

Verlässliche Echtzeitsysteme

Zusammenfassung

Peter Ulbrich

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

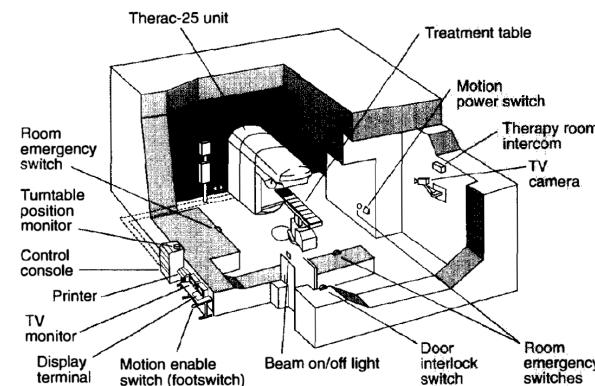
14. Juli 2015



Einleitung



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



(Quelle: Nancy Leveson)



Ziel: zuverlässiger Betrieb, minimierte Ausfallwahrscheinlichkeit



Überblick

14. April 2015

Einleitung

Kapitel II

© fs, pu (FAU/INF4)

Verlässliche Echtzeitsysteme (SS 2015) – Kapitel XIII Zusammenfassung
1 Zusammenfassung – 1.1 Einleitung

3/37

Überblick

14. April 2015

Einleitung

Kapitel II

21. April 2015

Softwaredefekte

Kapitel III

Grundlagen

←

→

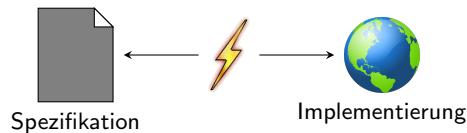
Fehlertoleranz

© fs, pu (FAU/INF4)

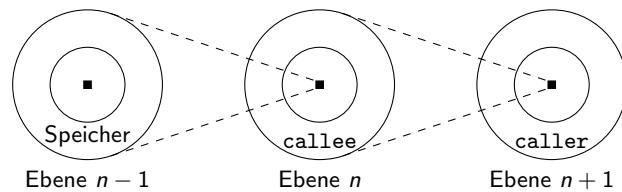
Verlässliche Echtzeitsysteme (SS 2015) – Kapitel XIII Zusammenfassung
1 Zusammenfassung – 1.2 Grundlagen

5/37

- **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!**



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!

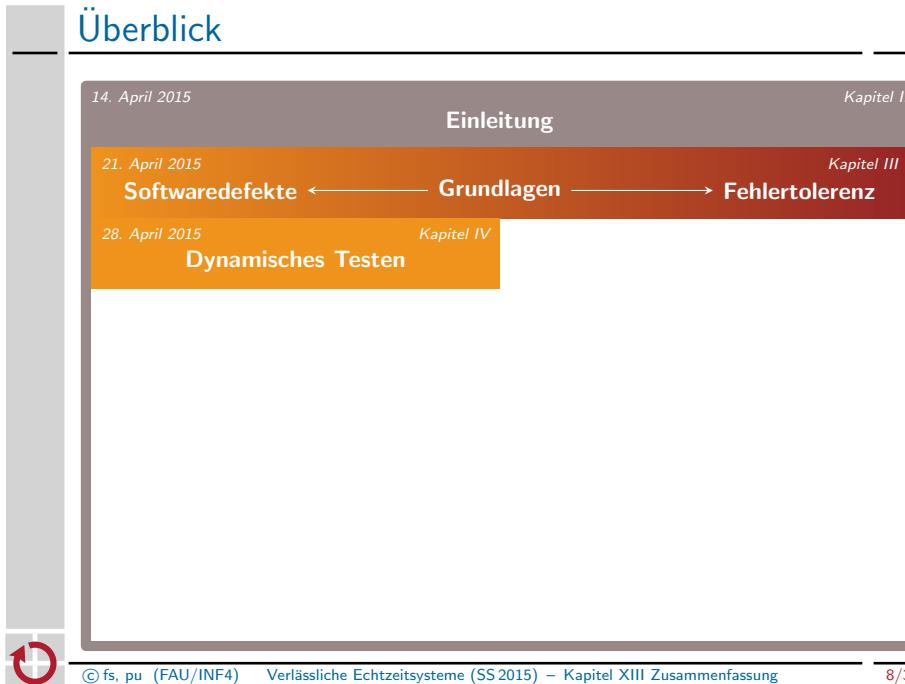
- **Gefahren-, Risiko- und Fehlerbaumanalyse**

