Verlässliche Echtzeitsysteme

Zusammenfassung

Fabian Scheler

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

`www4.informatik.uni-erlangen.de`

10. Juli 2012

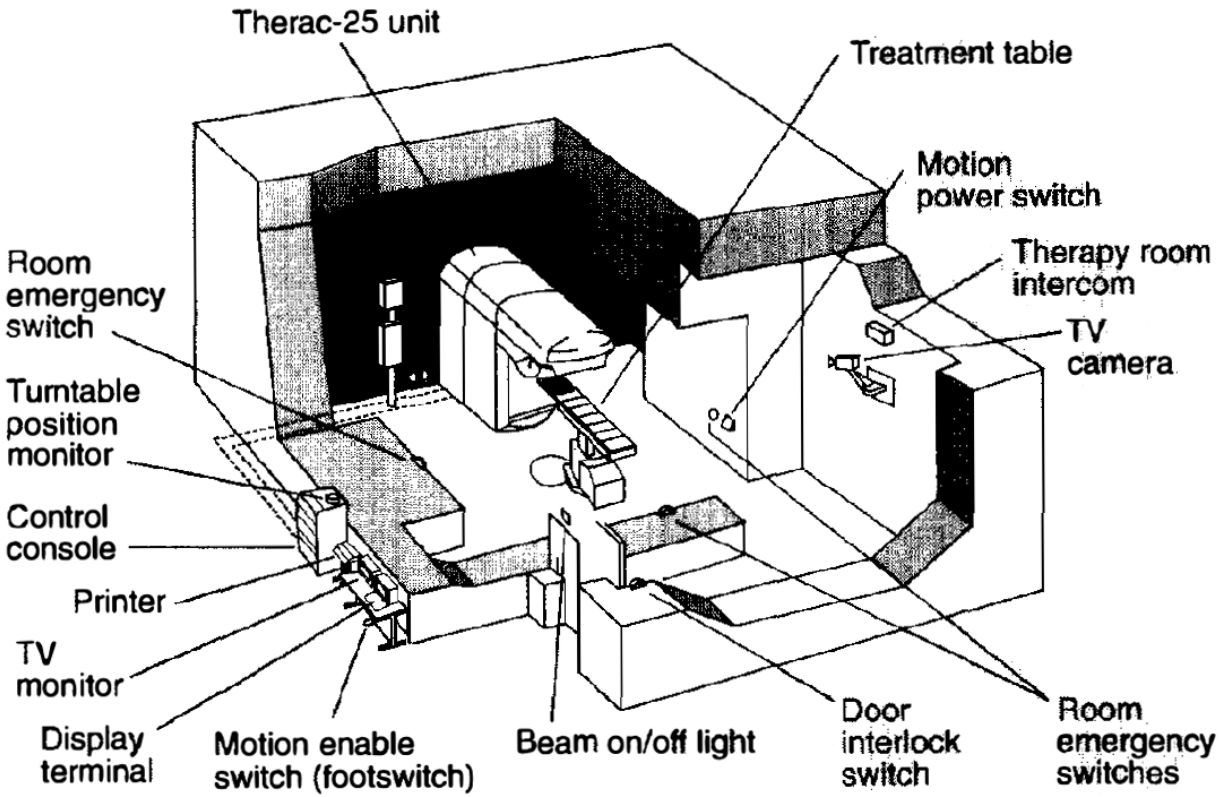
Gliederung

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

17. April 2012 *Kapitel II*

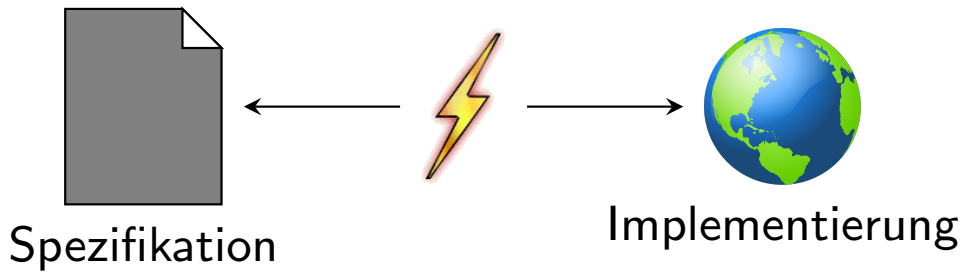
Einleitung

24. April 2012 *Kapitel III*

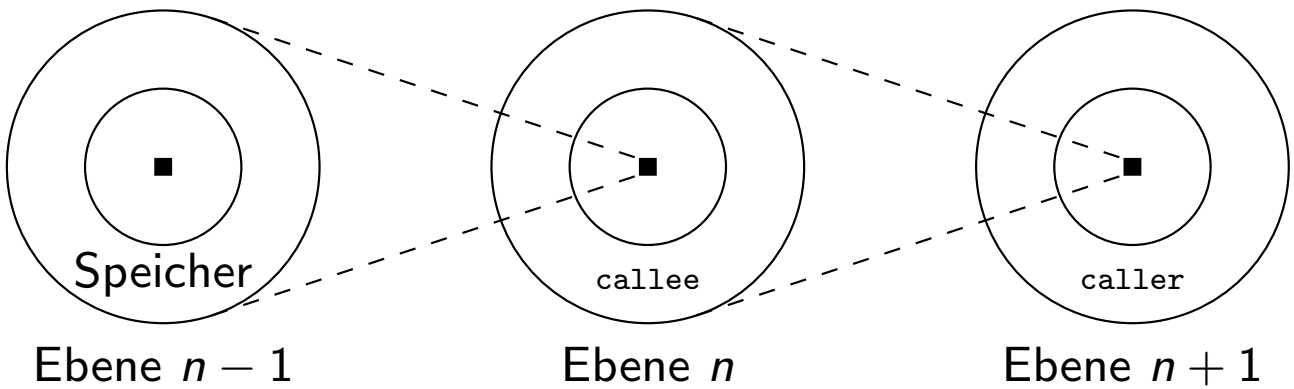
Software-Defekte ← **Grundlagen** → **transiente Fehler**

