

Verlässliche Echtzeitsysteme

Testen

Peter Ulbrich

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

28. April 2015



Warum testet man eigentlich?

- Ziel: Aussage zu **nicht-funktionalen Eigenschaften** von Software
 - 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, ...
- in dieser Vorlesung steht das **Testen des Verhaltens** im Vordergrund



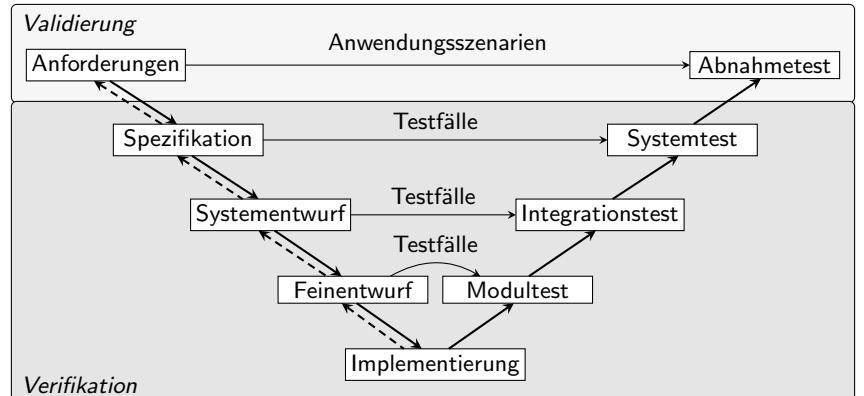
Gliederung

- 1 Überblick
- 2 Testarten und Vorgehen
 - Entwicklungsprozess
 - Modultests
 - Black-Box- vs. White-Box-Tests
- 3 Bewertung
 - McCabe's Cyclomatic Complexity
 - Testüberdeckung
- 4 Testen verteilter Echtzeitsysteme
 - Problemfeld
 - Reproduzierbarkeit
 - Beobachtbarkeit
 - Kontrollierbarkeit
- 5 Zusammenfassung



Einordnung in den Entwicklungsprozess

Softwareentwicklung nach dem V-Modell wird zugrunde gelegt



- weit verbreitetes Vorgehensmodell in der Softwareentwicklung
 - **absteigender Ast** ~ Spezifikation, Entwurf, Implementierung
 - **aufsteigender Ast** ~ Verifikation & Validierung
 - **Querbeziehungen** ~ 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 mehrere 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

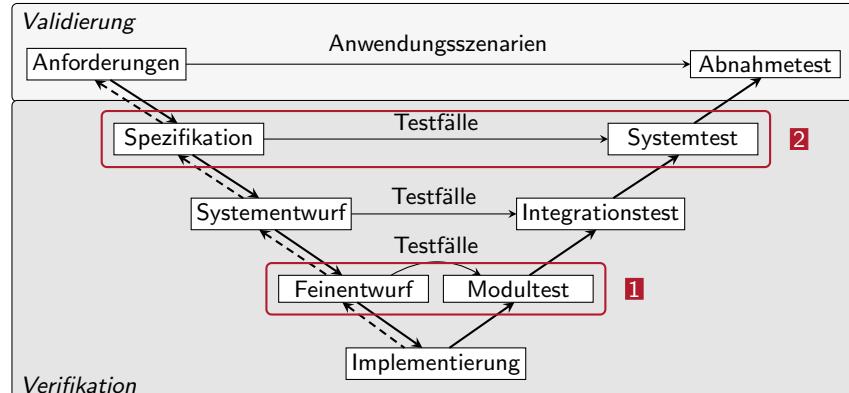


Eigenschaften von Modultests

- Modultests beziehen sich auf **kleine Softwareeinheiten**
 - meist auf Ebene einzelner Funktionen
 - die **Testbarkeit** ist zu gewährleisten ∼ 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 sollte auf ihre Verträglichkeit geprüft werden
 - ∼ **Regressionstests** (engl. *regression testing*) ∼ 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 ∼ **Black-Box-Tests**
 - interne Details (∼ **White-Box-Tests**) führen zu fragilen Testfällen



Fokus der heutigen Vorlesung



- **1 Modultests** ∼ Grundbegriffe und Problemstellung
 - ↪ Black- vs. White-Box, Testüberdeckung
- **2 Systemtest** ∼ Testen verteilter Echtzeitsysteme
 - ↪ Problemstellung und Herausforderungen



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 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 implementiert?
- weiterer Verlauf der Vorlesung: Fokus auf **White-Box-Verfahren**
 - abstrakte Interpretation, Model Checking, Coverage, WP-Kalkül, ...



