

Verlässliche Echtzeitsysteme

Zusammenfassung

Peter Ulbrich

Lehrstuhl für Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität Erlangen-Nürnberg

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

15. Juli 2019

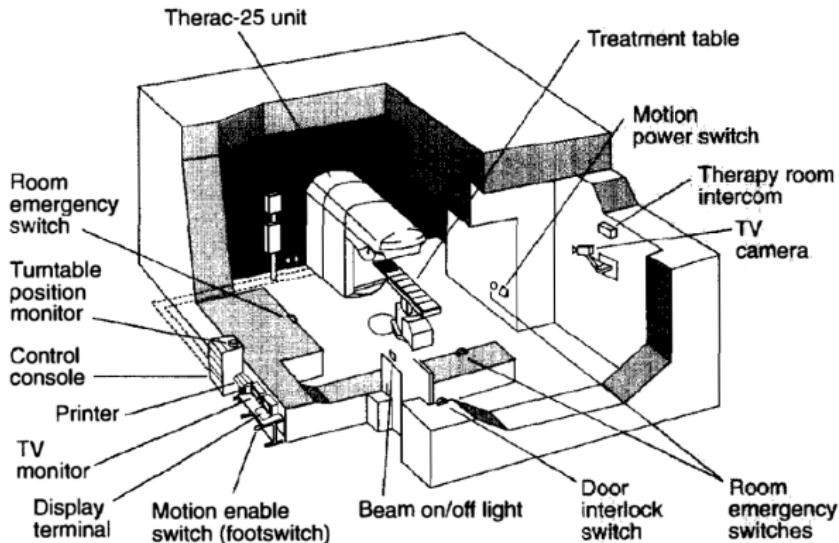


Einleitung





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



(Quelle: Nancy Leveson)



