Verlässliche Echtzeitsysteme

Zusammenfassung

Peter Ulbrich

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)

`www4.informatik.uni-erlangen.de`

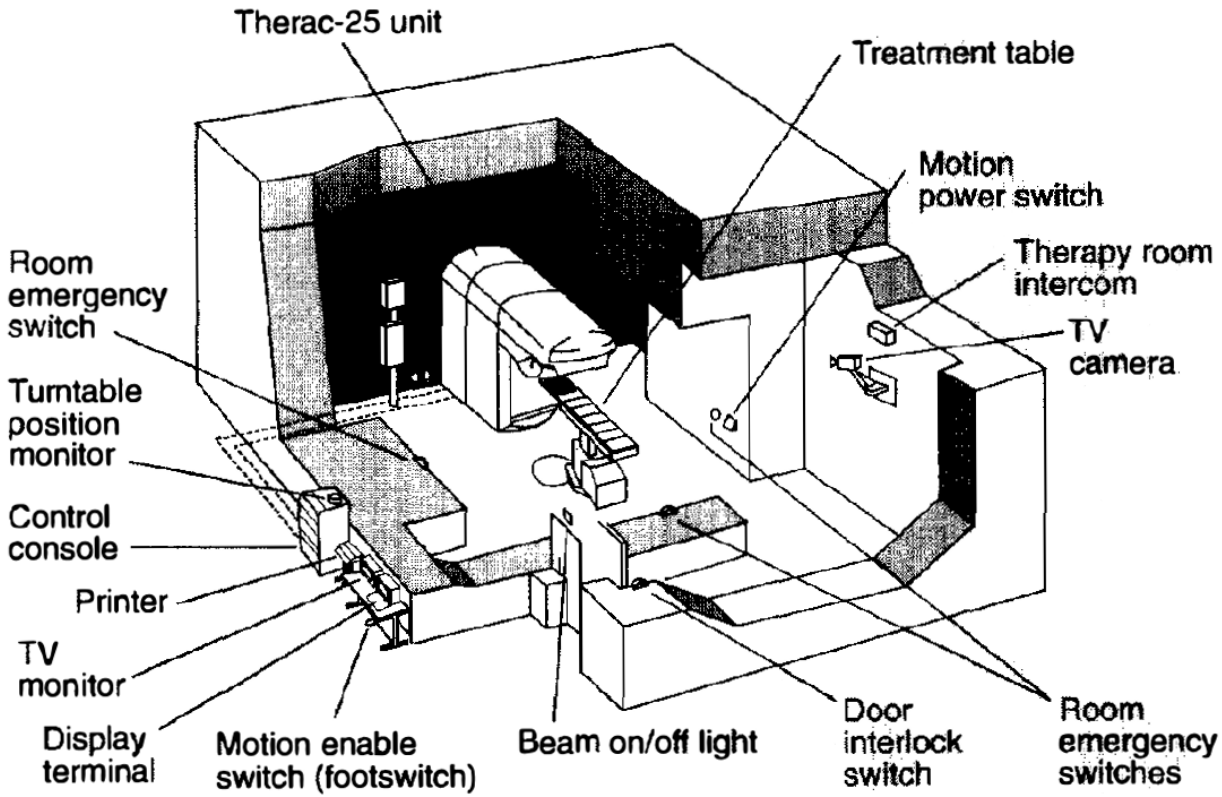
14. Juli 2015

Überblick

Einleitung

Einleitung

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



(Quelle: Nancy Leveson)

