Írta:
BESZÉDES ÁRPÁD, GERGELY TAMÁS

TESZTELÉSI MÓDSZEREK

Egyetemi tananyag

TYPOTEX

2011

LEKTORÁLTA: Dr. Kovács Attila, ELTE Informatikai Kar Komputeralgebra Tanszék

TÁMOGATÁS:

Készült a TÁMOP-4.1.2-08/1/A-2009-0008 számú, „Tananyagfejlesztés mérnök informatikus, programtervező informatikus és gazdaságinformatikus képzésekhez” című projekt keretében.

ISBN 978-963-279-501-0

KÉSZÜLT: a Typotex Kiadó gondozásában

FELELŐS VEZETŐ: Votisky Zsuzsa

AZ ELEKTRONIKUS KIADÁST ELŐKÉSZÍTETTE: Dudás Kata

KULCSSZAVAK:

tesztelés (testing), kód alapú tesztelés (whitebox testing), specifikáció alapú tesztelés (blackbox testing), teszteset priorizáció (test case prioritization), teszteset szelekció (test case selection), hibakeresés (debugging), szeletelés (slicing), statikus analízis (static analysis), formális módszerek (formal methods).

ÖSSZEFOGLALÁS:

A szoftvertesztelés, mint szakma rendkívül sokrétű, különböző képességeket és képesítéseket igényel. A szükséges ismeretanyag a tesztelés minden vonatkozására kiterjed úgymint, alapelvek, módszertanok, technikák, módszerek, szabványok, eszközök, menedzsment. A Szegedi Tudományegyetem Szoftverfejlesztés Tanszéke tudatos módon építi a teszteléssel speciálisan foglalkozó oktatási kínálatát. Ennek részeként, a Tesztelési Módszerek tárgy a teszt-tervezési és -végrehajtási módszerekre fekteti a hangsúlyt. Tartalmi szempontból, a jegyzet a legfontosabb tesztelési módszereket ismerteti, főleg technikai szempontból. A tesztelési módszereket sokféleképpen tudjuk csoportosítani, ezek áttekintése után egy lehetséges csoportosítás szerint, megadjuk a módszerek alap megközelítéseit, technikákat, algoritmusokat, a gyakorlati alkalmazás lehetőségeit. A jegyzet a kód és specifikáció alapú módszerekkel, majd harmadik csoportban az egyéb módszerekkel foglalkozik. Az utolsó fejezetben foglalkozik a hibaeredet-keresés és hibaeltávolítás „debugging” témájával is.

Tartalomjegyzék

1. Bevezetés

A szoftvertesztelés, mint szakma rendkívül sokrétű. Különböző képességeket és képesítéseket igényel, melyek elsajátítása gyakorlattal, illetve összetett képzéssel lehetséges. Az ISTQB [1] képzési séma három szintben határozza meg az elvárt ismerethalmazt: Alap, Haladó és Szakértő, továbbá az elvárt ismeretanyag a tesztelés minden vonatkozására kiterjed úgymint, alapelvek, módszertanok, technikák, módszerek, szabványok, eszközök, menedzsment. Mindezen ismeretek együttes megléte egy szakembernél jelenthet csak igazi garanciát a szakma magas szintű művelésére. Természetesen a különböző szerepkörökben dolgozó tesztelőknek más és más tudást kell kidomborítaniuk, például egy teszt vezetőnek a menedzsment ismereteket, míg a specifikáció alapú tesztelést folytató szakembernek a teszt-tervezési módszereket.

A nemzetközi ajánlásokat és gyakorlatot követve, a Szegedi Tudományegyetem Szoftverfejlesztés Tanszéke tudatos módon építi az szoftverminőséggel általánosan, és a teszteléssel speciálisan foglalkozó oktatási kínálatát. Ennek részeként, a Tesztelési Módszerek tárgy a fent említett vonatkozások közül egyre, a teszt-tervezési és -végrehajtási módszerekre fekteti a hangsúlyt. A tárgy elsajátításához feltételezzük a tesztelés alapismereteinek ismeretét, kifejezetten a Tanszék által oktatott Szoftvertesztelés Alapjai kurzus sikeres elvégzését. A Tesztelési Módszerek szakirányos mesterszakos tárgy, ennek megfelelően a szakterület mély vizsgálata, és nem csak alapszintű tárgyalása, olykor különleges, a hétköznapi gyakorlatban ritka körülmények között alkalmazott módszerek ismertetése a célja. A szoftvertesztelés alapjain kívül elvárjuk a számítástudományban és a szoftverfejlesztésben mint mérnöki, ipari ágazatban alkalmazott magas szintű általános ismereteket is, amelyekre az egyes módszereknél építünk.

Tartalmi szempontból, a tárgy – és ennek megfelelően a jelen jegyzet is – a legfontosabb tesztelési módszereket ismerteti, főleg technikai szempontból, azaz csak érintőlegesen foglalkozunk a módszerek alkalmazásának folyamatbeli, szervezési és egyéb kérdéseivel. Egy lehetséges csoportosítás szerint, megadjuk a módszerek alap megközelítéseit, technikákat, algoritmusokat. Általában a módszerek leírását először elméleti oldaláról közelítjük meg, majd megadjuk a gyakorlati alkalmazás lehetőségeit. A legtöbb módszert példákkal illusztráljuk, valamint gyakorló feladatokkal látjuk el.

A tesztelési módszereket sokféleképpen tudjuk csoportosítani: életciklus fázisa szerint, tesztelés szintje szerint, cél szerint, alap megközelítés szerint, stb. Ezek áttekintése a Módszertani megközelítés c. (5.) fejezetben található. A jegyzet további része az egyes módszereket tárgyalja, kezdve a kód alapúakkal (pl. lefedettség-alapú tesztelés), majd a specifikáció alapú módszerek következnek (pl. ekvivalencia partícionálás), míg a harmadik csoportban az egyéb módszerek kaptak helyet (ilyenek a statikus módszerek, a hatásanalízis és a formális módszerek). A jegyzet utolsó (6.) fejezete foglalkozik egy olyan témával, amely nem szigorúan a tesztelési tevékenységek közé sorolt, de rendkívül szoros

[1] Az International Software Testing Qualifications Board (ISTQB) a vezető világszervezet, amely a tesztelést mint szakmát népszerűsíti, és definiálja az ezen a területen dolgozó szakemberektől elvárt szakmai tudást. Magyarországi képviselete a Magyar Szoftvertesztelői Tanács Egyesület (HTB – Hungarian Testing Board).

kapcsolatban áll azzal, ami a hibaeredet-keresés és hibaeltávolítás „debugging” (a tesztelés hatásköre általában a hibák jelenlétének kimutatásáig és a rendszer általános minőségi szintjének meghatározásáig terjed).

A tárgy kidolgozásánál törekedtünk a teljességre, ez nyilván nem sikerülhetett maradéktalanul, tekintve a terület nagyságát és mélységét. A legfontosabb módszerek azonban ismertetve lettek, ami jó kiindulási alap lehet azok számára, akik a tesztelési szakmában fognak dolgozni. Bizonyos módszerek teljes részletességgel, azonnali alkalmazhatósággal, míg mások érintőlegesen vannak bemutatva. Természetesen, mint minden szakmában, itt is a gyakorlat teszi a mestert, így a módszerek alkalmazása valós projektekben, valós problémákra fogja igazán megmutatni azok hasznát, esetleges hátrányait. Az ismertetett módszerekre ne, mint elszigetelt kész csomagokra gondoljunk, amiket a „polcról levéve” azonnal alkalmazni tudunk, hanem mint fontos eszközöket, szerszámokat a kezünkben, melyeket megfelelő szaktudással és gyakorlattal sikeresen alkalmazhatunk. Ez utóbbiak a módszerek módszertani alkalmazásának a rejtelmei, ami talán egy következő kurzus témája lehet…

Mielőtt elmerülünk a tesztelés és teszt tervezés rejtelmes világában, szeretnénk köszönetet mondani lelkes és segítőkész kollégáinknak, Gyimóthi Zoltánnak és Hegedűs Dánielnek a tárgy anyagának készítéséhez nyújtott segítségükért, amiből – reméljük – valamit ők maguk is megtanultak, mint ahogy mi is.

Beszédes Árpád és Gergely Tamás, Szeged, 2011. március

2. Kód alapú módszerek

A kód alapú tesztelés (vagy más neveken struktúra alapú tesztelés, fehér doboz tesztelés) számos tesztelési módszert magába foglaló kategória.

A kód alapú tesztelési módszereket gyakran használják olyan esetekben, amikor a megbízhatóság különösen fontos a tesztelés során. Jól alkalmazhatóak a fekete-doboz tesztelési technikák kiegészítéseként, mert szigorúbbak és logikusabban felépítettebbek azoknál. Fő jellemzőik, hogy a forráskódra alapulnak, így pontosabb szabályokkal írhatóak le, mechanikusabbak és pontosabban mérhetők. A fejezet során sokszor fogjuk használni a vezérlési folyam gráf fogalmát, így most elsőként ezt ismertetjük.

2.1. Vezérlési folyam gráf

Ha adott egy imperatív programnyelven írt programkód, akkor az ahhoz tartozó vezérlési folyam gráf (*Control Flow Graph – CFG*) egy olyan irányított gráf, ahol a csomópontok utasításoknak felelnek meg, míg az élek a vezérlés folyamatát jelzik. Az i és j csomópontok között akkor létezik él a gráfban, ha a j csomópont közvetlenül i után végrehajtódhat a program valamely végrehajtása során.

2.1.1. Példa

Készítsünk vezérlési folyam gráfot az alábbi pszeudokódból:

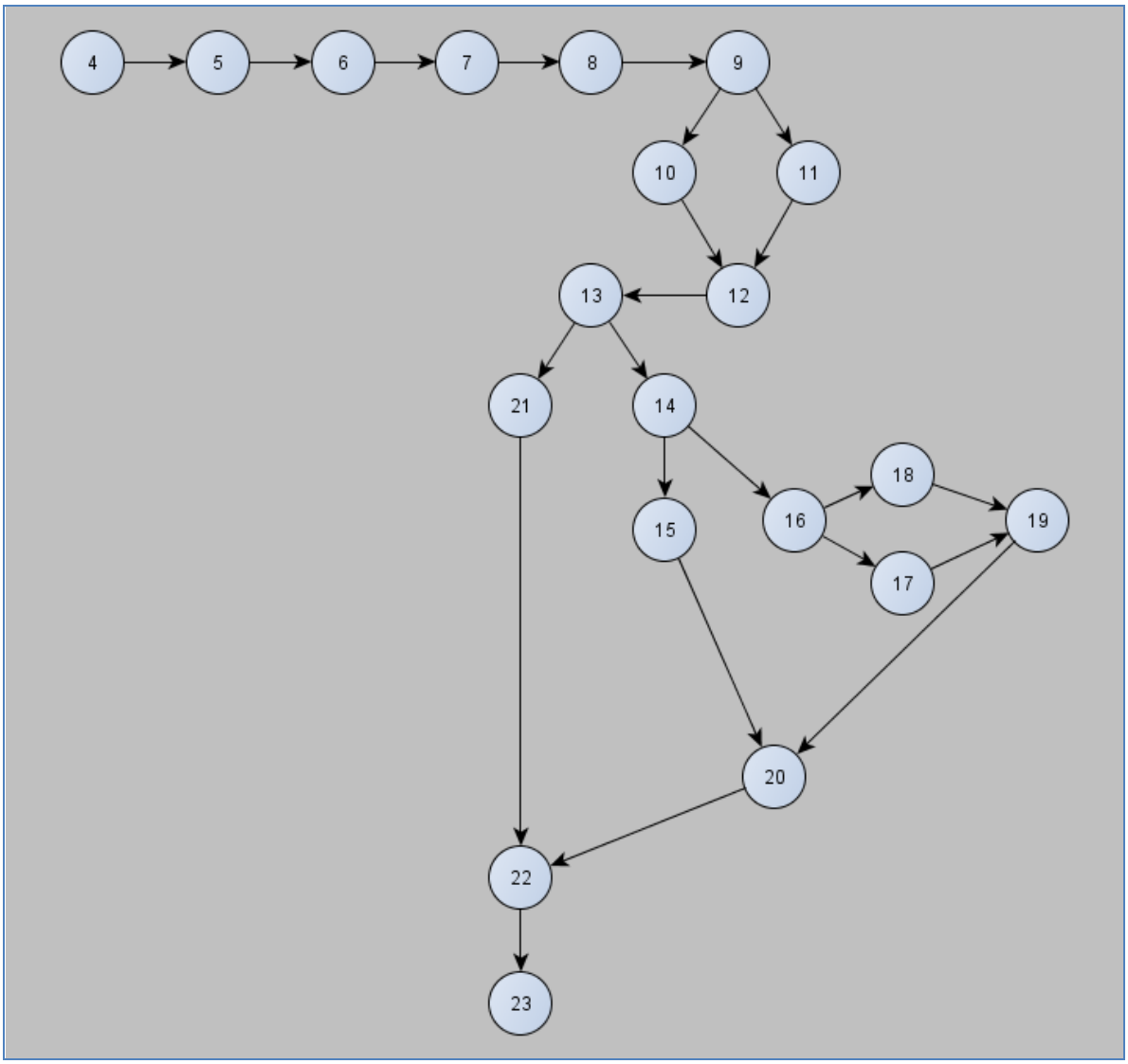
```
1    Program Háromszög
2    Def a,b,c Integer
3    Def háromszög Boolean

4    Kiir(„Adjunk meg 3 értéket, amik a háromszög oldalai”)
5    Beolvas(a,b,c)
6    Kiir(„A oldal: ”, a)
7    Kiir(„B oldal: ”, b)
8    Kiir(„C oldal: ”, c)

9    if (a < b + c) AND (b < a + c) AND (c < a + b)
10        then háromszög = True
11        else háromszög = False
12   endif

13   if (háromszög)
14        then if (a = b) AND (b = c)
15                  then Kiir(„Szabályos”)
16                  else if (a ≠ b) AND (a ≠ c) AND (b ≠ c)
17                            then Kiir („Egyenlőtlen”)
18                            else Kiir („Egyenlő szárú”)
19                  endif
20             endif
21   else Kiir(„Nem háromszög”)
22   endif
23   end Háromszög
```

A fenti programkódból készített vezérlési folyam gráf az alábbiakban látható (1. ábra).