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

17. April 2012 *Kapitel II*

Einleitung

24. April 2012 *Kapitel III*

Software-Defekte ← **Grundlagen** → **transiente Fehler**

30. April 2012 *Kapitel IV*

Gesetzliche Grundlage: ISO 26262

Überblick

17. April 2012 — *Kapitel II*

Einleitung

24. April 2012 — *Kapitel III*

Software-Defekte ← **Grundlagen** → **transiente Fehler**

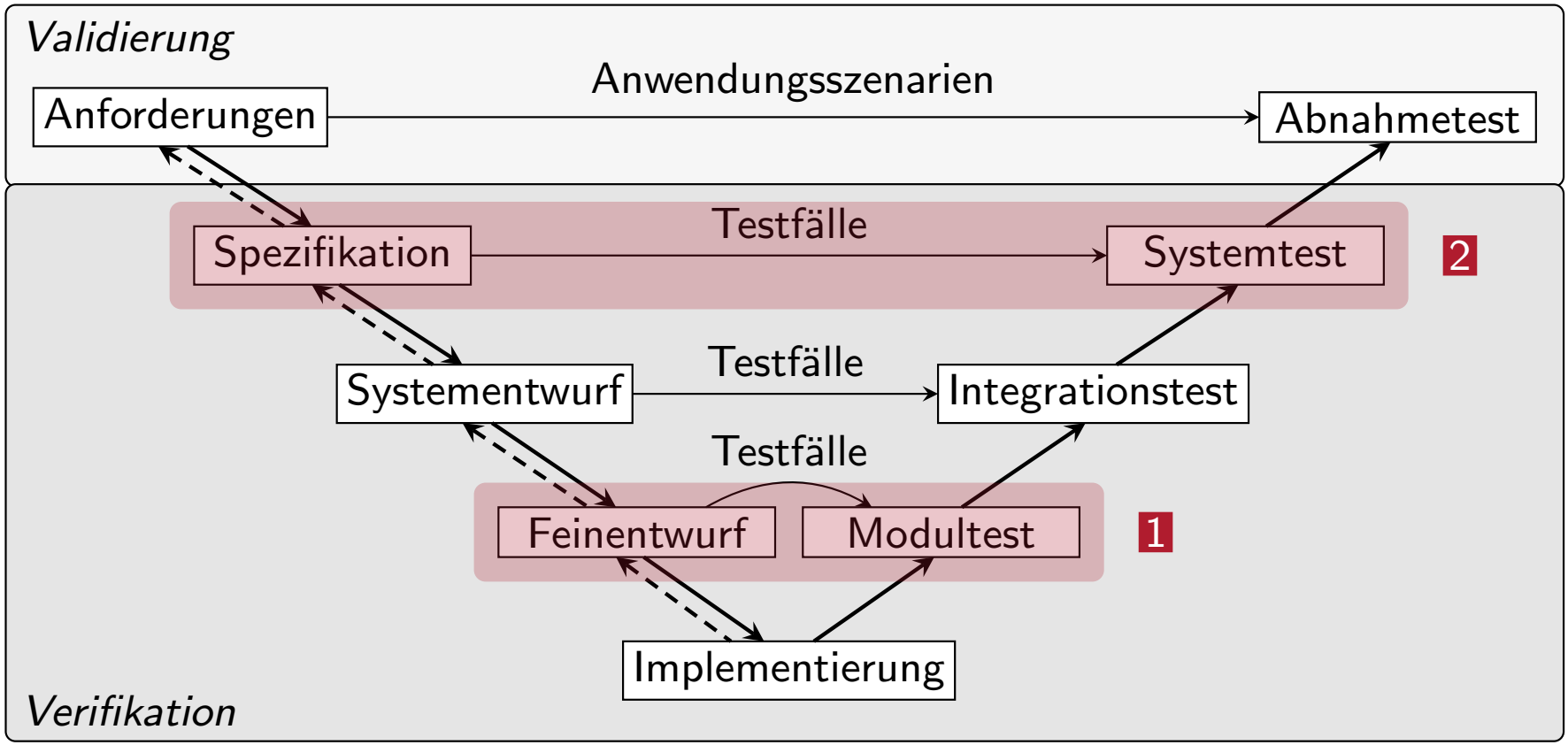
30. April 2012 — *Kapitel IV*

Gesetzliche Grundlage: ISO 26262

08. Mai 2012 — *Kapitel V*

Testen

Testen



1 Modultests ⇝ Grundbegriffe und Problemstellung

⇝ Black- vs. White-Box, Testüberdeckung

2 Systemtest ⇝ 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

17. April 2012 — *Kapitel II*

Einleitung

24. April 2012 — *Kapitel III*

Software-Defekte ← **Grundlagen** → **transiente Fehler**

30. April 2012 — *Kapitel IV*

Gesetzliche Grundlage: ISO 26262

08. Mai 2012 — *Kapitel V*

Testen

22. Mai 2012 — *Kapitel VI*

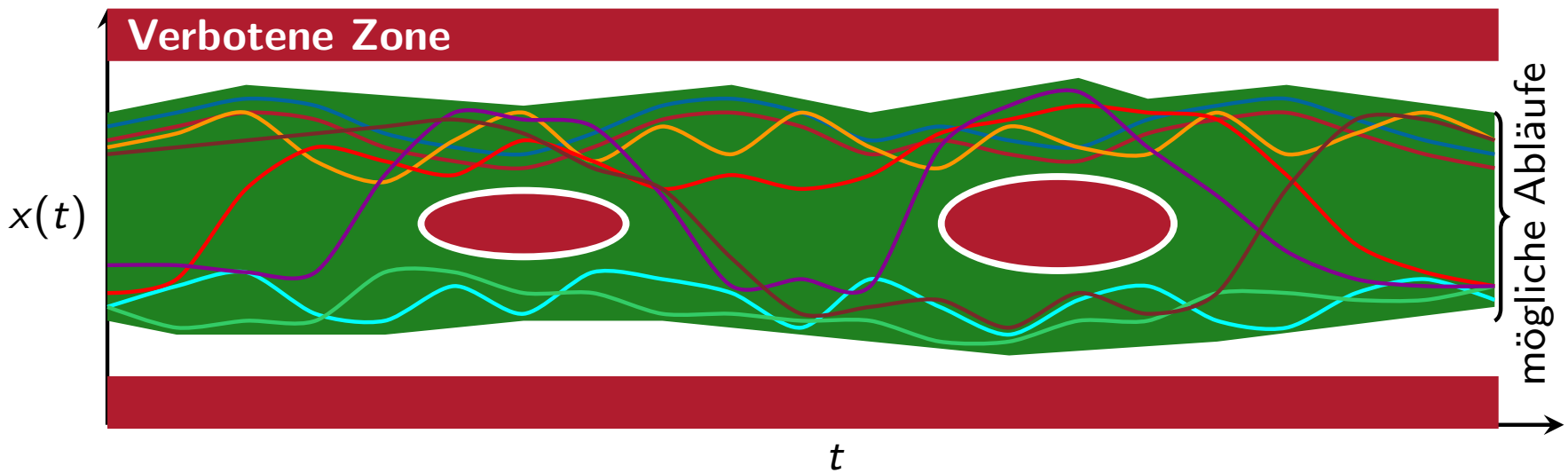
Abstrakte Interpretation

Abstrakte Interpretation

- **Ziel:** Enthält das Programm Software-Defekte?
  - 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.



Abstrakte Interpretation (Forts.)

