**Tanulási útmutató**

**Összefoglalás**

**Tesztelés**

A tesztelés célja, hogy minél több hibát megtaláljunk a programban. Ahhoz, hogy az összes hibát fölfedezzük, kézenfekvőnek tűnik a programot kipróbálni az *összes*lehetséges bemenő adattal. Ez azonban sajnos nem lehetséges. Nézzünk egy egyszerű példát:

|  |  |
| --- | --- |
| **Pszeudokódos algoritmus** | **Struktogram** |
| ***Program****:*  ***Változó*** *I,J:Egész*  ***Be****: I,J*  ***Ki****: I,J*  ***Program******vége****.* | A mellékelt program algoritmusa, struktogrammal. |

**Megjegyzés**

A struktogramban – általában – nem jelennek meg az adatok deklarációi, ez kivétel. A külső dobozrészben az algoritmus neve – általában – az eljárás, ill. függvény fejsora, azaz neve és paraméterezése.

Mivel 216 különböző értékű egész számot tudunk tárolni, ezért az összes lehetőség 232, aminek a leírásához már 9 számjegyre van szükség. Ez rengeteg időt venne igénybe, így nem is járható út. A másik végletet a következő – két szám osztására (!) készült – meglepő programrészleten mutatjuk be:

|  |  |
| --- | --- |
| **Pszeudokódos algoritmus** | **Struktogram** |
| ***Program****:*  ***Változó*** *A,B:Egész*  ***Be****: A,B*  ***Ki****: A\*B [A/B kiírása]*  ***Program vége****.* | A mellékelt program algoritmusa, struktogrammal. |

Ha ezt a programot olyan bemenő adatokkal próbáljuk ki, amelyben A=0 vagy B=1, akkor a program helyesen működik, a hibát nem tudjuk felfedezni. Ezután azt gondolhatnánk, hogy reménytelen helyzetbe kerültünk: hiszen minden lehetséges adattal nem tudjuk kipróbálni a programot; ha pedig kevesebbel próbáljuk ki, akkor lehet, hogy nem vesszük észre a hibákat. A helyzet azért nem ennyire rossz: célunk csak az lehet, hogy a tesztelést olyan módszerrel hajtsuk végre, amellyel a próbák száma erősen lecsökkenthető.

**Tesztesetnek a be- és kimeneti adatok és feltételek együttes megadását nevezzük.** Akkor tudunk a tesztelés eredményeiről bármit is mondani, ha van elképzelésünk arról, hogy adott bemenő adatra milyen eredményt várunk.

Fogalmazzuk meg a tesztelés alapelveit:

* A jó teszteset az, ami nagy valószínűséggel egy még felfedetlen hibát mutat ki a programban. Például két szám legnagyobb közös osztóját számoló programot az [5,5] adatpár után a [6,6]-tal teljesen felesleges kipróbálni (ugyanis igencsak rafinált, valószínűtlen elírás esetén viselkedhet a program [6,6]-ra másként, mint [5,5]-re).
* A teszteset nemcsak bemenő adatokból, hanem a hozzájuk tartozó eredményekből is áll. Egyébként nem tudnánk a kapott eredmény helyes vagy hibás voltáról beszélni. A későbbi felhasználás miatt célszerű a teszteseteket is leírni a fejlesztői dokumentációban vagy egy önálló tesztelési jegyzőkönyvben.
* A meg nem ismételhető tesztesetek kerülendők, feleslegesen megnövelik a program-tesztelés költségeit, idejét. Nem is beszélve arról a bosszúságról, amikor a programunk egy hibás futását nem tudjuk megismételni, és így a hiba is felfedetlen marad.
* Teszteseteket mind az érvénytelen, mind az érvényes adatokra kell készíteni.
* Minden tesztesetből a lehető legtöbb információt „ki kell bányászni”, azaz minden teszteset eredményét alaposan végig kell vizsgálni. Ezzel jelentősen csökkenthető a szükséges próbák száma.
* Egy próba eredményeinek vizsgálata során egyaránt fontos megállapítani, hogy miért nem valósít meg a program valamilyen funkciót, amit elvárunk tőle, illetve hogy miért végez olyan tevékenységeket is, amelyeket nem feltételeztünk róla.
* A program tesztelését csak a program írójától különböző személy képes hatékonyan elvégezni. Ennek oka, hogy a tesztelés nem „jóindulatú” tevékenység, saját munkájának vizsgálatához mindenki úgy áll hozzá, hogy önkéntelenül jónak feltételezi.

