Testen
SS 2019
Automatisches Testen

Prof. Dr. Dirk Beyer,
Thomas Lemberger

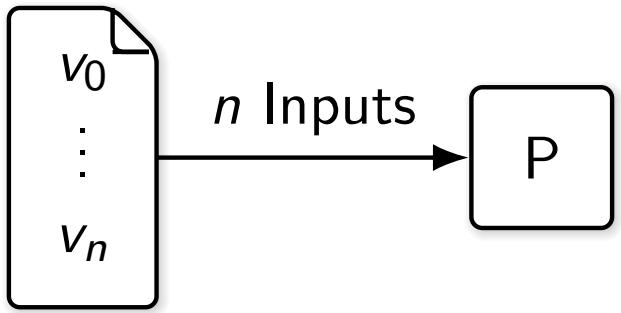
Outline

- [ ] Spezifikation von Testzielen: FQL
- [ ] Automatische Testerzeugung
  - [ ] Blackbox Testing (Random)
  - [ ] Greybox Testing (Distanzmetriken)
  - [ ] Whitebox Testing (Model Checking, Symbolic Execution)
- [ ] (Mutations-Testen)
- [ ] Tutorial: PRTest, AFL-fuzz, KLEE

FQL

Automatische Testerzeugung

Tests



- Testfall-Beschreibung durch Testvektor $\langle v_0, \ldots v_n \rangle$ mit n Programmeingabe-Werten v_0 bis v_n.
- Menge von Testfällen: Test Suite.

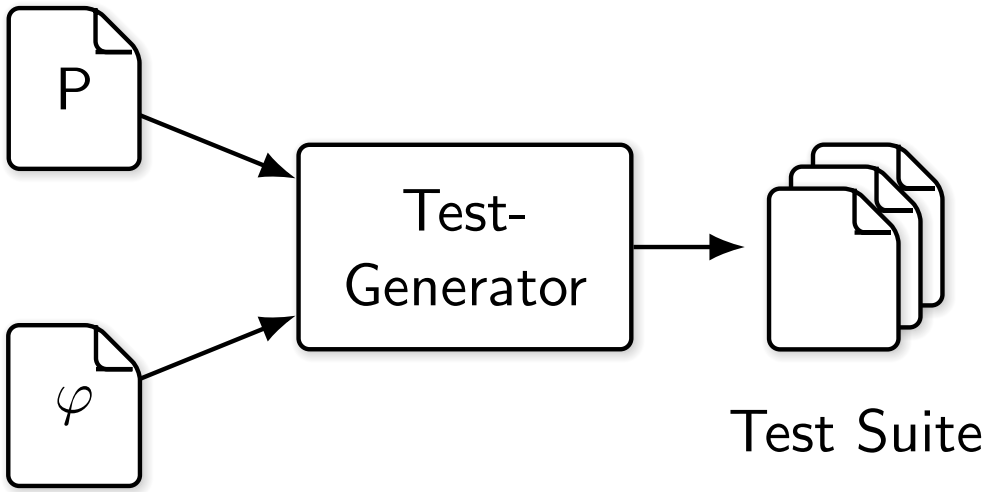
Tests

```
int main() {
  int x = input();
  int y = input();
  while (y > 0 && x < 10) {
    if (x > 0) {
      x--;
    } else {
      x++;
    }
    y--;
  }
}
```

- Branch-Coverage: $\{\langle 1, 2\rangle\}$
- Condition-Coverage: $\{\langle 1, 2\rangle, \langle 10, 2\rangle\}$

Test Generator

- Definition Test-Generator:

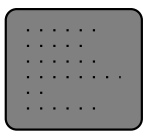


- Input:
  - Program-under-test P
  - Test-Goal Spezifikation φ
- Output: Test Suite (Menge von Testfällen)

Überblick

- 3 Klassen von Test-Generatoren:

Blackbox: Verwendung von Inputs/Outputs



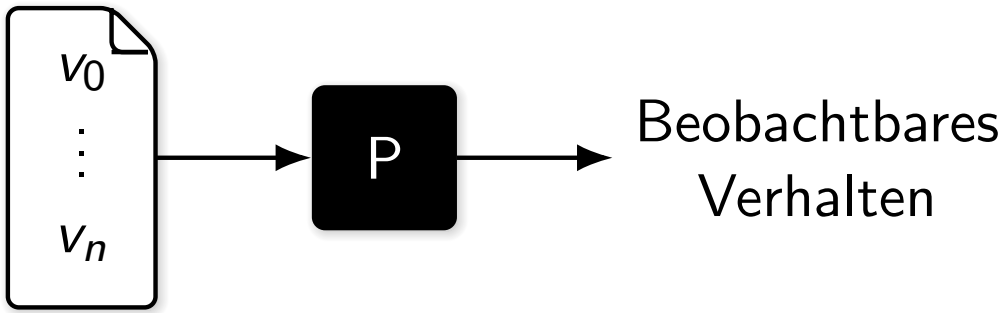
Graybox: + Verwendung der Programm-Struktur (Syntax)

Whitebox: + Verwendung der Programm-Semantik

Blackbox Testing

Blackbox

- Program-under-Test wird als Blackbox betrachtet



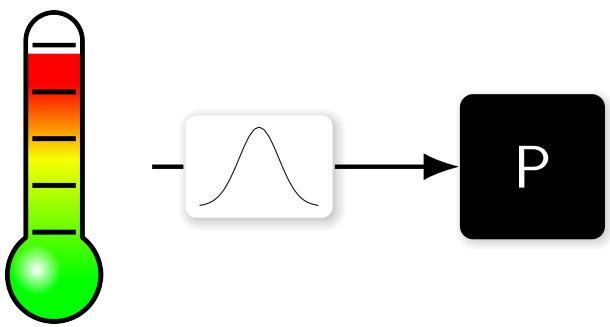
- Beobachtbares Verhalten z. B.:
  - Programmausgabe
  - Crash
  - Ressourcen-Verbrauch
- Manchmal einzige Testmöglichkeit, z. B. für kompilierte Programme, Libraries, remote Server, etc.
- Extrem schnell (brute force)
- Extrem^2 geringe Wahrscheinlichkeit, seltene Pfade abzudecken

