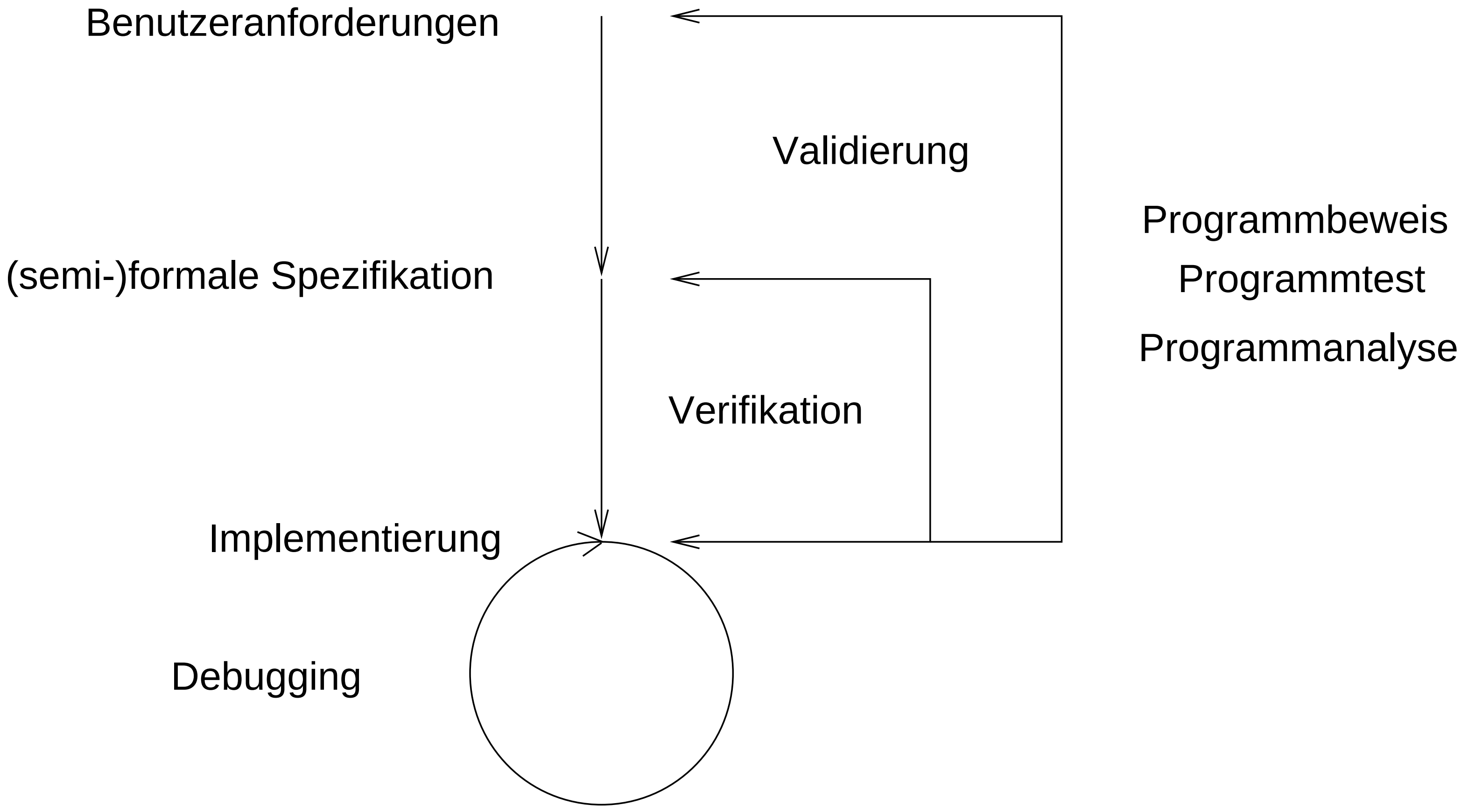
7 Qualitätssicherung

7.1 Grundlagen

7.1.1 Begriffsbildung

- Validierung: Sichern der Anforderungstreue: Are we building the right product? Stimmen Wunsch (Benutzererwartungen) und Wirklichkeit (Implementierung) überein? schliesst Verifikation ein
- Verifikation: Sichern der Spezifikationstreue: Are we building the product right? im engeren Sinn: formaler Programmbeweis
- Testen (dynamische Verfahren): Fehlerexistenz wird aus beobachtbarem Verhalten des laufenden Programms abgeleitet; z.B. Überdeckungstests

  "Program testing can be used to show the presence of bugs, but never to show their absence." (Dijkstra)
- Analyse (statische Verfahren): Fehlerexistenz wird aus Programmtext abgeleitet; z.B. Code-Inspektion, symbolische Ausführung
- Debugging: Fehlerlokalisierung, Fehlerbeseitigung


Benutzeranforderungen
Validierung
Programmbeweis
Programmtest
Programmanalyse
(semi-)formale Spezifikation
Verifikation
Implementierung
Debugging

7.1.2 Validierungsziele

- Qualität der Software sichern (Korrektheit, Effizienz, Robustheit, Bedienbarkeit, ...)
- frühe Fehlerentdeckung
- Wiederholbarkeit: Fehlverhalten muß reproduzierbar sein
- Automatisierbarkeit: Regressionstests (Wiederholung bereits durchgeührter Tests mit aktueller Implementierungsversion) sollen über Nacht ablaufen

Probleme:

- Validierung der Validierung
- Validierung unscharfer oder subjektiver Qualitätskriterien
- hoher Aufwand
- vollständiges Testen (exhaustive testing) i.a. unmöglich
- je später Fehler entdeckt werden, desto höher die Kosten bzw. der Aufwand zur Fehlerbeseitigung

7.1.3 Fehlerbegriff

deutsches Wort Fehler spiegelt verschiedene Begriffsbedeutungen nicht wider; im Englischen wird daher unterschieden zwischen:

- error: Eigenschaft des Quelltextes
- fault: falscher Zwischenzustand bei der Berechnung
- failure: von außen beobachtbares Symptom eines Fehlers

zwischen error, fault und failure gelten folgende Beziehungen:

