Verlässliche Echtzeitsysteme

Zusammenfassung

Peter Ulbrich

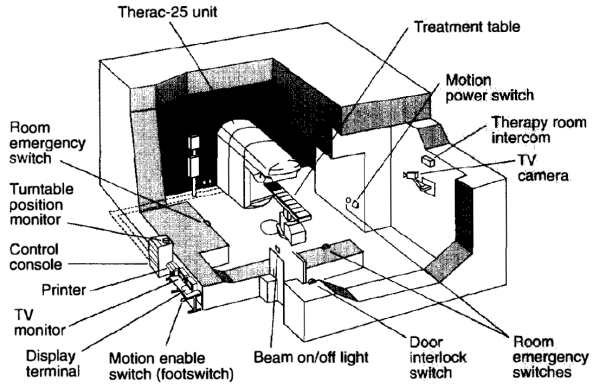
Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

07. Juli 2014

Überblick

Einleitung

- der **Fehlerfall** verlässlicher Echtzeitsystem übersteigt die Kosten des Normalsfalls um Größenordnungen ⇝ Beispiel: Therac 25



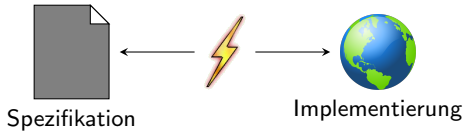
(Quelle: Nancy Leveson)

☞ **Ziel:** zuverlässiger Betrieb, minimierte Ausfallwahrscheinlichkeit

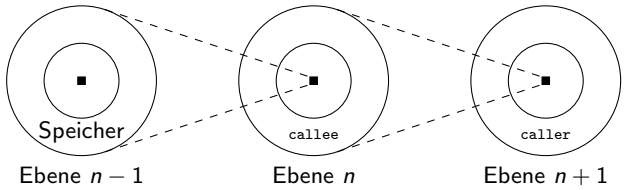
Überblick

Grundlagen

- **Fokus:** Wir kümmern uns ausschließlich um Fehler!
- Fehler bedeuten eine Abweichung von der Spezifikation



- Fehler breiten sich aus und führen zu beobachtbarem Fehlerverhalten



☞ **Ziel:** Reduktion des vom Benutzer beobachtbaren Fehlverhaltens!

Grundlagen (Forts.)

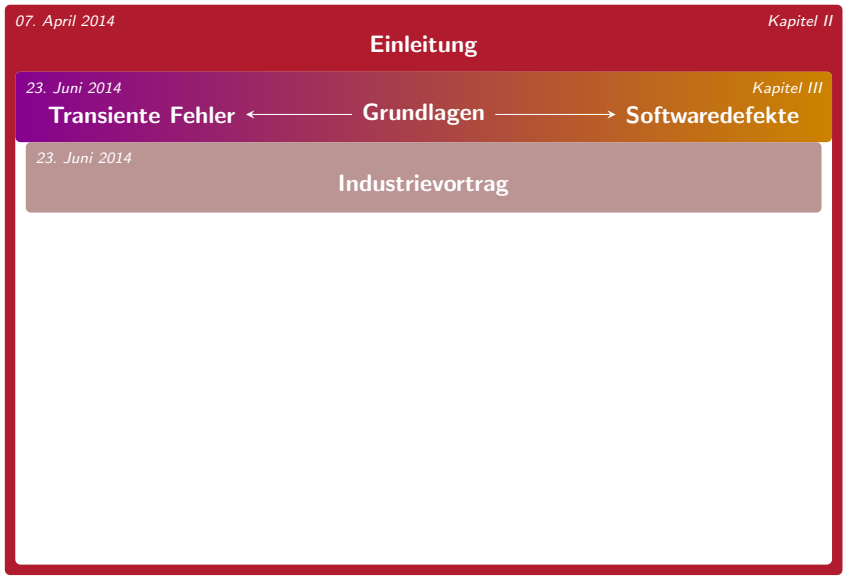
Fehler $\rightsquigarrow$ Alles dreht sich ausschließlich um Fehler!
- Fehlerfortpflanzung: fault $\rightsquigarrow$ error $\rightsquigarrow$ failure-Kette
- permanente, sporadische und transiente Fehler
- Vorbeugung, Entfernung, Vorhersage und Toleranz