Software- vs. Hardwarefehler \leadsto Klassifikation & Ursachen

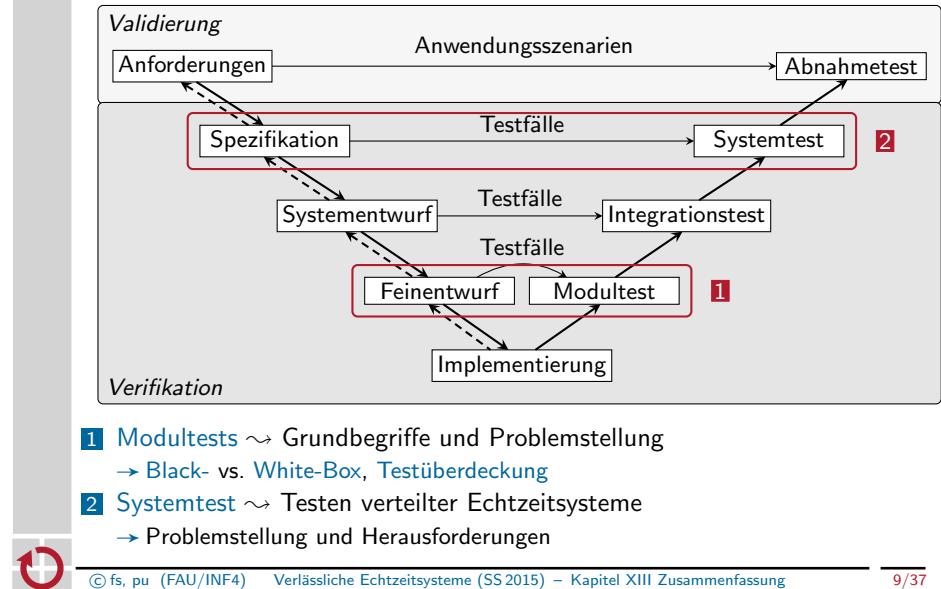
- **Softwarefehler** \leftrightarrow permanente Defekte, Komplexität
- **Hardwarefehler** \leftrightarrow permanente & transiente Fehler, Fertigung, ionisierende Strahlung, elektromagnetische Interferenz



Überblick



Testen



- ⚠ 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 \sim 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



14. April 2015

Einleitung

Kapitel II

21. April 2015

Softwaredefekte ← Grundlagen → Fehlertoleranz

Kapitel III

28. April 2015

Dynamisches Testen

Kapitel IV

05. Mai 2015

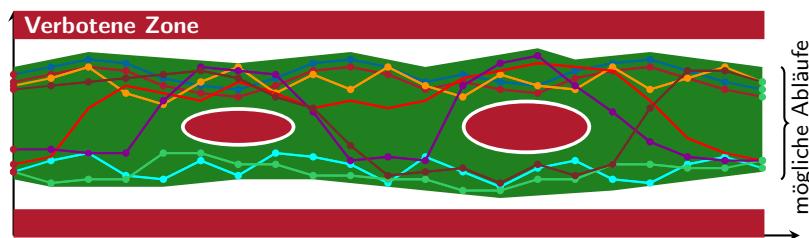
Statische Programmanalyse

Kapitel V



Abstrakte Interpretation

- Enthält das Programm **Laufzeitfehler**?
 - 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
 - Für Zugriffe auf Felder ist nur der möglichen Wertebereich des Index wichtig
 - Welcher konkrete Wert wann angenommen wird, ist nicht von Belang.
- ☞ Einsatz einer **abstrakten Programmsemantik**