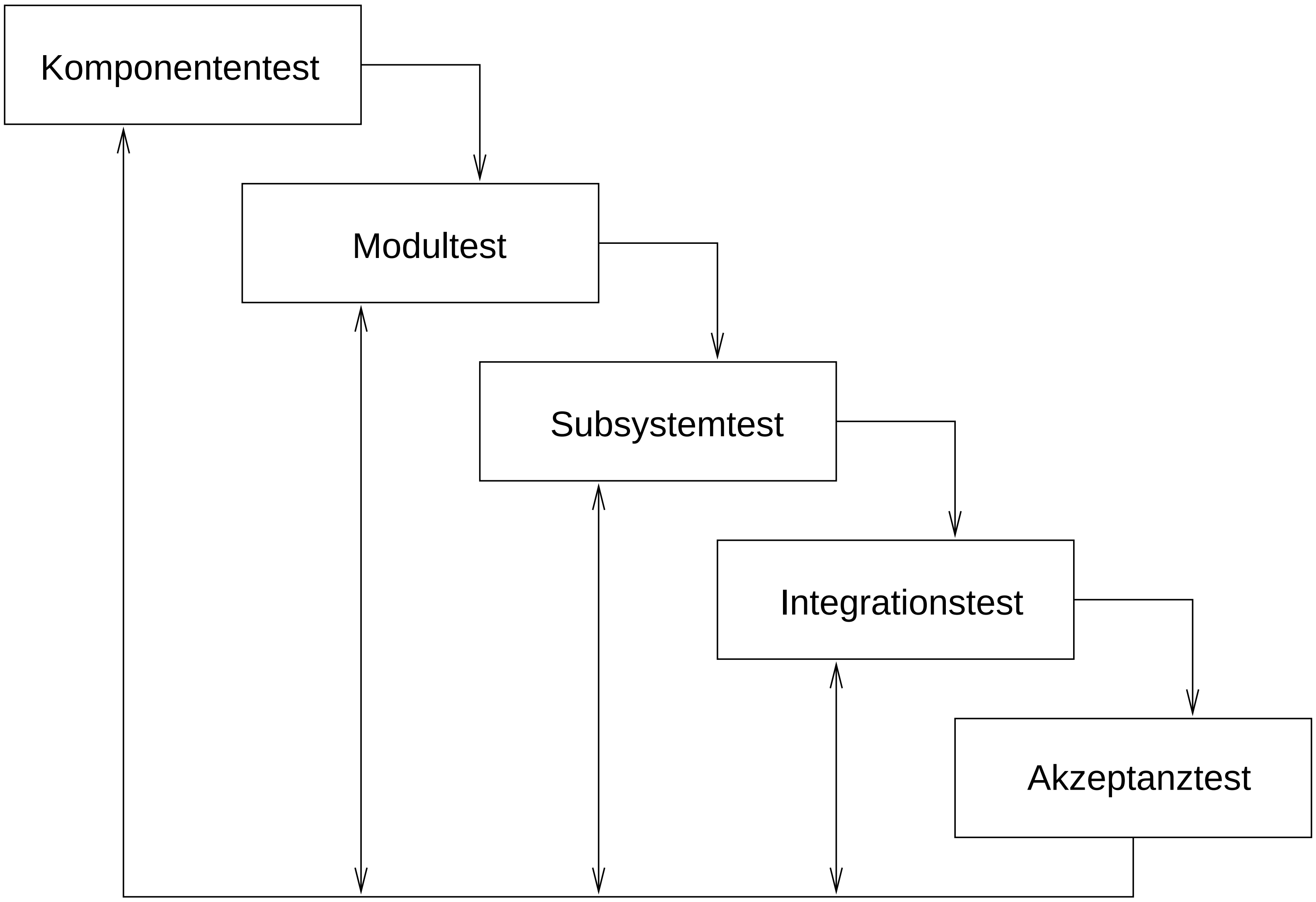
- das Auftreten eines Fehlersymptoms (failure) impliziert einen falschen Berechnungszustand (fault)
- falscher Berechnungszustand (fault) bedeutet, dass das Programm Fehler (error) enthält
- failure $\Rightarrow$ fault $\Rightarrow$ error
- Umkehrungen gelten nicht immer, d.h. es gibt Errors, die sich nicht (offensichtlich) als Failures manifestieren: Kernproblem des Testens

Beispiel

```
PROCEDURE max3(x,y,z:Integer):Integer
  IF x > y
  THEN max3 := x
  ELSE max3 := max(y,z)
  ENDIF
END max3;
```

- failure: max3(5,2,9) liefert 5
- fault: Variable max3 hat den Wert 5 bei Aufruf von max3(5,2,9)
- error: statt max3:=x muß es max3:=max(x,z) heißen

7.1.4 Stufen des Testprozesses



- Komponententest (z.B. Operationen) und Modultest (z.B. Klassen)

  Test jeder Funktion (jedes Moduls) gegen die Spezifikation; erfolgt für jede Funktion (jedes Modul) getrennt (stand-alone testing); daher Testrahmen notwendig, der lauffähige Umgebung schafft;
  stubs (Stümpfe) simulieren modulexterne Komponenten (z.B. importierte Prozeduren);
  drivers (Testtreiber) übernehmen die Versorgung von außen (z.B. Initialisierung, Aufruf mit gültigen Parametern)

- Subsystemtest (z.B. Packages)

  insbesondere Überprüfung der Schnittstellen

- Integrationstest (z.B. Kooperation der Packages)

  Test des kompletten Systems; umfaßt auch Überprüfung nicht-funktionaler Eigenschaften (z.B. Performance, Bedienbarkeit) und stress testing (Überlastbetrieb, Fehlbedienung, u.ä.)

- Akzeptanztest (Einsatztest mit Benutzern)

  Produktfreigabe nur nach bestandenem Akzeptanztest; Alpha-Test: System wird von Entwickler und Auftraggeber unter Realbedingungen (im laufenden Betrieb) getestet; Beta-Test durch ausgewählte Kunden, die Fehler zurückmelden

Organisatorische und psychologische Faustregeln

- Validierung muß so früh wie möglich erfolgen
- Definition eines Test-Abbruchkriteriums bzw. Testgütemaßes
- Testfälle zielgerichtet entwerfen und dokumentieren (Testplan)
- Testen ist destruktiver Prozeß, Ziel ist Fehlernachweis
- Implementierer nicht alleiniger Tester

7.1.5 Teststrategien

top-down testing

- Test beginnt mit den Steuerungsmoduln
- Designfehler werden früh entdeckt
- ausgeteste Teilsysteme früh verfügbar, insbesondere bei Zweigintegration (depth first integration)
- hoher Aufwand für stub-Implementierung
- aufwendige Testfalldefinition

bottom-up testing

- Test beginnt mit Moduln die keine anderen Moduln verwenden
- Designfehler werden spät entdeckt
- daher Kombination mit prototyping sinnvoll
- geringer Aufwand für Treiberimplementierung
- einfache Testfalldefinition

bottom-up und top-down testing sind komplementäre Verfahren und werden häufig kombiniert (sandwich testing)

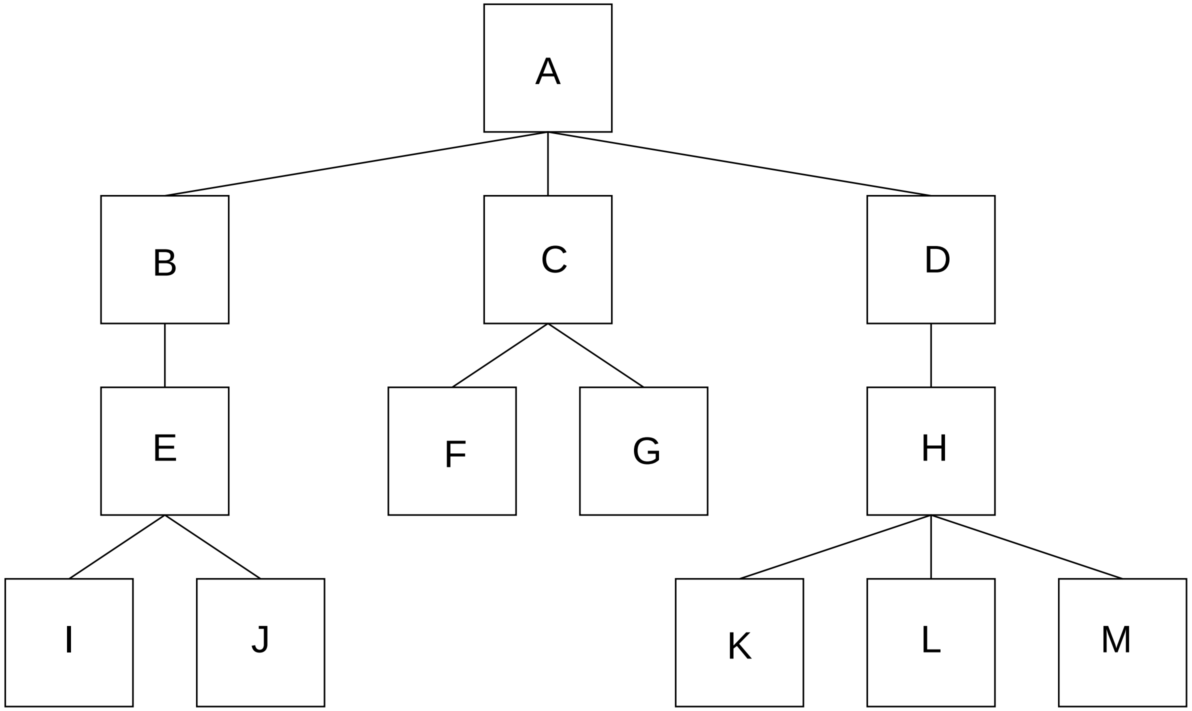
inkrementelles Testen

- Integration vieler Module in einem Schritt (Urknall-Verfahren) erschwert Fehlerlokalisierung
- Module werden daher einzeln in das System integriert
- nach jedem Schritt Regressionstest

