

# TESTEN

Alexander Knopp

Zusammenfassung

## SOFTWARE QUALITÄT

>> Software errors are the cause of poor software quality <<  
Galen 2004

### Sicherungsprinzipien

- > Qualitätszielbestimmung
- > quantitative Qualitäts sicherung
- > maximal-konstruktive Qualitäts sicherung
- > frühzeitige Fehlererkennung + Behebung
- > entwicklungsbegleitende Qualitäts sicherung
- > unabhängige Qualitäts sicherung

### Konstruktive Maßnahmen

- technisch, z.B. Werkzeuge
- organisatorisch, z.B. Standards

### Analytische Maßnahmen

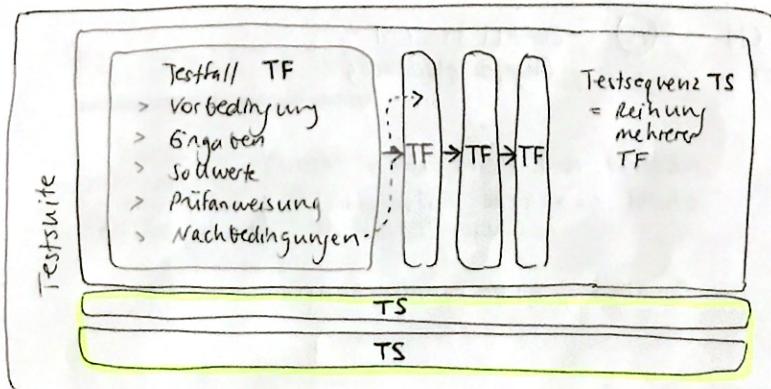
- analyserend: statische Analyse, Programmreihenfolge, Review
- testend: dynamischer Test, symbolischer Test, Simulation

## GRUNDLAGEN



>> Testing is the process of executing a program with the intention of finding errors << Myers 1979

DEBUGGING: Lokalisieren + Beheben eines Fehlers

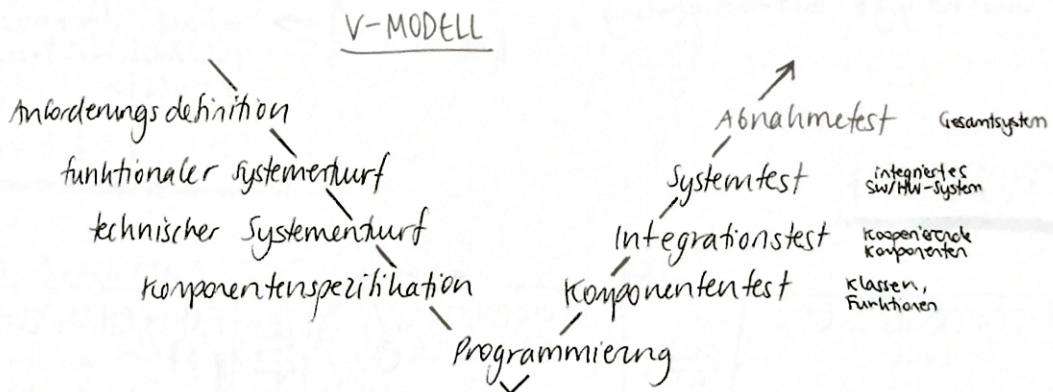
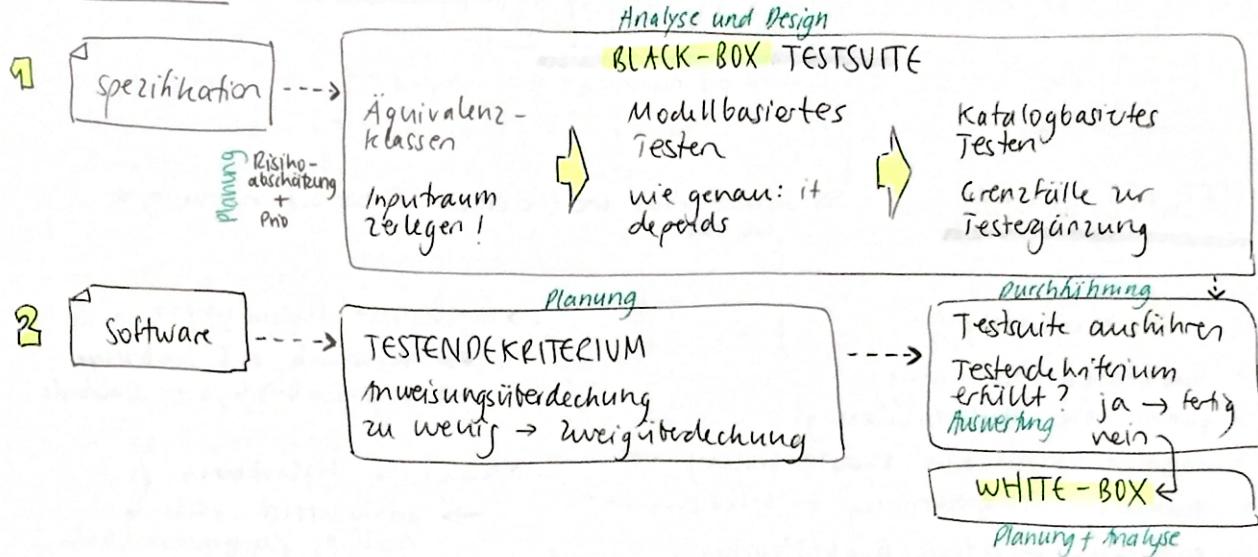