Random Testing

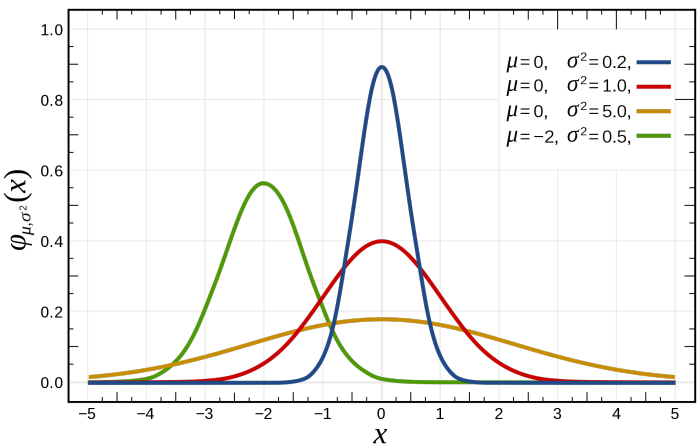
- Methode zum Blackbox-Testing
- Zufällige/beliebige Erzeugung von Testwerten
- Unabhängig von vorherigen Testausführungen
- Unterschiedliche Strategien:
  - Nach Zufallsverteilung (z.B. gleichmäßig)
  - Durch Evolution existierender Inputs
  - Grammatik-basiert

Random Testing: Zufallsverteilung

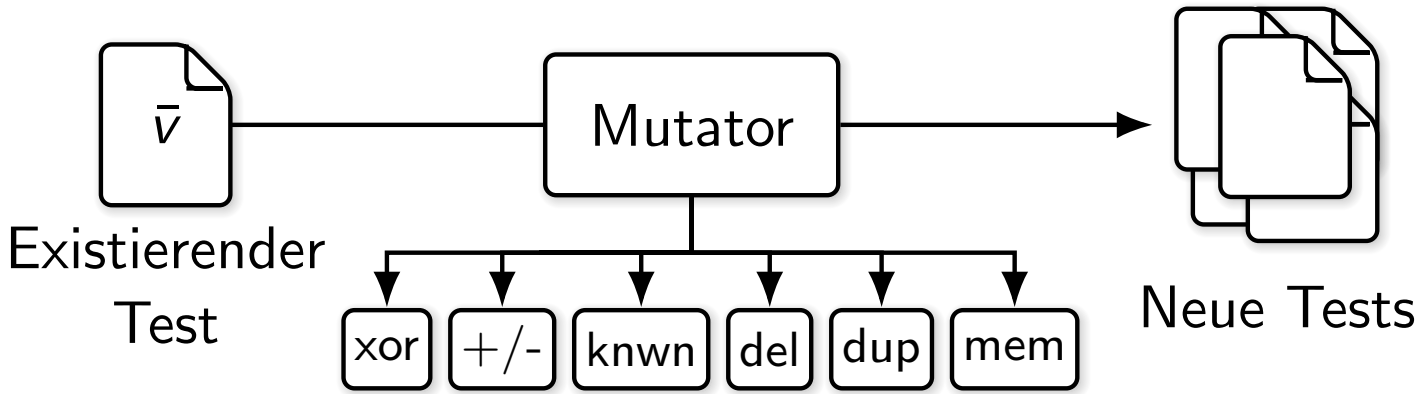
- Auswahl von Test-Inputs aus möglichen Input-Werten nach Zufallsverteilung
- Möglichkeiten:
  - Allgemeine Verteilung (gleichmässig, normal, etc.)
  - Erwartete Verteilung (kann von typischen Verteilungen stark abweichen)



Temperatur-
Sensor



Random Testing: Evolutionär



- Anderer Name: Fuzzing
- Test-Generator erhält Anfangsmenge von Tests
- Versucht durch unterschiedliche Mutationen, weitere, vielfältige Tests zu erhalten
- Reihenfolge der Mutatoren (Mutations-Strategie) oft festgelegt
- Mutations-Operatoren: Deterministisch oder Zufällig
- Meist auf Bit- oder Byte-Ebene

Random Testing: Input-Evolution (Operatoren)

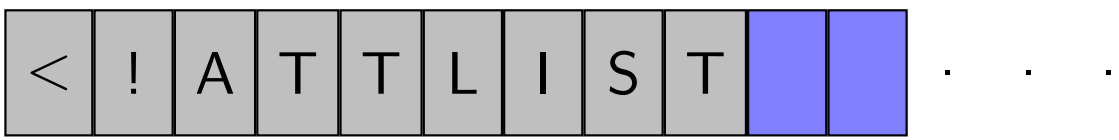
- Bitflip (xor auf einzelne Bits oder Bytes)
- Arithmetische Operationen (+/- x auf einzelne Bytes oder Wörter)
- Known Values
  - Wörter in Eingabe durch typische, „kritische“ Zahlen ersetzen
  - 0, -1, INT_MAX, INT_MIN, etc.
- Deletion (Bytes oder ganze Blöcke (max. 1 kB))
- Duplication (Blöcke)
  - Blöcke bestimmter Größe kopieren
  - Andere Blöcke überschreiben oder Input verlängern
- Bit Masks
  - Maskiere bestimmte, wichtige Teile des Inputs, so dass diese nicht mehr verändert werden
- Splicing (Unterschiedliche Testteile zusammenfügen) (auch genannt Crossover)

Random Testing: Input-Evolution (Beispiel Bit Mask)

```
int parseXml(char* input) {
  if (strncmp(input, "<!ATTLIST", 9) ) {
      // further processing of input
  }
}
```