Regressionstest

- Systemänderungen und -erweiterungen können neue Fehler einführen
- Wiederholung alter Tests, Vergleich mit den ursprünglichen Ergebnissen

Beispiel



bottom-up-Integration: (I, J, E), (F, G, C), (K, L, M, H), B, D, A

top-down-Integration:

depth-first: A, (B, E, I, J), (C, F, G), (D, H, K, L, M)

breadth-first: A, B, C, D, E, F, G, H, I, J, K, L, M

7.2 Testverfahren

7.2.1 Fehlerklassifikation

- Berechnungsfehler: Komponente berechnet falsche Funktion z.B. Verwendung falscher Variablen, Konstanten oder Operatoren; häufig: Konvertierungsfehler

  Die Raumsonde Mariner IV flog am Mars vorbei, weil man vergessen hatte, eine Real-Variable zu deklarieren, deren Name mit I anfing; dadurch nahm der Fortran-Compiler automatisch Typ Integer für die Variable an; in den arithmetischen Ausdrücken, in denen sie vorkam, wurden automatisch Konvertierungsoperationen eingefügt, und es gab erhebliche Rundungsfehler

- Schnittstellenfehler: syntaktische oder semantische Inkonsistenz zwischen Aufruf und Definition einer Komponente z.B. Übergabe falscher Parameter; häufig: Vorbedingung einer Funktion wird nicht eingehalten

- Kontrollflußfehler: Ausführung eines falschen Programmpfades z.B. Vertauschung von Anweisungen, falsche Kontrollbedingungen; häufig: one-too-many oder one-too-less: Schleife wird einmal zu oft oder zu wenig durchlaufen

- Initialisierungsfehler: falsche oder fehlende Initialisierung; häufig: Zugriff auf nicht initialisierte Variablen

- Datenflußfehler: falsche Zugriffe auf Variablen und Datenstrukturen z.B. falsche Arrayindizierung, Zuweisung an die falsche Variable; häufig: Pointerfehler, z.B. Zugriff mit NIL/NULL-Pointer und/oder Zugriff auf bereits freigegebene Objekte

Beispiel: fehlerhaftes Programm; Korrektur jeweils in [[[...]]]

Klassifikation: 1. Initialisierungsfehler, 2. Kontrollflußfehler, 3. Berechnungsfehler, 4. Datenflußfehler, 5. Schnittstellenfehler

```
PROCEDURE countVowels(s: sentence; VAR count:Integer)
// sentence = array [1..max] of char
// Counts vowels in a sentence; sentence terminated by a dot "."
VAR i:integer;
BEGIN
count:=0; [[[count:=0; i:=1;]]]                              // 1.
WHILE s[i] <> dot DO
  IF s[i]="a" OR s[i]="e" OR s[i]="i" OR
     s[i]="o" [[[OR s[i]="u"]]] THEN                         // 2.
    count:=count+2; [[[count:=count+1;]]]                    // 3.
  END;
  count:=i+1; [[[i:=i+1;]]]                                  // 4.
END;
END countVowels;
Aufruf: countVowels("this is a test.",count)
Aufruf: countVowels("to be ... or not to be ...",count) // 5.
```

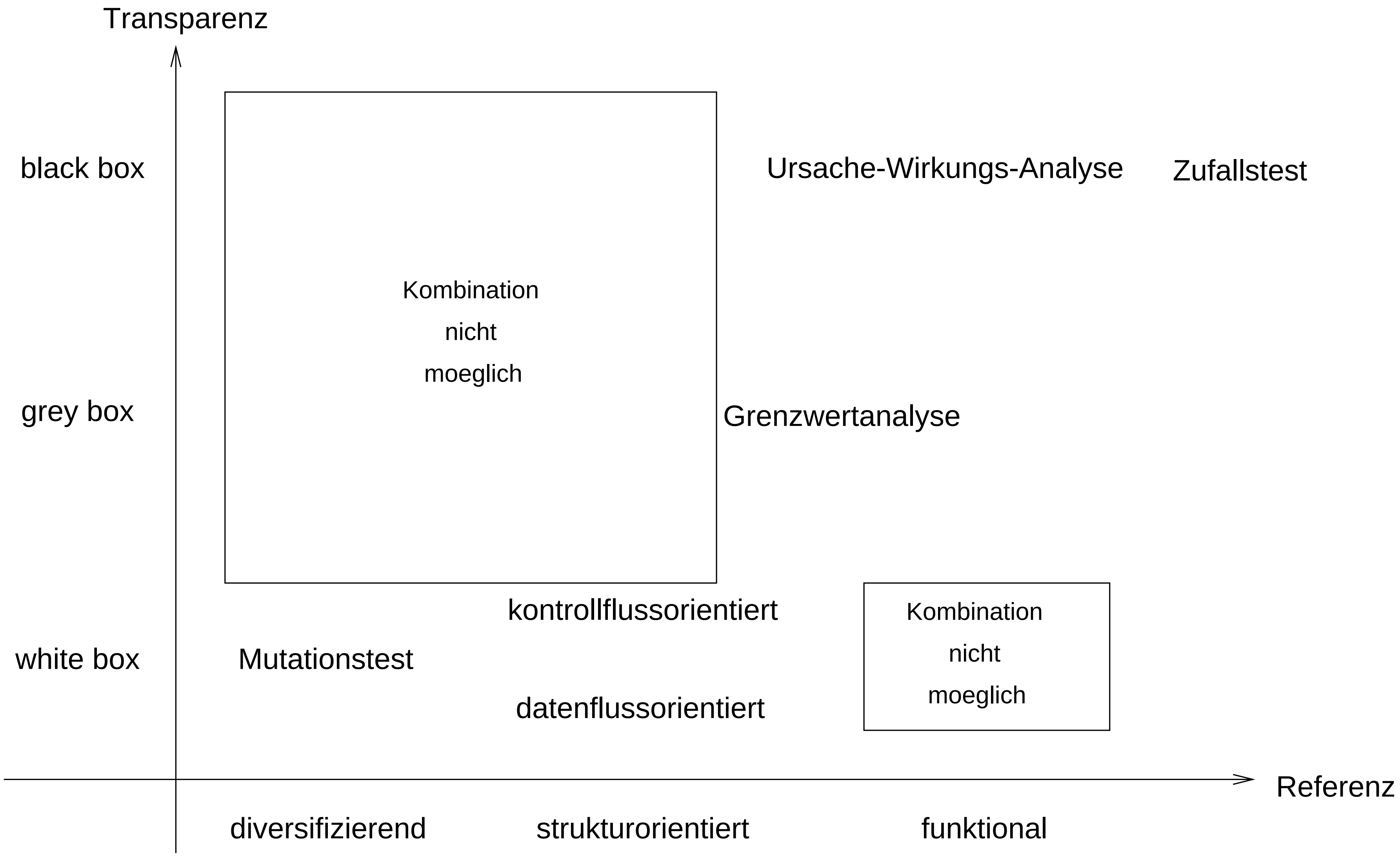
7.2.2 Klassifikation von Testverfahren

(1) Klassifikation nach Strukturinformation (Transparenz des Prüflings):

