglichkeiten:

1. Konstruiere eine LTL-Formel Ψ mit $[[K]] = [[\Psi]]$ und zeige $\Psi \to \Phi$. Dies ist sehr aufwändig.
2. Die Menge der Ausführungen von K lässt sich leicht als Sprache eines Büchi Automaten $BA(K)$ d.h. $[[K]] = L(BA(K))$ auffassen. Konstruiere einen weiteren Büchi Automaten $BA(\neg\Phi)$ der die Sprache $[[\neg\Phi]]$ akzeptiert und entscheide
$$L(BA(K)) \cap L(BA(\neg\Phi)) = \varnothing$$

Büchi-Automaten

Büchi-Automaten sind im Prinzip endliche Automaten mit einem anderen Akzeptanzkriterium. Ein Tupel $B = (S, S_0, A, \Delta, F)$ heißt **Büchi-Automat**, wobei

- S ... eine endliche Menge von Zuständen
- $S_0 \subseteq S$ eine Menge von Startzuständen
- $\Delta \subseteq S \times A \times S$ die Übergangsrelation
- A ein endliches Alphabet
- $F \subseteq S$ eine Menge von Endzuständen (Akzeptanzzuständen) ist.

Man kann Büchi Automaten ebenso durch Diagramme darstellen, wie man das von endlichen Automaten kennt lediglich die Sprache ist anders definiert:
Sei $B = (S, S_0, A, \Delta, F)$ ein Büchi-Automat.
Für ein unendliches Wort $\sigma \in A^\omega$ ist $s(1)\sigma(1)s(2)\sigma(2)\ldots s(i)\sigma(i)\ldots$ ein Lauf in B mit Beschriftung σ genau dann wenn

- $s(1) \in S_0$ und
- für alle i gilt $s(i)\sigma(i)s(i+1) \in \Delta$

Ein Lauf heißt akzeptierend, wenn für unendlich viele i gilt: $s(i) \in F$.

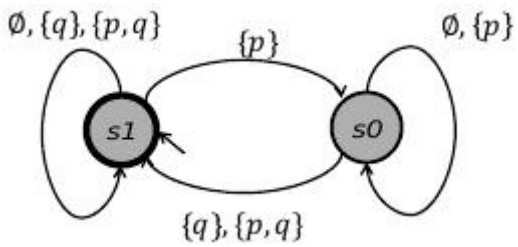
Ein Büchi-Automat $B = (S, S_0, A, \Delta, F)$ akzeptiert die Sprache $L(B) \subseteq A^\omega$ genau dann, wenn es für jedes $\sigma \in L(B)$ einen akzeptierenden Lauf mit Beschriftung σ gibt.

Büchi-Automaten und LTL:

Sei AP wieder eine Menge atomarer Eigenschaften. Setzt man $A = 2^{AP}$, dann sind Worte aus A^ω wieder unendliche Verhaltensmuster über AP.

Wir werden sehen, dass man zu jeder LTL Formel Φ über AP einen Büchi Automaten $BA(\Phi)$ angeben kann, mit $L(BA(\Phi)) = [[\Phi]]$ d.h. der nur Läufe akzeptiert auf denen die Formel gilt. Zeigen tun wir das zunächst an einem Beispiel:

Für die Formel $G(p \to Fq)$ über $AP = \{p, q\}$ akzeptiert folgender Automat die Sprache $[[G(p \to Fq)]]$



Der Produktautomat

Gegeben seien zwei Büchi-Automaten $B_1 = (S_1, S_{1,0}, A, \Delta_1, F_1)$, $B_2 = (S_2, S_{2,0}, A, \Delta_2, F_2)$ als Akzeptoren zweier ω regulärer Sprachen $L(B_1), L(B_2)$:
wir definieren den **Produktautomaten** $B_{1\times 2} = (S_{1\times 2}, S_{1\times 2,0}, A, \Delta_{1\times 2}, F_{1\times 2})$ durch