Ziel: zuverlässiger Betrieb, minimierte Ausfallwahrscheinlichkeit



25. April 2019

Einleitung

Kapitel II

29. April 2019

Grundlagen

Kapitel III



25. April 2019

Kapitel II

Einleitung

29. April 2019

Kapitel III

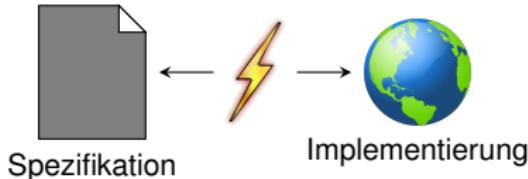
Softwaredefekte

Grundlagen

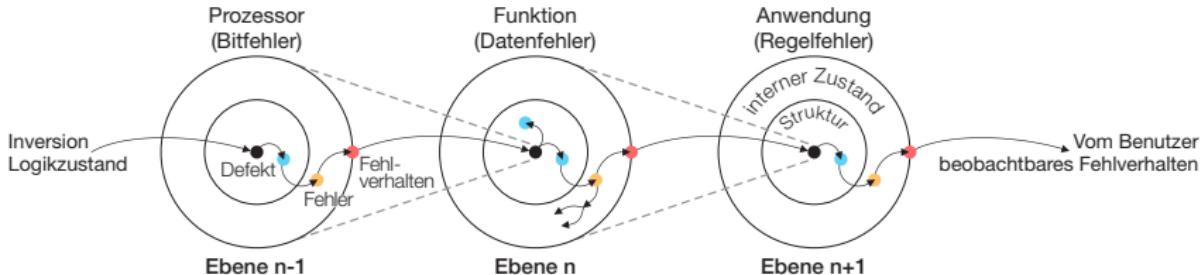
Fehlertoleranz



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



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



Ziel: Reduktion des **vom Benutzer beobachtbaren Fehlerverhaltens!**



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



25. April 2019

Kapitel II

Einleitung

29. April 2019

Kapitel III

Softwaredefekte

Grundlagen

Fehlertoleranz

06. Mai 2019

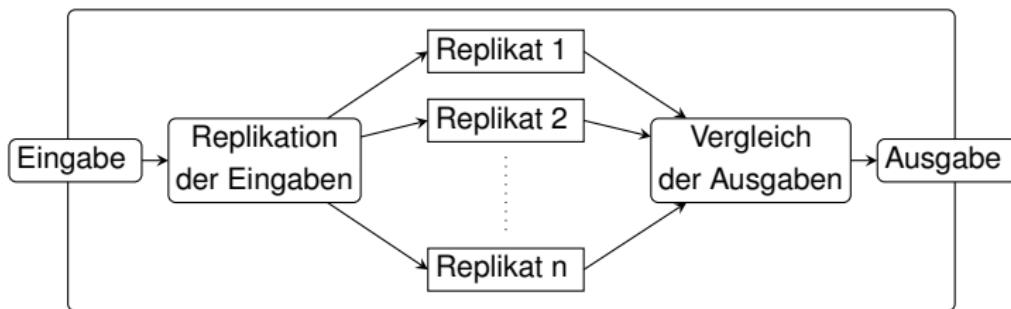
Kapitel IV

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



Fehlerarten → SDCs und DUEs

Kritische Bruchstellen → Bereiche ohne Redundanz

Hardwareasierte Replikation → TMR

- {hot, warm, cold} standby
- Dreifache Auslegung, toleriert Fehler im Wertebereich
- Zuverlässigkeit von Replikat und Gesamtsystem

Process Level Redundancy → „TMR in Software“

- Reduziert Kosten von TMR, zulasten eines geringeren Schutzes

Diversität → versucht Gleichtaktfehler auszuschließen



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 Fehlerfassung von **Operanden-, Berechnungs- und Operatorfehlern**
- Codierung des Kontrollflusses ~ **Signaturen für Grundblöcke**

CoRed-Ansatz → ANBD-Codierung der Replikationsinfrastruktur

- **Durchgehende arithmetische Codierung** wäre zu teuer



■ ANBD-Codierung härtet Daten und Kontrollfluss

- Operanden-, Berechnungs- und Operatorfehler

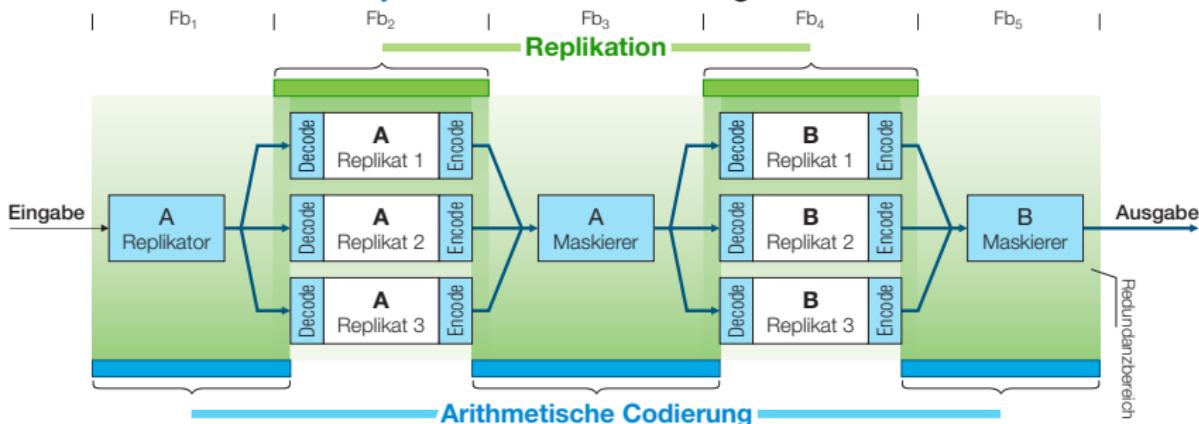
$$v_c = Av + B_v + D; \quad A > 1 \wedge B_v + D < A$$