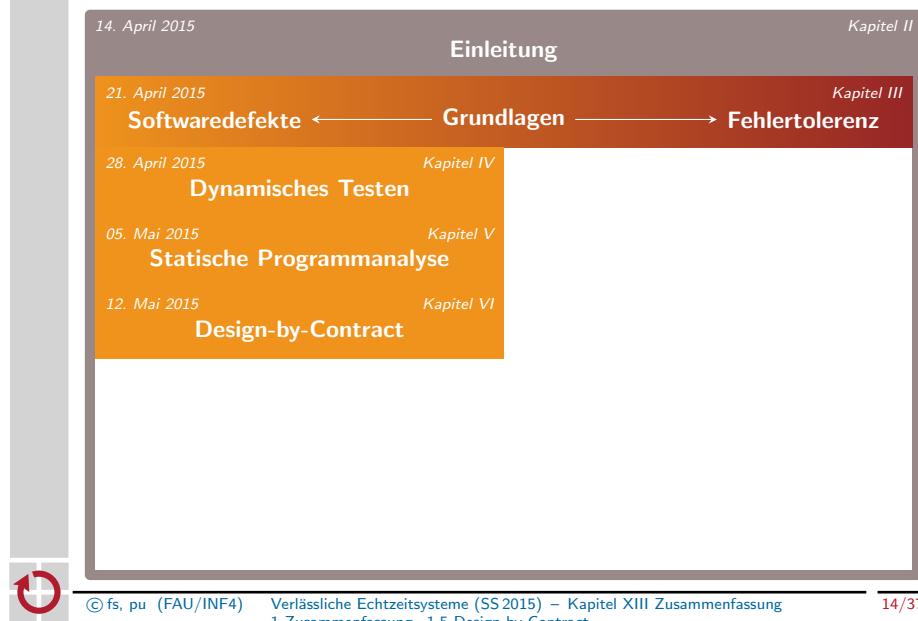


Abstrakte Interpretation (Forts.)

- Die **abstrakte Semantik** stellt eine Approximation dar
 - **Korrektheit** (Vollständigkeit) ist entscheidend
 - Nur so kann man einen **Sicherheitsnachweis** führen
 - Die Approximation muss **präzise** sein
 - Nur so kann man **Fehlalarme** vermeiden
 - Gleichzeitig eine **geringe Komplexität** aufweisen
 - Nur so kann sie **efficient berechnet** werden
- Abstraktion und Konkretisierung implizieren keinen Präzisionsverlust!
- Beschreibung durch ein **Transitionssystem**
 - **Pfadsemantiken** beschreiben die konkrete Programmsemantik
 - Approximation durch **Pfadpräfixe** und **Sammelsemantik**
- Mathematische Grundlagen der abstrakten Interpretation
 - **Formalisierung** durch **Galoiseinbettungen**, geordnete Mengen & Verbände
 - **Optimierung** durch Widening-Operator
 - Kein Bestandteil der Prüfung



Überblick



Design-by-Contract (Forts.)

Funktionale Programmeigenschaften \leftrightarrow 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 \leadsto 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“

Praxisbezug → Astreeé implementiert dieses Konzept nur teilweise!

Design-by-Contract

- Überprüfung benutzerdefinierte Korrektheitsbedingungen
 - Angabe als Vor- und Nachbedingungen \leadsto „Design by Contract“
 - Hoare-Kalkül/WP-Kalkül \leadsto denotationelle Semantik
 - Schließt die Brücke zwischen Vertrag und Implementierung