Gliederung

- 1 Überblick
- 2 Testarten und Vorgehen
 - Entwicklungsprozess
 - Modultests
 - Black-Box- vs. White-Box-Tests
- 3 Bewertung
 - McCabe's Cyclomatic Complexity
 - Testüberdeckung
- 4 Testen verteilter Echtzeitsysteme
 - Problemfeld
 - Reproduzierbarkeit
 - Beobachtbarkeit
 - Kontrollierbarkeit
- 5 Zusammenfassung



Problem: Kombinatorische Explosion

Ohne Einsicht in die Programmstruktur ist Testen sehr mühsam!

- Beispiel: Modultests für OSEK OS [2]
 - verschiedene Betriebssystemdienste
 - Fadenverwaltung, Fadensynchronisation, Nachrichtenkommunikation, ...
 - hohe Variabilität
 - 4 Konformitätsklassen: BCC1, BCC2, BCC3, BCC4
 - 3 Varianten der Ablaufplanung: NON, MIXED, FULL
 - 2 Betriebsmodi: Betrieb (STANDARD), Entwicklung (EXTENDED)
 - ~ 24 Varianten für jeden Testfall
- Black-Box ~ 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
 - ~ Unabhängigkeit der Parameter kann evtl. sichergestellt werden
 - ~ Reduktion der Testfälle bzw. deren Varianten



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?
 - Praktisch sind Tests für einen **Korrektheitsnachweis** ungeeignet!
 - „... wir haben schon lange keinen Fehler mehr gefunden ...“
 - eine Auffassung, die man oft begegnet
 - ~ der entscheidende Fehler kann sich immer noch versteckt halten
 - der Therac 25 (s. Folie II/3 ff.) wurde > 2700 Stunden betrieben
 - ohne dass ein „nennenswerter“ Fehler aufgetreten wäre
 - trotzdem kam es zu den verheerenden Vorfällen
 - ~ Testen kann nur das **Vertrauen in Software** erhöhen!
- ☞ **Formale Methoden** gehen einen anderen Weg
 - Weisen Übereinstimmung anstatt Abweichung nach
 - Deutlich komplexere Handhabung ~ **Tests oft unumgänglich**
 - Gegenstand kommender Vorlesungen



Hat man genug getestet?

Wie viele Testfälle sind genug Testfälle?

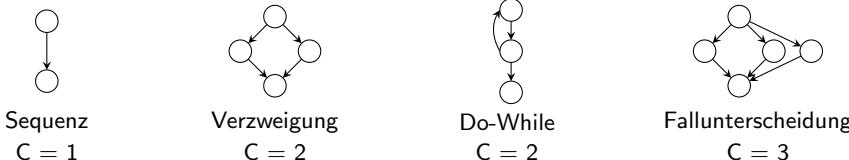
- Kriterium: **Anzahl der Testfälle**
 - basierend auf Metriken
 - McCabe's Cyclomatic Complexity (MCC), Function/Feature Points, ...
 - mithilfe von Statistiken aus früheren Projekten
 - Kennzahlen früherer Projekte ~ Anzahl zu erwartender Defekte
 - Wie viele Defekte hat man bereits gefunden, wie viele sind noch im Produkt?
 - Wie viele Defekte will/kann man ausliefern?
 - ~ Übertragbarkeit?
- 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 [1, 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**
 - Knoten repräsentieren **Anweisungen**, Kanten **Pfade**
 - Komplexität C :
$$C = e - n + 2$$

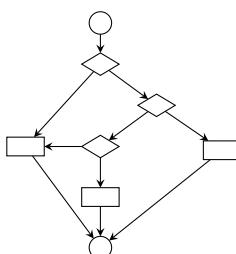
- $e \hat{=} \text{Anzahl der Kanten}$, $n \hat{=} \text{Anzahl der Knoten}$
- Beispiele:



- Untere Schranke für die Anzahl der Testfälle!
 - in der Praxis gilt ein Wert im Bereich 1 - 10 als akzeptabel