- $S_{1\times 2} := S_1 \times S_2$
- $S_{1\times 2,0} := S_{1,0} \times S_{2,0}$
- $\Delta_{1\times 2} := \{((s_1, s_2), a, (t_1, t_2))|(s_1, a, t_1) \in \Delta_1 \wedge (s_2, a, t_2) \in \Delta_2\}$
- $F_{1\times 2} := F_1 \times F_2$

LTL-Model-Checking

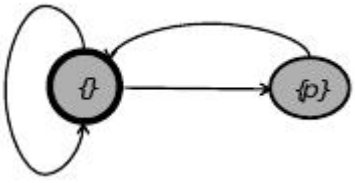
Wir haben nun eine Möglichkeit entwickelt, mit der man (im Prinzip, schnelle Heuristiken sind nach wie vor Gegenstand der aktuellen Forschung) Model Checking für LTL Formeln algorithmisch durchführen kann:
Gegeben sei eine Kripke-Struktur $K = (S, \rightarrow, St, AP, L)$ und eine LTL-Formel Φ über AP. Entscheide, ob $[[K]] \subseteq [[\Phi]]$.

1. die Kripke Struktur K stammt in der Regel von einer Spezifikation des Systems in Form eines Automaten. Bei unendlichem Betrieb lässt sich diese leicht durch einen Büchi-Automaten $BA(K)$ ausdrücken.
2. übersetze die LTL Formel Φ in einen Büchi-Automaten $BA(\neg\Phi)$ der die Sprache $[[\Phi]]$ akzeptiert.
3. bilde den Produktautomaten $BA(L(BA(K))) \cap L(BA(\neg\Phi))$ und entscheide, ob die Menge der akzeptierenden Läufe leer ist.
4. gibt es einen akzeptierenden Lauf, ist dieser ein Gegenbeispiel.

Chapter 2.3

Baum-Zeit-Logik
Linear-Zeit Logik ist eine Logik, die sich auf alle unendlichen Sequenzen von Belegungen mit atomaren Eigenschaften bezieht. Es gibt nur eine mögliche Zukunft. Man kann nicht über alternative Möglichkeiten den Entwicklung Aussagen machen. Baum-Zeit Logik erlaubt Aussagen über die Möglichkeiten eines Systems. Prominentester Vertreter ist CTL (Computation Tree Logic).

Beispiel (siehe Bild): $\Rightarrow$ lässt sich nicht in LTL adäquat beschreiben. Es gibt sowohl Folgen, in denen p nie gilt, als auch Folgen, in denen p irgendwann mal gilt.



Belegungsbaum zu einer Kripke-Struktur:

Sei $K = (S, \rightarrow, s_0, AP, L)$ eine Kripke-Struktur mit Startzustand s_0. Für $s \in S$ sei ein Belegungsbaum $T_K(s)$ zu K mit Wurzel s wie folgt definiert:

Zu jedem $s' \in S$ mit $s \rightarrow s'$ gibt es genau einen zu $T_K(s')$ isomorphen Unterbaum von $T_K(s)$.

$T_K(s)$ beschreibt quasi eine Entfaltung der Kripke-Struktur in einen unendlich tiefen Baum.

CTL-Formeln werden über Berechnungsbäume interpretiert. Sie machen bei jedem temporalen Operator

eine Aussage über die Menge der möglichen Entwicklungen im Baum durch existenzielle oder universelle Quantifizierung.

Syntax von CTL:

Sei AP eine Menge von atomaren Aussagen. Dann ist die Menge der CTL-Formeln über AP wie folgt definiert:

- jedes $p \in AP$ ist eine CTL-Formel
- sind Φ_1 und Φ_2 Formeln, dann auch $\neg\Phi_1$, $\Phi_1 \vee \Phi_2$, $\mathbf{EX}\Phi_1$, $\mathbf{EG}\Phi_1$,$\Phi_1\mathbf{EU}\Phi_2$
- nichts sonst ist eine CTL-Formel

EX ... exists next, **EG** ... exists globally und **EU** ... exists until sind temporale Operatoren.
Alle anderen Operatoren kann man durch Formeln über diese ausdrücken.

