

# Verlässliche Echtzeitsysteme

## Dynamisches Testen

---

Wintersemester 2024/25

Peter Wägemann

Lehrstuhl für Systemsoftware

Friedrich-Alexander-Universität Erlangen-Nürnberg

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

# Funktionale und nicht-funktionale Eigenschaften

- ☞ Korrektheit der Software hat viele Gesichter
  - Wurde das 1) richtige 2) korrekt implementiert?
- Alle relevanten Eigenschaften sind zu überprüfen!

# Funktionale und nicht-funktionale Eigenschaften

- ☞ Korrektheit der Software hat viele Gesichter
  - Wurde das 1) richtige 2) korrekt implementiert?
  - Alle relevanten Eigenschaften sind zu überprüfen!
- 1. *Funktionale Eigenschaften* (Übereinstimmung mit Spezifikation)
  - Müssen *explizit implementiert* werden  $\mapsto$  `int regelschritt(int sensorwert)`
  - Eine **fehlerhafte** Funktion kann nicht-funktional **korrekt** sein

# Funktionale und nicht-funktionale Eigenschaften

- ☞ Korrektheit der Software hat viele Gesichter
    - Wurde das 1) richtige 2) korrekt implementiert?
    - Alle relevanten Eigenschaften sind zu überprüfen!
  - 1. *Funktionale Eigenschaften* (Übereinstimmung mit Spezifikation)
    - Müssen *explizit implementiert* werden  $\mapsto$  `int regelschritt(int sensorwert)`
    - Eine **fehlerhafte** Funktion kann nicht-funktional **korrekt** sein
  - 2. *Nicht-funktionale Eigenschaften* (z.B. Laufzeitverhalten)
    - Können nur *implizit implementiert* werden
    - Sind **querschneidend**  $\rightsquigarrow$  erst im konkreten Kontext bestimmt
    - Eine **korrekte** Funktion kann nicht-funktional **fehlerhaft** sein
- ⚠ *Robustheit* (Kapitel 3-6) ist eine nicht-funktionale Eigenschaft

# Funktionale und nicht-funktionale Eigenschaften

- ☞ Korrektheit der Software hat viele Gesichter
    - Wurde das 1) richtige 2) korrekt implementiert?
    - Alle relevanten Eigenschaften sind zu überprüfen!
  - 1. *Funktionale Eigenschaften* (Übereinstimmung mit Spezifikation)
    - Müssen *explizit implementiert* werden  $\mapsto$  `int regelschritt(int sensorwert)`
    - Eine **fehlerhafte** Funktion kann nicht-funktional **korrekt** sein
  - 2. *Nicht-funktionale Eigenschaften* (z.B. Laufzeitverhalten)
    - Können nur *implizit implementiert* werden
    - Sind **querschneidend**  $\rightsquigarrow$  erst im konkreten Kontext bestimmt
    - Eine **korrekte** Funktion kann nicht-funktional **fehlerhaft** sein
    - ⚠ *Robustheit* (Kapitel 3-6) ist eine nicht-funktionale Eigenschaft
- ⚠ Es kommt auf die Betrachtungsebene an!

# Zuverlässig Software entwickeln?

- ☞ Ziel: Aussagen zur Korrektheit von *funktionalen* und *nicht-funktionalen Eigenschaften* von Software treffen
  - Fokus der Vorlesung: *Korrektheit* oder zumindest *Absenz von Defekten*
  - Schrittweise Annäherung über *Qualität* und *Verhalten*

# Zuverlässig Software entwickeln?

- ☞ *Ziel*: Aussagen zur Korrektheit von *funktionalen* und *nicht-funktionalen Eigenschaften* von Software treffen
  - Fokus der Vorlesung: *Korrektheit* oder zumindest *Absenz von Defekten*
  - Schrittweise Annäherung über *Qualität* und *Verhalten*
- Hierfür existieren unterschiedliche Ansätze:
  - *Informelle Methoden*
    - Inspection, Review, Walkthrough, ...
  - *Analytische Methoden*
    - Metriken, Kodierrichtlinien, ...
  - *Dynamisches Testen*
    - Black-Box, White-Box, Regression, Unit, ...
  - *Formale Methoden*
    - Statische Code-Analyse, Model Checking, ...

# Zuverlässig Software entwickeln?

- ☞ Ziel: Aussagen zur Korrektheit von *funktionalen* und *nicht-funktionalen Eigenschaften* von Software treffen
  - Fokus der Vorlesung: *Korrektheit* oder zumindest *Absenz von Defekten*
  - Schrittweise Annäherung über *Qualität* und *Verhalten*
- Hierfür existieren unterschiedliche Ansätze:
  - *Informelle Methoden*
    - Inspection, Review, Walkthrough, ...
  - *Analytische Methoden*
    - Metriken, Kodierrichtlinien, ...
  - *Dynamisches Testen*
    - Black-Box, White-Box, Regression, Unit, ...
  - *Formale Methoden*
    - Statische Code-Analyse, Model Checking, ...

} Aussagen über die *Qualität*



# Zuverlässig Software entwickeln?

- ☞ Ziel: Aussagen zur Korrektheit von *funktionalen* und *nicht-funktionalen Eigenschaften* von Software treffen
    - Fokus der Vorlesung: *Korrektheit* oder zumindest *Absenz von Defekten*
    - Schrittweise Annäherung über *Qualität* und *Verhalten*
  - Hierfür existieren unterschiedliche Ansätze:
    - *Informelle Methoden*
      - Inspection, Review, Walkthrough, ...
    - *Analytische Methoden*
      - Metriken, Kodierrichtlinien, ...
    - *Dynamisches Testen*
      - Black-Box, White-Box, Regression, Unit, ...
    - *Formale Methoden*
      - Statische Code-Analyse, Model Checking, ...
- } Aussagen über die *Qualität*
- } Aussagen über das *Verhalten*

# Zuverlässig Software entwickeln?

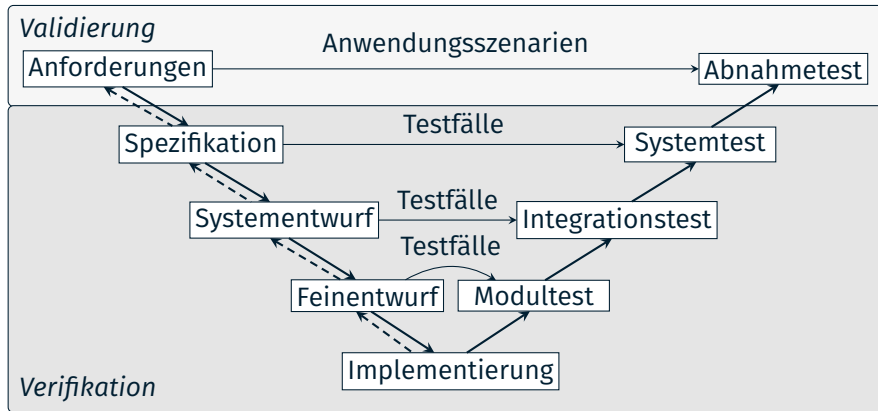
- ☞ Ziel: Aussagen zur Korrektheit von *funktionalen* und *nicht-funktionalen Eigenschaften* von Software treffen
  - Fokus der Vorlesung: *Korrektheit* oder zumindest *Absenz von Defekten*
  - Schrittweise Annäherung über *Qualität* und *Verhalten*
- Hierfür existieren unterschiedliche Ansätze:
  - *Informelle Methoden*
    - Inspection, Review, Walkthrough, ...
  - *Analytische Methoden*
    - Metriken, Kodierrichtlinien, ...
  - *Dynamisches Testen*
    - Black-Box, White-Box, Regression, Unit, ...
  - *Formale Methoden*
    - Statische Code-Analyse, Model Checking, ...