Testfallspezifikation:  
Anforderungs-  
beschreibung

Testlauf:  
Ausführen 1-n  
TF mit best.  
Version + Testobjekt

Testzenario:  
Menge von TS

## TESTPROZESS



### EXPLORATIVES TESTEN

manuelles  
Aufprobieren  
in Paaren möglich

### FUNKTIONALES TESTEN

= Black-Box-Testen  
Testfälle aus funktionaler  
Spezifikation

### MODELLBASIERTES TESTEN

Testfälle aus formaler  
Anforderungsdefinition  
Modell z.B. FSM, Vor/Nachbd.

### STRUKTURELLES TESTEN

= White-Box-Testen  
Testfälle aus  
Implementierung

### SYSTEMATISCHES TESTEN

VS

### ZUFALLSTESTEN

Annäherung an Adäquatstekriterium

### Adäquatstekriterium

Prädikat aus Testobjekten + Testsuiten  
erfüllt  $\Leftrightarrow$  jede Testverpflichtung wird von  
einem Testfall oder Testsuite erfüllt

### Überdeckungsgrad

# ÄQUIVALENZKLASSEN TESTEN

Black Box

## KATEGORIEN-PARTITIONS-TESTEN

Spezifikation → Parameter → Kategorien → Werteklassen →  
 Annotationen [error] [single] [property xy] [if xy] → *sieht nur mit Domänenwissen*  
 Generierung Testfallspezifikation durch gültige Kombinationen  
 der Werteklassen

**m-ÜBERDECKUNG** *geht auch ohne Domänenwissen* für jedes  $m \geq 2$  gibt es eine m-Überdeckung von D mit höchstens  $\#_{\text{Werteklassen}}^m \cdot (\ln(\frac{k}{m}) + m \cdot \ln(1))$  Elementen.

- wächst nur logarithmisch in der Anzahl der Kategorien
- es wird nicht jede Kategorie mit jeder anderen kombiniert, sondern es werden nur Paare / Triplets ... getestet

Weitere Möglichkeiten der Reduktion:

- > direkter Ausschluss
- > Aufteilen in Tabellen gültiger Kombinationen

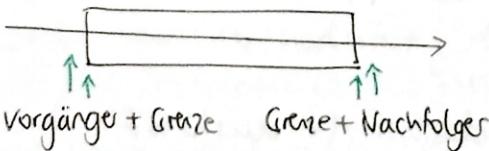
# TESTFALLERGÄNZUNG

Black Box

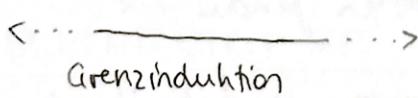
## GRÄNZWERTBETRACHTUNG

*boundary values*

- > geschlossenes Wertintervall



- > offene Wertklasse



## KATALOGBASIERTES TESTEN

Ergänzen der initialen Testfallspezifikationen um weitere Testfallspezifikationen aus Katalog

- > common bugs
- > Routine
- > Reduktion menschlicher Fehler

# PARTITIONENTESTEN

Black Box

## vs ZUFALLSTESTEN

i.A. Steigerung der Fehlschlagswahrscheinlichkeit durch Partitionentesten nicht besonders hoch

Optimierung:

- 1 so aufteilen, dass Partition mit höherer Fehlschlagswahrscheinlichkeit kleiner
- 2 mit Erwartungswert statt Wahrscheinlichkeiten modellieren (# Fehler vorab eh nicht bekannt)

⚡ wie? Domänenwissen?

