

Verlässliche Echtzeitsysteme

Zusammenfassung

Fabian Scheler

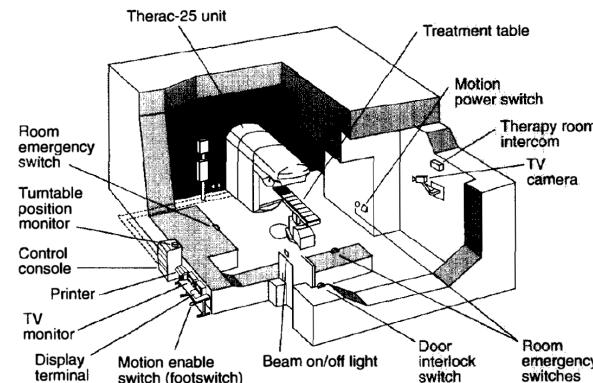
Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

15. Juli 2013



Einleitung

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



(Quelle: Nancy Leveson)

Ziel: zuverlässiger Betrieb, minimierte Ausfallwahrscheinlichkeit



Überblick

15. April 2013

Einleitung

Kapitel II

© fs (FAU/INF4)

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

3/33

Überblick

15. April 2013

Einleitung

Kapitel II

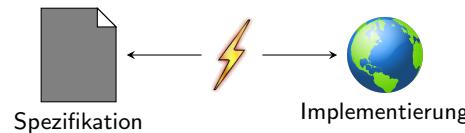
22. April 2013

Software-Defekte ← Grundlagen → transiente Fehler

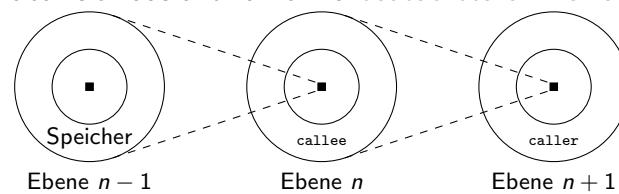
Kapitel III

Grundlagen

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



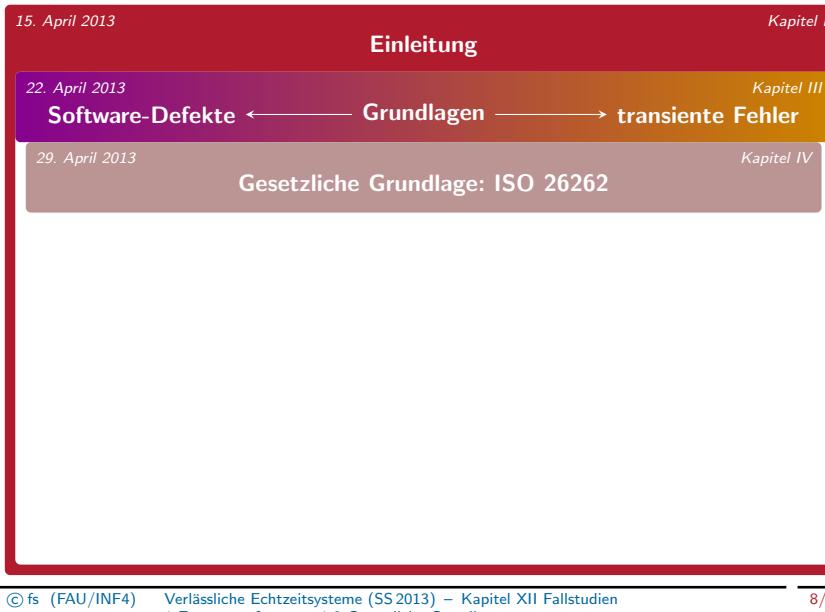
- Fehler breiten sich aus und führen zu **beobachtbarem Fehlverhalten**



Ziel: Reduktion des vom Benutzer beobachtbaren Fehlverhaltens!



Überblick



Grundlagen (Forts.)

Fehler ~ Alles dreht sich ausschließlich um Fehler!

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

Verlässlichkeitsmodelle ~ Wie gut kann man mit Fehlern umgehen?

- Verlässlichkeit, Zuverlässigkeit, Wartbarkeit und **Verfügbarkeit**

Systementwurf ~ Bereits hier werden Fehler berücksichtigt!

