Technische Fakultät
Universität Bielefeld

Vorlesung

Softwaretest und -optimierung

Version 2012

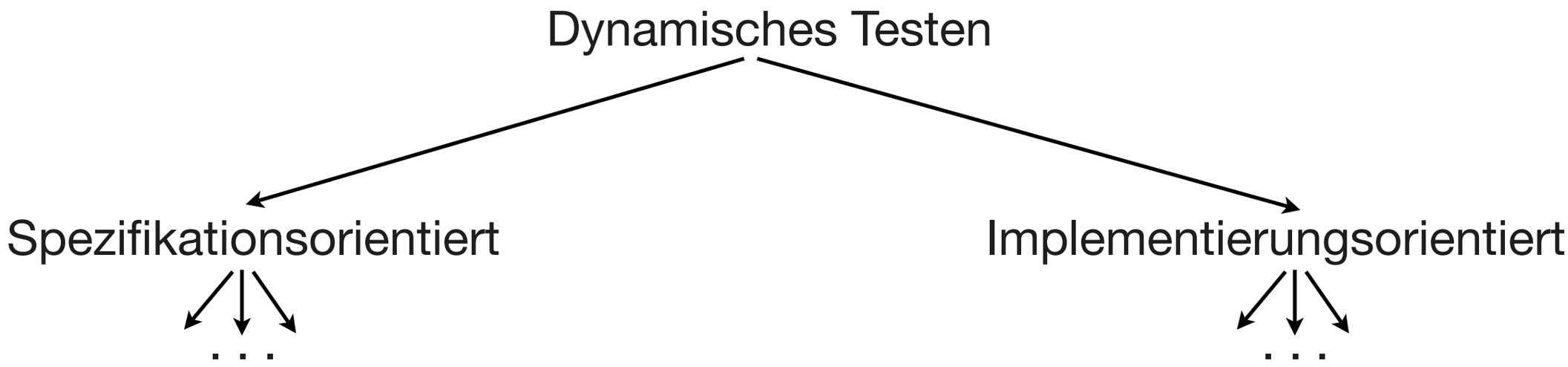
Dr. Carsten Gnörlich

Rechnerbetriebsgruppe

Kap. 6 - Statische Analyse

(= Kap. 12 aus Riedemann)

Einordnung



→ Testfälle generieren, Programm durch Ausführen testen

Statische Analyse

→ Testen des Programms *ohne* tatsächliche Ausführung

Übersicht:
Analyse der verschiedenen Entwurfsphasen

Ziele der statischen Analyse

1. Anforderungsspezifikation, Systemspezifikation
2. Entwurfsspezifikation (Grob- und Feinentwurf)
3. Quellkode des Programms ⟵ wird unser Schwerpunkt sein!
4. Objektkode des Programms

Anforderungs-/Systemspezifikation

Ziele der Untersuchung

- Notwendigkeit - für Systemziele
- Vollständigkeit - alle Eingaben,Ausgaben,Fälle berücksichtigt
  - Umgebung, Leistung, Benutzung spezifiziert?
- Konsistenz - Anforderungen untereinander
  - Nomenklatur, Maßeinheiten
- Durchführbarkeit - mit gegebener Technologie
- Testbarkeit - Entscheidbarkeit ob eine Anforderung erfüllt ist
  - Konstruktion von Testfällen

Anforderungs-/Systemspezifikation

Hilfsmittel zur Analyse

- Vergleich mit bestehenden Systemen / Standards
- mathematische Analyse algorithmischer / zeitlicher Anforderungen
- Simulationsmodelle / analytische Modelle
- automatische oder manuelle Dokumentationsanalyse

Entwurfsspezifikation

Ziele der Untersuchung

- Notwendigkeit - bzgl. Anforderungsspezifikation
- Vollständigkeit - bzgl. Anforderungsspezifikation
- Konsistenz - Schnittstellen, Ein-/Ausgabeformate, Datenbankschemata,...
- Korrektheit - Algorithmen, mathematische Gleichungen
  - Kontrolllogik
  - Leistung, Ressourcenverbrauch

Entwurfsspezifikation

Hilfsmittel zur Analyse

- Korrektheit: nur von Hand beweisbar
- Vollständigkeit / Notwendigkeit
  - im Groben automatisch per Numerierung:
    - Anforderungen eindeutig numerieren
    - bei Entwurfselementen zugehörige Anforderung angeben
- Konsistenz
  - bei Formalisierung von E/A-Variablen automatisch machbar
  - Problem: Synonyme, z.B. guint32 vs. unsigned int

Quellkode

Hilfsmittel zur Analyse

- Syntax (automatisch testbar)
- Datenflußfehler → LIVE/AVAIL-Algorithmen, zweite Stunde
- symbolische Ausführung
- Korrektheitsbeweise
- Informelle Analyse