A programtesztelés módszereit két csoportba oszthatjuk aszerint, hogy a tesztelés során végrehajtjuk-e a programot, vagy nem. Ha csak a program kódját vizsgáljuk, akkor **statikus**, ha a programot végre is hajtjuk a tesztelés során, akkor **dinamikus**tesztelésről beszélünk.

**1. Statikus tesztelési módszerek**

**1.1. Kódellenőrzés**

A kódellenőrzés a program szövegének megvizsgálását jelenti. Az algoritmus logikáját kell ekkor a programban végigkövetni, s megfigyelni, hogy a kettő eltér-e egymástól. Csupán a kód alapján is viszonylag könnyen tud hibákat felfedezni egy avatatlan, a program készítőjétől különböző személy. Sokszor szerencsés, ha a program készítője elmondja valakinek – soronként –, hogy mit csinál a program. Ilyenkor gyakran saját maga fedezi fel a hibákat. Ez általában is igaz: a helyes megértés leghatásosabb próbája az új ismeretek továbbadása, megmagyarázása. Eközben derülhet fény a gondolatsor, a program rosszul, felületesen értelmezett vagy kidolgozatlan részeire.

**1.2. Formai ellenőrzés, kereszthivatkozási táblázatok**

Egy programban az előforduló hibákat két csoportra oszthatjuk: formai (szintaktikai), illetve tartalmi (szemantikai) jellegű hibákra. Ha értelmezőt használunk, akkor a formai ellenőrzést megfelelően választott tesztadatokkal való kipróbálással lehet elérni. A program minden utasítását végre kell hajtani legalább egyszer.

Igen sok információt szolgáltat egy programról, ha különböző kereszthivatkozás-táblázatokat készítünk róla (keresztreferencia). Ennek egyik típusa a változókról készült táblázat, melynek oszlopai a következőket tartalmazhatják: adatnév, típus és a hivatkozási helyek pontosítása.

A hivatkozási helyeket azonosító lista lehet egy sorszámlista (vagy eljárásazonosító és azon belüli relatív sorszám), amiben azok a sorok sorszámai szerepelnek, amelyekben az adott változó előfordul. A sorszám mögött speciális jellel (pl. \*-gal) jelölhetjük, hogy a változó értéket kapott az adott sorban.

**Példa**

Az alábbi példában előfordul egy G globális változó és egy ALFA a Forgat eljáráson belül definiált, lokális adat.

|  |  |  |
| --- | --- | --- |
| **adatnév** | **típus** | **hivatkozási helyek azonosítása** |
| Forgat.ALFA | Valós | Forgat 19\*, 22, 25 |
| G | Egész | 29, 53\* |

**Megjegyzés**

Természetesen kézzel készíteni ilyen táblázatot nagyon munkaigényes feladat, sok esetben azonban a fordítóprogram biztosítja automatikusan ezt a lehetőséget.

**1.3. Tartalmi ellenőrzés, ellentmondás keresés**

A formai hibáknál sokkal nehezebb felfedni a programban a tartalmi hibákat. Ezek formailag helyes programokban fordulnak elő, de mégis felismerhetők formális eszközökkel. Ezek általában önmagukban értelmetlenek, a programba valószínűleg elírás, figyelmetlenség miatt kerültek bele (de korántsem biztos, hogy tényleg hibásak).

**1.3.1. Felhasználatlan „objektum”**

Az egyes programegységek deklarációs részében szereplő konstansok, típusok, változók, eljárások, függvények, operátorok, más modulokra hivatkozások a programegység törzsében szerepelnek-e.

Ez a vizsgálat a beágyazott programegységekre is kiterjesztendő.

**1.3.2. Felhasználatlan változóérték**

Ilyen hiba lehet az, hogy egy változónak értéket adunk, de ezután nem használjuk semmire, vagy közvetlenül utána még egyszer értéket kap.

