# Hardwareverifikation: Einführung

Martin Steffen

Sommersemester 1998





## Gliederung

- Einleitung
- Ein wenig über Graphen
- Das Modell
- Berechnungen





## Ziele

- zentrale Frage: Vergleich von Architekturen: also

  Wann sind zwei Architekturen gleich?

- zentrale Antwort: falls die Programme

  keinen Unterschied feststellen können

- z.B: merken die Programme etwas
  - von der Einführung von Caches
  - vom Schritt vom Monoprozessor Multiprozessor mit gemeinsamen Speicher
  - vom Pipelining
- Formaler Rahmen für Architekturen
- Architektur:
  - Abstraktion des Rechnerarchtekturen aus Sicht des "Programmierers"
  - legt das Verhalten von Programmen fest, und zwar
  - gegeben als

    Architektur: Menge von Regeln

  - d.h. Architektur entpricht Semantik





## spezifischere Fragestellungen

1. wie formalisiert man den Begriff der Architektur?
2. wie spezifiziert man spezielle Architekturen?
3. wie vergleicht man gegebene Architekturen
4. welches sind wichtige spezielle Architekturen (bzw. Eigenschaften)

Die Antworten von Collier:

zu 1) graphentheoretische Darstellung, basierend auf Halbordnungen
zu 2) durch Angabe von Regeln, bzw. Konstruktion der Kanten
zu 3) Vergleich des I/O-Verhaltens der darauf laufenden Programme
zu 4) Das kommt in den einzelnen Abschnitten des Seminars . . .

## Multiprozessorsystem: intuitiv

- Programm = endliche, feste Anzahl von Prozessen
- Prozeß
  - endliche Sequenz von Anweisungen, d.h.,
  - keine Schleifen, Sprünge, oder ähnliches
  - nur
    - fetch: Lesen des Operanden
    - store: Schreiben des Operanden
  - lokale Kopie aller Variablen
- gemeinsamer Speicher
- globale Daten: auf die mehr als Prozess zugreifen kann
- Werteänderungen der Operanden können für die Prozesse zu unterschiedlicher Zeit sichtbar werden
- Kommunikation über gemeinsamen Speicher





## Syntax

**Definition 1 (Programm)** Sei *Var* eine unendlicher Vorrat an Variablen- oder Operandensymbolen. Die Menge der *Programme* ist durch folgende Grammatik gegeben:

| | | | |
|---|---|---|---|
| *stmnt* | ::= | *sink* := *source* \| | Statement |
| | | *sink* := *source* + *source* | Addition |
| *source* | ::= | *Var* \| | Variablen |
| | | *Int* | Integerliterale |
| *sink* | ::= | *Var* | Variablen |
| *proc* | ::= | *stmnt* \| | Process |
| | | *stmnt*; *prog* | Sequenz |
| *init* | ::= | $\epsilon$ \| *Var* = *Int*; *init* | Initialisierung |
| *initproc* | ::= | *init*; *proc* | init. Prozeß |
| *prog* | ::= | *initproc* \| | Programm |
| | | *prog* \|\| *initproc* | Parallelschaltung |





## Konkrete Syntax

Für die konkrete Syntax: folgende Konventionen

- Prozesse $P_1$, $Q_2$, ...
- globale Operanden: Großbuchstaben $A, B, \ldots,$ lokale mit Kleinbuchstaben
- Anweisungen seien eindeutig markiert: $L_1, L_2 \ldots$
- Wir aktzeptieren nur Programme, deren Initialisierung allen Operanden einen eindeutigen Wert zuweisen.

**Beispiel 2 (Programm)** Hier ein Beispiel für konkrete Syntax für ein Programm aus zwei Prozessen $P_1$ und $P_2$. In dem Beispiel stimmen die Anfangswerte aller Operanden für beide Prozesse überein. Aber: jeder hat eine eigene Kopie[1]

```
Initially  A = B = x = y = 0

           P1              P2

L1:      x:= B;          L1:  y := A;
L2:      A:= 1;          L2:  B := 1;
```

[1] Die Intialisierungsbedingung nennt man bei Datenbanken auch Konsistenzzusicherung oder Integritätscontraint.