- black box: interne Modulstruktur unnötig
- white box: interne Modulstruktur nötig weil Testgrundlage
- grey box: Zwischenkategorie

(2) Klassifikation nach Prüfreferenz:

- diversifizierend: Prüfung gegen andere Programmversion(en)
- strukturorientiert: Prüfung gegen Implementierung
- funktional: Prüfung gegen Spezifikation


Transparenz
black box
Ursache-Wirkungs-Analyse
Zufallstest
Kombination
nicht
moeglich
grey box
Grenzwertanalyse
kontrollflussorientiert
Kombination
nicht
moeglich
white box
Mutationstest
datenflussorientiert
Referenz
diversifizierend
strukturorientiert
funktional

7.2.3 Kontrollflußorientierte Testverfahren

- Begriff: Kontrollflußorientierte Testverfahren betrachten bestimmte Strukturelemente eines Programmes (Anweisungen, Zweige, Bedingungen, Pfade) und fordern, daß der Kontrollfluß jedes dieser Elemente mindestens einmal erreicht (Überdeckungstests)
- Überdeckung kann gemessen und als Testgütemaß verwendet werden
- Programmdarstellung als Kontrollflußgraph:

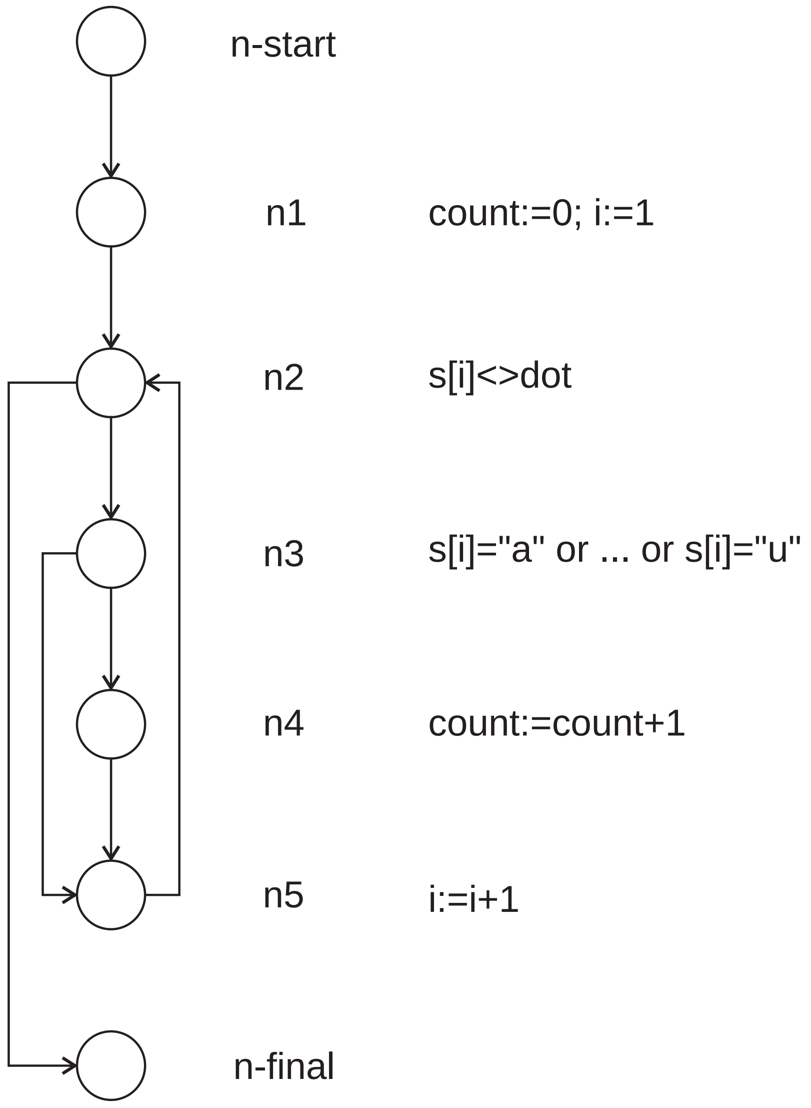
  Knoten: ausführbare Anweisung (Zuweisung, Aufruf) oder Bedingung einer Kontrollstruktur

  davon zwei ausgezeichnet: n-start und n-final

  Kante: möglicher Programmfluß zwischen zwei Anweisungen

  Pfad: Kombination von Kanten, beginnt in n-start, endet in n-final

Kontrollflußgraph des Beispielprogramms



Überdeckungstests

Anweisungsüberdeckung (C_0-Test)

- jeder Knoten des Kontrollflußgraphen muß einmal ausgeführt werden
- Minimalkriterium
- nicht sehr leistungsfähig
- Problem: unentdeckte Fehler in nicht ausgeführten Zweigen

Zweigüberdeckung (C_1-Test)

- jede Kante des Kontrollflußgraphen muß einmal durchlaufen werden
- realistisches Minimalkriterium
- schließt Anweisungsüberdeckung ein
- Problem: unentdeckte Fehler bei Kombination bzw. Wiederholung von Zweigen z.B. Schleifen
- Problem: nicht-ausführbare Zweige

Realisierung

- Aufbau des Kontrollflußgraphen
- Eingabe der Testdaten
- Markieren berührter Knoten und Kanten während der Ausführung
- Berechnung des Überdeckungsmaßes
  #(berührte Knoten) / #(Gesamtknoten)
- Wiederholung mit anderen Testdaten bis Gesamtüberdeckung ausreichend

Bedingungsüberdeckung

- jede (ausgewählte Teil-)Bedingung muß mindestens einmal den Wert false und true annehmen
- atomare Bedingungsüberdeckung: jede einfache Bedingung muß einmal den Wert true bzw. false annehmen

  `s[i]<>dot`: 1. Test mit `s[i]<>dot`; 2. Test mit `s[i]=dot`
- minimale Mehrfachbedingungsüberdeckung: In allen zusammengesetzten Bedingungen müssen alle möglichen true/false-Kombinationen der einfachen Bedingungen vorkommen; realistischer Kompromiß; orientiert sich an der syntaktischen Struktur von Bedingungen; jeder Knoten muß überdeckt werden; umfaßt Zweigüberdeckung

  `s[i]="a" OR ... OR s[i]="u"`: 1. Test mit `s[i]="a"`; 2. Test mit `s[i]="e"`; 3. Test mit `s[i]="i"`; 4. Test mit `s[i]="o"`; 5. Test mit `s[i]="u"`; 6. Test mit `s[i] not-in {a,e,i,o,u}`