} Aussagen über die *Qualität*

} Aussagen über das *Verhalten*
- ☞ In dieser Vorlesung steht das *Testen des Verhaltens* im Vordergrund

- 1 Testarten und Konzepte
- 2 Bewertung von Testfällen
  - McCabe's Cyclomatic Complexity
  - Testüberdeckung
  - Grenzen dynamischen Testens
- 3 Durchführung und Testumgebung
- 4 Dynamisches Testen durch Fuzzing
- 5 Zusammenfassung

# Einordnung in den Entwicklungsprozess



Softwareentwicklung nach dem V-Modell wird zugrunde gelegt:

- Weit verbreitetes Vorgehensmodell in der Softwareentwicklung
  - *Absteigender Ast*  $\leadsto$  Spezifikation, Entwurf, Implementierung
  - *Aufsteigender Ast*  $\leadsto$  Verifikation & Validierung
  - *Querbeziehungen*  $\leadsto$  Testfallableitung

# Tests in den verschiedenen Phasen des V-Modells

## **Modultest** (engl. *unit testing*)

- Diskrepanz zwischen Implementierung und Entwurf/Spezifikation

## **Integrationstest** (engl. *integration testing*)

- Probleme beim Zusammenspiel mehrerer Module

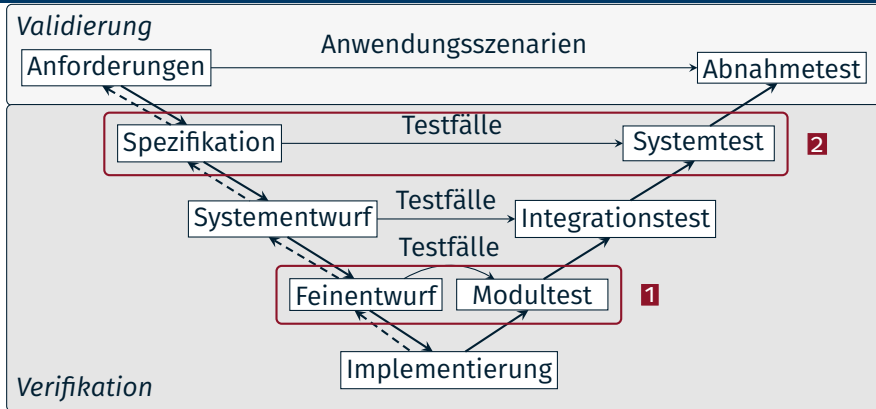
## **Systemtest** (engl. *system testing*)

- Black-Box-Test auf Systemebene
- Vergleich: geforderte Leistung ↔ tatsächliche Leistung
  - Funktional: sind alle Merkmale verfügbar
  - Nicht-funktional: wird z.B. ein bestimmter Durchsatz erreicht

## **Abnahmetest** (engl. *acceptance testing*)

- Erfüllt das Produkt die Anforderungen des Auftraggebers
- Korrektheit, Robustheit, Performanz, Dokumentation, ...
- Wird durch Anwendungsszenarien demonstriert/überprüft
  - Hier findet also eine Validierung statt, keine Verifikation

# Fokus der heutigen Vorlesung



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

# Eigenschaften von Modultests

- Modultests beziehen sich auf *kleine Softwareeinheiten*
  - Meist auf Ebene einzelner Funktionen
    - *Testbarkeit* ist zu gewährleisten  $\leadsto$  Begrenzung der notwendigen Testfälle

# Eigenschaften von Modultests

- Modultests beziehen sich auf *kleine Softwareeinheiten*
  - Meist auf Ebene einzelner Funktionen
    - *Testbarkeit* ist zu gewährleisten  $\leadsto$  Begrenzung der notwendigen Testfälle
- Modultests erfolgen in *Isolation*
  - Für den (Miss-)Erfolg ist **nur** das getestete Modul verantwortlich
  - Andere Module werden durch *Attrappen* (engl. *mock-objects*) ersetzt



# Eigenschaften von Modultests

- Modultests beziehen sich auf *kleine Softwareeinheiten*
  - Meist auf Ebene einzelner Funktionen
    - *Testbarkeit* ist zu gewährleisten  $\leadsto$  Begrenzung der notwendigen Testfälle
- Modultests erfolgen in *Isolation*
  - Für den (Miss-)Erfolg ist **nur** das getestete Modul verantwortlich
  - Andere Module werden durch *Attrappen* (engl. *mock-objects*) ersetzt
- Modultests werden *fortlaufend* durchgeführt
  - Jede Änderung am Quelltext auf Verträglichkeit prüfen
  - *Regressionstests* (engl. *regression testing*)  $\leadsto$  Automatisierung notwendig

# Eigenschaften von Modultests

- Modultests beziehen sich auf *kleine Softwareeinheiten*
  - Meist auf Ebene einzelner Funktionen
    - *Testbarkeit* ist zu gewährleisten  $\leadsto$  Begrenzung der notwendigen Testfälle
- Modultests erfolgen in *Isolation*
  - Für den (Miss-)Erfolg ist **nur** das getestete Modul verantwortlich
  - Andere Module werden durch *Attrappen* (engl. *mock-objects*) ersetzt
- Modultests werden *fortlaufend* durchgeführt
  - Jede Änderung am Quelltext auf Verträglichkeit prüfen
  - *Regressionstests* (engl. *regression testing*)  $\leadsto$  Automatisierung notwendig
- Modultests sollten auch den *Fehlerfall* prüfen
  - Es genügt nicht, zu prüfen, dass ein korrektes Ergebnis berechnet wurde
  - Der Fehlerfall (Eingaben, Zustand, ...) soll einbezogen werden

# Eigenschaften von Modultests

- Modultests beziehen sich auf *kleine Softwareeinheiten*
  - Meist auf Ebene einzelner Funktionen
    - *Testbarkeit* ist zu gewährleisten  $\leadsto$  Begrenzung der notwendigen Testfälle
- Modultests erfolgen in *Isolation*
  - Für den (Miss-)Erfolg ist **nur** das getestete Modul verantwortlich
  - Andere Module werden durch *Attrappen* (engl. *mock-objects*) ersetzt
- Modultests werden *fortlaufend* durchgeführt
  - Jede Änderung am Quelltext auf Verträglichkeit prüfen
  - *Regressionstests* (engl. *regression testing*)  $\leadsto$  Automatisierung notwendig
- Modultests sollten auch den *Fehlerfall* prüfen
  - Es genügt nicht, zu prüfen, dass ein korrektes Ergebnis berechnet wurde
  - Der Fehlerfall (Eingaben, Zustand, ...) soll einbezogen werden
- Modultest betrachten die *Schnittstelle*
  - Anwendung des *Design-By-Contract*-Prinzips  $\leadsto$  *Black-Box-Tests*
  - Interne Details ( $\leadsto$  *White-Box-Tests*) führen zu fragilen Testfällen

# Black-Box- vs. White-Box-Tests

## ■ *Black-Box-Tests*

- Keine Kenntnis der internen Struktur
- Testfälle basieren ausschließlich auf der Spezifikation
- Synonyme: funktionale, datengetriebene, E/A-getriebene Tests

☞ **Frage:** Wurden **alle** Anforderungen (**fehlerfrei**) implementiert?

# Black-Box- vs. White-Box-Tests

## ■ *Black-Box-Tests*

- Keine Kenntnis der internen Struktur
- Testfälle basieren ausschließlich auf der Spezifikation
- Synonyme: funktionale, datengetriebene, E/A-getriebene Tests

☞ **Frage:** Wurden **alle** Anforderungen (**fehlerfrei**) implementiert?

## ■ *White-Box-Tests*

- Kenntnis der internen Struktur zwingend erforderlich
- Testfälle basieren auf Programmstruktur, Spezifikation wird ignoriert
- Synonyme: strukturelle, pfadgetriebene, logikgetriebene Tests

☞ **Frage:** Wurden **nur** Anforderungen (**fehlerfrei**) implementiert?

# Black-Box- vs. White-Box-Tests

## ■ *Black-Box-Tests*

- Keine Kenntnis der internen Struktur
- Testfälle basieren ausschließlich auf der Spezifikation
- Synonyme: funktionale, datengetriebene, E/A-getriebene Tests

☞ **Frage:** Wurden **alle** Anforderungen (**fehlerfrei**) implementiert?

## ■ *White-Box-Tests*

- Kenntnis der internen Struktur zwingend erforderlich
- Testfälle basieren auf Programmstruktur, Spezifikation wird ignoriert
- Synonyme: strukturelle, pfadgetriebene, logikgetriebene Tests

☞ **Frage:** Wurden **nur** Anforderungen (**fehlerfrei**) implementiert?

☞ Weiterer Verlauf der Vorlesung: Fokus auf *White-Box-Verfahren*

- Abstrakte Interpretation, Model Checking, Coverage, WP-Kalkül, ...

# Problem: kombinatorische Explosion

Ohne Einsicht in die Programmstruktur ist Testen sehr mühsam!

- Beispiel: Modultests für OSEK OS [4]
  - Verschiedene Betriebssystemdienste
    - Fadenverwaltung, Fadensynchronisation, Nachrichtenkommunikation, ...
  - Hohe Variabilität
    - **4 Konformitätsklassen**: BCC1, BCC2, ECC1, ECC2
    - **3 Varianten der Ablaufplanung (Verdrängbarkeit)**: NON, MIXED, FULL
    - **2 Betriebsmodi**: Betrieb (STANDARD), Entwicklung (EXTENDED)
    - **24 Varianten** für jeden Testfall

# Problem: kombinatorische Explosion

Ohne Einsicht in die Programmstruktur ist Testen sehr mühsam!