## I/O-Spezifikationen (= *"Executions"*)

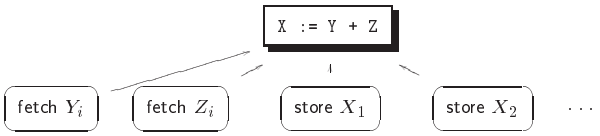
- Interesse am I/O-Verhalten der Programme ⇒ :
- I/O-Spezifikation bzw. Ausführung ist ein Programm (vgl. Definition 1, bei dem die erwartenen Resultate der Operanden für jeden Prozess angegeben sind.

**Beispiel 3 (Execution)** Das selbe Beispiel nochmal, erweitert um die Angabe der Resultate. Wieder sollen die die Resultatwerte für beide Prozesse für alle Operanden übereinstimmen.

```
Init          A = B = x = y = 0

           P1              P2

L1:      x:= B;          L1:  y := A;
L2:      A:= 1;          L2:  B := 1;

Terminally A = 1, B = 1, x = 0, y = 0
```

## Ereignisse

- Bisher: nur Syntax, bzw. Spezifikation des I/O-Verhaltens
- Modellvorstellung:
  - jeder Prozess mit eigener Kopie der Operanden/des Speichers
  - jede Anweisung zerfällt in mehrere atomare Operationen:
    * lokales Laden des/der Quell-Operanden
    * Rückspeichern des Ergebnisses in alle Kopien[2]
- Beispiel: sei $X := Y + Z$ ein Zusweisung aus Prozeß $P_i$:




[2] die eigentlichen Operationen, hier $+$, werden nicht modelliert.





## Ereignisse (II)

**Definition 4 (Ereignis)** Ein *Ereignis* ist ein Quintupel bestehend aus

1. Prozess
2. Label („Zeilennummer“ des Statements)
3. Art des Ereignisses $\in \{W, R\}$
4. betroffener Operand/Variable bzw. Literal
5. betroffener Speicher

Jedem Statement kann eine Menge von Ereignissen zugeordnet werden, genauso einem Programm.

Sobald wir Berechnungen haben, bekommen Ereignisse Werte zugeordet $\Rightarrow$ Sechstupel

6. Wert des Ereignisses (evtl. undefiniert) $\in Int \cup \{\perp\}$





**Beispiel 5 (Ereignisse)** Die zu dem Programm aus Beispiel 2 gehöhrenden Ereignisse lauten:

```
(P1, L1, R, _, B, S1)      (P2, L1, R, _, A, S2)
(P1, L1, W, _, x, S)       (P2, L1, W, _, y, S )
(P1, L2, R, 1, 1, S2)      (P2, L2, R, 1, 1, S2)
(P1, L2, W, 1, A, S)       (P2, L2, W, 1, B, S)
```

Dabei steht `(P1, L1, W, _, X, S)` *abkürzend* für die Ereignisse `(P1, L1, W, _, X, S1)` und `(P1, L1, W, _, X, S2)`.





## Graphen

**Definition 6 (Graph)** Ein Graph $\mathcal{G} = (S, R)$ besteht aus

- $S$: endliche Menge von Knoten
- $R \subseteq S \times C \times S$ einer Menge von beschrifteten Kanten. Wir schreiben oft

$$s_1 <_c s_2 \text{ für } (s_1, c, s_2) \in R$$

$C$ ist die Menge von Labels/Markierungen/Farben/Regelnamen.

- Pfad eines Graphen $s_1 <_{c_1} s_2 <_{c_2} s_3 \ldots$
- Kreis = Pfad, bei dem Anfang und Ende übereinstimmen

**Definition 7 (Graphmenge)** Für eine gegbene Menge von Kanten $S$ ist eine Graphmenge $\mathcal{GS}$ eine Menge von Graphen $\mathcal{G}_i = (S, R_i)$.