Verlässlichkeitsmodelle $\rightsquigarrow$ Wie gut kann man mit Fehlern umgehen?
- Verlässlichkeit, Zuverlässigkeit, Wartbarkeit und Verfügbarkeit

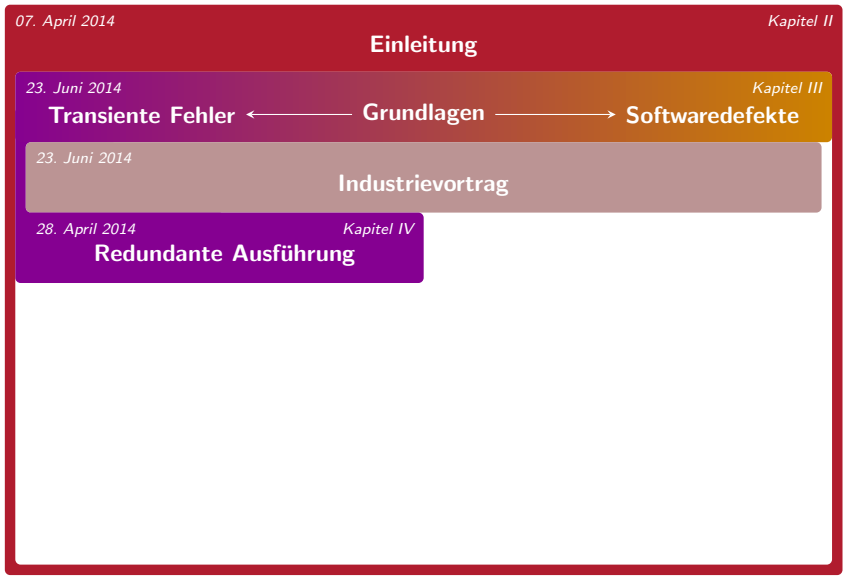
Systementwurf $\rightsquigarrow$ Bereits hier werden Fehler berücksichtigt!
- Gefahren-, Risiko- und Fehlerbaumanalyse

Software- vs. Hardwarefehler $\rightsquigarrow$ Klassifikation & Ursachen
- Softwarefehler $\mapsto$ permanente Defekte, Komplexität
- Hardwarefehler $\mapsto$ permanente & transiente Fehler, Fertigung, ionisierende Strahlung, elektromagnetische Interferenz

Überblick

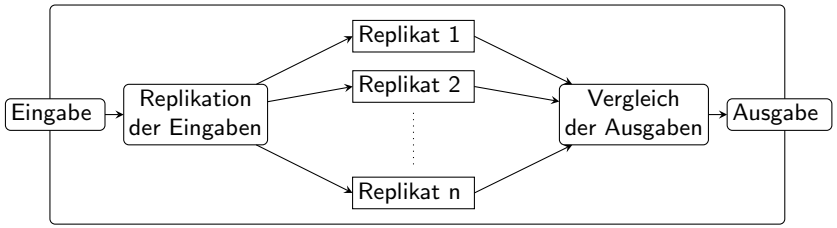


Überblick



Redundante Ausführung

- Fehlertoleranz erfordert Redundanz
  - Redundanz in der Struktur, Funktion, Information oder Zeit
- Maskierung von Fehlern durch redundante Ausführung (Replikation)
  - ein Mehrheitsentscheid kann ihre weitere Ausbreitung verhindern



- Reduktion der Kosten durch Redundanz auf Prozessebene
  - Replikation der Ausführung anstelle kompletter Knoten

  ⇝ Ausnutzung aktueller Mehrkernprozessoren

Redundante Ausführung (Forts.)

Fehlertypen ↦ Toleranz von SDCs und DUEs

Redundanz ↦ hat mehrere Dimensionen
- {hot, warm, cold} standby
- Fehlererkennung, -diagnose, -eindämmung, -maskierung