Ziel: zuverlässiger Betrieb, minimierte Ausfallwahrscheinlichkeit

Überblick

14. April 2015 *Kapitel II*

Einleitung

21. April 2015 *Kapitel III*

Grundlagen

Überblick

14. April 2015 *Kapitel II*

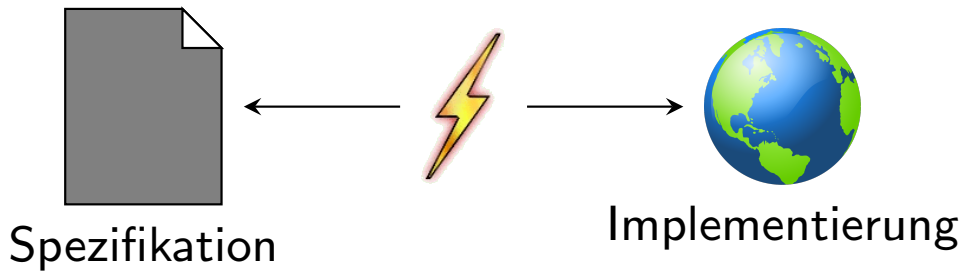
Einleitung

21. April 2015 *Kapitel III*

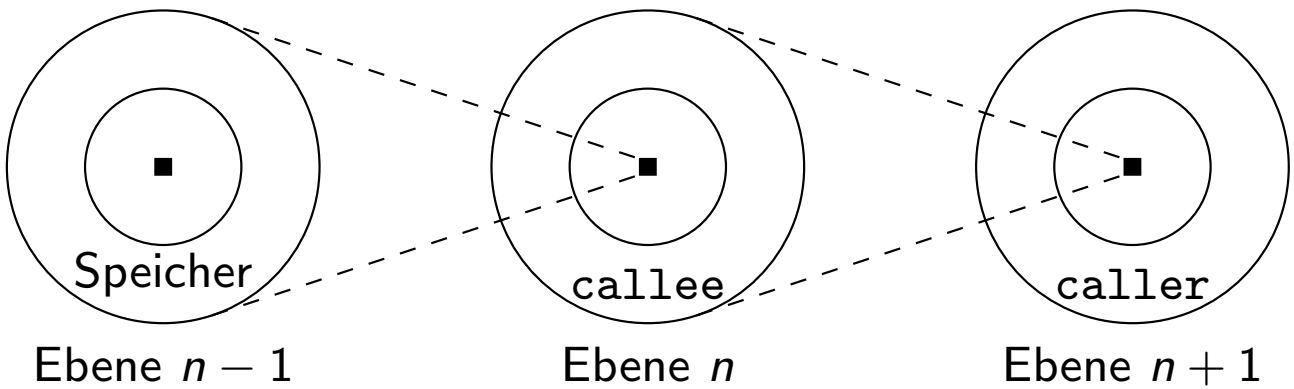
Softwaredefekte ← **Grundlagen** → **Fehlertolerenz**

Grundlagen

- **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 Fehlverhaltens!

Grundlagen (Forts.)