1. ábra: Példa vezérlési folyam gráf

A 4-es és a 23-as jelzésű csomópontok a program kezdetét, illetve végét jelölik. Mivel nincs ciklus a kódban, ezért ez egy körmentes, irányított gráf.

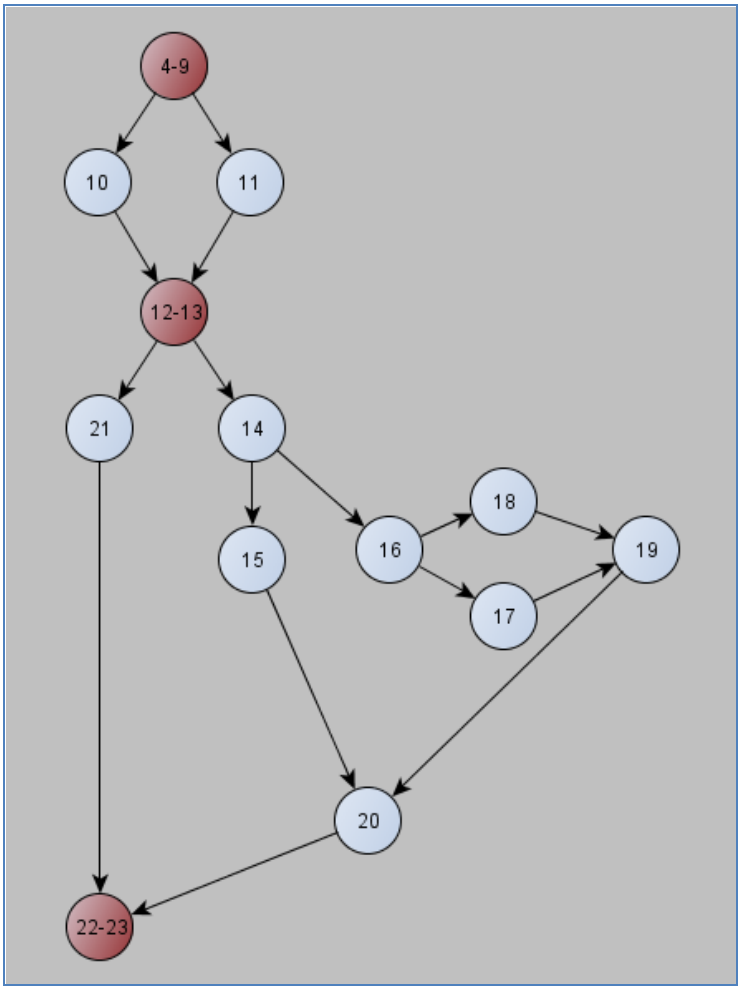
A vezérlési folyam gráf szerepe abban rejlik, hogy a futtatásnál a vezérlés a kezdőpontból („forrás”) valamelyik végpontba („nyelő”) fog eljutni. Mivel egy-egy tesztesettel egy-egy vezérlési útvonalat tesztelünk (kényszerítjük a programot, hogy az az ág hajtódjon végre), és struktúra alapú tesztelés során látjuk is, hogy melyiket, ezért képet kaphatunk arról, hogy egy adott teszteset vagy teszteset halmaz melyik programrészeket érinti.

2.1.2. Alap blokkok

A vezérlési folyam gráf utasítás-szinten történő ábrázolásánál van egy kifinomultabb megoldás, ez pedig az ún. alap blokkok (basic block) alkalmazása.

Informálisan, alap blokknak nevezzük az olyan egymás után következő utasításokat, melyekre teljesül az a feltétel, hogy ha a blokkban lévő első utasítás végrehajtásra kerül, akkor a blokk utolsó utasítása is végre fog hajtódni, és semelyik – a blokkban lévő – utasítás nem kaphatja meg a vezérlést máshonnan, csak egy olyan korábbi utasítástól, ami a blokkban volt. (Ez alól egyedül a blokk kezdőcsúcsa jelenthet kivételt.)

Tekintsük példaként a fent bemutatott vezérlési folyam gráfot. Látható, hogy a 4-es csúcstól a 9-es csúcsig a vezérlés csak egyféleképpen mehet, vagyis, ha a 4-es pont megkapja a vezérlést, akkor biztos, hogy a 9-es pont is meg fogja kapni. Így tehát a „4-5-6-7-8-9” csúcssorozat egy alap blokkot alkot. Hasonlóképpen látható, hogy a „12-13”-as, valamint a „22-23”-as csúcspárok is alap blokkot alkotnak. A többi csúcspont egymagukban alkotnak alap blokkot. Ezek után felrajzolhatjuk a leegyszerűsített ábránkat (2. ábra - Pirossal jelöltük az újonnan létrehozott alap blokkokat).



2. ábra: Példa vezérlési folyam gráf – alap blokkokkal

Az alap blokkok definiálása után most bevezetjük a teszt-lefedettség fogalmát.

2.1.3. Teszt-lefedettség

A tesztelés különböző területein alkalmazzák a lefedettség fogalmát a tesztelés teljességének ellenőrzésére. Ez jelentheti például a feldolgozott követelmények arányát, az előkészített vagy futtatott tesztesetek számát, a tesztelt adatok számát, stb. A lefedettség mérésnek fontos szerepe van a struktúra alapú módszereknél, ahol azt a teszt végrehajtása során érintett programkód elemek arányával értelmezzük a teljes (megváltozott) elemek számához képest.

A *teszt-lefedettség* azt mutatja meg, hogy az implementált, és lefuttatott tesztesetekkel a kód hány százalékát érintettük (teszteltük). Még egyszerűbben: azt vizsgáljuk, hogy a tesztesetek tényleg letesztelik-e a kódot?

A teszt-lefedettséget több szinten is definiálhatjuk. A leggyakrabban használt struktúra alapú lefedettségek az *utasítás*-, *branch*-, *döntési*- és *útvonal-lefedettségek*. A dokumentum további részében ezekről, és ezek használatáról lesz szó.

2.2. Utasítás/Alap blokk tesztelés

Az utasítás-lefedettség (statement coverage) azt mutatja meg, hogy egy adott kódrészletben az utasítások hány százalékát érintettük a teszteset-futtatások során. Szokás „line coverage”-ként is hivatkozni erre a típusú lefedettségre, bár ez valamivel lazább, hiszen nem veszi figyelembe a szintaxist, hanem csak lexikális szinten dolgozik.

Más szóval ez a metrika azt mutatja meg, hogy vajon minden utasítás futtatásra kerül-e? A vezérlési utasítások (`if, for, switch`) akkor tekinthetőek lefedettnek, ha a vezérlést irányító utasítás, valamint a vezérlésben lévő utasítások is lefedésre kerülnek.

Bár ezt a fajta lefedettséget az egyik legkönnyebb implementálni (lásd később), megvannak a maga hátrányai:

- Nem vesz figyelembe bizonyos vezérlési struktúrákat
- Nem tudja ellenőrizni, hogy egy ciklus eléri-e a végfeltételét.
- Nem veszi figyelembe a logikai operátorokat `(|| , &&)`

Tekintsük például az alábbi kódrészletet:

```
1  public String statementCoverageMinta(boolean condition) {
2       String foo = null;
3       if (condition) {
4            foo = "" + condition;
5       }
6       return foo.trim();
7  }
```

Ha mindig `true` értékű paraméterrel hívjuk meg a függvényt, akkor az utasítás-lefedettség 100% lesz. Ennek ellenére egy fontos futásidejű hiba észrevétlen marad. (Mégpedig az, hogy `false` értékkel meghívva a 6. sorban a nincs objektumunk.)

Az utasításlefedettség egy kis módosítása az ún. alap blokk lefedettség (basic block coverage), ahol is a minden esetben együtt végrehajtott utasítás sorozatokat a bennük szereplő utasítások számától függetlenül egy-egy elemnek tekintjük. Vagyis az utasítások helyett az alapblokkokat számoljuk. Ez akkor hasznos, ha az egyik feltétel ág (mondjuk egy

„`if`” vezérlési szerkezet egyik ága) jóval több utasítást tartalmaz, mint a másik. Ez esetben az utasítás-lefedettség nem mutatna valós értéket, ha csak az egyik ágat tesztelnénk. (Vagy nagyon alacsony, vagy nagyon magas értéket mutatna).

Az alap blokk lefedettségre látható egy példa az alábbiakban:

```
1   public void bigBranchMinta(boolean feltetel) throws ApplicationException {
1                   if (feltetel) {
2                         System.out.println("Kis ág #1");
3                         throw new ApplicationException("Érinteni kell!");
4                   } else {
5                         System.out.println("Nagy ág #1");
6                         System.out.println("Nagy ág #2");
                          ...
102                       System.out.println("Nagy ág #98");
103                  }
104 }
```

Ha a fent bemutatott metódust csak `false` értékkel hívjuk meg, akkor az utasítás-lefedettség 98%-os. Ez azonban túlzó lehet, ha a kisebb ágban egy fontos kódrészlet található. Ezért érdemes az alap blokk lefedettséget számolni, ami jelen esetben 50%.

2.3. Branch/Döntési tesztelés

A branch és a döntési tesztelés szorosan összekapcsolódnak (Valójában gyakran szinonimaként szerepelnek). Bizonyos esetekben (olyan komponensek, melyeknek egy belépési pontjuk van) 100%-os branch lefedettség garantálja a 100%-os döntési lefedettséget is, de vannak olyan esetek, amikor a két lefedettség nem egyezik meg.

Branch lefedettség esetén azt vizsgáljuk, hogy a vezérlési folyam gráfban lévő élek hány százalékát fedtük le tesztesetekkel, míg a döntési lefedettségnél azt nézzük, hogy az olyan vezérlési szerkezeteknél, ahol több branch fele is ágazhat a vezérlés, mennyi elágazást (döntést) fedtünk le a tesztesetekkel.

A kétféle lefedettség közötti különbséget egy példán keresztül demonstráljuk:

> *Legyen a következő komponens azért felelős, hogy eldöntse egy adott szó helyét egy ABC- sorrendbe rendezett szótárban (táblában). A komponensnek meg kell kapnia azt is, hogy hány szót kell végignéznie a táblában. Ha találat van, akkor a szó pozícióját kell visszaadni (0-tól kezdődően), egyébként pedig „-1”-et.*

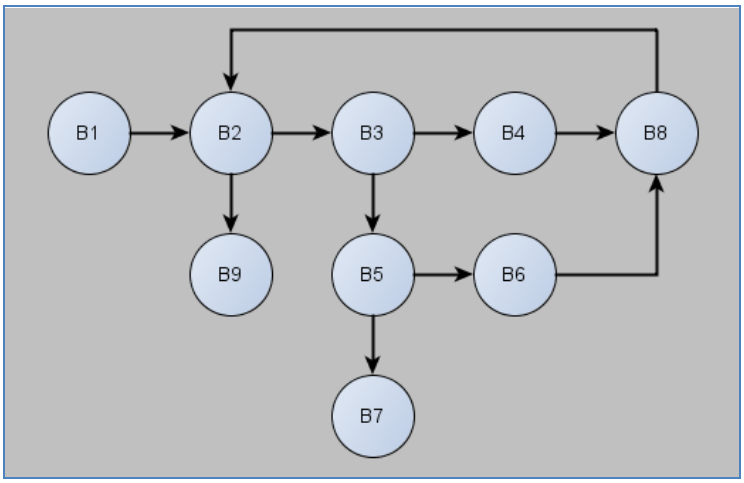
A fenti komponenst megvalósító kódrészlet itt látható (**félkövérrel** kiemeltük a döntéseket):

```
int binsearch (char *word, struct key tab[], int n) {
     int cond;
     int low, high, mid;
     low = 0;
     high = n - 1;
     while (low <= high) {
          mid = (low+high) / 2;
          if ((cond = strcmp(word, tab[mid].word)) < 0)
                    high = mid - 1;
          else if (cond > 0)
                    low = mid + 1;
               else
                    return mid;
     }
     return -1;
}
```

A korábbiakban megismert módszer szerint bejelöljük az alap blokkokat, és felrajzoljuk a vezérlési gráfot (3. ábra):

```
int binsearch (char *word, struct key tab[], int n) {
     int cond;
     int low, high, mid;

B1   low = 0;
     high = n - 1;
B2   while (low <= high){
B3        mid = (low+high) / 2
          if ((cond = strcmp(word, tab[mid].word)) < 0)
B4             high = mid - 1;
B5        else if (cond > 0)
B6             low = mid + 1;
B7        else
               return mid;
B8   }
B9   return -1;
}
```



3. ábra: A példa kód vezérlési folyam gráfja

A vezérlési gráfot felírhatjuk mátrix-formában:

	B1	B2	B3	B4	B5	B6	B7	B8	B9
B1	0	**1**	0	0	0	0	0	0	0
B2	0	0	**1**	0	0	0	0	0	**1**
B3	0	0	0	**1**	**1**	0	0	0	0
B4	0	0	0	0	0	0	0	**1**	0
B5	0	0	0	0	0	**1**	**1**	0	0
B6	0	0	0	0	0	0	0	**1**	0
B7	0	0	0	0	0	0	0	0	0
B8	0	**1**	0	0	0	0	0	0	0
B9	0	0	0	0	0	0	0	0	0

Látható, hogy a komponensnek egyetlen belépési pontja van (B1, mert B1 oszlopában nincs 1-es, ez azt jelenti, hogy B1 befoka 0), és két kilépési pontja (B7, B9 - , mert ezek kifoka 0, vagyis a táblázatban az őhozzájuk tartozó sorban nincsen 1-es). Az is megfigyelhető, hogy minden döntésnek kétféle kimenetele van, és mivel 3 döntés található a kódban (azok a pontok a gráfban, aminek egynél több kimenő éle van), ezért ez összesen 6-féle döntési útvonalat jelent. A branch-ek száma pedig 10 (mivel ennyi él van a gráfban).