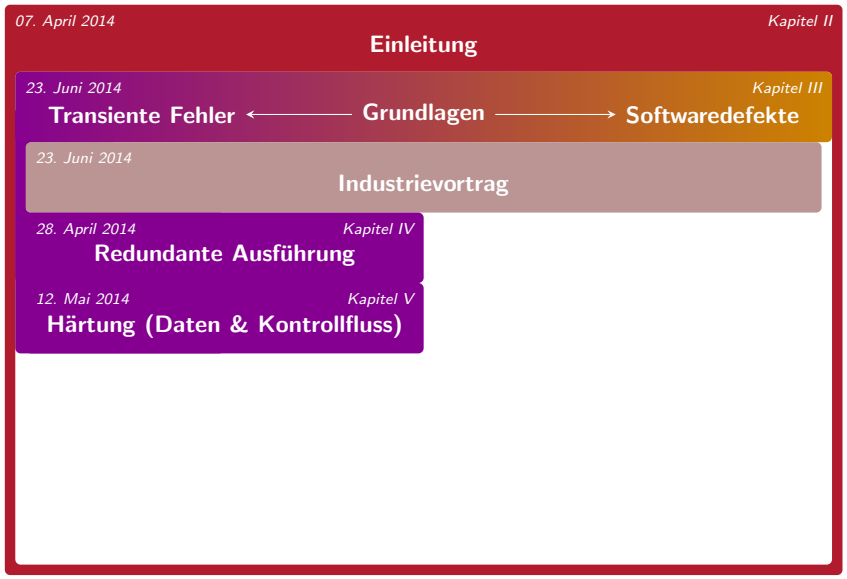
Replikation ↦ koordinierter Einsatz von struktureller Redundanz
- Replikation der Eingaben, Abstimmung der Ausgaben
- Fehlererkennung durch Relativtest
- Replikate für fail-silent, fail-consistent, malicious
- zeitliche und räumliche Isolation einzelner Replikate

Triple Modular Redundancy ↦ Hardwareredundanz
- dreifache Auslegung, toleriert Fehler im Wertbereich
- Zuverlässigkeit von Replikat und Gesamtsystem

Process Level Redundancy ↦ „TMR in Software“
- reduziert Kosten von TMR, zulasten eines geringeren Schutzes

Diversität ↦ versucht Gleichtaktfehler auszuschließen

Überblick



Härtung von Code & Daten

Fehlererkennung durch arithmetische Codierung

⇝ Einsatz von Informationsredundanz durch Prüfbits
- Fehlererkennung durch Absoluttest (auch Akzeptanztest)

AN-Codierung ⇝ Fehler im Wertbereich
- Codierung: Multiplikation mit einem konstanten Faktor A
- (nicht-)systematisch und (nicht-)separiert
- codierte Addition, Subtraktion, Multiplikation, Division
- Aussagenlogik, Schiebeoperatoren, Fließkommaarithmetik

ANBD-Codierung erweitert die AN-Codierung
- um statische Signaturen und dynamische Zeitstempel

⇝ Vollständige Fehlererfassung von Operanden-, Berechnungs- und Operatorfehlern
- Codierung des Kontrollflusses ⇝ Signaturen für Grundblöcke

CoRed-Ansatz ⇝ selektive Anwendung der ANBD-Codierung
- durchgehende arithmetische Codierung wäre zu teuer

Härtung von Code & Daten (Forts.)

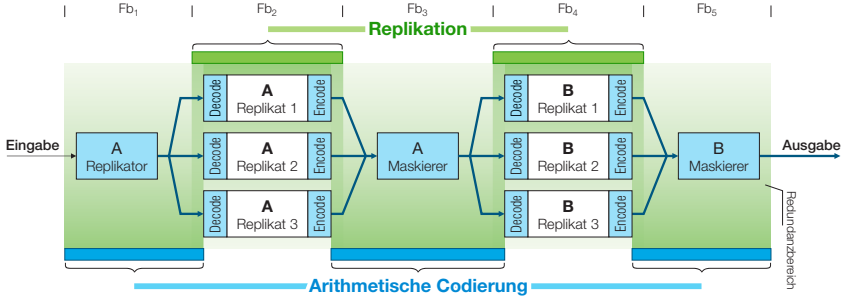
- ANBD-Codierung härtet Daten und Kontrollfluss
  - Operanden-, Berechnungs- und Operatorfehler

