

**Eigenschaften mobiler und  
eingebetteter Systeme:**

# **Software und Fehlertoleranz**

Dr.-Ing. Matthias Werner

Dipl.-Inf. Jan Richling

Wintersemester 2001/2002

# Bedeutung von Software

- Software spielt im Verhältnis zur Hardware ein immer größere Rolle
  - Hohe Komplexität, unsystematischer Entwurf ⇒ mögliche Fehlerursachen
  - Bestimmt Zusammenspiel der Hardware, leichte Änderbarkeit ⇒ mögliches Mittel der Fehlertoleranz
- Entsprechend werden zwei Gebiete betrachtet
  - Modelle für SW-Verlässlichkeit
  - SW-Techniken für Verlässlichkeit und Fehlertoleranz

# Modelle für SW-Zuverlässigkeit

- Lebenszyklen von Software sind schwer mit denen von Hardware vergleichbar
- Analogien
  - Fehlerrate  $\lambda$  - *hazard rate*  $z(t)$ : proportional zur Zeit zwischen der Entdeckung von zwei Fehlern
  - Lebenszeit - Kalenderzeit oder Laufzeit
- Vergleichbarkeit von SW-Projekten ist schwer - nachträglich können gute Modelle aufgestellt werden
- Noch viel zu forschen

# Modell nach JELINSKI/MORANDA

- Annahme: Hazard-Rate ist proportional zu den verbleibenden Fehlern
- $z(t) = C(n - (i - 1))$ 
  - $C$ : projektabhängige Proportionalitätskonstante
  - $e_p$ : Anzahl der ursprünglich im Programm vorkommenden Fehler
  - $i$ : Nummer des entdeckten Fehlers
- $z(t)$ : ist stückweise konstant
- $R(t) = e^{-C(e_p - (i-1))t}$
- $MTTF = \frac{1}{C(n - (i-1))}$

# Ähnliche Modelle

- Berücksichtigung von
  - Programmgröße
  - Behebung der Fehler
  - Programmlaufzeit statt Kalenderzeit
- SHOOMAN:  $R(t) = e^{-C \left( \frac{ep-ec}{s} \right) t}$ ,  $ec$ : bisher korrigierte Fehler,  $s$ : Anzahl Programm-Statements
- MUSA:  $R(t) = e^{-C \cdot T_A \left( \frac{ep-ec}{s} \right) t}$ ,  $T_A$ : durchschnittliche Zeit pro Befehl

# Ansatz nach SCHNEIDEWIND

- Annahme: Analytisch ist die SW-Zuverlässigkeit schwer zu erfassen.
- Empirischer Ansatz
- Ausprobieren “typischer” Zuverlässigkeitsverteilungen, wie Exponenti-  
alverteilung, Gamma-Verteilung, Weibull-Verteilung, Normalverteilung,  
etc.
- Auswahl der für das augenblickliche Projekt besten Verteilung
- Methode: z.B.  $\chi^2$ -Test
- Nutzen: Vorhersage künftigen Verhaltens bei ausreichend langen Beob-  
achtungszeitraum

# Komplexitätsmodelle

- Komplexitätsmodelle sollen den Schwierigkeitsgrad beim Entwurf (Programmieren) angeben
- Hilfe, um Zuverlässigkeit zu bestimmen
- Aber: Viele verschiedene Modelle, nichts ultimatives
- Typisches Maß: Fehler pro Zeile Quellcode
  - Bei industrieller Produktion gefordert: 0.3 - 2 Fehler pro 1000 Zeilen Quellcode
  - In Wirklichkeit typisch: Ein Fehler auf 100 Zeilen Code
  - Beobachteter Fall: 2.2 Fehler pro Zeile(!) Code