- **Nur** Input relevant, der mit <!ATTLIST beginnt

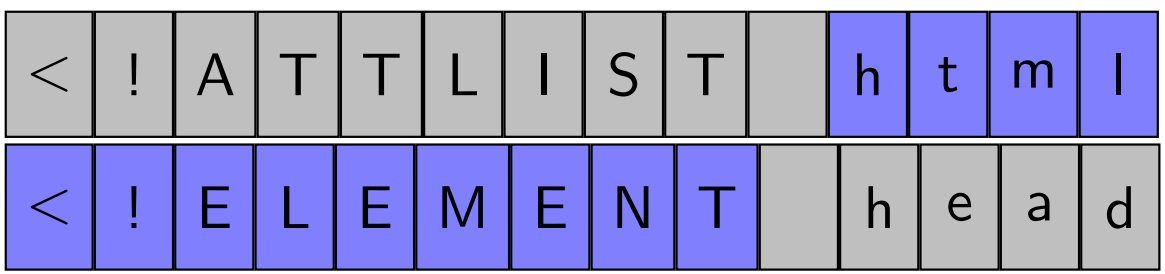
$\Rightarrow$ Maskiere erste 9 Characters in gültigem Test-Case, damit sie nicht mutiert werden



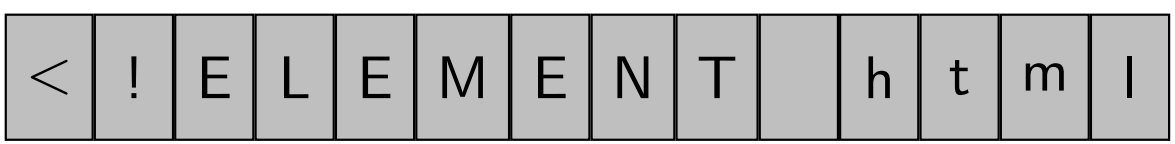
Random Testing: Input-Evolution (Beispiel Splicing)

```
int parseXml(char* input) {
  if (strncmp(input, "<!ATTLIST", 9)) {
      // ...
  } else if (strncmp(input, "<!ELEMENT", 9) {
      // ...
  }
  if (strstr(input, "html") != NULL) {
    // handle html
  }
}
```

- Existierende Testfälle:

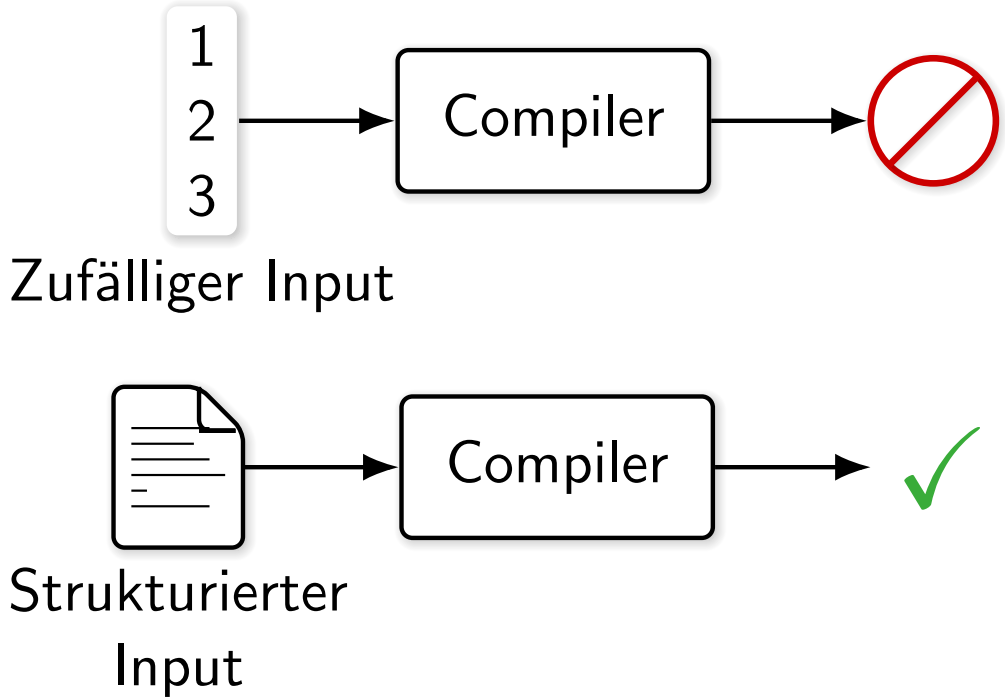


- Neuer Testfall:



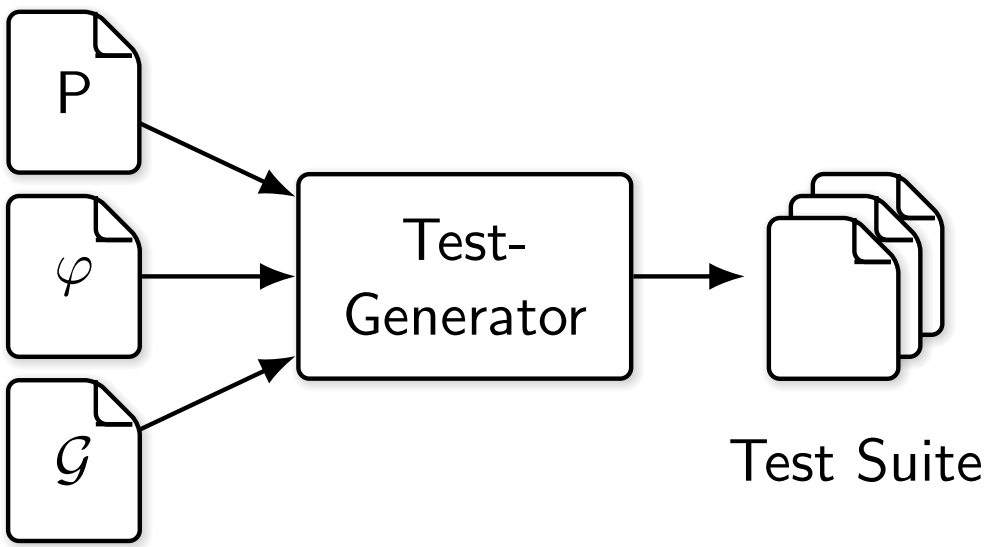
Random Testing: Grammatik-basiert I

```
void compile(char* input) {
  if (!isValid(input)) {
    exit(1);
  } else {
    // all the logic
  }
}
```