- Beispiel: Modultests für OSEK OS [4]
  - Verschiedene Betriebssystemdienste
    - Fadenverwaltung, Fadensynchronisation, Nachrichtenkommunikation, ...
  - Hohe Variabilität
    - **4 Konformitätsklassen**: BCC1, BCC2, ECC1, ECC2
    - **3 Varianten der Ablaufplanung (Verdrängbarkeit)**: NON, MIXED, FULL
    - **2 Betriebsmodi**: Betrieb (STANDARD), Entwicklung (EXTENDED)
    - **24 Varianten** für jeden Testfall
- Black-Box  $\leadsto$  kein Wissen über die interne Struktur nutzbar
  - **konservative Annahme**: Parameter beeinflussen sich gegenseitig
  - Alle Kombinationen sind relevant: **kombinatorische Explosion!**



# Problem: kombinatorische Explosion

Ohne Einsicht in die Programmstruktur ist Testen sehr mühsam!

- Beispiel: Modultests für OSEK OS [4]
  - Verschiedene Betriebssystemdienste
    - Fadenverwaltung, Fadensynchronisation, Nachrichtenkommunikation, ...
  - Hohe Variabilität
    - **4 Konformitätsklassen**: BCC1, BCC2, ECC1, ECC2
    - **3 Varianten der Ablaufplanung (Verdrängbarkeit)**: NON, MIXED, FULL
    - **2 Betriebsmodi**: Betrieb (STANDARD), Entwicklung (EXTENDED)
    - **24 Varianten** für jeden Testfall
- Black-Box  $\leadsto$  kein Wissen über die interne Struktur nutzbar
  - **konservative Annahme**: Parameter beeinflussen sich gegenseitig
  - Alle Kombinationen sind relevant: **kombinatorische Explosion!**
- Kombination aus Black- und White-Box-Tests  $\leadsto$  **Gray-Box-Tests**
  - Unabhängigkeit der Parameter kann evtl. sichergestellt werden
  - Reduktion der Testfälle bzw. deren Varianten

- 1 Testarten und Konzepte
- 2 Bewertung von Testfällen
  - McCabe's Cyclomatic Complexity
  - Testüberdeckung
  - Grenzen dynamischen Testens
- 3 Durchführung und Testumgebung
- 4 Dynamisches Testen durch Fuzzing
- 5 Zusammenfassung

### ■ Kriterium: *Anzahl der Testfälle*

#### ■ Basierend auf Metriken

- McCabe's Cyclomatic Complexity (MCC), Function/Feature Points, ...

#### ■ Mithilfe von Statistiken aus früheren Projekten

- Kennzahlen früherer Projekte  $\leadsto$  Anzahl zu erwartender Defekte
- Wie viele Defekte hat man bereits gefunden, wie viele sind noch im Produkt?
- Wie viele Defekte will/kann man ausliefern?
- Übertragbarkeit fraglich!?

### ■ Kriterium: *Testüberdeckung*

#### ■ Welcher Anteil des Systems wurde abgetestet?

- Wurden ausreichend viele Programmpfade absolviert?
- Wurden alle Variablen, die definiert wurden, auch verwendet?

# McCabe's Cyclomatic Complexity [2, Kapitel 8.1]

- ☞ Maß für die Anzahl der unabhängigen Pfade durch ein Programm
  - je höher die MCC, desto höher die Komplexität
- Berechnung basiert auf dem *Kontrollflussgraphen*
  - Betrachtungsebene wichtig (wir betrachten C-Code)
  - Knoten repräsentieren *Anweisungen*, Kanten *Pfade*
  - Komplexität  $C$ :  $C = e - n + 2$ 
    - $e \hat{=}$  Anzahl der Kanten (engl. edges),  $n \hat{=}$  Anzahl der Knoten (engl. nodes)

## ■ Beispiele:



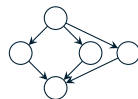
Sequenz  
 $C = 1$



Verzweigung  
 $C = 2$



Do-While  
 $C = 2$



Fallunterscheidung  
 $C = 3$

⚠ **Untere Schranke** für die Anzahl der Testfälle!

- In der Praxis gilt ein Wert im Bereich 1 - 10 als akzeptabel

# Grundlegende Überdeckungskriterien

Wie sehr wurde ein Modul durch Tests beansprucht?

$C_o = s/S$  *Anweisungsüberdeckung* (engl. *statement coverage*)

- $s \rightsquigarrow$  erreichte Anweisungen,  $S \rightsquigarrow$  alle Anweisungen
- Findet nicht erreichbaren/getesteten/übersetzten Code
- **Nachteile:**
  - Gleichgewichtung aller Anweisungen
  - Keine Berücksichtigung leerer Pfade oder Datenabhängigkeiten

# Grundlegende Überdeckungskriterien

Wie sehr wurde ein Modul durch Tests beansprucht?

$C_0 = s/S$  *Anweisungsüberdeckung* (engl. *statement coverage*)

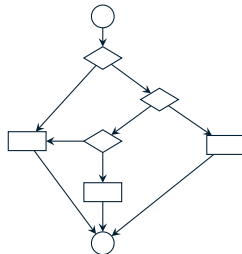
- $s \rightsquigarrow$  erreichte Anweisungen,  $S \rightsquigarrow$  alle Anweisungen
- Findet nicht erreichbaren/getesteten/übersetzten Code
- **Nachteile:**
  - Gleichgewichtung aller Anweisungen
  - Keine Berücksichtigung leerer Pfade oder Datenabhängigkeiten

$C_1 = b/B$  *Zweigüberdeckung* (engl. *branch coverage*)

- $b \rightsquigarrow$  ausgeführte primitive Zweige,  $B \rightsquigarrow$  alle primitiven Zweige
  - Verzweigungen hängen u.U. voneinander ab
  - Zweigüberdeckung und dafür benötigte Testfälle sind **nicht proportional**
  - Primitive Zweige sind *unabhängig* von anderen Zweigen
- Findet nicht erreichbare Zweige, **Defekterkennungsrate ca. 33%**
- **Nachteile:** unzureichende Behandlung von
  - Abhängigen Verzweigungen, Schleifen  $\rightsquigarrow$  *Pfadüberdeckung*
  - Komplexe Verzweigungsbedingungen  $\rightsquigarrow$  *Bedingungsüberdeckung*

# Beispiel: Anweisungs- und Zweigüberdeckung

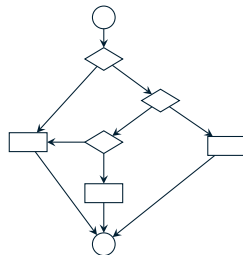
```
int foo(int a,int b,int c) {  
    if((a > b && a > c) || c < 0) {  
        if(a < b) return 1;  
        else {  
            if(b < c) return 2;  
        }  
    }  
    return 4;  
}
```



# Beispiel: Anweisungs- und Zweigüberdeckung

```
int foo(int a,int b,int c) {  
    if((a > b && a > c) || c < 0) {  
        if(a < b) return 1;  
        else {  
            if(b < c) return 2;  
        }  
    }  
    return 4;  
}
```

■ Anweisungsüberdeckung



■ Zweigüberdeckung

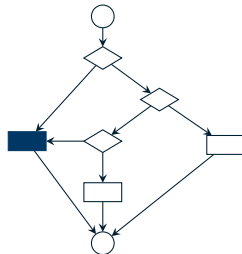


# Beispiel: Anweisungs- und Zweigüberdeckung

```
int foo(int a,int b,int c) {  
    if((a > b && a > c) || c < 0) {  
        if(a < b) return 1;  
        else {  
            if(b < c) return 2;  
        }  
    }  
    return 4;  
}
```

## ■ Anweisungsüberdeckung

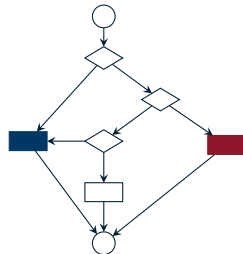
- *Test 1:* foo(0,0,0)



## ■ Zweigüberdeckung

# Beispiel: Anweisungs- und Zweigüberdeckung

```
int foo(int a,int b,int c) {  
    if((a > b && a > c) || c < 0) {  
        if(a < b) return 1;  
        else {  
            if(b < c) return 2;  
        }  
    }  
    return 4;  
}
```



