

Verlässliche Echtzeitsysteme

Zusammenfassung

Fabian Scheler

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

15. Juli 2013



Überblick

15. April 2013

Kapitel II

Einleitung

© fs (FAU/INF4)

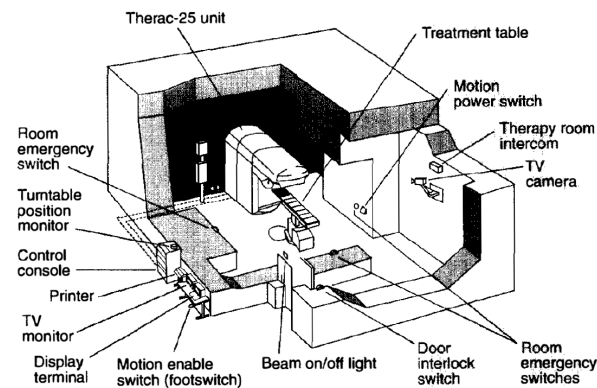
Verlässliche Echtzeitsysteme (SS 2013) – Kapitel XII Fallstudien
1 Zusammenfassung – 1.1 Einleitung

3/33



Einleitung

- Der **Fehlerfall** verlässlicher Echtzeitsystem übersteigt die Kosten des Normalfalls um Größenordnungen \leadsto Beispiel: Therac 25.



(Quelle: Nancy Leveson)

Ziel: zuverlässiger Betrieb, minimierte Ausfallwahrscheinlichkeit



© fs (FAU/INF4)

Verlässliche Echtzeitsysteme (SS 2013) – Kapitel XII Fallstudien
1 Zusammenfassung – 1.1 Einleitung

4/33

Überblick

15. April 2013

Kapitel II

Einleitung

22. April 2013

Kapitel III

Software-Defekte \leftarrow Grundlagen \rightarrow transiente Fehler

© fs (FAU/INF4)

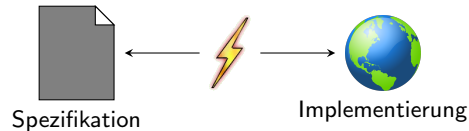
Verlässliche Echtzeitsysteme (SS 2013) – Kapitel XII Fallstudien
1 Zusammenfassung – 1.2 Grundlagen

5/33

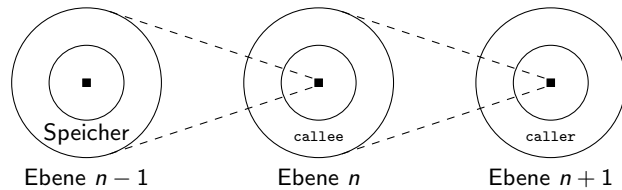


Grundlagen

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



- Fehler breiten sich aus und führen zu **beobachtbarem Fehlverhalten**



Ziel: Reduktion des vom Benutzer beobachtbaren Fehlverhaltens!

Grundlagen (Forts.)

Fehler \leadsto Alles dreht sich ausschließlich um Fehler!

- Fehlerfortpflanzung: fault \leadsto error \leadsto failure-Kette
- permanente, sporadische und transiente Fehler
- Vorbeugung, Entfernung, Vorhersage und Toleranz

Verlässlichkeitsmodelle \leadsto Wie gut kann man mit Fehlern umgehen?

- Verlässlichkeit, Zuverlässigkeit, Wartbarkeit und Verfügbarkeit

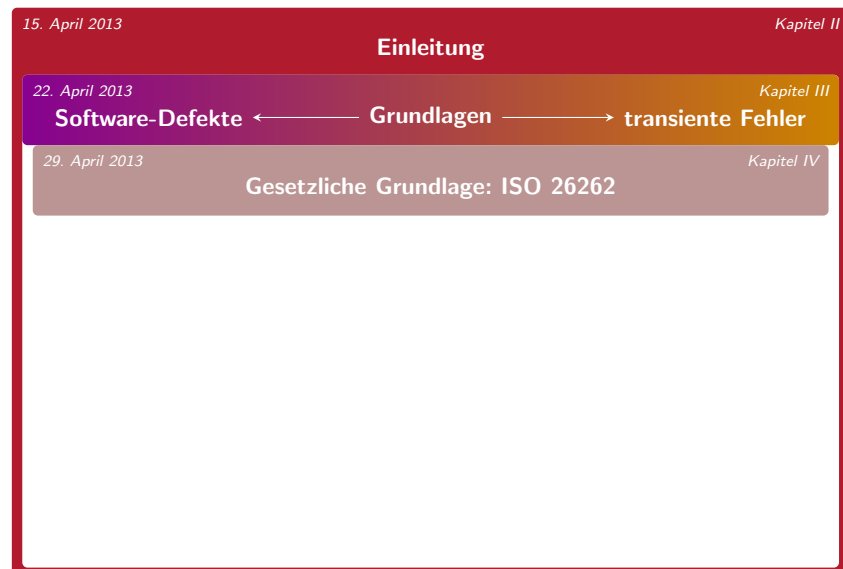
Systementwurf \leadsto Bereits hier werden Fehler berücksichtigt!