Vegyük a következő két tesztesetet:

1. A keresési tábla üres
2. A keresett szó a tábla 2. negyedében van

Nézzük végig, hogy az egyes tesztesetek mekkora branch-, illetve döntési-lefedettséget biztosítanak.

Az első esetben az alábbi útvonalat követi a vezérlés (**félkövérrel** jelöltük, ha az adott ponton egy döntési helyzetbe kerülünk):

{B1 → **B2** → B9}

- *Döntési lefedettség*: 1/6 – mivel egyetlen egy döntést érintettünk a 6-ból
- *Branch-lefedettség*: 1/5 – mivel 2 branch-et érintettünk 10-ből

A második esetben a vezérlési folyam:

{B1 → **B2** → **B3** → B4 → B8 → **B2** → **B3** → **B5** → B6 → B8 → **B2** → **B3** → **B5** → B7}

- *Döntési lefedettség*: 5/6 – mivel 5-féle döntést érintettünk a 6-ból
- *Branch-lefedettség*: 9/10 – mivel 9 branch-et érintettünk 10-ből

A két teszteset együtt 100%-os branch-, és döntési lefedettséget biztosít. A két teszteset táblázatos formában alább látható (**félkövéren** és **aláhúzva** a döntések láthatóak):

teszteset	inputok			útvonal	elvárt kimenet
	keresett szó	tábla	n		
1	chas	'üres tábla'	0	B1 → **B2** → B9	-1
2	chas	alf bert chas dirty eddy fred geoff	7	B1 → **B2** → **B3** → B4 → B8 → **B2** → **B3** → **B5** → B6 → B8 → **B2** → **B3** → **B5** → B7	2

2.4. Útvonal tesztelés

Az útvonal lefedettség azt mutatja meg, hogy vajon a program összes lehetséges végrehajtási útvonalát (összes branch sorozatát, a program vezérlési folyam gráfjának összes sétáját) leteszteltük-e. Előnye, hogy alapos tesztelést tesz lehetővé.

Gyakorlatban 100%-os útvonal-lefedettséget elérni lehetetlen, főként azért, mert a lehetséges útvonalak száma sokszor exponenciálisan nő a branch-ek számával. Sőt, szinte minden valós programban található ciklus, aminek jelenlétében a lehetséges útvonalak száma valószínűleg végtelen lesz (egy ciklusról ugyanis általában nem határozható meg a végrehajtási lépések száma, ami következik a Turing-féle megállási problémából). Továbbá előfordulhatnak olyan útvonalak is, amelyeket nem tudunk tesztesetekkel előidézni. Az utóbbira láthatunk egy példát az alábbiakban:

```
1  public void pathCoverageMinta(boolean foo) {
2  if (foo) {
3         System.out.println(„A1 path”);
4  } // az else-ág lenne az A2-es path
5
6  if (foo) {
7         System.out.println(„B1 path”);
8  } // az else-ág lenne a B2-es path
9  }
```

Ha megnézzük a példakódot, akkor láthatjuk, hogy elvileg 4 lehetséges útvonal van (`A1-B1; A1-B2; A2-B1; A2-B2`), azonban könnyen belátható, hogy ebből a 4-ből, csak 2 útvonalat tudunk letesztelni (`A1-B1; A2-B2`), a másik két ágat sosem tudjuk elérni.

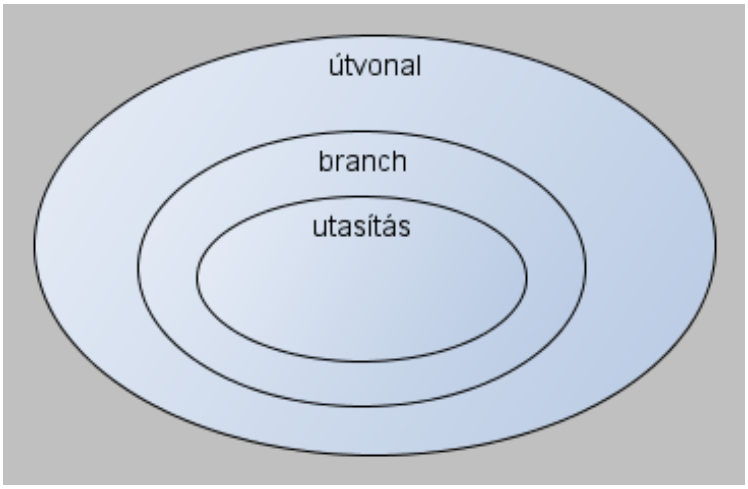
Éppen az ilyen jellegű hátrányok miatt, az útvonal lefedettségnek többféle változata is létezik. Az összes útvonal tesztelése helyett válasszunk ki az útvonalak közül néhányat, és az így keletkező alaphalmazhoz válasszunk teszteseteket. A kiválasztás történhet egyszerűen az útvonalak hosszának korlátozásával (legfeljebb n hosszúságú útvonalakat vizsgálunk), előzetes futási információk felhasználásával (leggyakoribb részek lefedése), vagy a ciklomatikus komplexitás alapján (lineárisan független útvonalak).

2.5. Módszerek összehasonlítása

Vajon az itt ismertetett módszerek közül melyik vizsgálja meg a legrészletesebben a kódot? A válasz az alábbi gondolatmenetben rejlik:

1. A branch tesztelés alaposabb az utasítás-tesztelésnél, mert:
   a. 100%-os utasítás lefedettséget elérhetünk anélkül, hogy 100%-os branch lefedettséget érnénk el.
   b. 100%-os branch lefedettség viszont nem érhető el 100%-os utasítás lefedettség nélkül, vagyis a teljes branch lefedettségből következik a teljes utasítás lefedettség is.
2. Az útvonal-tesztelés alaposabb a branch tesztelésnél, mert:
   a. 100%-os branch lefedettséget elérhetünk anélkül, hogy 100%-os útvonal-lefedettséget érnénk el.
   b. 100%-os útvonal lefedettség viszont nem érhető el 100%-os branch lefedettség nélkül, vagyis a teljes útvonal lefedettségből következik a teljes branch lefedettség is.

A fentiek alapján egy tetszőleges x kódrészletre fennáll az:
`utasítás-lefedettség(x) ≤ branch-lefedettség(x) ≤ útvonal-lefedettség(x)`
reláció (4. ábra).



4. ábra: A lefedettségek közti reláció

Ezek alapján azt gondolnánk, hogy érdemes lehet az útvonal lefedettséget megcélozni, azonban ez több okból is nehézkes lehet:

- A kódméret növekedésével egyre több branch kerül a program folyamatába, ami hatványozottan növeli a lehetséges útvonalak számát.
- Minél több útvonal van, annál több tesztesetre van szükség ezek lefedéséhez.
- Ha a program ciklust tartalmaz, akkor minden egyes iterációhoz külön teszteset szükséges, ami általános esetben végtelen is lehet.
- Így bármilyen ciklust tartalmazó program útvonal-lefedése rendkívül nehéz (lehetetlen) vállalkozás.

2.5.1. Példák

Tekintsük az alábbi kódot:

```
void Test_Me (Integer p, Integer q, Integer y)
{
     Integer x;

     if (p > 11) {       // s1
          x = 5;         // s2
     } // end if

     if (q < 5) {        // s3
          y = q / x;     // s4
     } // end if
} // Test_Me
```

A feladat, hogy a lehető legkevesebb tesztesettel érjünk el 100%-os utasítás-, branch-, valamint útvonal-lefedettséget.

Ebben a példában 4 utasítás található. A két „`if`" 1-1 utasításnak számít `(s1, s3)`, valamint a „`then`" ágak is 1-1 utasításként értelmezendőek `(s2, s4)`. (Abban az esetben, ha egy `then` ágban több – nem elágazó - utasítás is lenne, akkor lehetne alap blokk lefedettséget számolni az utasítás-lefedettség helyett. Jelen esetben az utasítás- és az alap blokk lefedettség ugyanazt az értéket szolgáltatja.)

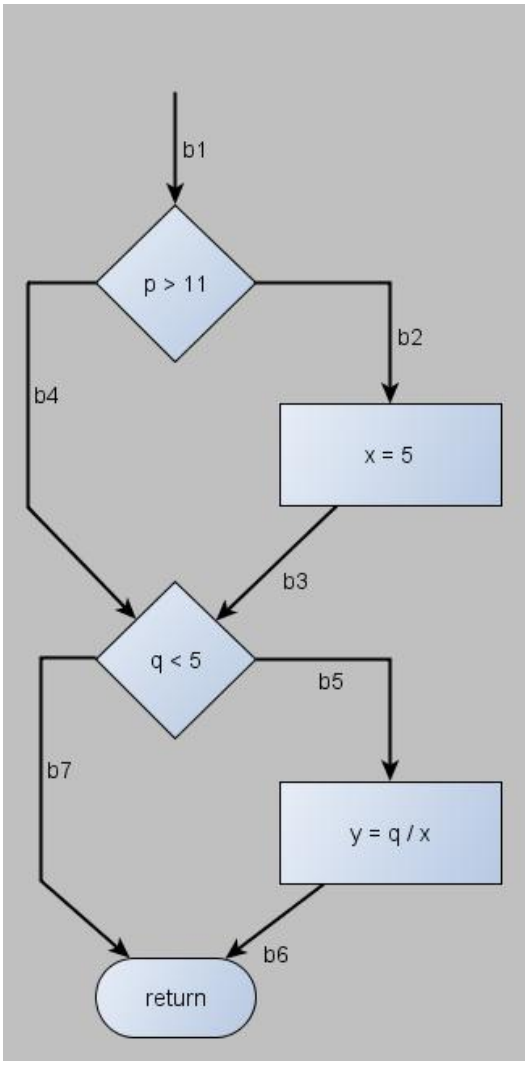
Ahhoz, hogy 100%-os utasítás lefedettséget érjünk el, mind a 4 fenti utasítást le kell fednünk tesztesettel. A kérdés az, hogy hány teszteset kell ahhoz, hogy 100%-os utasítás-lefedettséget érjünk el? Vizsgáljuk meg az alábbi tesztesetet:

```
{
Integer a, b, c;
     a = 12;
     b = 4;
     Test_Me (a, b, c);
}
```

Látható, hogy ezen értékek mellett mind a 4 utasítást érintjük, hiszen:

- az `s1` utasítás mindig lefut (`p` értékétől függetlenül)
- az `s2` utasítás lefut, hiszen `p = 12` kielégíti azt a feltételt, ami `s1`-ben van
- az `s3` utasítás szintén mindig lefut
- az `s4` utasítás lefut, hiszen `q = 4` kielégíti azt a feltételt, ami `s3`-ban van

Most térjünk át a branch lefedettségre. Először is, rajzoljuk fel a fenti programrészlet folyamat-ábráját (5. ábra):

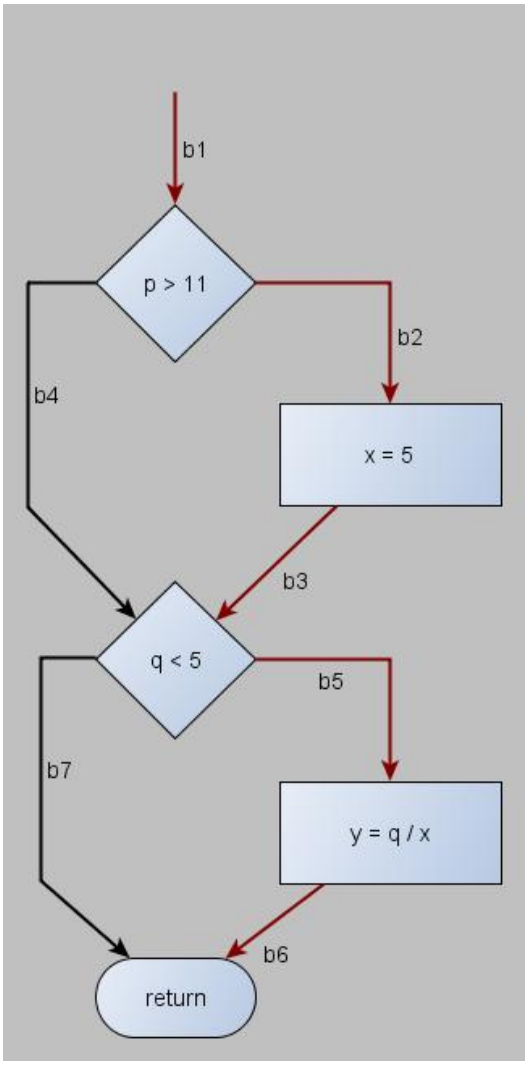


5. ábra: A példában megadott kódrészlet folyamatábrája

Vajon a korábban megadott – és 100%-os utasítás-lefedettséget biztosító – teszteset biztosít-e 100%-os branch lefedettséget is? Ahhoz, hogy ezt eldöntsük, vegyük a `p = 12` és a `q = 4` értékeket, és menjünk végig a megfelelő branch-eken. Látható, hogy a `b4` és a `b7` ágakat nem érinti ez a teszteset, így kell egy másik teszteset is, ami ezeket az ágakat is lefedi. Így 2 teszteset szükséges a 100%-os branch lefedettség eléréséhez:

```
{
     Integer a, b, c;
     a = 12;
     b = 4;
     Test_Me (a, b, c);  // b1, b2, b3, b5, b6
     a = 11;
     b = 5;
     Test_Me (a, b, c);  // b4, b7
}
```

Most vizsgáljuk meg a útvonal lefedettséget. Egy útvonal (path) egy lehetséges program-végrehajtási folyamat. Az alábbiakban látható pirossal kiemelve egy útvonal (6. ábra):



6. ábra: Egy útvonal (path)

Ahhoz, hogy 100%-os útvonal lefedettséget érjünk el, meg kell találnunk az összes lehetséges útvonalat a kódban, és ezeket kell lefednünk tesztesetekkel. Látható, hogy a fenti példában 4 lehetséges útvonal van:

- {b1 → b2 → b3 → b5 → b6}
- {b1 → b2 → b3 → b7}
- {b1 → b4 → b5 → b6}
- {b1 → b4 → b7}

Az előzőekben definiált 2 teszteset csak az első, illetve az utolsó útvonalat fedi le, így nem eredményez 100%-os útvonal-lefedettséget. Az alábbi teszteset-halmaz viszont már igen:

```
{
     Integer a, b, c;
     a = 12;
     b = 4;
     Test_Me (a, b, c);  // 1. útvonal

     a = 12;
     b = 5;
     Test_Me (a, b, c);  // 2. útvonal

     a = 11;
     b = 4;
     Test_Me (a, b, c);  // 3. útvonal

     a = 11;
     b = 5;
     Test_Me (a, b, c);  // 4. útvonal
}
```

2.6. Gyakorlati megvalósítás

Gyakorlatban a kód lefedettség méréséhez szükségünk van arra, hogy valamilyen módon „megjelöljük” azokat a kódrészeket, amiket érintettünk. Kód-instrumentálásnak nevezzük azt a folyamatot, melynek során kiegészítjük a kódot olyan kódrészletekkel, amik a lefedettségi információt szolgáltatják. Instrumentálni lehet a forráskódot vagy a program bináris formáját.