## Statistische Ansätze

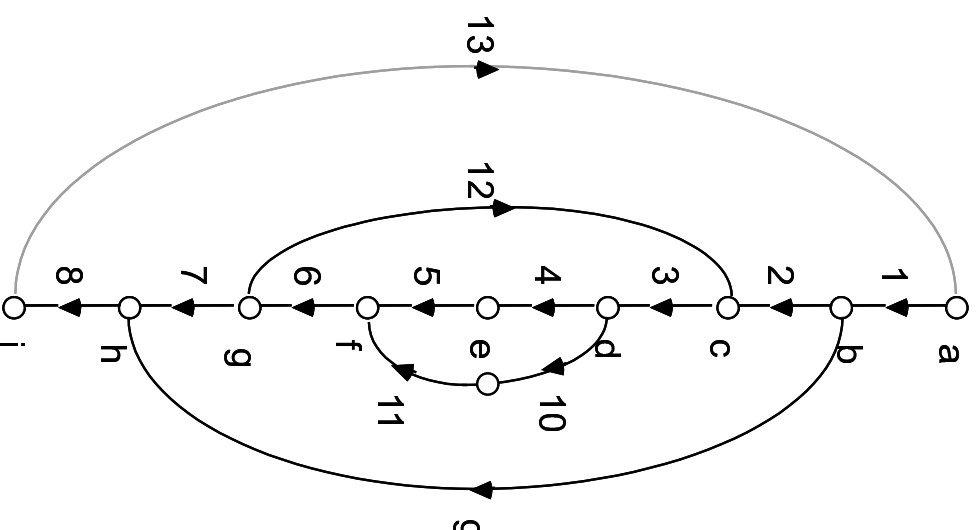
- Annahme, daß die Komplexität von Programmen von der Anzahl von **Operatoren** ( $n_1$ ) (direkte und Programmstatements) und der Anzahl von **Operanden** ( $n_2$ ) abhängt.
- Komplexität wird häufig als Länge  $N$  ausgedrückt.
- Regel nach ZIPF:  $N = (n_1 + n_2) \cdot (0.5772 + \ln(n_1 + n_2))$
- HALSTEAD:  $N = n_1 \ln n_1 + n_2 \ln n_2$

# Graphentheoretische Ansätze

- Untersuchung der Komplexität des Kontrollflusses
- Programmfluß wird als Graph  $G$  aufgefaßt (z.B. PAP)
- Ansatz nach MCCABE: Zyklomatische Komplexität
  - Zyklomatische Nummer eines stark verbundenen Graphen:
$$\mu(G) = E - N + K$$
  - $E$ : Anzahl der Kanten
  - $N$ : Anzahl der Knoten
  - $K$ : Anzahl der einzelnen Teile (bei zusammenhängendem Graphen 1)

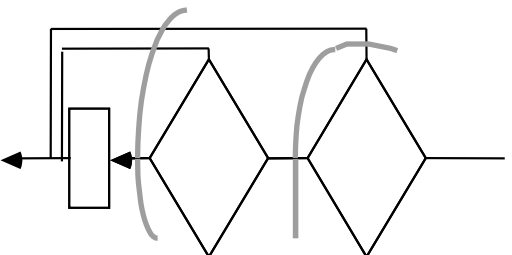
# Zyklomatische Komplexität

- Die zyklomatische Nummer eines PAP kann als Komplexitätsmaß angesehen werden
- Damit der Graph stark verbunden ist, wird eine Phantomkante hinzugefügt (grau)
- Im Beispiel ist die MCCABE-Zahl:  
$$\mu(G) = E - N + 1 = 13 - 10 + 1 = 4$$
- In der Praxis ist ein Wert von 10 eine gute Komplexitätsgrenze (Einzeln-Module, Top-down-Entwurf)

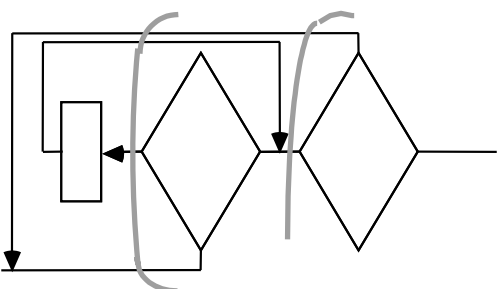


# Kumulatives minimales Schnittmengenmaß

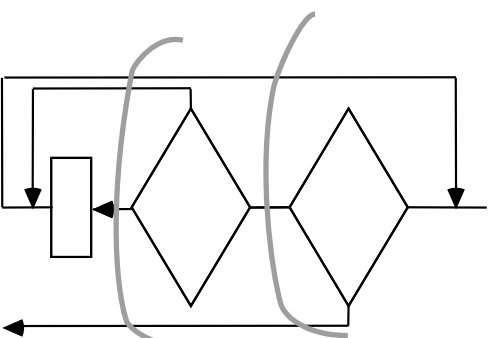
- nach HSU und MALEK
- beruht auf minimalen Graphenschnitten (minimal cut-sets)
- Komplexität ist die Summe ausgehender Kanten nach jedem Entscheidungsblock



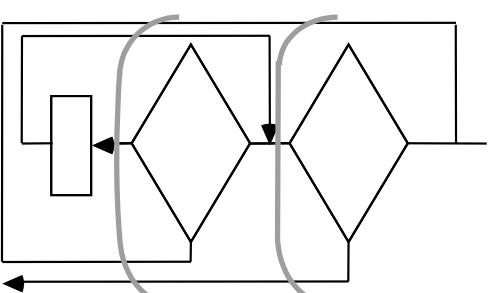
CMCS=5



CMCS=6



CMCS=7



CMCS=8

# Fehler im Designprozeß

- Mögliche Fehlerursachen im Produktionszyklus von Software
  - Unklare Problemstellung / unbekannte Bedingungen
  - Inkorrekte Spezifikation (etwas anderes spezifiziert als gemeint)
  - Inkorrekte Implementation
  - Mangelnde Information über Komponenten/Tool/Bibliotheken etc.
  - Partielle Fehlerbeseitigungsmaßnahmen
- Im Zeitalter der Quick-and-Dirty-Programmierung kommen SW-Fehler wesentlich häufiger als HW-Fehler vor (Softwarekrise)