Nützliche Abkürzungen für CTL-Formeln:

- $\Phi_1 \wedge \Phi_2 \equiv \neg(\neg\Phi_1 \vee \neg\Phi_2)$
- $true \equiv a \vee \neg a$
- $\mathbf{EF}\Phi \equiv true\mathbf{EU}\Phi$
- $\mathbf{AX}\Phi \equiv \neg\mathbf{EX}\neg\Phi$
- $\mathbf{AF}\Phi \equiv \neg\mathbf{EG}\neg\Phi$
- $\Phi_1\mathbf{AU}\Phi_2 \equiv \mathbf{AF}\Phi_2 \wedge (\Phi_1\mathbf{AW}\Phi_2)$
- $\Phi_1 \rightarrow \Phi_2 \equiv \neg\Phi_1 \vee \Phi_2$
- $false \equiv \neg true$
- $\Phi_1\mathbf{EW}\Phi_2 \equiv (\Phi_1\mathbf{EU}\Phi_2) \vee \mathbf{EG}\Phi_1$
- $\mathbf{AG}\Phi \equiv \neg\mathbf{EF}\neg\Phi$
- $\Phi_1\mathbf{AW}\Phi_2 \equiv \neg(\neg\Phi_2\mathbf{EU}\neg(\Phi_1 \vee \Phi_2))$

A ... all (für alle Pfade), **F** ... future, **W** ... weak until

Beispiele: siehe Übungsblätter

3 Ablaufplanung

Unterscheidungskriterien

- statische vs. dynamische – statische Ablaufpläne treten auf, wenn zur Übersetzungs-/Synthesezeit ein Ablaufplan erstellt werden muss, vor allem bei hardwarenahem Implementationstechnicken. 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 Ressoure wieder frei ist)

Dieser Algorithmus liefert einen Ablaufplan minimaler Latenz → 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 ⇔ relative Zeitbeschränkungen zum Startknoten ⇒ 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 abgearbeiten 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 Nachfolgeaufträge 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

$$F^S_{v,t} = \frac{1}{slack(v)+1} \cdot \sum_{i=\tau_0(v)}^{\tau_1(v)} q_{\beta(v),i} - q_{\beta(v),t}$$

- 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 Definiton 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
- zyklostatische 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 und $W(u, v)$ ist das Maximum der Bearbeitungszeiten der Knoten auf allen Pfaden von u nach v.

3.2.2 Retiming

Wir versuchen durch Indexverschiebungen die Länge des Iterationsintervalls zu minimieren → neufestlegung der Interationsindizes. 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 $S(.)$ und $W(.)$ durch Lösung eines all pair shortest path Problems $O(|V||E|log|V|)$
- Sortiere die Menge $\{W(u, v)|u, v \in V\}$ $O(|V|2log|V|)$
- 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. $O(|V|3log|V|)$

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

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

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 zyklostatische 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 Ressourcetyp (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 intressiert ist.

Man kennt **time sharing Systeme** vorallem 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 einreiht. 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:

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

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

Die **deadlinemonotone Planung (DM-scheduling)** ordnet Jobs gemäß ihrer relativen Deadlines – kleinste relative Deadline zuerst. Falls t_d^* proportional zu $P(v)$ ist, sind RM und DM äquivalent. Sind die Deadlines kürzer als die Perioden, so ist DM der RM überlegen.

Beides sind Verfahren mit statischer Priorität.