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

Überblick

14. April 2015 *Kapitel II*

Einleitung

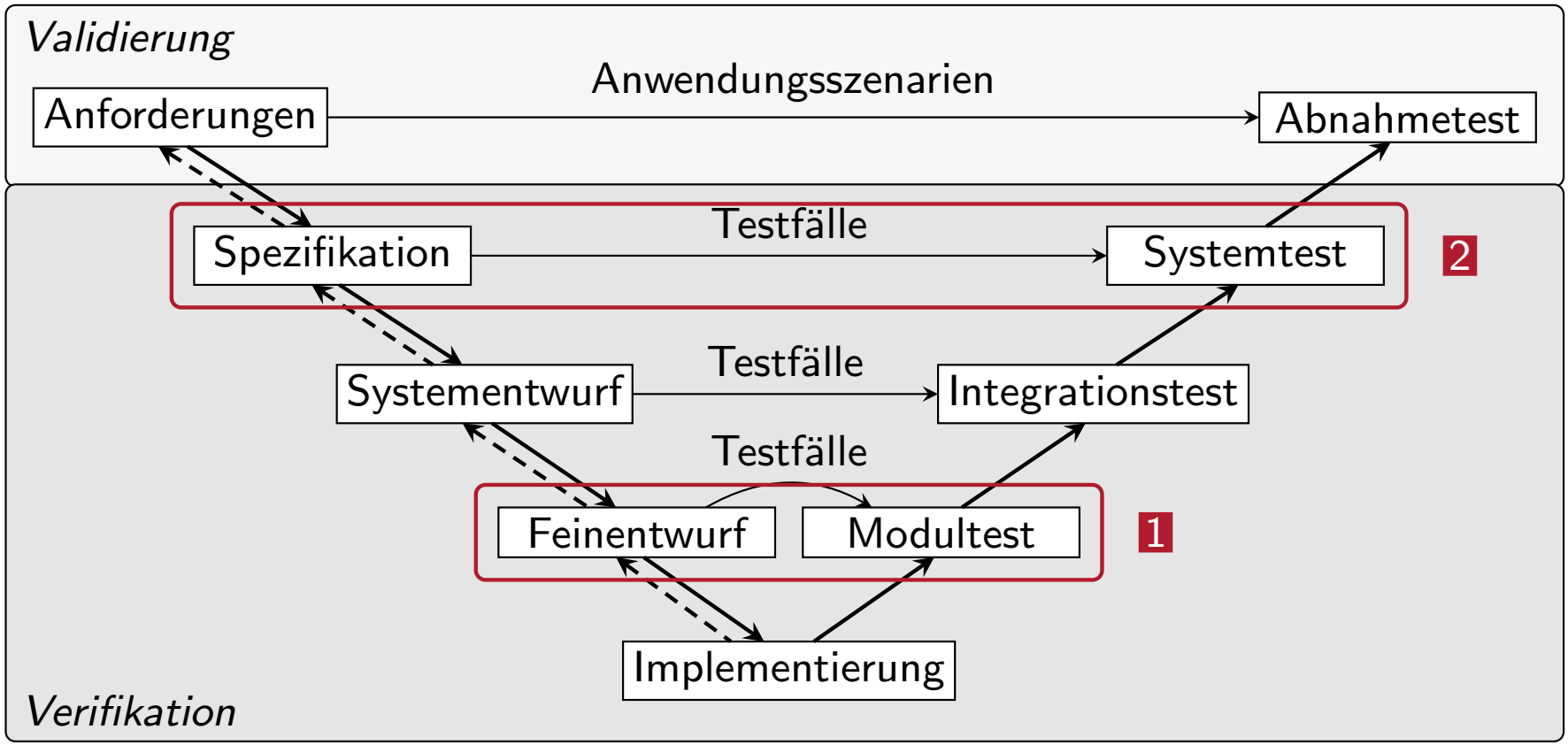
21. April 2015 *Kapitel III*

Softwaredefekte ← **Grundlagen** → **Fehlertolerenz**

28. April 2015 *Kapitel IV*

Dynamisches Testen

Testen



1. Modultests $\rightsquigarrow$ Grundbegriffe und Problemstellung
   → Black- vs. White-Box, Testüberdeckung
2. Systemtest $\rightsquigarrow$ Testen verteilter Echtzeitsysteme
   → Problemstellung und Herausforderungen

Testen (Forts.)

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 $\rightsquigarrow$ 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

Überblick

14. April 2015 *Kapitel II*

Einleitung

21. April 2015 *Kapitel III*

Softwaredefekte ← **Grundlagen** → **Fehlertolerenz**

28. April 2015 *Kapitel IV*

Dynamisches Testen

05. Mai 2015 *Kapitel V*

Statische Programmanalyse

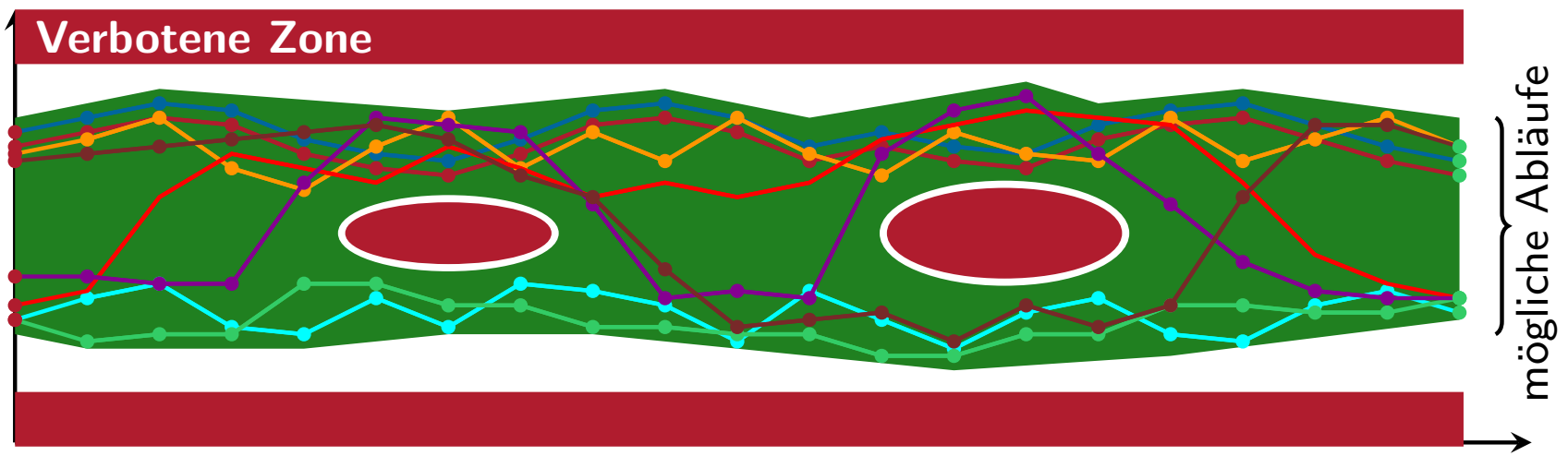
Abstrakte Interpretation

- 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 effizient 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 abstrakter Interpretation
  - Formalisierung durch Galoiseinbettungen, geordnete Mengen & Verbände
  - Optimierung durch Widening-Operator