# TESTEN MIT ZUSTANDSMASCHINEN

black  
box

**DETERMINISTISCHE MEALY-MASCHINE**  $M = (I, O, Q, \delta, \lambda, q_0)$

- > endlicher Eingabealphabet  $I$  finite state machine FSM
- > endliches Ausgabealphabet  $O$
- > endliche Zustandsmenge  $Q$
- > Übergangsfunktion  $\delta: Q \times I \rightarrow Q$
- > Ausgätekarte  $\lambda: Q \times I \rightarrow O$
- > Anfangszustand  $q_0$

Transitionen:  $(q, i, o, q') \in Q \times I \times O \times Q$  mit

$$\delta(q, i) = q' \text{ und } \lambda(q, i) = o$$

**stark zusammenhängend**: jeder Zustand ist von jedem anderen aus durch Eingabefolge erreichbar

Adäquatheitskriterien  
für modellbasiertes Testen

**ZUSTANDSÜBERDECKUNG**

**EINGABEÜBERDECKUNG**

**AUSGABEÜBERDECKUNG**

subsumiert unter ein paar Bedingungen  
**TRANSITIONSSÜBERDECKUNG**  
durch Touren

weitere Möglichkeit:  
Transitionsüberdeckung

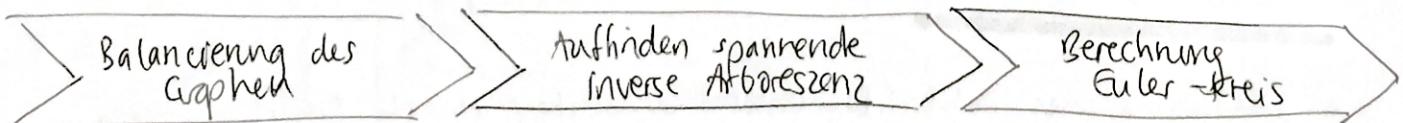
**Transitionstour**: Pfad, der jede Transition mindestens 1x auslöst

**Chinese - Postman - Tour**: Transitionstour minimaler Länge

**Euler - Tour**: Transitionstour, auf der jede Transition genau 1x ausgelöst wird

**Euler - Kreis**: Euler - Tour mit gleichem Anfangs- und Endknoten

**BERECHNUNG EULER-KREIS**



1 benrete Knoten mit ausgehende - eingehende Kanten

= Spannbaum mit max 1 ausgehende Kante pro Knoten

2 Kanten verdoppeln bis Graph balanciert

wähle solange Kante, die in SIA hineinführt, bis alle Knoten abgedeckt

- 1 starte mit Wurzel der SIA (Knoten ohne ausgehende Kante)
- 2 nehme Kanten hinzu, vermeide SIA Kanten so lange wie möglich
- 3 stop wenn jede Kante erziucht

## ERWEITERTE ENDLICHE ZUSTANDSMASCHINEN

extended finite state machine EFSM

- > strukturierte Zustände (Konfigurationen) ( $q, \beta$ )  
kontrollzustand Variablenbelegung
  - > Variablen mit endlichen Wertebereich
- Erweiterte Transitionen:  $(q, g, i, o, a, q')$  mit  
Wächter  $g$  (z.B.  $x=0$ ) und Aktion  $a$   
(z.B.  $x=x+1$ )

## MODEL CHECKING

Model checker findet Ablauf, der eine temporallogische Eigenschaft erhält, z.B. erreiche  
zustand  $q_s$

← Generierung von Testfällen mit EFSMs →

## CONSTRAINT-LOGIC PROGRAMMING

Symbolische Ausführung zur Berechnung von Abläufen, Repräsentation der EFSM als Constraint-Problem

## TESTEN MIT VOR- UND NACHBEDINGUNGEN

black box

### JAVA MODELING LANGUAGE JML

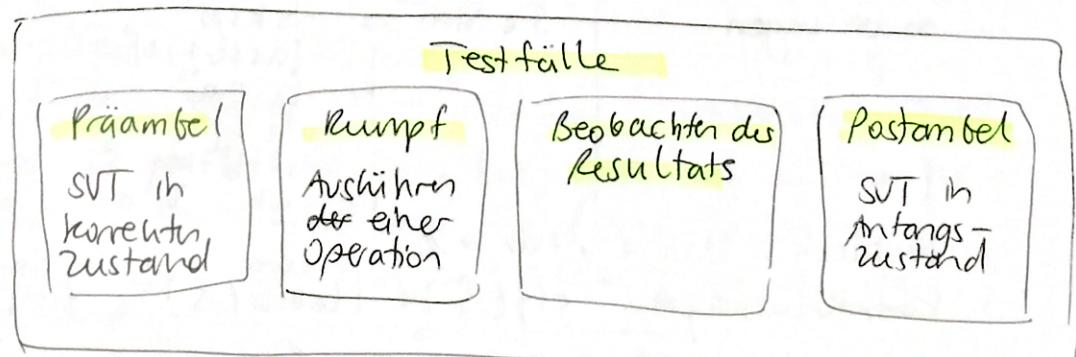
Design by contract, Verträge durch Annotationen für Vorbedingungen, Nachbedingungen, Invarianten bei Klassen / Methoden / ...  
 $\text{@ requires}$        $\text{@ ensures}$        $\text{@ invariant}$

jmlunit: für Methodentest, erzeugt Testrahmen, füllen der Testdaten in -JML-TestData