Konkrete Programmsemantik ist nicht berechenbar

$\leadsto$ Approximation durch eine abstrakte Semantik

- Korrektheit der Approximation ist entscheidend
  - nur so kann man einen Sicherheitsnachweis führen
- die Approximation muss präzise sein
  - nur so kann man Fehlalarme vermeiden
- die Approximation darf nicht zu komplex sein
  - nur so kann sie effizient berechnet werden

Transitionssystem beschreiben Programme

- Pfadsemantiken beschreiben die konkrete Programmsemantik
- Approximation durch Pfadpräfixe und Sammelsemantik

  $\leadsto$ abstrakte Interpretation approximiert die Sammelsemantik

Mathematische Grundlagen abstrakter Interpretation

- (vollständig) partiell geordnete Mengen, Verbände
- Galoiseinbettungen, lokale konsistente Funktionen, Widening
- Intervallabstraktion

Überblick

17. April 2012 — *Kapitel II*

Einleitung

24. April 2012 — *Kapitel III*

Software-Defekte ← **Grundlagen** → **transiente Fehler**

30. April 2012 — *Kapitel IV*

Gesetzliche Grundlage: ISO 26262

08. Mai 2012 — *Kapitel V*

Testen

22. Mai 2012 — *Kapitel VI*

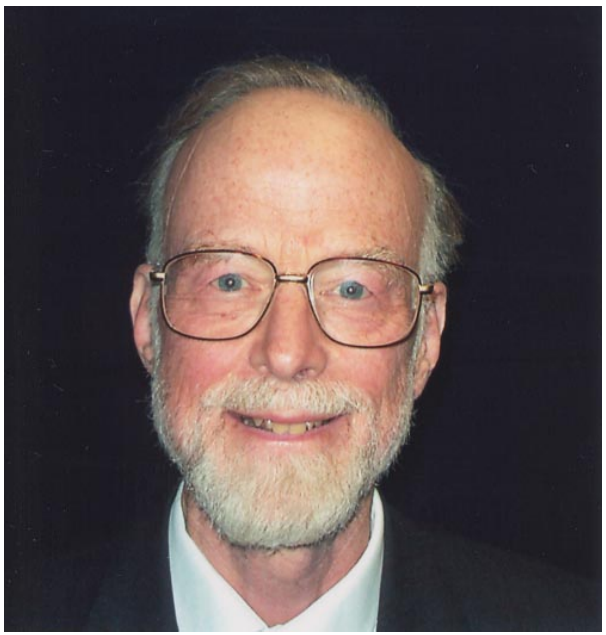
Abstrakte Interpretation

5. Juni 2012 — *Kapitel VII*

WP-Kalkül

WP-Kalkül

- 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

WP-Kalkül (Forts.)

Funktionale Programmeigenschaften $\mapsto$ Zusicherungen

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

Prädikatentransformation $\leadsto$ 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“

- wird von Frama-C in den Plug-Ins WP und Jessie implementiert

Grenzen des WP-Kalküls

Überblick

17. April 2012 — *Kapitel II*

Einleitung

24. April 2012 — *Kapitel III*

Software-Defekte ← **Grundlagen** → **transiente Fehler**

30. April 2012 — *Kapitel IV*

Gesetzliche Grundlage: ISO 26262

08. Mai 2012 — *Kapitel V*

Testen

12. Juni 2012 — *Kapitel VIII*

Redundante Ausführung

22. Mai 2012 — *Kapitel VI*

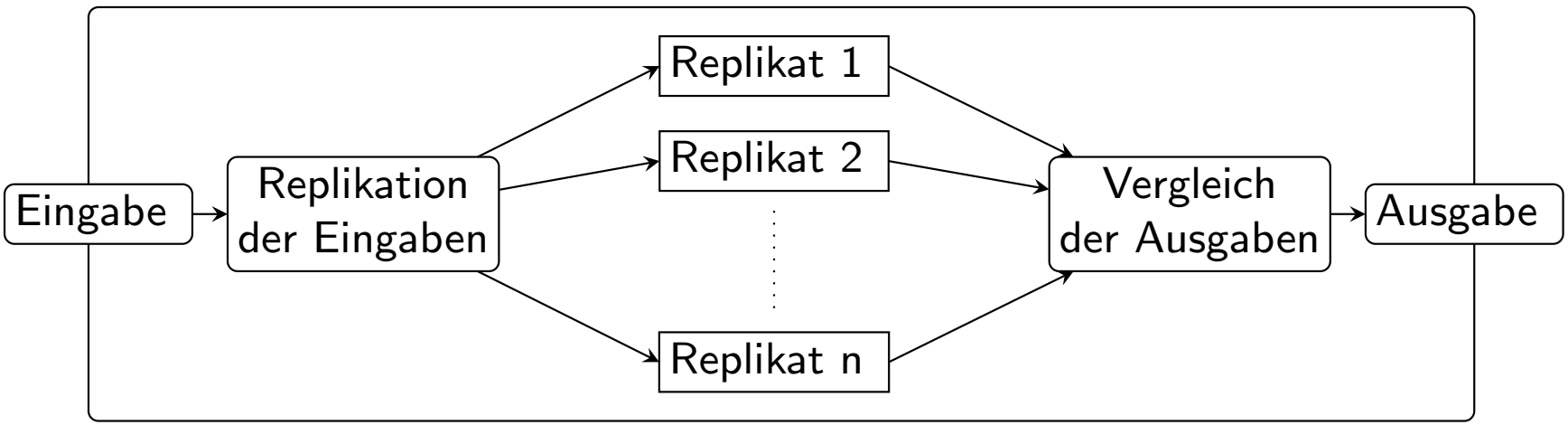
Abstrakte Interpretation

5. Juni 2012 — *Kapitel VII*

WP-Kalkül

Redundante Ausführung

- Fehlertoleranz erfordert Redundanz
  - räumliche, zeitliche oder funktionale Redundanz
- Maskierung von Fehlern durch redundante Ausführung
  - 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

- Grundvoraussetzung für Fehlertoleranz
- räumlich, zeitlich, funktional, {hot, warm, cold} standby
- Fehlererkennung, -diagnose, -eindämmung, -maskierung

Replikation $\mapsto$ koordinierter Einsatz von Redundanz

- Replikation der Eingaben, Abstimmung der Ausgaben
- Replikate für fail-silent, fail-consistent, malicious
- 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