C.A.R. Hoare



Edger W. Dijkstra

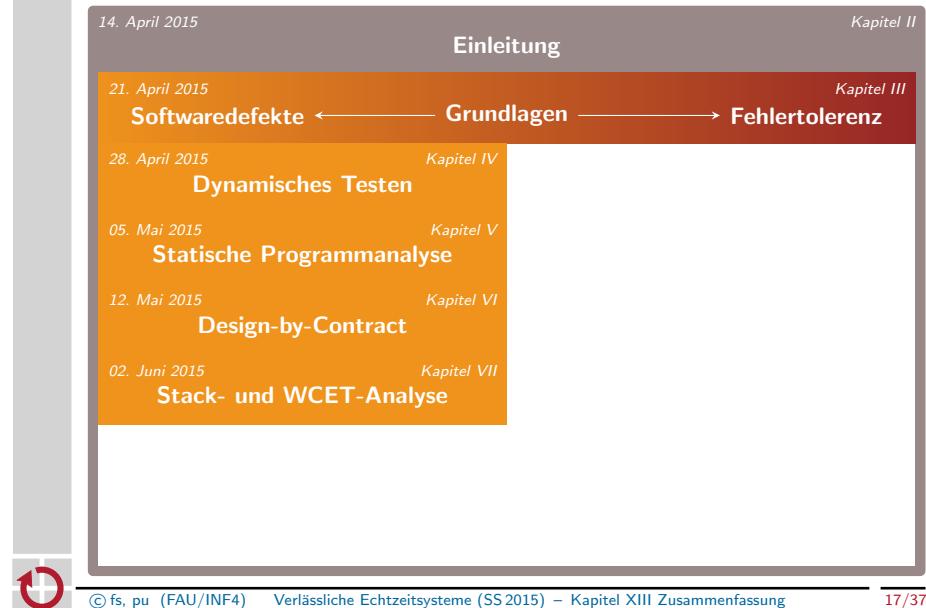


© fs_pu (FAU/INE)

4) Verlässliche Echtzeitsysteme (SS 2015) – Kapitel XIII Zusammenfassung
1 Zusammenfassung – 1.5 Design-by-Contract

15/37

Überblick



Der Stapelspeicher (Stack)

In eingebetteten Systemen typischerweise die einzige Form dynamischen Speichers

- Überabschätzung führt zu unnötigen Kosten

- ⚠ Unterabschätzung des Speicherverbrauchs führt zu Stapelüberlauf

- Schwerwiegendes und komplexes Fehlermuster
- Undefiniertes Verhalten, Datenfehler oder Programmabsturz
- Schwer zu finden, reproduzieren und beheben!

- ➡ Messbasierter Ansatz (Die Praxis!!)

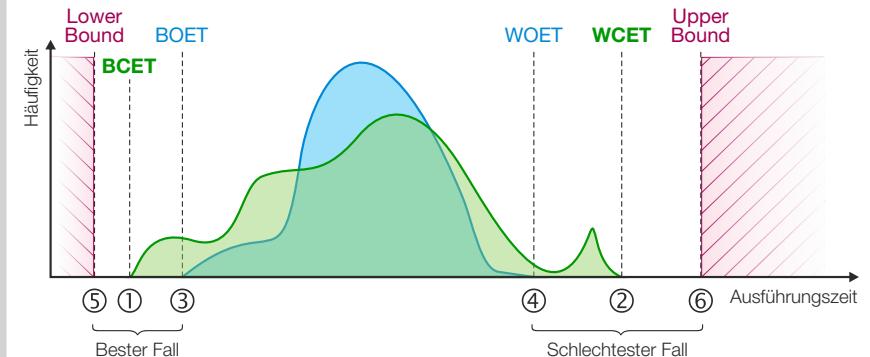
- Water-Marking, Überwachung zur Laufzeit
- Reaktiv → Keine Aussagen zum maximalen Verbrauch

- ➡ Statische Programmanalyse

- Pufferüberlauf als weitere Form von Laufzeitfehler
- Bestimmt obere Schranke für den Speicherverbrauch