**Példa**

Az alábbi példában valószínűleg a fölösleges *I:=1* értékadás egy korábbi „amíg”-os ciklus javítása után maradhatott benn:

|  |  |
| --- | --- |
| **Pszeudokódos algoritmus** | **Struktogram** |
| *I:=1*  ***Ciklus*** *I=1****-től*** *5****-ig***  *...*  ***Ciklus vége*** | A mellékelt kóddarab algoritmusa, struktogrammal. |

**1.3.3. Érték nélküli változó**

Az ellenkező véglet, amikor egy változó, amelyiknek nincs kezdőértéke, előbb szerepel kifejezésben, mint értékadás bal oldalán.

Ez a fajta hiba nehezen vehető észre mindig formálisan, a program (legalább szimbolikus) végrehajtása nélkül.

**Példa**

Példa erre a következő programrészlet:

|  |  |
| --- | --- |
| **Pszeudokódos algoritmus** | **Struktogram** |
| ***Ha*** *f* ***akkor*** *I:=1*  ***különben*** *J:=1*  *K:=I* | A mellékelt kóddarab algoritmusa, struktogrammal. |

**1.3.4. Nem változtató értékadás**

Ha egy változó önmagának ad értéket, esetleg néhány hatástalan érték kombinálásával, az vagy felesleges, vagy pedig egy esetleges elírásra utalhat.

**Példa**

*I:=1\*I-0 [talán az I:=I\*I-O a helyes, vagy I:=I\*I-I0?]*

*Van:=Van≠Hamis*

**Megjegyzés**

Itt és a továbbiakban, ha a pszeudokódos és a struktogramos algoritmus minimálisan tér el, csak a pszeudokódost adjuk meg, és nem „zárjuk” dobozba a műveleteket.

**1.3.5. Azonosan igaz/hamis logikai formulák**

Az elágazás- és ciklusfeltételek formailag helyesek lehetnek úgy is, hogy a bennük szereplő változók értékétől függetlenül mindig azonos értéket vehetnek fel. Ekkor az elágazás egyik ága biztosan nem hajtható végre, a ciklus pedig vagy végtelen lesz, vagy pedig a ciklusmagot egyszer sem hajtjuk végre.

**Példa**

Egy-két példa ilyen feltételekre: *I<1* ***és*** *I≥100*, *I<I+1*

**1.3.6. Konstans értékű kifejezések**

A logikai feltételekhez hasonlóan az aritmetikai kifejezéseknél is gyanús a konstansság, amikor a kifejezés vagy egy részének értéke nem függ a benne szereplő változóktól. Ez egyes esetekben viszonylagos bonyolultságuk miatt nehezen észrevehető lehet:

**Példa**

*X:=A2-B2-(A+B)\*(A-B)*

*Y:=cos(X)/sin(X)\*tan(x)*

Vagy egy kevésbé nyilvánvaló példa:

*X:=A3/((A-B)\*(A-C))+B3/((B-C)\*(B-A))+C3/((C-A)\*(C-B))*

ami egyszerűsítve mindössze:

*X:=A+B+C*

**Megjegyzés**

Ennek, illetve az előzőnek a felismeréséhez kifejezések algebrai egyszerűsítőjével kell rendelkeznünk! (Ez egyébként hatékonyságjavítóként is működhet, jó fordítóprogramokban van ilyen optimalizáló eszköz.)

**1.3.7. Végtelen ciklus**

Az eddigiektől eltérő végtelen ciklust okozhat, ha egy számlálós ciklusban – amennyiben a nyelv szintaxisa egyáltalán megengedi – megváltoztatjuk a ciklusváltozót:

|  |  |
| --- | --- |
| **Pszeudokódos algoritmus** | **Struktogram** |
| ***Ciklus*** *I=1****-től*** *N****-ig***  *...*  *I:=1*  ***Ciklus vége***  A ciklusbeli értékadás talán *I:=I-1* vagy *I:-1* ? | A mellékelt kóddarab algoritmusa, struktogrammal. |

Szintén végtelen ciklusra utal, ha egy feltételes ciklusnál a ciklusfeltételben szereplő változók a ciklusmagban nem változhatnak meg (mert pl. nem szerepelnek értékadás bal oldalán). Sajnos a programozási nyelvek megengedik, hogy a ciklusváltozót a ciklusmagban hívott eljárás is módosítsa, emiatt a biztos felismerése nagyon nehéz.

**1.3.8. Érték nélküli függvény, operátor**

Akkor fordul elő, ha függvény- vagy operátordefiníció elágazást tartalmaz, és az elágazás valamelyik ágán nem adunk vissza függvényértéket.