- Gefahren-, Risiko- und Fehlerbaumanalyse

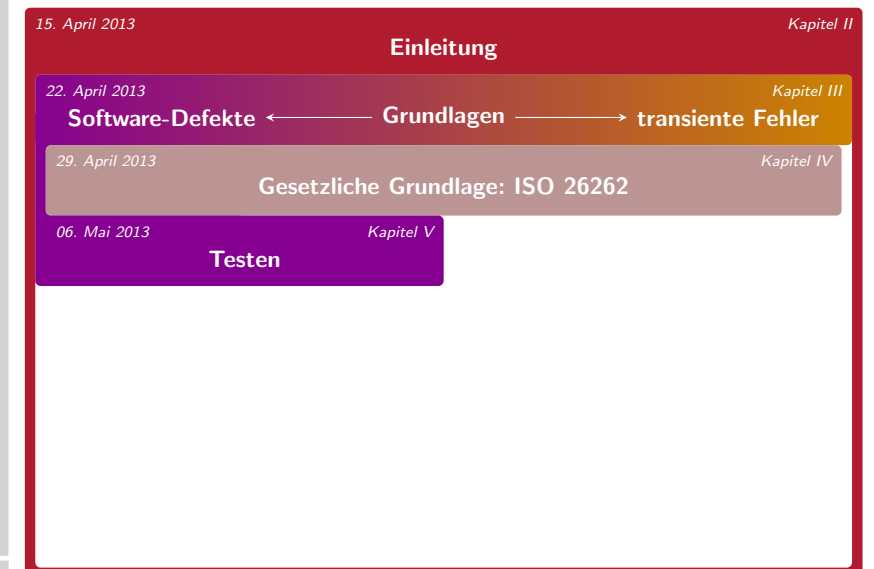
Software- vs. Hardwarefehler \leadsto Klassifikation & Ursachen

- Softwarefehler \mapsto permanente Defekte, Komplexität
- Hardwarefehler \mapsto permanente & transiente Fehler, Fertigung, ionisierende Strahlung, elektromagnetische Interferenz

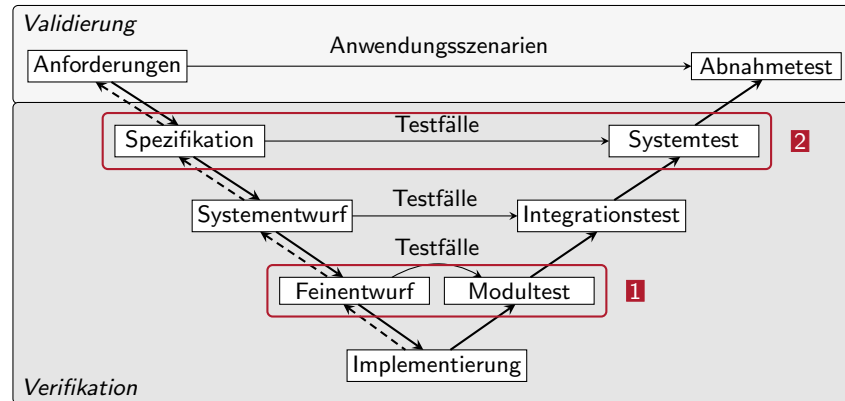
Überblick



Überblick



Testen



- 1 **Modultests** \leadsto Grundbegriffe und Problemstellung
 \leadsto Black- vs. White-Box, Testüberdeckung
- 2 **Systemtest** \leadsto Testen verteilter Echtzeitsysteme
 \leadsto 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