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)`
 - Test 4: `foo(2,1,1)`
- 100% Zweigüberdeckung \mapsto 100% Anweisungsüberdeckung
- Zweigüberdeckung: weite industrielle Verbreitung
 - moderater Aufwand, gute Defekterkennungsrate

Grundlegende Überdeckungskriterien

Wie sehr wurde ein Modul durch Tests beansprucht?

$$C_0 = s/S \text{ Anweisungsüberdeckung (engl. statement coverage)}$$

- $s \sim$ erreichte Anweisungen, $S \sim$ alle Anweisungen
- findet nicht erreichbaren/getesteten/übersetzten Code
- Nachteile:
 - Gleichgewichtung aller Anweisungen
 - keine Berücksichtigung leerer Pfade oder Datenabhängigkeiten

$$C_1 = b/B \text{ Zweigüberdeckung (engl. branch coverage)}$$

- $b \sim$ ausgeführte primitive Zweige, $B \sim$ 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 \sim **Pfadüberdeckung**
 - komplexe Verzweigungsbedingungen \sim **Bedingungsüberdeckung**

Pfadüberdeckung

$$C_2 = p/P \text{ Pfadüberdeckung (engl. path coverage)}$$

- Pfade vom Anfangs- bis zum Endknoten im Kontrollflussgraphen
- Abstufungen der Pfadüberdeckung
 - C_{2a} vollständige Pfadüberdeckung**
 - Abdeckung **aller** möglichen Pfade
 - Problem:** durch Schleifen entstehen u. U. unendlich viele Pfade
 - C_{2b} boundary-interior Pfadüberdeckung**
 - wie C_{2a} , Anzahl der Schleifendurchläufe wird auf ≤ 2 beschränkt
 - C_{2c} strukturierte Pfadüberdeckung**
 - wie C_{2b} , Anzahl der Schleifendurchläufe wird auf $\leq n$ beschränkt
- Bedeutung **Boundary-Interior**
 - boundary-** jede Schleife wird 0-mal betreten
 - jede Schleife wird 1-mal betreten, alle Pfade im Rumpf abgearbeitet
 - interior-** Beschränkung: mit 2 bzw. n Durchläufen erreichbare Pfade im Rumpf
- hohe **Defekterkennungsrate**
- bestimmte Pfade können nicht erreicht werden, **hoher Aufwand**

Bedingungsüberdeckung

C₃ 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 wird einmal mit `true` und `false` getestet
 - C_{3b} Mehrfachbedingungsüberdeckung
 - alle Kombinationen atomarer Bedingungen werden getestet
 - C_{3c} minimale Mehrfachbedingungsüberdeckung
 - jede atomare/komplexe Bedingung wird einmal mit `true` und `false` getestet
- MC/DC (engl. *modified condition/decision coverage*)
 - Sonderform der C_{3c}-Überdeckung
 - jede atomare Bedingung wird mit `true` und `false` getestet und ...
 - muss zusätzlich die umgebende komplexe Bedingung beeinflussen
- sehr hohe Fehlererkennungsrate
- bestimmte Pfade können nicht erreicht werden, **hoher Aufwand**



Gliederung

- 1 Überblick
- 2 Testarten und Vorgehen
 - Entwicklungsprozess
 - Modultests
 - Black-Box- vs. White-Box-Tests
- 3 Bewertung
 - McCabe's Cyclomatic Complexity
 - Testüberdeckung
- 4 Testen verteilter Echtzeitsysteme
 - Problemfeld
 - Reproduzierbarkeit
 - Beobachtbarkeit
 - Kontrollierbarkeit
- 5 Zusammenfassung



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 auf die Bedingung:
 $(a > b \&\& a > c) || c < 0$
- 3 atomare Teilbedingungen
 - $a > b$
 - $a > c$
 - $c < 0$