Random Testing: Grammatik-basiert II

- Oft erwarten Programme eine bestimmte Struktur ihrer Inputs
- Bildbearbeitung, Browser, Compiler, Protokolle, etc.
- Zufällige Inputs erfüllen Struktur mit extrem hoher Wahrscheinlichkeit nicht
  - $\Rightarrow$ werden direkt abgelehnt
  - $\Rightarrow$ kein sinnvolles Testen möglich
- Lösung: Generiere Inputs entlang Grammatik ($\mathcal{G}$)



Random Testing: Grammatik-basiert (Beispiel)

Beispiel-Grammatik:

```
<expr> ::= <term> "+" <expr>
         |  <term>

<term> ::= <factor> "*" <term>
         |  <factor>

<factor> ::= "(" <expr> ")"
           |  <const>

<const> ::= 0 | 1 | 2 | ...
```

Zufällige Generierung eines Test:

1. Initialisiere Test mit Start-Symbol ($test = <\text{expr}>$)
2. Solange Non-Terminale in Test vorhanden: Ersetze jeden Non-Terminal durch zufällig gewählte, rechte Seite.
3. Sobald keine Non-Terminale mehr vorhanden: Gebe Test zurück.

Random Testing: Challenges

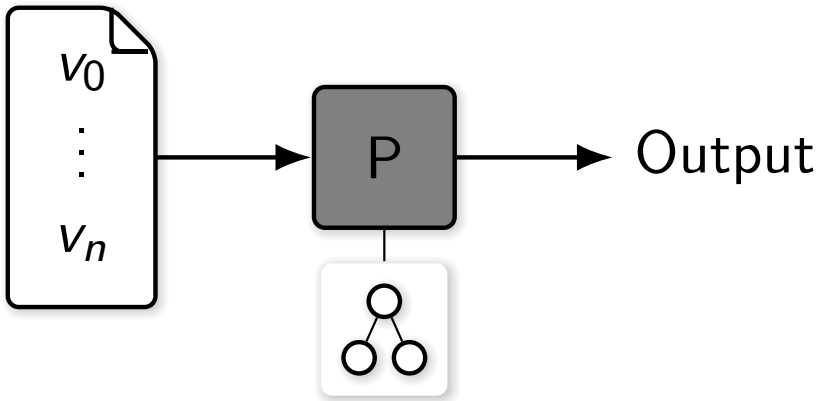
- Test-Suites wachsen extrem schnell

$\Rightarrow$ Extrem viele unnötige/sinnlose Tests (Test-Suite Reduction)

- Für evolutionäres Testen: Test-Vektoren wachsen (Trimming)