  → Kein Bestandteil der Prüfung

Überblick

14. April 2015 — *Kapitel II*

Einleitung

21. April 2015 — *Kapitel III*

Softwaredefekte ⟵ **Grundlagen** ⟶ **Fehlertolerenz**

28. April 2015 — *Kapitel IV*

Dynamisches Testen

05. Mai 2015 — *Kapitel V*

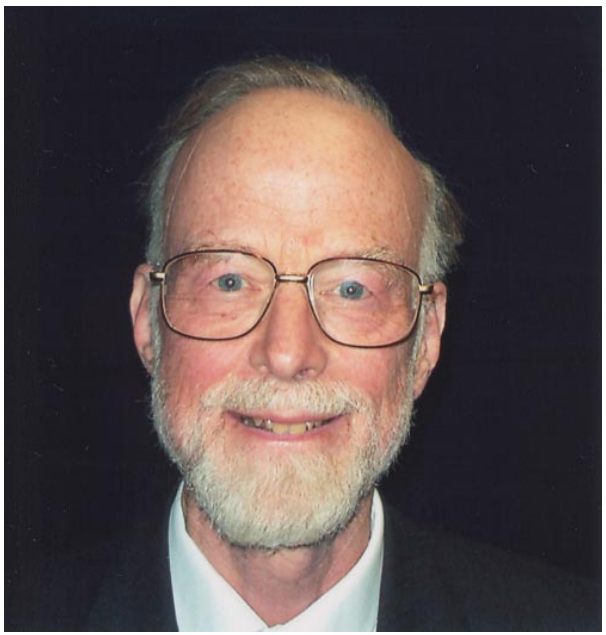
Statische Programmanalyse

12. Mai 2015 — *Kapitel VI*

Design-by-Contract

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

Prädikatentransformation $\rightsquigarrow$ 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 $\rightsquigarrow$ Astreé implementiert dieses Konzept nur teilweise!