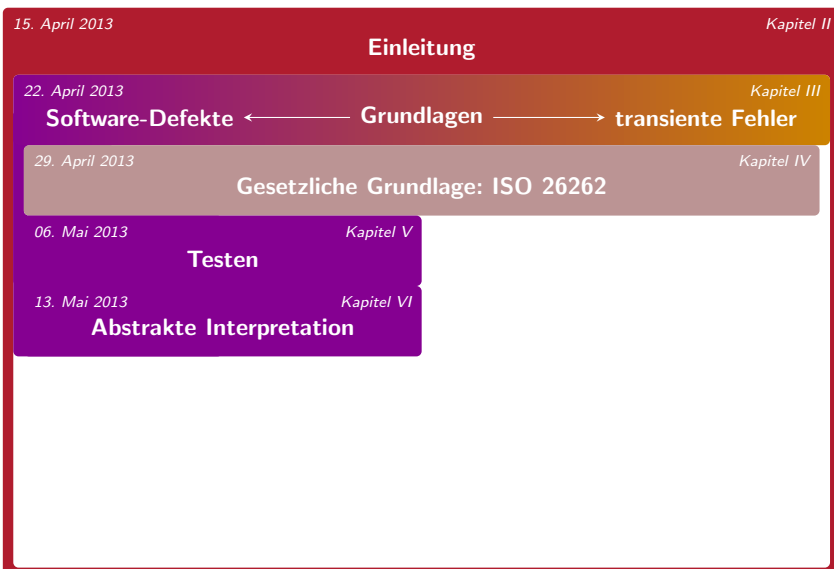
Modultests sind i. d. R. **Black-Box-Tests**

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

Systemtests für verteilte Echtzeitsysteme sind **herausfordernd**!

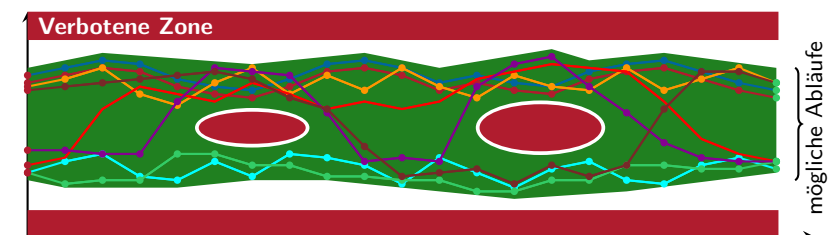
- Problemfeld: Testen verteilter Echtzeitsysteme
 - SW-Engineering, verteilte Systeme, Echtzeitsysteme
 - Probe-Effect, Beobachtbarkeit, Kontrollierbarkeit, Reproduzierbarkeit

Überblick



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 **sichere Abstraktion** könnte für diesen Zweck aber ausreichen.
 - \leadsto 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**

~ 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.

Transitionssysteme beschreiben Programme

- **Pfadsemantiken** beschreiben die konkrete Programmsemantik
- Approximation durch **Pfadpräfixe** und **Sammelsemantik**

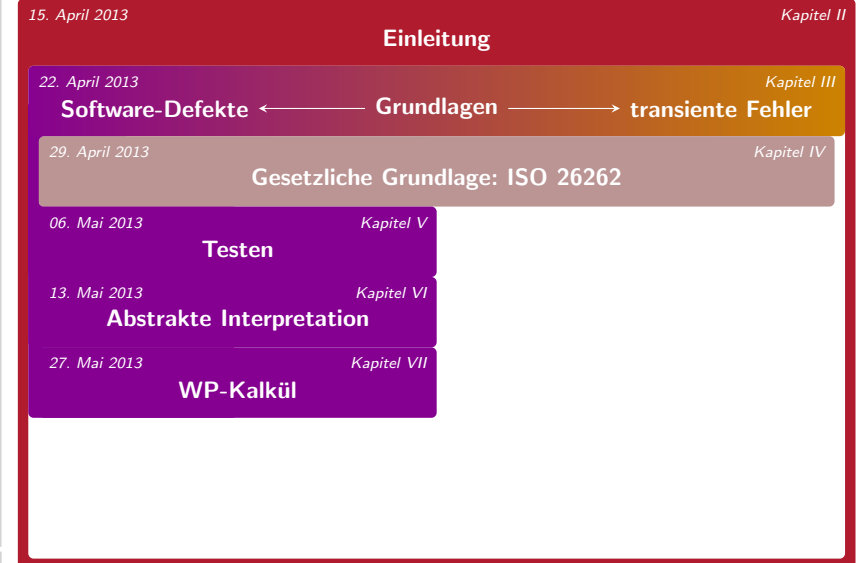
~ abstrakte Interpretation approximiert die Sammelsemantik