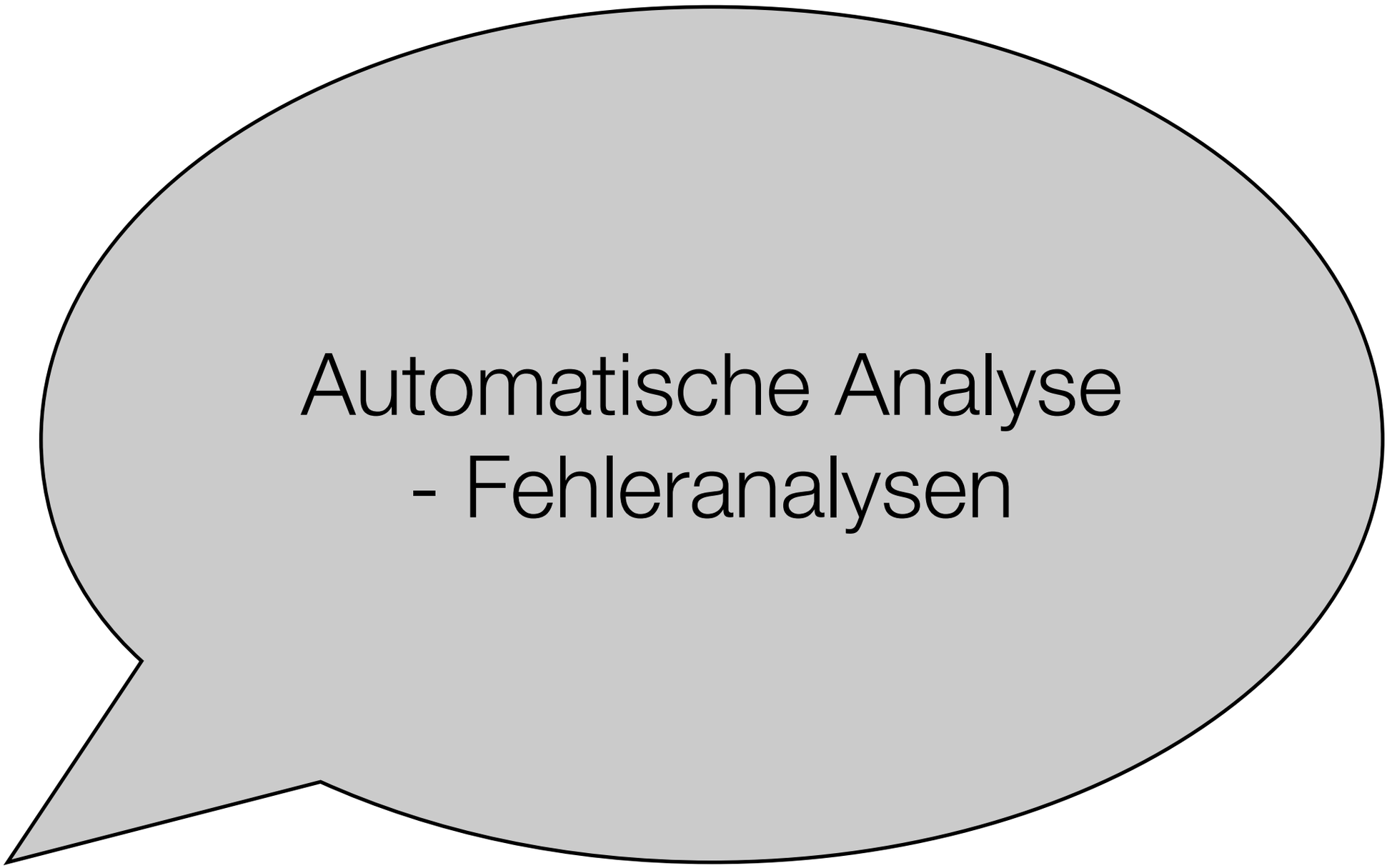
Objektkode

Für besonders kritische Software

- Vertraue nicht der Übersetzung Hochsprache → Maschinenkode

Hilfsmittel

- Maschinenkode rückübersetzen und erneut analysieren


Automatische Analyse
- Fehleranalysen

Syntaxfehler

typischerweise trivial abgehandelt:

- Compiler
- Präprozessor

Hilfreich in C/C++:

CFLAGS=-Wall -Werror

→ alle Warnungen aktivieren und als Defekt betrachten

Typ-Analyse

Formen der Typisierung

- Statisch (z.B. C, C++, Java, Haskell)
- Dynamisch (z.B. LISP, Python, Ruby)

Sprachen sollten *typsicher* sein, also folgendes nicht zulassen:

a = 5; b =“37“; c = a + b; (Potentielle Ergebnisse: 42 oder „537“)

Wünschenswert:

- Zähler- und Indextypen (für Schleifen/Arrays)
- Maßeinheitstypen

```
int a:cm/s;
int w:cm;
int z:g;

a = w/z;  /* Defekt */
```

Lebensdauer und Gültigkeitsbereiche von Variablen

Beispiel:

```
char *ptr1 = strdup(„zeichenkette“);

if(*ptr1==‘a‘)
{  char *ptr2 = strdup(„neue kette“);
   free(ptr1);
}

printf(ptr2);   /* Fehlermeldung zur Compilezeit */

printf(ptr1);   /* Infektion/Fehlfunktion zur Laufzeit */
```

→ Datenfluß-/Kontrollflußanalyse
→ garbage collection statt *free()*

Analyse von Prozeduren

i) Prozeduren sind selbst Parameter (anderer Prozeduren)