- 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



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

20. Mai 2019

Kapitel VI

Fehlerinjektion



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

Zuverlässigkeitssmetriken → Messung und Auswertung

- Absolute Zahlen versus Fehlerwahrscheinlichkeit



25. April 2019

Kapitel II

Einleitung

29. April 2019

Kapitel III

Softwaredefekte

Grundlagen

Fehlertoleranz

27. Mai 2019

Kapitel VIII

06. Mai 2019

Kapitel IV

Dynamisches Testen

Fehlertoleranz durch Redundanz

13. Mai 2019

Kapitel V

Härtung v. Daten- & Kontrollfluss

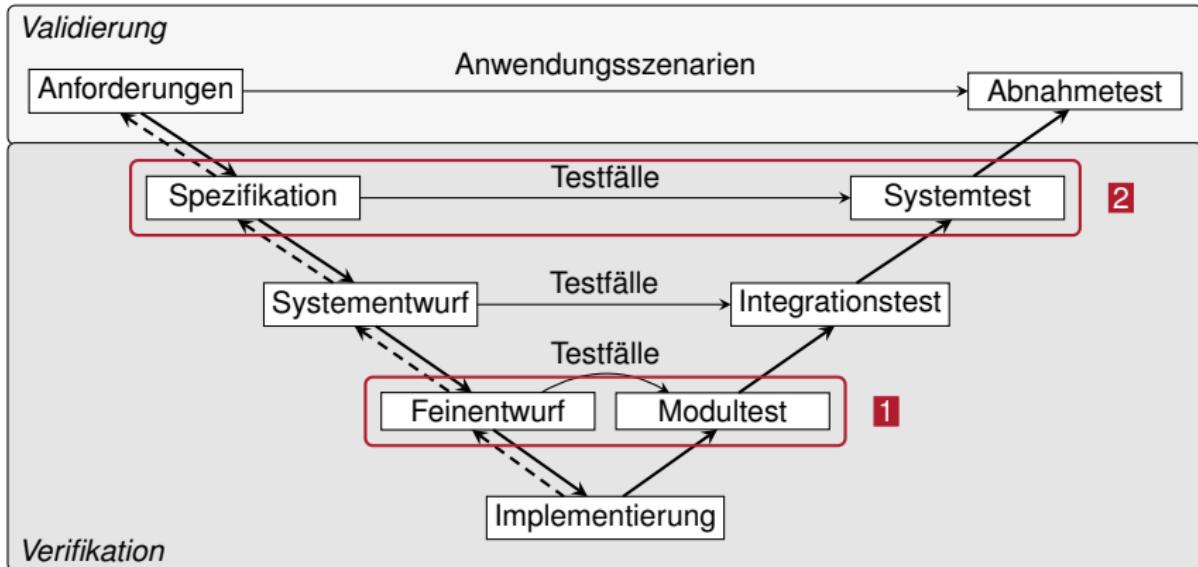
20. Mai 2019

Kapitel VI

Fehlerinjektion



Testen



- 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 → 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



25. April 2019

Kapitel II

Einleitung

29. April 2019

Kapitel III

Softwaredefekte ← Grundlagen → Fehlertoleranz

27. Mai 2019

Kapitel VIII

06. Mai 2019

Kapitel IV

Dynamisches Testen

03. Juni 2019

Kapitel IX

Fehlertoleranz durch Redundanz

Statische Programmanalyse

13. Mai 2019

Kapitel V

Härtung v. Daten- & Kontrollfluss

20. Mai 2019

Kapitel VI

Fehlerinjektion



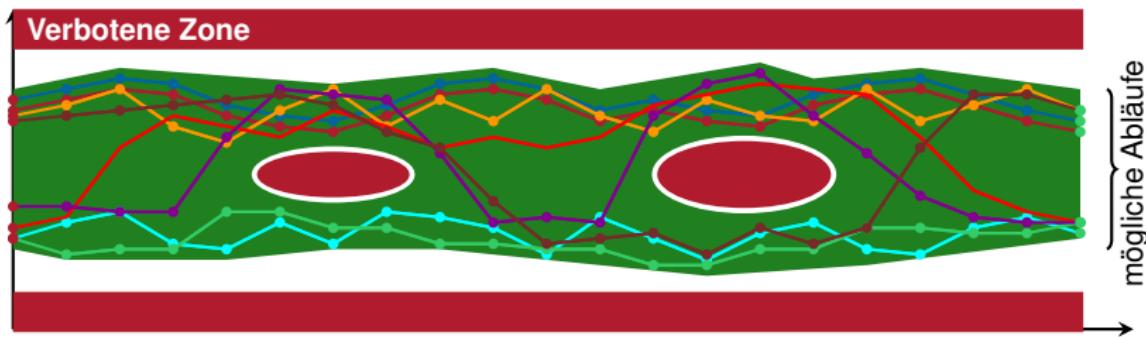
Abstrakte Interpretation

- 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**



25. April 2019

Kapitel II

Einleitung

29. April 2019

Kapitel III

Softwaredefekte ← Grundlagen → Fehlertoleranz

27. Mai 2019

Kapitel VIII

06. Mai 2019

Kapitel IV

Dynamisches Testen

03. Juni 2019

Kapitel IX

13. Mai 2019

Kapitel V

Statische Programmanalyse

24. Juni 2019

Kapitel X

20. Mai 2019

Kapitel VI

Verifikation nicht-funkt. Eigenschaften

Fehlertoleranz durch Redundanz

Härtung v. Daten- & Kontrollfluss

Fehlerinjektion



Der Stapselspeicher (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!!)