**Definition 8 (Produkt)** Gegeben zwei Graphmengen $\mathcal{GS}_1 = \{\mathcal{G}_{11} \ldots \mathcal{G}_{1n}\}$ und $\mathcal{GS}_2 = \{\mathcal{G}_{21} \ldots \mathcal{G}_{2m}\}$. Das “Produkt” ist definiert als die $n * m$-elementige Graphmenge

$$\mathcal{GS}_1 \times \mathcal{GS}_2 = \{\mathcal{G}_{11} + \mathcal{G}_{21}, \ldots \mathcal{G}_{1n} + \mathcal{G}_{2m}\}$$

## Architekturen

- Zur Erinnerung: Ziel ist Definition und Vergleich von Architekturen
- Im Prinzip:

Architektur = charakterisiert erlaubte Ordnungen der Events

- d.h. Architektur $\sim$ Halbordnungssemantik
- genauer: Architektur $\mathcal{A}$

$$\begin{array}{lcl} \mathcal{A} & : & prog \to \mathcal{GS} \\ \mathcal{A} & : & P \mapsto [\![P]\!]^{\mathcal{A}} \end{array}$$

- Wie kann man Architekturen beschreiben? $\Rightarrow$ $\mathcal{A}$ ist spezifiziert durch eine Menge von Regeln: $\mathcal{A}(R_1, \ldots R_n)$ $\Rightarrow$
- Architekturspezifikation $\sim$ operationeller (HO-)Semantik





## Spezifikation von Architekturen/Regeln

- prinzipiell: Regel = eine Menge von Kanten eines Graphen oder Graphsets $\Rightarrow$ nicht sehr praktikabel
- Methode zur Spezifikation von Kantenmengen:
  - Verbindung von Syntax und semantischem Bereich: hier: semantischer Bereich = Graphmengen. Verbindung:

    Knoten = Ereignisse
  - Spezifikation des Verhaltens durch Deduktionssystem. (hier Verhalten = Graphkanten)
    - Axiome: hier Templates
    - Inferenz"schema": später am Beispiel der Berechnungsregel CMP: Algorithmus zur Generierung von der entspechenden $<$-Relation
- das Vorgehen der Semantikgebung ist zumindest ähnlich zu (struktureller) operationeller Semantik
- Ein bestimmtes Deduktionssystem charakterisiert eine (Klasse von) Rechnerarchitekturen





## Templates

- Templates sind Grundbausteine der Architekturspezifikation
- Dienen auch zur Angabe des "Informationsflußes": Welche Information

**Definition 9 (Template)** Templates[3] sind Paare von bewerteten Events, mit

- Matchen: „=“
- "don't-care"-Termen: „_“

von der Gestalt

$$(\mathtt{P}, \mathtt{L}, \mathtt{A}, \mathtt{V}, \mathtt{O}, \mathtt{S}) <_R (\mathtt{P}, \mathtt{L}, \mathtt{A}, \mathtt{V}, \mathtt{O}, \mathtt{S})$$

- $A$: Aktion, Metavariable für $W$ oder $R$.
- $<_R$: Name der Regel

**Beispiel 10 (Templates)** Später mehr Beispiele, hier ein erstes:

$$(\mathtt{P}, \mathtt{L}, \mathtt{R}, \mathtt{V}, \mathtt{O}, \mathtt{S}) <_{SRW} (=, =, \mathtt{W}, \_, \_, \_)$$

[3] Collier nennt sie *transition templates*.