- Typüberprüfung

ii) Funktionsprozeduren

- dürfen keine Seiteneffekte haben (nur Rückgabewert)
- also z.B. keine globalen Variablen setzen

iii) Parameteranzahl und -Typen

- Problem: Beim Linken von zwei Objektdateien wird nichts mehr geprüft

Ereignisfolgen

Beispiele:

- fopen() vor fread()
- fopen(name, „w“) vor fwrite()

→statische Überprüfung anhand der Kontrollflußwege

→Zustandsdiagramm/endlichen Automaten bauen

Programmstruktur-Analyse

i) Analyse ob das Programm strukturiert ist
   - falls nein (gotos verwendet)
   →unerreichbare Anweisungen?
   →nicht verwendete label?

ii) Gibt es nicht ausführbare Wege?

iii) Terminierungsbedingungen von Schleifen
   - datenunabhängige Schleifen: trivial
   - datenabhängige Schleifen: Konvergenzbeweis von Hand

Datenabhängige Schleife: Indexvariable wird innerhalb der Schleife manipuliert

```
for(i=0; i<100; i++)
{  ...
   if(b) i--;          /* typischerweise unübersichtlich: keine gute Idee */
   ...
 }
```

Analyse vom Programmierrichtlinien

i) formale Richtlinien für Kommentare

ii) Layout-Richtlinien, zum Beispiel Klammerstil

iii) Einhaltung von Komplexitätsgrenzen

- Kodelänge pro Modul / Funktion
- Schachtelung von Schleifen / Fallunterscheidungen

iv) Namenskonventionen, z.B. CamelCase vs. under_score

v) verbotene Konstrukte, z.B. nur „sichere“ Speicherverwaltung

```
char buffer[80];                    char *buffer = xmalloc(80);
~~gets(buffer);~~                   buffer = getline(buffer, 80, stdin);
```

Ausdrucks-Analyse

i) Indexgrenzen bei Arrays, idealerweise wie in Pascal:

- a: array[1..10] of int;
- b: [1..20]; (* Indextyp *)
- a[b] → Fehlermeldung zur Compilezeit

ii) Test von Fließkommazahlen auf (Un-)Gleichheit:

- ~~if(r == s)~~ → if(fabs(r,s) < EPSILON)

iii) Fehlende Klammern

- a∗b/c∗d → ((a∗b)/c)∗d oder (a∗b)/(c∗d)

iv) Rechenfehler wie Division durch Null

Defekte durch symbolische Ausführung aufdecken

Programmkode:

① y := x -1

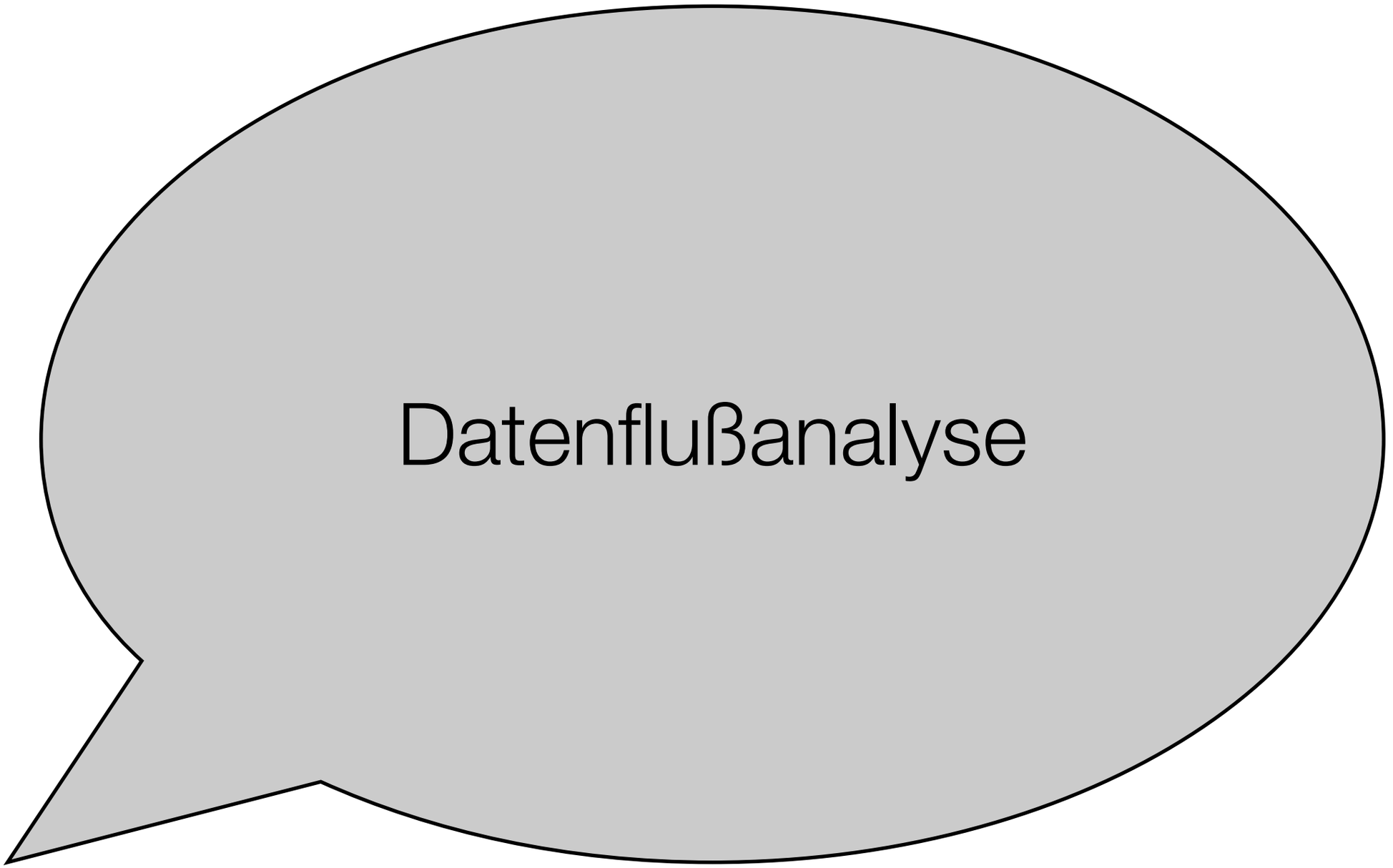
② if x = 1 then a = x/y

Symbolische Ausführung:
Substituiere x ← α

① y := α - 1

② „if“ soll erfüllbar sein, also wähle α = 1

x = α = 1
y = α - 1 = 0
a = 1 / 0 ⇒ Division durch Null aufgedeckt


Datenflußanalyse

Vorgehensweise

Benötigt: Datenflußgraph (aus der letzten Woche)

1. Alle Pfade durch den Datenflußgraphen durchlaufen

2. Für jeden Pfad und jede Variable notieren, wenn sie definiert, referenziert oder undefiniert wird (mit Abkürzungen d,r,u)

   → ergibt Pfadausdruck über {d,r,u} für jede Variable

3. Anomalie liegt bei folgenden Pfadausdrücken vor:

   i) ..ur.. undef/ref - Anomalie

   ii) ..dd.. def/def - Anomalie

   iii) ..du.. def/undef - Anomalie

Mögliche Probleme

1. Prozeduraufrufe
   - Aufblähen des Datenflußgraphen; insbes. bei globalen Variablen
2. Dynamisch wachsende Strukturen (Listen, Bäume, Graphen)
   - als monolithische Einheit (wie eine Variable) betrachten
3. Arrays
   - typischerweise nicht als n einzelne Variablen behandelbar - Bsp: a[i]
   - wie oben als monolithische Einheit nehmen
4. Schleifen
   - Anzahl der zu betrachtenden Wege? → siehe LIVE/AVAIL-Algorithmus

LIVE and AVAIL-Algorithmus (1)

Annotation des Kontrollflußgraphen mit folgenden Mengen:

$A \in gen(n)$ $\Leftrightarrow$ Token A wird in Knoten n „erzeugt“

$A \in kill(n)$ $\Leftrightarrow$ Token A wird in Knoten n „verbraucht“

$A \in null(n)$ $\Leftrightarrow$ Token A wird in Knoten n nicht verändert

Sei *tok* die Menge aller Tokens, dann gilt:

$gen(n) \cup kill(n) \cup null(n) = tok$

$gen(n) \cap kill(n) \cap null(n) = \varnothing$

LIVE and AVAIL-Algorithmus (2)

Bildung von Pfadausdrücken P(A; w) für Weg w und Variable A

- jeweils Buchstaben g,k,n für die besuchten Knoten notieren

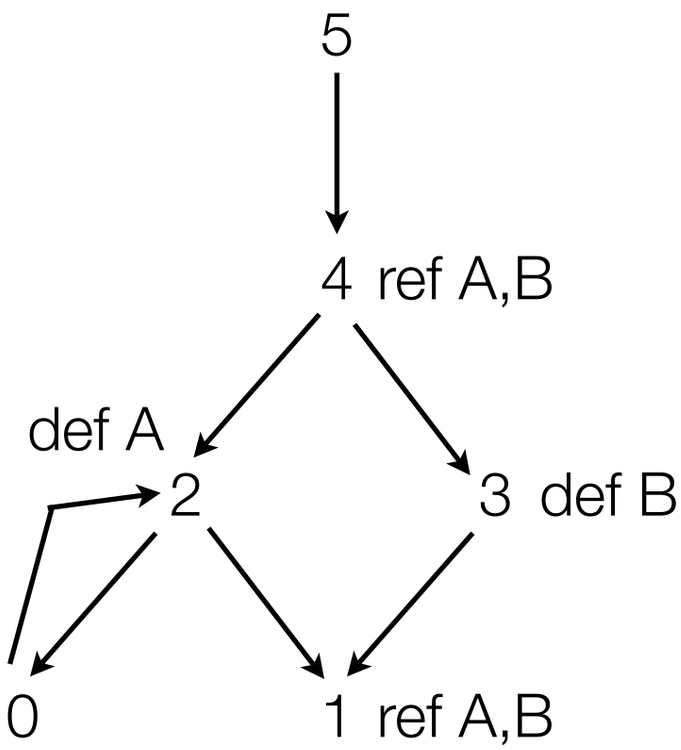
Reduzierter Pfadausdruck:

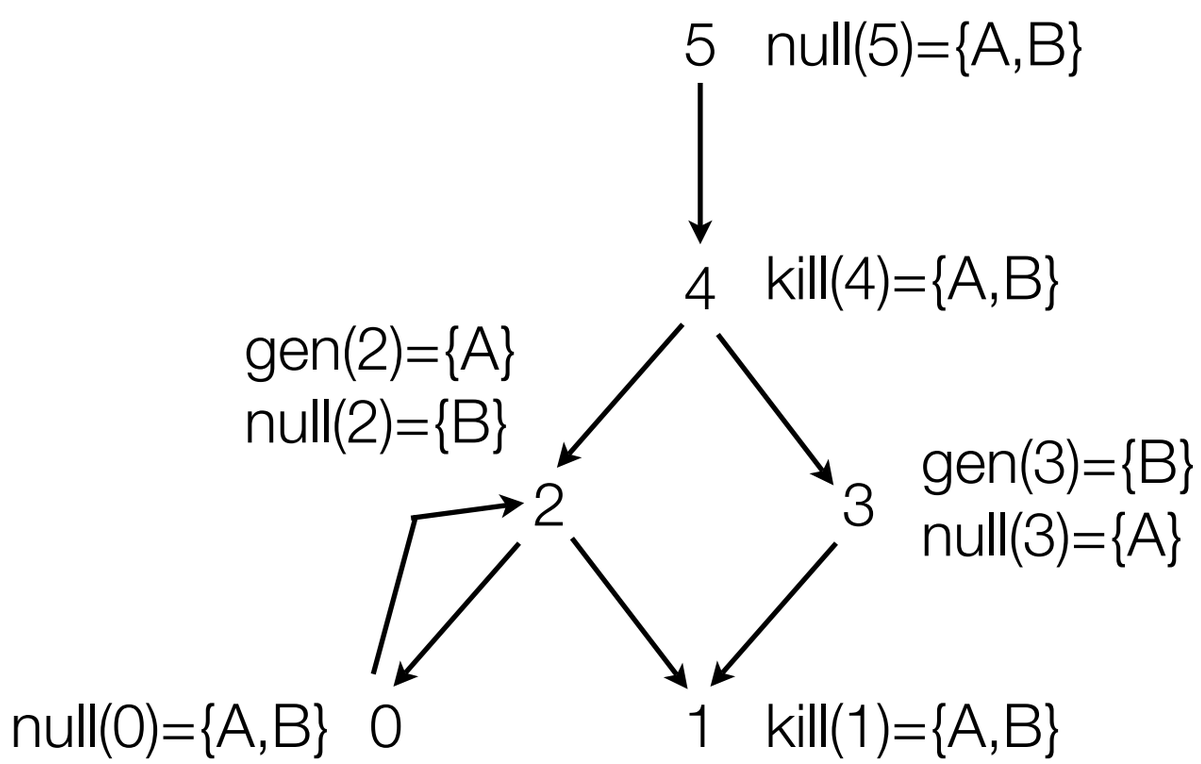
- Symbole n weglassen

Spezielle Notationen mit Hilfe regulärer Ausdrücke:

- P(A; k→) : Alle möglichen Wege die vom Knoten k ausgehen
- P(A; →k) : Alle möglichen Wege die zum Knoten k hinführen

Beispiel


5
4 ref A,B
def A
2
3 def B
0
1 ref A,B


5 null(5)={A,B}
4 kill(4)={A,B}
gen(2)={A}
null(2)={B}
2
3
gen(3)={B}
null(3)={A}
null(0)={A,B}
0
1 kill(1)={A,B}

LIVE and AVAIL-Algorithmus (3)

(Informelle Definition)

A $\in$ live(k) $\Leftrightarrow$ *einer* der Wege, der von Knoten k ausgeht, beginnt mit „g“

→ A kann in der Zukunft auf wenigstens einem Weg generiert werden

A $\in$ avail(k) $\Leftrightarrow$ *alle* Wege, die zu k hinführen, enden mit „g“

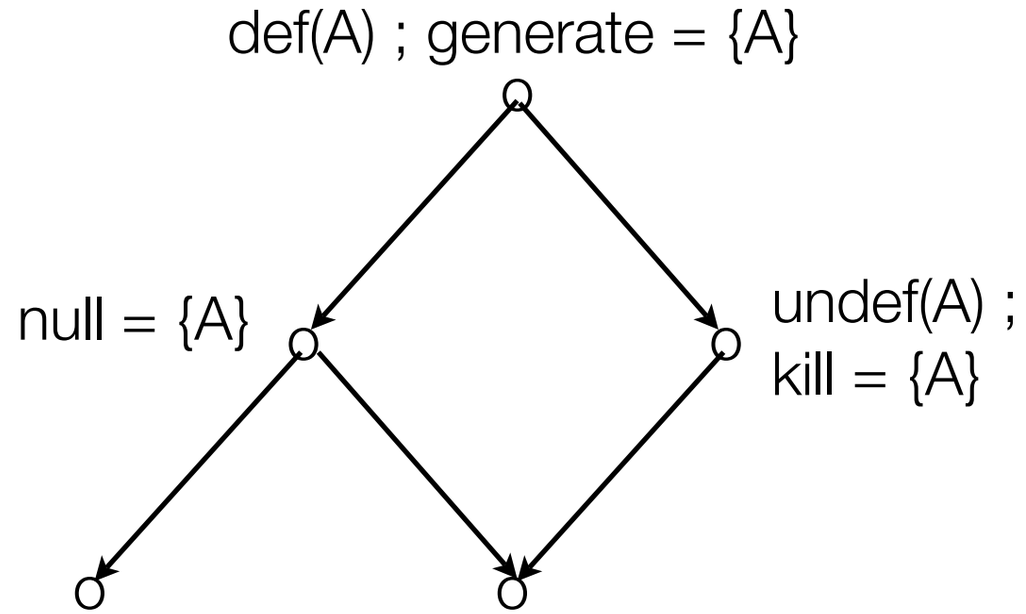
→ A ist in k immer verfügbar, egal auf welchem Weg k erreicht wird