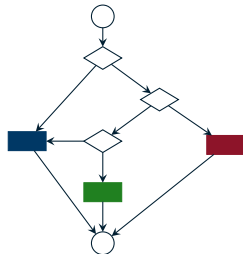
## ■ Anweisungsüberdeckung

- *Test 1:* foo(0,0,0)
- *Test 2:* foo(0,1,-1)

## ■ Zweigüberdeckung

# Beispiel: Anweisungs- und Zweigüberdeckung

```
int foo(int a,int b,int c) {  
    if((a > b && a > c) || c < 0) {  
        if(a < b) return 1;  
        else {  
            if(b < c) return 2;  
        }  
    }  
    return 4;  
}
```



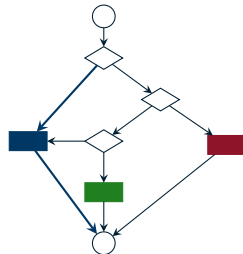
## ■ Anweisungsüberdeckung

- *Test 1:* foo(0,0,0)
- **Test 2:** foo(0,1,-1)
- **Test 3:** foo(2,0,1)

## ■ Zweigüberdeckung

# Beispiel: Anweisungs- und Zweigüberdeckung

```
int foo(int a,int b,int c) {  
    if((a > b && a > c) || c < 0) {  
        if(a < b) return 1;  
        else {  
            if(b < c) return 2;  
        }  
    }  
    return 4;  
}
```



## ■ Anweisungsüberdeckung

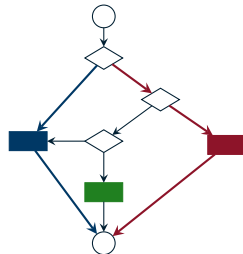
- *Test 1:* foo(0,0,0)
- **Test 2:** foo(0,1,-1)
- **Test 3:** foo(2,0,1)

## ■ Zweigüberdeckung

- *Test 1:* foo(0,0,0)

# Beispiel: Anweisungs- und Zweigüberdeckung

```
int foo(int a,int b,int c) {  
    if((a > b && a > c) || c < 0) {  
        if(a < b) return 1;  
        else {  
            if(b < c) return 2;  
        }  
    }  
    return 4;  
}
```



## ■ Anweisungsüberdeckung

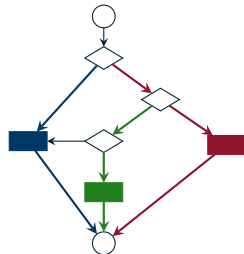
- *Test 1:* foo(0,0,0)
- **Test 2:** foo(0,1,-1)
- **Test 3:** foo(2,0,1)

## ■ Zweigüberdeckung

- *Test 1:* foo(0,0,0)
- **Test 2:** foo(0,1,-1)

# Beispiel: Anweisungs- und Zweigüberdeckung

```
int foo(int a,int b,int c) {  
    if((a > b && a > c) || c < 0) {  
        if(a < b) return 1;  
        else {  
            if(b < c) return 2;  
        }  
    }  
    return 4;  
}
```



## ■ Anweisungsüberdeckung

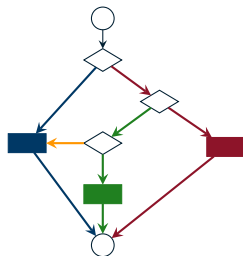
- *Test 1:* foo(0,0,0)
- **Test 2:** foo(0,1,-1)
- **Test 3:** foo(2,0,1)

## ■ Zweigüberdeckung

- *Test 1:* foo(0,0,0)
- **Test 2:** foo(0,1,-1)
- **Test 3:** foo(2,0,1)

# Beispiel: Anweisungs- und Zweigüberdeckung

```
int foo(int a,int b,int c) {  
    if((a > b && a > c) || c < 0) {  
        if(a < b) return 1;  
        else {  
            if(b < c) return 2;  
        }  
    }  
    return 4;  
}
```



## ■ Anweisungsüberdeckung

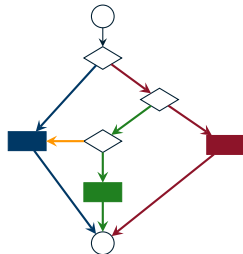
- *Test 1:* foo(0,0,0)
- **Test 2:** foo(0,1,-1)
- **Test 3:** foo(2,0,1)

## ■ Zweigüberdeckung

- *Test 1:* foo(0,0,0)
- **Test 2:** foo(0,1,-1)
- **Test 3:** foo(2,0,1)
- **Test 4:** foo(2,1,1)

# Beispiel: Anweisungs- und Zweigüberdeckung

```
int foo(int a,int b,int c) {  
    if((a > b && a > c) || c < 0) {  
        if(a < b) return 1;  
        else {  
            if(b < c) return 2;  
        }  
    }  
    return 4;  
}
```



## ■ Anweisungsüberdeckung

- *Test 1:* foo(0,0,0)
- **Test 2:** foo(0,1,-1)
- **Test 3:** foo(2,0,1)

■ 100% Zweigüberdeckung  $\mapsto$  100% Anweisungsüberdeckung

■ Zweigüberdeckung: weite industrielle Verbreitung

- Moderater Aufwand, gute Defekterkennungsrate

## ■ Zweigüberdeckung

- *Test 1:* foo(0,0,0)
- **Test 2:** foo(0,1,-1)
- **Test 3:** foo(2,0,1)
- **Test 4:** foo(2,1,1)



$C_2 = p/P$  Pfadüberdeckung (engl. *path coverage*)

- Pfade vom *Anfangs-* bis zum *Endknoten* im Kontrollflussgraphen

$C_2 = p/P$  Pfadüberdeckung (engl. *path coverage*)

- Pfade vom *Anfangs-* bis zum *Endknoten* im Kontrollflussgraphen

- Abstufungen der Pfadüberdeckung

$C_{2a}$  *vollständige Pfadüberdeckung*

- Abdeckung *aller* möglichen Pfade ( $C_1$  plus Beachtung des Pfadkontexts)
- **Problem:** durch Schleifen entstehen u. U. unendlich viele Pfade

$C_{2b}$  *boundary-interior Pfadüberdeckung*

- Wie  $C_{2a}$ , Anzahl der Schleifendurchläufe wird auf  $\leq 2$  *beschränkt*

$C_{2c}$  *strukturierte Pfadüberdeckung*

- Wie  $C_{2b}$ , Anzahl der Schleifendurchläufe wird auf  $\leq n$  *beschränkt*

- Bedeutung *Boundary-Interior*

**boundary** Jede Schleife 0-mal betreten

Jede Schleife 1-mal betreten, alle Pfade im Rumpf abgearbeitet

**interior** mit 2 bzw. n Durchläufen erreichb. Pfade im Rumpf

$C_2 = p/P$  Pfadüberdeckung (engl. *path coverage*)

- Pfade vom *Anfangs-* bis zum *Endknoten* im Kontrollflussgraphen
- Abstufungen der Pfadüberdeckung
  - $C_{2a}$  *vollständige Pfadüberdeckung*
    - Abdeckung *aller* möglichen Pfade ( $C_1$  plus Beachtung des Pfadkontexts)
    - **Problem:** durch Schleifen entstehen u. U. unendlich viele Pfade
  - $C_{2b}$  *boundary-interior Pfadüberdeckung*
    - Wie  $C_{2a}$ , Anzahl der Schleifendurchläufe wird auf  $\leq 2$  *beschränkt*
  - $C_{2c}$  *strukturierte Pfadüberdeckung*
    - Wie  $C_{2b}$ , Anzahl der Schleifendurchläufe wird auf  $\leq n$  *beschränkt*
- Bedeutung *Boundary-Interior*
  - boundary** Jede Schleife 0-mal betreten  
Jede Schleife 1-mal betreten, alle Pfade im Rumpf abgearbeitet
  - interior** mit 2 bzw. n Durchläufen erreichb. Pfade im Rumpf
- **Hohe Defekterkennungsrate**

$C_2 = p/P$  Pfadüberdeckung (engl. *path coverage*)

- Pfade vom *Anfangs-* bis zum *Endknoten* im Kontrollflussgraphen
- Abstufungen der Pfadüberdeckung
  - $C_{2a}$  *vollständige Pfadüberdeckung*
    - Abdeckung *aller* möglichen Pfade ( $C_1$  plus Beachtung des Pfadkontexts)
    - **Problem:** durch Schleifen entstehen u. U. unendlich viele Pfade
  - $C_{2b}$  *boundary-interior Pfadüberdeckung*
    - Wie  $C_{2a}$ , Anzahl der Schleifendurchläufe wird auf  $\leq 2$  *beschränkt*
  - $C_{2c}$  *strukturierte Pfadüberdeckung*
    - Wie  $C_{2b}$ , Anzahl der Schleifendurchläufe wird auf  $\leq n$  *beschränkt*
- Bedeutung *Boundary-Interior*
  - boundary** Jede Schleife 0-mal betreten  
Jede Schleife 1-mal betreten, alle Pfade im Rumpf abgearbeitet
  - interior** mit 2 bzw. n Durchläufen erreichb. Pfade im Rumpf
- **Hohe Defekterkennungsrate**
- Best. Pfade können nicht erreicht werden ( $C_{2b}$ ,  $C_{2c}$ ), **hoher Aufwand**