Mathematische Grundlagen abstrakter Interpretation

- (vollständig) **partiell geordnete Mengen**, **Verbände**
- **Galoiseinbettungen**, **lokale konsistente Funktionen**, **Widening**
- **Intervallabstraktion**

14/33

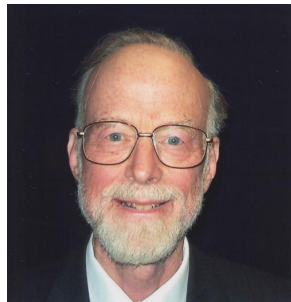
Überblick



15/33

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

16/33

WP-Kalkül (Forts.)

Funktionale Programmeigenschaften \mapsto **Zusicherungen**

- **Vorbedingungen**, **Nachbedingungen** und **Invarianten**
- beschrieben durch Ausdrücke der **Prädikatenlogik**

Prädikamentransformation ~ symbolische Ausführung

- bildet Semantik durch Transformation von Zusicherungen nach
- **strongest postcondition**, **weakest precondition**

Hoare-Kalkül ~ deduktive Ableitung von Nachbedingungen

- **Hoare-Tripel**, **Axiome für leere Anweisungen** und **Zuweisungen**
- **Ableitungsregeln für Sequenzen**, **Verzweigungen** und **Iterationen**
- **Konsequenzregel** passt Vor-/Nachbedingungen an

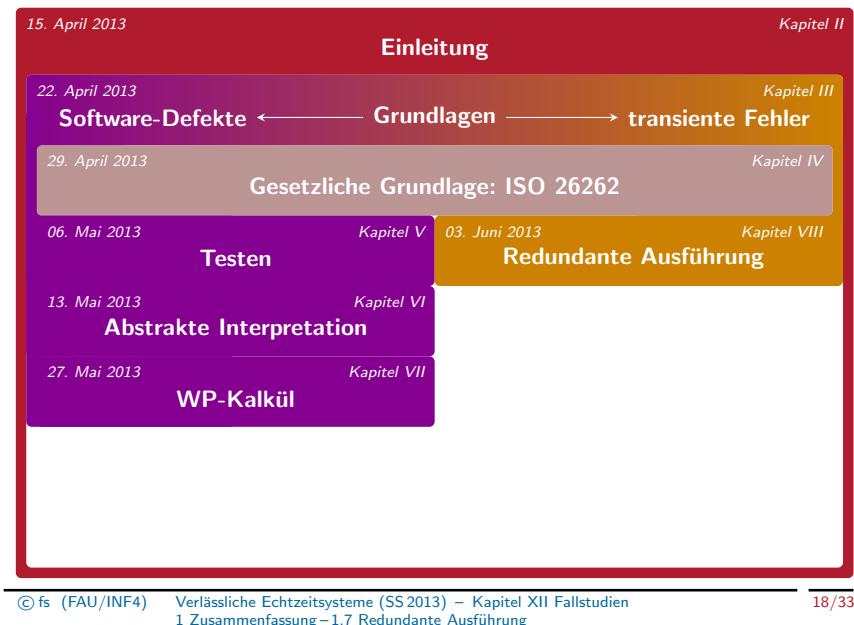
WP-Kalkül \mapsto „Hoare-Kalkül rückwärts“

- wird von Frama-C in den Plug-Ins WP und Jessie implementiert

Grenzen des WP-Kalküls

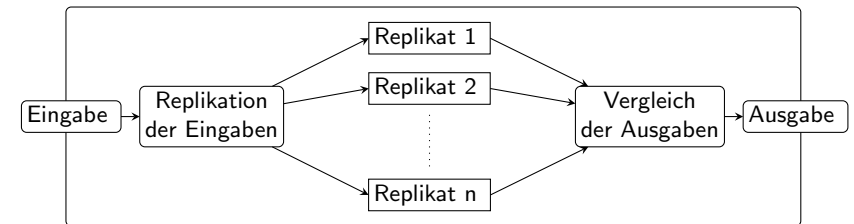
17/33

Überblick



Redundante Ausführung

- Fehlertoleranz erfordert **Redundanz**
 - räumliche, zeitliche oder funktionale Redundanz
- Maskierung von Fehlern durch **redundante Ausführung**
 - 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