Az instrumentálást számos eszköz segíti, ilyenek például:

- JAVA esetén: Clover, Cobertura, JCoverage, GroboUtils
- C/C++ esetén: Insure++, Tessy, TestWell CTC++, Trucov

2.6.1. Példa

Az alábbiakban egy JAVA példán keresztül szemléltetünk egy instrumentált kódot.

A `java.lang.instrument` csomagot fogjuk használni. A JVM elindítható a `javaagent` opcióval, ekkor meg kell adnunk egy speciális `jar` fájl, egy úgynevezett `java agent` nevét. A JVM az agenteket még az alkalmazás `main()` metódusának meghívása előtt elindítja. Az agent ekkor regisztrálhat egy `java.lang.instrument.ClassFileTransformer` objektumot, aminek a `transform()` metódusa minden osztály betöltésekor meghívódik. Ennek segítségével pedig már bárhogyan módosíthatjuk a betöltendő osztályt.

Mivel a JVM nem tudja magától, hogy melyik osztályban van az a metódus, amiben beregisztráljuk a `ClassFileTransformert`, ezért először is szükségünk van egy osztályra, amiben van egy

```
premain(String arguments, Instrumentation inst)
```

vagy egy

```
premain(Instrumentation inst)
```

metódus.

Továbbá a `jar` fájl `manifest`-jében szerepelnie kell a `Premain-Class` attribútumnak, aminek az értéke a `premain()` metódust tartalmazó osztály neve. A JVM indulás után

megpróbálja meghívni ennek az osztálynak a kétparaméteres `premain()` metódusát, ha pedig ilyen nincs, akkor az egyparamétereset. (A kettő között az a különbség, hogy az elsővel fel tudjuk dolgozni a `javaagent` opcióval átadott argumentumokat is.)

A `manifest` egy másik fontos attribútuma a `Boot-Class-Path`, ahol megadhatjuk azt a classpath-ot, amin azok az osztályok megtalálhatók, amiket az agent használ.

Az alábbi példa annyit csinál, hogy egy osztály betöltésekor kiírja annak nevét. A `PremainClass` így néz ki:

```
package ex.instrumentation;

import java.lang.instrument.Instrumentation;

public class PremainClass {
          public static void premain(String arguments,
Instrumentation inst){
               inst.addTransformer(new Transformer());
          }
}
```

Az `inst.addTransformer()` hívással regisztráljuk a `Transformer` osztály egy példányát az osztálybetöltésekre.

```
public class Transformer implements ClassFileTransformer{
     public byte[] transform(ClassLoader loader, String className,
                             Class<?> classBeingRedefined,
                             ProtectionDomain protectionDomain,
                             byte[] classfileBuffer)
                               throws IllegalClassFormatException {
          System.out.println(className);
          return null;
     }
}
```

A `ClassFileTransformer` egyetlen implementálandó metódusa a `transform()`, ami megkapja többek között a betöltendő osztály nevét. Ezt beolvashatja, módosíthatja és egy byte tömbben visszaadhatja az új osztálydefiníciót. Esetünkben most csak kiírja az osztálynevet és null-t ad vissza.

Egy tetszőleges alkalmazást ezután így indíthatunk el az agent-tel (feltéve, hogy az `agent.jar` tartalmazza a fenti osztályokat és a megfelelő `manifestet`):

```
java -javaagent:agent.jar
```

2.7. Komplexitás-alapú tesztelés

A komplexitás-alapú tesztelés az útvonal tesztelés egy specializált változata, a ciklomatikus komplexitási mértéket használjuk fel ahhoz, hogy kiválasszuk a program teszteléséhez szükséges végrehajtási útjainak alaphalmazát. Az alaphalmazban szereplő utak alapján elkészítve az inputokat hatékonyabb teszteléshez jutunk.

A komplexitás számítása a gráfelméleten alapul, az egyik metrika számítási módszer megalkotója Thomas J. McCabe, Sr. volt. A McCabe ciklomatikus komplexitás a tesztelt program vezérlési gráfjában lévő lineárisan független utak maximális halmazát határozza

meg, az optimális teszteléshez az ebben szereplő utakat kell a teszteléskor bejárnunk. Ha egy metódus nem tartalmaz döntési utasításokat (mint például `if` feltétel, vagy `for` ciklus), akkor egyetlen végrehajtási utat kapunk, csak ezen haladhat a program végrehajtása. Ha egy metódusban csak egyetlen feltétel szerepel, akkor az két utat határoz meg a végrehajtásban.

A módszer szigorúbb feltételeket határoz meg, mint az utasítás vagy elágazás alapú tesztelés, ennek köszönhetően hatékonyabb lehet a hibák detektálása is. A ciklomatikus komplexitás segítségével egy felső korlát számítható a teljes branch lefedettség eléréséhez szükséges tesztesetek számáról. Továbbá a ciklomatikus komplexitás egy alsó korlátot is jelent a vezérlési folyam gráfban meghatározható összes útra. Akkor lehet szükség ilyen mélységű tesztelésre, ha a megbízhatóság egy kiemelten fontos szempontja a fejlesztett szoftvernek.

A három lefedettség között a következő összefüggés írható tehát fel:

branch lefedettség $\leq$ ciklomatikus komplexitás $\leq$ útvonalak száma (útvonal lefedettség)

2.7.1. A tesztelt útvonalak és a komplexitás kapcsolata

A komplexitás alapú tesztelés gyakran használt más kód-alapú tesztelési módszerek mellett, vagy követelmény alapú teszteléssel együtt. A vezérlési folyam gráf analízisének segítségével elő tudjuk állítani útvonalak egy alaphalmazát, amit tesztelésre használhatunk. A módszerre példát is mutatunk.

Az alaphalmazban a lineárisan független utakat szeretnénk meghatározni. Ez azt jelenti, hogy egy újonnan kiválasztott útvonal előáll az alaphalmazban szereplő utak lineáris kombinációjaként. A lineáris függetlenség egy heurisztika az útvonal tesztelés korlátozására. A matematikai bizonyítás túl messzire vezetne minket a gyakorlati alkalmazás területétől, ehelyett inkább vegyünk egy szemléletes példát, amin keresztül érthetővé válik a teszt alaphalmaz meghatározása.

Legyen G = < E, N> egy erősen összefüggő gráf, ahol E az élek halmaza, N pedig a csúcsok halmaza G-ben. Egy gráf erősen összefüggő, ha bármely csúcsból bármely csúcsba vezet út. Az ilyen típusú gráfok esetében a lineárisan független utak száma megadható a következőképpen:

$v(G) = |E| - |N| + 1$

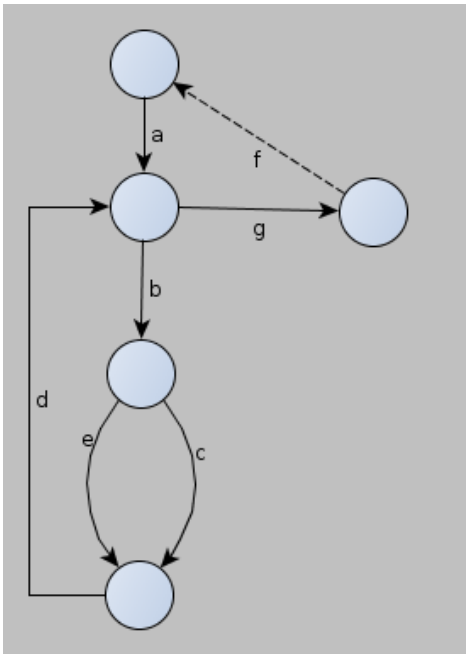
A v(G) érték a McCabe ciklomatikus komplexitás, ami felhasználható programok komplexitásának meghatározására. A v(G) értékre összegzésül a következő megállapításokat tehetjük:

- $v(G) \geq 1$
- v(G) a maximális lineárisan független utak száma G-ben, ami egyben a tesztfuttatások során bejárandó utak száma is
- bármilyen 1-kifokú csúcs beszúrása vagy törlése nem változtatja meg v(G)-t
- ha G csak egy utat tartalmaz, akkor v(G) = 1
- v(G) egyedül a G által reprezentált elágazási struktúrától függ.

Előfordulhat, hogy nem tudunk minden utat végigtesztelni a gráfban. Ez olyankor lehetséges, ha nincs input, mellyel a kérdéses út végrehajtható lenne. Például egymást követő két teljesen megegyező feltétel esetén nem tudunk olyan inputot megadni, aminek feldolgozásakor a kimenet függene az első feltételtől, de a másodiktól nem, és fordítva.

Ezek a vezérlési függőségek eltávolíthatóak, vagy a teszt kritériumok relaxálhatóak, így a gyakorlatban feltehetjük, hogy minden út tesztelhető. A továbbiakban olyan gráfokról beszélünk, amiknek pontosan egy bemenete és egy kimenete van.

Egy vezérlési folyam gráfra még nem lenne igaz az erősen összefüggőség, de ezt a kritériumot könnyen teljesíteni tudjuk azáltal, hogy a kilépési csúcstól a belépési csúcsig hozzáveszünk plusz egy élet. Tekintsük a következő vezérlési gráfot (7. ábra):



7. ábra: Egy példa program vezérlési folyam gráfja

Hogyha a fenti módon képzeletben erősen összefüggővé bővítettük a vezérlési gráfot, akkor a végleges ciklomatikus komplexitás számítása az előző képlet segítségével adódik; a képlet a következőképp módosul: v(G) = |E| - |N| + 2. Megmutatható továbbá az is, hogy egy jól strukturált programnál – amiben nincs *goto* utasítás, valamint pontosan egy belépési és egy kilépési ponttal rendelkezik – a ciklomatikus komplexitást megkaphatjuk a v(G) = |D| + 1 képletből, ahol *D* a döntési pontok száma G-ben.

2.7.2. Az alaphalmaz meghatározása

Vezessük be a következő reprezentációt a végrehajtási utak lineáris függetlenségének bemutatásához: az *abcdefg* élek reprezentálhatóak egy vektorral, aminek értékei azt mutatják meg, hogy az egyes élek részt vesznek-e az útban. Például az *abedg* utat felírhatjuk a következőképp: abedg = <1 1 0 1 1 0 1 >. Ezzel a jelöléssel minden úthoz felírható egy vektor, amiket ha mátrixba rendezünk matematikai módszerrel megoldható feladatot kapunk. A célunk az alaphalmaz meghatározása, azaz a maximális lineárisan független utak kiválasztása.

Az utak lineáris kombinációján a vektor reprezentációjukon képzett lineáris kombinációjukat értjük. Az előző példában tekintsük a *bedg* utat, ami nem tartozik az alaphalmazhoz, mivel megadható két másik út lineáris kombinációjaként:

$$bedg = be + dg = < 0\ 1\ 0\ 0\ 1\ 0\ 0 > + < 0\ 0\ 0\ 1\ 0\ 0\ 1 > = < 0\ 1\ 0\ 1\ 1\ 0\ 1 >$$

Utak meghatározott halmaza lineárisan független, ha egyik út sem áll elő a halmazban lévő utak lineáris kombinációjaként. Az {abcdg, abedg, ag} halmaz lineárisan független, de például az {abcdg, abcdgbcdg, ag} nem, mert abcdg + abcdg - abcdbcdg = ag.

Az {abcdg, abeg, ag} halmaz az alaphalmaz is egyben, mivel a példánkon v(G) = 3. Ez azt jelenti, hogy az így meghatározott három utat kell a tesztelés során végrehajtani.

Az alaphalmaz megadásának elvi algoritmusa egyszerűen megkonstruálható. Adjunk egy tetszőleges utat a halmazhoz. Ezután minden iterációs lépésnél úgy adjunk hozzá újabb utakat a halmazhoz, hogy a halmaz útvonalaira teljesüljön a lineáris függetlenség. Ha a halmaz elemszáma eléri a ciklomatikus komplexitással kiszámolt értéket, akkor készen vagyunk, hiszen ekkor már nem találhatunk újabb utat, ami kielégítheti a lineáris függetlenség feltétellét. Ezután minden halmazba került úthoz keressünk egy megfelelő inputot. Az algoritmus a meghatározandó teszt halmazt eredményezi.

2.7.3. A baseline módszer

A forráskódon alapuló tesztmeghatározás egy jól automatizálható folyamat, így egy automatizált eszköz használata számos előnyt jelenthet. Ennek ellenére természetesen lehetőségünk van manuális módszerrel is dolgozni.

A manuális tesztelési folyamat során a tesztelendő szoftver moduljait vizsgáljuk, a részekre bontott kód alapján elkészítjük a vezérlési folyam gráfot, amiből meghatározzuk az utak teszt alaphalmazát, majd minden útnak megfelelő inputtal elvégezzük a tesztelést.