Überblick

14. April 2015 — *Kapitel II*

Einleitung

21. April 2015 — *Kapitel III*

Softwaredefekte ← **Grundlagen** → **Fehlertolerenz**

28. April 2015 — *Kapitel IV*

Dynamisches Testen

05. Mai 2015 — *Kapitel V*

Statische Programmanalyse

12. Mai 2015 — *Kapitel VI*

Design-by-Contract

02. Juni 2015 — *Kapitel VII*

Stack- und WCET-Analyse

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

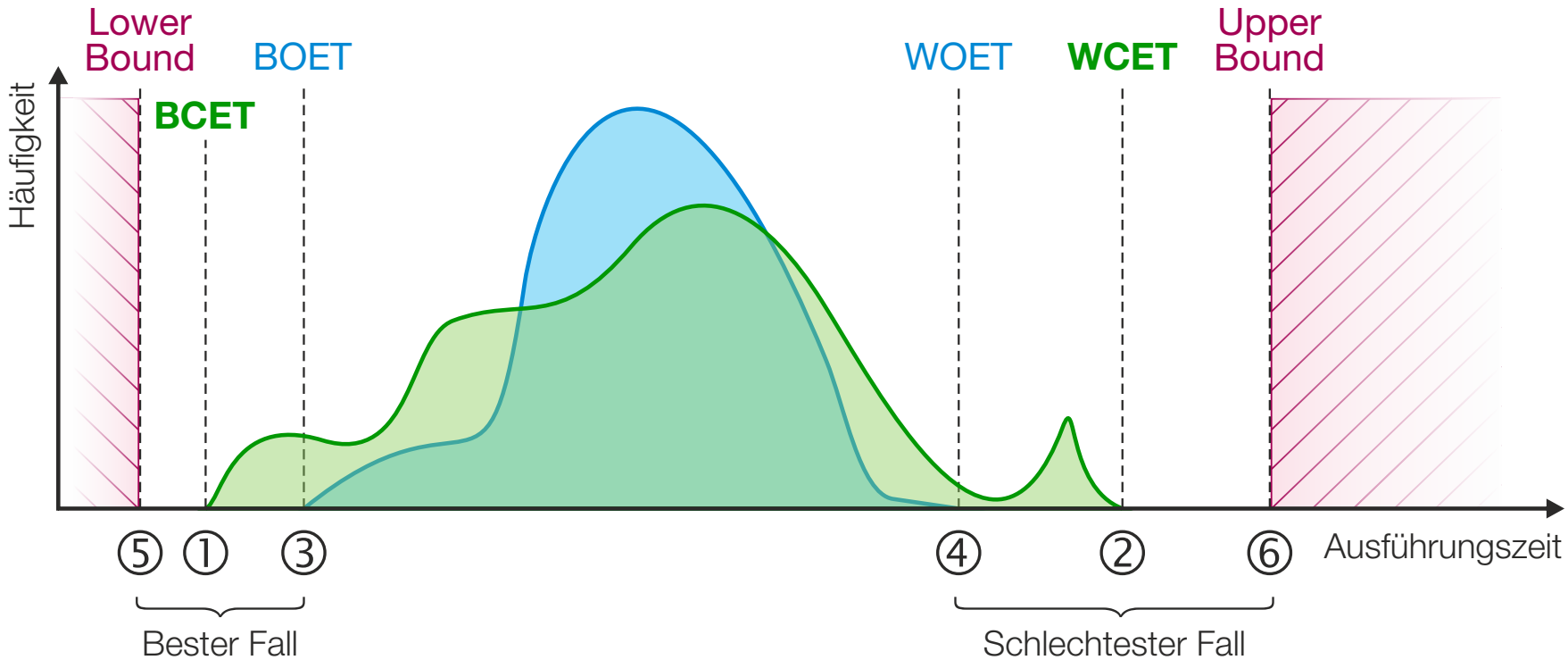
→ Reaktiv $\rightsquigarrow$ Keine Aussagen zum maximalen Verbrauch

☞ Statische Programmanalyse

- Pufferüberlauf als weitere Form von Laufzeitfehler

→ Bestimmt obere Schranke für den Speicherverbrauch

Die Laufzeit



- Messbasierte Laufzeitbestimmung $\rightsquigarrow$ Beobachtung
- Statische WCET-Analyse $\rightsquigarrow$ Obere/untere Schranke
  - Zu finden: Längster Pfad (Timing Schema, Zeitanalysegraph)
  - Dauer der Elementaroperationen: Hardware-Analyse