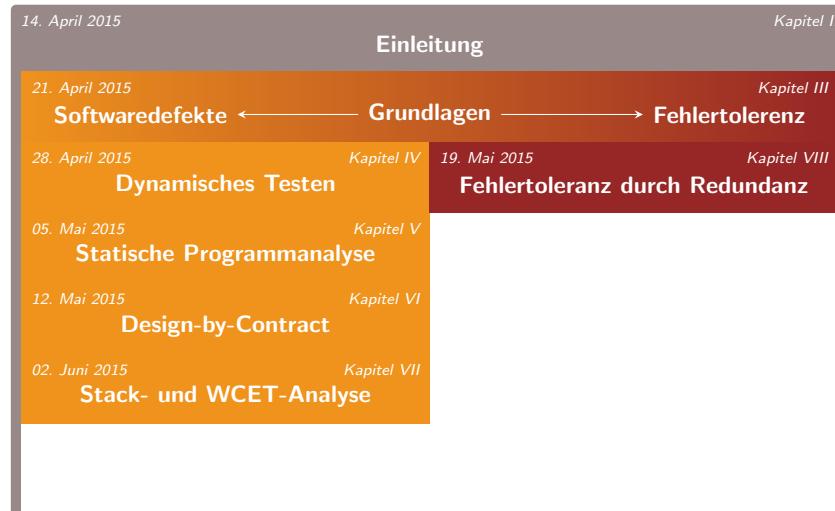
Die Laufzeit



- Messbasierte Laufzeitbestimmung → Beobachtung
- Statische WCET-Analyse → Obere/untere Schranke
- Zu finden: Längster Pfad (Timing Schema, Zeitanalysegraph)
- Dauer der Elementaroperationen: Hardware-Analyse
- Die Analyse ist sicher (sound) falls Upper Bound \geq WCET



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



Fehlerarten \mapsto Toleranz von SDCs und DUEs

Redundanz \mapsto hat mehrere Dimensionen

- {hot, warm, cold} standby
- Fehlererkennung, -diagnose, -eindämmung, -maskierung

Replikation \mapsto koordinierter Einsatz von struktureller Redundanz

- Replikation der **Eingaben**, Abstimmung der **Ausgaben**
- Fehlererkennung durch **Relativtest**
- **Zeitliche** und **räumliche Isolation** einzelner Replikate

Triple Modular Redundancy \mapsto Hardware-redundanz

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

Process Level Redundancy \mapsto „TMR in Software“

- Reduziert Kosten von TMR, zulasten eines geringeren Schutzes

Diversität \mapsto versucht **Gleichtaktfehler** auszuschließen



14. April 2015

Einleitung

Kapitel II

21. April 2015

Softwaredefekte

Grundlagen

Kapitel III

28. April 2015

Dynamisches Testen

Kapitel IV

19. Mai 2015

Fehlertoleranz durch Redundanz

Kapitel VIII

05. Mai 2015

Statische Programmanalyse

Kapitel V

09. Juni 2015

Härtung v. Daten- & Kontrollfluss

Kapitel IX

12. Mai 2015

Design-by-Contract

Kapitel VI

02. Juni 2015

Stack- und WCET-Analyse

Kapitel VII



Härtung von Code & Daten

Fehlererkennung Durch arithmetische Codierung

\leadsto Einsatz von **Informationsredundanz** durch Prüfbits

- Fehlererkennung durch **Absoluttest** (auch Akzeptanztest)

AN-Codierung \leadsto 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**

\leadsto Vollständige Fehlererkennung von **Operanden-, Berechnungs- und Operatorfehlern**

- Codierung des Kontrollflusses \leadsto **Signaturen für Grundblöcke**

CoRed-Ansatz \leadsto selektive Anwendung der ANBD-Codierung

- Durchgehende arithmetische Codierung wäre zu teuer



Härtung von Code & Daten (Forts.)