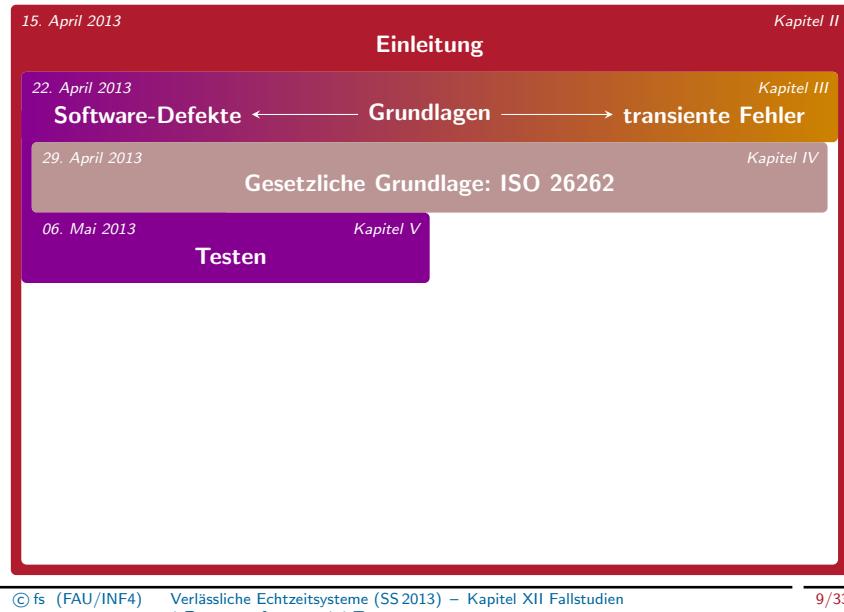
- Gefahren-, Risiko- und **Fehlerbaumanalyse**

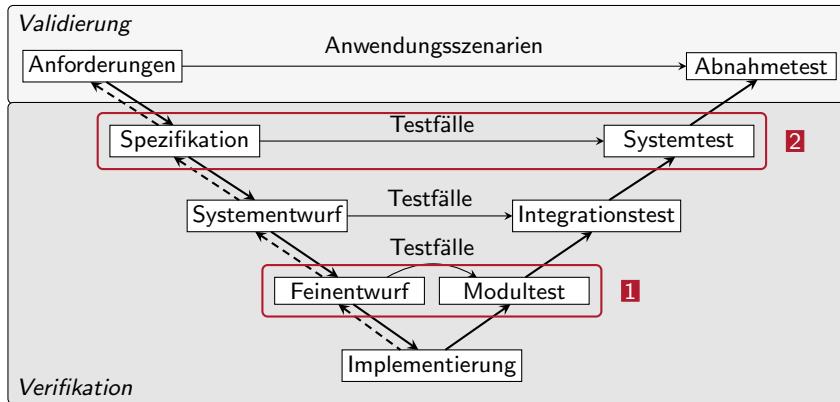
Software- vs. Hardwarefehler ~ Klassifikation & Ursachen

- **Softwarefehler** ↔ permanente Defekte, Komplexität
- **Hardwarefehler** ↔ permanente & transiente Fehler, Fertigung, ionisierende Strahlung, elektromagnetische Interferenz



Überblick





1 **Modultests** ~ Grundbegriffe und Problemstellung

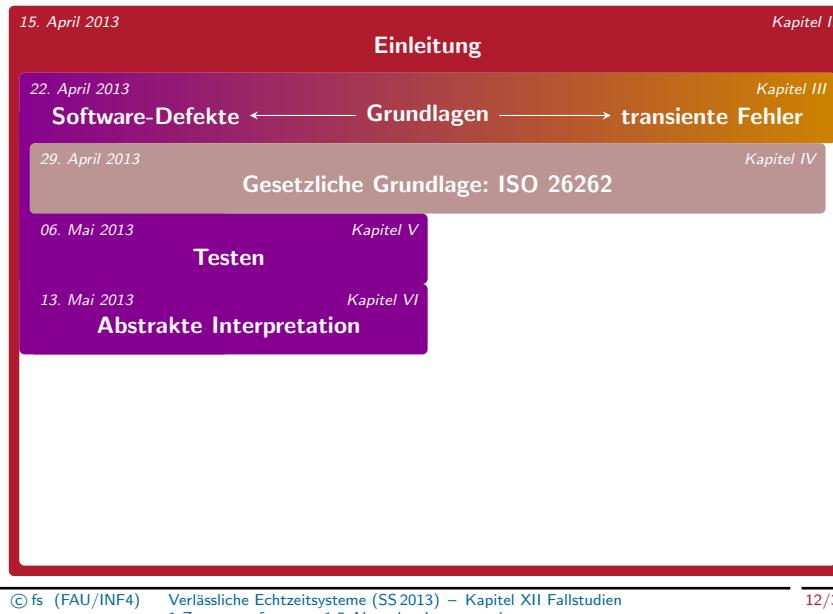
~ Black- vs. White-Box, Testüberdeckung

2 **Systemtest** ~ Testen verteilter Echtzeitsysteme

~ Problemstellung und Herausforderungen



Überblick



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 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