Einfachbedingungsüberdeckung

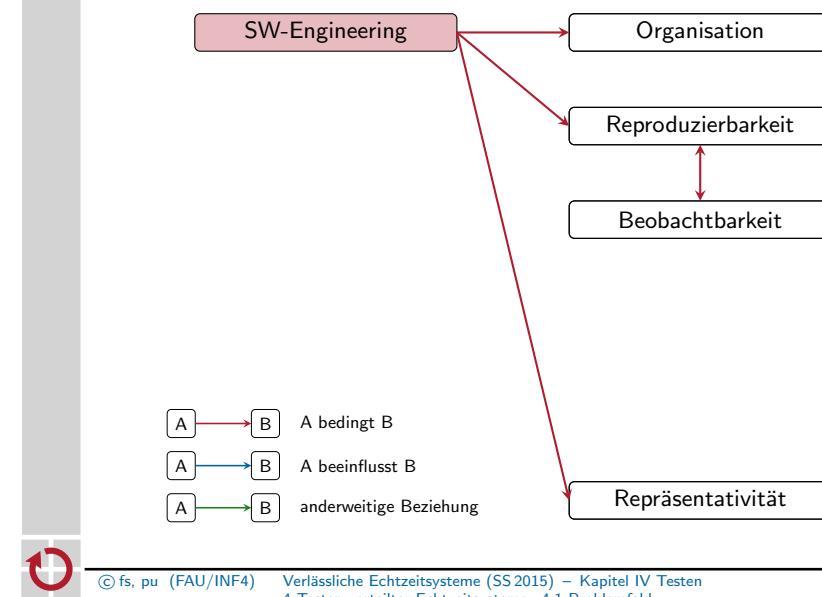
a > b	a > c	c < 0	Testfall
w	w	w	f(1,0,-1)
f	f	f	f(0,1,1)

Modified Condition/Decision Coverage

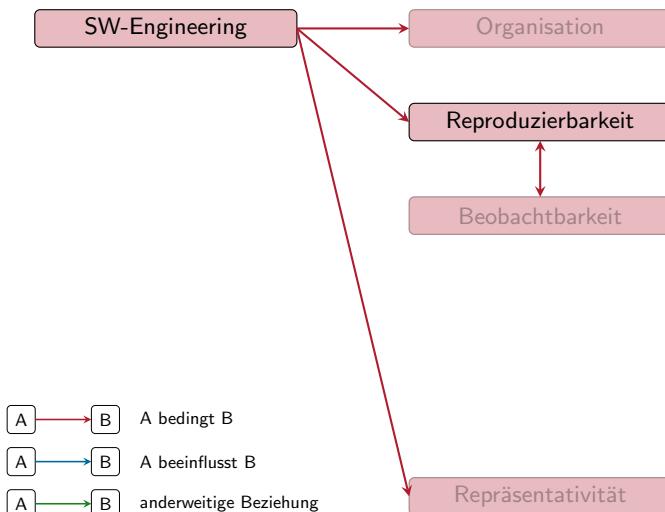
a > b	a > c	c < 0	(a > b && a > c) c < 0	Testfall
w	w	f	w	f(1,0,0)
f	w	f	f	f(1,1,0)
w	f	f	f	f(1,0,1)
f	f	w	w	f(-1,0,-1)



Testen: Ein Problem des „SW-Engineering“



Problemfeld: Reproduzierbarkeit



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?
- Voraussetzung für Regressionstests \leadsto Reproduzierbarkeit
 - 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
 - Wie sonst soll man das Fehlverhalten nachstellen?



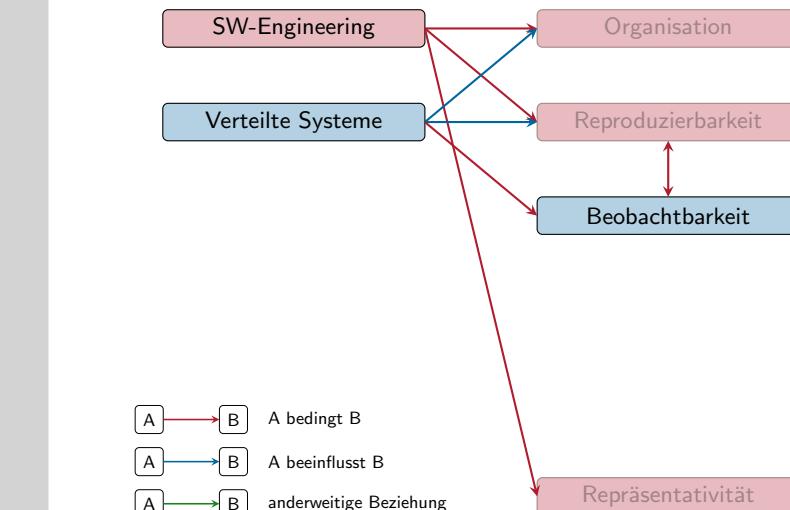
Reproduzierbarkeit \leftrightarrow Beobachtbarkeit

Fehlverhalten zu reproduzieren erfordert mehr Wissen, als es zu erkennen.