Pfadüberdeckung

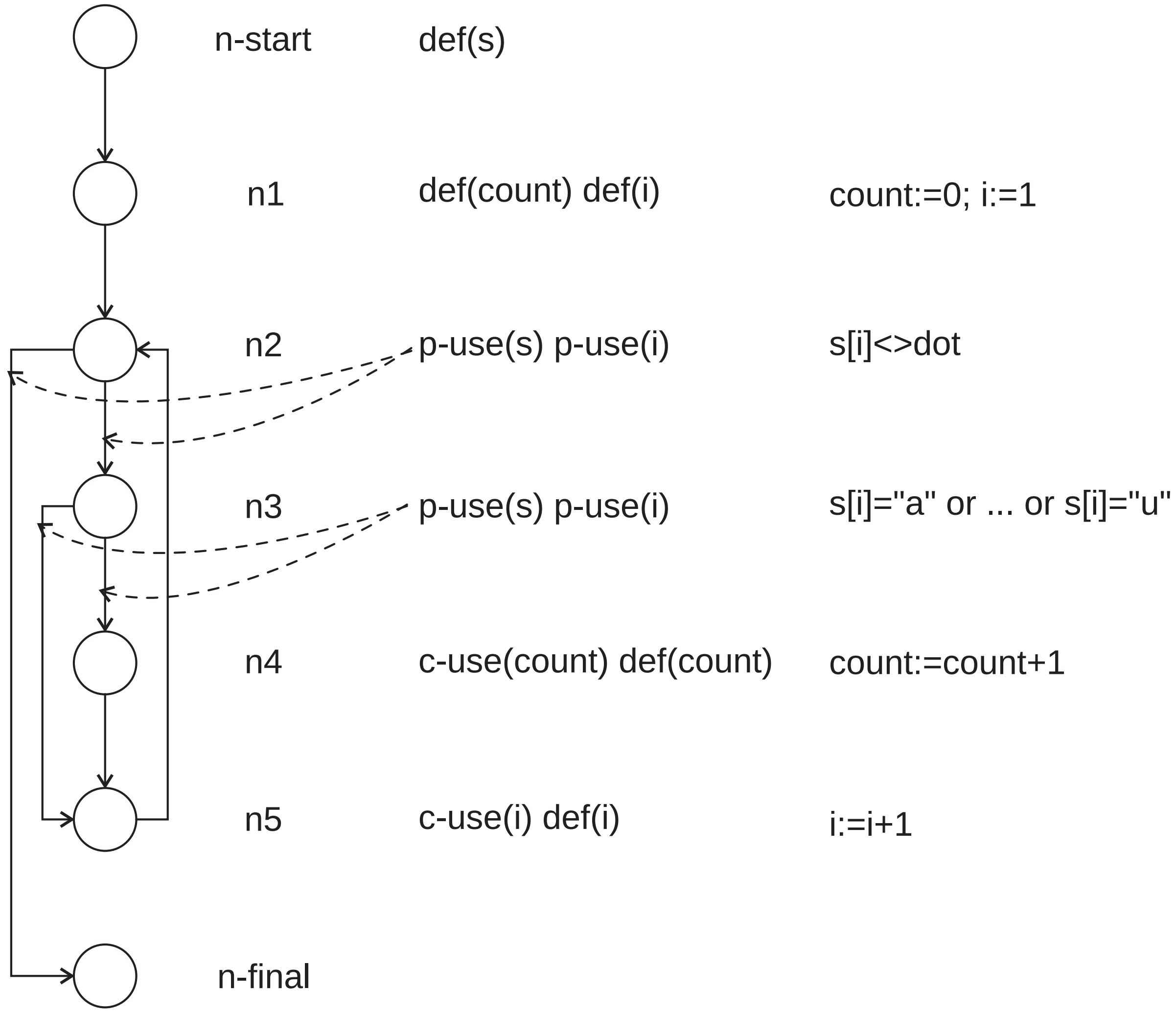
- jeder Pfad des Kontrollflußgraphen muß einmal durchlaufen werden
- i.a. unendlich viele Pfade (z.B. durch Schleifen) daher nur theoretisches Kriterium
- Vergleichsmaßstab für andere Überdeckungstests
- findet nicht alle Fehler (z.B. Berechnungsfehler)
- verschiedene abgeleitete praktikable Verfahren existieren

7.2.4 Datenflußorientierte Testverfahren

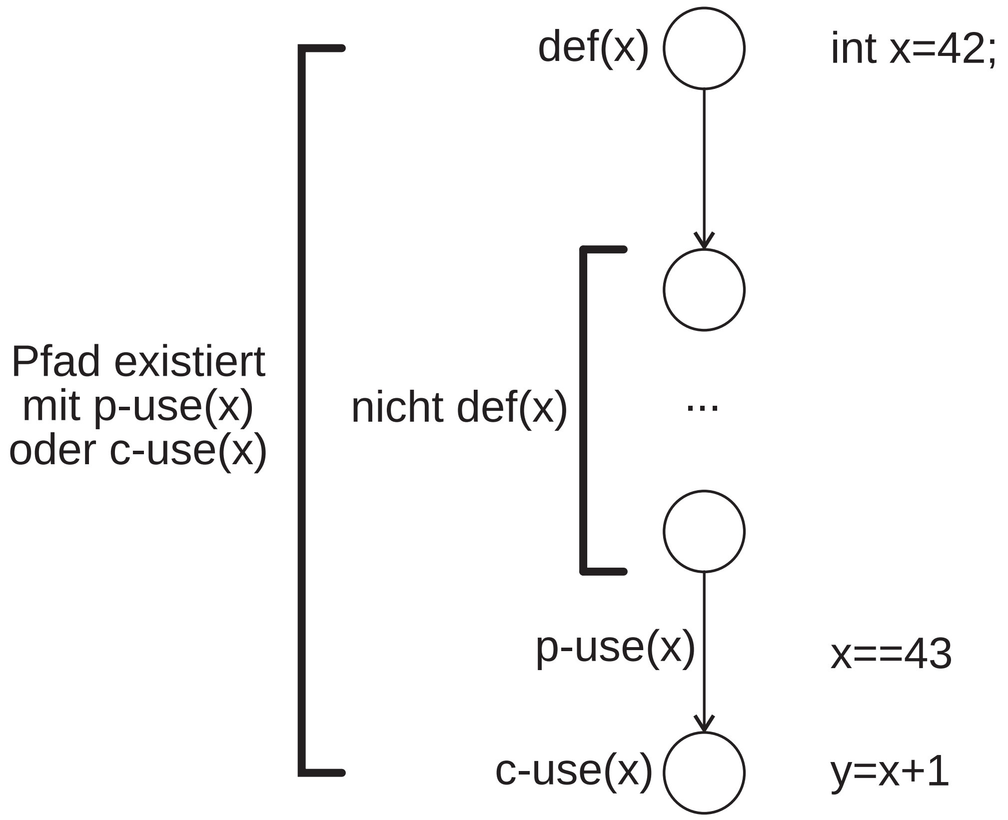
- Begriff: Datenflußorientierte Testverfahren teilen Variablenzugriffe in verschiedene Klassen ein und fordern, daß für jede Variable ein Programmpfad durchlaufen wird, für den bestimmte Zugriffskriterien zutreffen (defs/uses-Kriterien)
- Zugriffe werden unterschieden in (1) Zuweisung (definition, def), (2) berechnende Benutzung (computational use, c-use), (3) prädikative Benutzung (predicative use, p-use)
- Programmdarstellung als Datenflußgraph:

  - erweiterter Kontrollflußgraph
  - Knoten attributiert mit def und c-use
  - Kanten attributiert mit p-use
  - Extra-Knoten für Beginn und Ende von Sichtbarkeitsbereichen der Variablen
- führt zu defs/uses-Kriterien