Wir nennen die Priorität eines Jobs v_i nun $\pi(v_i)$. Wir nennen $R(v_i) = \max\{\tau_e(v_i,j) - (t_r^*(v_i) + j \cdot P(v_i)) | j \in \mathbb{N}\}$ die **maximale Flußzeit** oder auch **maximale Response** einer Instanz des Jobs v_i. Instanzen (v_i, j) mit Flußzeit $\geq \min\{t_d^*(v_i), R(v_i)\}$ heißen **kritisch**.

Lemma: ein System periodischer Tasks mit statischer Prioritätszuweisung $\pi(v_1) > \cdots > \pi(v_n)$. Dann liefert π einen Ablaufplan mit maximaler Response $R(v_k) \leq t_k$ für jedes t_k, dass folgende Ungleichung erfüllt:

$$t_k \geq w(v_k) + \sum_{i=1}^{k-1} \left\lceil \frac{t_k}{P(v_i)} \right\rceil \cdot w(v_i)$$

Dieses Lemma liefert uns bei vorgegebener Priorität ein hinreichendes Kriterium dafür, ob für jeden Job kürzer ist als dessen Deadline.

Ein System periodischer Jobs heißt **einfach periodisch**, wenn für alle Jobs v_i gilt: $P(v_{i+1})$ ist ein Vielfaches von $P(v_i)$ und $t_d^*(v_i) = P(v_i)$.

Satz: Ein System einfach periodischer Jobs hält unter DM/RM alle Deadlines, solange die Auslastung nie über 100% ist.

Ein System heißt **p-kritisch** genau dann wenn es ein Zeitintervall $[t - p_n, t]$ gibt mit: $t = t_d^*(v_n) + m \cdot p_n$, in dem zu jedem Zeitpunkt unter DM-Planung ein Job auf dem Prozessor läuft. Offensichtlich kann man bei diesen die Laufzeit keines Jobs mehr erhöhen, ohne dass der letzte Job seine Deadline verpasst.

Satz (Layland & Liu): Ein System von n periodischen Jobs mit $t_d^*(v) = P(v)$ ist stets korrekt DM/RM-planbar, solange für die Auslastung U gilt:

$$U \leq n(2^{\frac{1}{n}} - 1)$$

Dynamische Systeme wie EDF liefern für Auslastungen ≤ 1 stets einen Plan, welcher jedoch unvorhersagbar ist.

Einfaches Belegungslemma: Gegeben sei eine Menge von periodischen Jobs V ohne Datenabhängigkeiten mit $t_d^*(v) = P(v)$. Dann belegt jede Aufgabe v, die in einem Intervall der Länge t mindestens eine Deadline hat, die CPU in einem korrekten Ablaufplan für mindestens

$$\left\lfloor \frac{t - t_r^*(v)}{P(v)} \right\rfloor \cdot w(v)$$

Zeiteinheiten.

Satz von Liu: Gegeben sei eine Menge von periodischen Tasks V ohne Datenabhängigkeiten. Dann liefert dynamisches EDF Scheduling einen Ablaufplan, der alle Deadlines einhält, falls

$$1 \geq \sum_{v \in V} \frac{w(v)}{t_d^*(v)}$$

4 Architektursynthese

4.1 Binding

Eine **Bindung** eines Sequenzgraphen $G_S = (V_S, E_S)$ bezüglich eines Ressourcengraphen $G_R = (V_S \cup V_T, E_R)$ und einer Allokation $\alpha : V_T \to \mathbb{N}_0$ ist ein Paar von Funktionen $\beta : V_S \to V_T$ und $\gamma : V_S \to \mathbb{N}$ mit

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

Eine Bindung ordnet jeder Aufgabe eine verfügbare Instanz einer Ressource zu, die diese Aufgabe ausführen kann. Damit ein Ablaufplan also legal sein kann, darf zu jeder Zeit jeder Ressource nur eine Aufgabe zugeordnet sein.

Zwei Aufgaben können der gleichen Ressource zugeordnet werden, wenn sie vom gleichen Typ sind und sich ihre Ausführungszeiten nicht überlappen. Aufgaben können genau dann nebenläufig ausgeführt werden, wenn sie nicht auf einem gerichteten Pfad im Sequenzgraphen liegen.