**Példa**

|  |  |
| --- | --- |
| **Pszeudokódos algoritmus** | **Struktogram** |
| ***Függvény*** *Absz(X):Egész*  ***Ha*** *X<0* ***akkor*** *Absz:=-X*  ***Függvény******vége****.* | A mellékelt függvény algoritmusa, struktogrammal. |

**1.3.9. Mellékhatással járó függvény, operátor**

Matematikai értelemben a függvények a függvényérték meghatározásán kívül mást nem csinálhatnak. A programozási nyelvek többsége azonban megengedi, hogy a függvények megváltoztassák paramétereik értékét, vagy akár globális változókat is.

**Példa**

Ebből sok probléma származhat, például az *Y:=f(X)+f(X)*  és az *Y:=2\*f(X)*  kifejezés értéke különböző is lehet!.

**2. Dinamikus tesztelési módszerek**

Az előző részben a statikus tesztelési módszereket vizsgáltuk, ahol a tesztelést a program végrehajtása nélkül, a program szövegének vizsgálatával végeztük. A dinamikus tesztelési módszerek alapelve éppen az, hogy a programot működés közben vizsgáljuk.

Teszteseteket kétféle módon tudunk választani. Egy lehetőség az ún. **feketedoboz-módszer**, más néven **adatvezérelt tesztelés**. E módszer alkalmazásakor a tesztelő nem veszi figyelembe a program belső szerkezetét, pontosabban nem azt tekinti elsődleges szempontnak, hanem a teszteseteket a feladat meghatározás alapján választja meg.

A cél természetesen a lehető leghatékonyabb tesztelés elvégzése, azaz az összes hiba megtalálása a programban. Ez ugyan elvileg lehetséges, **kimerítő bemenet tesztelést** kell végrehajtani, a programot ki kell próbálni az összes lehetséges bemenő adatra. Ezzel a módszerrel azonban, mint korábban láttuk, mennyiségi akadályba ütközhetünk.

Egy másik lehetőség a **fehérdoboz-módszer**(**logika vezérelt tesztelés**). Ebben a módszerben a tesztesetek megválasztásánál lehetőség van a program belső szerkezetének figyelembevételére is.

A cél a program minél alaposabb tesztelése, erre jó módszer a **kimerítő út tesztelés**. Ez azt jelenti, hogy a programban az összes lehetséges utat végigjárjuk, azaz annyi tesztesetet hozunk létre, hogy ezt elérhessük vele. Az a probléma, hogy még viszonylag kis programok esetén is igen nagy lehet a tesztelési utak száma. Gondoljunk a ciklusokra! Sőt ezzel a módszerrel a hiányzó utakat nem lehet felderíteni.

Mivel sem a fehérdoboz-módszerrel, sem a feketedoboz-módszerrel nem lehetséges a kimerítő tesztelés, el kell fogadnunk, hogy nem tudjuk egyetlen program hibamentességét sem szavatolni. A további cél ezek után az összes lehetséges teszteset halmazából a lehető leghatékonyabb teszteset-csoport kiválasztása lehet.

A tesztelés hatékonyságát kétféle jellemző határozza meg: a *tesztelés költsége*és a *felfedett hibák aránya*. A leghatékonyabb teszteset-csoport tehát minimális költséggel maximális számú hibát fed fel.

**Definíció**: *Próbának* nevezzük a tesztesetek azon halmazát, amellyel a programot teszteléskor kipróbáljuk.

**Definíció**: *Ideális próba* az a próba, amellyel a programban szereplő összes hibajelenség felfedezhető.

**Definíció**: Adott e szinten *megbízható próba*az a próba, amellyel 1–e valószínűséggel felfedjük az összes hibajelenséget.

A megbízhatóság pontos mérése sem oldható meg általában, így a teszteléskor csak megérzéseinkre alapozhatunk.