$$v_c = Av + B_v + D; \quad A > 1 \wedge B_v + D < A$$

  - Signatur B_v und Zeitstempel D

↝ **Nachteil:** enorme hohe Laufzeitkosten

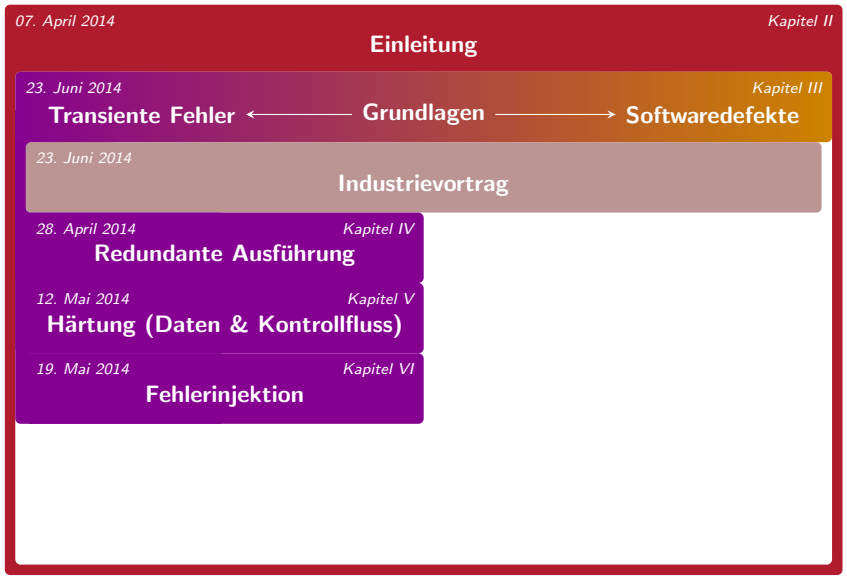
☞ „Combined Redundancy“ ↝ ANBD-Codierung selektiv anwenden



- sichert den „single point of failure“ replizierter Ausführung

↝ codierte Implementierung des Mehrheitsentscheids

Überblick



Fehlerinjektion

- Verifikation von Fehlertoleranzimplementierungen
  - durch das gezielte einbringen von Fehlern

☞ der Kreis schließt sich

- Evaluation der Fehlertoleranz ist im Produktivbetrieb nicht möglich

  - der durch Fehler verursachte Schaden ist nicht hinnehmbar
  - das Auftreten von Fehlern ist nicht deterministisch/reproudzierbar

Fehlerinjektion (Forts.)

FARM-Modell für Fehlerinjektion

- Fault, Activation, Readout, Measure
- Auswahl, Ausführung, Beobachtung, Auswertung
- Abstraktionsebenen – axiomatisch, empirisch, physikalisch
- genereller Aufbau und Ablauf von Fehlerinjektionswerkzeugen

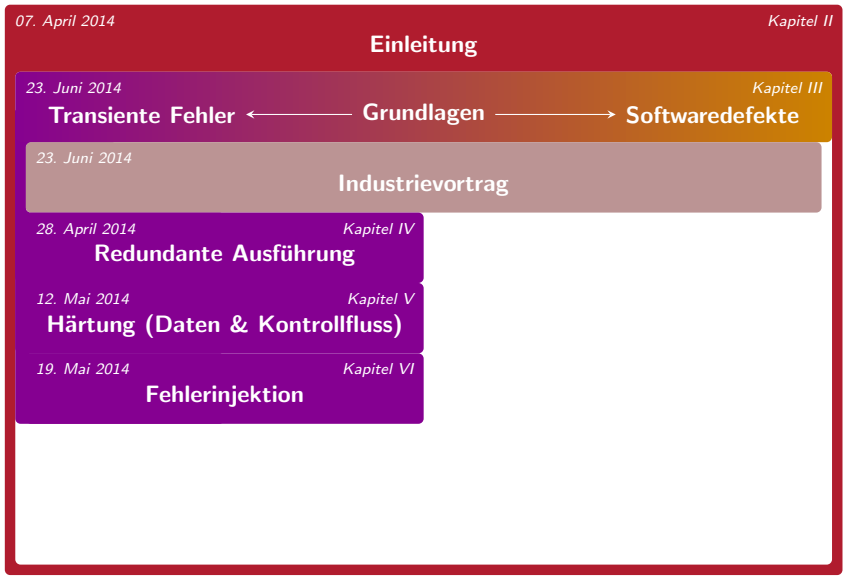
Fehlerinjektionstechniken ↦ grundlegende Kategorisierung

- {hardware, software, simulations, emulations}-basiert

FAIL* ↦ Grundlage für generische Fehlerinjektion?

