1.4 Schrittweise Programmentwicklung

Nutzen der Verifikationsmethodik:

- „Konstruktion" von Programmen durch präzise Definition (den Zusicherungen!) von Programmzuständen
- Dokumentation des Codes

Systematisches Durchdenken der Bedingungen und Formulieren in anderer Sprache als Programmiersprache hilft Fehler erkennen.
(aber nicht alle....: Rundungsfehlern Gleitkommaarithmetik, Wertebereich von Typen, Typkorrektheit)

Beispiel für systematische Programmentwicklung

Anforderungsdefinition:
Gegeben eine Liste von Tagekursen einer Aktie für n0 Tage.
Gesucht: der maximale Spekulationsgewinn, das ist die größte Differenz von zwei Aktienkursen, wobei der Verkaufstag größer als der Tag des Kaufs ist.

(n0 ist hier eine Konstante)

Fallstudie Programmentwicklung (2)

Spezifikation: (P,Q) mit

P ≡ a.length = n0, ∀ 0 <= i < n0 a[i] = A[i]
(alle Werte definiert) ∧ n0 > 1

mindestens 2 Tage

Q ≡ g = max(a[j] - a[i]) ∧ 0 <= i < j < n0

0 <= i < j < n0

Fallstudie Programmentwicklung (3)

Invariante bestimmen

Methode B:

Konstante in Q durch Variable ersetzen,
n0 durch Variable n , also 2 <= n <= n0

INV : g = max a[j] -a[i], 0 <= i < j < n ∧ 2 <= n <= n0

g enthält immer den maximalen Kursgewinn der ersten n Tage

Schleifenbedingung :

folgt aus INV ∧ n = n0 => Q , also n != n0

Fallstudie Programmentwicklung (4)

Variablen: `g, n, a, i, j`
Invariante herstellen:

g = a[1] - a[0]

Erste Teillösung:

{n0 >= 2}
```
g = a[1] - a[0];
n = 2;
```
{INV]
```
while (n!=n0) {   // {INV ∧ n != n0}
```
bestimme Maximum der Differenz für 0 <= i < j <= n
und erhöhe n um 1
```
}
```
{INV}

Fallstudie Programmentwicklung (5)

Schrittweise Verfeinerung: Schleifenrumpf konstruieren

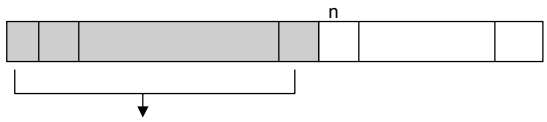
Zerlegen :

{Inv ∧ n != n0}
„ n berücksichtigen"
{INV $_{n+1}$ } // Invariante, bei der n durch n+1 ersetzt
n = n+1;
{INV}

„ n berücksichtigen "

INV $_{n+1}$: g = max a[j] - a[i], 0 <= i < j < n+1 ∧ 2 <= n+1 <= n0

Fallstudie Programmentwicklung (6)



g = max a[j]-a[i], 0 <= i < j < n

g vergrößert sich nur bei Erweiterung auf n+1 Feldelemente, wenn eine der Differenzen a[n] – a[i] > g, i<n

g_{n+1} = max (a[j] -a[i]) , 0 <= i < j < n+1
= max2 (max (a[j] - a[i]), 0 <= i < j < n , max (a[n] - a[i] , 0 <= i < n))

Fallstudie Programmentwicklung (7)

Maximum max (a[n] - a[i], 0 <= i < n) = a[n] – min (a[i], 0<=i < n))
=>
g = max2 (max (a[n] - a[i], 0 <= i < n) , a[n] - min (a[i], 0<= i < n))

Minimum der a [i] merken, 0 <= i <n, Variable `mina` !

=> **`g = max2 (g, a[n] - mina)`**

Zusätzliche Invariante: INV' = mina = min a[i] , 0 <= i < n

=> Erweiterte Invariante: INV_{ges} = INV $\wedge$ INV'

Fallstudie Programmentwicklung (5)

Herstellen von INV':

```
mina = min2(a[0], a[1])
```

Herstellen von INV' beim Übergang n -> n+1:

```
mina = min a[i], 0 <= i < n
```

rekursiv: `mina = min2(mina, a[n-1])`

Fallstudie Programmentwicklung (6): Java-Methode

```
static float maxWin(float a[]) {
   //requires:  a.length = n0 > 1,
   //           for all 0 <= i < n0  a[i] = A[i]
   //effects:  returns g = max a[j] - a[i] , 0 <= i < j < n0
    int n0 = a.length;
    int n = 2;
    float mina = min2( a[1], a[0]); // min2: Minimum of 2 values
    float g = a[1] - a[0];
    //assert: g = max ( a[j] - a[i]  0 <= i < j < n, 2 <= n <= n0,
    //        mina = min a [i], 0 <= i < n
    while ( n!=n0) {              //  {INV and n != n0}
       mina = min2(mina, a[n-1]);
       g = max2(g, a[n] - mina);
       n = n+1;
     }
     return g;
  }
```

Linearer Algorithmus !

