

# Verlässliche Echtzeitsysteme

## Zusammenfassung

**Peter Ulbrich**

Lehrstuhl für Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität Erlangen-Nürnberg

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

KW30 2020

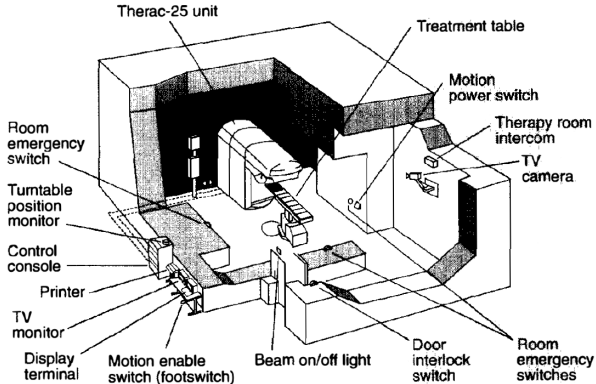


## Einleitung





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



(Quelle: Nancy Leveson)



**Ziel:** zuverlässiger Betrieb, minimierte Ausfallwahrscheinlichkeit



KW17 2020

Kapitel II

## Einleitung

KW18 2020

Kapitel III

## Grundlagen



KW17 2020

Kapitel II

## Einleitung

KW18 2020

Kapitel III

Softwaredefekte



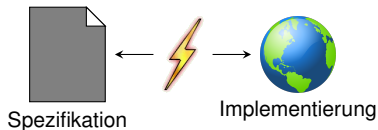
Grundlagen



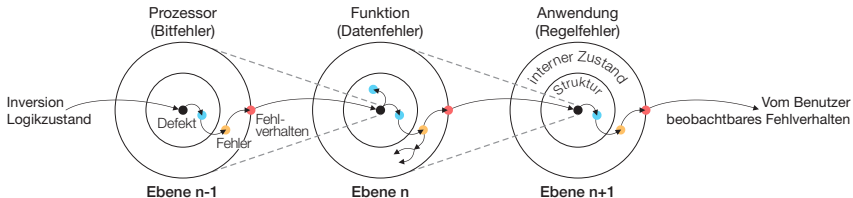
Fehlertoleranz



- **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**  $\rightsquigarrow$  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



KW17 2020

Kapitel II

## Einleitung

KW18 2020

Kapitel III

Softwaredefekte



Grundlagen



Fehlertoleranz

KW19 2020

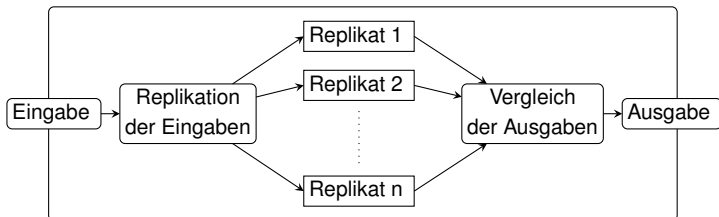
Kapitel IV

Fehlertoleranz durch Redundanz



# 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



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



KW17 2020

Kapitel II

## Einleitung

KW18 2020

Kapitel III

Softwaredefekte ← Grundlagen → Fehlertoleranz

KW19 2020

Kapitel IV

**Fehlertoleranz durch Redundanz**

KW20 2020

Kapitel V

**Härtung v. Daten- & Kontrollfluss**



Fehlererkennung  $\mapsto$  Durch Codierung

- $\leadsto$  Einsatz von Informationsredundanz durch Prüfbits
- Fehlererkennung durch Akzeptanztest (Absoluttest)

AN-Codierung  $\mapsto$  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  $\mapsto$  Erweitert die AN-Codierung

- Um statische Signaturen und dynamische Zeitstempel
- $\leadsto$  Vollständige Fehlererfassung von Operanden-, Berechnungs- und Operatorfehlern
- Codierung des Kontrollflusses  $\leadsto$  Signaturen für Grundblöcke