Idee: keine blind gewählten Testdaten sondern solche, die dem Zustandsmodell nach Sinn machen

## KOMPONENTENTEST MIT OPERATIONSSPEZIFIKATION

Testziel  
aus Adäquatheitstestkriterium  
> Bedingungen  
> alle Fingaben



# MODELLBASIERTES TESTEN

Idee: Modell für Generierung und Validierung von Testfällen  
 Konkretisierung von Modell zu SVT-Ausgaben  
 Abstraktion von SVT zu Modell-Ausgaben

## DON'TS

- > 1 Modell für Generierung von Testfällen UND Code
- > Spezifikation für Implementierung, Generierung eines Modells aus dem Code, Modell für Testfälle
- | DOS Küller
- | > Spezifikation für Implementierung, Modell für Testfälle, Abstimmung ~~spezif.~~ Modell auf Spezifikation
- | > 2 Modelle, 1 für Testfälle, 1 für Implementierung, unabhängig

# KONTROLLFLUSSTESTEN

## KONTROLLFLUSSGRAPH

- > Knoten: Code, meist Basisblöcke = maximale Folge Anweisungen, sodass jeder eingehende Kontrollfluss auf die erste Anweisung, jeder ausgehende Kontrollfluss aus der letzten Anweisung
- > Kanten: Kontrollfluss vom Ende einer Region zum Beginn einer anderen
- > ausgezeichnete Quelle und Senke(n)

zusammenhängend: jede Knoten von Quelle aus erreichbar, (eine) Senke von jedem Knoten aus erreichbar

## IMPERATIVE SPRACHE IMP<sup>Lab</sup> mit Annotation

- > ganzzahlige Variablen  $x \in \text{IntVar}$
- > Anweisungen  $\hat{S} \in \text{Stm}^{\text{Lab}} :=$
- |  $a \in A\text{Exp}; b \in B\text{Exp}$
- |  $[\text{skip}]^L$
- |  $[\text{assert } b]^I$
- |  $[x := a]^I$
- |  $S_1; S_2$
- |  $\text{if } [b]^I \text{ then } \hat{S}_1 \text{ else } \hat{S}_2 \text{ fi}$
- |  $\text{while } [b]^I \text{ do } \hat{S} \text{ od}$

> Zustände State = IntVar  $\rightarrow \mathbb{Z}$

> Kontrollflussgraph:  $\text{cfg}(\hat{S}) = (\text{labels}(\hat{S}), \text{flow}(\hat{S}))$

Knoten  
 berechnet alle Marken einer annotierten Anweisung

Kanten  
 berechnet, von welcher Menge aus man zu welchen anderen Mengen kommt

## ANWEISUNGSBÜERDECKUNG C0

Kontrollflussgraph aus Basisblöcken

Programm	Testsuite	TER-1
C statement	(P, T) =	
# von T ausgeführte Anweisungen von P		
# Anweisungen von P		

Schwaches, minimalis Kriterium, fehlende Pfade werden nicht erkannt

## PFA DÜBERDECKUNG

C4, C $\infty$

unendlich viele Pfade möglich

Beschränkung auf endliche Pfadanzahl aus Äquivalenzlassen, mehrere Möglichkeiten (1-3)

- (1) Endlicher Pfad in  $\hat{S}$   $p = \langle l_1, \dots, l_n \rangle$  oder  $p = \langle l_1, l_2, \dots \rangle$   
 Folge in Labels( $\hat{S}$ ) mit  $n \geq 1$ ,  $(l_i, l_{i+1}) \in \text{flow}(s)$   
 $|p| = n$   $|p| = \infty$

Ausführungs Pfad path( $\hat{S}, \sigma$ ) =  $\langle l_1, l_2, \dots \rangle$  von  $\hat{S}$  ab Zustand  $\sigma$   
 $\langle \hat{S}, \sigma \rangle \xrightarrow{\text{L}} \langle \hat{S}_1, \sigma_1 \rangle \xrightarrow{\text{L}} \dots$  nach strukturell-operationaler Semantik