Fallstudie Programmentwicklung (7)

- Quadratischer Algorithmus

```
static float maxWin(float a[]) {
   //requires:  a.length = n0 > 1,
   //           for all 0 <= i < n0  a[i] = A[i]
   //effects:  returns g = max a[j] - a[i] , 0 <= i < j < n0
    int n0 = a.length;
    float g = a[1]-a[0];
    //assert:  Inv:  1<= j < n0 , 0 <= i < j,
     //               diff = max (a[j]-a[i])
    for (int j = 1; j < n0; j++) {
       for (int i = 0; i<j; i++) {
         float diff= a[j]-a[i];
        if (diff>g) g=diff;
         }
     }
     return g;
  }
```

Quadratischer Algorithmus !

Zusammenfassung Spezifikation und Verifikation

1.5 Spezifikation, Verifikation, Testen

Semantik einer Programmiersprache dient u.a

- zur Definition der Programmiersprache (hier: axiomatisch), ist Grundlage zur Implementierung von Übersetzer / Interpretierer
- Programmspezifikation
  was soll das Programm tun?
  Präzisierung der Anforderungsdefinition
- Programmverifikation
  Beweis, dass Programm tut, was es soll

Verifizieren oder Testen?

- **Programmfehler:** Programm tut nicht das, was es soll
  oder tut etwas, was es nicht soll
- Es gibt praktische keine (sehr großen) fehlerfreien Programme
  häufig völlig unabhängige Entwicklung (Betriebssystem, Anwendungsprogramm),
  Probleme im Zusammenwirken der Teile!
- Für manche Programme ist formale Verifikation ein Qualitätsmerkmal
  Zertifizierung von Programmen nach formaler Verifikation,
  Bsp.: Steuerprogramme in Verkehrsleitsystemen,
  ABS-Software,...

Verifizieren oder Testen?

- Begleitende Verifikation vermindert Fehler und unterstützt die Konstruktion von Algorithmen
- Ohne Spezifikation keine Verifikation,
  .... nicht einmal ein sinnvolles Programm
- meist aufwendig
  Verifikationsprogramme wünschenswert
- Tests beweisen (ggf.) die Anwesenheit von Fehlern,
  nie die Abwesenheit
  (wenn unendlich viele mögliche Eingaben)

Anmerkungen zum Testen(1)

Klassifikation

Schnittstellen- (Funktions-) test (blackbox test)

Ein- / Ausgabe-Relation auf Konformität zur Spezifikation prüfen

Programmabhängiger Test (whitebox/glassbox test)

Überprüfung möglichst großer Teile aller Pfade durch das Programm

Möglichst große Überdeckung (des Programmcodes) wünschenswert

Hauptproblem: kombinatorische Explosion der Testfälle, deshalb Vollständigkeit meist illusorisch.

Anmerkungen zum Testen(2)

Systematische Auswahl von Testfällen:

Schnittstellentest (Funktionstest)

pro spezifizierter Bedingung mindestens ein Testfall

Randbereiche (ggf. von beiden Seiten) prüfen, Maximal-, Minmalwerte

Genügende Anzahl von Normalfällen

Zufällige Erzeugung von Testfällen

Überdeckungstest erwünscht, aber kaum machbar:

Wegüberdeckung: jeden Weg einmal durchlaufen

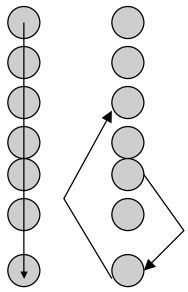
Abschwächung: jede Anweisung einmal durchlaufen (**Anweisungsüberdeckung [C0]**)

stärker : **zweigüberdeckend [C1]**: jeden Zweig mindestens einmal durchlaufen

Beispiel

```
count=0;
cin = read...;
  while (!in.eof){
  count++;
  if ((cin=='a') || (cin =='e')
      ||...)) vowl++;

cin = read...;
  }
```



anweisungsüberdeckend zweigüberdeckend

Wegüberdeckend: jeder mögliche Pfad im Programm wird einmal durchlaufen, etwa bei der Fallunterscheidung 2^n Möglichkeiten bei n Schleifendurchläufen!

Werbung.....