# Spezifikation

- Spezifikation erfolgt häufig in natürlicher Sprache
- Natürliche Sprachen sind nicht eindeutig
- Spezifikation in Spezifikationssprachen
- Grundideen
  - Strukturelle Modularisierung
  - *Separation of Concerns*: Semantische Modularisierung

# Spezifikationssprachen

- Höhere Spezifikationssprachen z.B. SDL, Estrel, VHDL
- Vorteile:
  - Höhere Eindeutigkeit als natürliche Sprache
  - Prüfung der Konsistenz ist z.T. automatisch möglich
  - Evtl. automatische Generierung von Code
- Nachteil
  - Erforderliche Abstraktion bildet zusätzliche Fehlerquelle
- Mitunter erfolgt die Spezifikation in noch einfacheren Sprachen, z.B. Petri-Netzen
- Vorteile
  - im einzelnen leicht verständlich
  - wohl-definierte Semantik
- Nachteile:
  - schnell wachsende Komplexität
  - eingeschränkte Ausdrucksfähigkeit

# Umgang mit Implementationsfehlern

- Gleiche Grundidee wie bei HW: Redundanz
- Aber: Unabhängigkeit kann nicht angenommen werden
- Betrachtete Ansätze
  - *n*-Version Programming
  - Recovery Blocks
  - Checkpointing
  - Rejuvenation

# Forward und Rollback Recovery

Zwei grundsätzliche Prinzipien bei SW-Fehlertoleranz:

- *Forward Error Recovery* auch *Roll-forward*
  - “Verkraftbarer” Schaden
  - häufig in Echtzeitsystemen (alte Daten sind obsolet)
  - Performance ist wichtiger als Korrektheit
- *Backward Error Recovery* auch *Roll-back*
  - Schaden nicht beherrschbar
  - meist teurer
  - Korrektheit ist wichtiger als Performance

# NVP - N-Version Programming

- Prinzip: Diversität (wie auch in Hardware)
- Idee:  $n$  Versionen eines Programms ( $n > 1$ ), die voneinander unabhängig nach einer gemeinsamen Spezifikation entwickelt wurden, werden ausgeführt und ihre Ergebnisse verglichen
- Die einzelnen Versionen sollten von unterschiedlichen Programmierern geschrieben worden sein.
- Es können unterschiedliche Algorithmen und Sprachen verwendet werden
- Ausführung erfolgt
  - zeitlich versetzt auf einem Prozessor
  - parallel auf gleichartigen Prozessoren
  - parallel auf verschiedenartigen Prozessoren (HW+SW-Heterogenität)

# Steuerungsprogramm

- NVP benötigt ein Steuerungsprogramm (*driver program*)
- Dieses ist verantwortlich dafür,
  - die einzelnen Versionen zu starten
  - sie zu synchronisieren
  - ihre Ergebnisse einzusammeln und zu vergleichen
  - auf der Basis der Ergebnisse zu handeln
- Das Steuerungsprogramm bildet einen “single point of failure” und sollte daher so einfach wie möglich sein

# Problem des Datenvergleichs

- Die Ergebnisse der  $n$  Versionen müssen verglichen werden, entweder nur durch das Steuerungsprogramm, oder durch HW-Komponente (Voter)
- Vergleich sollte anwendungsunabhängig sein
- **aber:** häufig werden “ungefähre” Vergleiche benötigt (Rundungsfehler, algorithmische Unstabilitäten, etc.)
- Nutzung von Fuzzy-Logik oder gleitenden Bereichschecks
- Allgemein: “ungefähres” Voting ist ein schwieriges Problem