Ein Knotenpaar $v_i, v_j \in V_S$ heißt

- **schwach verträglich**, wenn $\exists r_k \in V_R$ mit $(v_i, r_k) \in E_R$ und $(v_j, r_k) \in E_R$
- **ablaufplanverträglich**, wenn sie schwach verträglich sind und $\tau(v_i) \geq \tau(v_j) + w(r_k)$ oder umgekehrt
- **stark verträglich**, wenn sie schwach verträglich sind und es im Sequenzgraphen einen Pfad von v_i nach v_j oder umgekehrt gibt

Schwach verträgliche Knoten kann man grundsätzlich an die gleiche Ressource binden – allerdings sind dann evtl. nurnoch sehr stark sequentielle Pläne möglich. Ablaufplanverträgliche Knoten kann man unter Beibehaltung der Gültigkeit eines gegebenen Ablaufplans an die selbe Ressource binden. Stark verträgliche Knoten kann man immer an die selbe Ressource binden.

Der **Verträglichkeitsgraph** $G_V = (V_S, E_V)$ ist ein ungerichteter Graph, der genau dann eine Kante zwischen zwei Knoten hat, wenn sie auf eine Art verträglich sind. Er besitzt also immer mindestens genau so viele unabhängige Mengen, wie es Ressourcentypen gibt. Eine Menge gegenseitig verträglicher Aufgaben ist eine Clique (vollständiger Untergraph). ⇒ Minimierung der Kosten entspricht einer Überdeckung des Verträglichkeitsgraphen mit möglichst wenig Cliquen.

Leider ist Clique-Cover NP-vollständig, jedoch existieren brauchbare Heuristiken. Wir überführen dazu das Problem auf Minimum Vertex Cover auf dem **Hypergraphen** $H = (V, E, \rho)$ – wobei $\rho : E \to 2^V$ eine Funktion ist, die jeder Kante ihren Rand zuordnet – mit Bewertungsfunktion $c : E \to \mathbb{N}_0$. Um dieses Problem möglichst schnell lösen zu können, benötigen wir für den **Branch&Bound**-Algorithmus möglichst gute untere Schranken. Diese Schranken finden wir über das Gewicht der größten Clique des Hypergraphen. Diese größte Clique zu finden ist auch NP-vollständig. Die **Max-Clique-Heuristik** liefert jedoch ausreichend gute Ergebnisse mit Hilfe des Greedy-Verfahrens.

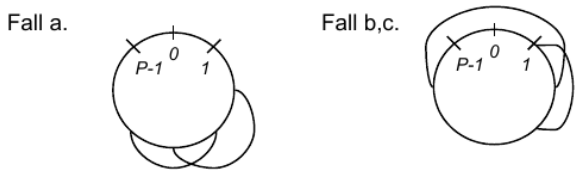
Duale Betrachtungsweise

Der **Ressourcen-Konfliktgraph** G_K ist der Komplementgraph des Verträglichkeitsgraph $G_V = (V_S, E_V)$, also $G_K = (V_S, \binom{V_S}{2} \setminus E_V)$ ($\binom{V_S}{2}$ ist die Menge aller zweielementigen Mengen mit Elementen aus V_S, also alle möglichen Kanten). Also entspricht eine maximale Verträglichkeitsmenge einer maximalen unabhängigen Menge des Konfliktgraphen. So lässt sich das Problem der kostenminimalen Bindung auch auf ein Graphfärbungsproblem des Konfliktgraphen mit möglichst wenig Farben überführen – was leider auch NP-vollständig ist.

