

Verlässliche Echtzeitsysteme

Zusammenfassung

Peter Ulbrich

Lehrstuhl für Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität Erlangen-Nürnberg

<https://www4.cs.fau.de>

15. Juli 2019

Überblick

25. April 2019

Kapitel II

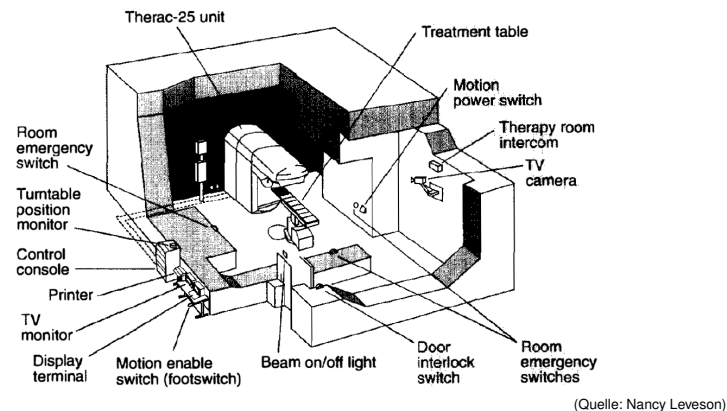
Einleitung

pu Verlässliche Echtzeitsysteme (SS 19) – Kapitel XII Zusammenfassung
1 Zusammenfassung – 1.1 Einleitung

3/35

Einleitung

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



Ziel: zuverlässiger Betrieb, minimierte Ausfallwahrscheinlichkeit

Überblick

25. April 2019

Kapitel III

Einleitung

29. April 2019

Kapitel III

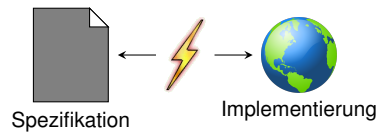
Softwaredefekte ← Grundlagen → Fehlertolerenz

pu Verlässliche Echtzeitsysteme (SS 19) – Kapitel XII Zusammenfassung
1 Zusammenfassung – 1.2 Grundlagen

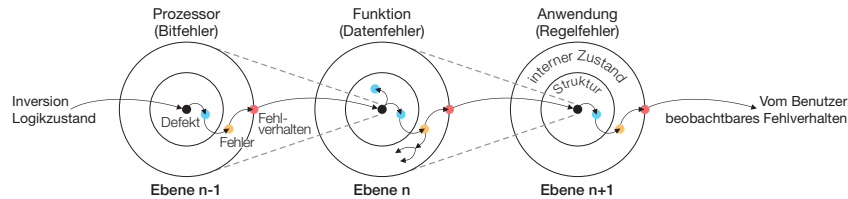
5/35

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!

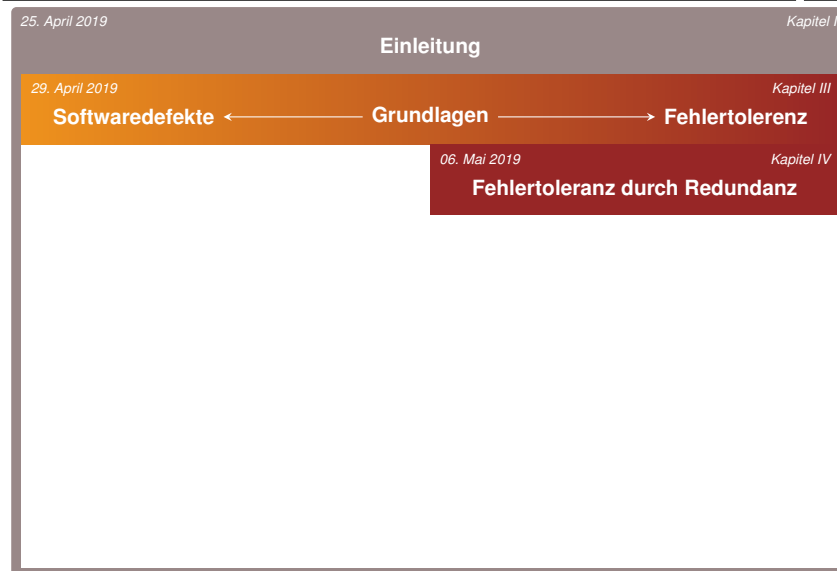
- Gefahren-, Risiko- und Fehlerbaumanalyse