■ ANBD-Codierung härtet **Daten und Kontrollfluss**

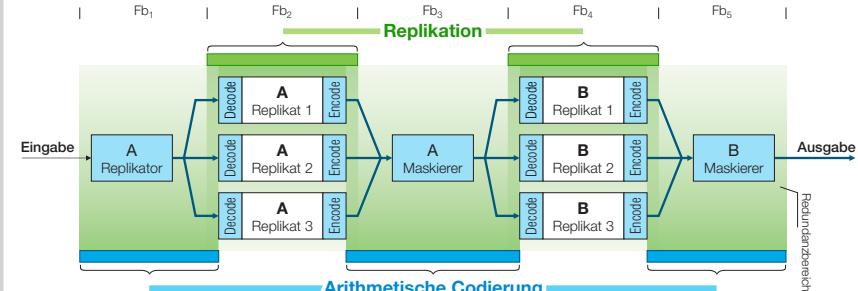
- Operanden-, Berechnungs- und Operatorfehler

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

\leadsto Signatur B_v und Zeitstempel D

\leadsto **Nachteil:** enorme hohe Laufzeitkosten

„Combined Redundancy“ \leadsto ANBD-Codierung selektiv anwenden

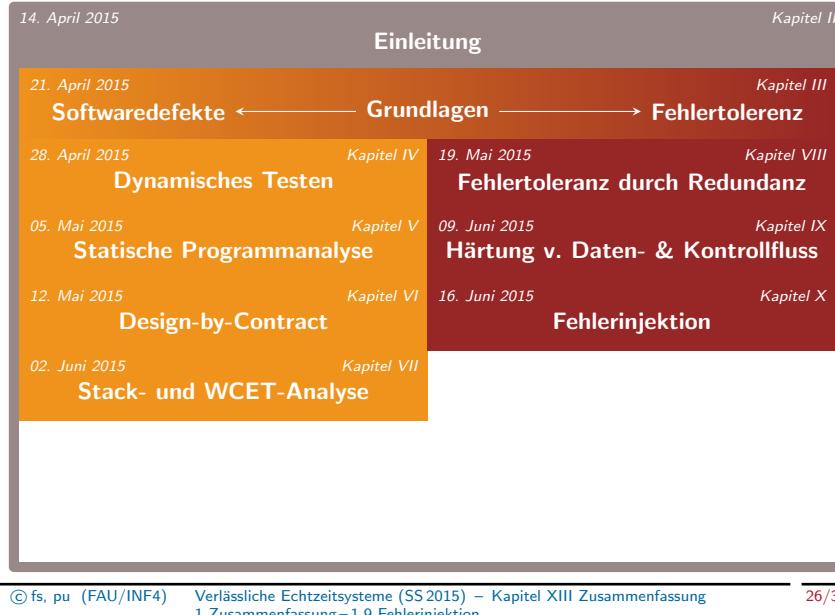


- Sichert den „single point of failure“ replizierter Ausführung

\leadsto Codierte Implementierung des Mehrheitsentscheids



Überblick



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



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



Überblick



Fehlertoleranz durch Verteilung

⚠ Hochgradig anwendungsspezifisch

- Zuschnitt und Platzierung von Funktionen
- Schnittstellen behalten im Fehlerfall ihre Gültigkeit
- Beispiel: Fly-by-Wire von Airbus

☞ Das Kommunikationssystem ist der technische Unterbau!

- Fehlererkennung auch im verteilten Fall?
- Fehlereingrenzung durch eine Sicherheitshüllen