CoRed-Ansatz  $\mapsto$  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

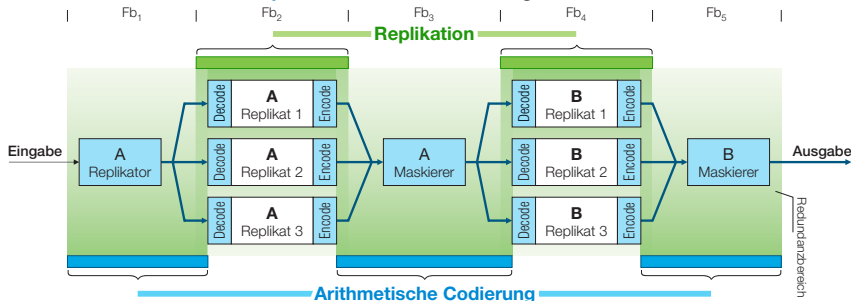
$$v_c = Av + B_v + D; \quad A > 1 \wedge B_v + D < A$$

- 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



KW17 2020

Kapitel II

## Einleitung

KW18 2020

Kapitel III

Softwaredefekte

← Grundlagen →

Fehlertoleranz

KW19 2020

Kapitel IV

Fehlertoleranz durch Redundanz

KW20 2020

Kapitel V

Härtung v. Daten- & Kontrollfluss

KW21 2020

Kapitel VI

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



## 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



KW17 2020

Kapitel II

## Einleitung

KW18 2020

Kapitel III

Softwaredefekte

← Grundlagen →

Fehlertoleranz

KW22 2020

Kapitel VII

Dynamisches Testen

KW19 2020

Kapitel IV

Fehlertoleranz durch Redundanz

KW25 2020

Kapitel VIII

Statische Programmanalyse

KW20 2020

Kapitel V

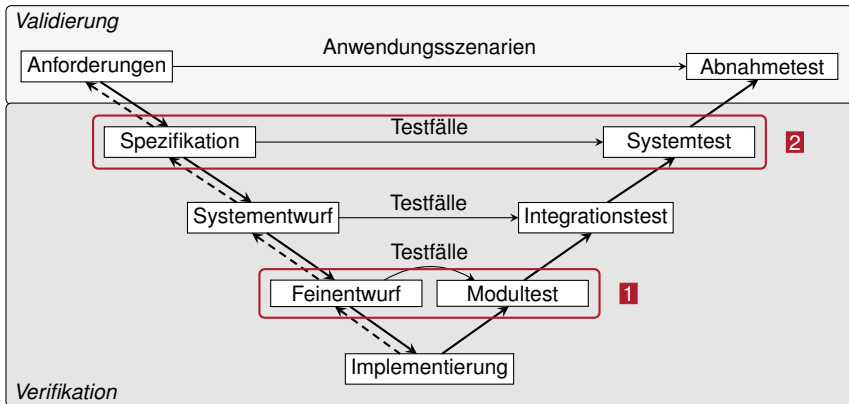
Härtung v. Daten- & Kontrollfluss

KW21 2020

Kapitel VI

Fehlerinjektion





**1** **Modultests** ~ Grundbegriffe und Problemstellung

→ Black- vs. White-Box, Testüberdeckung

**2** **Systemtest** ~ Testen verteilter Echtzeitsysteme

→ Problemstellung und Herausforderungen





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

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

## ■ Modultests

- **Black-Box- vs. White-Box-Tests**
- **McCabe's Cyclomatic Complexity**  $\rightsquigarrow$  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