$C_3$  *Bedingungsüberdeckung* (engl. *condition coverage*)

- $C_{0,1,2}$ : unzureichende Betrachtung von Bedingungen
  - Ihre Zusammensetzung/Hierarchie wird nicht berücksichtigt

## $C_3$ *Bedingungsüberdeckung* (engl. *condition coverage*)

- $C_{0,1,2}$ : unzureichende Betrachtung von Bedingungen
  - Ihre Zusammensetzung/Hierarchie wird nicht berücksichtigt
- Abstufungen der Bedingungsüberdeckung

### $C_3a$ *Einfachbedingungsüberdeckung*

- Jede atomare Bedingung einmal mit `true` und `false` testen

### $C_3b$ *Mehrfachbedingungsüberdeckung*

- Alle Kombinationen atomarer Bedingungen werden getestet

### $C_3c$ *minimale Mehrfachbedingungsüberdeckung*

- Jede atomare/komplexe Bedingung einmal mit `true` und `false` testen

### **MC/DC** (engl. *modified condition/decision coverage*)

- Sonderform der  $C_3c$ -Überdeckung
- Jede atomare Bedingung mit `true` und `false` testen und ...
- Man muss zusätzlich die umgebende komplexe Bedingung beeinflussen

## $C_3$ Bedingungsüberdeckung (engl. *condition coverage*)

- $C_{0,1,2}$ : unzureichende Betrachtung von Bedingungen
  - Ihre Zusammensetzung/Hierarchie wird nicht berücksichtigt

- Abstufungen der Bedingungsüberdeckung

### $C_3a$ Einfachbedingungsüberdeckung

- Jede atomare Bedingung einmal mit `true` und `false` testen

### $C_3b$ Mehrfachbedingungsüberdeckung

- Alle Kombinationen atomarer Bedingungen werden getestet

### $C_3c$ minimale Mehrfachbedingungsüberdeckung

- Jede atomare/komplexe Bedingung einmal mit `true` und `false` testen

### **MC/DC** (engl. *modified condition/decision coverage*)

- Sonderform der  $C_3c$ -Überdeckung
- Jede atomare Bedingung mit `true` und `false` testen und ...
- Man muss zusätzlich die umgebende komplexe Bedingung beeinflussen

- **Sehr hohe Fehlererkennungsrate**

## $C_3$ Bedingungsüberdeckung (engl. *condition coverage*)

- $C_{0,1,2}$ : unzureichende Betrachtung von Bedingungen
  - Ihre Zusammensetzung/Hierarchie wird nicht berücksichtigt
- Abstufungen der Bedingungsüberdeckung

### $C_3a$ Einfachbedingungsüberdeckung

- Jede atomare Bedingung einmal mit `true` und `false` testen

### $C_3b$ Mehrfachbedingungsüberdeckung

- Alle Kombinationen atomarer Bedingungen werden getestet

### $C_3c$ minimale Mehrfachbedingungsüberdeckung

- Jede atomare/komplexe Bedingung einmal mit `true` und `false` testen

### **MC/DC** (engl. *modified condition/decision coverage*)

- Sonderform der  $C_3c$ -Überdeckung
- Jede atomare Bedingung mit `true` und `false` testen und ...
- Man muss zusätzlich die umgebende komplexe Bedingung beeinflussen

- **Sehr hohe Fehlererkennungsrate**
- Bestimmte Pfade (z.B. in Schleifen) können nicht erreicht werden,  
**hoher Aufwand**



# Beispiel: Bedingungsüberdeckung

```
int foo(int a,int b,int c) {  
    if((a > b && a > c) || c < 0) {  
        if(a < b) return 1;  
        else {  
            if(b < c) return 2;  
        }  
    }  
    return 4;  
}
```

- Fokus dieses Beispiels hier nur auf **erste Bedingung**:

$(a > b \ \&\& \ a > c) \ || \ c < 0$

- 3 atomare Teilbedingungen

- $a > b$
- $a > c$
- $c < 0$

# Beispiel: Bedingungsüberdeckung

```
int foo(int a,int b,int c) {  
    if((a > b && a > c) || c < 0) {  
        if(a < b) return 1;  
        else {  
            if(b < c) return 2;  
        }  
    }  
    return 4;  
}
```

## ■ Einfachbedingungsüberdeckung

a > b	a > c	c < 0	Testfall
w	w	w	f(1,0,-1)
f	f	f	f(0,1,1)

## ■ Fokus dieses Beispiels hier nur auf **erste Bedingung**:

(a > b && a > c) || c < 0

### ■ 3 atomare Teilbedingungen

- a > b
- a > c
- c < 0

# Beispiel: Bedingungsüberdeckung

```
int foo(int a,int b,int c) {  
    if((a > b && a > c) || c < 0) {  
        if(a < b) return 1;  
        else {  
            if(b < c) return 2;  
        }  
    }  
    return 4;  
}
```

- Fokus dieses Beispiels hier nur auf **erste Bedingung**:

$(a > b \ \&\& \ a > c) \ || \ c < 0$

- 3 atomare Teilbedingungen

- $a > b$
- $a > c$
- $c < 0$

- Einfachbedingungsüberdeckung

$a > b$	$a > c$	$c < 0$	Testfall
w	w	w	$f(1,0,-1)$
f	f	f	$f(0,1,1)$

- Modified Condition/Decision Coverage

$a > b$	$a > c$	$c < 0$	$(a > b \ \&\& \ a > c) \    \ c < 0$	Testfall
w	w	f	w	$f(1,0,0)$
f	w	f	f	$f(1,1,0)$
w	f	f	f	$f(1,0,1)$
f	f	w	w	$f(-1,0,-1)$

# Testen hat seine Grenzen!

- ☞ Testen ist im Allgemeinen sehr **aufwändig!**
  - Ziel müssen möglichst vollständige Tests sein!
  - Aber woher weiß man, dass man genügend getestet hat?

# Testen hat seine Grenzen!

- ☞ Testen ist im Allgemeinen sehr **aufwändig!**
  - Ziel müssen möglichst vollständige Tests sein!
  - Aber woher weiß man, dass man genügend getestet hat?
- ⚠ Vollständige Tests sind in der Praxis **unrealistisch**
  - „...wir haben schon lange keinen Fehler mehr gefunden ...“
    - Eine Auffassung, der man oft begegnet
    - Der entscheidende Fehler kann sich immer noch versteckt halten
  - Therac 25 (s. Folie II/3 ff.) wurde > 2700 Stunden betrieben

# Testen hat seine Grenzen!

- ☞ Testen ist im Allgemeinen sehr **aufwändig!**
  - Ziel müssen möglichst vollständige Tests sein!
  - Aber woher weiß man, dass man genügend getestet hat?
- ⚠ Vollständige Tests sind in der Praxis **unrealistisch**
  - „...wir haben schon lange keinen Fehler mehr gefunden ...“
    - Eine Auffassung, der man oft begegnet
    - Der entscheidende Fehler kann sich immer noch versteckt halten
  - Therac 25 (s. Folie II/3 ff.) wurde > 2700 Stunden betrieben

## Fehlerfreie Software durch Testen?

- Praktisch sind Tests für einen **Korrektheitsnachweis** ungeeignet!
- Testen kann nur das **Vertrauen in Software erhöhen!**

# Testen hat seine Grenzen!

- ☞ Testen ist im Allgemeinen sehr **aufwändig!**
  - Ziel müssen möglichst vollständige Tests sein!
  - Aber woher weiß man, dass man genügend getestet hat?
- ⚠ Vollständige Tests sind in der Praxis **unrealistisch**
  - „...wir haben schon lange keinen Fehler mehr gefunden ...“
    - Eine Auffassung, der man oft begegnet
    - Der entscheidende Fehler kann sich immer noch versteckt halten
  - Therac 25 (s. Folie 11/3 ff.) wurde > 2700 Stunden betrieben

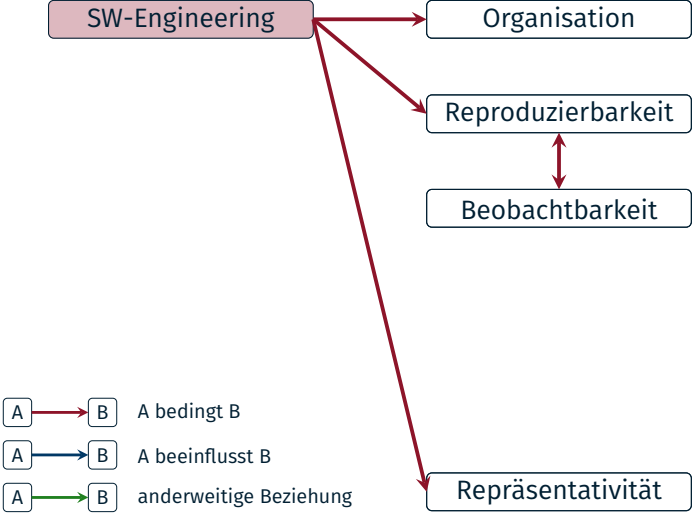
## Fehlerfreie Software durch Testen?