## Architekturen = Menge von Regeln

- Falls eine Relation $<_R$ durch mehrere Templates gegeben ist: Vereinigung der Kanten
- Architektur = Menge von Regeln $\mathcal{A}(R_1 \ldots R_n)$

$\Rightarrow$ Konjuktion der Einschränkungen der einzenlnen Regeln

$\Rightarrow$ Produkt der Graphmengen

- binärer Fall: $\mathcal{A}(R_1, R_2) = \mathcal{A}(R_1) \times \mathcal{A}(R_2)$

## Exkurs: Andere Halbordnungssemantiken





## Regel für Berechnungen

- Berechnungsregel (*computation rule(s)*): fundamentalste der Regel (aber komplex)
- Eine Architek
- besteht aus vier Unterregeln
  - SRW: Statement Read-Write
  - CWR: Compute Write-Read
  - CRW: Compute Read-Write
  - CWW: Compute Write-Write
- Intuitiv:

  Lesen und Schreiben der Operanden





## Statement Read-Write

- sehr einfache Regel
- für jedes Statement:
  - erst Operanden holen (R)
  - dann Ergebnis speichern (W)
- Template:

$$(\mathrm{P}, \mathrm{L}, \mathrm{R}, \_, \_, \_) <_{srw} (=, =, \mathrm{W}, \_, \_, \_)$$

### Beispiel 11 (SRW)

```
Initially  A = x = 0;

         P1            P2
L1:   A := 1;      L1: x:= A

Terminally A = 1, x = 0;
```





## Konflikte

- Neben der Statement-Regel, die die Zugriffe für ein Statement regelt:
- Konflikt: die Reihenfolge der Events kann das Ergebniss der Berechnung beeinflußen
- Konflikte bei Zugriff auf selbe Speicherzelle:
  - Lese/Schreib-Konflikt
  - Schreib/Schreib-Konflikt

## Compute-Regeln W-R/R-W/W-W

- Lese- und Schreibzugriff auf die selbe Speicherzelle $\Rightarrow$ Konflikt
- drei unterschiedliche Konflikte
- Compute Read-Write
  - Read vor Write
  - Template:

$$(\_, \_, \mathtt{W}, \mathtt{V}, \mathtt{O}, \mathtt{S}) <_{cwr} (\_, \_, \mathtt{R}, =, =, =)$$

  - beachte: der Wert wird übernommen
- Compute Write-Read
  - Write vor Read
  - Template:

$$(\_, \_, \mathtt{R}, \mathtt{V}, \mathtt{O}, \mathtt{S}) <_{crw} (\_, \_, \mathtt{W}, \_, =, =)$$

- Copmpute Write-Write
  - 2 Writes
  - Template:

$$(\_, \_, \mathtt{W}, \mathtt{V}, \mathtt{O}, \mathtt{S}) <_{cww} (\_, \_, \mathtt{W}, \_, =, =)$$





## Generierung der CMP-Semantik

- Ziel: Generierung der Graphmenge für Berechnungen CMP
- Schrittweises Vorgehen:
  1. Partitioniere die Menge der Ereignisse in maximale Konfliktmengen:
     - daß jede Partition nur Events enthält, die paarweise miteinander in Konflikt stehen[4]
     - keine zwei Events zweier verschiedener Partitionen in Konflikt stehen.
  2. Linearisierung der Konflikte: bei $n$ Ereignissen in einer Konfliktenge: $n!$ Anordnungen $= <_{sos}$-Ketten
  3. Benutze SOS-Ordnung um
     (a) CWR
     (b) CRW
     (c) CWW
     zu bestimmen (siehe nächste Folie)
  4. Füge die SRW-Ordnung hinzu
  5. Wertzuweisung an die Ereignisse

[4] zur Not einelementig





## Bestimmung der Graphmenge

Gegeben sei ein Programm $P$. Seine zugehörige Eventmenge sei $\mathcal{E}$. Die Partitionierung in maximale Konfliktmengen sei $\mathcal{E} = \bigcup \mathcal{E}_i$. Für eine gegebene Linearisierung $\mathcal{E}^l$ sind die Relationen $<_{cwr}$, $<_{crw}$, $<_{cww}$ und $<_{srw}$ definiert wie folgt:

$$\frac{}{(P, L, R, \_, \_, \_) <_{srw} (P, L, W, \_, \_, \_)} \quad \text{(SRW)}$$