A vezérlési folyam gráf összes útjának részhalmazát jelentő alaphalmaz meghatározására szolgál az úgynevezett „baseline” módszer. A baseline szó arra utal, hogy az algoritmus egy kiindulási útvonal felhasználásával építi fel az alaphalmazt. Az alapötlet, hogy a kiindulási útvonal pontosan egy döntésének kimenetelét megváltoztatva képezzünk új útvonalat, egészen addig, amíg az iteráció során új utakat kapunk.

Több variációja is létezik az algoritmusnak. Az egyszerűsített változat a program kimenetéig tartó legrövidebb távolság szerint választ a döntéseknél, így határozza meg a kiindulási utat. Hátránya, hogy nem flexibilis a kiindulás kötött meghatározása miatt, és könnyen eredményezhet nem végrehajtható teszteseteket. A gyakorlatban jobban használható változatot ismertetjük, ami lehetőséget biztosít a kiindulási út kiválasztására. Ez azért hasznos, mert léteznek fontosabb végrehajtási útvonalak a kódban, amiket ilyen módon kiemelhetünk a teszt alaphalmaz meghatározásához, valamint segít elkerülni a nem végrehajtható útvonalakat is.

Elsőként kiválasztunk a függvényben egy megfelelő utat, ami a kiindulási út lesz. Ezt önkényesen tesszük meg, a cél egy olyan út kiválasztása, ami legjobban reprezentálja a tesztelendő függvényt, és a normál működést valósítja meg, mintsem a kivételes viselkedést. A kivételes végrehajtási utak majd előállnak a kiindulási út módosítása során. Egy megfelelő kezdeti választás lehet például az, amelyik a legtöbb döntést érint. A következő út generálásához vegyük a baseline út első döntését, és változtassunk meg a kimenet irányát, törekedve arra, hogy a baseline lehető legtöbb későbbi döntésének kimenetelét megtartva a végpontot a legegyszerűbben érjük el. (A baseline által nem érintett szakaszok is bekerülnek az így kapott útba. A változtatás helye és a baseline-ba történő visszacsatlakozás közé eső döntések kimenetei közül a funkcionalitás szerint választva törekedhetünk a robosztusságra.) A következő úthoz vegyük ismét a baseline utat, de most a második döntés kimenetelét változtassuk meg. Ha a baseline összes döntésén

végighaladtunk, akkor vegyük a keletkezett útvonalak új döntéseit, és azok szerint is végezzük el a most leírt változtatásokat. Amikor minden döntés szerint meghatároztuk az utakat, akkor készen vagyunk. Többszörös szelekció esetében az összes lehetséges kimeneten végre kell hajtani a módosításokat.

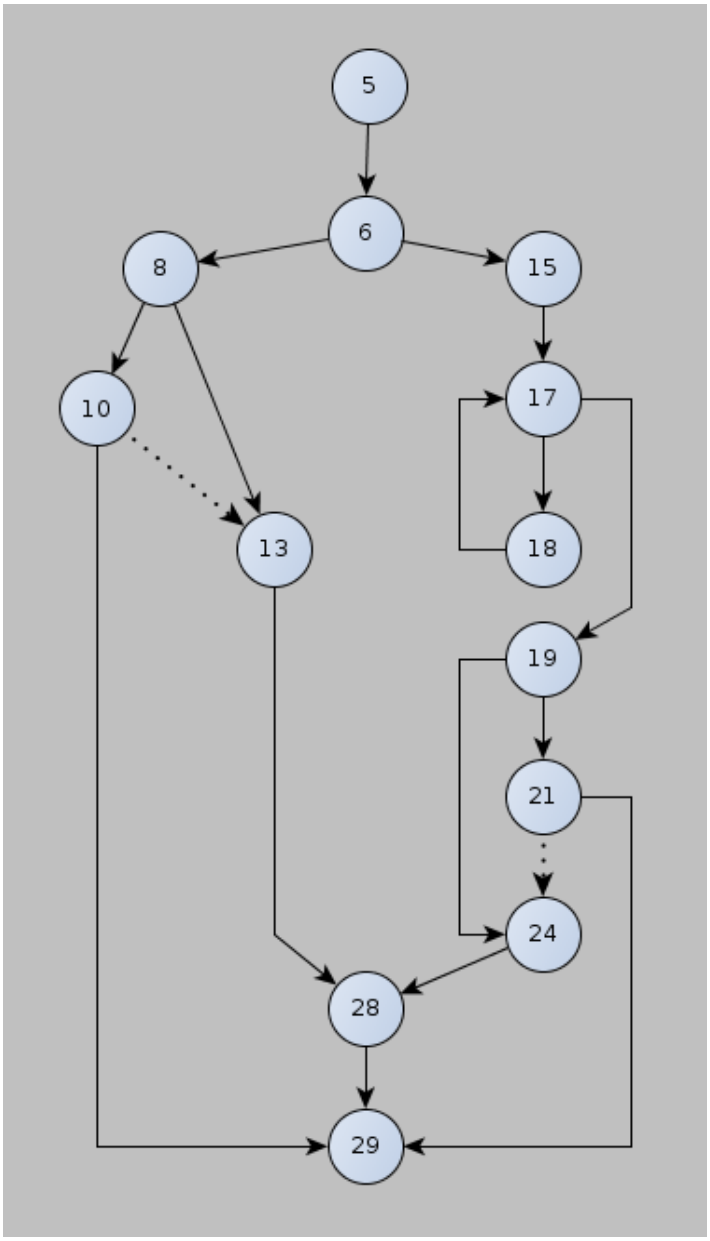
Gyakorlatban az új útvonalak konstruálása közben a tesztelő nagyobb szabadságra vágyik, a változtatott döntést követően flexibilisen szeretné megadni az út további irányát, ez azonban könnyen megszeghetné a független utak szabályát. Automatizáló eszközök segítségével lehetőség van úgy használni az algoritmust, hogy a tesztelő a függvény működésének megfelelően igazítsa a generált útvonalakat, és ellenőrizze, hogy érvényes marad-e az alaphalmaz függőségek szempontjából. Az automatizált eszközök használata nagyban segítheti a tesztelés sikerességét, és mivel a megfelelő alaphalmaz megtalálása különösen fontos, ezért érdemes erre a folyamatra a teszttervezés során több erőforrást áldozni.

2.7.4. A baseline módszer bemutatása példán

Az algoritmus működését egy láncolt lista beszúrását megvalósító metóduson mutatjuk be. Az utasításokat tartalmazó sorokat a könnyebb hivatkozás miatt számoztuk, méghozzá úgy, hogy az alapblokkok, valamint a be- és kilépési pontok sorszámait **kiemeltük**. A programkód a következő:

```
 1    struct node{
 2                int data;
 3                struct node *next;
 4     };
 5     struct node* add(struct node *head, int data, int debug) {
 6                struct node *tmp;
 7                if(head == NULL) {
 8                     head=(struct node *)malloc(sizeof(struct node));
 9                     if(head == NULL) {
10                          printf("Error! memory is not available\n");
11                          exit(0);
12                     }
13                     head-> data = data;
14                     head-> next = head;
15                } else {
16                     tmp = head;
17                     while (tmp-> next != head)
18                          tmp = tmp-> next;
19                     tmp-> next = (struct node *)malloc(sizeof(struct
node));
20                     if(tmp -> next == NULL) {
21                          printf("Error! memory is not available\n");
22                          exit(0);
23                     }
24                     tmp = tmp-> next;
25                     tmp-> data = data;
26                     tmp-> next = head;
27                }
28                return head;
29     }
```

Rajzoljuk fel a metódushoz tartozó vezérlési folyam gráfot az alap blokkok segítségével (8. ábra - az ábrán a számok az alapblokk kezdő utasításának sorát jelölik):



8. ábra: A példa programkódhoz tartozó vezérlési folyam gráf
(A sorszám az alapblokk kezdő utasításának sorszáma)

(A **10** és **21** pontokból az *exit* utasítás hatására megváltozott vezérlést is jelöltük a gráfon a megfelelő élek berajzolásával. Feltüntettük pontozott vonallal a feltételes utasítás által meghatározott vezérlési ágat is, amely az *exit* utasítás miatt lehetetlen él.)

A (lehetetlen élek törlésével előálló) vezérlési folyam gráfban azok a csomópontok tartalmaznak döntéseket, ahol pontosan két kimenő él van. Az egyes csomópontok megfelelnek az alap blokkoknak, az élek pedig a szekvenciális végrehajtást megszakító ugrásokat jelentik.

Válasszuk kiindulási útnak a következő, egy általános működést megvalósító végrehajtási utat: **teszt út 1** (kiindulás): 5 6 15 17 18 17 19 24 28 29. Ez megfelel az egy listaelemet tartalmazó inputnak.

A teszthalmazunk második útját megkapjuk, ha az első döntés kimenetelén változtatunk. Ez a döntés a 7. kódsorban található, a változtatás hatására a gráf **6 -> 8** ugrásán visszük tovább a vezérlést. Ebből a **teszt út 2**: 5 6 8 13 28 29. Ez az út megfelel az inicializálatlan listával futtatott tesztesetnek.

A harmadik úthoz a kiindulási út második döntésének kimenetelét változtassuk meg, ekkor **teszt út 3**: 5 6 15 17 19 24 28 29. Ez azt az esetet fedi le, amikor üres listával futtatjuk le a kódot.

A következő út a 19 sorszámú *alap blokk*ban szereplő döntés megfordításával áll elő. ami olyan tesztvégrehajtásnak felel meg, amikor hibás memóriafoglalás történik (ilyen előfordulhat például ha a heap memóriaterületen a megengedett méretnél nagyobbat akarunk lefoglalni, vagy ha elfogyott a memória). Az eredmény a **teszt út 4**: 5 6 15 17 18 17 19 21 29 lesz, mivel a 21 *alap blokk* kimenő éle lehetetlen, a szaggatott vonallal jelzett szakaszon folytatódik a végrehajtás az *exit* utasítás hatására.

Mivel a kiindulási úton nincs több olyan döntés, aminek a kimenetelén még nem változtattunk, ezért megnézzük az eddig keletkezett utakat. A *teszt út 2* a 8. csúcsnál elágazik egy döntésnél, így ebből képezünk új teszt utat, ami a következő lesz: **teszt út 5**: 5 6 8 10 29. Ez is hibás memóriafoglalást ellenőrző kódrészt fed le, most inicializálatlan lista esetében.

A program 22. sorában végezzünk el egy módosítást, töröljük az *exit* utasítást. Ez nyilván egy nem várt működést fog eredményezni, mert memóriafoglalási hiba esetén nem térünk vissza hibaüzenettel, aminek következtében a visszatérési értéket adó változó értéke memóriaszeméttel töltődik fel. A tesztelési alaphalmazt megadó algoritmus segítségével meghatározott teszt utak közül a *teszt út 4*-nek megfelelő futtatás deríti fel ezt a hibát.

Éles tesztelési feladatoknál a baseline algoritmus önmagában nem túlzottan használható, de a már említett automatizáló eszközök segítségével jó lefedettséget elérő módszereket lehet felépíteni. Egy általánosan használt megközelítés, hogy először a meglévő funkcionális teszthalmazunkat hajtjuk végre, figyeljük az érintett utakat, majd kiszámítjuk a teszt alaphalmazt. Ha egy megkapott tesztútvonal javítja az útvonal lefedettséget, akkor végrehajtjuk a programot az útvonalat eredményező inputok segítségével, különben elvetjük azt.

2.8. Használat, hátrányok

2.8.1. Használat

Egy termék release előtt mindig definiálni kell egy lefedettség értéket, ami a tesztelés kilépési feltételeként funkcionálhat. Ez az érték függ a rendelkezésre álló tesztelési erőforrásról, valamint a követelményspecifikációban lefektetett minőségi céloktól. Nyilvánvaló, hogy egy kritikus szoftver esetén magasabb kritériumot kell támasztani.

Ha utasítás vagy branch lefedettséget mérünk, akkor általában 80-90%-os lefedettséget tűzzünk ki célul. A 100%-os célkitűzés gyakran nem optimális abból a szempontból, hogy

100%-os lefedettséget elérni nagyon nehéz, és az ebbe fektetett energia helyett inkább olyan területekre fókuszálhatunk, ahol sokkal eredményesebb lehet a tesztelés. A cél tehát, hogy a magas tesztelési produktivitást fenntartsuk: minél több eredményt elérni, minél kevesebb befektetéssel.

Érdemesebb először mindig a kevésbé megszorító lefedettséggel kezdeni (vagyis az utasítás-lefedettséggel). Ha ott értékelhető eredményt érünk el, akkor mehetünk tovább branch, valamint útvonal-szintre.

2.8.2. Kód-lefedettség mérő eszközök

Manapság már léteznek olyan eszközök, amik direkt a fent felvázolt kód-lefedettséget mérik. Sajnos beüzemelésük nem mindig olyan egyszerű, mint amilyennek látszik. Ennek okai a következőkre vezethetőek vissza:

- Az eszköz nem ismeri fel az elemzendő kód 100%-át
- Az instrumentálás nem tökéletes
- Az instrumentálás utáni eredmény egyértelműen valótlan
- A tesztek lefuttatása nagyobb idő-ráfordítást igényel

Ha mégis sikerül olyan eszközt találni, ami komoly hiányosság nélkül működik, akkor az eszköz által generált riport a segítségünkre lehet, hogy mely területek nincsenek lefedve a kódban.

2.9. Adatfolyam tesztelés

Az adatfolyam tesztelés olyan kód alapú strukturális tesztelés, melynek során a változók értékadási és a kapott értékek felhasználási pontjait vizsgáljuk a végrehajtási utakon. Emiatt az útvonal tesztelés egyik változatának is szokás tekinteni.

A legtöbb program adatokon dolgozik, és a hibák jó része is az adatfeldolgozás során következik be. Már az 1960-as évek elején is foglalkoztak a forráskód analízisével, hogy megtalálják a programban a változók értékadása és felhasználása között jelentkező anomáliákat. Statikus analízis segítségével futtatás nélkül fény derülhet a kód problémáira, mint például ha egy változó definiálva van, de sosincs használva, hivatkozva, vagy épp ellenkezőleg: használunk egy olyan változót, amit nem (vagy többször) definiáltunk. Ezek a vizsgálati módszerek később a fordítókba is beépültek.