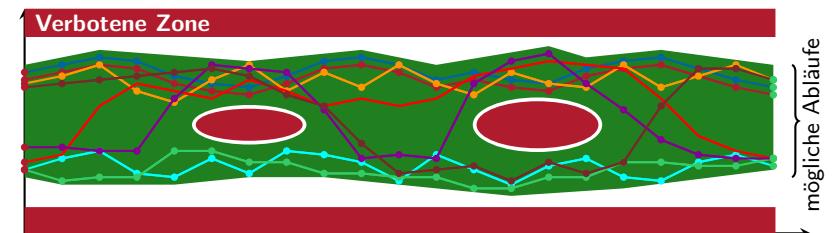
Abstrakte Interpretation

- **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.
 - ~ Für Zugriffe auf Felder ist nur der möglichen Wertebereich des Index wichtig.
 - Welcher konkrete Wert wann angenommen wird, ist nicht von Belang.



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

15. April 2013

Einleitung

Kapitel II

22. April 2013

Software-Defekte ← → **Grundlagen** → → **transiente Fehler**

Kapitel III

29. April 2013

Gesetzliche Grundlage: ISO 26262

Kapitel IV

06. Mai 2013

Testen

Kapitel V

13. Mai 2013

Abstrakte Interpretation

Kapitel VI

27. Mai 2013

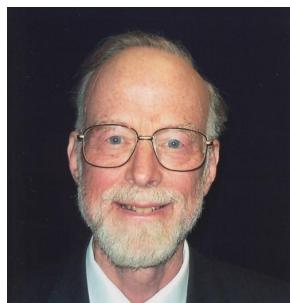
WP-Kalkül

Kapitel VII



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 → Zusicherungen

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

Prädikatentransformation ~ 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

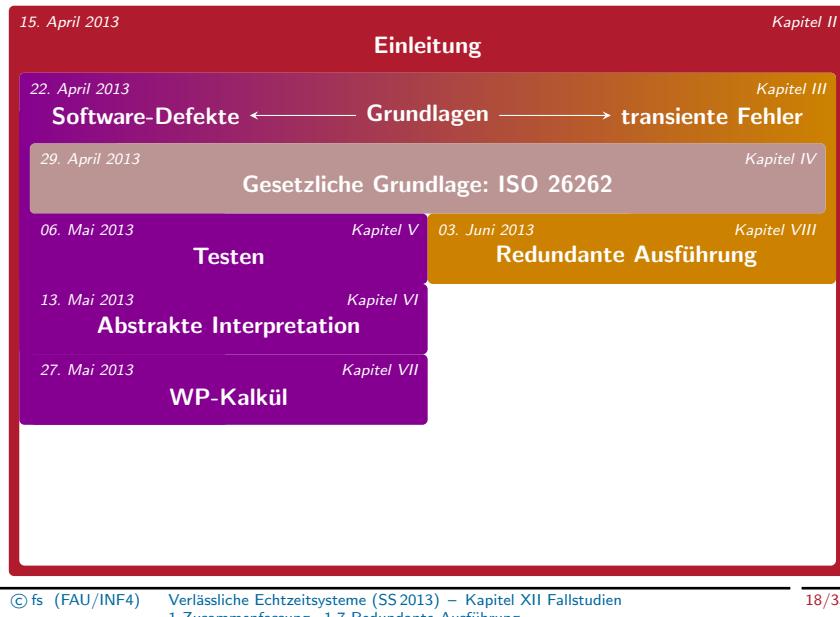
WP-Kalkül → „**Hoare-Kalkül rückwärts**“

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

Grenzen des WP-Kalküls



Überblick



Redundante Ausführung (Forts.)

Fehlerarten → Toleranz von SDCs und DUEs

Redundanz \mapsto hat mehrere Dimensionen

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

Replikation \mapsto 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 \leftrightarrow Hardware-redundanz

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

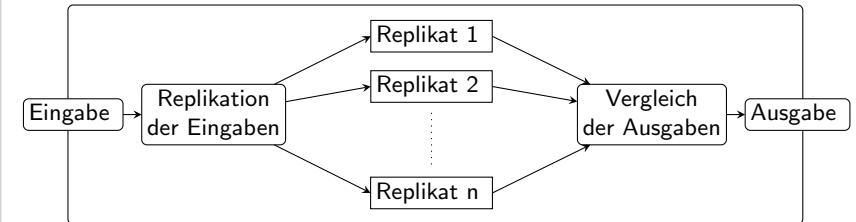
Process Level Redundancy \leftrightarrow TMR in Software“