- Grundvoraussetzung für Fehlertoleranz
- räumlich, zeitlich, funktional, {hot, warm, cold} standby
- Fehlererkennung, -diagnose, -eindämmung, -maskierung

Replikation → koordinierter Einsatz von Redundanz

- Replikation der **Eingaben**, Abstimmung der **Ausgaben**
- Replikate für fail-silent, fail-consistent, malicious
- zeitliche und räumliche Isolation einzelner Replikate

Triple Modular Redundancy → **Hardwareredundanz**

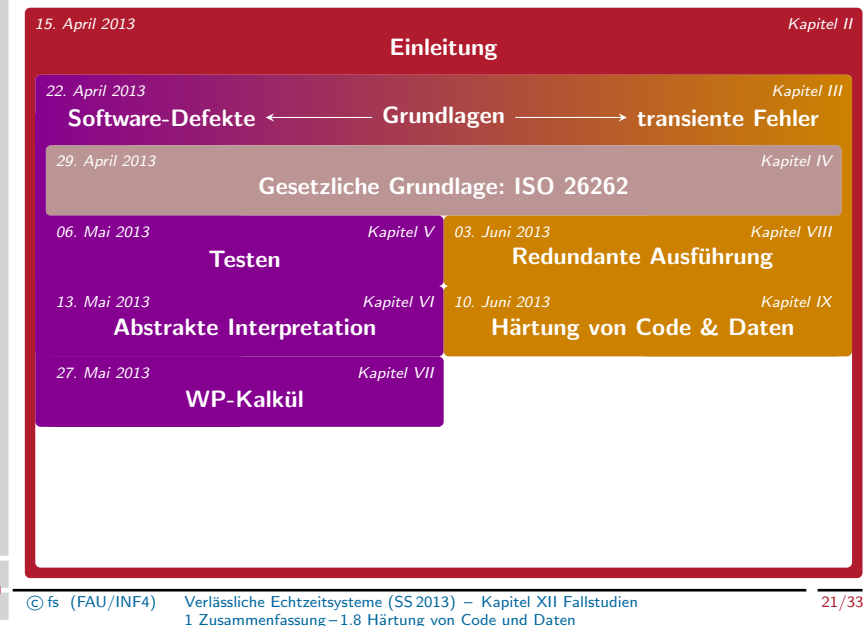
- dreifache Auslegung, toleriert **Fehler im Wertbereich**
- Zuverlässigkeit** von Replikat und Gesamtsystem

Process Level Redundancy → „TMR in Software“

- reduziert **Kosten** von TMR, zulasten eines geringeren Schutzes

Diversität → versucht **Gleichtaktfehler** auszuschließen

Überblick



Härtung von Code & Daten

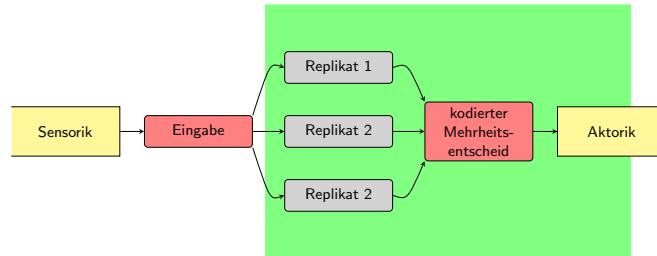
- ANBD-Kodierung härtet Daten und Kontrollfluss
 - Operanden-, Berechnungs- und Operatorfehler

$$x_c = Ax + B_x + D; \quad A > 1 \wedge B_x + D < A$$

- Signatur B_x und Zeitstempel D

↪ **Nachteil:** enorme hohe Laufzeitkosten

☞ „Combined Redundancy“ ↪ ANBD-Kodierung selektiv anwenden



- sichert den „single point of failure“ replizierter Ausführung
 - ↪ kodierte Implementierung des Mehrheitsentscheids

Härtung von Code & Daten (Forts.)