KW17 2020

Kapitel II

## Einleitung

KW18 2020

Kapitel III

**Softwaredefekte** ← ————— **Grundlagen** ————— → **Fehlertoleranz**

KW22 2020

Kapitel VII

**Dynamisches Testen**

KW19 2020

Kapitel IV

**Fehlertoleranz durch Redundanz**

KW25 2020

Kapitel VIII

**Statische Programmanalyse**

KW20 2020

Kapitel V

**Härtung v. Daten- & Kontrollfluss**

KW26 2020

Kapitel IX

**Verifikation nicht-funkt. Eigenschaften**

KW21 2020

Kapitel VI

**Fehlerinjektion**



# Abstrakte Interpretation

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

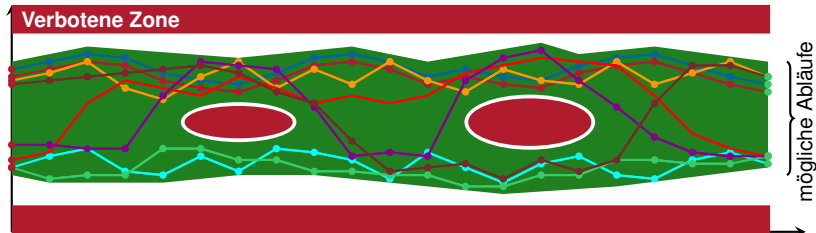


Für die **konkrete Programmsemantik** geht das nicht

- 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**



KW17 2020

Kapitel II

## Einleitung

KW18 2020

Kapitel III

**Softwaredefekte** ← ————— **Grundlagen** ————— → **Fehlertoleranz**

KW22 2020

Kapitel VII

**Dynamisches Testen**

KW19 2020

Kapitel IV

**Fehlertoleranz durch Redundanz**

KW25 2020

Kapitel VIII

**Statische Programmanalyse**

KW20 2020

Kapitel V

**Härtung v. Daten- & Kontrollfluss**

KW26 2020

Kapitel IX

**Verifikation nicht-funkt. Eigenschaften**

KW21 2020

Kapitel VI

**Fehlerinjektion**

KW27 2020

Kapitel X

**Verifikation funktionaler Eigenschaften**



# 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  $\leadsto$  Keine Aussagen zum maximalen Verbrauch



Statische Programmanalyse

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





KW17 2020

Kapitel II

## Einleitung

KW18 2020

Kapitel III

**Softwaredefekte** ← **Grundlagen** → **Fehlertoleranz**

KW22 2020

Kapitel VII

**Dynamisches Testen**

KW19 2020

Kapitel IV

**Fehlertoleranz durch Redundanz**

KW25 2020

Kapitel VIII

**Statische Programmanalyse**

KW20 2020

Kapitel V

**Härtung v. Daten- & Kontrollfluss**

KW26 2020

Kapitel IX

**Verifikation nicht-funkt. Eigenschaften**

KW21 2020

Kapitel VI

**Fehlerinjektion**

KW27 2020

Kapitel X

**Verifikation funktionaler Eigenschaften**

KW28 2020

Kapitel XI

**Fallstudie: Sizewell B**



# Design-by-Contract

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



C.A.R. Hoare



Edger W. Dijkstra



Funktionale Programmeigenschaften  $\mapsto$  Zusicherungen

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

Prädikamentransformation  $\rightsquigarrow$  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“

Praxisbezug  $\rightsquigarrow$  Astaré implementiert dieses Konzept nur teilweise!



KW17 2020

Kapitel II

## Einleitung

KW18 2020

Kapitel III

**Softwaredefekte** ← **Grundlagen** → **Fehlertoleranz**

KW22 2020

Kapitel VII

**Dynamisches Testen**

KW19 2020

Kapitel IV

**Fehlertoleranz durch Redundanz**

KW25 2020

Kapitel VIII

**Statische Programmanalyse**

KW20 2020

Kapitel V

**Härtung v. Daten- & Kontrollfluss**

KW26 2020

Kapitel IX

**Verifikation nicht-funkt. Eigenschaften**

KW21 2020

Kapitel VI

**Fehlerinjektion**

KW27 2020

Kapitel X

**Verifikation funktionaler Eigenschaften**

KW28 2020

Kapitel XI

**Fallstudie: Sizewell B**

KW29 2020

**Industrievortrag**



## ■ 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**  $\leadsto$  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 Replikat