- reduziert Kosten von TMB zulasten eines geringeren Schutzes

Diversität → versucht **Cleichtaktfehler** auszuschließen

Redundante Ausführung

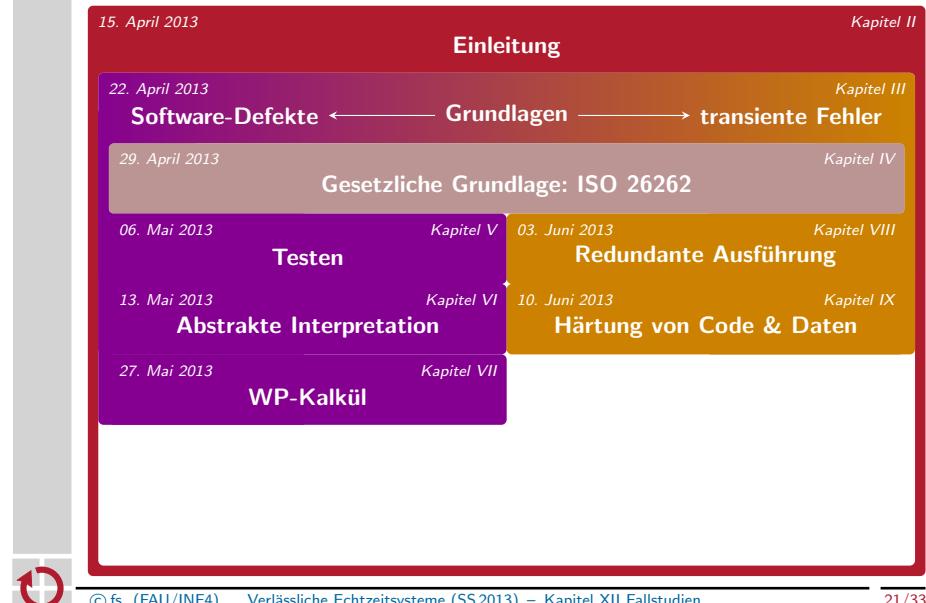
- Fehlertoleranz erfordert **Redundanz**
 - räumliche, zeitliche oder funktionale Redundanz
 - Maskierung von Fehlern durch **redundante Ausführung**
 - ein **Mehrheitsentscheid** kann ihre weitere Ausbreitung verhindern



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



Überblick



Härtung von Code & Daten

- ANBD-Kodierung härtet **Daten** und **Kontrollfluss**

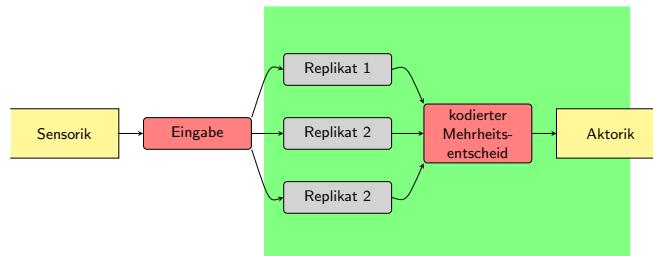
- Operanden-, Berechnungs- und Operatorfehler