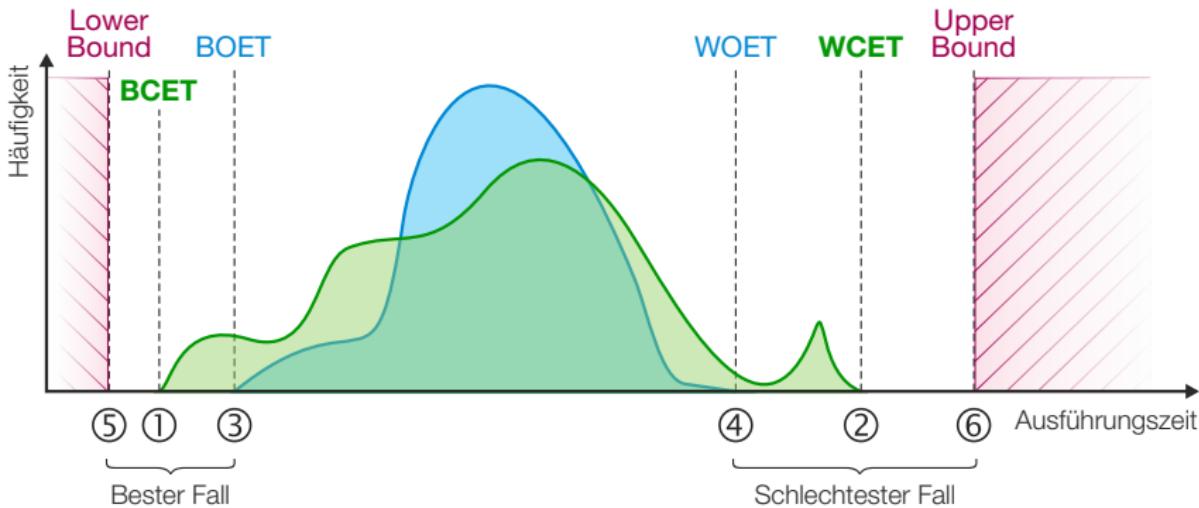
- Water-Marking, Überwachung zur Laufzeit
- Reaktiv ↪ Keine Aussagen zum maximalen Verbrauch

- 👉 Statische Programmanalyse

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



Die Laufzeit



- Messbasierte Laufzeitbestimmung \leadsto Beobachtung
- Statische WCET-Analyse \leadsto 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



25. April 2019

Kapitel II

Einleitung

29. April 2019

Kapitel III

Softwaredefekte ← Grundlagen → Fehlertoleranz

27. Mai 2019

Kapitel VIII

06. Mai 2019

Kapitel IV

Dynamisches Testen

03. Juni 2019

Kapitel IX

13. Mai 2019

Kapitel V

Statische Programmanalyse

24. Juni 2019

Kapitel X

20. Mai 2019

Kapitel VI

Verifikation nicht-funkt. Eigenschaften

01. Juli 2019

Kapitel XI

Verifikation funktionaler Eigenschaften

Fehlertoleranz durch Redundanz

Härtung v. Daten- & Kontrollfluss

Fehlerinjektion



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

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“

Praxisbezug → Astreeé implementiert dieses Konzept nur teilweise!