**2.1. Feketedoboz-módszerek**

**2.1.1. Ekvivalenciaosztályok keresése**

A tesztelés alapelveinek ismertetésénél jó tesztesetnek neveztük az olyant, amelyre minél nagyobb valószínűséggel áll, hogy hibát találunk vele a programban. Másrészt megállapítottuk, hogy a kimerítő bemeneti tesztelés gyakorlatilag megvalósíthatatlan, így meg kell elégednünk a bemenő adatok egy szűk részhalmazának tesztelésével. Ezek után azért, hogy ez a részhalmaz minél hatásosabb legyen, a benne szereplő tesztesetekre teljesüljenek a következők:

* minden tesztesetnek annyi bemeneti feltételt (a bemenő adatokat leíró állítást) kell kielégítenie, amennyit csak lehetséges, hogy ezzel a szükséges tesztesetek számát csökkentsük;
* a bemeneti tartományt valamilyen módon részekre kell osztani, és ezekre a részekre jellemző teszteseteket kell választani. Ezekre a részekre (osztályokra) legyen igaz a következő: ha egy ilyen osztályból választunk egy tesztesetet, és ezzel hibát találunk a programban, akkor az osztály más elemét választva is nagy valószínűséggel hibát találnánk a programban. Hasonlóan: ha a kiválasztott tesztesetre a program jól működik, az osztály másik elemét választva is nagy valószínűséggel helyes eredményt adna.

Ezeket az elveket veszi figyelembe az ekvivalenciaosztályok módszere. Ekvivalenciaosztályokat nemcsak az érvényes, hanem az érvénytelen adatokhoz is létre kell hozni, és a programot azokkal is kipróbálni.

Néhány jó tanács az ekvivalenciaosztályok megtalálásához:

* Ha a bemeneti feltétel értéktartományt definiál, az érvényes ekvivalenciaosztály legyen a megengedett bemenő értékek halmaza, az érvénytelen ekvivalenciaosztályok pedig az alsó és a felső határoló tartomány. Például ha az adatok osztályzatok (értékük 1 és 5 között van), akkor ezek az ekvivalenciaosztályok rendre: 1 ≤ i ≤ 5 és i < 1 és i > 5.
* Ha a bemeneti feltétel értékek számát határozza meg, akkor az előzőhöz hasonlóan járjunk el. Például ha be kell olvassunk legfeljebb 6 karaktert, akkor az érvényes ekvivalenciaosztály: 0–6 karakter beolvasása, az érvénytelen ekvivalenciaosztály: 6-nál több karakter beolvasása. (0-nál kevesebb nem fordulhat elő.)
* Ha a bemenet feltétele azt mondja ki, hogy a bemenő adatnak valamilyen meghatározott jellemzője legyen, akkor két ekvivalenciaosztályt kell felvenni: egy érvényeset és egy érvénytelent.
* Ha okunk van feltételezni, hogy a program valamelyik ekvivalenciaosztályba eső elemeket különféleképpen kezeli, akkor a feltételezésnek megfelelően bontsuk az ekvivalenciaosztályt további osztályokra.

**Példa**

Számoljuk meg egy maximum 40 karakteres szöveg magánhangzóit!

Az ekvivalenciaosztályok:

* 0 hosszúságú szöveg;
* >40 karakterből álló szöveg;
* 1..40 karakterből álló szöveg, nincs benne magánhangzó;
* 1..40 karakterből álló szöveg, van benne magánhangzó.

Ez utóbbi osztályt esetleg újabb osztályokra bonthatjuk:

* a szövegben van A betű;
* a szövegben van Á betű;
* ...

Lehetséges azonban egy másik osztályokra bontás is:

* a szöveg magánhangzói nagybetűk;
* a szöveg magánhangzói kisbetűk.

Ha már rendelkezésünkre állnak az ekvivalenciaosztályok, akkor a teszteseteket a következő két elv alapján határozhatjuk meg:

* amíg az érvényes ekvivalenciaosztályokat le nem fedtük, addig készítsünk olyan teszteseteket, amelyek minél több érvényes ekvivalenciaosztályt lefednek;
* minden érvénytelen ekvivalenciaosztályra írjunk egy-egy, az osztályt lefedő tesztesetet! Több hiba esetén ugyanis előfordulhat, hogy a hibás adatok lefedik egymást, a második hiba kijelzésére az első hibajelzés miatt már nem kerül sor.

**2.1.2. Határeset-elemzés**

A határeset-elemzés két dologban különbözik az ekvivalenciaosztályok keresésének módszerétől; annak bizonyos szempontból kiegészítő módszere:

* az ekvivalenciaosztály kiválasztott elemének a határon levő elemeket választja;
* nemcsak a bemeneti, hanem a kimeneti ekvivalenciaosztályt is figyelembe veszi.