## 1 Linear code sequence and jump LCSAJs

= Folge von sequentiellen Anweisungen ab Sprungziel bis zu einem Sprung inklusive

Unterschied zu Basisblöcken: LCSAJs können sich überlappen, weil jumps innerhalb der sequentiellen Anweisungen erlaubt

TEST EFFECTIVENESS RATIO TER(n+2) :

alle Kombinationen von  $n \geq 1$  aufeinanderfolgenden LCSAJs

TER-3	# von T ausgeführte LCSAJs in P
	# LCSAJs in P

## 2 Zyklomatisches Testen

Auswahl einer Basis von Pfaden, Größe der Basis = zyklomatische Zahl

> obere Schranke für # Testfälle, um Zweigüberdeckung zu erreichen

> untere Schranke für # Pfade durch den Kontrollflussgraphen

> mit 'virtuellen' Kante von Senke zu Quelle (dadurch stark zusammenhängend):

$$(\# \text{Kanten} + 1) - \# \text{Knoten} + 1$$

Subsumiert

## ZWEIGÜBERDECKUNG C1

entspricht Kantenüberdeckung

Programm	Testsuite	TER-2
C Branch	(P, T) =	
# von T ausgeführte Zweige von P		
# Zweige von P		

keine Berücksichtigung von Datenfehlern, Schleifen werden nicht genügend getestet

### 3 Boundary - Interior

**BOUNDARY:** alle Pfade ohne Schleifenwiederholung, Schleifen werden max. 1x betreten

**INTERIOR:** alle Pfade mit Schleifenwiederholung, Schleifen werden max. 2x betreten

**K-STRUKTURIERTE PFADÜBERDECKUNG:** Schleifen werden max. k-mal betreten

### Modifizierte Boundary - Interior - Überdeckung

beim zweiten Durchlauf einer äußeren Schleife keine Pfadunterscheidung für innere Schleifen

## GENERIERUNG VON TESTFÄLLEN

### Transformation

transformiere  $\hat{S}$  zu  $\hat{S}'$ , das den gewünschten Pfad repräsentiert

- > Schleife  $\text{while } [b]^l \text{ do } \hat{S} \text{ od:}$   
 $[\text{assert } b]^{l,1}; \hat{S}^{-1}; [\text{assert } b]^{l,2}; \dots; \hat{S}^{-k}; [\text{assert not } b]^{l,k+1};$
- > Verzweigung  $\text{if } [b]^l \text{ then } \hat{S}_1 \text{ else } \hat{S}_2 \text{ fi}$   
 $[\text{assert } b]^l; \hat{S}_1; \text{ oder}$   
 $[\text{assert not } b]^l; \hat{S}_2;$

### wp-Kalkül

berechne schwächste Voraussetzung  $\text{wp}(\hat{S}', tt)$ , so dass  $\hat{S}'$  erfolgreich terminiert; ergibt alle Zustände  $\sigma$ , sodass  $\hat{S}$  angescannt auf  $\sigma$  in einem  $\sigma' \in \text{state}$  mit  $\sigma' \models \varphi$  terminiert; Rückwärtsanalyse durch den Code ausgehend von Nachbedingung

wie strong diamond, nur für deterministische Programmiersprache

## BEDINGUNGSTESTEN

### EINFACH-BEDINGUNGSÜBERDECKUNG

jede atomare Bedingung in einer Bedingung muss mit tt und mit ff belegt werden	Bsp. a & b
a	b
tt	ff
ff	tt

### MINIMALE MEHRFACH-BEDINGUNGSÜBERDECKUNG

jede Teilbedingung einer Bedingung muss zu tt und zu ff auswerten, d.h. auch die gesamte Bedingung	Bsp. a && b
a	b
tt	tt
ff	ff