25. April 2019

Kapitel II

Einleitung

29. April 2019

Kapitel III

Softwaredefekte ← Grundlagen → Fehlertoleranz

27. Mai 2019

Kapitel VIII

06. Mai 2019

Kapitel IV

Dynamisches Testen

03. Juni 2019

Kapitel IX

Fehlertoleranz durch Redundanz

Kapitel V

Statische Programmanalyse

24. Juni 2019

Kapitel X

13. Mai 2019

Härtung v. Daten- & Kontrollfluss

Kapitel VI

Verifikation nicht-funkt. Eigenschaften

01. Juli 2019

Kapitel XI

Fehlerinjektion

Verifikation funktionaler Eigenschaften

08. Juli 2019

Kapitel XII

Fallstudie: Sizewell B



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 → 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



25. April 2019

Kapitel II

Einleitung

29. April 2019

Kapitel III

Softwaredefekte ← Grundlagen → Fehlertoleranz

27. Mai 2019

Kapitel VIII

06. Mai 2019

Kapitel IV

Dynamisches Testen

03. Juni 2019

Kapitel IX

Fehlertoleranz durch Redundanz

Kapitel V

Statische Programmanalyse

24. Juni 2019

Kapitel X

13. Mai 2019

Härtung v. Daten- & Kontrollfluss

Kapitel VI

Verifikation nicht-funkt. Eigenschaften

01. Juli 2019

Kapitel XI

Fehlerinjektion

Verifikation funktionaler Eigenschaften

08. Juli 2019

Kapitel XII

Fallstudie: Sizewell B



25. April 2019

Einleitung

Kapitel II



29. April 2019

Softwaredefekte

Grundlagen

Fehlertoleranz

Kapitel III



27. Mai 2019

Kapitel VIII

06. Mai 2019

Kapitel IV

Dynamisches Testen

Fehlertoleranz durch Redundanz

03. Juni 2019

Kapitel IX

13. Mai 2019

Kapitel V

Statische Programmanalyse

Härtung v. Daten- & Kontrollfluss

24. Juni 2019

Kapitel X

20. Mai 2019

Kapitel VI

Verifikation nicht-funkt. Eigenschaften

Fehlerinjektion

01. Juli 2019

Kapitel XI

Verifikation funktionaler Eigenschaften

08. Juli 2019

Kapitel XII

Fallstudie: Sizewell B



Überblick

25. April 2019

Einleitung

Kapitel II



29. April 2019

Softwaredefekte

Grundlagen

Fehlertoleranz

Kapitel III



27. Mai 2019

Kapitel VIII

06. Mai 2019

Kapitel IV

Dynamisches Testen

Fehlertoleranz durch Redundanz

03. Juni 2019

Kapitel IX

13. Mai 2019

Kapitel V

Statische Programmanalyse

Härtung v. Daten- & Kontrollflüssen



24. Juni 2019

Kapitel X

20. Mai 2019

Kapitel VI

Verifikation nicht-funkt. Eigenschaften

Fehlerinjektion



01. Juli 2019

Kapitel XI

Verifikation funktionaler Eigenschaften

08. Juli 2019

Kapitel XII

Fallstudie: Sizewell B



pu

Überblick

25. April 2019

Einleitung

Kapitel II



29. April 2019

Softwaredefekte

Grundlagen

Fehlertoleranz

Kapitel III



27. Mai 2019

Kapitel VIII

Dynamisches Testen



06. Mai 2019

Kapitel IV

Fehlertoleranz durch Redundanz



03. Juni 2019

Kapitel IX

Statische Programmanalyse



13. Mai 2019

Kapitel V

Härtung v. Daten- & Kontrollflüssen



24. Juni 2019

Kapitel X

Verifikation nicht-funkt. Eigenschaften



20. Mai 2019

Kapitel VI

Fehlerinjektion



01. Juli 2019

Kapitel XI

Verifikation funktionaler Eigenschaften



08. Juli 2019

Kapitel XII

Fallstudie: Sizewell B

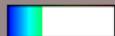


Überblick

25. April 2019

Einleitung

Kapitel II



29. April 2019

Softwaredefekte

Grundlagen

Fehlertoleranz

Kapitel III



27. Mai 2019

Kapitel VIII

Dynamisches Testen



06. Mai 2019

Kapitel IV

Fehlertoleranz durch Redundanz



03. Juni 2019

Kapitel IX

Statische Programmanalyse



13. Mai 2019

Kapitel V

Härtung v. Daten- & Kontrollflüssen



24. Juni 2019

Kapitel X

Verifikation nicht-funkt. Eigenschaften



20. Mai 2019

Kapitel VI

Fehlerinjektion



01. Juli 2019

Kapitel XI

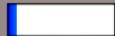
Verifikation funktionaler Eigenschaften



08. Juli 2019

Kapitel XII

Fallstudie: Sizewell B



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!