Fehlererkennung möglichst ohne redundante Ausführung

- Erkennung von Operanden-, Berechnungs- und Operatorfehlern
 - ↪ Einsatz räumlicher Redundanz durch Prüfbits

arithmetisch Kodierung

- (nicht-)systematisch und (nicht-)separiert

AN-Kodierung ↪ Fehler im Wertbereich

- Kodierung: Multiplikation mit einem konstanten Faktor A
- kodierte Addition, Subtraktion, Multiplikation, Division
- Aussagenlogik, Schiebeoperatoren, Fließkommaarithmetik

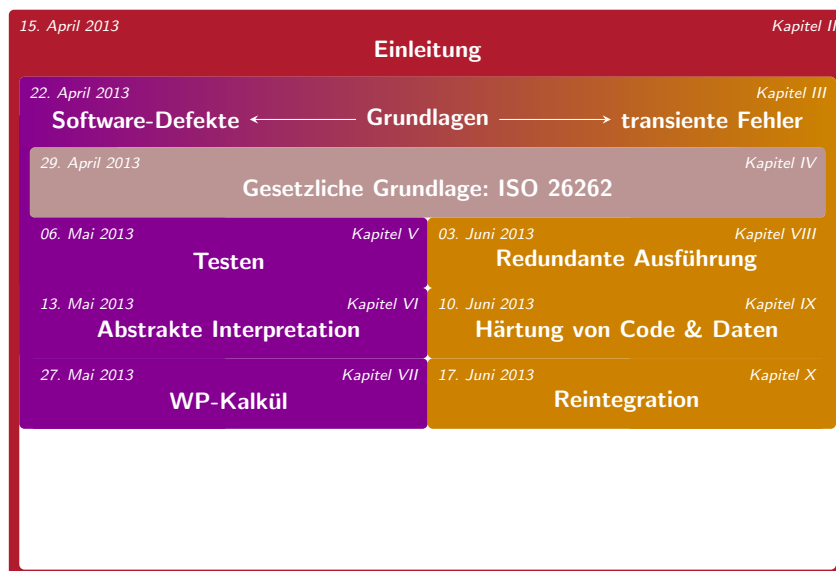
ANBD-Kodierung erweitert die AN-Kodierung

- um statische Signaturen und dynamische Zeitstempel
- Kodierung des Kontrollflusses ↪ Signaturen für Grundblöcke

CoRed-Ansatz ↪ selektive Anwendung der ANBD-Kodierung

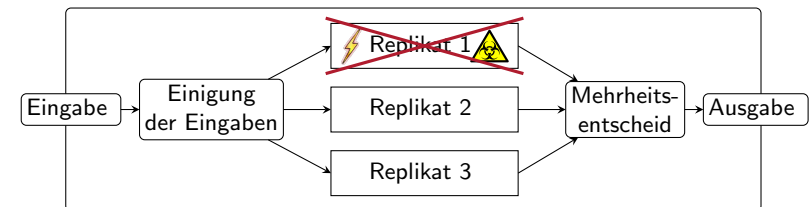
- durchgehende arithmetische Kodierung wäre zu teuer

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 Replik 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 \mapsto ein Replikat fällt aus

- dies führt zu einer **verminderten Fehlertoleranz**
 \leadsto Reintegration des ausgefallenen Knotens

Grundlagen für die Reintegration

- reaktiv, proaktiv und reaktiv-proaktiv
- Vorwärts- und Rückwärtsbewegung
- Initialzustand und dynamischer Zustand
- Bestandteile und Minimierung des dynamischen Zustands