$$\frac{(e, e) \in \mathcal{E}_i^l \text{ für ein } \mathcal{E}_i^l \in \mathcal{E}^l}{e <_{sos} e'} \quad \text{(SOS-Ax)}$$

$$\frac{e_1 <_{sos} e_2 \qquad e_2 <_{sos} e_3}{e_1 <_{sos} e_3} \quad \text{(SOS-Trans)}$$

$$\frac{w <_{sos} r}{w <_{cwr} r} \quad \text{(CWR)}$$

$$\frac{r <_{sos} w}{r <_{crw} w} \quad \text{(CRW)}$$

$$\frac{w_1 <_{sos} w_2}{w_1 <_{cww} w_2} \quad \text{(CWW)}$$

$\Rightarrow$ jedes $\mathcal{E}^l$ liefert einen Graphen

$\Rightarrow$ $\mathcal{E}$ liefert eine Graphmenge

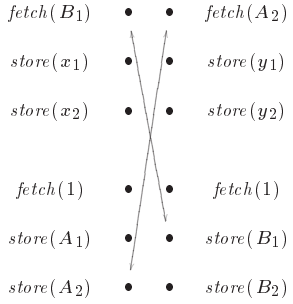




## Beispiel: Konfliktmengen

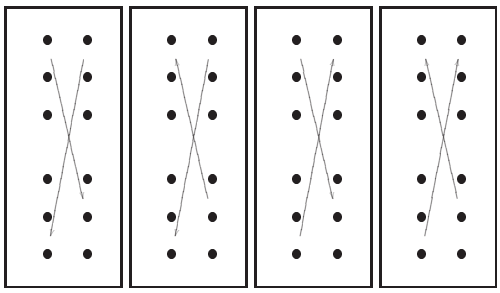
**Beispiel 12**

```
Init        A = B = x = y = 0

          P1              P2

L1:     x:= B;          L1:  y := A;
L2:     A:= 1;          L2:  B := 1;

Terminally A = 1, B = 1, x = 0, y = 0
```

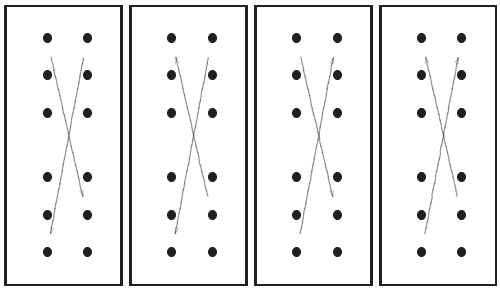


## Beispiel Graphmengen: SOS

Bestimmung der Konflikte ergibt 4 $<_{sos}$-Graphmengen:



Die Klassifizierung nach Konfliktarten führt zu

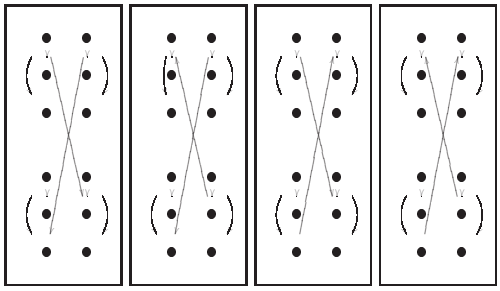






## Beispiel: SRW

Die Hinzuname der SRW-Abhängigkeiten führt zu folgendem Graphen:







## Werteberechung

- die Ereignisse einerBerechnung[5] können mit Werten versehen werden.
- jedem Ereignis eines Graphen in einer Graphmenge wird ein Integer-Wert zugewiesen.
- Wert = Wert des betroffenen Operanden im Ablauf
- Algorithmus zur Berechnung der Werte: Gegeben ein azyklischer Event-Graph $\mathcal{G}$ zu einem Programm $P$