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

- Signatur B_x und Zeitstempel D

~ **Nachteil:** enorme hohe Laufzeitkosten

☞ „Combined Redundancy“ ~ ANBD-Kodierung selektiv anwenden

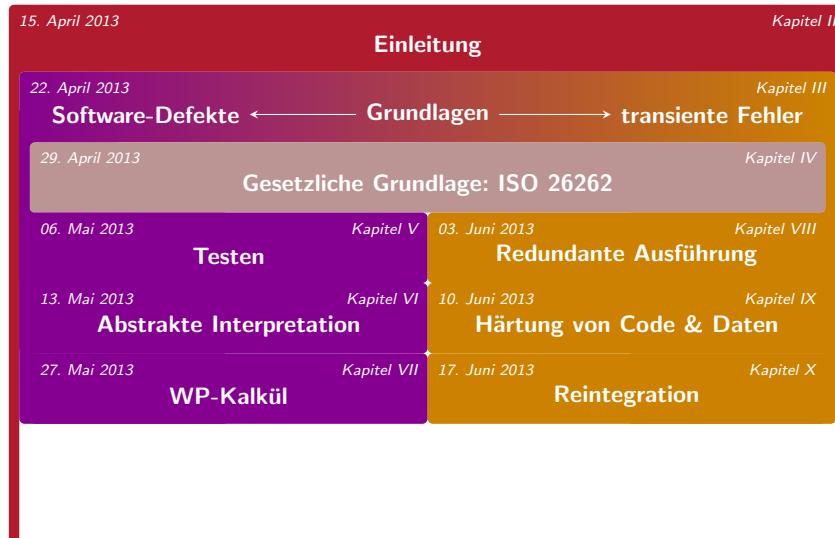


- sichert den „single point of failure“ replizierter Ausführung

~ kodierte Implementierung des Mehrheitsentscheids



Überblick



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**

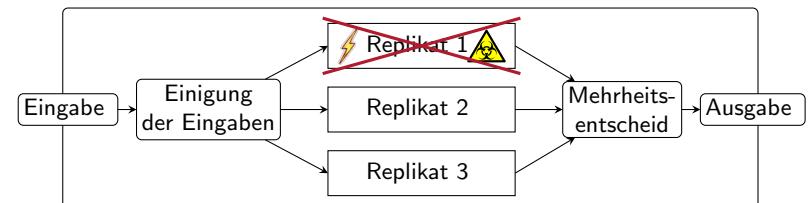
CoRed-Ansatz ~ selektive Anwendung der ANBD-Kodierung

- **durchgehende arithmetische Kodierung** wäre zu teuer



Reintegration

- Ein Replikat 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



Problemstellung → ein Replikat fällt aus

- dies führt zu einer **verminderten Fehlertoleranz**
- ~ Reintegration des ausgefallene Knotens

Grundlagen für die Reintegration

- **reakтив, proaktiv** und **reakтив-proaktiv**
- Vorwärts- und Rückwärtsbewegung
- 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**“



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



15. April 2013

Einleitung

Kapitel II

22. April 2013

Software-Defekte ←———— Grundlagen —————→ **transiente Fehler**

Kapitel III

29. April 2013

Gesetzliche Grundlage: ISO 26262

Kapitel IV

06. Mai 2013

Testen

03. Juni 2013

Redundante Ausführung

Kapitel VIII

13. Mai 2013

Abstrakte Interpretation

10. Juni 2013

Härtung von Code & Daten

Kapitel IX

27. Mai 2013

WP-Kalkül

17. Juni 2013

Reintegration

Kapitel X

24. Juni 2013

Fehlerinjektion

Kapitel XI



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

Xception → ein Werkzeug für SWIFI

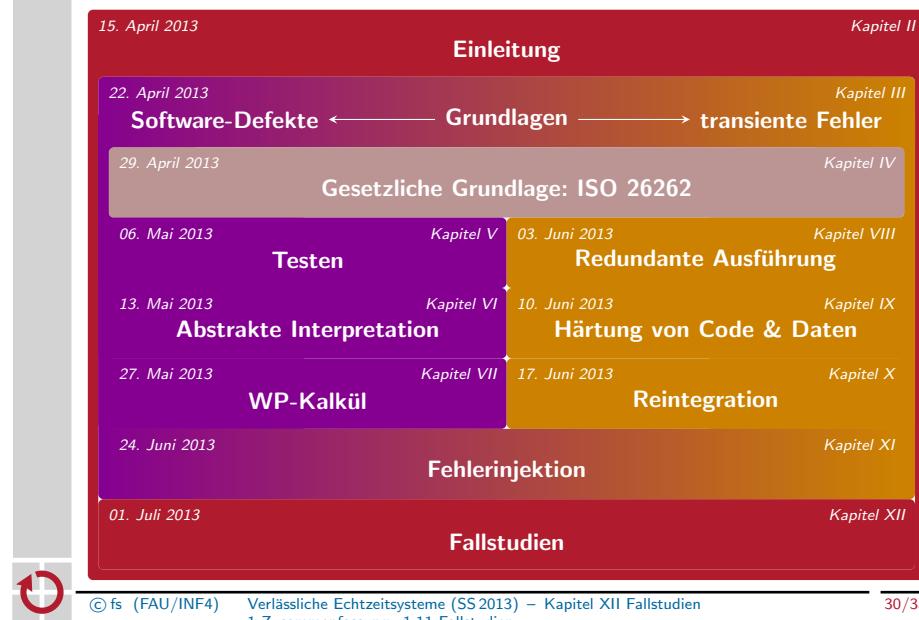
- Verwendung von **prozessorspezifischen Ausnahmen**
- **vielseitige Fehlerinjektion** (Stelle, Fehlertyp, Auslösung)

FAIL* → Grundlage für **generische Fehlerinjektion?**

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



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 Binärkode