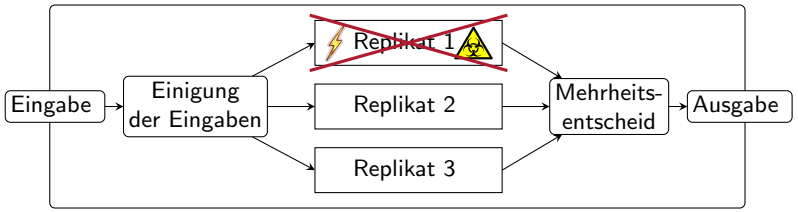
- basierend auf virtuellen Zielsystemen
- flexible Plattform für Fehlerinjektion
- schnelle Experimentdurchführung durch Parallelisierung

Überblick



Reintegration

- Ein Replikate fällt aus! ⇝ **Was dann?**



- Solange die verbliebenen Replikate korrekt arbeiten, ist alles in Ordnung.
- Was aber, wenn sie unterschiedliche Ergebnisse liefern?
  - Welches Replikat hat recht? ⇝ Patt-Situation

☞ eine „Reparatur“ ist für einen dauerhaften Betrieb unausweichlich

1. Fehlererkennung und -diagnose
2. Rekonfiguration ⇝ Isolation des fehlerhaften Knotens
3. Fehlererholung und Reintegration

Reintegration (Forts.)

Problemstellung ↦ ein Replikat fällt aus
- dies führt zu einer verminderten Fehlertoleranz

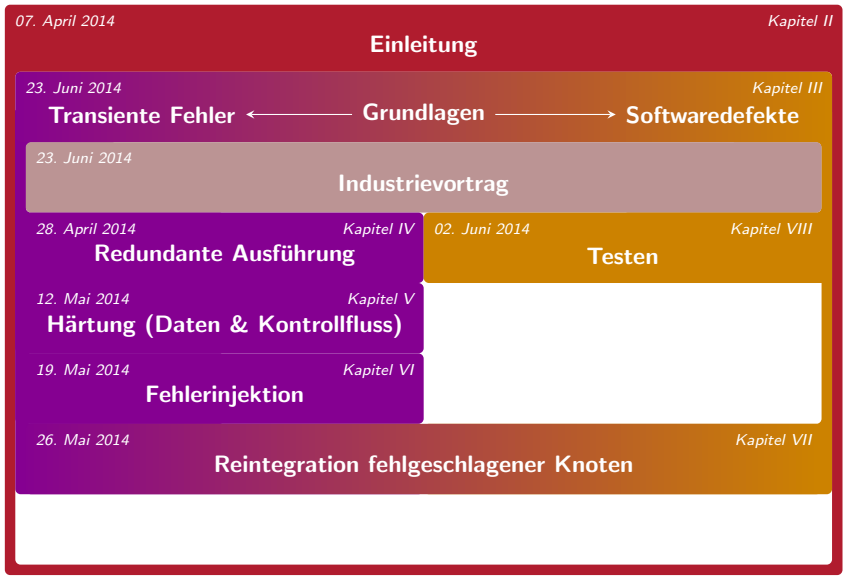
⇝ Reintegration des ausgefallene Knotens

Grundlagen für die Reintegration
- reaktiv, proaktiv und reaktiv-proaktiv
- Vorwärts- und Rückwartsbewegung
- Initialzustand und dynamischer Zustand
- Bestandteile und Minimierung des dynamischen Zustands