Existiert bereits ein Ablaufplan, so lässt sich das Färben des Konfliktgraphen effizient exakt lösen, wenn es zu jedem Knoten nur einen Ressourcentyp gibt oder eine Typbindung schon gegeben ist. Diese effiziente Lösung arbeitet mit der speziellen Struktur des Konfliktgraphen, der sich in einen **Intervallgraphen** überführen lässt. Dabei wird jeder Kante ein rechtsoffenes Intervall zugeordnet, entsprechend der β-Funktion der Bindung. Im Intervallgraphen haben zwei Aufgaben genau dann eine Kante, wenn sich die Intervalle der beiden Knoten überlappen. Das Färben dieser Intervalle erfolgt über den **left-edge Algorithmus**.

Bei **Bindung unter periodischer Ablaufplanung** sind gegeben: ein iterativer Problemgraph $G_S = (V_S, E_S, s)$ und ein Ressourcengraph $G_R = (V_S \cup V_T, E_R)$, ein gültiger periodischer Ablaufplan mit Periode

P unter statischer Bindung und Ressourcentypbindung. Gesucht wird eine Bindung, die die Anzahl der benutzten Ressourcen minimiert. Beim Aufstellen des Konfliktgraphen ergeben sich unendlich viele periodische Intervalle, die sich auf unterschiedliche Art schneiden können. Eine geeignete Darstellung dieser Intervalle ist:



Bei einem **Graph mit zirkulären Kanten** wird jedem Knoten ein Kreissegment zugeordnet und zwischen zwei Knoten befindet sich genau dann eine Kante, wenn die Kreissegmente überlappen. Hier ist es ausreichend, diese Segmente zu kennen und sie zu färben. Dazu wird der **Sort&Match Algorithmus** verwendet:

- bestimme d_{min}, d_{max} und ein t mit $d_{min} = d(t)$ und die Menge V_A aller v für die gilt: t liegt auf dem Segment $(l(v), r(v))$.
- Überführe alle Segmente für $v \in V_B = V \setminus V_A$ in:

  $$l'(v) = (l(v) + P - t) \mod P \text{ und } r'(v) = (r(v) + P - t) \mod P$$

  ("Entrolle den Kreis ab der Stelle t und übernehme seine Segmente auf die entstehende Strecke")
- Färbe die Knoten aus V_B nach dem **Left-edge Algorithmus** (möglich, weil die ausgerollten Segmente Intervalle bilden!)
- Bestimme eine möglichst große Menge aus V_A, die mit den bisher vergebenen Farben gefärbt werden kann.
- Gebe jedem verbleibenden Knoten aus V_A eine bisher ungenutzte Farbe aus V_A

Dieser bildet eine Faktor-2-Approximation, es werden also höchstens doppelt so viele Farben verwendet wie in der optimalen Lösung. ($ALG = 2 \cdot OPT$)

Sowohl **Left-edge** als auch **Sort&Match** können dazu verwendet werden, ebenfalls die Anzahl der benötigten Register zu minimieren. Dazu muss jedem Job zusätzlich eine Lebensdauer zugeordnet werden. Aus diesen baut man dann Lebenszeitintervalle und verwendet die bereits bekannten Techniken, um sie mit möglichst wenig Farben zu färben.

4.2 Hardwaresynthese

Effektiv, aber teuer: Verbinden der Ressourcen und Register mittels **Multiplexern**. Dazu schaltet man einfach jedem Register einen ♯-Ressourcen zu 1 Multiplexer vor, der mit einem Steuerindex arbeitet. Jedem Operand wird außerdem ein ♯-Operationen zu 1 Multiplexer vorgeschaltet. Die Steuerung dieser Schaltung ergibt sich als Moore-Automat direkt aus Ablaufplan, Bindung und Registerbindung mittels eines zurücksetzbaren Latenzzählers. Dabei entsteht jedoch ein sehr komplexer Ausgabeschaltkreis – alternativ kann auch ein geeigneter ROM zum Nachschlagen der Ausgabe verwendet werden → mikroprogrammierte Steuerung. Probleme mit der Lösung durch Multiplexer:

- sehr teuer!
- Multiplexerdelays kommen zur Verarbeitungszeit hinzu.
- Halteoperationen der Register könnten auch in die Steuerregister integriert werden.
- Die Leistungsaufnahme ist hoch (koppeln auch inaktiver Elemente an Register).
- Es gibt keinen Weg, mehrere so erzeugte Komponenten zu einem System zu verschmelzen.