Berechnung: einfache iterative Algorithmen; siehe Riedemann Seite 320

Undef/Ref-Anomalie finden

Verwende: generate(k) = def(k)
kill(k) = undef(k)

Berechne: avail(k)



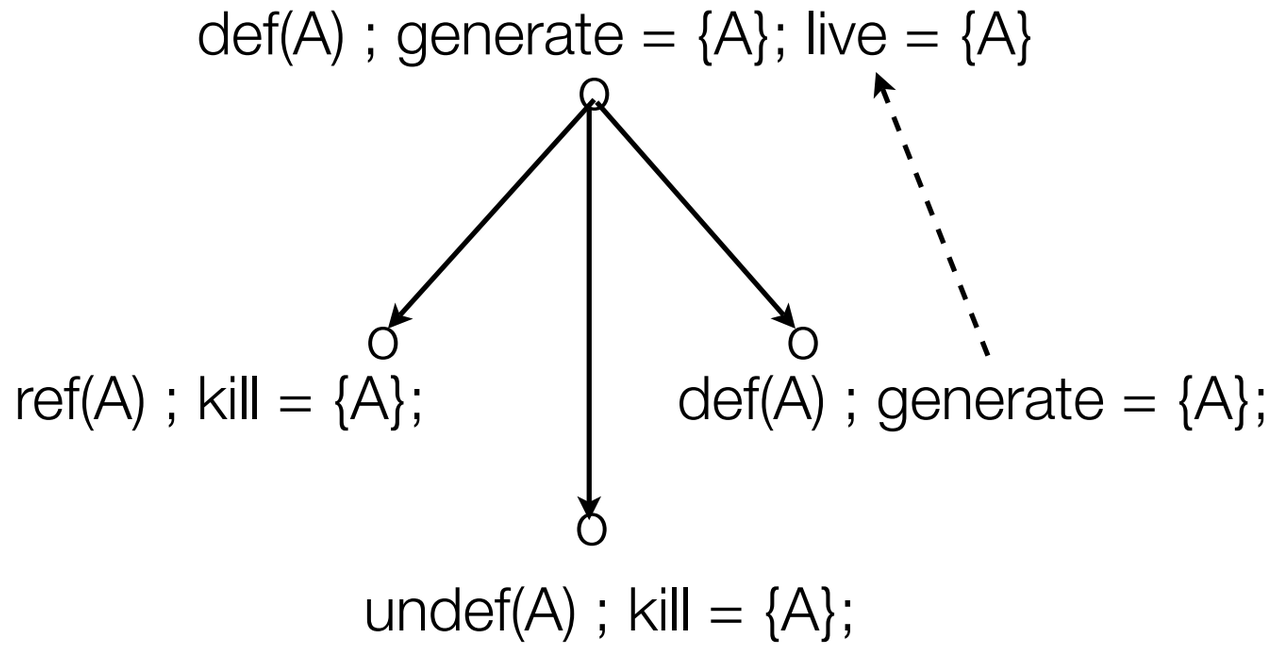
Falls A ∉ avail(k) und A ∈ ref(k) ⇒ Undef/Ref-Anomalie gefunden

Begründung: A ∉ avail(k) ⇒ ein Vorgänger von K endet auf kill/undef

Def/def-Anomalie finden

Verwende: generate(k) = def(k)
kill(k) = ref(k) ∪ undef(k)

Berechne: live(k)