$\rightarrow$ Die Analyse ist sicher (sound) falls Upper Bound $\geq$ WCET

Überblick

14. April 2015 — *Kapitel II*

Einleitung

21. April 2015 — *Kapitel III*

Softwaredefekte ⟵ **Grundlagen** ⟶ **Fehlertolerenz**

28. April 2015 — *Kapitel IV*

Dynamisches Testen

05. Mai 2015 — *Kapitel V*

Statische Programmanalyse

12. Mai 2015 — *Kapitel VI*

Design-by-Contract

02. Juni 2015 — *Kapitel VII*

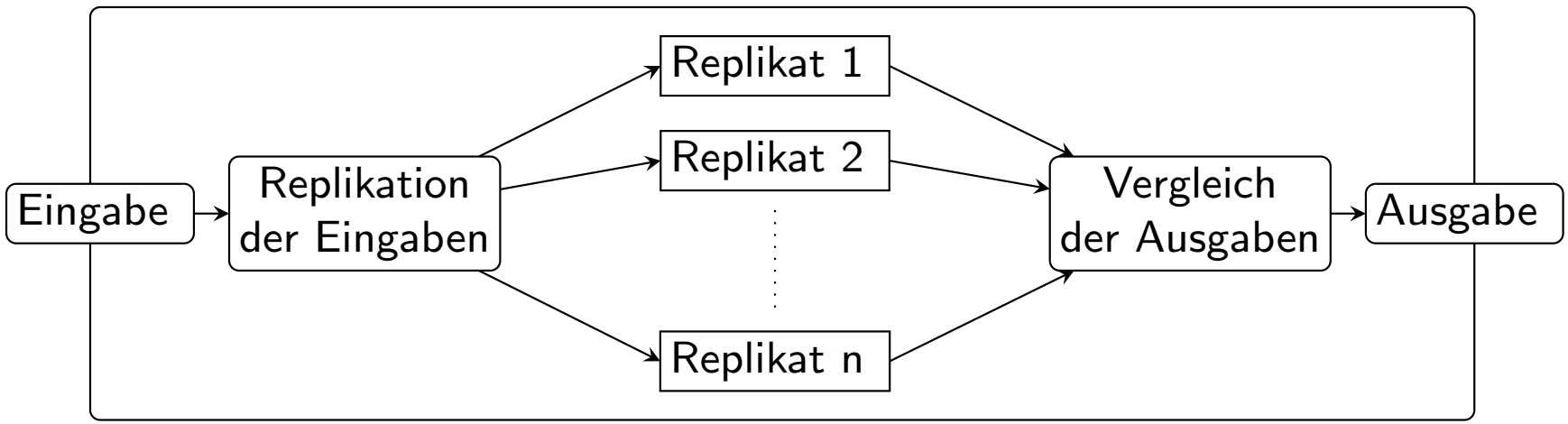
Stack- und WCET-Analyse

19. Mai 2015 — *Kapitel VIII*

Fehlertoleranz durch Redundanz

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

Redundante Ausführung (Forts.)

Fehlertypen $\mapsto$ 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 $\mapsto$ Hardwareredundanz

- 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

Überblick

14. April 2015 — *Kapitel II*

Einleitung

21. April 2015 — *Kapitel III*

Softwaredefekte ⟵ **Grundlagen** ⟶ **Fehlertolerenz**

28. April 2015 — *Kapitel IV*

Dynamisches Testen

05. Mai 2015 — *Kapitel V*

Statische Programmanalyse

12. Mai 2015 — *Kapitel VI*

Design-by-Contract

02. Juni 2015 — *Kapitel VII*

Stack- und WCET-Analyse

19. Mai 2015 — *Kapitel VIII*

Fehlertoleranz durch Redundanz

09. Juni 2015 — *Kapitel IX*

Härtung v. Daten- & Kontrollfluss

Härtung von Code & Daten

Fehlererkennung Durch arithmetische Codierung

$\rightsquigarrow$ Einsatz von Informationsredundanz durch Prüfbits

- Fehlererkennung durch Absoluttest (auch Akzeptanztest)

AN-Codierung $\rightsquigarrow$ 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

$\rightsquigarrow$ Vollständige Fehlererfassung von Operanden-, Berechnungs- und Operatorfehlern

- Codierung des Kontrollflusses $\rightsquigarrow$ Signaturen für Grundblöcke

CoRed-Ansatz $\rightsquigarrow$ selektive Anwendung der ANBD-Codierung

- Durchgehende arithmetische Codierung wäre zu teuer

Härtung von Code & Daten (Forts.)

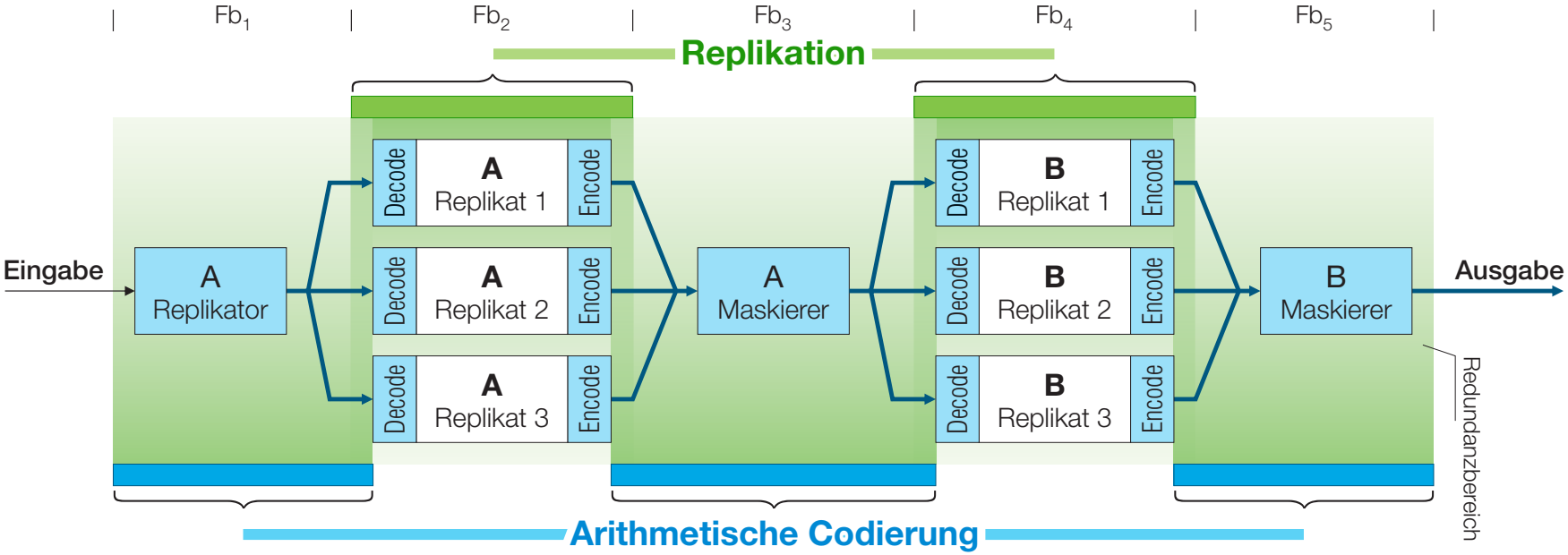
- ANBD-Codierung härtet Daten und Kontrollfluss
  - Operanden-, Berechnungs- und Operatorfehler