- 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



KW17 2020

Kapitel II

## Einleitung

KW18 2020

Kapitel III

**Softwaredefekte** ← ————— **Grundlagen** ————— → **Fehlertoleranz**

KW22 2020

Kapitel VII

**Dynamisches Testen**

KW19 2020

Kapitel IV

**Fehlertoleranz durch Redundanz**

KW25 2020

Kapitel VIII

**Statische Programmanalyse**

KW20 2020

Kapitel V

**Härtung v. Daten- & Kontrollfluss**

KW26 2020

Kapitel IX

**Verifikation nicht-funkt. Eigenschaften**

KW21 2020

Kapitel VI

**Fehlerinjektion**

KW27 2020

Kapitel X

**Verifikation funktionaler Eigenschaften**

KW28 2020

Kapitel XI

**Fallstudie: Sizewell B**

KW29 2020

**Industrievortrag**



KW17 2020

Kapitel II

## Einleitung



KW18 2020

Kapitel III

Softwaredefekte



Grundlagen



Fehlertoleranz



KW22 2020

Kapitel VII

Dynamisches Testen

KW19 2020

Kapitel IV

Fehlertoleranz durch Redundanz

KW25 2020

Kapitel VIII

Statische Programmanalyse

KW20 2020

Kapitel V

Härtung v. Daten- & Kontrollfluss

KW26 2020

Kapitel IX

Verifikation nicht-funkt. Eigenschaften

KW21 2020

Kapitel VI

Fehlerinjektion

KW27 2020

Kapitel X

Verifikation funktionaler Eigenschaften

KW28 2020

Kapitel XI

Fallstudie: Sizewell B

KW29 2020

Industrievortrag



KW17 2020

Kapitel II

## Einleitung



KW18 2020

Kapitel III

Softwaredefekte



Grundlagen



Fehlertoleranz



KW22 2020

Kapitel VII

KW19 2020

Kapitel IV

Dynamisches Testen

Fehlertoleranz durch Redundanz



KW25 2020

Kapitel VIII

KW20 2020

Kapitel V

Statische Programmanalyse

Härtung v. Daten- & Kontrollflüssen



KW26 2020

Kapitel IX

KW21 2020

Kapitel VI

Verifikation nicht-funkt. Eigenschaften

Fehlerinjektion



KW27 2020

Kapitel X

Verifikation funktionaler Eigenschaften

KW28 2020

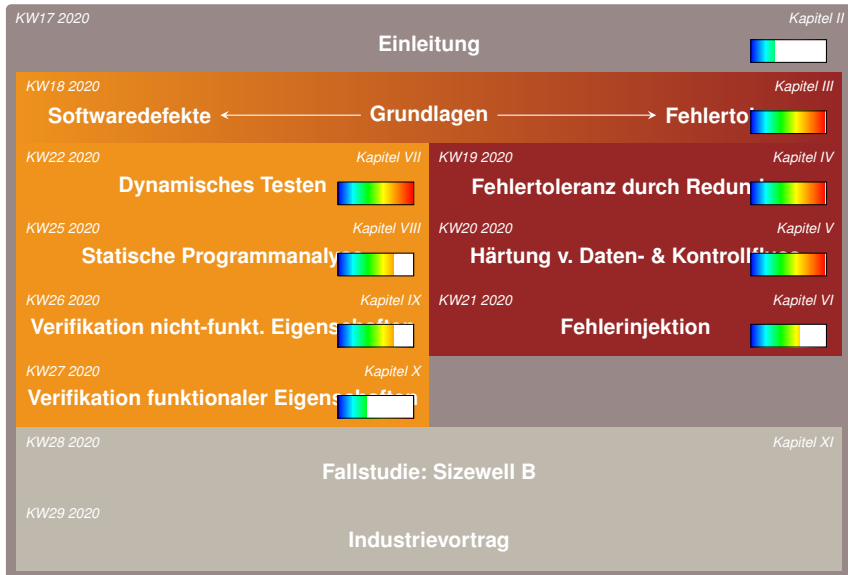
Kapitel XI

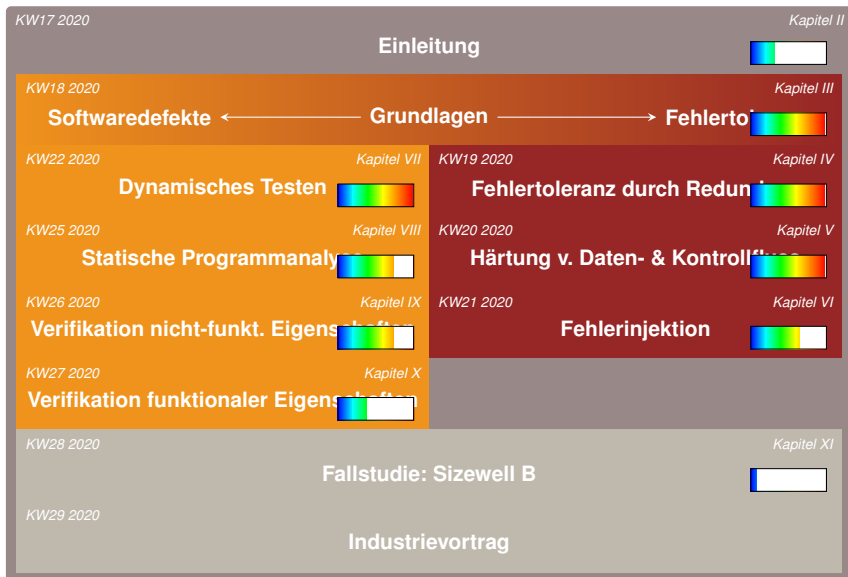
Fallstudie: Sizewell B

KW29 2020

Industrievortrag







## 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!