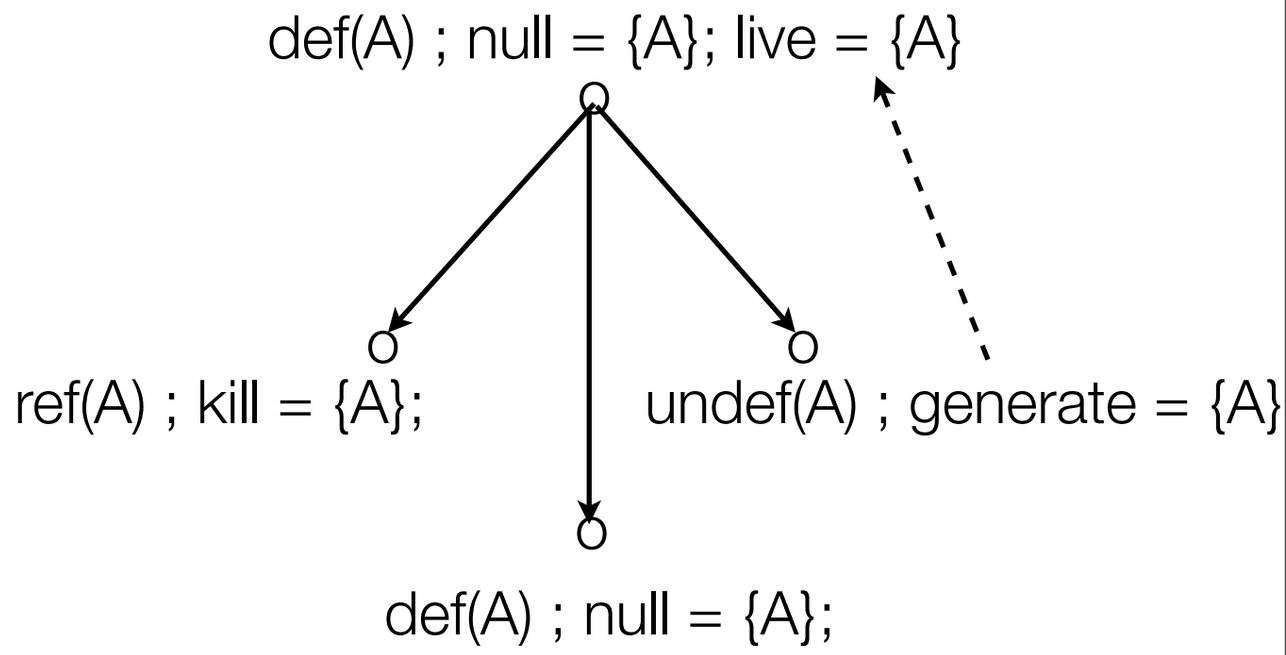
Falls A ∈ live(k) und A ∈ def(k) ⇒ def/def-Anomalie

Begründung: A ∈ live(k) ⇒ es findet unmittelbar mindestens eine weitere Def.

Def/undef-Anomalie finden

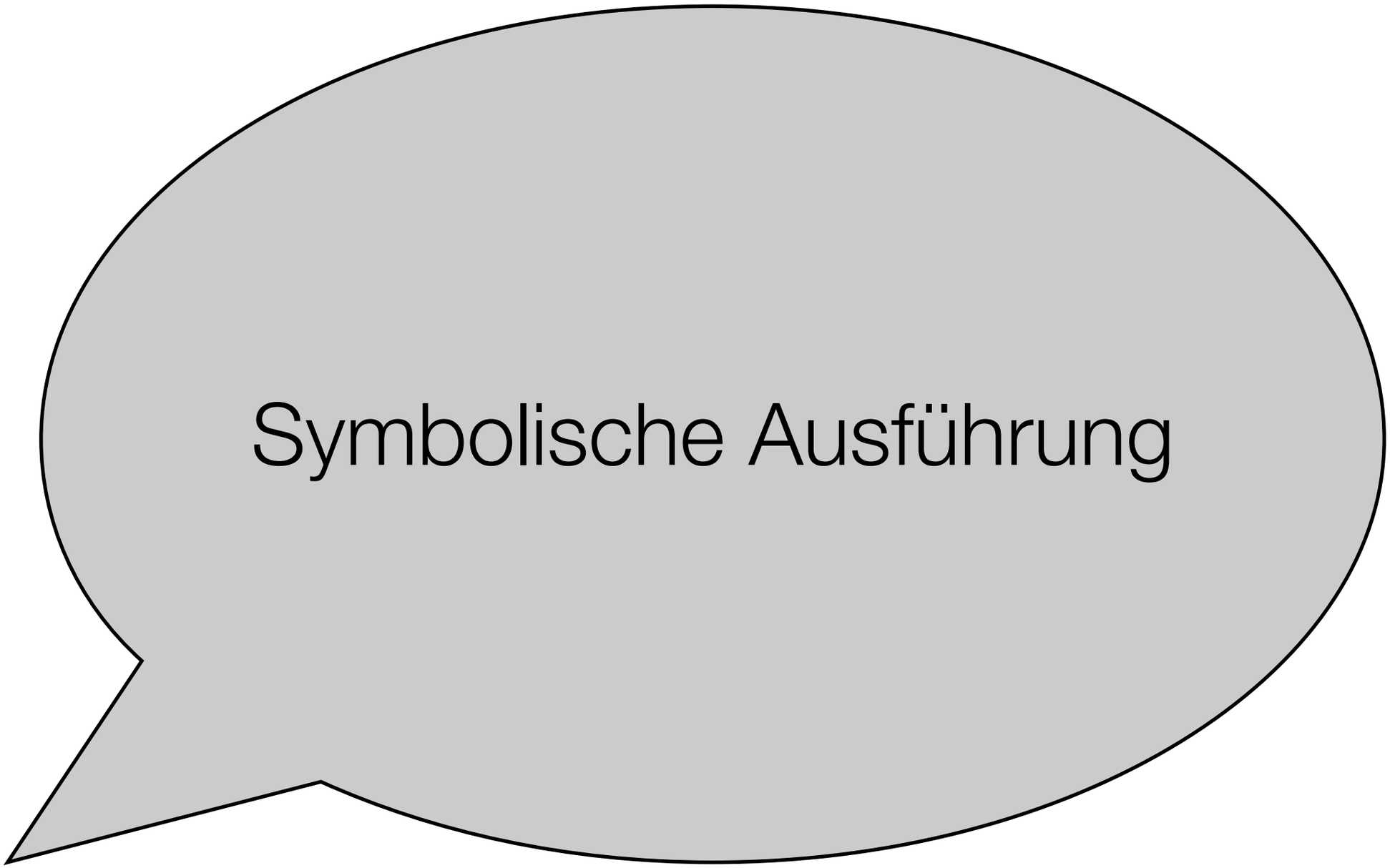
Verwende: generate(k) = undef(k)
kill(k) = ref(k)