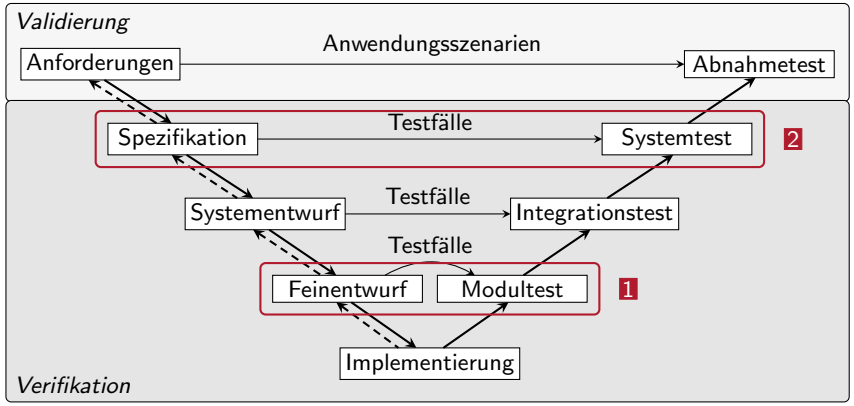
„Recovery Blocks“ Reintegration durch Rückwärtsbewegung
- „Distributed Recovery Blocks“ ↦ parallele Ausführung
- vorsorgliche Fehlererholung ⇝ Vorwärtsbewegung
  - Rückwärtsbewegung nur für die Fehlerbeseitigung

Zustandstransfer von einem funktionsfähigen Replikat
- „one-shot SR“ vs. Zustandstransfer über mehrere Schritte
- „Running SR“ vs. „Recursive SR“

Überblick



Testen



1 Modultests ⇝ Grundbegriffe und Problemstellung
⇝ Black- vs. White-Box, Testüberdeckung
2 Systemtest ⇝ Testen verteilter Echtzeitsysteme
⇝ Problemstellung und Herausforderungen

Testen (Forts.)

Testen ist **die** Verifikationstechnik in der Praxis!

- Modul-, Integrations-, System- und Abnahmetest
- ☞ kann die Absenz von Defekten aber nie garantieren

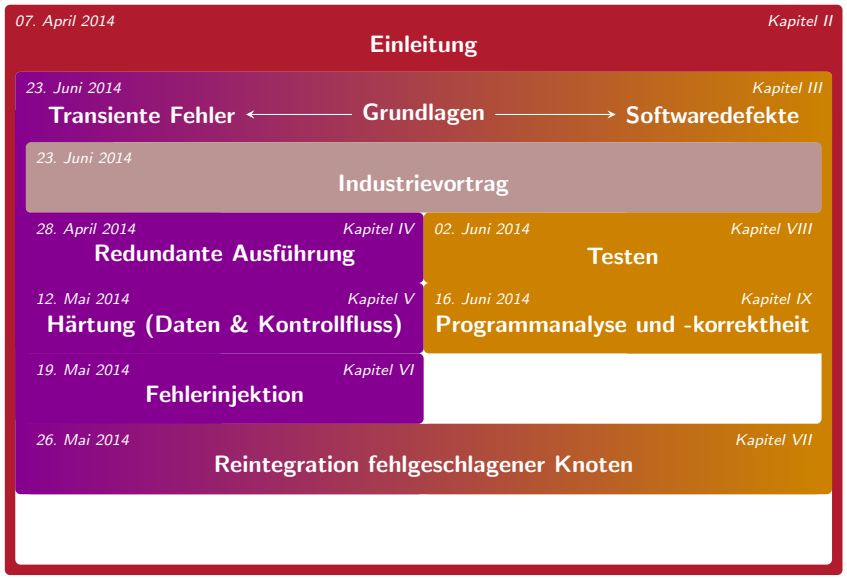
Modultests sind i. d. R. Black-Box-Tests

- Black-Box- vs. White-Box-Tests
- McCabe's Cyclomatic Complexity ⇝ Minimalzahl von Testfällen
- Kontrollflussorientierte Testüberdeckung
  - Anweisungs-, Zweig-, Pfad- und Bedinungsüberdeckung
  - Angaben zur Testüberdeckung sind immer relativ!

Systemtests für verteilte Echtzeitsysteme sind herausfordernd!

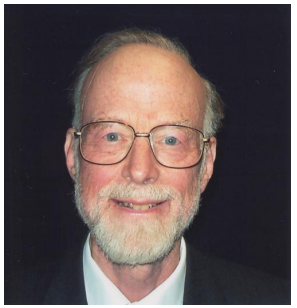
- Problemfeld: Testen verteilter Echtzeitsysteme
  - SW-Engineering, verteilte Systeme, Echtzeitsysteme
  - Probe-Effect, Beobachtbarkeit, Kontrollierbarkeit, Reproduzierbarkeit

Überblick



WP-Kalkül

- Überprüfung benutzerdefinierte Korrektheitsbedingungen
  - Angabe als Vor- und Nachbedingungen ⇝ „Design by Contract"
- Hoare-Kalkül/WP-Kalkül ⇝ denotationelle Semantik
  - schließt die Brücke zwischen Vertrag und Implementierung



C.A.R. Hoare

Edger W. Dijkstra

WP-Kalkül (Forts.)