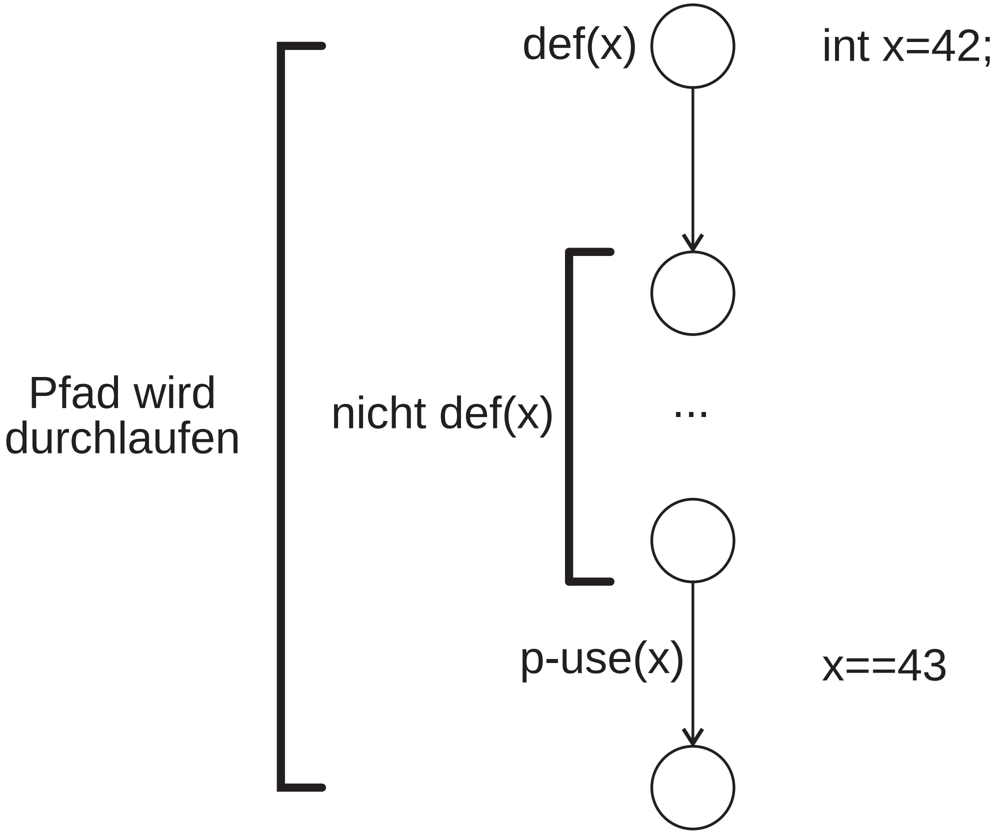
Datenflußgraph des Beispielprogramms



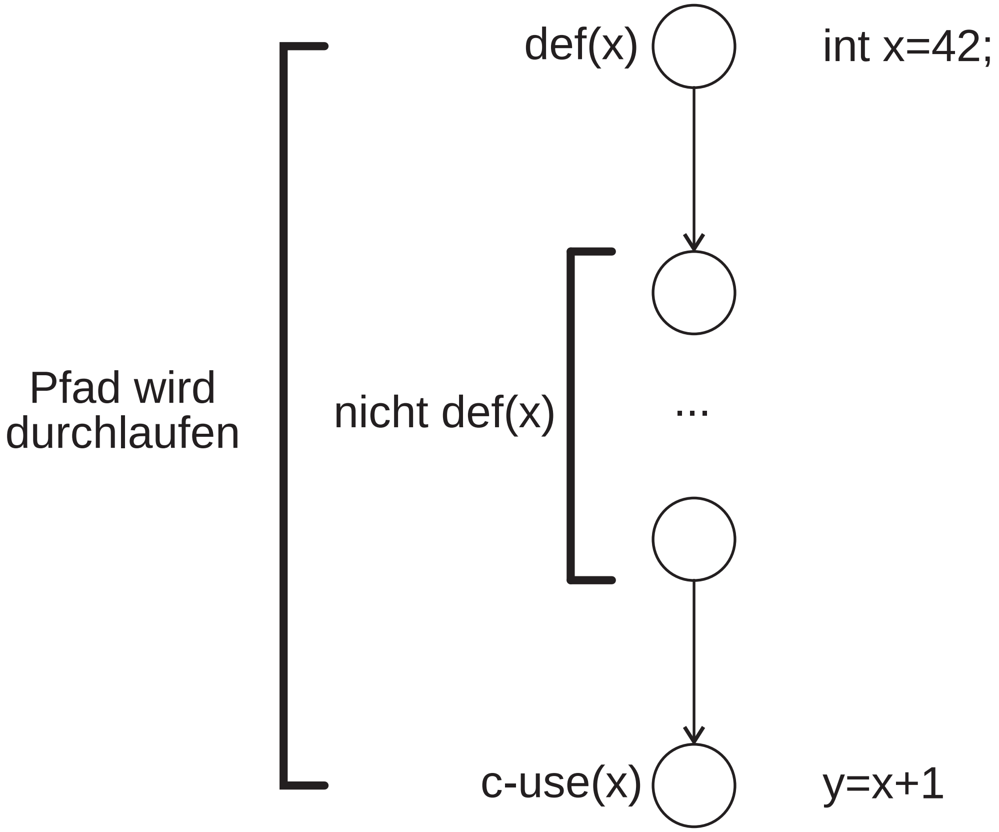
all defs-Kriterium: statische Überprüfung (anhand des Programmtextes), ob jede Definition einer Variablen in einer Berechnung oder Bedingung verwendet wird



all p-uses-Kriterium: Tests entwicklen für alle Kombinationen von Definitions- und prädikativen Benutzungsstellen einer Variablen



all c-uses-Kriterium: Test entwicklen für alle Kombinationen von Definitions- und berechnenden Benutzungstellen einer Variablen



verschiedene Kombinationen dieser Basisverfahren möglich

7.2.5 Funktionale Testverfahren

- Begriff: Funktionale Testverfahren testen gegen die Spezifikation und lassen die interne Programmstruktur unberücksichtigt; sie verlangen, daß die spezifizierte Funktionalität vollständig und natürlich spezifikationstreu implementiert ist (Funktionsüberdeckung)
- komplementär zu white-box-Verfahren
- benötigen vollständige und widerspruchsfreie Spezifikation
- Problem: Testvollständigkeit schwierig zu messen
- einfachste Ausprägung: Zufallstest; beim Zufallstest wird das Programm mit zufälligen Werten gefüttert, und es wird geprüft, ob das richtige Ergebnis erzielt wird
- Funktionale Verfahren benötigen ein Orakel: Orakel prüft, ob ein vom Prüfling errechnetes Ergebnis korrekt im Hinblick auf die Spezifikation ist; da dies i.a. unentscheidbar ist, spielt meist der Mensch die Rolle des Orakels

Funktionale Äquivalenzklassenbildung

- Wertebereiche von Ein- und Ausgaben werden in Äquivalenzklassen eingeteilt
- Bildung der Äquivalenzklassen nur an der Spezifikation orientiert
- Äquivalenzklassen für gültige und ungültige Werte
- getestet wird nur noch für jeweils einen Repräsentanten der Klasse
- Wahl des Repräsentanten z.B. zufällig oder an den Rändern der Klassen (Grenzwertanalyse)
- Testfälle enthalten möglichst viele gültige Repräsentanten aber höchstens einen ungültigen
- Regeln zur Äquivalenzklassenbildung: jeder spezifizierte Eingabebereich induziert mindestens eine gültige und mindestens eine ungültige Klasse; jede Eingabebedingung induziert eine gültige (Bedingung erfüllt) und eine ungültige (Bedingung nicht erfüllt) Klasse; analog für Ausgaben

Beispiel: im Beispielprogramm kommen die Eingabebedingungen s[i]<>dot, s[i]='a', ... vor; entsprechend werden die Eingabesätze unterteilt in leerer Satz, Sätze ohne Punkt, Sätze mit einem Punkt, Sätze mit mehreren Punkten, Sätze ohne 'a', Sätze mit einem 'a', Sätze mit mehreren 'a' usw; zu jeder dieser Klassen wird (mindestens) ein Testfall gewählt