17. April 2012 — *Kapitel II*

Einleitung

24. April 2012 — *Kapitel III*

Software-Defekte ← **Grundlagen** → **transiente Fehler**

30. April 2012 — *Kapitel IV*

Gesetzliche Grundlage: ISO 26262

08. Mai 2012 — *Kapitel V*

Testen

12. Juni 2012 — *Kapitel VIII*

Redundante Ausführung

22. Mai 2012 — *Kapitel VI*

Abstrakte Interpretation

19. Juni 2012 — *Kapitel IX*

Härtung von Code & Daten

5. Juni 2012 — *Kapitel VII*

WP-Kalkül

Härtung von Code & Daten

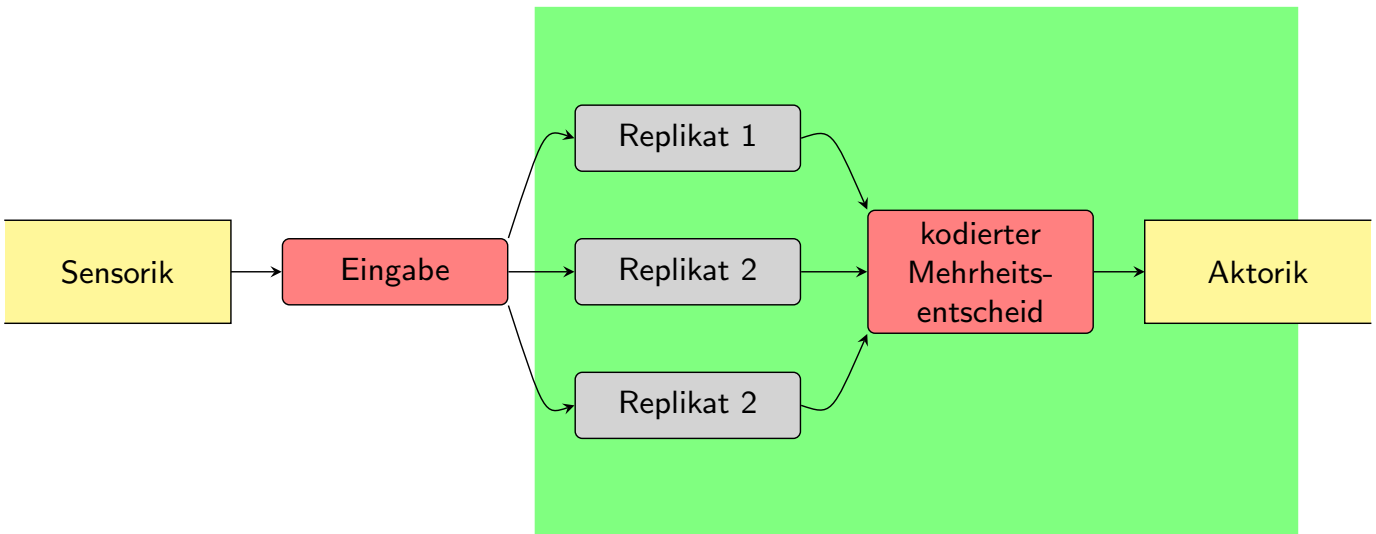
- ANBD-Kodierung härtet Daten und Kontrollfluss
  - Operanden-, Berechnungs- und Operatorfehler

$$x_c = Ax + B_x + D; \quad A > 1 \wedge B_x + D < A$$

    - Signatur B_x und Zeitstempel D

⇝ **Nachteil:** enorme hohe Laufzeitkosten

☞ „Combined Redundancy“ ⇝ ANBD-Kodierung selektiv anwenden



- sichert den „single point of failure“ replizierter Ausführung

  ⇝ kodierte Implementierung des Mehrheitsentscheids

Härtung von Code & Daten (Forts.)

Fehlererkennung möglichst ohne redundante Ausführung

- Erkennung von Operanden-, Berechnungs- und Operatorfehlern

$\rightsquigarrow$ Einsatz räumlicher Redundanz durch Prüfbits

arithmetisch Kodierung

- (nicht-)systematisch und (nicht-)separiert

AN-Kodierung $\rightsquigarrow$ Fehler im Wertbereich

- Kodierung: Multiplikation mit einem konstanten Faktor A
- kodierte Addition, Subtraktion, Multiplikation, Division
- Aussagenlogik, Schiebeoperatoren, Fließkommaarithmetik

ANBD-Kodierung erweitert die AN-Kodierung

- um statische Signaturen und dynamische Zeitstempel
- Kodierung des Kontrollflusses $\rightsquigarrow$ Signaturen für Grundblöcke

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

- durchgehende arithmetische Kodierung wäre zu teuer

Überblick

17. April 2012 — *Kapitel II*

Einleitung

24. April 2012 — *Kapitel III*

Software-Defekte ← **Grundlagen** → **transiente Fehler**

30. April 2012 — *Kapitel IV*

Gesetzliche Grundlage: ISO 26262

08. Mai 2012 — *Kapitel V*

Testen

22. Mai 2012 — *Kapitel VI*

Abstrakte Interpretation

5. Juni 2012 — *Kapitel VII*

WP-Kalkül

12. Juni 2012 — *Kapitel VIII*

Redundante Ausführung

19. Juni 2012 — *Kapitel IX*

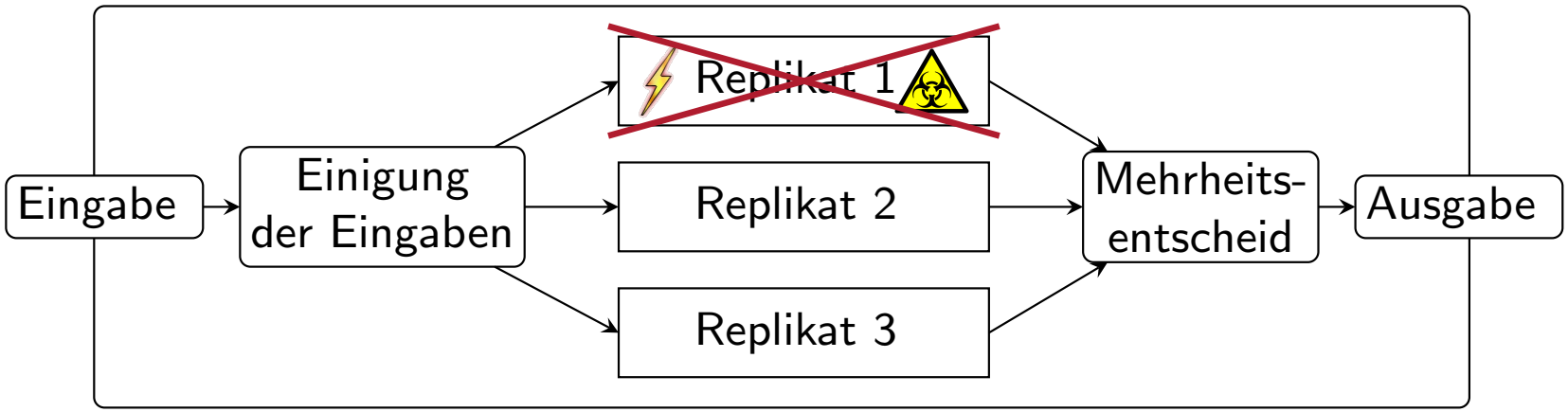
Härtung von Code & Daten

26. Juni 2012 — *Kapitel X*

Reintegration

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

Reintegration (Forts.)

Problemstellung $\mapsto$ ein Replikat fällt aus

- dies führt zu einer verminderten Fehlertoleranz

$\rightsquigarrow$ Reintegration des ausgefallene Knotens

Grundlagen für die Reintegration

- reaktiv, proaktiv und reaktiv-proaktiv
- Vorwärts- und Rückwartsbewegung
- Initialzustand und dynamischer Zustand
- Bestandteile und Minimierung des dynamischen Zustands

„Recovery Blocks“ Reintegration durch Rückwärtsbewegung

- „Distributed Recovery Blocks“ $\mapsto$ parallele Ausführung
- vorsorgliche Fehlererholung $\rightsquigarrow$ Vorwärtsbewegung
  - Rückwärtsbewegung nur für die Fehlerbeseitigung

Zustandstransfer von einem funktionsfähigen Replikat

- „one-shot SR“ vs. Zustandstransfer über mehrere Schritte
- „Running SR“ vs. „Recursive SR“

Überblick

17. April 2012 — *Kapitel II*

Einleitung

24. April 2012 — *Kapitel III*

Software-Defekte ← **Grundlagen** → **transiente Fehler**

30. April 2012 — *Kapitel IV*

Gesetzliche Grundlage: ISO 26262

08. Mai 2012 — *Kapitel V*

Testen

12. Juni 2012 — *Kapitel VIII*

Redundante Ausführung

22. Mai 2012 — *Kapitel VI*

Abstrakte Interpretation

19. Juni 2012 — *Kapitel IX*

Härtung von Code & Daten

5. Juni 2012 — *Kapitel VII*

WP-Kalkül

26. Juni 2012 — *Kapitel X*

Reintegration

03. Juli 2012 — *Kapitel XI*

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

Xception $\mapsto$ ein Werkzeug für SWIFI

- Verwendung von prozessorspezifischen Ausnahmen
- vielseitige Fehlerinjektion (Stelle, Fehlertyp, Auslösung)

FAIL* $\mapsto$ Grundlage für generische Fehlerinjektion?