Software- vs. Hardwarefehler \leadsto Klassifikation & Ursachen

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

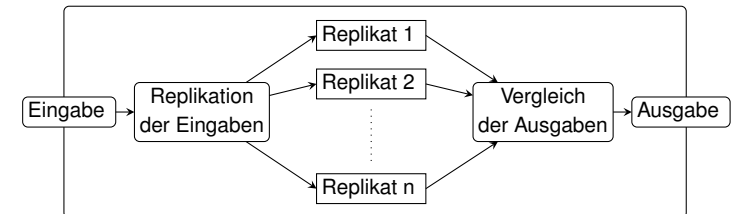


Überblick



Redundante Ausführung

- Fehlertoleranz erfordert **Redundanz**
 - Redundanz in der **Struktur**, **Funktion**, **Information** oder **Zeit**
- Ausnutzung struktureller Redundanz \leadsto **Replikation**
 - Replikation der **Eingaben**, Abstimmung der **Ausgaben**
 - Fehlererkennung durch **Relativtest**
 - **Zeitliche** und **räumliche Isolation** einzelner Replikate



- **Replikdeterminismus**
 - Einigung über die Eingabewerte \mapsto Akzeptanzmaskierer
 - Deterministische Umsetzung der Funktion



Redundante Ausführung (Forts.)

Fehlertypen → SDCs und DUEs

Kritische Bruchstellen → Bereiche ohne Redundanz

Hardwarenasierte Replikation → TMR

- {hot, warm, cold} standby
- 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

25. April 2019

Kapitel II

Einleitung

29. April 2019

Kapitel III

Softwaredefekte ← Grundlagen → Fehlertoleranz

06. Mai 2019

Kapitel IV

Fehlertoleranz durch Redundanz

13. Mai 2019

Kapitel V

Härtung v. Daten- & Kontrollfluss



Härtung von Code & Daten

Fehlererkennung → Durch Codierung

→ Einsatz von Informationsredundanz durch Prüfbits

- Fehlererkennung durch Akzeptanztest (Absoluttest)

AN-Codierung → Codierung von Berechnungen

- 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 → ANBD-Codierung der Replikationsinfrastruktur

- Durchgehende arithmetische Codierung wäre zu teuer



Härtung von Code & Daten (Forts.)

- ANBD-Codierung härtet Daten und Kontrollfluss

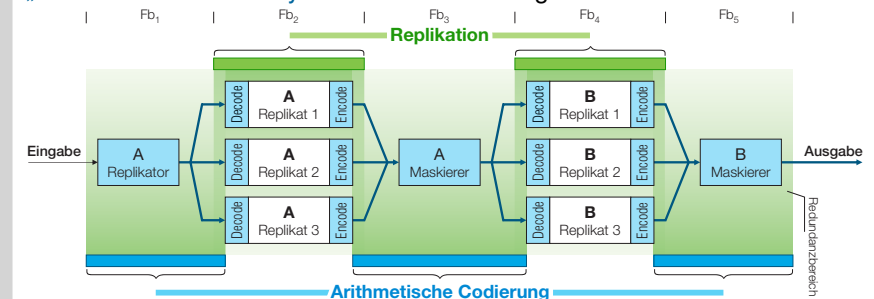
- Operanden-, Berechnungs- und Operatorfehler

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

- 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/reproduzierbar



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}-basiert

FAIL* → Grundlage für generische Fehlerinjektion?