Graybox Testing

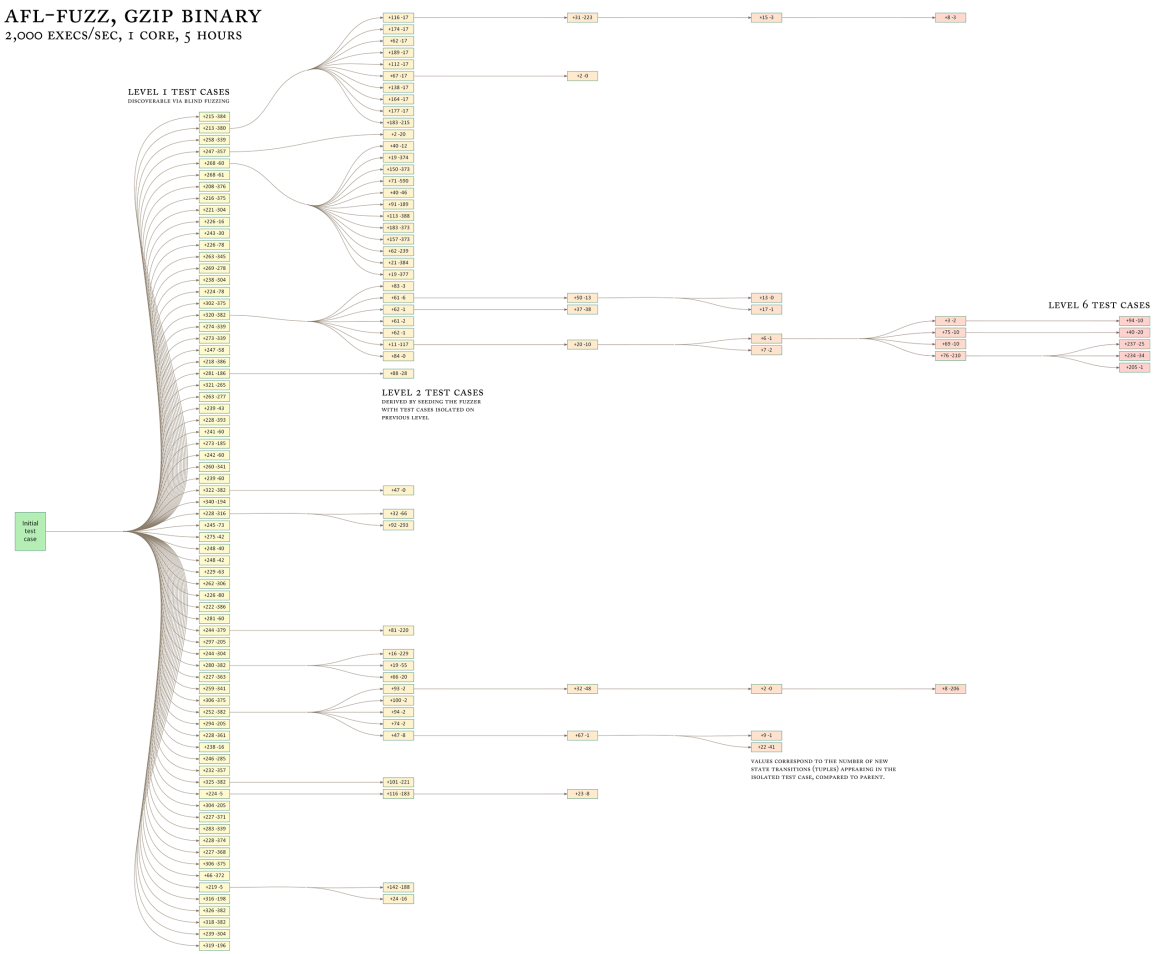
Graybox Testing: Überblick



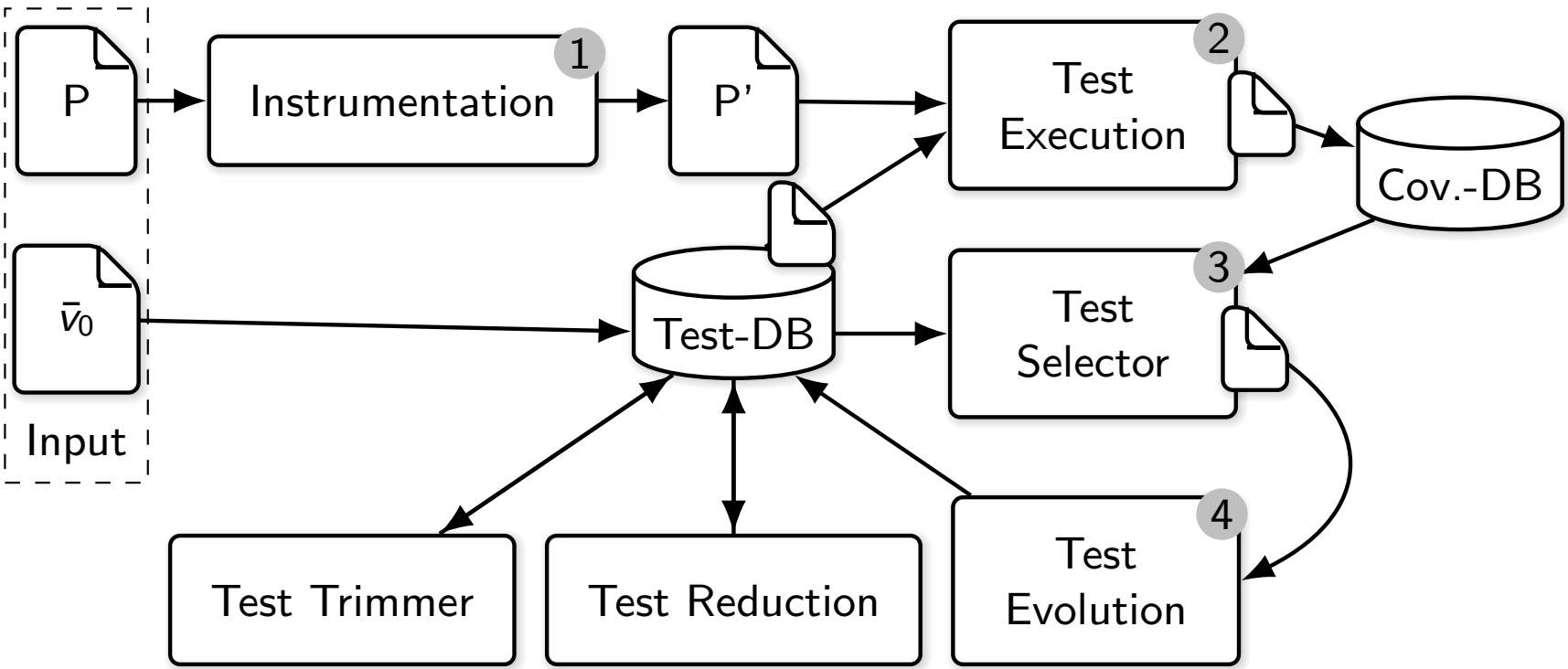
- Nutzt Programm-Syntax (CFA) und konkrete Testabdeckung
- **Keine** Semantik!
- Test-Erstellung: weiterentwickelte Blackbox-Techniken
- Eingeschränkt auch auf kompilierten Programmen verwendbar
- Extrem schnell (directed brute force)
  - Balance zwischen Brute Force und Testoptimierung
- Potential, auch seltene Pfade abzudecken

Graybox Tester: AFL

- Beispiel-Ansatz: American Fuzzy Lop (AFL)
  `http://lcamtuf.coredump.cx/afl/`
- Evolutionäres Testen mit Decision-Coverage-Measurement + Hit Counts



AFL: Workflow



AFL: Instrumentation

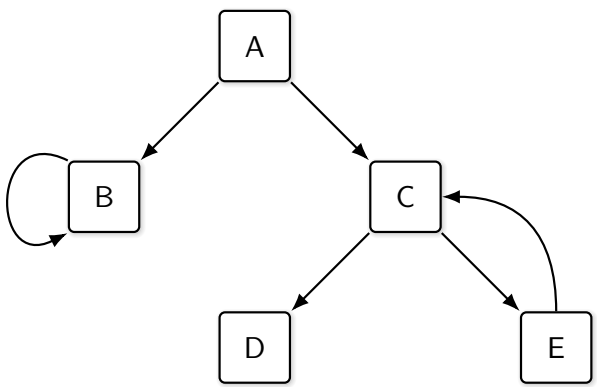
- Zu Beginn jedes Code-Blocks N :

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

- shared_mem wird als zusätzlicher Eingabe-Parameter zu Programm hinzugefügt

shared_mem

Transition	Hit Count
A-B	1
B-B	3
A-C	0
C-D	0
C-E	0
E-C	0



- Random IDs für Blöcke: Kollisionen möglich, aber dafür keine aufwendige Berechnung nötig

AFL: Test Execution + Selection I

"hmm, this does something new!"