- basierend auf virtuellen Zielsystemen
- flexible Plattform für Fehlerinjektion
- schnelle Experimentdurchführung durch Parallelisierung

Überblick

17. April 2012 — *Kapitel II*

Einleitung

24. April 2012 — *Kapitel III*

Software-Defekte ← **Grundlagen** → **transiente Fehler**

30. April 2012 — *Kapitel IV*

Gesetzliche Grundlage: ISO 26262

08. Mai 2012 — *Kapitel V*

Testen

12. Juni 2012 — *Kapitel VIII*

Redundante Ausführung

22. Mai 2012 — *Kapitel VI*

Abstrakte Interpretation

19. Juni 2012 — *Kapitel IX*

Härtung von Code & Daten

5. Juni 2012 — *Kapitel VII*

WP-Kalkül

26. Juni 2012 — *Kapitel X*

Reintegration

03. Juli 2012 — *Kapitel XI*

Fehlerinjektion

10. Juli 2012 — *Kapitel XII*

Fallstudien

Fallstudien

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

☞ Betrachtung zweier Fallstudien

- primäres Reaktorschutzsystem „Sizewell B“
- digitale Flugsteuerung Airbus A320/A330/A340

Fallstudien (Forts.)

Sizewell B $\rightsquigarrow$ primäres Reaktorschutzsystem

- einziger Zweck: sichere Abschaltung des Reaktors

Airbus $\rightsquigarrow$ digitale „Fly-by-Wire“-Flugsteuerung

- die Lenkung moderner Verkehrsflugzeuge

Redundanz $\rightsquigarrow$ Absicherung gegen Systemausfälle

- bis 7-fach redundante Systeme

Diversität $\rightsquigarrow$ Abfedern von Software-Defekten

- unterschiedliche Hardware und Software

Isolation $\rightsquigarrow$ Abschottung der einzelnen Replikate

- technisch $\mapsto$ optische Kommunikationsmedien
- zeitlich $\mapsto$ nicht-gekoppelte, eigenständige Rechner
- räumlich $\mapsto$ verschiedene Aufstellorte und Kabelrouten

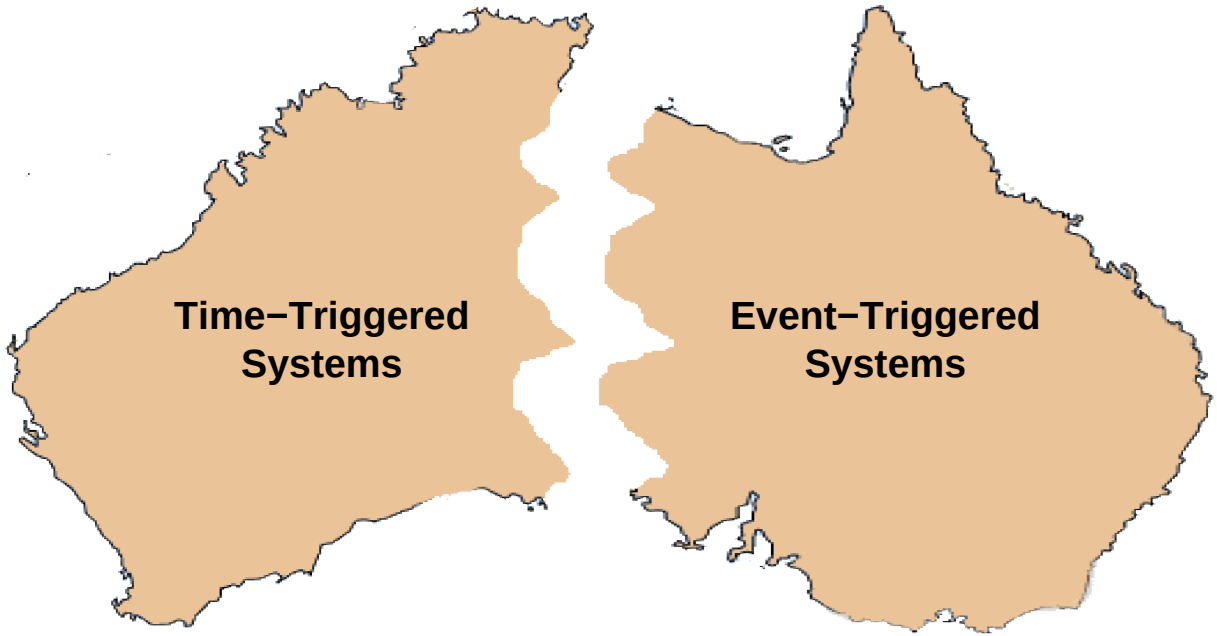
Verifikation $\rightsquigarrow$ umfangreiche statische Prüfung von Software

- vielschichtiger Prozess, Betrachtung von Quell- und Binärcode

Gliederung

Aspektorientierte Echtzeitsystemarchitekturen

Zeit- und ereignisgesteuerte Echtzeitsysteme sind grundverschieden



. . . wenn es um die Implementierung von Abhängigkeiten geht

- implizit sichergestellt
  - statische Ablaufplanung
- explizit sichergestellt
  - Schlossvariablen, Semaphore
  - Nachrichten
  - . . .

Aspektorientierte Echtzeitsystemarchitekturen (Forts.)

☞ Auswirkungen auf die Implementierung von Echtzeitanwendungen!

Fadenabstraktion

- taktgesteuerte Systemen: einfach Ereignisbehandlungen
- vorranggesteuerte Systemen: komplexe Ereignisbehandlungen

Portabilität

- Fadenabstraktionen sind mit der Anwendung verwoben
- Fäden ...
  - sperren Schlossvariablen
  - versenden Nachrichten
  - warten auf Signale anderer Fäden
- oder laufen einfach nur durch (engl. *run-to-completion*)

☞ Portierung zwischen Takt-/Vorrangsteuerung ist sehr schwierig!