- Praktisch sind Tests für einen **Korrektheitsnachweis** ungeeignet!
- Testen kann nur das **Vertrauen in Software erhöhen!**
- **Formale Methoden** gehen einen anderen Weg
  - Weisen Übereinstimmung anstatt Abweichung nach
  - Gegenstand kommender Vorlesungen

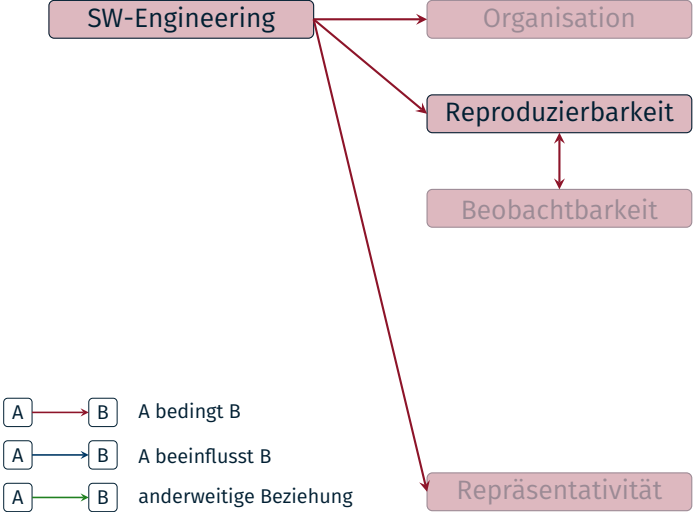
- 1 Testarten und Konzepte
- 2 Bewertung von Testfällen
  - McCabe's Cyclomatic Complexity
  - Testüberdeckung
  - Grenzen dynamischen Testens
- 3 Durchführung und Testumgebung**
- 4 Dynamisches Testen durch Fuzzing
- 5 Zusammenfassung



# Testen: Ein Problem des „SW-Engineering“



# Problemfeld: Reproduzierbarkeit



Für die Fehlersuche muss man das Fehlverhalten nachstellen können!

- ☞ Wichtige Testvariante: *Regressionstests* (engl. *regression testing*)
  - Wurde der Fehler auch wirklich korrigiert?
  - Hat die Korrektur neue Defekte verursacht?

Für die Fehlersuche muss man das Fehlverhalten nachstellen können!

- ☞ Wichtige Testvariante: *Regressionstests* (engl. *regression testing*)
  - Wurde der Fehler auch wirklich korrigiert?
  - Hat die Korrektur neue Defekte verursacht?
- Voraussetzung für Regressionstests  $\leadsto$  *Reproduzierbarkeit (d. Fehlverhaltens)*
  - Andernfalls ist keine Aussage zur Behebung des Fehler möglich
  - Verschiedene Ursachen können dasselbe Symptom hervorrufen

Für die Fehlersuche muss man das Fehlverhalten nachstellen können!

☞ Wichtige Testvariante: *Regressionstests* (engl. *regression testing*)

- Wurde der Fehler auch wirklich korrigiert?
- Hat die Korrektur neue Defekte verursacht?

■ Voraussetzung für Regressionstests  $\leadsto$  *Reproduzierbarkeit (d. Fehlverhaltens)*

- Andernfalls ist keine Aussage zur Behebung des Fehler möglich
- Verschiedene Ursachen können dasselbe Symptom hervorrufen

⚠ Voraussetzung für die Reproduzierbarkeit ist:

- *Beobachtbarkeit* und die
  - *Kontrollierbarkeit* des Systems
- Testfälle müssen sich *deterministisch* verhalten  
(vgl. Replikdeterminismus IV/16 ff)

# Reproduzierbarkeit $\leftrightarrow$ Beobachtbarkeit

Fehlverhalten zu reproduzieren erfordert mehr Wissen, als es zu erkennen

## Nicht-deterministische Operationen

- Abhängigkeiten z. B. vom Netzwerkverkehr
- Zufallszahlen, interne Systemzustände (`syscall()`), ...

# Reproduzierbarkeit ↔ Beobachtbarkeit

Fehlverhalten zu reproduzieren erfordert mehr Wissen, als es zu erkennen

## ⚠ Nicht-deterministische Operationen

- Abhängigkeiten z. B. vom Netzwerkverkehr
- Zufallszahlen, interne Systemzustände (`syscall()`), ...

## ⚠ Ungenügendes Vorabwissen

- Fadensynchronisation
- Asynchrone Programmunterbrechungen (engl. *interrupts*)
- Zeitbasis der untersuchten Systeme

# Reproduzierbarkeit ↔ Beobachtbarkeit

Fehlverhalten zu reproduzieren erfordert mehr Wissen, als es zu erkennen

## ⚠ Nicht-deterministische Operationen

- Abhängigkeiten z. B. vom Netzwerkverkehr
- Zufallszahlen, interne Systemzustände (`syscall()`), ...

## ⚠ Ungenügendes Vorabwissen

- Fadensynchronisation
- Asynchrone Programmunterbrechungen (engl. *interrupts*)
- Zeitbasis der untersuchten Systeme

## ☞ Dies sind *relevante Ereignisse*

- Sie beeinflussen den Programmablauf
  - Hängen von der Anwendung ab
- *Identifikation* und *Beobachtung* erforderlich





# Beobachtbarkeit verteilter EZS

- ☞ Beobachtung aller relevanten Zustände des Systems
  - *Ausgaben* bzw. *Ergebnisse*
  - *Zwischenzustände* und *-ergebnisse*

---

<sup>1</sup>Der Effekt/der Einflussnahme auf eine Komponente oder ein System durch die Messung. [1]

# Beobachtbarkeit verteilter EZS

- ☞ Beobachtung aller relevanten Zustände des Systems
  - *Ausgaben* bzw. *Ergebnisse*
  - *Zwischenzustände* und *-ergebnisse*
- **Problem: Ausgaben** beeinflussen das Systemverhalten
  - Ausgaben verzögern Prozesse, Nachrichten, ...  $\rightsquigarrow$  **Termin**

---

<sup>1</sup>Der Effekt/der Einflussnahme auf eine Komponente oder ein System durch die Messung. [1]

# Beobachtbarkeit verteilter EZS

- ☞ Beobachtung aller relevanten Zustände des Systems
  - *Ausgaben* bzw. *Ergebnisse*
  - *Zwischenzustände* und *-ergebnisse*
- **Problem: Ausgaben** beeinflussen das Systemverhalten
  - Ausgaben verzögern Prozesse, Nachrichten, ...  $\rightsquigarrow$  **Termin**
- **Problem: Debuggen**  $\mapsto$  Unmöglichkeit globaler Haltepunkte
  - Perfekt synchronisierte Uhren existieren nicht
  - $\rightarrow$  Wie soll man Prozesse gleichzeitig anhalten?

