Wykład 10

Testowanie w JEE 5.0 (3)

Autor: Zofia Kruczkiewicz

Testowanie programów
Problem stopu programu (1)

Przykład programu, który *zatrzymuje się* (podano tekst w pseudojęzyku) dla liczb naturalnych nieparzystych, natomiast nie zatrzymuje się dla liczb parzystych.

dopóki X≠1 dopóty wykonuj X ← X-2

- **zatrzymaj się**

Testowanie programów
Problem stopu programu (2)

Przykład programu, który się *zawsze zatrzymuje* dla dowolnych liczb naturalnych, ale *nie można tego formalnie udowodnić*.

Oznacza to brak możliwości pełnej automatyzacji testowania.

- **dopóki X $\neq$ 1, dopóty wykonuj:**
  - **jeśli X jest parzyste, wykonuj X $\leftarrow$X/2**
  - **w przeciwnym przypadku (X nieparzyste) wykonaj X$\leftarrow$3*X +1**
- **zatrzymaj się**

3. Proces testowania symbolicznego

Testowanie i usuwanie błędów prowadzą do osiągnięcia dużej niezawodności programów. Jednak nawet przy dużych nakładach na testowanie można przeanalizować zaledwie małą część wszystkich możliwych kombinacji danych wejściowych wielkiego systemu oprogramowania.

Uwagi dotyczące poprawy testowania:

- unikać struktur typu *goto*
- dostosować odpowiednio struktury danych do wykonywanych algorytmów, i na odwrót (Wirth)
- tworzyć programy łatwo modyfikowalne ze względu na struktury danych
- ograniczyć powiązania między modułami programu
- inicjować zmienne, ograniczyć zmienne globalne, rozważnie operować wskaźnikami, przydziałem pamięci, indeksami tablic (C,C++!)
- należy zabezpieczyć program przed przekroczeniem zakresu wartości danych i przed pomyłkami przy ich wprowadzaniu (nie należy zakładać, że ten obowiązek powinien spoczywać na „użytkowniku” np. zapobieganie dzielenia przez zero)

Problemy testowania

- trudność w określeniu możliwie najmniejszej liczby zachowań programu, wynikającego z pewnego zbioru danych, które należy sprawdzić i uogólnić indukcyjnie uzyskane wyniki
- w podejściu statystycznym istnieje tendencja do ułatwiania postępowania i opierania się na często niezbyt dobrze uzasadnionych założeniach (losowy rozkład danych, wzajemna niezależność czynników badanych procesów, operowanie średnią lub wariancją)

Rozwiązanie dużej liczby rozpatrywanych danych można zastąpić metodą wykonywania symbolicznego, opartej na:

- symbole bądź wyrażenia algebraiczne używane są jako wartości zmiennych. Instrukcje podstawienia podstawiają za zmienne wyrażenia algebraiczne
- wybór gałęzi przy instrukcji warunku wprowadza ograniczenia dla symboli
- wykonywanie symboliczne dotyczy całych, często nieskończenie wielkich zbiorów instrukcji, co ogranicza wykorzystania szczególnych atrybutów wartości, które może przybrać symbol.

Przykład 1: Przykład symbolicznego wykonania programu sprowadzony do odpowiedniego testowania warunków bez analizowania wartości zmiennych

```
#include "stdio.h"
void main ()
{ float x,y,z;
   // zabezpieczenie przed niewłaściwą formą danych x i y
   if (scanf("%f%f",&x,&y)==2)
   {    z=2*x + y;
        if (z==0)  x=1;
                  //zabezpieczenie przed niewłaściwą wartością danych
        else x=1/z;}
}
```

Przykład 2: Testowanie błędnej wersji programu do znajdowania pierwiastka kwadratowego z *p*, gdy przedziału $0 \leq p < 1$ z dokładnością do *err*, gdzie $0 \leq err < 1$.