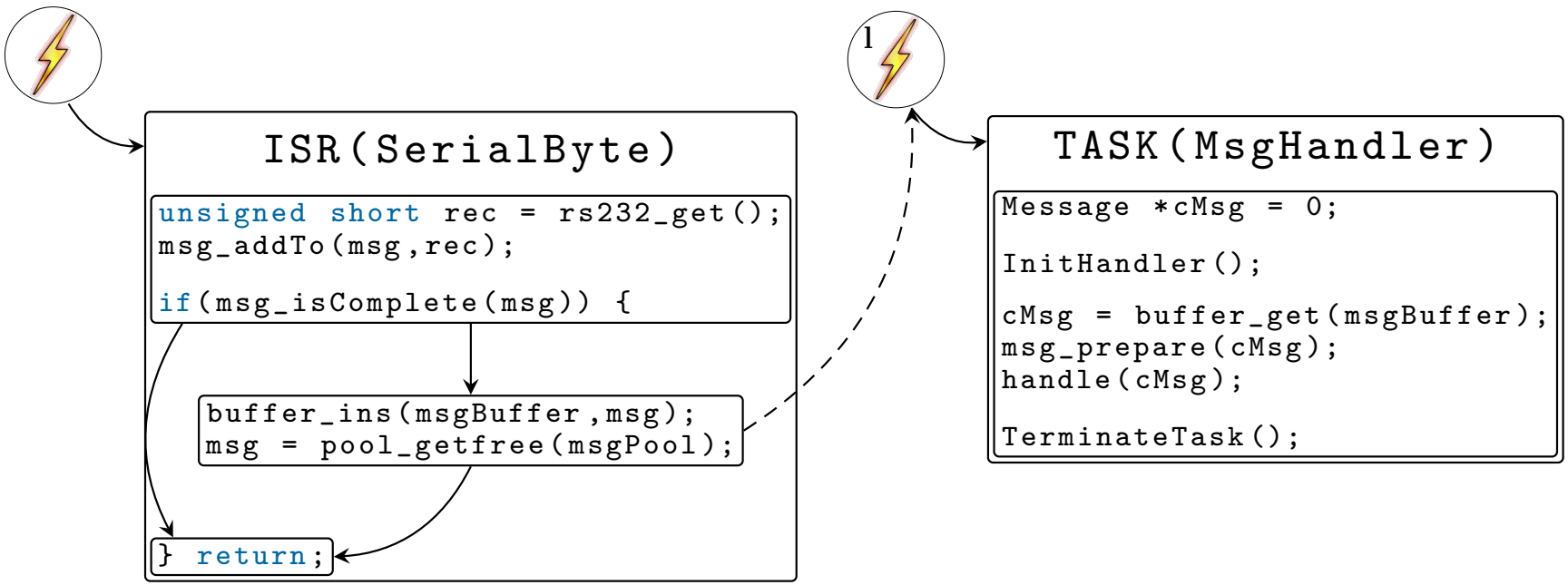
Atomic Basic Blocks (ABBs)

Abstrahieren von den Eigenheiten der Echtzeitsystemarchitektur

- stelle Abhängigkeiten unabhängig von der Fadenabstraktion dar
- Abbildung auf
  - taktgesteuerte Systeme oder
  - vorranggesteuerte Systeme
- Grundlage: Basisblöcke eines CFGs $\rightsquigarrow$ Atomic Basic Blocks
  - mehrere Grundblöcke werden zu einem ABB zusammengefasst
  - Abghängigkeiten verbinden ABBs $\rightsquigarrow$ ABB-Graphen
    - Datenabhängigkeiten, gerichtete und ungerichtete Abhängigkeiten
    - prinzipiell mithilfe der Echtzeitsystemarchitektur implementiert
  - ABB-Graphen überspannen mehrere Kontrollflüsse
  - im ABB: keine Abhängigkeiten zu anderen Kontrollflüssen
    - das macht sie aus Sicht anderer Kontrollflüsse „atomar“

    $\rightsquigarrow$ das ist der zweite Teil des Namens

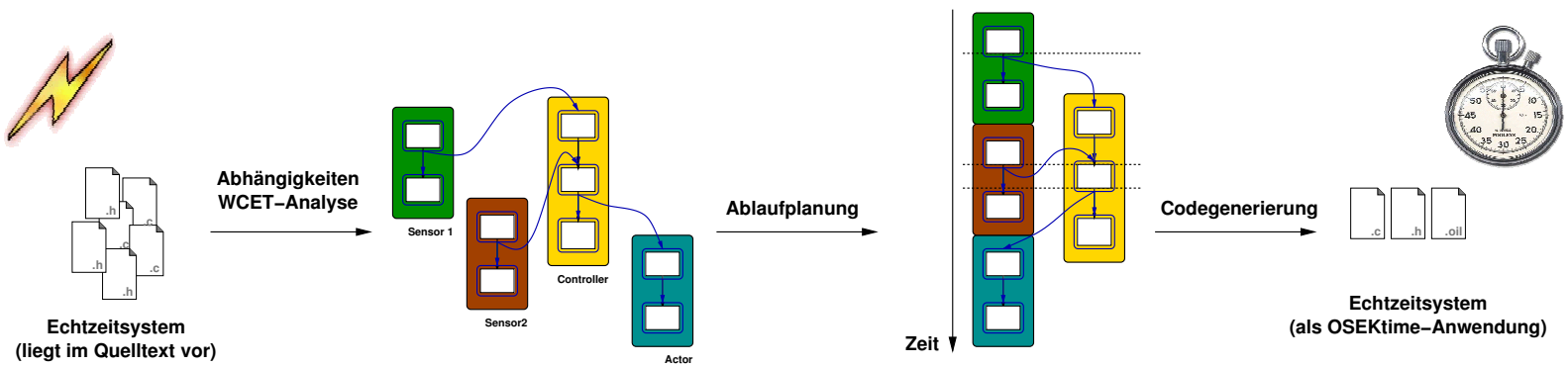
Atomic Basic Blocks (ABBs) — Beispiel



Der Real-Time Systems Compiler (RTSC)

- betriebssystemgewahrer Übersetzer
- vermittelt zwischen zeit- und ereignisgesteuerten Systemen
  - und verwendet dabei ABBs als Zwischendarstellung
- basiert auf der LLVM (Low-Level Virtual Machine)
- Gliederung wie in klassischen Compilern

Front-End	Abhängigkeitsgraph erzeugen, WCET-Analyse
Middle-End	Transformationen des ABB-Graphen, Ablaufplanung
Back-End	Codegenerierung für ein bestimmtes Betriebssystem



Forschungsziele im AORTA-Projekt

Echtzeitbetriebssysteme — Die „perfekte“ Schnittstelle?

- Semaphor, Mutex, Messages, Events, . . . $\rightsquigarrow$ riesige Auswahl
  - jedes Betriebssystem bringt seine ganz spezielle Lösung mit
- Welche Abhängigkeiten implementiert man eigentlich damit?
  - Welche Eigenschaften haben die erzeugten Abhängigkeiten?

Ereignisgesteuerte Systeme als Zielplattform — der Weg zurück

- bisher werden nur zeitgesteuerte Zielsysteme unterstützt

Verteilte Systeme bzw. Mehrkernsysteme statt Monoprozessoren

- Verteilung/Ablaufplanung auf mehreren Rechenknoten/-kernen
- Behandlung des Kommunikationssystems

Optimierung übergeordneter Eigenschaften

- z.B. Blockadezeiten durch blockierende Synchronisation

Die „perfekte RTOS-Schnittstelle“

Ausgangspunkt Schnittstellen verschiedener Echtzeitbetriebssysteme

- POSIX (QNX, . . .), eCos, Windows CE, AUTOSAR, . . .