Felsorolunk néhány szempontot a határeset-elemzés megvalósításának elősegítéséhez:

* Ha a bemeneti feltétel egy értéktartományt jelöl meg, írjunk teszteseteket az érvényes tartomány alsó és felső határára és az érvénytelen tartománynak a határ közelébe eső elemére! Például: ha a bemeneti tartomány a (0,1) nyílt intervallum, akkor a 0, 1, 0.01, 0.99 értékekre érdemes kipróbálni a programot.
* Ha egy bemeneti feltétel értékek számosságát adja meg, akkor hasonlóan járjunk el, mint az előző esetben. Például: ha rendeznünk kell 1–128 nevet, akkor célszerű a programot kipróbálni 0, 1, 128, 129 névvel.
* Használjuk a fenti feltételeket az összes kimeneti osztályra is!

**Példa**

Számoljuk meg egy maximum 40 karakteres szöveg magánhangzóit!

Bemenet szerint három próba kell: 0, 40, 41 karakteres szöveg.

Kimenet szerint két próba kell: nincs magánhangzó / minden betű magánhangzó.

**2.2. Fehérdoboz-módszerek**

Az ilyen a módszereknél a tesztelést a programszöveg ismeretében végezzük. A tesztelés három lépés egymásutánjából áll:

* egy kipróbálási stratégiát választunk a program szerkezete alapján;
* a stratégia alapján megadott tesztutakhoz tesztpredikátumokat rendelünk;
* a tesztpredikátumok ekvivalenciaosztályokat jelölnek ki, amelyekből egy-egy tesztesetet választunk.

**2.2.1. Kipróbálási stratégiák**

Itt azt kell meghatároznunk, hogy a programgráfban szereplő mely utak mentén kell kipróbálni a programot.

***Utasítások egyszeri lefedésének elve***

A módszer neve arra utal, hogy a programgráf csomópontjait kell lefedni. A módszer lényege olyan tesztesetek kiválasztása, amelyek alapján minden utasítást legalább egyszer végrehajthatunk a programban. Bár ez sokszor jó módszer, de nem tökéletes; nézzük meg ugyanis a következő egyszerű példát: ***Ha*** *X>0* ***akkor******Ki****: X*

Ebben a példában egyetlen próbával elérhetjük az összes utasítás végrehajtását (pl. X=1), de nem derülne ki az, ha az X>0 feltétel helyett az X≥0 szerepelne, azaz a program hibás lenne.

***Döntéslefedés elve***

A módszer neve arra utal, hogy a programgráf éleit kell lefedni. Itt az előzőnél egy kicsit erősebb követelményt alkalmazunk. A programban minden egyes elágazás igaz, illetve hamis ágát legalább egyszer be kell járni a tesztelés során. A döntéslefedés elvét figyelembe véve eleget teszünk az utasításlefedés követelményének is.

**Példa**

Itt is maradnak azonban problémák, nézzünk egy példát: ***Ha*** *X>0* ***és*** *Y>0* ***akkor Ki****: X\*Y*

Ebben az esetben az (X=1, Y=1) és az (X=-1, Y=-1) tesztesetek lefedik a döntéseket, de nem vennénk észre velük azt, ha a második feltételt (Y>0) rosszul írtuk (vagy lehagytuk) volna.

***A részfeltétel-lefedés elve***

Ebben az esetben olyan teszteseteket kell készíteni, amelyhez a döntésekben szereplő minden részfeltételt legalább egyszer HAMIS, illetve IGAZ eredménnyel értékelünk ki.

**Példa**

Az előbbi példát eszerint ki kell próbálni a következő tesztesetekkel: (X=1,Y=1), (X=-1, Y=-1), (X=1, Y=-1)

***A speciális tesztesetek elve***

Az előbbi elvek mellett – a feketedoboz-módszereknél ismertetett ötletek alapján –válasszunk az egyes utakhoz speciális teszteseteket is!

**Példa**

Az előző példát még próbáljuk ki például az (X=0, Y=-1) és az (X=1, Y=0) tesztesetekkel is!

**2.2.2. Teszteset-generáló módszerek**

Hogyan lehet eldönteni, hogy mely adatokkal próbáljuk ki a programot ahhoz, hogy a választott kipróbálási stratégiát jól alkalmazzuk?

A legegyszerűbb eset az, hogy véletlenszerűen választunk adatokat, és a program végrehajtása során vizsgáljuk, hogy mely utakon jártunk. A pontos leíráshoz egy-két definícióra van szükségünk, amelyek a program gráffal való leírására épülnek.