- nicht-deterministische Operationen
 - Abhängigkeiten z. B. vom Netzwerkverkehr
 - Zufallszahlen
 - ungenügendes Vorabwissen
 - Fadensynchronisation
 - asynchrone Programmunterbrechungen
 - Zeitbasis der untersuchten Systeme
 - dies sind relevante Ereignisse
 - sie beeinflussen den Programmablauf
 - hängen von der Anwendung ab
- \leadsto Identifikation und Beobachtung erforderlich



Problemfeld: Fokus „Verteilte Systeme“



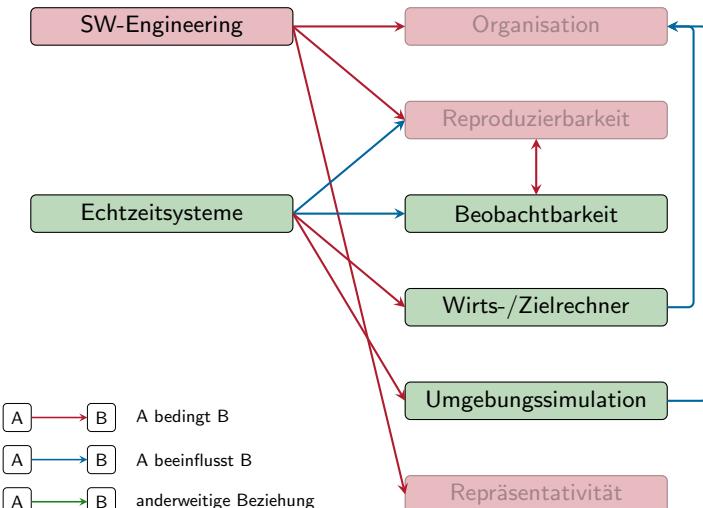
Beobachtbarkeit verteilter EZS

Erfassen des relevanten Verhaltens des Systems und der Umwelt

- Was kann man beobachten?
 - Ausgaben bzw. Ergebnisse
 - Zwischenzustände und -ergebnisse
 - erfordern u.U. zusätzliche Ausgaben (~ aufwändig, häufiges Übersetzen)
 - Inspektion des Speichers mit einem Debugger
- Problem: Ausgaben beeinflussen das Systemverhalten
 - Ausgaben verzögern Prozesse, Nachrichten, ... ~ Termin
- Problem: Debuggen Unmöglichkeit globaler Haltepunkte
 - perfekt synchronisierte Uhren existieren nicht
 - ~ Wie soll man Prozesse gleichzeitig anhalten?
- bekanntes Phänomen: **Probe Effect**
 - ~ „Vorführereffekt“ – sobald man hinsieht, ist der Fehler verschwunden
 - ~ muss vermieden oder kompensiert werden



Problemfeld: Fokus „Echtzeitsysteme“



„Probe Effect“: Verschärfung durch zeitlichen Aspekt

- Aspekt **verteilte Systeme**
 - „Probe Effect“ durch **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
- Aspekt **Echtzeitsysteme**
 - „Probe Effect“ durch **Zeitstempel**
 - neben dem Datum ist häufig ein **Zeitstempel** notwendig
 - das Erstellen des Zeitstempels selbst benötigt Zeit (Auslesen einer Uhr, ...)
 - die zu protokollierende Datenmenge wächst ebenfalls an
 - „Probe Effect“ durch Kopplung an die **physikalische Zeit**
 - das kontrollierte Objekt enteilt dem beeinflussten Prozess
 - ~ auch einzelne Prozesse sind anfällig