Funktionale Programmeigenschaften $\mapsto$ Zusicherungen
- Vorbedingungen, Nachbedingungen und Invarianten
- beschrieben durch Ausdrücke der Prädikatenlogik

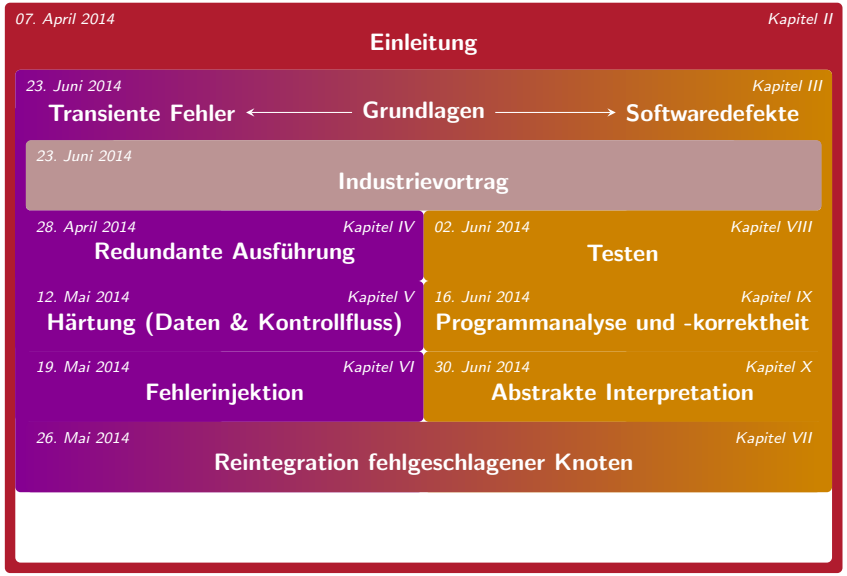
Prädikatentransformation $\rightsquigarrow$ symbolische Ausführung
- bildet Semantik durch Transformation von Zusicherungen nach
- strongest postcondition, weakest precondition

Hoare-Kalkül $\rightsquigarrow$ deduktive Ableitung von Nachbedingungen
- Hoare-Tripel, Axiome für leere Anweisungen und Zuweisungen
- Ableitungsregeln für Sequenzen, Verzweigungen und Iterationen
- Konsequenzregel passt Vor-/Nachbedingungen an

WP-Kalkül $\mapsto$ „Hoare-Kalkül rückwärts“
- wird von Frama-C in den Plug-Ins WP und Jessie implementiert

Grenzen des WP-Kalküls

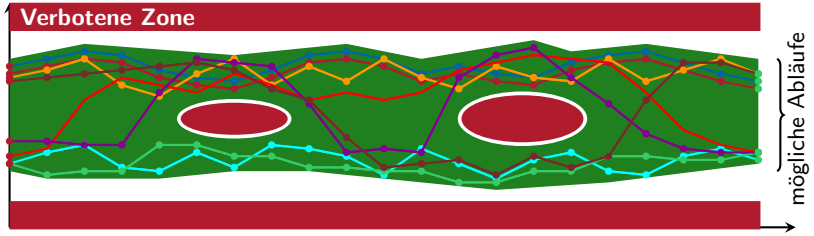
Überblick



Abstrakte Interpretation

- **Ziel:** Enthält das Programm Software-Defekte?
  - Ganzzahl- oder Fließkommaüberläufe, nicht-initialisierte Variablen, ...
  - Können wir diese Frage vor der Laufzeit beantworten?

☞ für die konkrete Programmsemantik geht das nicht
- eine sicher Abstraktion könnte für diesen Zweck aber ausreichen
  - $\rightsquigarrow$ für Zugriffe auf Felder ist nur der möglichen Wertebereich des Index wichtig
  - Welcher konkrete Wert wann angenommen wird, ist nicht von Belang.