---

<sup>1</sup>Der Effekt/der Einflussnahme auf eine Komponente oder ein System durch die Messung. [1]

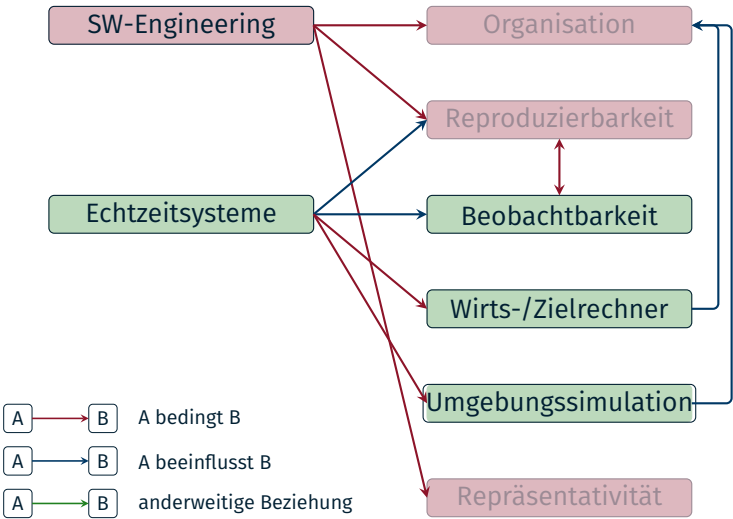
# Beobachtbarkeit verteilter EZS

- ☞ Beobachtung aller relevanten Zustände des Systems
  - *Ausgaben* bzw. *Ergebnisse*
  - *Zwischenzustände* und *-ergebnisse*
- **Problem: Ausgaben** beeinflussen das Systemverhalten
  - Ausgaben verzögern Prozesse, Nachrichten, ...  $\leadsto$  **Termin**
- **Problem: Debuggen**  $\mapsto$  Unmöglichkeit globaler Haltepunkte
  - Perfekt synchronisierte Uhren existieren nicht
  - $\rightarrow$  Wie soll man Prozesse gleichzeitig anhalten?
- ⚠ Bekanntes Phänomen: *Untersuchungseffekt* (engl. *probe effect*)<sup>1</sup>
  - Vergleiche Heisenbugs auf Folie III/25
  - $\rightarrow$  „Vorführeffekt“ – sobald man hinsieht, ist der Fehler verschwunden
  - $\rightarrow$  Muss vermieden oder kompensiert werden

---

<sup>1</sup>Der Effekt/der Einflussnahme auf eine Komponente oder ein System durch die Messung. [1]

# Problemfeld: Fokus „Echtzeitsysteme“



## Untersuchungseffekt: Verschärfung durch zeitlichen Aspekt

- Untersuchungseffekt auf **gleichzeitige Prozesse**
  - Systemzustand verteilt sich auf mehrere, gleichzeitig ablaufende Prozesse
  - Durch Beeinflussung einzelner Prozesse verändert sich der globale Zustand
- Andere Prozesse enteilen dem beeinflussten Prozess
- Ein Fehler lässt sich evtl. nicht reproduzieren

# Untersuchungseffekt: Verschärfung durch zeitlichen Aspekt

- Untersuchungseffekt auf **gleichzeitige Prozesse**
  - Systemzustand verteilt sich auf mehrere, gleichzeitig ablaufende Prozesse
  - Durch Beeinflussung einzelner Prozesse verändert sich der globale Zustand
  - Andere Prozesse enteilen dem beeinflussten Prozess
  - Ein Fehler lässt sich evtl. nicht reproduzieren
- Untersuchungseffekt auf **Zeitstempel**
  - Neben dem Datum ist häufig ein **Zeitstempel** notwendig
  - Erstellen des Zeitstempels selbst benötigt Zeit (Auslesen einer Uhr, ...)
  - Die zu protokollierende Datenmenge wächst ebenfalls an
- Untersuchungseffekt bei Kopplung an die **physikalische Zeit**
  - Das kontrollierte Objekt enteilt dem beeinflussten Prozess
  - Auch einzelne Prozesse sind anfällig



# Untersuchungseffekt: Lösungsmöglichkeiten

## Ignoranz

- Der *Untersuchungseffekt* wird schon nicht auftreten

# Untersuchungseffekt: Lösungsmöglichkeiten

## Ignoranz

- Der *Untersuchungseffekt* wird schon nicht auftreten

## Minimierung

- Hinreichend effiziente Datenaufzeichnung
- Tracing z.B. Intel Power Trace, ARM CoreSight
- Kompensation der aufgezeichneten Daten
  - Verhindert nicht die Verfälschung des globalen Zustands

# Untersuchungseffekt: Lösungsmöglichkeiten

## Ignoranz

- Der *Untersuchungseffekt* wird schon nicht auftreten

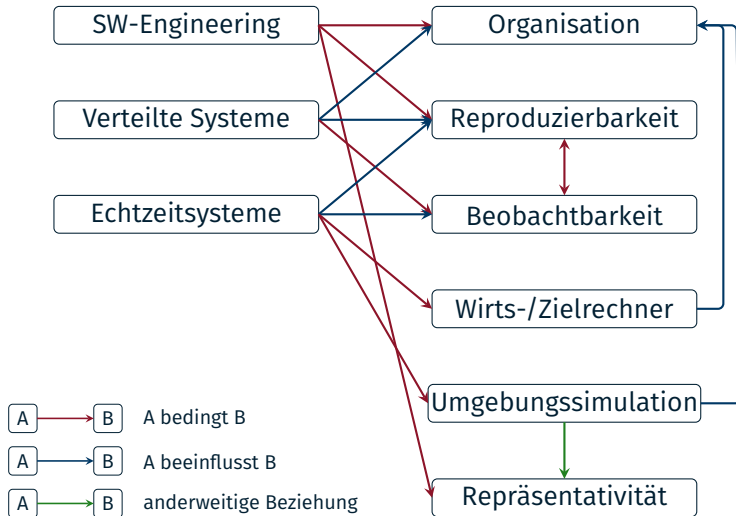
## Minimierung

- Hinreichend effiziente Datenaufzeichnung
- Tracing z.B. Intel Power Trace, ARM CoreSight
- Kompensation der aufgezeichneten Daten
  - Verhindert nicht die Verfälschung des globalen Zustands

## Vermeidung

- Datenaufzeichnung existiert auch im Produktivsystem
  - Einsatz dedizierter Hardware für die Datenaufzeichnung
  - Einflussnahme wird hinter einer logischen Uhr verborgen
    - Zeitliche Schwankungen sind nicht relevant
- Solange sich eine gewisse Reihenfolge nicht ändert

# Kontrollierbarkeit: Ein umfassendes Problem



- ☞ *Deterministische Ausführung relevanter Ereignisse*
  - Beibehaltung der ursprünglichen Reihenfolge
  - Zeitlich akkurat
  - Umfasst **alle relevanten Ereignisse**
    - Asynchrone Programmunterbrechungen
    - Interne Entscheidungen des Betriebssystems  $\rightsquigarrow$  Einplanung, Synchronisation

- ☞ *Deterministische Ausführung relevanter Ereignisse*
  - Beibehaltung der ursprünglichen Reihenfolge
  - Zeitlich akkurat
  - Umfasst **alle relevanten Ereignisse**
    - Asynchrone Programmunterbrechungen
    - Interne Entscheidungen des Betriebssystems  $\rightsquigarrow$  Einplanung, Synchronisation
- *Simulierte Zeit* statt **realer, physikalischer Zeitbasis**
  - Entkopplung von der Geschwindigkeit der realen Welt
  - Ansonsten könnte die Fehlersuche sehr, sehr lange dauern ...

- ☞ *Deterministische Ausführung relevanter Ereignisse*
  - Beibehaltung der ursprünglichen Reihenfolge
  - Zeitlich akkurat
  - Umfasst **alle relevanten Ereignisse**
    - Asynchrone Programmunterbrechungen
    - Interne Entscheidungen des Betriebssystems  $\leadsto$  Einplanung, Synchronisation
- *Simulierte Zeit* statt **realer, physikalischer Zeitbasis**
  - Entkopplung von der Geschwindigkeit der realen Welt
  - Ansonsten könnte die Fehlersuche sehr, sehr lange dauern ...
- Ansätze zur *Kontrollierbarkeit*
  - *Analytische* Ansätze
    - Record & Replay
  - *Konstruktive* Ansätze
    - Statische Quelltextanalyse
    - Quelltexttransformation

# Record & Replay

- ☞ *Vermessung* (engl. *monitoring*) zur Laufzeit
    - Aufzeichnung **aller** relevanten Ereignisse
    - Dieser Mitschnitt wird später erneut abgespielt
- *Event histories* bzw. *event traces*



- ☞ *Vermessung* (engl. *monitoring*) zur Laufzeit
  - Aufzeichnung **aller** relevanten Ereignisse
  - Dieser Mitschnitt wird später erneut abgespielt

→ *Event histories* bzw. *event traces*
- **Vorteil:** Lösungen für verteilte Echtzeitsysteme existieren
  - **Vermeiden** Untersuchungseffekt
  - Decken eine **Vielzahl verschiedener Ereignisse** ab
    - Systemaufrufe, Kontextwechsel, asynchrone Unterbrechungen, ...
    - Synchronisation, Zugriffe auf gemeinsame Variablen, ...