Fokus: Kontrollflussabstraktion $\rightsquigarrow$ Fäden, Unterbrechungen, . . .

- Welche Kontrollflussabstraktionen werden angeboten?
- Wie werden sie aktiviert, wie implementieren sie Abhängigkeiten?
  - gerichtete/ungerichtete, synchrone/asynchrone Aufrufsemantik
  - Welche Informationen werden übertragen — Daten, Zeit, Signale

Ergebnis ist eine Studie:

- Welche Abhängigkeiten kann man überall implementieren?
- Gibt es Abhängigkeiten, die nicht implementiert werden können?
- Sind die Schnittstellen orthogonal oder gibt es mehrere Möglichkeiten diesselbe Abhängigkeit zu implementieren?

☞ Untermauernde Experimente/Beispiele sind wünschenswert!

RTSC: Multicore Scheduling

- Automobilindustrie: 70 - 100 Steuergeräte je Premium-KFZ

$\rightsquigarrow$ Problem:

viele Controller $\mapsto$ viele Busse $\mapsto$
viel Kupfer $\mapsto$ viel Gewicht $\mapsto$ viel Verbrauch

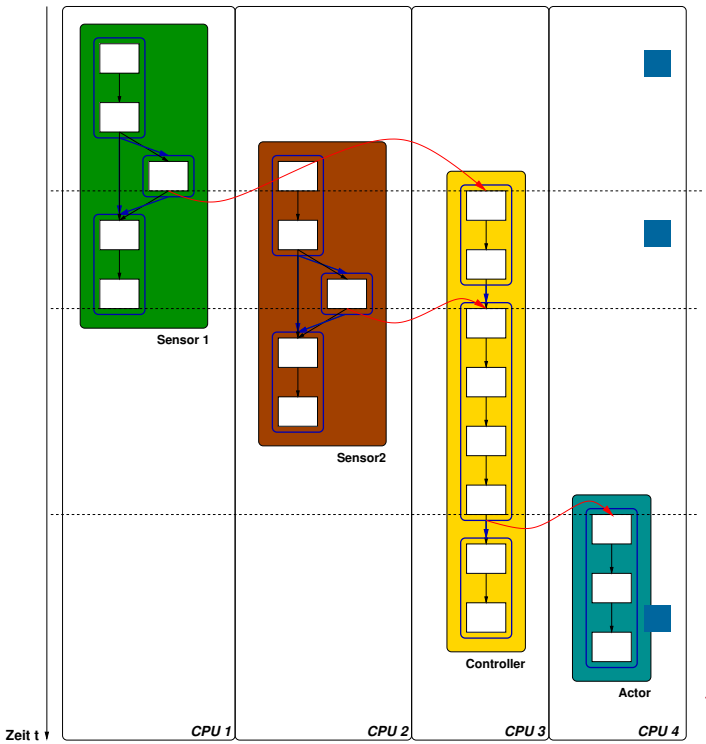
$\rightsquigarrow$ Lösung: Konsolidierung z.B. durch Mehrkernprozessoren, aber:

(1) Hardwareanforderungen – I/O
(2) Programmierung von Mehrkernprozessoren ist schwierig

- Idee: zumindest bei (2) könnte der RTSC helfen :-)
  - automatisch Abbildung von ABB-Graphen auf
    - Mehrkernsysteme bzw. verteilte Systeme
  - Implementierung eines existierenden Algorithmus
    - Peng, Shin und Abdelzaher (1997)
    - statische Allokation von *Modulen* ($\approx$ ABBs) auf Rechenknoten

  $\rightsquigarrow$ Voraussetzung ist ein Verständnis der Abhängigkeiten!

RTSC: Multicore Scheduling (Forts.)



- Abbildung auf verschiedene Knoten
  - Annahme einer globalen Zeitbasis
- Berücksichtigung von Kommunikation
  - gleicher Knoten
    - ⇝ gemeinsamen Speicher: Variablen
  - entfernter Knoten
    - ⇝ Nachrichten: TTEthernet/TTCAN
- Unterstützung von TTEthernet

⇝ weiteres Thema

RTSC: Next Generation RTSC

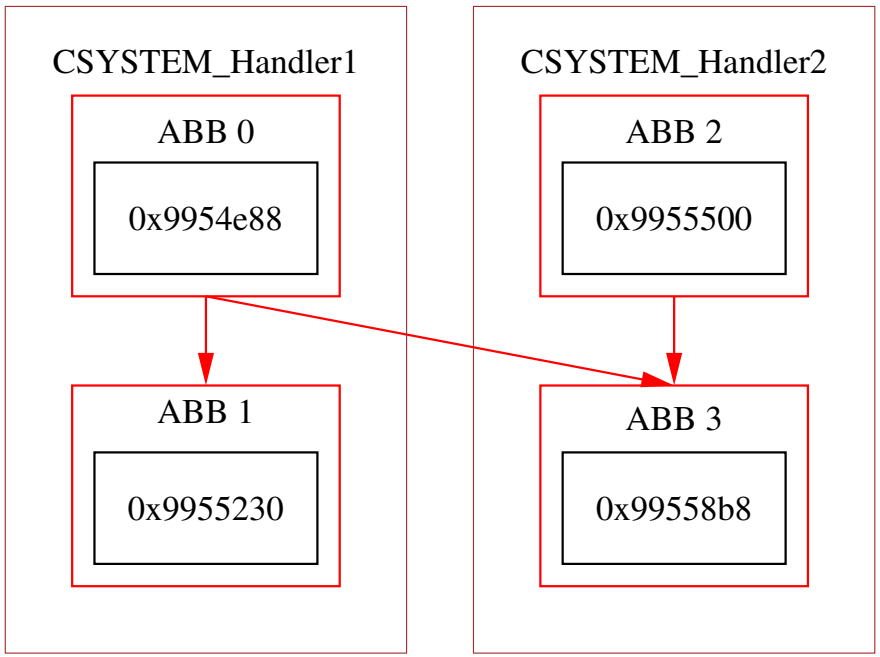
- ABBs im RTSC: Aggregation von Basisblöcken
  - implementiert als Menge von Zeigern

☞ Problem: passt nicht so ganz zu Übersetzern
  - Annahme: jede Analyse kann bei Bedarf neu berechnet werden
    - notwendig Information steckt ja im Quellcode :-)

⇝ bei ABBs ist das aber nicht der Fall :-(
  - ABBs müssen immer manuell aktualisiert werden
  - zyklische Abfolgen von Transformationen sind nicht möglich
  - ...