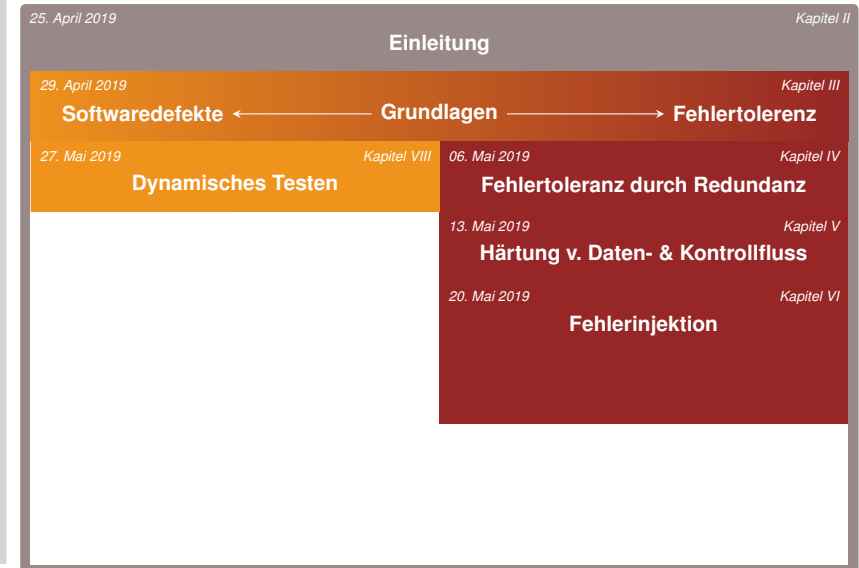
- Basierend auf virtuellen Zielsystemen
- Flexible Plattform für Fehlerinjektion
- Schnelle Experimentdurchführung durch Parallelisierung

Zuverlässigkeitsmetriken → Messung und Auswertung

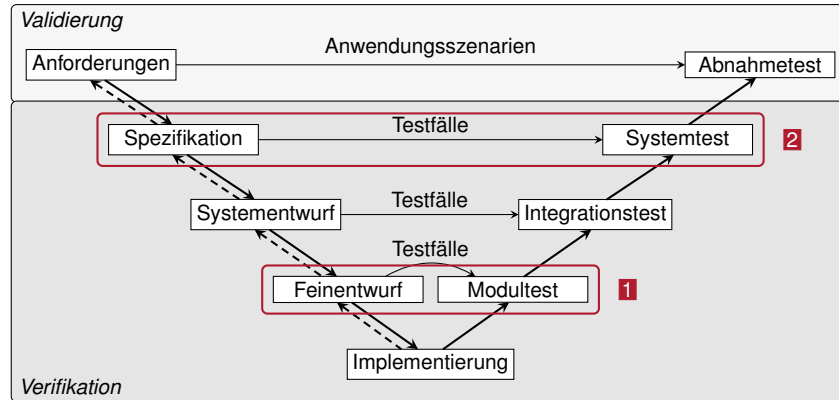
- Absolute Zahlen versus Fehlerwahrscheinlichkeit



Überblick



Testen



1 **Modultests** \leadsto Grundbegriffe und Problemstellung

\rightarrow Black- vs. White-Box, Testüberdeckung

2 **Systemtest** \leadsto Testen verteilter Echtzeitsysteme

\rightarrow Problemstellung und Herausforderungen



Testen (Forts.)



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

- Modul-, Integrations-, System- und Abnahmetest
- \rightarrow Kann die Abwesenheit von Defekten aber nie garantieren

■ Modultests

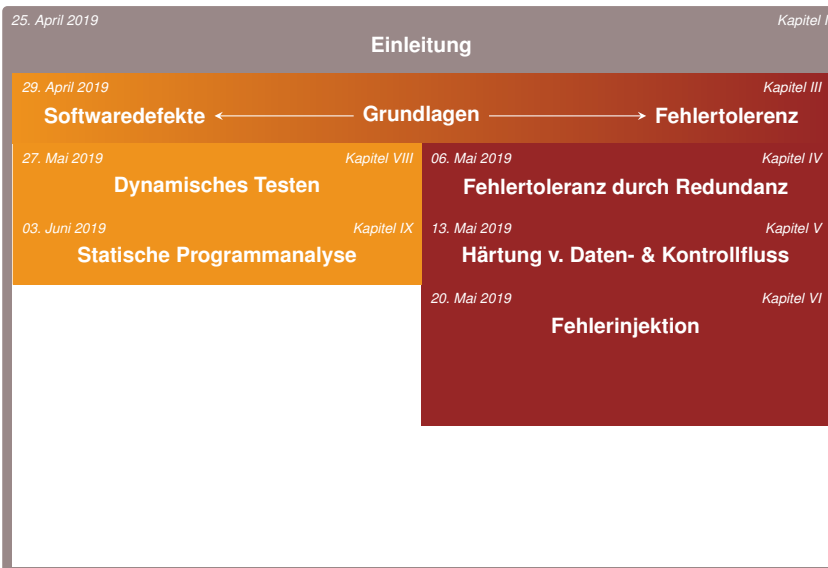
- Black-Box- vs. White-Box-Tests
- McCabe's Cyclomatic Complexity \leadsto Minimalzahl von Testfällen
- Kontrollflussorientierte Testüberdeckung
 - Anweisungs-, Zweig-, Pfad- und Bedingungsüberdeckung
 - Angaben zur Testüberdeckung sind immer **relativ**!