A fejezet első fele az ilyen statikus elemzési módszerekkel foglalkozik, ami nem kapcsolódik szorosan az adatfolyam teszteléshez és a lefedettség méréshez, azonban segít megértenünk, milyen problémák előfordulása teszi szükségessé a fejezet második felében bemutatott adatfolyam tesztelési módszerek alkalmazását.

2.9.1. Fogalmak, jelölések

Az adatfolyam tesztelés a kód utasítás alapú vezérlési folyam gráf reprezentációjából indul ki. A továbbiakban a P programhoz tartozó vezérlési folyam gráfot G(P)-vel, a gráfban lévő változók halmazát V-vel jelöljük. A P-beli lehetséges utak halmazára a PATH(P) jelölést vezetjük be.

Azt mondjuk, hogy egy $n \in G(P)$ csúcs definíciós csúcs a $v \in V$ változóra nézve – jelölésben DEF(v, n), röviden Def – ha v változó értéke az *n* csúcshoz tartozó

utasítás(részek) végrehajtása során kerül definiálásra. Ilyen csúcsok lehetnek például az input utasítások, értékadási utasítások, ciklus vezérlési utasítások, eljáráshívások:

- input(v)
- v := exp(.)

Egy ilyen csúcs végrehajtása után a változóhoz tartozó memóriaterület tartalma megváltozik.

A használati csúcs olyan n ∈ G(P) csúcs a v ∈ V változóra nézve – jelölésben USE(v, n), röviden Use – amelyhez tartozó utasításban a *v* változó értéke felhasználásra kerül. Az output utasítások, értékadási utasítások, feltételek, valamint a ciklus vezérlési utasítások használati csúcsok:

- output(v)
- x := exp(v)
- if cond(v) then
- while cond(v) do

Egy ilyen csúcs végrehajtása után a változóhoz tartozó memóriaterület tartalma változatlan marad.

Ha egy használati csúcs USE(v, n) esetén az n utasítás predikátum (elágazási) utasítás, akkor a csúcsot predikátum használati (predicate use) csúcsnak nevezzük, jelölésben p-Use. Ellenkező esetben az elnevezés számítási használati (computation use) csúcs, jelölésben c-Use. Ezt úgy is definiálhatjuk, hogy az olyan használati csúcsok a predikátum használati csúcsok, amelyek kimenő éleinek száma ≥ 2, és számítási használati csúcs az olyan, amelyre a kimenő élek száma ≤ 1.

Azt mondjuk, hogy egy p ∈ PATH(P) út definíció-használati út (du-út) egy v ∈V változóra nézve, ha DEF(v, m) és USE(v, n), ahol *m* az út kezdő csúcsa, *n* pedig az út befejező csúcsa.

Definíció mentes útnak nevezzük az olyan du-utakat a v ∈ V változóra nézve, amelyek a kezdő csúcson kívül nem tartalmaznak más *v* változóhoz tartozó definíciós csúcsot.

2.9.2. Statikus adatfolyam analízis

A forráskód statikus vizsgálatával az adatfelhasználás mintáit keresve fény derülhet bizonyos anomáliákra. Hogy az adatok kezelése során fellépő helyzetek közül melyek okoznak meghibásodást, az az adott programozási nyelvtől is függ. Ebben az alfejezetben most egy meghatározott programkód kitüntetett változójára mutatunk adatfolyam jellemzőket. Az előző definíciókhoz képest egyszerűsítsük a jelöléseket a következőképpen:

- d – definiált
- u – használat (use)
    - c – számítási használat (computational)
    - p – predikátum használat (predicate)
- k – megszüntetett (killed), terminált, definiálatlan
- ˜x – azt jelöli, hogy a megelőző műveletek nincsenek hatással x-re
- x˜ – azt jelöli, hogy a következő műveletekre nincs hatással x

A felsorolt jelöléseket kombinálva a következő anomáliák azonosíthatók (ahol lehetett példát is megadtunk az előző fejezet baseline módszerhez használt kódjából, feltüntetve a vizsgált változót):

Anomália		Magyarázat	Példa
~d	először definiált	Megengedett	5-6-7-15-16 DEF(tmp, 16)
du	definiált – használt	Megengedett, normál működés	5-6-7 DEF(head, 5), USE(head,7)
dk	definiált – megszüntetett	Potenciális hibaforrás. Használat nélküli megszüntetés definíciót követően.	5-6-7-15-16-17-19-20-23-24-25-26-27-28-29 (debug)
~u	először használt	Potenciális hibaforrás. Adat definiálás nélkül használt.	
ud	használt – definiált	Megengedett, a már felhasznált adat újradefiniálása.	7-8 (head)
uk	használt – befejezett	Megengedett	25-26-27-28-29 (data)
~k	először megszüntetett	Potenciális hibaforrás. Definiálás nélküli megszüntetés.	
ku	megszüntetett – használt	Súlyos defektus. Adat megszüntetése után akarjuk használni.	
kd	megszüntetett – definiált	Megengedett, a megszüntetett adat újradefiniálása.	
dd	definiált – definiált	Potenciális hibaforrás. Kettős definiálás.	
uu	használt – használt	Megengedett, normál működés	16-17 (head)
kk	megszüntetett – megszüntetett	Potenciális hibaforrás.	
d~	utoljára definiált	Potenciális hibaforrás.	
u~	utoljára használt	Megengedett	25-26-27-28-29 (data)
k~	utoljára megszüntetett	Megengedett, normál működés	28-29 (head)

A statikus adatfolyam tesztelés során minden változóra megvizsgáljuk a fenti kapcsolatokat. Ezzel számos lehetséges hiba beazonosítható a kód futtatása nélkül is, olyankor azonban nem alkalmazható, amikor egy adatváltozó állapota csak futás közben derül ki. Ez akkor fordul elő, amikor a vizsgált változó egy kollekció, és egy másik változóval indexelve használjuk. Ilyenkor az adatváltozó állapotáról csak dinamikus körülmények között tudunk megállapításokat tenni, amihez a megfelelő utakat lefedő tesztesetek kiválasztására és végrehajtására van szükség, így most a dinamikus adatfolyam teszteléssel folytatjuk a téma tárgyalását.

2.9.3. Adatfolyam tesztszelekciós stratégiák

Teszteset szelekcióhoz különféle stratégiák szerint határozhatunk meg kritériumokat. Az egyes stratégiák a definiált fogalmakon alapulnak. A vezérlési folyam gráf alapján minden egyes változóra meg kell határozni a Use, p-Use, c-Use, Def csúcsokat, valamint meg kell adni a köztük lévő du-utakat, majd ezen utak közül a kritérium által meghatározottakat megtartva meg tudjuk adni lehetséges utak egy részhalmazát, amivel elvégezhető a tesztelés. A következőkben legyen T a vezérlési folyam gráfon lehetséges utak egy ilyen részhalmaza.

2.9.3.1. Minden du-út tesztelés (All du-path testing)

Utak egy T halmaza kielégíti a Minden du-út kritériumot a $v \in V$ változóra, ha T lefedi a *v* változóhoz tartozó összes, v bármely definíciójától bármely használatáig tartó definíció- és körmentes utat.

2.9.3.2. Minden Use tesztelés (All Use testing)

A T halmaz kielégíti a Minden Use kritériumot a $v \in V$ változóra, ha T lefed legalább egy definíció- és körmentes utat a *v* változóhoz tartozó bármely definíciótól bármely használatig. Ez az előzőben definiált kritériumnál gyengébb, mert nem követeli meg egy adott definíció és egy adott használat közti összes út lefedését, elég csak legalább egy ilyet lefedni.

2.9.3.3. Minden p-Use/néhány c-Use tesztelés (All p-Use/some c-Use testing)

A T halmaz kielégíti a Minden p-Use/néhány c-Use kritériumot a $v \in V$ változóra, ha T tartalmaz legalább egy definíció mentes utat a *v* változóhoz tartozó bármely definíciótól *v* bármely predikátum használatáig. Ha a *v* változó egy definíciójához nincsen predikátum használat, akkor T-nek legalább egy c-Use használatig kell utat lefedni. Ennek a gyengébb változata a Minden p-Use teszt, amikor csak a p-Use használatokkal rendelkező változókat teszteljük.

2.9.3.4. Minden c-Use/néhány p-Use tesztelés (All c-Use/some p-Use testing)

A T halmaz kielégíti a Minden c-Use/néhány p-Use kritériumot a $v \in V$ változóra, ha T tartalmaz legalább egy definíció mentes utat a *v* változóhoz tartozó bármely definíciótól v bármely c-Use használatáig. Ha a *v* változó egy definíciójához nincsen c-Use használat, akkor legalább egy p-Use használatig kell T-nek utat lefednie. Ennek a gyengébb változata a Minden c-Use teszt, amikor csak a c-Use használatokkal rendelkező változókat teszteljük.

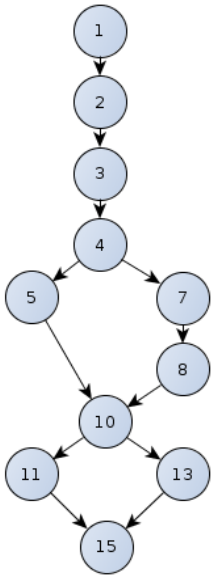
2.9.3.5. Minden Def tesztelés (All Def testing)

A T halmaz kielégíti a Minden Def kritériumot a v ∈ V változóra, ha T tartalmaz a *v* változóhoz tartozó minden definíciótól legalább egy kiinduló du-utat.

2.9.4. A stratégiák bemutatása egy példán keresztül

Tekintsük a következő kódrészletet, és a hozzá tartozó programgráfot!

```
1    void sample(int param_A, int param_B) {
2         cin >> first;
3         cin >> second;
4         if (param_A<100) {
5              cin >> first;
6         } else {
7              first=second;
8              paramA=100;
9         }
10        if (param_A>param_B) {
11             printf("%d\n",second);
12        } else {
13             first=paramB;
14        }
15   }
```



9. ábra: A példához tartozó programgráf

A feladatunk az lesz, hogy a Minden Use kritérium szerint határozzunk meg utak egy kielégítő T halmazát.

A következő táblázatban összegyűjtöttük a jellemzőket a példából, majd kihúztuk a du-utak közül azokat, amelyek nem voltak definíciómentesek. Az így megmaradt utak közül az ugyanabban a használatban végződőkből is csak egyet tartottunk meg (mivel a kritérium csak annyit köt ki, hogy legalább egy út fedje le).

változó	definiált	p-Use	c-Use	du-utak
param_A	1, 8	4, 10	-	1-2-3-4, 1-2-3-4-5-10, ~~1-2-3-4-7-8-10~~
param_B	1	10	13	1-2-3-4-5-10, ~~1-2-3-4-7-8-10~~, 1-2-3-4-5-10-13, ~~1-2-3-4-7-8-10-13~~
first	2, 5, 7, 13	-	-	
second	3	-	7, 11	3-4-7, 3-4-5-10-11, ~~3-4-7-8-10-11~~

Vegyük észre, hogy az 1-2-3-4 utat lefedi például az 1-2-3-4-5-10 út is, így ezt is törölhetjük a vizsgálatból.

Ezek után felírjuk az összes lehetséges utat a vezérlési folyam gráfban, és táblázatban jelöljük, hogy a kritérium szerint maradt egyes utakat melyek fedik le. Ezután az összes

kiválasztott utat lefedő lehetséges utak minimális halmazát véve megkapjuk a kritérium által meghatározott T teszt halmazt.

A gráfban a lehetséges végrehajtási utak: 1-2-3-4-5-10-11-15 (α), 1-2-3-4-7-8-10-11-15 (β), 1-2-3-4-5-10-13-15 (γ), 1-2-3-4-7-8-10-13-15 (δ)

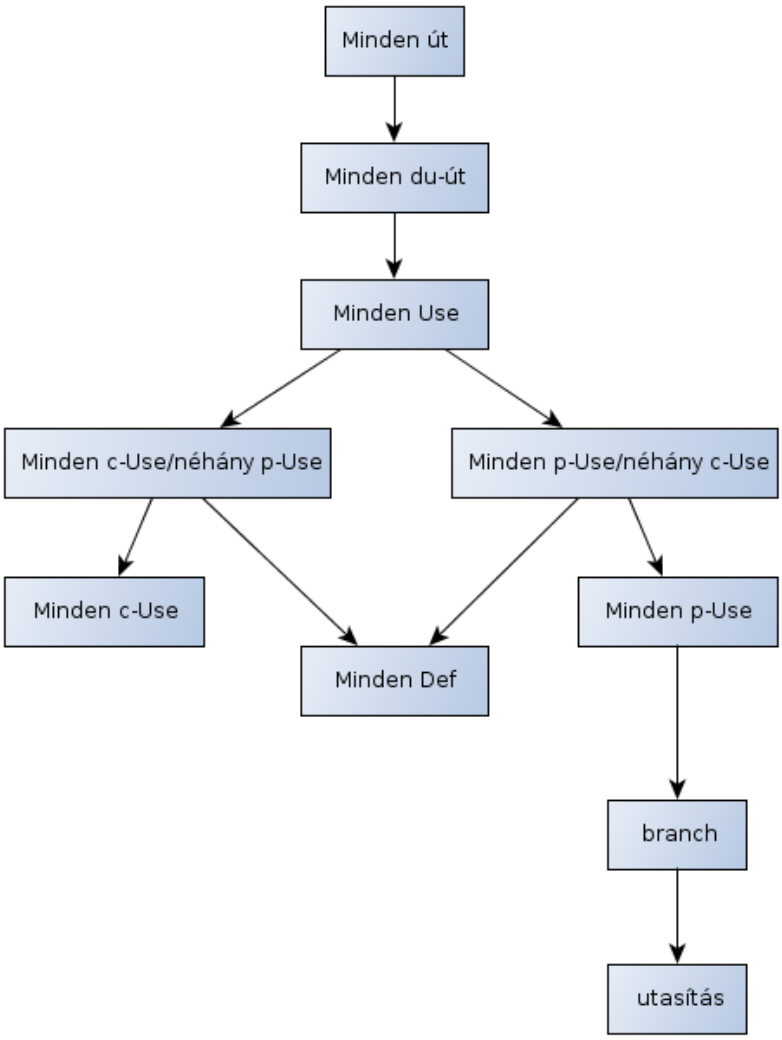
A Minden Use-tesztelésre kiválasztott utakat ezek a következőképpen fedik le:

	1-2-3-4-5-10	1-2-3-4-5-10-13	3-4-7	3-4-5-10-11
α	x			x
β			x	
γ	x	x		
δ			x	

Tehát az α, γ és δ úton elégséges tesztelnünk a teljes Minden Use-teszt lefedettséghez (vagy α, β, γ is jó megoldás természetesen).