„Recovery Blocks“ Reintegration durch Rückwärtsbewegung

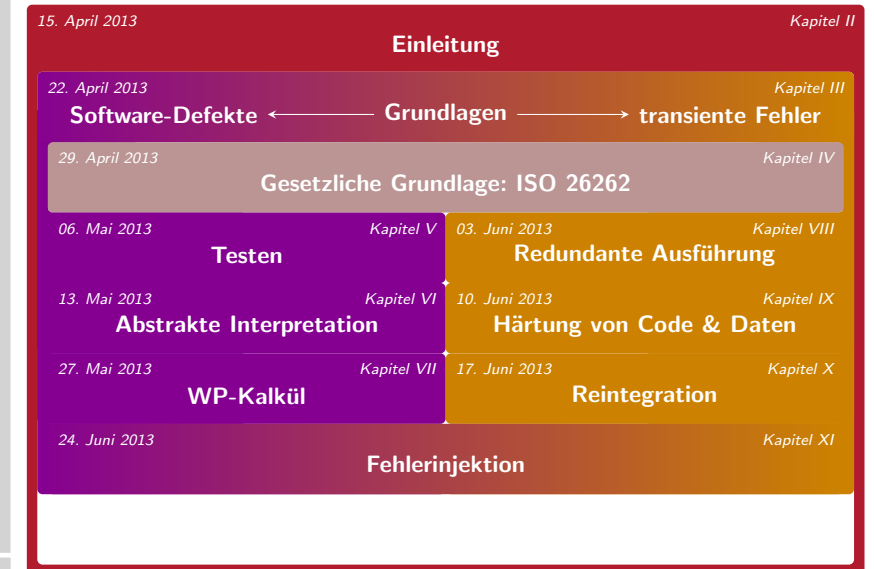
- „Distributed Recovery Blocks“ \mapsto parallele Ausführung
- vorsorgliche Fehlererholung \leadsto Vorwärtsbewegung
 - Rückwärtsbewegung nur für die Fehlerbeseitigung

Zustandstransfer von einem funktionsfähigen Replikat

- „One-shot SR“ vs. Zustandstransfer über mehrere Schritte
- „Running SR“ vs. „Recursive SR“

26/33

Überblick



27/33

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/reproduzierbar

28/33

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 \mapsto grundlegende Kategorisierung

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

Xception \mapsto ein Werkzeug für SWIFI

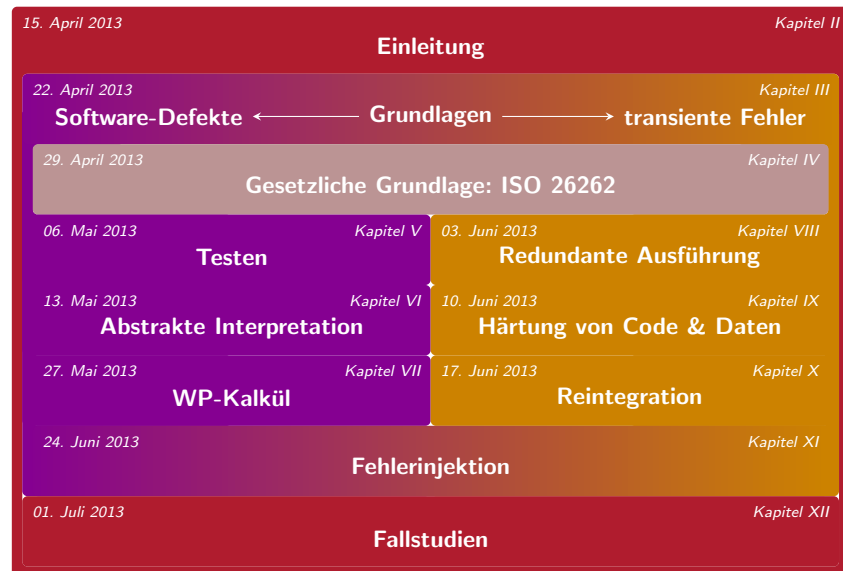
- Verwendung von prozessorspezifischen Ausnahmen
- vielseitige Fehlerinjektion (Stelle, Fehlertyp, Auslösung)

FAIL* \mapsto Grundlage für generische Fehlerinjektion?

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

29/33

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 \leadsto primäres Reaktorschutzsystem

- einziger Zweck: sichere Abschaltung des Reaktors

Airbus \leadsto digitale „Fly-by-Wire“-Flugsteuerung

- die Lenkung moderner Verkehrsflugzeuge

Redundanz \leadsto Absicherung gegen Systemausfälle

- bis 7-fach redundante Systeme

Diversität \leadsto Abfedern von Software-Defekten

- unterschiedliche Hardware und Software

Isolation \leadsto Abschottung der einzelnen Replikate

- technisch \mapsto optische Kommunikationsmedien
- zeitlich \mapsto nicht-gekoppelte, eigenständige Rechner
- räumlich \mapsto verschiedene Aufstellorte und Kabelrouten

Verifikation \leadsto umfangreiche statische Prüfung von Software

- vielschichtiger Prozess, Betrachtung von Quell- und Binärcode