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

  - 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

14. April 2015 — *Kapitel II*

Einleitung

21. April 2015 — *Kapitel III*

Softwaredefekte ⟵ **Grundlagen** ⟶ **Fehlertolerenz**

28. April 2015 — *Kapitel IV*

Dynamisches Testen

05. Mai 2015 — *Kapitel V*

Statische Programmanalyse

12. Mai 2015 — *Kapitel VI*

Design-by-Contract

02. Juni 2015 — *Kapitel VII*

Stack- und WCET-Analyse

19. Mai 2015 — *Kapitel VIII*

Fehlertoleranz durch Redundanz

09. Juni 2015 — *Kapitel IX*

Härtung v. Daten- & Kontrollfluss

16. Juni 2015 — *Kapitel X*

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/reproudzierbar

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 $\mapsto$ grundlegende Kategorisierung

- {hardware, software, simulations, emulations}-basiert

FAIL* $\mapsto$ Grundlage für generische Fehlerinjektion?

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

Überblick

14. April 2015 — Kapitel II
Einleitung

21. April 2015 — Kapitel III
Softwaredefekte ⟵ **Grundlagen** ⟶ **Fehlertolerenz**

28. April 2015 — Kapitel IV
Dynamisches Testen

05. Mai 2015 — Kapitel V
Statische Programmanalyse

12. Mai 2015 — Kapitel VI
Design-by-Contract

02. Juni 2015 — Kapitel VII
Stack- und WCET-Analyse

19. Mai 2015 — Kapitel VIII
Fehlertoleranz durch Redundanz

09. Juni 2015 — Kapitel IX
Härtung v. Daten- & Kontrollfluss

16. Juni 2015 — Kapitel X
Fehlerinjektion

30. Juni 2015 — Kapitel XI
Fehlertoleranz durch Verteilung

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 $\rightsquigarrow$ Keine zeitliche Kapselung