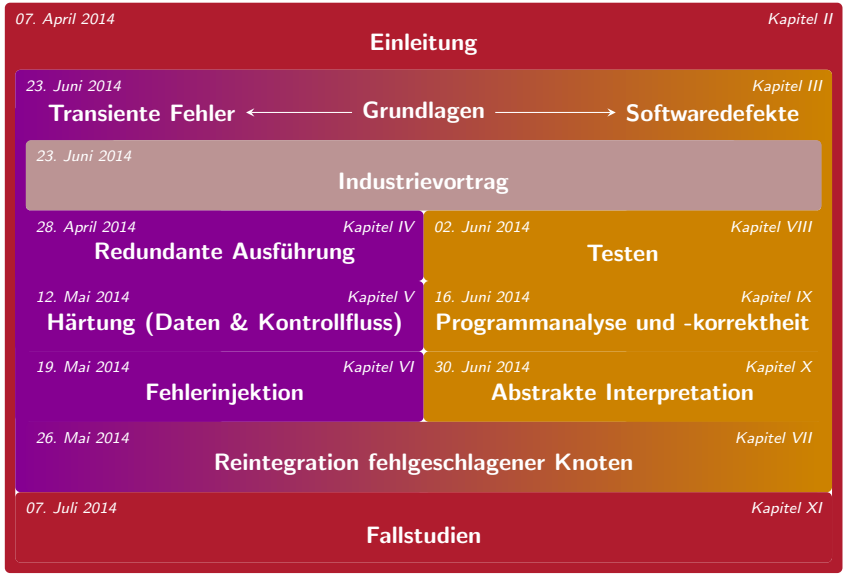
Abstrakte Interpretation (Forts.)

Konkrete Programmsemantik ist nicht berechenbar
$\rightsquigarrow$ Approximation durch eine abstrakte Semantik
- Korrektheit der Approximation ist entscheidend
  - nur so kann man einen Sicherheitsnachweis führen
- die Approximation muss präzise sein
  - nur so kann man Fehlalarme vermeiden
- die Approximation darf nicht zu komplex sein
  - nur so kann sie effizient berechnet werden

Transitionssystem beschreiben Programme
- Pfadsemantiken beschreiben die konkrete Programmsemantik
- Approximation durch Pfadpräfixe und Sammelsemantik
  - $\rightsquigarrow$ abstrakte Interpretation approximiert die Sammelsemantik

Mathematische Grundlagen abstrakter Interpretation
- (vollständig) partiell geordnete Mengen, Verbände
- Galoiseinbettungen, lokale konsistente Funktionen, Widening
- Intervallabstraktion

Überblick



Fallstudien

- Wie werden echte verlässliche Echtzeitsysteme entwickelt?
  - Wie wird die Korrektheit von Software sichergestellt?
  - Welche Laufzeitfehler sind insbesondere von Belang?
  - Welche Fehlertoleranzmechanismen werden implementiert?

☞ Betrachtung zweier Fallstudien
- primäres Reaktorschutzsystem „Sizewell B“
- digitale Flugsteuerung Airbus A320/A330/A340

Fallstudien (Forts.)

Sizewell B ⇝ primäres Reaktorschutzsystem
- einziger Zweck: sichere Abschaltung des Reaktors

Airbus ⇝ digitale „Fly-by-Wire“-Flugsteuerung
- die Lenkung moderner Verkehrsflugzeuge

Redundanz ⇝ Absicherung gegen Systemausfälle
- bis 7-fach redundante Systeme

Diversität ⇝ Abfedern von Software-Defekten
- unterschiedliche Hardware und Software

Isolation ⇝ Abschottung der einzelnen Replikate
- technisch ↦ optische Kommunikationsmedien
- zeitlich ↦ nicht-gekoppelte, eigenständige Rechner
- räumlich ↦ verschiedene Aufstellorte und Kabelrouten

Verifikation ⇝ umfangreiche statische Prüfung von Software
- vielschichtiger Prozess, Betrachtung von Quell- und Binärcode

Gliederung

{S, D, B, M}-Arbeiten . . . Doktorarbeiten

Forschungs-/Entwicklungsprojekte: Universität, Forschungseinrichtungen, Industrie

weitere Themen im Internet/UnivIS:
`http://www4.informatik.uni-erlangen.de/Theses`