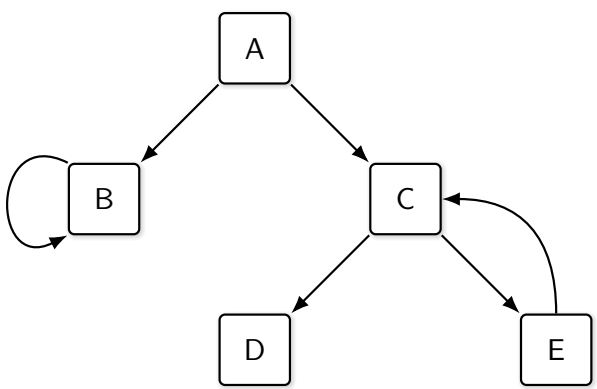
- Ausführung von noch nicht ausgeführten Tests (pro Evolutions-Level)
- Resultat pro Ausführung: shared_mem-Map mit Hit Counts der Branch-Transitions (+ Test-Laufzeit)
- Falls mind. eine bisher unbesuchte Branch-Transition getroffen wurde: Test-ID und Map wird in Coverage-Datenbank gespeichert
- Beispiel:

  A-C, C-E ✓

  A-C, C-D ✓

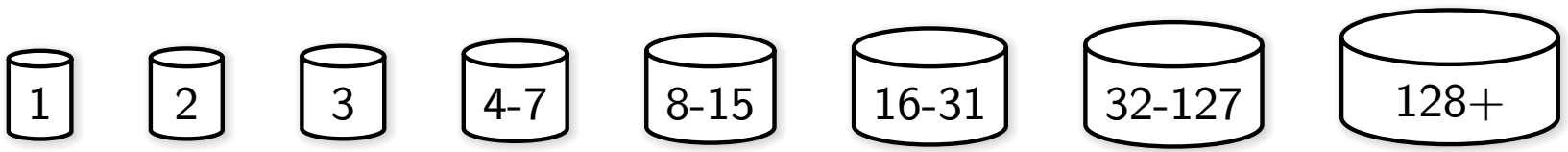
  A-C, C-E, E-C, C-D ✓

  A-C, C-E, E-C, C-E *X*



AFL: Test Execution + Selection II

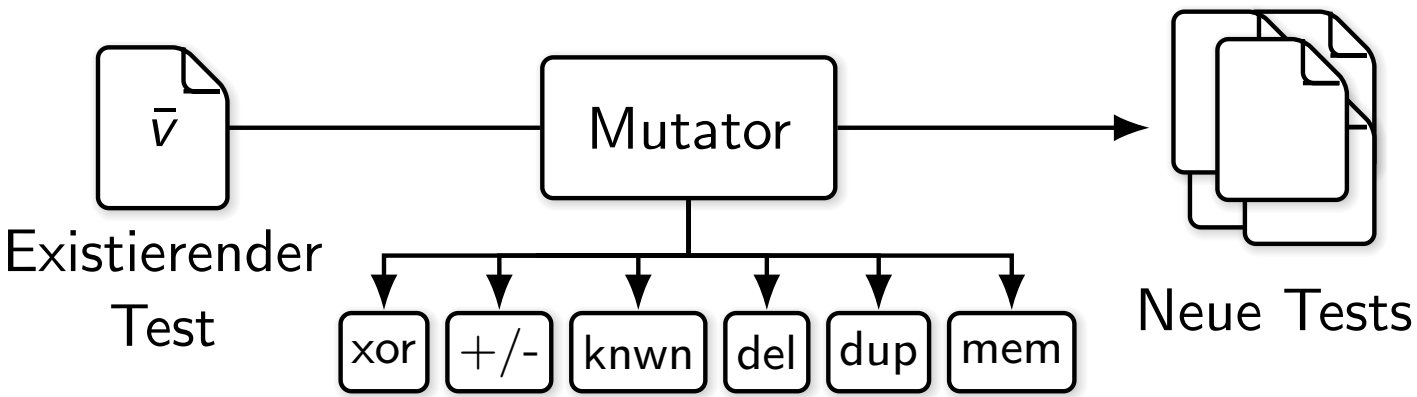
- Weitere Metrik um Tests auszuwählen:
- Branch-Hit-Counts werden in Buckets eingeteilt



- Tests, die bei einem Hit-Count in anderen Bucket als bisherige Tests fallen, gelten zusätzlich als interessant
- Alle Tests ohne neue Branch-Transition oder neuen Bucket werden verworfen
- Verbleibende Tests werden für Evolution verwendet

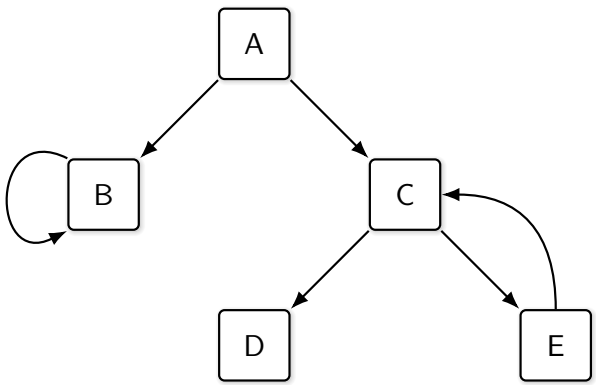
AFL: Test Evolution (Fuzzing)

- Zwei Phasen:
  1. Deterministische Mutationen
  2. Zufällige Mutationen
- Alle bereits vorgestellten Mutations-Operatoren



- + Splicing von zwei Tests an zufälliger Stelle

AFL: Test Reduction



- Tests in Test-Set:
  1. A-C, C-E
  2. A-C, C-D
  3. A-C, C-E, E-C, C-D
- Test 3 deckt alle Branches ab