**Definíció**: *Bázisútnak* nevezzük a programgráf olyan útját, amely

* a kezdőponttól a legelső elágazás- vagy ciklusfeltétel kiértékeléséig tart;
* elágazás- vagy ciklusfeltételtől a következő elágazás- vagy ciklusfeltétel helyéig vezet;
* elágazás- vagy ciklusfeltételtől a program végéig tart, s közben más feltételkiértékelés nincs.

**Megjegyzés**

E definíció az élek egyszeres lefedéséhez illeszkedik, a csomópontok egyszeres lefedéséhez el kell hagyni az olyan bázisutakat, amelyek részei más bázisútnak.

**Definíció**: A *vezérél* a bázisút első éle.

Most már megoldhatjuk az előbb felvetett problémát. Helyezzünk el a program minden vezérélén egy-egy számlálót! Próbáljuk ki a programot újabb és újabb tesztadatokkal mindaddig, amíg mindegyik számláló értéke legalább 1 nem lesz!

Alapvető probléma, hogy ez a módszer nagyon gazdaságtalan, hiszen semmilyen segítséget nem ad a tesztadatok választásához, és így esetleg csak nagyon sok kipróbálással tudjuk megvalósítani a kipróbálási stratégiát. A jobb megoldáshoz újabb definíciókat nézzünk.

**Definíció**: *Tesztutaknak* nevezzük a programgráfon átvezető, a kezdőponttól a végpontig haladó olyan utakat, amelyek minden bennük szereplő élt pontosan egyszer tartalmaznak.

**Definíció**: *Tesztpredikátumnak* nevezzük azokat a bemenő adatokra vonatkozó feltételeket, amelyek teljesülése esetén pontosan egy tesztúton kell végighaladni.

A teszteset-generálás első lépése tehát a minimális számú olyan tesztút meghatározása, amelyek lefedik a kipróbálási stratégiának megfelelően a programgráfot.

Ha ezzel készen vagyunk, akkor határozzuk meg ezen tesztutak tesztpredikátumait! Ehhez a program **szimbolikus végrehajtására** van szükség. Induljunk ki az előfeltételből! Haladjunk a programban az első elágazás- vagy ciklusfeltételig, és a formulát a közbülső műveleteknek megfelelően transzformáljuk! A tesztútnak megfelelő ág feltételét **és**kapcsolattal kapcsoljuk hozzá a tesztpredikátumhoz, majd folytassuk a szimbolikus végrehajtást egészen a program végpontjáig!

Probléma lehet ezzel az, hogy lehetetlen tesztutat választottunk, azaz a kapott tesztpredikátum azonosan hamis lesz.

Harmadik lépésként minden egyes tesztpredikátumhoz (amelyek egy-egy ekvivalenciaosztályt reprezentálnak) válasszunk egy-egy tesztesetet!

**Példa**

Egy osztály tanulói átlagának alapján a jelesek számának meghatározása.

|  |  |
| --- | --- |
| **Pszeudokódos algoritmus** | **Struktogram** |
| ***Program****:*  ***Be****: N,ÁT*  *DB:=0; I:=1*  ***Ciklus amíg*** *I≤N ← 1*  ***Ha*** *ÁT[I]>4.5* ***akkor*** *DB:=DB+1 ← 2*  *I:=I+1*  ***Ciklus vége***  ***Ki****: DB*  ***Program vége****.* | A mellékelt program algoritmusa, struktogrammal. |

A programban két feltételkiértékelés szerepel. A programon átvezető tesztutak:

* A. a ciklusfeltétel eleve hamis, a ciklusba nem lépünk be,
* B. a ciklusfeltétel először igaz, a ciklusba belépve az elágazásfeltétel igaz, s utána a ciklusfeltétel hamissá válik,
* C. a ciklusfeltétel először igaz, a ciklusba belépve az elágazásfeltétel hamis, s utána a ciklusfeltétel hamissá válik,

E három útból minimálisként az utolsó kettőt választjuk, az első mindegyiküknek része.