- ☞ *Vermessung* (engl. *monitoring*) zur Laufzeit
  - Aufzeichnung **aller** relevanten Ereignisse
  - Dieser Mitschnitt wird später erneut abgespielt

→ *Event histories* bzw. *event traces*
- **Vorteil:** Lösungen für verteilte Echtzeitsysteme existieren
  - **Vermeiden** Untersuchungseffekt
  - Decken eine **Vielzahl verschiedener Ereignisse** ab
    - Systemaufrufe, Kontextwechsel, asynchrone Unterbrechungen, ...
    - Synchronisation, Zugriffe auf gemeinsame Variablen, ...
- ⚠ **Nachteil:** enorm hoher Aufwand
  - Häufig ist **Spezialhardware** erforderlich
  - Es fallen **große Datenmengen** an
    - Aufzeichnung erfolgt i. d. R. auf Maschinencodeebene, Eingaben, ...
  - Es können **nur beobachtete Szenarien** wiederholt werden
    - Änderungen am System machen existierende Mitschnitte u. U. wertlos
  - Wiederholung & Mitschnitt müssen auf **demselben System** stattfinden

# Konstruktion von Ausführungsszenarien

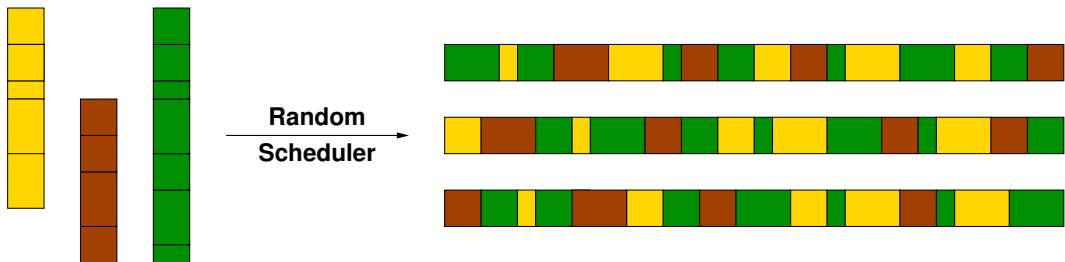
- Identifizierung möglicher Ausführungsszenarien
  - Berücksichtigung von Kommunikation, Synchronisation, Einplanung, ...

# Konstruktion von Ausführungsszenarien

- Identifizierung möglicher Ausführungsszenarien
  - Berücksichtigung von Kommunikation, Synchronisation, Einplanung, ...
- Ausführungsszenarien werden erzwungen
  - *Random Scheduler*
    - Gleichzeitige Prozesse → sequentielles Programm
  - Teste Sequentialisierungen statt der gleichzeitigen Prozesse

# Konstruktion von Ausführungsszenarien

- Identifizierung möglicher Ausführungsszenarien
  - Berücksichtigung von Kommunikation, Synchronisation, Einplanung, ...
- Ausführungsszenarien werden erzwungen
  - *Random Scheduler*
    - Gleichzeitige Prozesse → sequentielles Programm
  - Teste Sequentialisierungen statt der gleichzeitigen Prozesse
- Vorgehen ist mit *Model Checking* vergleichbar



# Herausforderungen beim Testen verteilter EZS [5]

Herausforderungen ergeben sich vor allem aus der Systemebene:

- Herausforderungen spezifisch für *Echtzeitsysteme*
  - Starke **Kopplung zur Umgebung**
    - Echtzeitsysteme interagieren vielfältig mit dem kontrollierten Objekt
  - **Voranschreiten der realen Zeit** nicht vernachlässigbar
    - Physikalische Vorgänge im kontrollierten Objekt sind an die Zeit gekoppelt
  - **Umgebung kann nicht beliebig beeinflusst werden**
    - Kontrollbereich der Aktuatoren ist beschränkt
- Herausforderungen spezifisch für *verteilte Systeme*
  - **Hohe Komplexität**
    - Verteilung erhöht Komplexität  $\rightsquigarrow$  Allokation, Kommunikation, ...
  - **Beobachtung und Reproduzierbarkeit** des Systemverhaltens
  - **Fehlende globale Zeit**  $\rightsquigarrow$  kein eindeutiger globaler Zustand
    - Globale, konsistente Abbilder sind ein großes Problem



**Ähnliche Herausforderungen wie bei Fehlerinjektion**

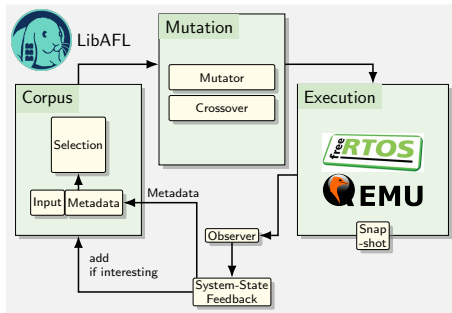
- 1 Testarten und Konzepte
- 2 Bewertung von Testfällen
  - McCabe's Cyclomatic Complexity
  - Testüberdeckung
  - Grenzen dynamischen Testens
- 3 Durchführung und Testumgebung
- 4 Dynamisches Testen durch Fuzzing**
- 5 Zusammenfassung

# Dynamisches Testen durch Fuzzing

- Random Testing + Heuristik  $\approx$  Fuzzing [3]
- Oft *coverage-guided Fuzzing*
- Hier Heuristik: Maximierung der Überdeckung
- Qualität der Heuristiken  $\rightsquigarrow$  Qualität der Überdeckung
- Corpus: Menge and möglichen “spannenden” Testfällen
- Genetischer Algorithmus als Teil des Fuzzings
  - Mutation: Veränderung Eingabe
  - Crossover: Kombination von Eingaben
- Fuzzing sehr gut *parallelisierbar*



# Fuzzing-Implementierung von AFL



- American Fuzzy Lop (AFL) [6]
- Instrumentierung des Programms ähnlich gcov (White-Box-Test?)  $\leadsto$  Überdeckungskriterium
- Feedback: Verbesserung der Überdeckung?
- Diversität  $\leadsto$  Vermeidung lokaler Minima
- Kriterien für Aufnahme neuer Testfälle in Corpus: z.B. Pfadlänge
- Abbruchkriterien: Laufzeit, Fortschritt, ...

- 1 Testarten und Konzepte
- 2 Bewertung von Testfällen
  - McCabe's Cyclomatic Complexity
  - Testüberdeckung
  - Grenzen dynamischen Testens
- 3 Durchführung und Testumgebung
- 4 Dynamisches Testen durch Fuzzing
- 5 Zusammenfassung**

# Zusammenfassung

- 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*  $\leadsto$  Minimalzahl von Testfällen
  - Kontrollflussorientierte *Testüberdeckung*
    - *Anweisungs-, Zweig-, Pfad- und Bedinungsüberdeckung*
- Testdurchführung in (verteilten) Echtzeitsystemen sind **herausfordernd!**
  - Problemfeld: Testen verteilter Echtzeitsysteme
    - SW-Engineering, verteilte Systeme, Echtzeitsysteme
    - Untersuchungseffekt, Beobachtbarkeit, Kontrollierbarkeit, Reproduzierbarkeit
- *Fuzzing* als weitverbreitete dynamische Testmethode

- [1] Hamburg, M. ; Hehn, U. :  
**ISTQB®/GTB Standardglossar der Testbegriffe.**  
(2010)
- [2] Laplante, P. A.:  
***Real-Time Systems Design and Analysis.***  
3.  
John Wiley & Sons, Inc., 2004. –  
ISBN 0-471-22855-9

[3] Miller, B. P. ; Fredriksen, L. ; So, B. :

**An empirical study of the reliability of UNIX utilities.**

33, Nr. 12, S. 32-44.

<http://dx.doi.org/10.1145/96267.96279>. –

DOI 10.1145/96267.96279. –

ISSN 0001-0782

[4] OSEK/VDX Group:

**Operating System Specification 2.2.3 / OSEK/VDX Group.**

2005. –

Forschungsbericht

[5] Schütz, W. :

**Fundamental issues in testing distributed real-time systems.**

In: *Real-Time Systems Journal* 7 (1994), Nr. 2, S. 129–157.

<http://dx.doi.org/10.1007/BF01088802>. –

DOI 10.1007/BF01088802. –

ISSN 0922–6443

[6] Zalewski, M. :

**Technical “whitepaper” for afl-fuzz.**

(2014).

[http://lcamtuf.coredump.cx/afl/technical\\_details.txt](http://lcamtuf.coredump.cx/afl/technical_details.txt)