Alternativ könnte man die Implementierung auch **busbasiert** vornehmen. Es gibt dazu folgendes zu beachten:

- Billiger als Multiplexer, da die Registerports über Switches durchgeschaltet werden können.

- Busdelays kommen zur Verarbeitungszeit hinzu.
- Die Zahl der Busse muss nach dem maximalen Grad an Aktivitäten gewählt werden: Maximum aus Zahl der produzierten Resultatbits und Zahl der zu lesenden Operandenbits.
- Immernoch relativ teuer, da sehr viele Steuerleitungen benötigt werden – evtl. sogar mehr als bei Multiplexer-Lösung.

Es ist zu beachten, dass wir die ganze Zeit davon ausgehen, dass die Verarbeitungszeiten der Jobs für die Ressourcen, an die die Knoten des Problemgraphen gebunden sind, **Vielfache der Taktperiode** des Systems sind. Es genügt, nur Takte T zu betrachten, für die für mindestens einen Job$v$ $\frac{\delta(v)}{T}$ ganzzahlig ist.

Unter einer Taktperiode T ist der **mittlere Taktschlupf** gegeben durch:

$$mS(T) = \frac{\sum_{v \in V_T} \alpha(v) \cdot S(v,T)}{\sum_{v \in V_T} \alpha(v)}$$

Eine heuristische Möglichkeit, eine geeignete Taktperiode zu finden, besteht nun darin, für alle Taktperioden aus der Menge $\{\delta(v) | v \in V_T\}$ eine Periode mit minimalem Taktschlupf zu suchen und diese zu wählen.

4.3 Optimierungsverfahren zur Hardwaresynthese

Module mit Fließbandverarbeitung: Ressourcen mit Pipeline-Registern, die eine Periode $\pi(x)$ haben, mit der sie wieder gebunden werden können. **Bandbreitenbeschränkungen:** Bei busbasierten Zielarchitekturen hat man beschränkte Bandbreite auf den Operanden- und Resultatbussen, die von der Registerbank abhängen.

4.3.1 Erweiterung von List Scheduling

List Scheduling kann sehr leicht auf die Hardwaresynthese angepasst werden. Es funktioniert im wesentlichen genau wie normales List Scheduling, jedoch weisen wir den Jobs Ressourcen zu. Es kommen ähnliche Prioritätssysteme zur Anwendung. Diese Anpassungen funktionieren ohne Performanzverlust beim List Scheduling, es bleibt jedoch immernoch eine Heuristik. Durch das einführen von **Zeitschlitzen** abhängig von den Latenzschranken der Jobs lässt sich ein ILP aufstellen, dass uns einen Ablaufplan mit minimalen Ressourcenkosten liefert.

Im ILP gibt es zu jeder Zeitschlitz/Job/Ressourcen-Kombination eine Variable $x_{v,u,t}$. Außerdem gibt es für jeden Ressourcentyp u eine Variable y_u, ihr Wert gibt an, wieviele Ressourcen dieses Typs alloziiert werden müssen. $\Rightarrow$ Zielfunktion: $\min \sum_{u \in V_R} c(u) \cdot y_u$
Bindungsbeschränkungen: Es darf zu einem Zeitpunkt nur genau ein Job auf einer Ressource laufen. Die Zeitbeschränkungen für jeden Job müssen eingehalten werden. Keinen Ressourcentyp u dürfen mehr als y_u Aufgaben gleichzeitig belegen.