Berechne: live(k)



Falls A ∈ live(k) und A ∈ def(k) ⇒ def/undef-Anomalie

Begründung: A ∈ live(k) ⇒ es findet unmittelbar mindestens ein undef


Symbolische Ausführung

Symbolisches Ausrechnen von Ausdrücken/Seq.

Normale Programmausführung rechnet mit konkreten Werten

→ Erweiterung: Rechne mit symbolischen Eingabewerten

→ entsprechend weiterrechnen mit resultierenden Ausdrücken

Beispiel: Symbolisches Ausrechnen von Ausdrücken / Sequenzen

C := A + 2∗B
D := C - A

c = a+2b | Setze Symbole a,b für A,B ein
d = a+2b-a | Setze symbol. Ausdruck für C und a für A ein
d = 2b | Vereinfachung
(Vorsicht: Rundungsfehler bei Fließkommazahlen)

Symbolisches Ausrechnen von Kontrollelementen

Exemplarisch am Beispiel **if** B **then** A1 **else** A2

Als Resultat des Ausrechnens von B kann herauskommen:

i) B ist immer wahr ⇒ symbol. Wert der Anweisung ist A1

ii) B ist immer falsch ⇒ symbol. Wert der Anweisung ist A2

iii) sonst: Pfadbedingungen für die Fälle wahr und falsch angeben

→ Baumstruktur von Wegen

→ Pfadbedingungen in den Zweigen mitschleppen / kombinieren

- bei Schleifen: Anzahl Iterationen begrenzen oder Induktionsbeweis

Komplexeres Beispiel

Exemplarischer Weg: 1, 2, 3 true, 4, 5 true, 6

① read a,b

② x = a * b + 2

③ if x>100

then x = 100-x

④ x = x - 50

⑤ if x < 0

then x = 0

⑥ print x

② x = a * b + 2

③ true → a*b+2 > 100 ⇔ a*b > 98

x = 100-(a*b+2) ⇔ x = 98-a*b

④ x = 98-a*b-50 ⇔ x = 48-a*b

⑤ true → 48-a*b < 0 ⇔ a*b > 48

da bereits a*b>98 aus ③ folgt ⇒ ⑤ immer wahr

⇒ x = 0

⑥ symbolische Ausgabe ist 0

Symbolisches Ausrechnen von Aufrufen

a) Symbolisches Ausrechnen gemäß Parameterbelegung und Prozedurkörper:

```
int foo(int a, int b) { return a + b; }

a := foo(x,y)    /* x,y in Prozedurkörper einsetzen */

... mit a = x+y weiterrechnen ...
```

b) Ausdruck nicht weiter auflösen

```
a := foo(x,y);
b := 2*foo(x,y);
c := a + b;

... mit c = 3*foo(x,y) weiterrechnen ...
```

Vorteile der symbolischen Ausführung

- Ein symbolischer „Test“ deckt viele reale Testdaten ab
- findet Defekte, bei denen für eine Teilmenge der Eingaben falsche Ergebnisse berechnet werden
- man findet nicht ausführbare if-Zweige
- unterstützt Finden von Schleifen-Invarianten

Nachteile der symbolischen Ausführung

- Überprüfen der Ergebnisse ist schwieriger
  - insbesondere bei nicht aufgelösten Prozeduraufrufen
  - ungenügend vereinfachte Ausdrücke
- Programmiersprache muß formal definiert sein
- Symbolischer Interpreter muß vorhanden sein
- Umformungen können Maschineneigenschaften verdecken (Rundungsfehler)
- es wird ein Theorembeweiser benötigt (der nicht vollständig sein kann)
- keine Leistungsmessung

Vorteile der statischen Analyse

- keine Codeänderungen, Treibermodule notwendig
- Analyse unausführbarer Systeme möglich
  (unvollständige / noch nicht lauffähige bzw. existierende Systeme)
- Mehrere Defekte können in einem Durchgang gefunden werden
- findet auch neue / unerwartete Klassen von Defekten

Grenzen der statischen Analyse

- zeitliche Zusammenhänge nur begrenzt analysierbar:

```
r = 0;
for(i=1; i<100; i++)
{  if(i==1) a=5;
   r=a+(i+1)+r;          /* keine undef/ref-Anomalie bzgl. a ! */
}
```

- Diagnose nicht automatisierbar
  (Warnungen/potentielle Defekte von Hand bewerten)

- Abhängigkeit vom Programmierstil; siehe obiges Beispiel

Ende der heutigen Vorlesung

Danke fürs Zuhören!

Bis nächste Woche :-)