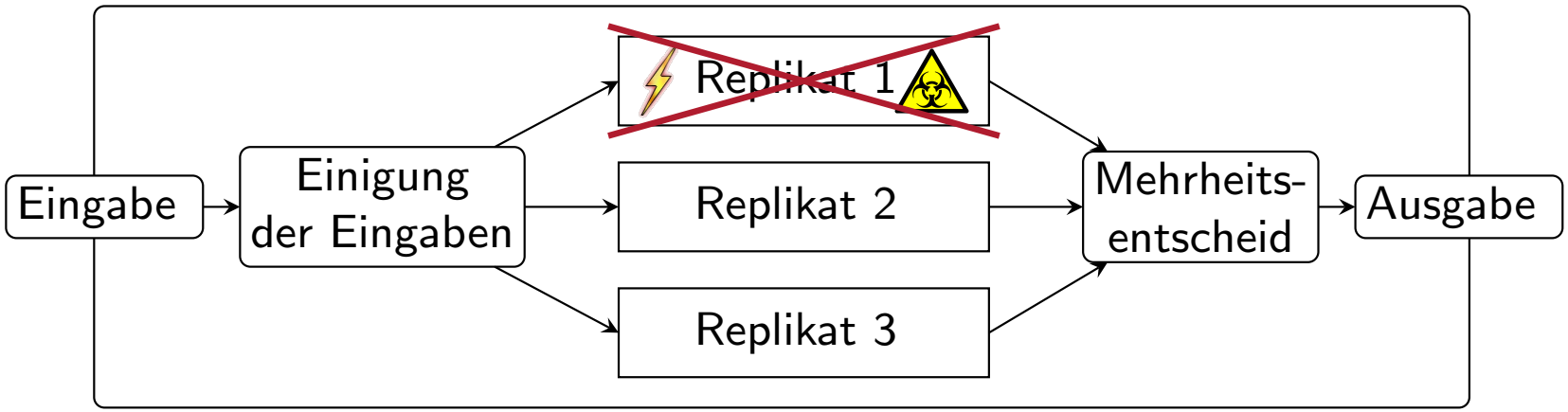
Zeitgesteuerte Kommunikation

- Auslastung vorab bestimmbar
- Zustandsnachrichten $\rightsquigarrow$ Zeitliche Kapselung

$\rightarrow$ Aufwendige Uhrensynchronisation erforderlich

Reintegration

- Ein Replikate fällt aus! $\rightsquigarrow$ **Was dann?**



  - Solange die verbliebenen Replikate korrekt arbeiten, ist alles in Ordnung.
  - Was aber, wenn sie unterschiedliche Ergebnisse liefern?
    - Welches Replikat hat recht? $\rightsquigarrow$ Patt-Situation

☞ Eine „Reparatur“ ist für einen dauerhaften Betrieb unausweichlich

1. Fehlererkennung und -diagnose
2. Rekonfiguration $\rightsquigarrow$ Isolation des fehlerhaften Knotens
3. Fehlererholung und Reintegration

Überblick

14. April 2015 — *Kapitel II*

Einleitung

21. April 2015 — *Kapitel III*

Softwaredefekte ← **Grundlagen** → **Fehlertolerenz**

28. April 2015 — *Kapitel IV*

Dynamisches Testen

05. Mai 2015 — *Kapitel V*

Statische Programmanalyse

12. Mai 2015 — *Kapitel VI*

Design-by-Contract

02. Juni 2015 — *Kapitel VII*

Stack- und WCET-Analyse

19. Mai 2015 — *Kapitel VIII*

Fehlertoleranz durch Redundanz

09. Juni 2015 — *Kapitel IX*

Härtung v. Daten- & Kontrollfluss

16. Juni 2015 — *Kapitel X*

Fehlerinjektion

30. Juni 2015 — *Kapitel XI*

Fehlertoleranz durch Verteilung

07. Juli 2015 — *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

Fallstudien (Forts.)

Sizewell B $\rightsquigarrow$ primäres Reaktorschutzsystem

- Einziger Zweck: sichere Abschaltung des Reaktors

Redundanz $\rightsquigarrow$ Absicherung gegen Systemausfälle

- Vierfach

Diversität $\rightsquigarrow$ Abfedern von Software-Defekten

- Unterschiedliche Hardware und Software
- Analoges Sekundärsystem

Isolation $\rightsquigarrow$ Abschottung der einzelnen Replikate

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

Verifikation $\rightsquigarrow$ umfangreiche statische Prüfung von Software

- Vielschichtiger Prozess, Betrachtung von Quell- und Binärcode

Überblick

14. April 2015 — *Kapitel II*

Einleitung

21. April 2015 — *Kapitel III*

Softwaredefekte ← **Grundlagen** → **Fehlertolerenz**

28. April 2015 — *Kapitel IV*

Dynamisches Testen

05. Mai 2015 — *Kapitel V*

Statische Programmanalyse

12. Mai 2015 — *Kapitel VI*

Design-by-Contract

02. Juni 2015 — *Kapitel VII*

Stack- und WCET-Analyse

19. Mai 2015 — *Kapitel VIII*

Fehlertoleranz durch Redundanz

09. Juni 2015 — *Kapitel IX*

Härtung v. Daten- & Kontrollfluss

16. Juni 2015 — *Kapitel X*

Fehlerinjektion

30. Juni 2015 — *Kapitel XI*

Fehlertoleranz durch Verteilung

07. Juli 2015 — *Kapitel XII*

Fallstudie: Sizewell B

23. Juni 2015 & 07. Juli 2015 — –

Vorträge: Industrie & Forschung

Gliederung

{B, M}-Arbeiten . . . Promotion

Forschungs-/Entwicklungsprojekte: Universität, Forschungseinrichtungen, Industrie

`https://www4.cs.uni-erlangen.de/Theses`

oder besser noch: Kommt vorbei!