☞ Lösung: packe ABBs in den Quelltext :-)
  - implementiert als Aufrufe von *Magic Functions*
  - *Magic Functions* sind keine echten Funktionen
  - werden aber vom RTSC interpretiert

RTSC: Next Generation RTSC (Forts.)




```
def void @CSYSTEM_Handler1() {
  entry:
  call void RTSC_ABB_begin(0)
  ...
  call void RTSC_ABB_succ(1)
  call void RTSC_ABB_succ(3)
  call void RTSC_ABB_end(0)
  split:
  call void RTSC_ABB_begin(1)
  ...
  call void RTSC_ABB_end(1)
  ret void
}
```

Weiter Themen auf Anfrage

- Unterstützung zusätzliche Betriebssysteme
  - eCos, FreeRTOS, ROS, etherNut, ...
- Analyse von Blockadezeiten
  - Wie lange werden Fäden durch blockierende Synchronisation aufgehalten?
- Übersetzer-gestützte Replikation
  - der Übersetzer führt Replikation auf Prozessebene durch
- Verbesserung der WCET-Analyse
  - bisher: jeder Grundblock wird einzeln analysiert

  ⇝ Analyse kompletter Gruppen von Grundblöcken, Funktionen, ...
- LLVM-Optimierungen für den RTSC
  - bessere Nutzung der LLVM-Infrastruktur

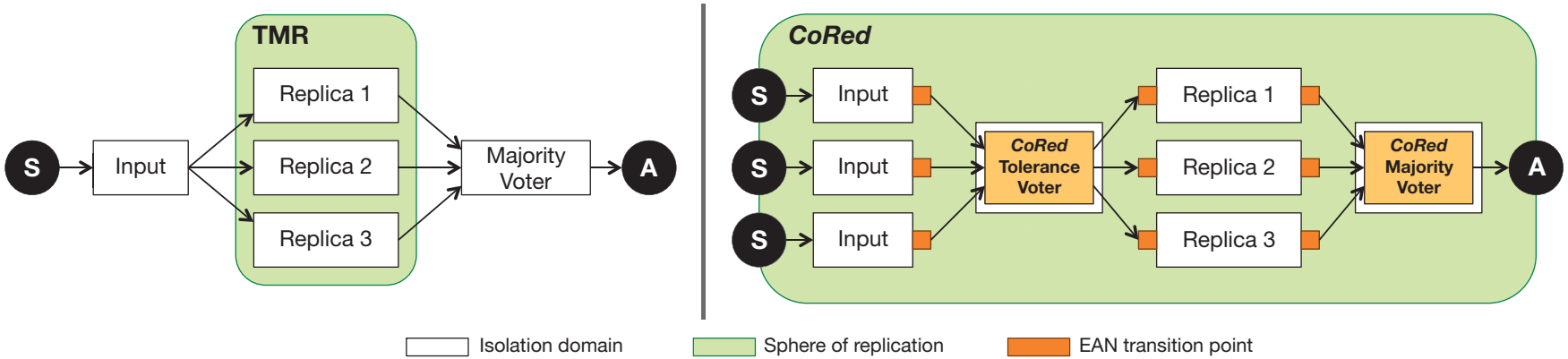
  ⇝ Wie und wo kann man LLVM-Optimierungen in den RTSC einbringen?

Verlässliche eingebettete Systeme

Problem: **Moderne Hardware wird immer unzuverlässiger**

- ⇝ Transiente Hardwarefehler (Bitkipper)
- ⇝ In sicherheitskritischen Anwendungen nicht tolerierbar

Verlässliche eingebettete Systeme



CoRed

- Absicherung auf der Ebene der Anwendung
- Selektiver Schutz sicherheitskritischer Anwendungsteile

DanceOS

- Absicherung auf der Ebene des Betriebssystems
- Gezielte Absicherung kritischer Datenstrukturen und Operationen

Zuverlässigkeit nachweisen

Fehlerraumanalyse

- Welche Bitkipper wirken sich wirklich aus?
- Einschränkung des möglichen Fehlerraums

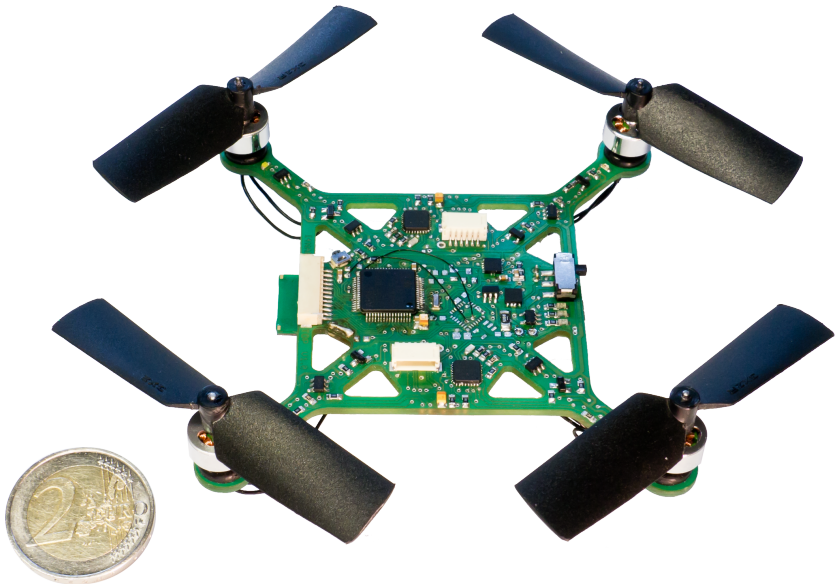
Zuverlässigkeitsanalyse mittels symbolischer Ausführung

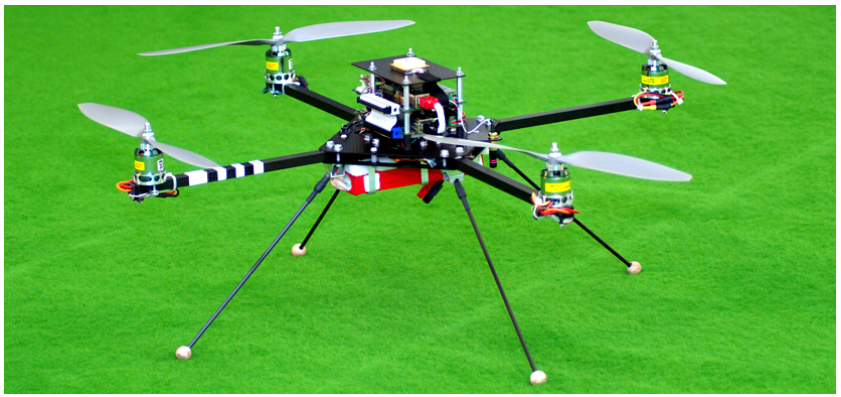
- Mathematischer Nachweis ist schwierig und wenig praxisrelevant
- Stattdessen: Nutzung neuer Techniken wie der *symbolischen Ausführung*

Fehlerinjektion im realen/virtuellen System

- Zuverlässigkeitsanalyse hat ihre Grenzen
- Experimentelle Fehlerinjektion

Zuverlässigkeit implementieren





Weitere Themen

- Sind im Umfeld von DanceOS/CoRed/AORTA möglich
- Auch für Projektarbeiten (z.B., Masterprojekt)

{S, D, B, M}-Arbeiten . . . Doktorarbeiten

Forschungs-/Entwicklungsprojekte: Universität, Forschungseinrichtungen, Industrie

weitere Themen im Internet/UnivIS:

`http://www4.informatik.uni-erlangen.de/Theses`