Az előfeltétel: *N≥0 és ∀j(1≤j≤N): 1≤ÁT[j]≤5*

Mindkét tesztútnál el kell jutni először a ciklusig, emiatt tesztpredikátumuk így alakul:

B-C. *N≥0 és ∀j(1≤j≤N): 1≤ÁT[j]≤5 és DB=0 és I=1*

Vegyük most figyelembe, hogy a ciklusfeltételnek először igaznak kell lennie:

B-C. *N≥0 és ∀j(1≤j≤N): 1≤ÁT[j]≤5 és DB=0 és I=1 és I≤N*

Ebből következik:

B-C. *N≥ 1 és ∀j(1≤j≤N): 1≤ÁT[j]≤5 és DB=0 és I=1*

Következik az elágazásfeltétel, amelyet a két tesztútnál kétféleképpen kell figyelembe venni:

B. *N≥1 és ∀j(1≤j≤N): 1≤ÁT[j]≤5 és DB=0 és I=1 és ÁT[I]>4.5*

C. *N≥1 és ∀j(1≤j≤N): 1≤ÁT[j]≤5 és DB=0 és I=1 és ÁT[I]≤4.5*

Az első esetben végre kell hajtani az elágazásban szereplő növelést, a másodikban nem, utánuk a tesztpredikátum egy egyszerű átalakítással így alakul:

B. *N≥1 és ∀j(1≤j≤N): 1≤ÁT[j]≤5 és DB=1 és I=1 és ÁT[1]>4.5*

C. *N≥1 és ∀j(1≤j≤N): 1≤ÁT[j]≤5 és DB=0 és I=1 és ÁT[1]≤4.5*

A ciklusváltozó növelése után a ciklusfeltételnek már nem szabad teljesülnie:

B. *N≥1 és ∀j(1≤j≤N): 1≤ÁT[j]≤5 és DB=1 és I=2 és ÁT[1]>4.5 és I>N*

C. *N≥1 és ∀j(1≤j≤N): 1≤ÁT[j]≤5 és DB=0 és I=2 és ÁT[1]≤4.5 és I>N*

Ebből újabb transzformációval kapjuk a végleges tesztpredikátumokat (a bemenetben nem szereplő változók elhagyásával):

B. *N=1 és ∀j(1≤j≤N): 1≤ÁT[j]≤5 és ÁT[1]>4.5*

C. *N=1 és ∀j(1≤j≤N): 1≤ÁT[j]≤5 és ÁT[1]≤4.5*

Maradt még a konkrét tesztesetek meghatározása:

B. *(N=1,ÁT[1]=5)*

C. *(N=1,ÁT[1]=4)*

**2.3. Speciális tesztek**

Van a tesztelésnek néhány speciális változata, amikor nem a teljes (vagy parciális helyesség) belátása a cél. Vizsgáljuk sorra ezeket!

**2.3.1. Elfogadhatósági teszt**

Annak vizsgálata, hogy a specifikáció ellentmondásmentes-e, tartalmaz-e hiányosságokat, ...

**2.3.2. Funkcióteszt**

Szerepel-e a programban az összes megvalósítandó funkció? Menürendszerű programban a megjelenő, választható funkciók megvannak-e? A kívánt eredmények mindegyike megjelenik-e? ...

**2.3.3. Biztonsági teszt**

A program ellenőrzi-e a felhasználótól kapott adatokat? Hibás adatokkal meg lehet-e zavarni működését? Ellenőrzi-e a szükséges perifériák, file-ok létét, típusát, a file-ok elhelyezkedését a háttértáron?

**2.3.4. Stressz teszt**

A program számára az adatok nagyon nagy sebességgel érkeznek, egyszerre többfelől is érkeznek, ...: mi történik ilyenkor? Megfelelő hatékonysággal működik-e ekkor a program?

**2.3.5. Volumen teszt**

A programnak nagyon sok adatot kell kezelnie (az előfeltételben megadott maximális számút). Helyesen működik-e ekkor is a program? Megfelelő hatékonysággal működik-e ekkor a program?

**2.3.6. Hatékonysági teszt**

A program végrehajtási idejének (átlagos, minimális, maximális), valamint változói helyfoglalásának meghatározása tesztesetekkel, a javítás és az eredeti összevetése e szempontból.

**2.3.7. Modulteszt**

Önálló modulokból álló programrendszer moduljainak egyenkénti tesztelése, a többi modultól függetlenül.

**2.3.8. Összeépítési teszt**

Önálló modulokból álló programrendszer moduljai összeépítésének ellenőrzése.