auch nicht das gelbe von ti

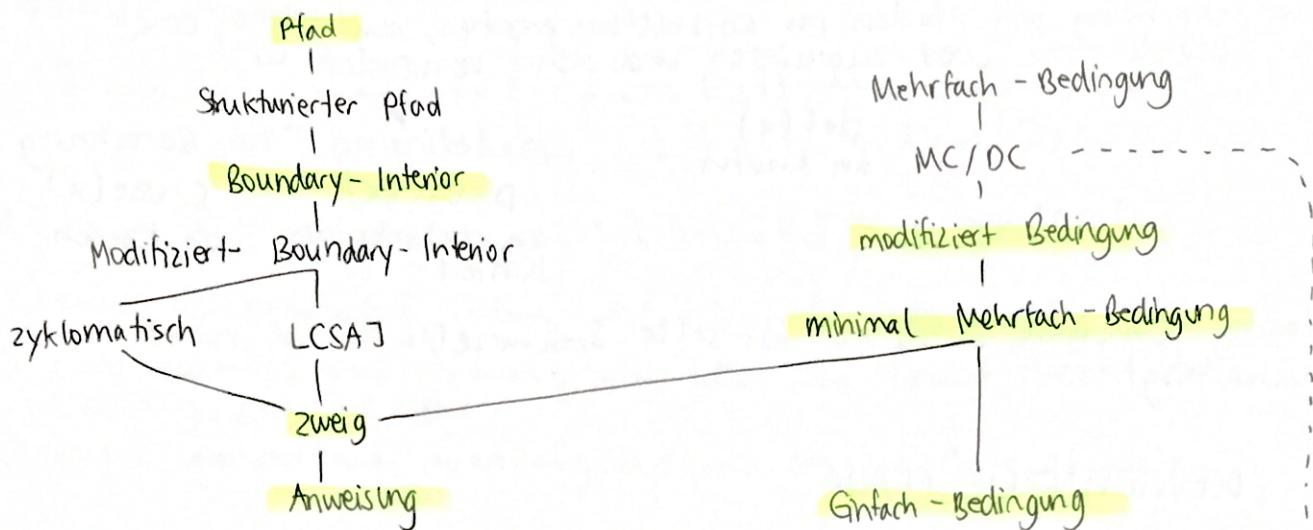
### MEHRFACH-BEDINGUNGSÜBERDECKUNG

alle Belegungsmöglichkeiten der atomaren Bedingungen	Bsp. a && b
a	b
tt	tt
tt	ff
ff	tt
ff	ff

### MODIFIZIERTE BEDINGUNGSÜBERDECKUNG

Flippen einer atomaren Bedingung muss das Ergebnis der Bedingung beeinflussen	Bsp. a && b
a	b
tt	tt
tt	ff
ff	tt

# VERGLEICH STRUKTURELLER TESTMETHODEN



MC/DC: Zweigüberdeckung + modifiziert Bedingungsüberdeckung

- > jeder Start / Endpunkt wird abgedeckt
- > jede Bedingung werft min. 1x zu tt / ff aus
- > jede atomare Bedingung werft min. 1x zu tt / ff aus
- > jede Belegung der atomaren Bedingung beeinflusst die Bedingung

## FUZZING

Black Box

= Generierung unerwarteter / zufälliger Eingaben,  
Start oft noch mit gültigen Eingaben

## KONKOLISCHES TESTEN

White Box

mit Fuzzing verwendet

= auch zufällige Eingaben,  
Aufsammeln von Constraints bei bedingten Anweisungen



auch Directed Automated Random Testing DART

- + man braucht kein Testorakel  
wenn man lange genug probiert, findet man so einen schlechten Input

## WHITEBOX FUZZING

White Box

= Fuzzing + DART

generative Suche möglichst viele neue Testfälle (Kinder) pro Elternlauf,  
Bewertung anhand Anweisungsüberdeckung, bester Testfall als Eltern

# DATENFLOWSSTESTEN

white  
box

- Betrachtung von Pfaden im Kontrollflusgraphen, auf denen einer Variable ein Wert zugewiesen und dann verwendet wird

def(x)  
an Knoten

in Bedingung → für Berechnung  
p-use(x) c-use(x)  
an ausgehenden an Knoten  
Knoten

Aliasing (mehrere Bezeichner für selbe Speicherzelle) wird nicht berücksichtigt?

## DEFINITIONSFREIE PFADE

def-clear<sub>p</sub>(x)

def-clear<sub>c</sub>(x)

Pfade für x, sodass x nicht erneut definiert wird

## PFADE ZUR VERWENDUNG

dpu(v, x)

dcu(v, x)

Menge an Knoten/Knoten, sodass Pfad ab v bis Verwendung von x definitionsfrei

## KRITERIEN

Pfadmenge P adäquat, wenn

↑ subsumiert Zustand + Anweisung

all-defs : alle Definitionen werden angeschaut

↑ subsumiert d.h. es gibt für jede Definition von x einen Pfad mit mindestens einer Verwendung in P

all-uses : alle Def-Use-Paare werden angeschaut

↑ subsumiert d.h. es gibt für jede Definition von x einen Pfad für jede Verwendung in P

all-du-paths : alle Pfade zwischen Def und Use werden angeschaut

↑ subsumiert d.h. jeder mögliche Pfad für jede Definition von x zu jeder Verwendung von x ist in P