[J.M Brady, Informatyka teoretyczna w ujęciu programistycznym]

```
#include "stdio.h"
float pierwiastek_kw(float p, float err)
{ float d=1, ans=0, tt=0, c=2*p;
    //wylicz pierwiastek kwadratowy z p, 0<=p<1 z dokładnością do err, 0 <= err < 1
  if (c >= 2) return 0;                          //punkt rozgałęzienia A, p<1 ?
  do
  { if (d<=err) return ans;                      //punkt rozgałęzienia B
    d=0.5 * d;
    tt=c-(d+2*ans);
    if (tt>=0)                                   //punkt rozgałęzienia C
     { ans=ans+d;                          //ten i kolejny wiersz powinny być zamienione
       c=2*(c-(2*ans+d)); }
    else c=2*c;
   } while (1);
}
```

Sekwencje programu	p	err	d	ans	tt	c
A false	p < 1		1	0	0	2*p < 2
B false	p < 1	**err < 1**	d > err	0	0	2*p < 2
C true ?	p >=0.25	err < 1	0.5	0.5	2*p - 0.5 >= 0	4*p - 3
B true, exit	**0.25<=p<1**	**0.5<=err<1**	0.5	0.5	2*p - 0.5 >= 0	4*p - 3
A false ?	p < 1		1	0	0	2*p < 2
B false ?	p < 1	err < 1	d > err	0	0	2*p < 2
C true ?	p >=0.25	err < 1	0.5	0.5	2*p – 0.5 >= 0	4*p - 3
B false ?	p >=0.25	err < 0.5	0.5	0.5	2*p – 0.5 >= 0	4*p – 3
C false !	0.25<=p<1	**err < 0.5**	0.25	0.5	4*p – 4.25 < 0	8*p - 6
B true exit	**0.25<=p<1**	**0.25<= err<0.5**	0.25	0.5	4*p – 4.25 < 0	8*p - 6

Po sekwencji <*A false, B false, C true, B true*> mamy:

- **ans = 0.5**
- $\mathbf{p^{1/2} - err \leq ans \leq p^{1/2} + err}$
- $p = err = 0.995$, $p^{1/2} \approx 0.997$
- $\mathbf{p^{1/2} – err = 0.997 – 0.995 = 0.002}$
- $\mathbf{p^{1/2} + err = 0.997 + 0.997 = 1.994}$

Po sekwencji <*A false, B false, C true, B false,C false, B true*> mamy jednak

- **ans = 0.5**
- $\mathbf{ans < p^{1/2} - err}$
- $p = 0.995$, $err = 0.49$, $p^{1/2} \approx 0.997$
- $\mathbf{p^{1/2} – err = 0.997 – 0.49 = 0.507}$

Program nie przeszedł pomyślnie testu, jednak nie znaleziono przyczyny błędu.

Przykład 3: Wyszukiwanie binarne

```
int SzukP(int L, int P, element klucz, int& ktory, element T[])
{  int jest=0;
   while (L<=P && jest==0)
    {  ktory = (L + P) / 2;
       if (T[ktory] < klucz) L = ktory + 1;
       else
           if (T[ktory] > klucz) P = ktory - 1;
           else
        jest = 1;
     }
    return jest;
  }
```

```
int SzukP(int L, int P, element klucz, int& ktory,
                element T[])
{   int jest=0;
    while (L<=P && jest==0)                                  //A
        {
          ktory = (L + P) / 2;
          if (T[ktory] < klucz) L = ktory;                   //B
          else
             if (T[ktory] > klucz) P = ktory - 1;            //C
             else
                 jest = 1;
        }
    return jest;
 }
```

Sekwencje programu	L	P	ktory	T[ktory]	jest
A true	L<=P	L	(L+L)/2 = L		0
B true	L<=L	L	L	T[L] < klucz1	0
A true błąd (pętla nieskończona)	L<=P	L	L	T[L] < klucz1	0
A true	L<=P	L	(L+L)/2 = L	T[L] > klucz2	0
C true	L<=P	L-1	L	T[L] > klucz2	0
A false Exit	L>L-1	L-1	L	T[L] > klucz2	0