■ Systemtests für verteilte Echtzeitsysteme sind **herausfordernd**!

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



Überblick



Abstrakte Interpretation



Enthält das Programm **Laufzeitfehler**?

- Ganzzahl- oder Fließkommaüberläufe, nicht-initialisierte Variablen, ...
- Können wir diese Frage **vor der Laufzeit** beantworten?

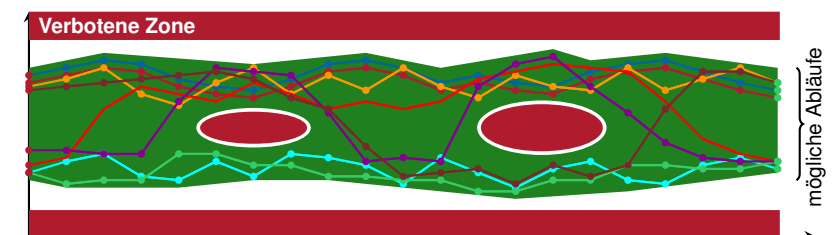


Für die **konkrete Programmsemantik** geht das nicht

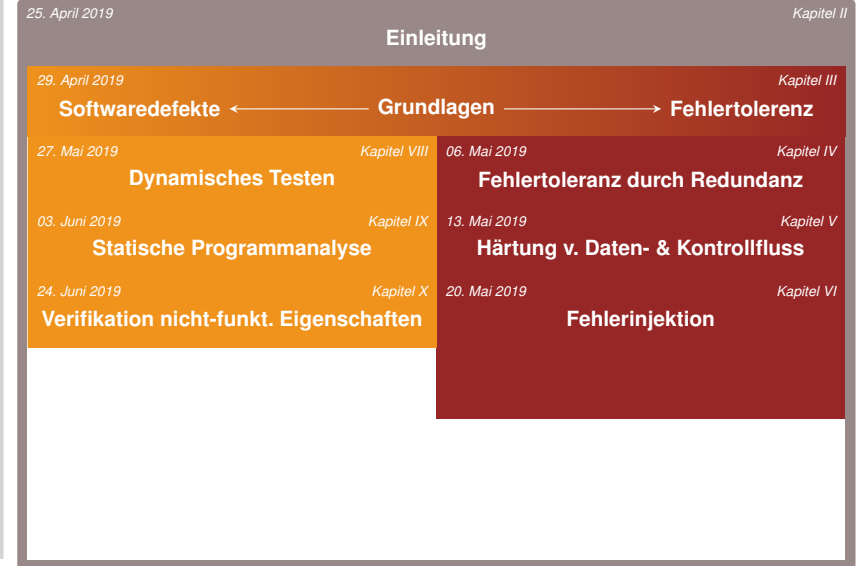
- Eine **sicher Abstraktion** könnte für diesen Zweck aber ausreichen
 - Für Zugriffe auf Felder ist nur der möglichen Wertebereich des Index wichtig
 - Welcher konkrete Wert wann angenommen wird, ist nicht von Belang.



Einsatz einer **abstrakten Programmsemantik**



- 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 **effizient berechnet** werden
- Abstraktion und Konkretisierung implizieren keinen Präzisionsverlust!
- **Analyse und Vereinfachung**
 - **Pfadsemantiken** beschreiben die konkrete Programmsemantik
 - Approximation durch **Pfadpräfixe** und **Sammelsemantik**