Um dieses System nun auf die Hardwaresynthese anwenden zu können, müssen die zu Grunde liegenden Ungleichungen geringfügig geändert werden. Man ersetzt $w(u)$ durch die Periode $\pi(u)$. Die startenden Aufgaben dürfen zu keinem Zeitpunkt t die Operandenbandbreite übersteigen – analoges gilt für die terminierenden Aufgaben und die Resultatbandbreite. Außerdem bedarf es binärer Variablen um die Lebensdauer der Knoten anzuzeigen und einer Variable, die die Registerzahl anzeigt – diese gilt es zu minimieren.

Die **Flussmodelle** sind lineare Programme, die sowohl ganzzahlige als auch reelwertige Variablen zulassen. Sie eignen sich zur Latenz- und Ressourcenminimierung. Dabei gibt es je Aufgabe und Ressource zwei Variablen, die Quellen und Senken des Flusses. Zusätzlich gibt es Flussvariablen, die angeben, dass eine Ressourceninstanz eine Aufgabe weitergibt und reelwertige Variablen für die Startzeiten und eine Latenzschranke.

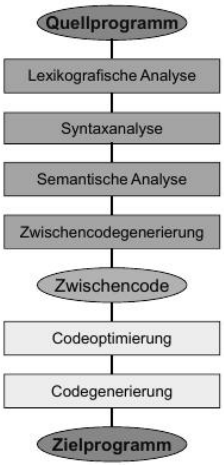
Die **Bindungsbeschränkungen** ergeben sich wieder aus dem Belegen der Ressourcen. Außerdem gibt es wieder Datenabhängigkeitsbeschränkungen, abhängig vom Delay. Eine Aufgabe kann eine Ressource aus der Quelle oder von einer anderen Ressource übernehmen. Ebenso geben sie Ressourcen an andere Aufgaben oder eine Senke weiter – es gilt also stets die **Flusserhaltung**. Für die Aufgaben gilt eine Latenzschranke, die auch Teil der Minimierung sein kann. Es gelten die gleichen Ressourcenbeschränkungen wie oben, außerdem dürfen Aufgaben, sie sich eine Ressource teilen, nicht gleichzeitig laufen.

Man betrachtet dann folgende Zielfunktion: $\min \sum_{u \in V_T} c(u) \cdot a_u$

5 Softwaresynthese

5.1 Übersetzer

Phasen eines Übersetzers



5.1.1 Zwischencode

Aufbrechen komplexer Konstrukte in 3 Adresskonstrukte → Portierbarkeit. Zerlegung in Unterprogramme, besteht nurnoch aus Codeeinheiten für einzelne Routinen. Lässt sich leicht in eine Graphdarstellung konvertieren. z.B. A3
Optimierungsaufgaben: Registervergabe – Bindung von symbolischen an physikalische Register. Codeminimierung – Quellcode → längenminimaler Maschinencode.

Zwischencode kann als **Kontrollflussgraphen** – Knoten-markierte Graphen – dargestellt werden. Ein **Grundblock** bezeichnet eine maximal lange Folge fortlaufender Anweisungen. Verschmelzen von Grundblöcken aus dem Kontrollflussgraphen zu Knoten → Grundblockgraph. Außerdem können Grundblöcke in **Taskgraphen** übersetzt werden. Grundblock → DAG → Tasksystem → Hardware.

5.2 Registervergabe und Registerbindung

Fest vorgegebene Registerzahl, Vergabe kann lokal oder global betrachtet werden, Konfliktgraphen sind nicht immer Intervallgraphen. Variablen in Blöcken können **Lebensspannen** zugeordnet werden. Das Problem der lokalen Registervergabe kann durch Graphfärben gelöst werden – left-edge Algorithmus.
Globale Registervergabe: Erfahrungsregel: Programme verbringen die größte Ausführungszeit in inneren Schleifen → Konzentration der Registervergabe auf Schleifen. In Schleifen ist darauf zu achten, eine möglichst kleine Zahl von Speicherzugriffen zu erzeugen.