Pfadüberdeckung

## IMP Lab

### DEF/USE - PAARE

indirekte Definition

clear(S)(x, l, l') keine Definition von x auf Pfad von l nach l'

vd(S)(x, l') Def-Use-Paar, Use in l', Def davor, liefert alle Def

du(S)(x, l) Use-Def-Paar, Def in l, Use danach, liefert alle Use

### REACHING DEFINITIONS

konstruktive  
Definition

[x := a]' erreicht Programm Punkt, wenn in diesem Punkt x zuletzt in l definiert

rd(S)exit(l) was aus l rausfließt

rd(S)entry(l) was in l reinfließt =

was aus Vorgängern rausfließt

vd(S)(x, l') = { l | (x, l) ∈ rd(S)entry(l') } falls use(S)(x, l')

∅ const

-10-

## REACHING - DEFINITIONS - ANALYSE

## Datenflussgleichungen

$$rd(\hat{S})_{\text{entry}}(l) = \begin{cases} \{(x, ?) \mid x \in \text{vars}(\hat{S})\} & \text{falls } l = \text{init}(S) \\ \cup \{rd_{\text{exit}}(l') \mid (l'; l) \in \text{flow}(\hat{S})\} & \text{sonst} \end{cases}$$

$$rd(\hat{S})_{exit}(l) = (rd(\hat{S})_{entry}(l) \setminus kill[B]^l) \cup gen([B]^l) \quad \text{für } [B]^l \in \hat{S}$$

- damit Gleichungssystem, `rdentry` und `rdexit` für jedes Label
  - dann verwendet man auf magische Weise den Fixpunkt-Operator, um das Gleichungssystem zu lösen
  - daraus versucht man dann möglichst alle Def-Use-Paare abzuleiten

### ERWEITERTE DEF/USE - PAARE noch komplizierter

- man schaut nicht mehr nur Variablen an, sondern Speicherzellen, um Aliasung in den Griff zu kriegen

## Äquivalenzrelation $\alpha$

$(r, r') \in \alpha$ , falls  $r$  und  $r'$  dieselbe Speicherzelle referenzieren; Äquivalenzrelation

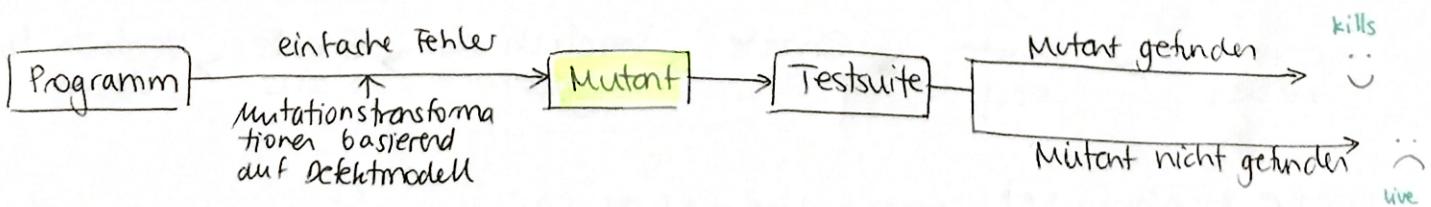
Knoten werden mit Alias-Relationen markiert

Ablese erweiterter Def/Use-Pfade durch Zusammenfassen von Pfaden mit gleicher Alias-Relation an Programm punkten

## DIVERSIFIZIERENDES TESTEN

# MUTATIONEN - TEST

so ein bisschen der Test für den Test



Testsuite T adäquat für Programm P, wenn alle von P unterscheidbaren Mutationen erkannt werden.

Annahmen:

- > erfahrene Programmierer erstellen fast korrekt Programme
  - > Identifizierbarkeit komplexer Fehler aus Erkennung einfacher Fehler möglich

## BACK-TO-BACK-TESTEN

Vergleich mehrere Programmversionen durch Ausgabevergleich , Ausgaben nicht gleich  
→ mindestens eins der Programme nicht korrekt