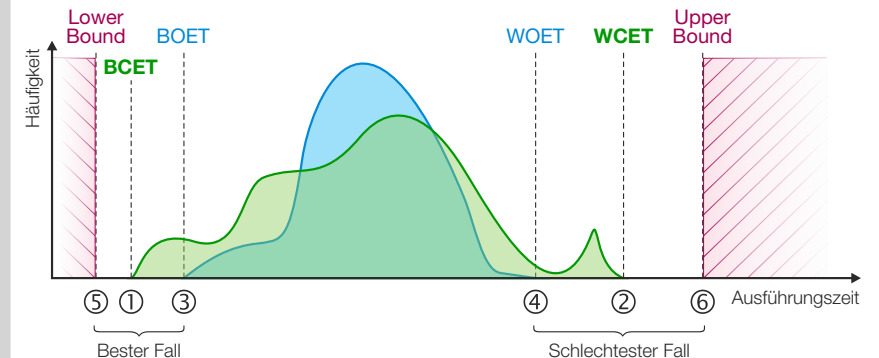
Der Stapelspeicher (Stack)

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



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



Design-by-Contract (Forts.)

Funktionale Programmeigenschaften \mapsto Zusicherungen

- Vorbedingungen, Nachbedingungen und Invarianten
- Beschrieben durch Ausdrücke der Prädikatenlogik

Prädikamentransformation \leadsto symbolische Ausführung

- Bildet Semantik durch Transformation von Zusicherungen nach
- Strongest postcondition, weakest precondition

Hoare-Kalkül \leadsto deduktive Ableitung von Nachbedingungen

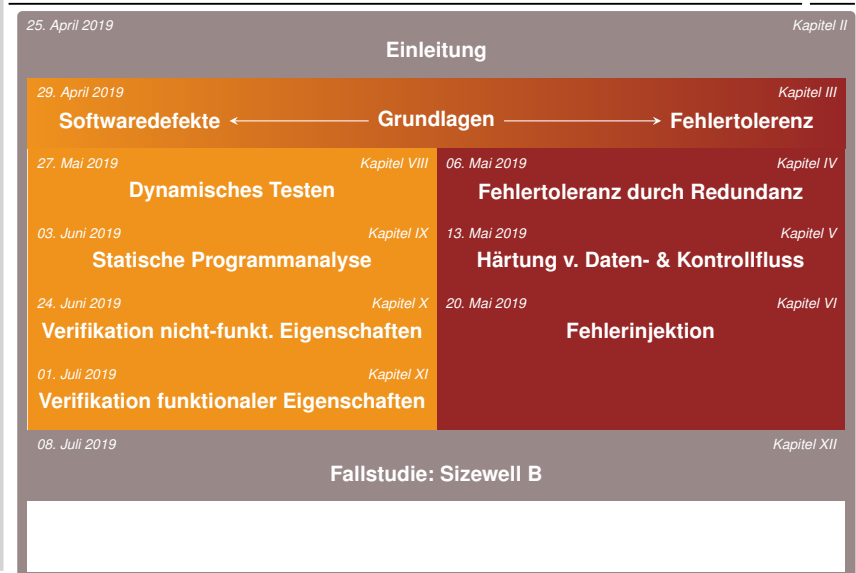
- Hoare-Tripel, Axiome für leere Anweisungen und Zuweisungen
- Ableitungsregeln für Sequenzen, Verzweigungen und Iterationen
- Konsequenzregel passt Vor-/Nachbedingungen an

WP-Kalkül \mapsto „Hoare-Kalkül rückwärts“

Praxisbezug \leadsto Astreé implementiert dieses Konzept nur teilweise!




Überblick



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



Fallstudien (Forts.)

Sizewell B \leadsto primäres Reaktorschutzsystem

- Einziger Zweck: sichere Abschaltung des Reaktors

Redundanz \leadsto Absicherung gegen Systemausfälle

- Vierfach

Diversität \leadsto Abfedern von Software-Defekten

- Unterschiedliche Hardware und Software
- Analoges Sekundärsystem

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



Überblick



Überblick



Gliederung

- 1 Zusammenfassung
 - Einleitung
 - Grundlagen
 - Redundante Ausführung
 - Härtung von Daten- und Kontrollfluss
 - Fehlerinjektion
 - Testen
 - Statische Programmanalyse
 - Statische Analyse nicht-funktionaler Eigenschaften
 - Statische Analyse funktionaler Eigenschaften
 - Fallstudie: Sizewell B
 - Vorträge
 - Prüfungsrelevanz

2 Abschlussarbeiten



{B, M}-Arbeiten ... Promotion

Forschungs-/Entwicklungsprojekte: Universität, Forschungseinrichtungen, Industrie

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