2.9.5. Stratégiák összehasonlítása

A 10. ábrán látható az eddig megismert módszerek kapcsolata. A nyilak az erősebb módszerek/stratégiák felé mutatnak. Az ábra jobb oldalához egy kis magyarázat tartozik. A Minden p-Use és branch tesztelés közötti reláció akkor igaz, ha a p-Use általunk használt pont alapú definícióját él alapúra bővítjük méghozzá úgy, hogy a pontból kimenő mindkét élet p-Use használatnak tekintjük. Ezzel biztosítható, hogy egy döntésnek – amelyben egy adott v változó szerepel – mindkét ágát teszteljük.



10. ábra: Adatfolyam alapú teszteset szelekciós stratégiák közti összefüggések

2.10. Mutációs tesztelés

Az eddig megismert módszerek segítségével teszteseteket tudtunk meghatározni a program felépítésének ismeretében. Most olyan technikát mutatunk be, amely új tesztesetek megírását ugyan kevésbé segíti, de sokkal inkább alkalmas magának a teszteset-halmaznak a tesztelésére.

Mutációs tesztelés során előre definiált mutációs műveletek segítségével gyakorlatilag programhibákat helyezünk el a kódban. A P programnak P' egy mutációja, ha P egy előre meghatározott szintaktikai változtatásával áll elő. A változtatások különféle operátorok segítségével írhatjuk le. Ezek az operátorok az alkalmazott programozási nyelv szerint különbözőek lehetnek. A következőkben az objektumorientált nyelvekre általánosan jellemző, valamint a java-specifikus mutációkkal ismerkedhetünk meg. A továbbiakban ismertetünk néhány mutációt.

2.10.1. Osztály szintű mutációk

2.10.1.1. Egységbezárás

- ***Láthatósági módosítók változtatás operátor***: metódusok, változók hozzáférési szintjét változtatja meg. A mutáció célja a hozzáférhetőség helyességének igazolására létrehozni teszteseteket. Például Public Stack s; -> mutánsok: private Stack s; protected Stack s; Stack s;

2.10.1.2. Öröklődés

Az öröklődés egy hasznos és hatékony eszköze az objektumorientált nyelveknek, óvatlanul használva viszont számos hibát is eredményezhetnek. Az öröklődési hierarchiában korábban definiált változó, vagy metódus felüldefiniálása például potenciális hibaforrást jelenthet, amelyet operátorokkal is igyekeznek vizsgálni.

- ***A változó elrejtése törléssel operátor*** olyan esetben érvényesíthető, ha a leszármazott osztály változója ugyanazon névvel és típussal elrejti az ősben definiált változót. A mutált programban törlésre kerül a leszármazottból ennek a változónak a definíciója. A mutáns csak akkor deríthető fel teszttel, ha kimutatható, hogy a leszármazottban az így ősből használt referencia helytelen.
- A fordított mutáció, amikor nincs változó rejtés a leszármazottban, de a mutáció létrehoz egy ilyen felüldefiniálást, ez *a* **változó *elrejtése beszúrással operátor***.
- Felüldefiniált metódushívások tesztelésénél is fontos kérdés, hogy a helyes metódus hívódik-e meg futtatáskor. Erre irányuló mutáció a ***metódus elrejtése törléssel operátor***, ami törli a felüldefiniált metódust a leszármazottban, így az erre a metódusra irányuló referenciák az ősben definiált metódust érik el.
- Néha szükség van a felüldefiniált metódusból az őst meghívni, például ha egy ősben lévő private adattagot nem lehet közvetlenül kezelnünk. Könnyen elkövethető az a hiba, hogy nem jókor hívjuk meg az őst, ami hibás állapotot idéz elő az objektumunkban. A ***felüldefiniált metódus hívási helyének változtatása operátor*** mutánsaihoz ezt a hívást mozgatja el a felüldefiniált metódus első vagy utolsó utasításának helyére, valamint egy utasítással feljebb vagy lejjebb.
- A következő operátort azért tervezték, hogy kiszűrje a felüldefiniálás egyes mellékhatásait. Előfordulhat olyan helyzet, hogy ősben szerepel két függvény, amik hívják egymást, de az öröklés során csak az egyik függvényt definiáljuk felül. Ha az öröklött, de nem felüldefiniált függvényt használjuk a leszármazottban, akkor az általa hívott felüldefiniált függvény miatt nem várt következményeket idézhet a helyzet elő. Ennek a tesztelésére a ***felüldefiniált metódus átnevezése operátor*** a származtatott metódust egy új metódusnak nevezi át az ősben, megszüntetve ezzel a függőségüket.
- A super kulcsszó gyakran használt, ha az ősben lévő adattagokat szeretnénk elérni a leszármazottból. Az öröklődés során változó elrejtése vagy metódus felüldefiniálás miatt a használata hibaforrást jelenthet, mert megváltoztatja a referenciát a leszármazott osztályból az ős osztályba. A ***super kulcsszó beillesztés operátorral*** a felüldefiniált vagy elrejtett metódus, változó helyes használatát hivatott ellenőrizni a referencia megváltoztatásával. Ennek a törlési változata, amikor elhagyjuk a mutánsban a super hívást, ezzel a leszármazott osztályban lesz érvényes a referencia. A super kulcsszóval a leszármazott konstruálásakor speciális módon is meghívhatjuk az ős konstruktorát. Az ***őskonstruktor explicit hívásának törlése operátor*** kitörli ezt a hívást, hogy a default konstruktorhívás történjen meg.

2.10.1.3. Polimorfizmus

- A polimorfizmus tulajdonságokat szintén fontos ellenőriznünk a lehetséges referencia kötéseken keresztül. ***A new hívás leszármazott típussal operátor*** az öröklődési lánc szerint módosítja a new meghívását leszármazott típussal.
- Az ***adattag definíció ős típussal operátor*** a leszármazott példányosításánál használt változó típusát változtatja meg őstípusra. Egy olyan teszteset tudja ezt a mutánst semlegesíteni, ami hibás viselkedést derít fel az ily módon ősként definiált objektum használatánál. Hasonló operátor létezik a függvények paraméter típusának ősre változtatásával.
- A ***type cast operátor beszúrása operátor*** ős típusra konvertál egy lokális változót a kódban. Ennek ellentett művelete, amikor a meglévő cast operátort törli, valamint egy harmadik mutációs lehetőség, ha megváltoztatja konvertálást az öröklődések szerint egy másik típusra.
- A ***referencia típus változtatása kompatibilis típusra operátor*** a változó értékadásoknál változtat a kódon, például *Integer* típusú változónak *Object* típusú értéket ad.
- Metódus overloading esetén a tesztelő meg akar bizonyosodni róla, hogy helyesen történik a kiterjesztett metódus hívása, a jó paraméterlistájú metódus került meghívásra, stb. Ezekre irányulnak a következő operátorok: a kiterjesztett metódus lecserélése operátor helyettesíti a mutált metódus törzsét a másik kiterjesztett metóduséval, törölhető is az egyik kiterjesztett metódus, valamint a paraméterlista permutálása is egy alkalmazott mutálási operátor.

2.10.1.4. Java specifikus operátorok

- Elrejtett változó esetén a ***this kulcsszó beszúrása operátor*** ellenőrzi a változó helyes használatát. Ennek ellentéte a kulcsszó törlése művelet hasonló esetben.
- A ***static módosító beszúrása/törlése operátorok*** módosítják a változók példány- vagy osztályváltozó szerepét.
- Az adattag inicializálás törlése operátor törli a konstruktorból az adattag értékadását, így a java szerinti megfelelő default értékkel lesz inicializálva az adattag.
- Ha készítünk default konstruktort, akkor a Java default konstruktort már nem készít. A ***Java-támogatott default konstruktor készítés operátor*** kikényszeríti ezt úgy, hogy törli az implementált default konstruktort, ezzel ellenőrizhetjük annak helyes implementálását.
- A Javában minden referenciával érhetünk el, ami segíti a típusellenőrzést, mégis elkövethetőek hibák emellett. Tipikus hiba, ha objektumreferenciát használjuk az objektum tartalma helyett, értékadásnál és összehasonlításnál egyaránt. Ennek tesztelésére a clone() lés equals() függvények beszúrásával és törlésével kaphatunk mutáns kódokat.
- Getter/setter metódusok használatánál könnyen hibát véthet a programozó, ha sok hasonló nevű és paraméterlistájú függvényt készít. Egy kompatibilis (ugyanolyan típusú adattagot beállító) metódushívásra változtatva ennek tesztelésére is lehetőség van a mutáció révén.

2.10.2. Metódus szintű mutációk

Metódus szinten számos mutációt megkülönböztethetünk, az elemi műveletek változtatása történik egy mutálás során. Most a leggyakrabban és legeredményesebben használtakat említjük meg.

Aritmetikai, logikai, feltételes és relációs operátorokat változtató mutáció során lecserélhetünk, beszúrhatunk, törölhetünk operátort a mutánsban. Javában aritmetikai operátor például az +, -, *, /, %, op++, ++op, op--, --op, logikai operátor az &, |, ^, ~, feltételes operátor &&, ||, &, |, ^, !, relációs operátor a <, <=, >, >=, ==, !=.

Bitléptető és értékadási operátorok esetén az operátorok cseréje, amit használni szoktak mutálásnál. A Java bitléptető operátorai a >>, <<, >>>>, értékadási operátorai a +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=.

2.10.3. Mutációk használata a tesztelésben

Mint említettük, a mutációs tesztelés sokkal inkább a meglévő teszteset halmazunk ellenőrzésére szolgál, mint annak direkt bővítését célozná. A tesztelés szempontjából ez is eléggé nyilvánvalóan fontos, hiszen ha kiderül, hogy a teszteseteink valamilyen szempontból hibásak, rosszak vagy elégtelenek, akkor maga a tesztelés hiába mutat jó eredményt (kevés hibát), ebből a program jó minőségére nem következtethetünk. A mutációs tesztelés tehát a tesztjeink hiányosságainak és gyengeségeinek felderítésére alkalmas azáltal, hogy a mutáns programokat a kérdéses teszteset halmazzal teszteli, és olyan mutációkat keres, amelyeket a teszteset halmazunk nem volt képes felfedezni. Ez például olyan esetekben lehetséges, ha a mutáns kód nem is lett futtatva vagy pedig a tesztesetek nem alkalmasak az adott mutánsba injektált hiba felfedezésére.

Először néhány fogalom. Legyen P’ a P programból képzett mutáns. Egy t teszteset megkülönbözteti a P’-t a P programtól, ha a végrehajtás során a két program eltérő eredményt szolgáltat. Ha egyetlen t sem tud különbséget tenni P’ és P között, akkor azt mondjuk, hogy a mutáns az eredetivel funkcionálisan ekvivalens.

Az alkalmazható módszer kiindulásaként rendelkezésre kell, hogy álljon a programhoz a teszteseteknek egy halmaza. Ha meggyőződtünk róla, hogy a tesztesetek 100%-ban sikeres végrehajtásúak, akkor alkalmazzuk a mutációs operátorokat a programon, elkészítjük az M mutáns programkódok halmazát (számos hibát injektálva a kódba). Miután ez megtörtént, a mutáns kódokon is végrehajtjuk a tesztjeinket. Ha jól összeállított a teszthalmaz, akkor valamelyik teszteset – amelyik lefedi a megváltoztatott utasítást a mutánsban – el fog bukni. Ellenkező esetben az injektált hiba nem került felfedezésre, mert egyik teszt sem, vagy nem megfelelő módon hajtotta végre a mutált utasítást, esetleg mert a mutáns program az eredetivel funkcionálisan ekvivalens. Ilyen esetben az eredeti programot, a mutánsát és a teszteseteket is újra kell vizsgálni, és ha nem ekvivalens mutációról volt szó, akkor új teszteseteket kell létrehozni a mutációs hiba kimutatására. A módszer alkalmazásával biztosíthatjuk, hogy a teszteset halmazunk képes a mutációk létrehozásakor felhasznált hibák felfedezésére, másrészt sikeres tesztelés esetén azt, hogy a program mentes a mutációktól.

A módszerrel a gyakorlatban számos probléma adódik. Nem megoldható például a generált mutált változatokról úgy általában eldönteni, hogy funkcionálisan ekvivalensek-e az eredeti programmal, vagy valóban a teszteset halmaz hiányossága okozza, hogy a mutáns program nem megkülönböztethető. Továbbá nagy programokon sokféle mutációt elvégezve rendkívül nagy mennyiségű mutáns kód készül, ezek tesztelése költséges, ami visszaveti a módszer életképességét. Az objektumorientált nyelvek és unit teszt keretrendszerek növekvő használata vezetett oda, hogy a legtöbb ilyen nyelvre készültek mutációs tesztelést támogató, mutációkat generáló illetve mutációs tesztet végző automatizált eszközök, amelyeket eredményesen használhatunk fel az alkalmazások kisebb önálló egységeinek teszteléséhez.

2.11. Egyéb struktúra alapú módszerek

Említés szintjén érintünk egyéb lefedettséget számító technikákat.