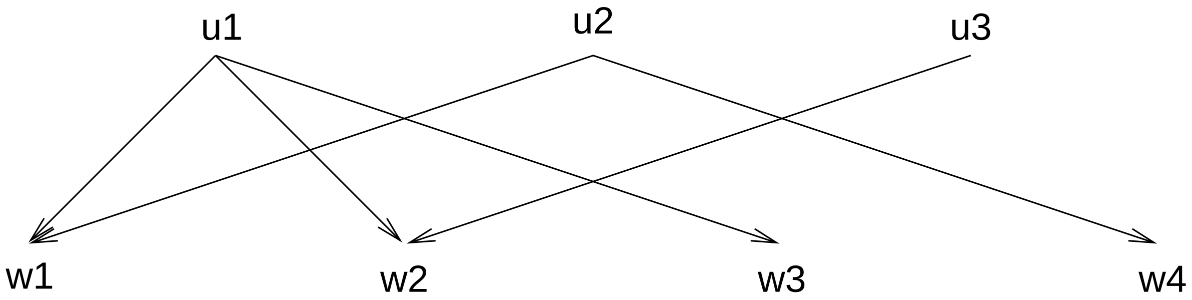
7.2.6 Ursache-Wirkungs-Analyse

- berücksichtigt Wechselwirkungen und Abhängigkeiten zwischen Eingaben/Ursachen und Ausgaben/Wirkungen
- Umsetzung der Spezifikation in einen Graphen, der Ursachen und Wirkungen verknüpft
- Darstellung z.B. in graphischer Form oder als Entscheidungstabelle
- Konstruktion von Testfällen entspricht der Erfüllung der Prämissen
- Ursache-Wirkungs-Verhältnisse werden durch Boolesche Basisfunktionen Identität, Negation, Konjunktion und Disjunktion dargestellt
- Abhängigkeiten zwischen Ursachen beschreiben Einschränkungen: gegenseitiger Ausschluß; m-aus-n-Auswahl; 1-aus-n-Auswahl; notwendige Voraussetzung

Beispiel: Annahme, formale Spezifikation liegt vor

in countVowels kann man drei Ursachen u1: Vokal, u2: dot, u3: beliebiges anders Zeichen und vier Wirkungen identifizieren: w1: nächstes Zeichen untersuchen, w2: i erhöhen, w3: Vokalanzahl erhöhen, w4: Ende

resultierendes Graph



zu jeder Kante bzw. zu jeder Menge durch "und" verknüpfter Kanten muß mindestens ein Testfall konstruiert werden

7.2.7 Statistische Testverfahren

Zuverlässigkeitsmaße

- Fehlerwahrscheinlichkeit pro Anforderung: Wahrscheinlichkeit, daß ein Systemaufruf sich irregulär verhält
- Fehlerauftrittsrate: Zahl der Fehler pro (Zeit-)Einheiten; hierbei kann Zeit reale Zeit, Systemzeit, Transaktionschritte, Interaktionsschritte, etc. bedeuten
- Durchschnittliche Zeit zwischen Fehlern: aus Hardware entlehnt (durchschnittliche Bausteinlebensdauer)
- Verfügbarkeit: Zeitanteil, in dem das System benutzt werden kann (z.B. Telefonvermitlung)
- bei kritischer Software sind Vorgaben für diese Parameter Teil der Spezifikation

statistisches Testen

- Vorgehen: statistische Verteilung der Eingabedaten im Eingaberaum ermitteln
- Eingabeprofil: Einteilung der Eingaben in möglichst kleine Äquivalenzklassen (s.o.), Angabe der Wahrscheinlichkeit für jede Klasse entweder Schätzen oder aus Monitoring des realen Betriebs
- möglichst viele Testdatensätze konstruieren, die dieser Verteilung entsprechen
- Programm mit diesen Testdaten laufen lassen und obige Parameter nebst ihren Varianzen messen
- nach hinreichend vielen Tests gewinnt man halbwegs zuverlässige Mittelwerte und Standardabweichungen für die Fehlerquote

Probleme:

- Anforderungsprofil schwer zu messen bzw. schätzen
- für hohe Zuverlässigkeitswerte mit kleiner Varianz sind extrem viele Tests erforderlich
- $\Rightarrow$ teuer, es sei denn, Testdaten können automatisch erzeugt werden mittels Testdatengeneratoren

7.2.8 Diversifizierende Testverfahren

- Begriff: Diversifizierende Testverfahren betrachten verschiedene Programmversionen bzw. -implementierungen und erwarten, daß beobachtbare Abweichungen zwischen Original und Varianten erzeugt werden
- Varianten entstehen (1) aus der Spezifikation durch Mehrfachimplementierung (back-to-back testing), (2) aus dem Programm durch systematische Anwendung von Transformationsregeln (Mutationstesten)
- werden keine Verhaltensabweichungen aufgedeckt, so weiß man, daß (1) die Zahl der Testfälle nicht ausreichend war $\Rightarrow$ Validierung der Testdaten (2) oder die Varianten sich tatsächlich im Verhalten nicht unterscheiden

N-Versionen-Programmierung

- Programm wird nach gegebener Spezifikation von N getrennten Teams implementiert und getestet
- Teams dürfen nicht in Kontakt stehen
- Hardware u.U. gleichfalls N-fach beschaffen
- Versionen laufen parallel; bei nicht identischen Resultaten findet Mehrheitsabstimmung statt
- Annahme: Programmfehler in den Versionen sind statistisch unabhängig
- Probleme: (1) N-fache Kosten; (2) grundlegende Annahme u.U. nicht gewährleistet: (a) Fehler in der Spezifikation; (b) systematische Fehler: z.B. alle Implementierer haben zufällig denselben falschen Algorithmus gelernt

7.2.9 Mutationstesten

Grundideen

- Programme sind (fast richtig) (competent programmer hypothesis)
- kleine Änderungen am Programm sollten bei hinreichend umfassenden Testdatensätzen zu beobachtbaren Verhaltensänderungen führen
- die möglichen kleinen Änderungen (Mutationen) kann man systematisch erzeugen und die geänderten Programme (Mutanten) mit den Testdatensätzen füttern
- perfekter Testdatensatz sollte alle Mutanten erkennen
- Anteil aufgedeckter Mutanten kann als Testgütemaß verwendet werden: “Testdatensatz hat nur 60% aller Mutanten gefunden; man braucht also noch mehr Testfälle”

Fehlermodellierung durch Mutationsoperatoren

- beschreiben Erzeugung semantisch veränderter aber syntaktisch korrekter Programmversionen
- für jede Fehlerklasse spezielle Operatoren
- jeder Mutant enthält nur eine Abweichung

Vorteile

- Verfahren zur Beurteilung von Testdatensätzen
- weitgehend automatisierbar
- explizite Fehlerorientierung
- Modellierung anderer Verfahren möglich z.B. Anweisungsüberdeckung, special value testing

Nachteile

- es werden nur (einfache) Fehler erzeugt
- es wird angenommen, daß komplexe Fehler Kombinationen einfacher Fehler sind (nicht bewiesene Hypothese)

Mutationsoperatoren

- Berechnungsfehler: Ändern von arithmetischen Operatoren (+ statt -); Löschen von arithmetischen (Teil-)Ausdrücken; Ändern von Konstanten
- Schnittstellenfehler: Vertauschen bzw. Ändern von Parametern; Aufruf anderer Prozeduren eines Moduls
- Kontrollflußfehler: Ersetzen von logischen (Teil-)Ausdrücken durch true und false; Ändern von logischen und relationalen Operatoren (AND statt OR, $>=$ statt $<$); Aufruf anderer Prozeduren; Ändern von Sprungzielen; Löschen von Anweisungen; Einfügen von HALT-Anweisungen
- Initialisierungsfehler: Ändern von Konstanten; Löschen von Zuweisungen bzw. Initialisierungsanweisungen
- Datenflußfehler: Vertauschen von Variablen in festem Sichtbarkeitsbereich; Änderungen in der Indexberechnung