„Probe Effect“: Lösungsmöglichkeiten

Ignoranz

- der „Probe Effect“ wird schon nicht auftreten

Minimierung

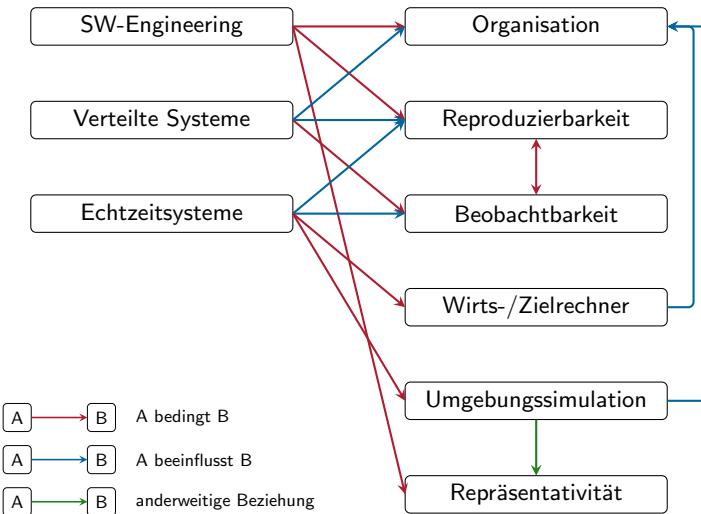
- hinreichend effiziente Datenaufzeichnung
- Kompensation der aufgezeichneten Daten
 - verhindert nicht die Verfälschung des globalen Zustands

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



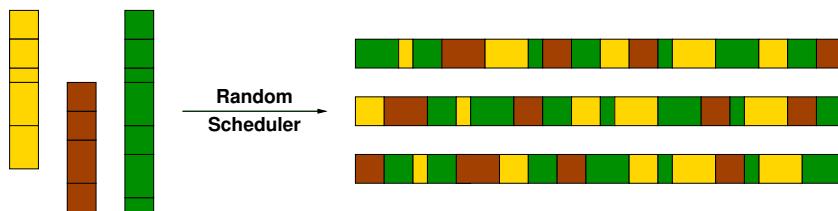
Kontrollierbarkeit

- Abspiele relevanter Ereignisse
 - Beibehaltung der ursprünglichen Reihenfolge
 - zeitlich akkurat
 - umfasst alle relevanten Ereignisse
 - asynchrone Programmunterbrechungen
 - interne Entscheidungen des Betriebssystems → 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
 - sprachbasierte Ansätze
 - statische Quelltextanalyse
 - Quelltexttransformation
 - implementierungsbasierte Ansätze
 - Record & Replay



Statische Quelltextanalyse

- 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 grob-granularem Model Checking vergleichbar



Record & Replay

- Monitoring zur Laufzeit
 - Aufzeichnung aller relevanten Ereignisse
 - event histories bzw. event traces
- dieser Mitschnitt wird später erneut abgespielt
- Vorteil: Lösungen für verteilte Echtzeitsysteme existieren
 - vermeiden „Probe Effect“
 - 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



Herausforderungen beim Testen verteilter EZS [3]

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 \sim Allokation, Kommunikation, ...

■ Beobachtung und Reproduzierbarkeit des Systemverhaltens

■ fehlende globale Zeit \sim kein eindeutiger globaler Zustand

- globale, konsistente Abbilder sind ein großes Problem



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 \sim Minimalzahl von Testfällen
- Kontrollflussorientierte Testüberdeckung
 - Anweisungs-, Zweig-, Pfad- und Bedingungsüberdeckung
 - Angaben zur Testüberdeckung sind immer relativ!

Systemtests für verteilte Echtzeitsysteme sind herausfordernd!

- Problemfeld: Testen verteilter Echtzeitsysteme
 - SW-Engineering, verteilte Systeme, Echtzeitsysteme
 - Probe-Effect, Beobachtbarkeit, Kontrollierbarkeit, Reproduzierbarkeit



Gliederung

1 Überblick

2 Testarten und Vorgehen

- Entwicklungsprozess
- Modultests
- Black-Box- vs. White-Box-Tests

3 Bewertung

- McCabe's Cyclomatic Complexity
- Testüberdeckung

4 Testen verteilter Echtzeitsysteme

- Problemfeld
- Reproduzierbarkeit
- Beobachtbarkeit
- Kontrollierbarkeit

5 Zusammenfassung



Literaturverzeichnis

[1] LAPLANTE, P. A.:

Real-Time Systems Design and Analysis.
third.
John Wiley & Sons, Inc., 2004. –
ISBN 0-471-22855-9

[2] OSEK/VDX GROUP:

Operating System Specification 2.2.3 / OSEK/VDX Group.
2005. –
Forschungsbericht. –
<http://portal.osek-vdx.org/files/pdf/specs/os223.pdf>, visited 2014-09-29

[3] 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