1. Bestimme alle $<$-minimale Ereignisse $e$ (dies sind wg. SRW $r$-Ereignisse) $\Rightarrow$ $val(e) = init(e.o)$.
2. Bestimme die Menge $W$ der Wert-freien Schreib-Ereignisse $w$, für die gilt: $e <_{srw} w$, dann $val(e) \neq \bot$. Setze $val(w) := \sum\{val(e) \mid e <_{srw} w\}$ für alle $w \in W$.
3. Bestimme die Menge $R$ aller Wert-freien Lese-Ereignisse $r$, für die gilt: falls $w <_{cwr} r$ dann $val(w) \neq \bot$; Setze $val(r) := val(w)$. Fahre fort mit 2.
4. Falls 2 oder 3 keine neuen Ereignisse mehr liefern $\Rightarrow$ Stop

[5] also einer Graphenmenge, die den CMP-Regeln gehorcht





## Werteberechnung II

**Beobachtung 13 (Terminierung)** Bei einem gegebenen ayzklischen CMP-Graphen terminiert der Algorithmus.

**Beobachtung 14** In jedem Graphen hat jedes $r$-Ereignis maximal einen $<_{cwr}$-Vorgänger.

**Beobachtung 15 (Determinismus)**

- Der Algorithmus ist wohldefiniert, deterministisch[6] und nach der Terminierung besitzt jedes Ereignis einen Wert.
- Nach der Terminierung besitzt jeder Operand lokal einen deterministischen Wert. Die Ausgabe/ schließliche Speicherbelegung eines Graphen $\mathcal{G}$ bezeichnen wir mit $\mathcal{G}_{out} : proc \times Var \to Int$.

**Beweisskizze:** Wohldefiniertheit und Determinismus von Punkt 2 hängt von Beobachtung 14 ab. □

**Frage 16 (Write-Write)** Preisfrage: Warum spielen Write-Write-Konflikte keine Rolle.?

[6] für einen Graphen

## Semantik

Nun sind wir endlich soweit, Semantik und Erfülltheit zu definieren.

**Definition 17 (Semantik)** Die *Semantik* eines Programmes $P$ unter einer Architektur $A$ ($[\![P]\!]^{\mathcal{A}}_{\mathcal{G}}$) sei die Menge der azyklischen Graphen.

Falls die Architektur eine Berechnungen fordert (d.h. Graphen gemäß CMPgeneriert), dann kann man die Erergnisse der Graphen, wie gezeigt bewerten. $\Rightarrow$ die Menge der Beobachtungen sei in diesem Fall die Menge der resultierenden Speicherbelegungen

$$[\![P]\!]_{out} = \{\mathcal{G}_{out} \mid \mathcal{G} \in [\![P]\!]_{cmp}\}$$

Eine I/O-Spezifikation $P, O$ ist erfüllbar durch eine Architektur $\mathcal{A}$ (oder beobachtbar auf $\mathcal{A}$, $P \models_{\mathcal{A}} O$), falls $O \in [\![P]\!]^{\mathcal{A}}_{out}$





**Beispiel 18** Für unser Beispiel sind unter $\mathcal{A}(CMP)$ vier mögliche Resultate $[\![P]\!]_{out}$ beobachtbar:

| $A$ | $B$ | $x$ | $y$ |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 |





## Vergleich von Architekturen

- Nach der Definition der Semantik: formaler Vergleich von Architekturen mögliche (siehe zu Beginn):

**Definition 19 (Ununterscheidbarkeit)** Zwei Architekturen $\mathcal{A}_1$ sind $\mathcal{A}_2$ sind *ununterscheidbar*, wenn

$$[\![P]\!]^{\mathcal{A}_1}_{\mathcal{G}} \neq \emptyset \text{ gdw. } [\![P]\!]^{\mathcal{A}_2}_{\mathcal{G}} \neq \emptyset$$

Im Falle von Berechnungen verwedet man besser als Definition

$$[\![P]\!]^{\mathcal{A}_1}_{out} = [\![P]\!]^{\mathcal{A}_2}_{out}$$





## Verwandte Themengebiete

Ähnliche Probleme auf verschiedenen *Ebenen*, unter anderem

- Transaktionen bei Datenbanken, Concurrency Control
- Sequentialle Konsistenz von DSM-Systemen
- ...