7.3 Analyseverfahren

7.3.1 Informelle Analysemethoden

- Begriff: Als informelle Analysemethoden werden Organisationsmodelle bezeichnet, die auf einer unter verschiedenen Kriterien durchgeführten manuellen Analyse durch eine Gruppe von Personen beruhen (human testing)
- Gruppen bestehen aus drei bis fünf Personen und umfassen z.B Moderator, Programmierer, Designer, Testexperte, erfahrener Programmierer, Programmieranfänger, Programmiersprachspezialisten
- Ziel ist Fehleraufdeckung
- Fehlerbeseitigung erfolgt anschließend durch den Programmierer
- Code-Inspection (1) Programmierer erklärt Programm Anweisung für Anweisung; (2) Programm wird auf typische (Implementierungs-)Fehler untersucht
- Code-Walkthrough (1) benötigt einige Testfälle; (2) Teilnehmer führen das Programm manuell aus (spielen Computer)
- Beurteilung: (erstaunlich) leistungsfähig, bis zu 80% Aufdeckungsqoute; billig; gruppendynamisch anspruchsvoll (Inspektion kann leicht zum Tribunal ausarten)

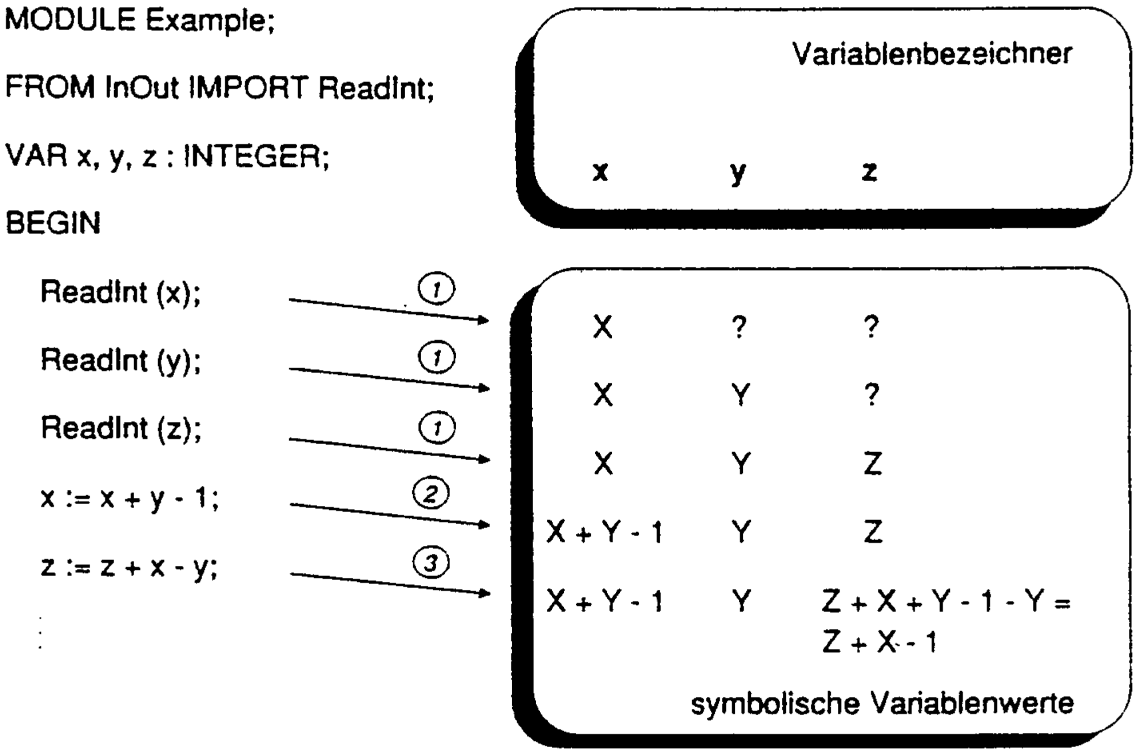
7.3.2 Symbolische Interpretation

- Begriff: Bei der symbolischen Interpretation wird das Programm mit symbolischen Ausdrücken statt konkreten Werten ausgeführt; Ergebnis der Interpretation ist ein Gleichungssystem, aus dem allgemeine Aussagen über die Korrektheit abgeleitet werden können
- Mischtechnik zwischen Test und formaler Verifikation
- keine formale Spezifikation nötig
- benötigt speziellen Interpreter

Vorgehensweise

- für jede Eingabevariable wird neue symbolische Variable eingeführt
- bei einer Zuweisung x := Expression wird der symbolische Wert von Expression ausgerechnet und an x gebunden; dabei werden algebraische Umformungen so weit wie möglich angewandt
- errechnete symbolische Werte gehen in die nächste Anweisung ein
- Ergebnis sind symbolische Werte für die Ausgabevariablen

Beispiel:



Legende:

1. Initialisierung durch den Interpreter
2. Einsetzen der symbolischen Werte von x und y in die rechte Seite der Zuweisung, Auswertung des symbolischen Ausdrucks und Zuweisung an x
3. Einsetzen der symbolischen Werte in die rechte Seite der Zuweisung, Bildung und Vereinfachung des symbolischen Ausdrucks. Zuweisung an z.

Pfadbedingungen

- lassen sich die symbolischen Werte von Kontrollbedingungen nicht vollständig zu konkreten Werten true, false reduzieren, müssen Pfadbedingungen eingeführt werden:

  nicht-deterministische bzw. interaktive Auswahl eines Zweiges

  Ergänzung der Pfadbedingung um den entsprechenden symbolischen Wert der Kontrollbedingung

  Fortsetzung der Interpretation entlang des gewählten Zweiges

  Pfadbedingung zu Beginn der Interpretation = Vorbedingung des Programms

- errechnete Pfadbedingungen können zur Testdatengenerierung verwendet werden; insbesondere zur Konstruktion von Testfallklassen

- entspricht der errechnete symbolische Ausdruck der Nachbedingung für den speziellen Pfad, so ist das Programm zumindest für diesen Pfad verifiziert

- formal: Programm P, Vorbedingung A, Nachbedingung B, also $\{ A \}$ P $\{ B \}$

  symbolische Auswertung liefert Pfadbedingung X, symbolische Ausgabe Y, also $\{ A \wedge X \}$ P $\{ Y \}$

  noch zu zeigen: $X \wedge Y \Rightarrow B$

7.4 Cleanroom Software Development

- Organisationsmodell zur Produktion hochwertiger Software; insbesondere hohe Zuverlässigkeit und Wartbarkeit
- drei getrennte Teams: Spezifikation, Implementierung, Test
- Prinzipien:

  inkrementelles Entwurfsmodell: Entwicklung ausführbarer Teilprodukte

  Einsatz formaler Spezifikations-, Entwicklungs- und Validierungsmethoden

  Implementierung ohne Programmausführung: Implementierungsgruppe darf lediglich statische Methoden (Inspektion, (Syntax-)Analyse, formale Verifikation) einsetzen, aber nicht Testen

  kein Modultest; Integrationstest nach statistischen Verfahren durch unabhängige Testgruppe
- Ergebnisse: (1) Mißverständnisse zwischen Entwicklern und Auftraggeber werden seltener (formale Spezifikation erzwingt gedankliche Durchdringung der Anforderungen); (2) inkrementelles Entwurfsmodell führt zu kontinuierlichen Projektfortschritten und besserer Projektsteuerung; (3) Korrektheit wird nicht "hineingetestet" sondern "hineinentwickelt"; (4) sehr zuverlässige Software bei kaum erhöhten Entwicklungskosten, in großen Projekten (IBM, NASA) erfolgreich eingesetzt