# Recovery Blocks

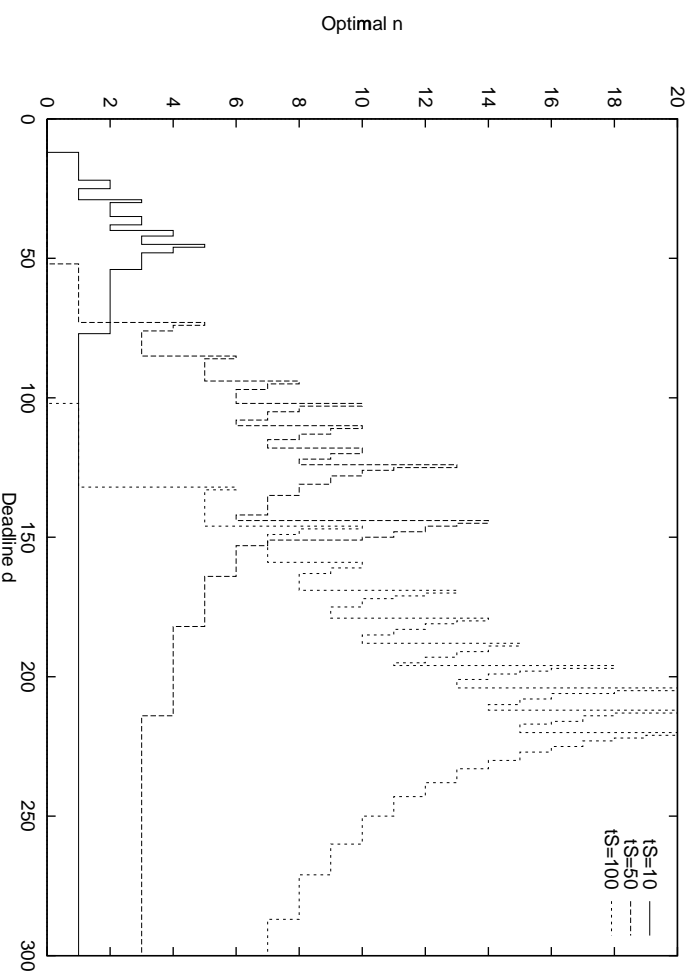
- Idee: Schaffung (einer oder mehrerer) Rückfallebenen, auch bei HW-Fehlern anwendbar
- **Primäres Modul:**
  - ‘Normale’ Software
  - nichtredundant
  - hält (hoffentlich) Spezifikation ein
  - vor Ausführung werden am *Recoverpoint* Eingangsdaten gespeichert
- **Akzeptanztest:**
  - Maßnahmen zur Fehlererkennung
  - diverse Checks sind möglich, siehe Vorlesung 9
  - Signalisiert eine Exception, wenn der Check nicht bestanden wird
- **Alternatives Modul**
  - “Vereinfachte” Software
  - wird ausgeführt, wenn primäres Modul versagt
  - unter Umständen Kaskadierung

# Checkpointing

- Idee: Sicherung von fehlerfreien Zuständen
- Nur bei transienten SW-Fehlern, nicht bei “reinen” Design-Fehlern
- Zu bestimmten Zeitpunkten wird der Zustand des Systems in einen nicht-flüchtigen Speicher gesichert
- Bei auftretenden Fehlern wird der letzte korrekte Zustand zurückgeschrieben
- Probleme:
  - Verteilter Schnappschuß
  - Feststellung von Fehlern
  - Auswahl geeigneter Checkpointing-Zeitpunkte

# Wahl des Checkpointing-Zeitpunktes

- Bei Nicht-Echtzeitsystemen: 10% Overhead sind okay
- Daumenregel:  $t_I = \sqrt{2 \cdot t_G \cdot \overline{MTTF}}$ ,  $t_G$ : Dauer der Checkpointnahme
- Bei Echtzeitsystemen: Maximierung der Wahrscheinlichkeit, daß Dienst rechtzeitig **und** korrekt ausgeführt wird.



# Software Rejuvenation

- Idee: Vorsorglicher Neustart
- Beobachtung: Wenn PC täglich neu gebootet wird, weniger Abstürze als bei dauerhaften Laufen
- Ursache: Akkumulierende Fehler (Speicherleaks, Prozeßverklemmungen, etc.)
- Nach einer vorgegebenen Laufzeit wird das System regelmäßig neu initialisiert, auch wenn kein Fehler aufgetreten ist (Zeitoverhead)
- Realisiert in der ft-lib der Bell Labs

Neustarts	ohne	monatlich	zweiwöchentlich
Downtime	7.19 h	6.83 h	6.36 h

# Softwareengineering

Allgemein gibt es beim Entwurf von verlässlichen Systemen einige Prinzipien, die sich als gut herausgestellt haben (*best practices*):

- Spezifiziere und dokumentiere Verlässlichkeitsmaßnahmen zusammenhängend
- Benutze Fehlermodelle. Stelle mögliche Fehlerauswirkungen in einem *fault dictionary* zusammen
- Folge den ACID-Prinzipien (atomicity, consistency, isolation, durability)
- Vermeide *Single Points of Failure*
- Beachte, daß die häufigsten Ursachen für Ausfälle das Versagen der Stromversorgung und Überlast sind
- Beantworte für jede HW- und SW-Komponente die “Was wäre wenn?” - Frage