■ Ereignisgesteuerte Kommunikation

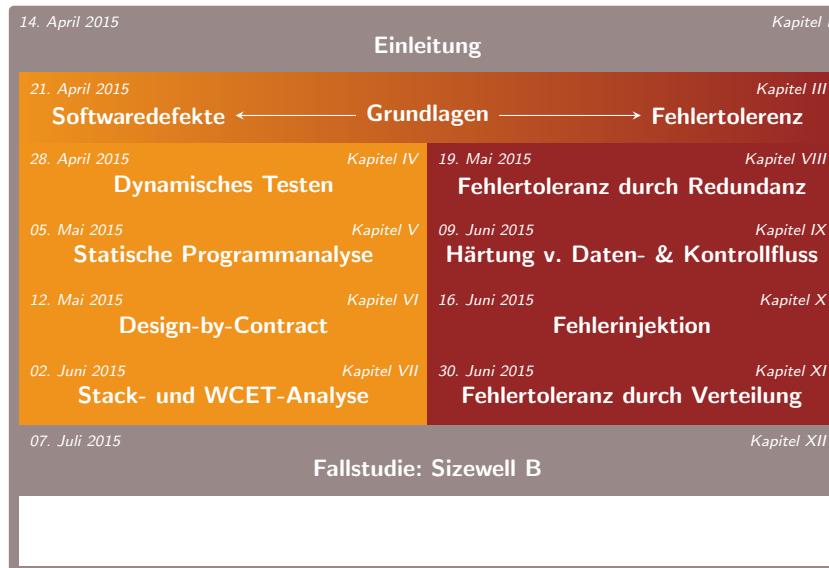
- Übertragung abhängig von Lastsituation
- Ereignisnachrichten → Keine zeitliche Kapselung

■ Zeitgesteuerte Kommunikation

- Auslastung vorab bestimmbar
- Zustandsnachrichten → Zeitliche Kapselung
- Aufwendige Uhrensynchronisation erforderlich

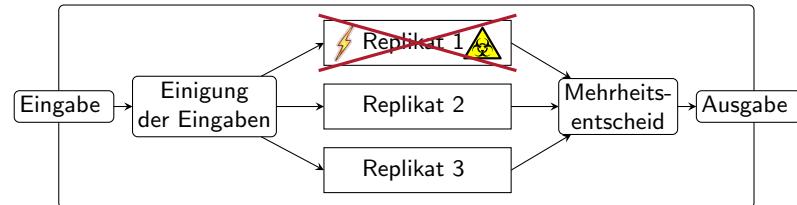


Überblick



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



Fallstudie: Sizewell B

■ Wie werden **echte verlässliche Echtzeitsysteme** entwickelt?

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

☞ Betrachtung am Beispiel des primären Reaktorschutzsystems (PPS) des Sizewell B Kernkraftwerks



Sizewell B ~ primäres Reaktorschutzsystem

- Einziger Zweck: sichere Abschaltung des Reaktors

Redundanz ~ Absicherung gegen Systemausfälle

- Vierfach

Diversität ~ Abfedern von Software-Defekten

- Unterschiedliche Hardware und Software
- Analoges Sekundärsystem

Isolation ~ Abschottung der einzelnen Replikate

- Technisch \mapsto optische Kommunikationsmedien
- Zeitlich \mapsto nicht-gekoppelte, eigenständige Rechner
- Räumlich \mapsto verschiedene Aufstellorte und Kabelrouten

Verifikation ~ umfangreiche statische Prüfung von Software

- Vielschichtiger Prozess, Betrachtung von Quell- und Binärcode



14. April 2015

Einleitung

Kapitel II

21. April 2015

Softwaredefekte \longleftrightarrow Grundlagen

Kapitel III

28. April 2015

Dynamisches Testen

Kapitel IV

19. Mai 2015

Kapitel VIII

Fehlertoleranz durch Redundanz

05. Mai 2015

Statische Programmanalyse

Kapitel V

09. Juni 2015

Kapitel IX

Härtung v. Daten- & Kontrollfluss

12. Mai 2015

Design-by-Contract

Kapitel VI

16. Juni 2015

Kapitel X

Fehlerinjektion

02. Juni 2015

Stack- und WCET-Analyse

Kapitel VII

30. Juni 2015

Kapitel XI

Fehlertoleranz durch Verteilung

07. Juli 2015

Fallstudie: Sizewell B

Kapitel XII

23. Juni 2015 & 07. Juli 2015

Vorläufe: Industrie & Forschung



{B, M}-Arbeiten . . . Promotion

Forschungs-/Entwicklungsprojekte: Universität, Forschungseinrichtungen, Industrie

<https://www4.cs.uni-erlangen.de/Theses>
oder besser noch: Kommt vorbei!



2 Abschlussarbeiten