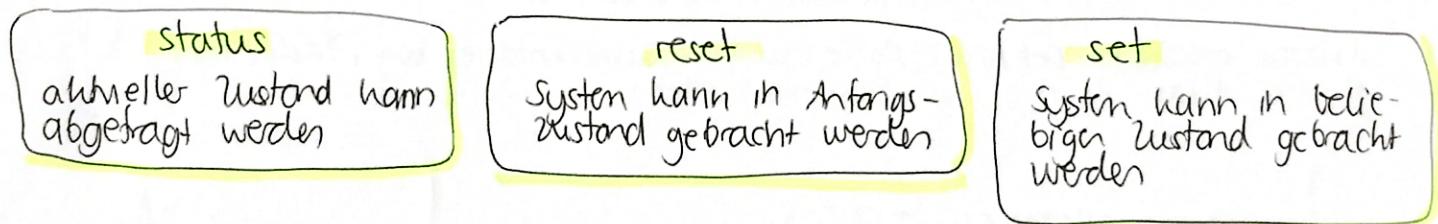
## EVALUATION UND EMPIRIE

- > fehlende Anweisungen / Zweige / Variablendefinitionen / Variablenreferenzen können nicht durch Tests aufgedeckt werden
- > LCSAJ hat die beste empirische Fehlererkennungsrate (85%)
- > die Fehleraufdeckungsrate steigt stark im Bereich zwischen 90% und 100% Überdeckungsgrad
- > auch 100% Überdeckung sind kein verlässlicher Indikator für Testeffektivität
- > konstruktive Qualitätsicherung deswegen umso wichtiger

## KONFORMANZ-TESTEN

Voraussetzung: Zustandsübergangsdiagramme für Spezifikation und Implementierung  
gleiche Zustände, Eing/Ausgaben und Übergänge können unterschiedlich sein  
Ziel: beobachtbares Verhalten einer Implementierung äquivalent zu Spezifikation

### EIGENSCHAFTEN DER IMPLEMENTIERUNG



### TESTEN mit status + reset + set Testsuite der Länge $4 \cdot |Q| \cdot |I|$

Für alle Zustände  $q \in Q$  und Eingaben  $i \in I$ :

Eingabe  $\text{reset} > \text{Eingabe } \text{Set } q > \text{Eingabe } i > \text{Vergleich der Ausgaben} > \text{Eingabe } \text{status} > \text{Vergleich der Status}$

### TESTEN ohne set Testsuite der Länge $1 + 2 \cdot |Q| \cdot |I|$

set  $q$  wird durch Transitionstour ersetzt

### TESTEN ohne reset $\rightarrow$ jede minimale Zustandsmaschine besitzt eine Homing-Folge

reset wird durch Homing-Folgen ersetzt, Folge von Eingaben, sodass Zustand nach Eingabe vollständig von erzeugter Ausgabe bestimmt

### TESTEN ohne status $\rightarrow$ existieren nicht immer

status wird durch UTG-Folgen oder Separationsfolgen ersetzt

Eingabefolge, Zustand  $\xleftarrow{\text{start-}} \nwarrow$  höchst vollständig von zugehöriger Ausgabe ab

$\Delta'$  unvollständig!

$O(|Q|^3 \cdot |I|)$

vollständig!  $\rightarrow$  gleiche Eingabe, unterschiedliche Ausgabe

# TESTEN OBJEKTOIENTIERTER SOFTWARE

## ZUSÄTZLICHE SCHWIERIGKEITEN:

- > Zustandsabhängiges Verhalten
- > Kapselung
- > Vererbung
- > Polymorphie und dynamische Bindung

# TESTEN NEBENLÄUFIGER SOFTWARE

## ZUSÄTZLICHE SCHWIERIGKEITEN:

- > Deadlocks
- > Atomaritätsverletzungen
- > Ordnungsverletzungen bei Zugriffsordnung auf den Speicher
- > race conditions wegen unkontrollierbarer events

## TESTSTRATEGIE

Ziel: Testen aller Schedules → Erhöhen von Verzögerungen

- ↳ manuell
- ↳ multithreaded TC, globale Uhr
- ↳ IMUnit, Ereignisse

# INTEGRATIONSTEST

ZUSÄTZLICHE SCHWIERIGKEITEN: Schnittstellen-Fehler, z.B. falsche/inkonsistente Datenstrukturen, falsche Methodenaute, Speichermanagement

## ARTEN VON INTEGRATIONSTESTS

BIG-BANG-INTEGRATION	BOTTOM-UP-INTEGRATION	TOP-DOWN-INTEGRATION
<ul style="list-style-type: none"> <li>kleines, stabiles SVT, letzter Ausweg, keine Methode im engeren Sinn, Fehlerdiagnose schwierig</li> </ul>	<ul style="list-style-type: none"> <li>robuste und stabile Schnittstellen, Beginn bei Komponenten mit wenig Abhängigkeiten, Einführung von Testfiebern</li> <li>- aufwendiger Testrahmen</li> <li>+ frühzeitig möglich</li> </ul>	<ul style="list-style-type: none"> <li>inkrementelle Entwicklung, Beginn bei Top-level-Kontrollkomponenten, ebenenweise Einführung von Teststubs für untere Komponenten</li> <li>- aufwendige Stubs</li> <li>+ frühzeitig möglich</li> </ul>