$\Rightarrow$ Nach AFL-Logik überflüssig, 1 und 2 zu mutieren

$\Rightarrow$ Periodische Analyse der bestehenden Test-Suite und Auswahl ausreichender und „guter“ Tests

AFL: Test Reduction Algorithm

1. Für jeden ausgeführten, aber noch nicht mutierten Test:
   Vergebe Score basierend auf Ausführungszeit und
   Test-Vektor-Länge
2. Sortiere Tests aufsteigend nach Score
3. Für jede Branch-Transition, die bereits abgedeckt wurde, und für die noch kein Test ausgewählt wurde:
   3.1 Wähle Test mit niedrigstem Score
   3.2 Wähle selben Test für alle anderen Transitions, die er abdeckt

- Auswahl ist ca. 10-20 % aller Kandidaten (in Experimenten)
- Nicht ausgewählte Tests werden **nicht** verworfen, sondern nur mit hoher Wahrscheinlichkeit ausgelassen

AFL: Test Trimming

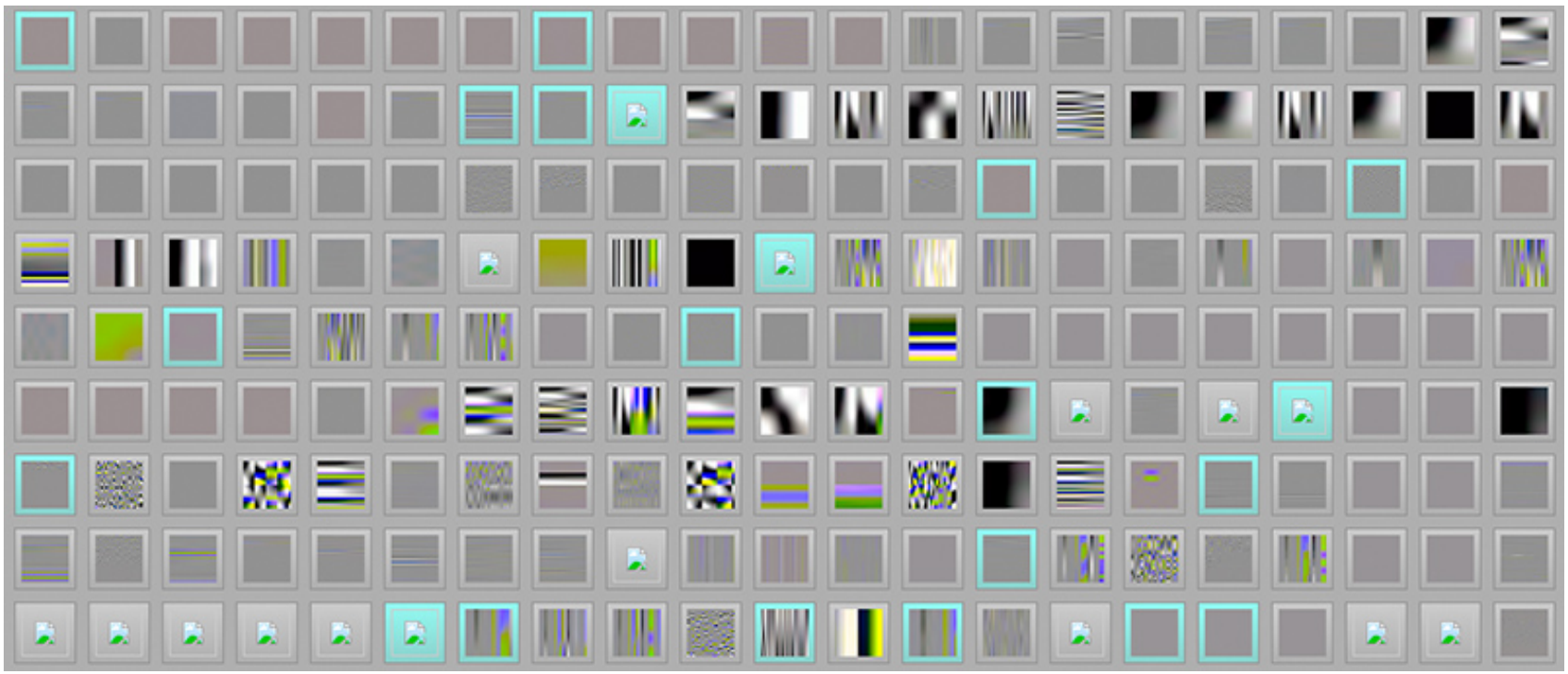
- Große Test-Inputs verlangsamen Testing extrem (Beispiel: Compiler, Bildbearbeitung-Programm)
- Wahrscheinlichkeit, dass große Test-Inputs notwendig sind, sehr gering

$\Rightarrow$ Reduziere Test-Inputs auf notwendigen Inhalt

- AFL minimiert nicht, sondern setzt Fenster für Lösch-Kandidaten abhängig von Ausführungsdauer
- Lange Testausführung $\rightarrow$ großes Fenster, um weniger Versuche zu verbrauchen
- Vorgehen:
  1. Lösche Daten in aktuellem Fenster
  2. Führe neuen Test aus
  3. Selbe Coverage $\Rightarrow$ ersetze alten Test durch aktuellen
  4. Andere Coverage $\Rightarrow$ mache Löschen rückgängig

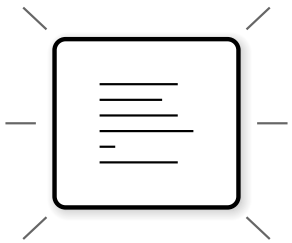
AFL: Beispiel djpeg

- Beispiel, dafür, wie viel Information der Kontroll-Fluss beinhalten kann
- Program under test: `djpeg` (dekomprimiert jpeg-Bilder)
- Einziger test seed: hello
- Nach 6 Stunden:



Whitebox Testing

Whitebox Testing



- Benutzt volle Programm-Semantik
- Kann alle Testziele finden
- Kann beweisen, dass Testziele nicht erreichbar sind
- Bei unerwarteten Aufgaben langsam

Vergleich

Blackbox

- Extrem schnell
- Oberflächliche Testabdeckung
- Viele Redundante Tests

Graybox

- Schnell für viele Testziele
- Tiefgehende Testabdeckung
- Redundante Tests

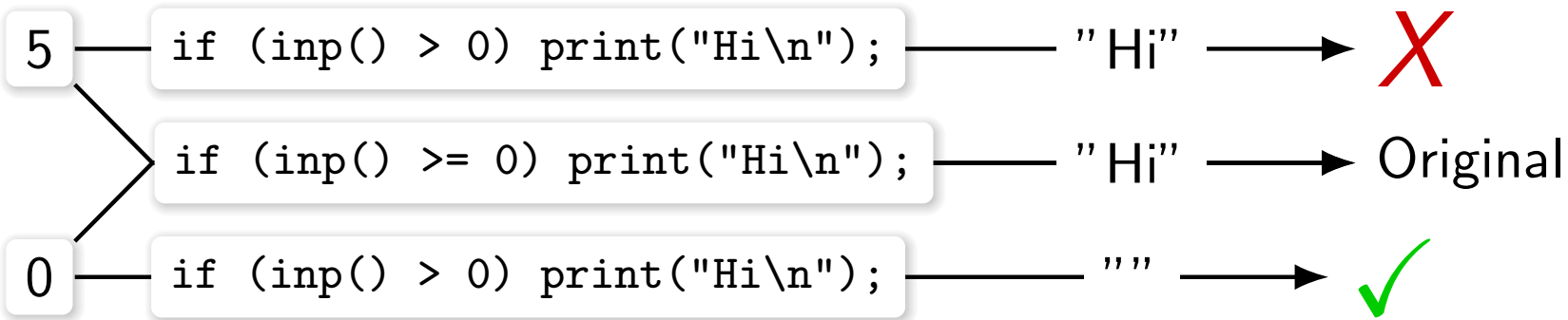
Whitebox

- Schnell für wenig Testziele
- Langsam für viele **oder** wenige Testziele
- Vollständige Testabdeckung
- Keine redundanten Tests

Mutations-Testen

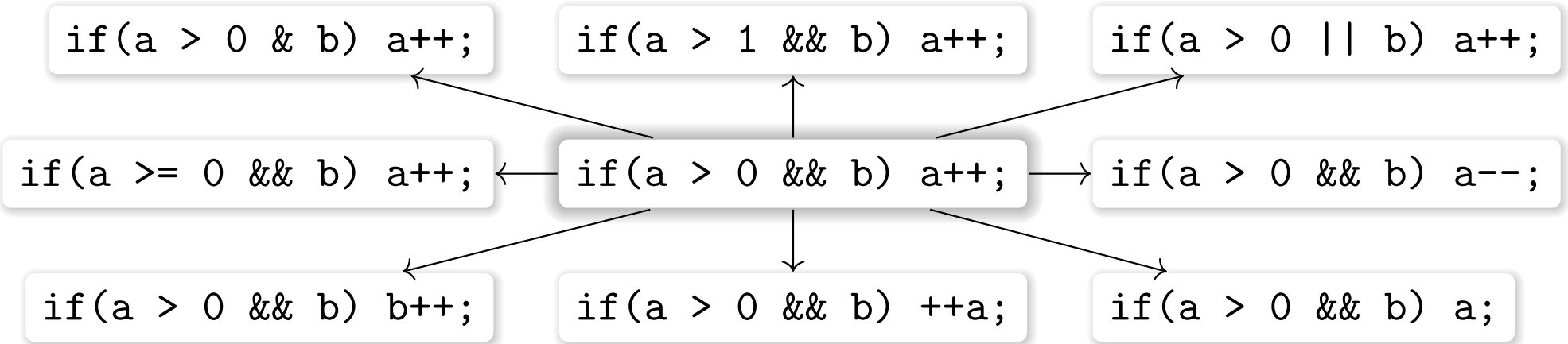
Mutations-Testen

- Kein Test-Generierungs-Verfahren, sondern Test Goal
- Test Goal: Decke möglichst viele Mutanten ab
- Mutant: Verändertes Original-Programm
- Mutant von Test abgedeckt: Test erkennt, dass Mutant Programmverhalten verändert
- Ein Test kann beliebig viele Mutanten abdecken



Programm-Mutationen

- Typische Mutations-Operatoren: Zahlen ersetzen, Operatoren durch andere Operatoren des gleichen Typ ersetzen



- Extrem viele Mutanten möglich
- Erstellung schnell möglich, aber Validierung skaliert nicht
- Offene Fragen:
  - Wieviele Programm-Stellen verändern?
  - Wieviele/Welche Mutanten notwendig?

Mutations-Testen: Annahmen

- Competent Programmer Hypothesis: Programmierer schreiben Programme, die nahe an korrektem Programm sind
- Coupling Effect: Tests, die viele kleine Fehler erkennen, können auch komplexere Fehler erkennen

$\Rightarrow$ First-Order Mutanten reichen (nur eine Änderung pro Mutant)

Kriterien für Mutanten-Abdeckung

- Gegeben Mutant P', ein Test muss folgende Bedingungen erfüllen, um P' abzudecken:

R Reachability: Mutierte Stelle muss erreicht werden

I Infection: Mutierte Stelle muss Programmzustand verursachen, der unterschiedlich von Zustand im Original-Programm ist

P Propagation: Fehlerhafter Programmzustand muss zu beobachtbarem Programmverhalten (z. B. Output) propagiert werden

- Existiert kein Test, der die Kriterien erfüllt, ist der Mutant *äquivalent*.
  - Erkennen von äquivalenten Mutanten unentscheidbar
  - Verschlechtern Test-Maß

Tutorial