## Serialisierbarkeit & Transaktionen

- wichtiger Begriff aus der Theorie der Datenbanken[7]
- atomare Datenbank ⇒ klassischer Korrektheitsbegriff für Datenbanken:
- Korrektheitsmaßstab = serieller Zugriff.
- seriell ist *ineffizient* ⇒ serialisierbar

  *Es existiert eine Ausführung des (Datenbank- )Systems in der die Transaktionen nicht-überlappend also seriell ausgeführt werden und die zum selben Ergebnis führen. (Nach [LMWF94], Original [EGLT76])*
- Transaktion: logische Einheit von mehreren Aktionen auf eine Datenbank. Ationen unterschieden in
  - Schreib- (W) und Leseaktionen (R). ⇒ jede Transaktion bekommt eine R-Set und ein W-Set ⇒
  - Konflikte zwischen Transaktionen: W-W, R-W, W-W.

**Beispiel 20 (Konflikte)**

[7] Das Gebiet ist bekannt als *database concurrency control*.





## Serialisierbarkeitstheorem

- "klassisches" Ergebnis aus der Datenbanktheorie
- verbindet die Serialisierbarkeit eines Ablaufs mit einer graphentheoretischen Aussage über seine Serialisierungsgraphen (siehe auch [BHG87]).
- ⇒ Kriterium für die Korrektheit *einer* parallelen Ausführung ("out-of-order"-Execution) von Transaktionen

**Satz 21 (Serializability [EGLT76], [SLR76] . . . )** Ein Log oder Schedule $L$ ist *serialisierbar*, genau dann wenn sein Graph $SG(L)$ *azyklisch* ist.

- $SG$: Serialisierungsgraph:
  - Knoten = Transaktionen
  - Kanten = Konflikte (R-W, W-R, W-W)
- Log über einer Menge $\mathcal{T}$ von Transaktionen
- $SG$: Serialisierungsgraph:
  - Knoten = Transaktionen
  - Kanten = Konflikte (R-W, W-R, W-W)
- Scheduler: Constraints auf die Ausführungen, sodaß nur serialisierbare Logs/Schedules enstehen





## Sequentielle Konsistenz

- Frage: Wann ist ein Programm auf einem DSM korrekt:
- häufige Antwort: wenn es sequentiell konsistent ist.

  *Ein System ist sequentiell konsistent, wenn das Ergebnis jeder möglichen Ausführungsreihenfolge auf allen Prozessores die selbe ist,*
  - *als wären die Instruktionen in irgendeiner sequentieller Reihenfolge ausgeführt, und*
  - *projiziert auf jedem einzelnen Prozessor erscheinen die Operationen der Seuquenz in der Reihenfolge, wie sie auch im Programm hintereinander stehen.*

  (nach [Lam79])

- Sequentielle Konsistenz. Wichtig beispielsweise für Compiler, Beispiel: gcc:

  `-mserialize-volatile -mno-serialize-volatile`

  Do, or don't, generate code to guarantee sequential consistency of volatile memory references. . . . When consistency must be guaranteed, GNU C generates special instructions, as needed, to force execution in the proper order. The MC88100 processor does not reorder memory references and so always provides sequential consistency. However, by default, GNU C generates the special instructions to guarantee consistency even when you use '-m88100', so that the code may be run on an MC88110 processor. If you intend to run your code only on the MC88100 processor, you may





use '-mno-serialize-volatile'. The extra code generated to guarantee consistency may affect the performance of your application. If you know that you can safely forgo this guarantee, you may use '-mno-serialize-volatile'.

-

## Ausblick

Der Rest des Buches

- verschiedene Architekturen,
- deren Beziehungen
- Beispiele:
  - Regeln für Programmordnung
  - Atomarizität
  - ...





## Literatur

Literatur

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locking in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LMWF94] Nancy Lynch, Michael Merrit, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kauffman, 1994.

[SLR76] R. E. Stearns, P. M. Lewis, and D. J. Rosenkrantz. Concurrency control for database systems. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, 1976.