2.11.1. Feltétel és döntési lefedettség

A sima feltétel lefedettség esetében a döntés egyes összetevőinek (logikai tag/tényező) mindkét lehetséges kimenetét le kell fedni a többi feltételtől függetlenül. 100% feltétel lefedettség viszont nem implikálja a 100% döntési lefedettséget (pl. logikai ekvivalencia), tehát a kétféle lefedettség kombinálása mindkettőnél szigorúbb lefedettséget eredményez. A feltétel/döntési lefedettség esetében a feltétel és döntési lefedettség egyidejűleg teljesítendő. Ennek egy szigorúbb változata a többszörös feltétel lefedettség, amikor a döntés összetevőinek minden lehetséges kombinációját le kell fedni.

Létezik a módosított feltétel/döntési lefedettség is, amikor a program minden be- és kilépési pontját és minden feltétel és döntés mindkét kimenetét legalább egyszer érintjük, valamint egy döntés minden feltételének önállóságát ellenőrizzük, vagyis lesz két olyan futás, ami csak ebben a feltételben tér el egymástól.

2.11.2. Eljárás lefedettség

Az eljárás lefedettség nagyon hasonlít az utasítás lefedettségre, csak éppen nem utasítás, hanem eljárás szinten dolgozik: a meghívott eljárások arányát számolja. Akkor lesz 100%-os, ha minden eljárást legalább egyszer futtatunk.

2.11.3. Hívási lefedettség

Ahogy az előző, úgy ez a lefedettség is egy utasítás szinten létező, nevezetesen a branch lefedettség analógiája. A hívási lefedettség azt mutatja meg, hogy a program összes lehetséges eljáráshívási pontja közül mennyit érintettünk legalább egyszer.

2.11.4. Lineáris utasítás sorozat és ugrás (LCSAJ) lefedettség

Az LCSAJ (Linear Code Sequence And Jump) lefedettséget a forráskód alapján számítjuk. A forráskódban meghatározzuk az összes olyan útvonalat, amely egybefüggő utasítás-sorozat és amelynek utasításai sorban egymás után hajtódnak végre. Ezek az útvonalak tartalmazhatnak elágazó utasítást is, ha ennek ellenére a forráskód utasításai sorban egymás után futnak le. A végrehajtott és az összes ilyen útvonal aránya az LCSAJ lefedettség.

2.11.5. Ciklus lefedettség

A ciklus lefedettség a ciklusokra koncentrál. Az előírás az, hogy minden ciklust háromféleképpen kell futtatnunk: a ciklusmag végrehajtása nélkül, a ciklusmag pontosan egyszeri végrehajtásával, és a ciklusmag egynél többszöri iterációjával.

2.12. Feladatok

Oldjuk meg az alábbi feladatokat!

1. Tekintsük az alábbi kódrészletet:

```
Read P
Read Q
IF P+Q > 100 THEN
Print "Large"
ENDIF
If P > 50 THEN
Print "P Large"
ENDIF
```

Rajzoljuk fel a vezérlési gráfot (írjuk fel mátrix alakban is), és határozzuk meg a 100%-os utasítás-, branch-, és útvonal-lefedettségekhez szükséges tesztesetek számát!

2. Tekintsük az alábbi programot:

```
1  Subroutine Bubble (A, N)
2   BEGIN
3       FOR I = 2 to N DO
4           BEGIN
5                IF A (I) GE A (I-1) THEN GO TO EXIT
6                J = I
7       LOOP: IF J LE 1 THEN GO TO EXIT
8                IF A (J) GE A (J-1) THEN GO TO EXIT
9                TEMP = A (J)
10               A (J) = A (J-1)
11               A (J-1) = TEMP
12               J = J-1
13               GO TO LOOP
14      EXIT:NULL
15          END
16 END
```

Rajzoljuk fel a vezérlési gráfot (írjuk fel mátrix alakban is), és határozzuk meg a 100%-os utasítás-, branch-, és útvonal-lefedettségekhez szükséges tesztesetek számát!

3. Tekintsük az alábbi kódrészletet:

```
1  while (1)
2  {
3   if (*str == 0)
4       break;
5   else
6   {  i = 0;
7       if (*str == '')
8       {
9           while (i<n && *str!='')
10          {
```

```
11                ++str;
12                i++;
13            }
14        }
15        while (i<n && *str='')
16        {
17            jel = *str;
18            switch (jel)
19            {
20                case 1:
21                    jel = 'A';
22                    break;
23                case 2:
24                    jel = 'B';
25                    break;
26                case 3:
27                    jel = 'B';
28                    break;
29            }
30            ++str;
31            i++;
32        }
33    }
```

Rajzoljuk fel a vezérlési gráfot (írjuk fel mátrix alakban is), és határozzuk meg az utasításokat, branch-eket és útvonalakat! Határozzuk meg, hogy hány tesztesettel lehet 100%-os útvonal-lefedettséget elérni!

4. a-b) Tekintsük a következő kódrészleteket! Rajzoljuk fel a vezérlési gráfot (írjuk fel mátrix alakban is), és határozzuk meg a 100%-os útvonal-lefedettséghez szükséges tesztesetek számát a ciklomatikus komplexitás segítségével. Ezután határozzuk meg a minimálisan szükséges lineárisan független útvonalakat tartalmazó teszt alaphalmazt a bemutatott *baseline* módszer segítségével!

   a.

```
1   int minimum(int a,int b,int c)
2   /*Gets the minimum of three values*/
3   {
4    int min=a;
5    if(b<min)
6        min=b;
7    if(c<min)
8        min=c;
9    return min;
10  }
```

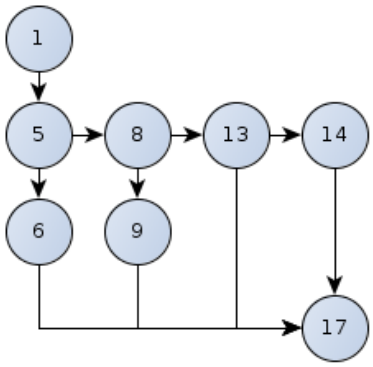
   b.

```
1   main(int argc, char *argv[])
2   {
3    int n, k, nak;
4    int i;
```

```
5
6    printf("Kérem n-t és k-t\n");
7    printf("az n alatt k értékének
   kiszámításhoz\n");
8    scanf("%d %d", &n, &k);
9
10   if (n < k || k < 0) {
11     printf("%d alatt %d nem értelmezett!\n", n, k);
12   } else {
13     nak = 1;
14     for (i = 1; i <= k; i++)
15       nak = nak * (n - i + 1) / i;
16     printf("%d alatt %d = %d\n", n, k, nak);
17   }
18 }
```

5. Tekintsük a következő kódrészletet, és a hozzá tartozó vezérlési folyam gráfot!

```
1  int tiphop()
2  {
3    int number;
4    cin >> number;
5    if ( number == 50 ) {
6           cout << "Congratulations! Right number!"
   << endl;
7    }
8    else if ( number < 50 ) {
9           cout << "Sorry! You were off!" << endl;
10   }
11   else {
12          cout << "Sorry! You were way over!" <<
   endl;
13          if ( number == 7) {
14               cout << "Oh, this is the magic
   number!" << endl;
15          }
16   }
17 }
```



11. ábra: Az 5. példához tartozó vezérlési folyamgráf

Az alábbi útvonalak közül melyek lehetséges végrehajtási utak?

a. 1 5 6 17
b. 1 5 8 6 17
c. 1 5 8 13 17
d. 1 5 8 13 14 17

Határozzuk meg a 100%-os útvonal-lefedettséghez szükséges tesztesetek számát a ciklomatikus komplexitás segítségével, majd határozzuk meg a minimálisan szükséges lineárisan független útvonalakat tartalmazó teszt alaphalmazt a bemutatott baseline módszer segítségével)!

6. Rajzoljuk fel az előző fejezet három feladatánál található kódrészletek programgráfját!
   a. Határozzuk meg az egyes változókhoz a definíciós csúcsokat!
   b. Az egyes változókra nézve melyek használati (és p-Use, c-Use) csúcsok?
   c. Határozzuk meg a programgráfban a lehetséges du-utakat (lehetséges du-út, ami lefedhető végrehajtási úttal)! A du-utakat a csomópontok számával azonosítsuk!

7. A tesztelés alapeleme a tesztvégrehajtások során elérhető programkomponensek meghatározása. Az alábbiak közül melyik válasz helyes a data-flow tesztelés során az elérhető komponensek meghatározására vonatkozóan?
   a. a program adatstruktúrái alapján történik
   b. az adat-folyam diagram útvonalai alapján történik
   c. egy végrehajtási út azon szegmenseiből történik, amik valamilyen adat definíciójával kezdődnek, és ezen definíció használatával végződnek
   d. egy végrehajtási út azon szegmenseiből történik, amik valamilyen adat definíciójával kezdődnek, és ezen adat újradefiniálásával végződnek
   e. egyik sem helyes a fentiek közül

8. A 10. ábra teszteset szelekciós módszerek közötti összefüggést mutatja. Melyik állítás hamis az alábbiak közül?
   i. Az *A* módszer lefedi a *B* módszert, ha *B* csúcs leszármazottja *A* csúcsnak a gráfban.
   ii. Az ábrából következik, hogy a Minden c-Use/néhány p-Use módszer lefedi a Minden Def módszert.
   iii. Az ábrából következik, hogy a Minden c-Use/néhány p-Use módszer lefedi az utasítás alapú módszert.
      a. Egyik sem
      b. (i) állítás
      c. (ii) állítás
      d. (iii) állítás
      e. Mind hamis

3. Specifikáció alapú tesztelés

Az előző fejezetben áttekintettük a kód alapú teszteset kiválasztást, most pedig rátérünk a program specifikáció alapján történő tesztelési módszerekre. Ezek a fekete doboz technikák módszeresen és alaposan felderítik a specifikált viselkedést. Leggyakrabban alkalmazott technikák az ekvivalencia partícionálás, határérték analízis, döntési tábla teszt, predikátum és részfüggvény tesztelés, állapotátmenet és használati eset tesztek.

Ebben a fejezetben megismerkedünk a részfüggvény teszteléssel, és annak egy változatával, a predikátum teszteléssel. Ezek a módszerek szorosan kapcsolódnak a döntési tábla teszthez, ami szintén a specifikációból indul ki, és az üzleti logikát meghatározó egyedi döntések kombinációit veszi sorra, és kapcsolja össze a lehetséges üzleti kimenetekkel.

Az ekvivalencia partícionálás szintén a részfüggvény tesztelés egy verziója. A motiváció ugyanaz: azok közül a bemenetek közül, amelyekre a program ugyanúgy viselkedik, elegendő egyet vizsgálni. Tehát az inputot egymással ekvivalens elemek valid (specifikációnak megfelelő) és invalid (specifikáció által kizárt, érvénytelennek tekintett) partícióira bontva partíciónként egyetlen teszteset elegendő.

Az ekvivalencia partícionálás kiterjesztéseként használható a határérték analízis, ezzel a módszerrel is részletesen fogunk foglalkozni a fejezetben. A motivációja az, hogy a legtöbb elkövetett hiba az értéktartományok határain jelentkezik, ezért kiemelten fontos ezeknek a területeknek a tesztelése. A határérték analízis az ekvivalencia partíciók határértékeit veszi alapul, a tesztesetek meghatározásához.

Felmerül a kérdés, hogy mi a kapcsolat a specifikáció alapú és a kód alapú módszerek használata között. Általánosságban elmondható, hogy a specifikáció alapú módszerek kellene, hogy a legfontosabbak legyenek, hiszen a specifikáció írja le a program ténylegesen elvárt működését. Ehhez persze szükséges a megbízható specifikáció. A struktúra alapú tesztelés ezeket egészíti ki, és tulajdonképpen a tesztesetek halmazának, a tesztelésnek a mérésével, ellenőrzésével, javításával járul hozzá ahhoz, hogy a tesztelés megbízható eredményt produkáljon. Az optimális tesztelés érdekében tehát mindkét tesztelési formára szükség van.

3.1. A specifikáció részei

Milyen komponenseket találhatunk egy specifikációban? Általában a specifikációk folyó szövegként, vagy pszeudo-kódként állnak rendelkezésre. Ezek közös jellemzője, hogy fel lehet írni belőlük feltételeket, és ezekhez a feltételekhez köthető akciókat, állapotokat.

„Ha egy A feltétel fennáll, akkor B eseménynek/állapotnak kell bekövetkeznie.”

A fenti gondolatmenetet egy példán keresztül szemléltetjük.

Specifikáció: Írjunk egy olyan programot, ami bemenetként három egész számot vár, melyek alapján eldönti, hogy a megadott értékek lehetnek-e egy háromszög oldalai, és ha igen, akkor milyen típusú az adott háromszög (szabályos, egyenlő szárú, általános háromszög). Pontosítva:

- Ha a megadott értékek különbözőek, tehát általános háromszöget alkotnak, akkor írassuk ki a következőt: „Általános háromszög”.
- Ha a megadott értékek egyenlő szárú háromszöget alkotnak, akkor írassuk ki a következőt: „Egyenlő szárú háromszög”.
- Ha a megadott értékek szabályos háromszöget alkotnak, akkor írassuk ki a következőt: „Szabályos háromszög”.
- Ha a megadott értékek nem alkotnak háromszöget, akkor írassuk ki a következőt: „Nem háromszög”.

Látható, hogy a fenti felsorolás mindegyike egy-egy feltételt, és a feltételhez köthető akciót/eseményt tartalmaz.

3.2. Részfüggvény tesztelés

Az előbbiekben ismertetett feltétel-esemény párok azért lesznek fontosak, mert megfeleltethetőek a teljes program egy kis komponensének, amihez aztán a teszteseteket megírjuk.

Részletesebben, feleltessünk meg egy adott programot egy függvénynek:

$$f : X \rightarrow Y ,$$

ahol f-et gyakran részfüggvények segítségével implementálják, vagyis $f = \{f_1, f_2, \ldots, f_n\}$, ahol $f_i : X_i \rightarrow Y$ és $X = X_1 \cup X_2 \cup \ldots \cup X_n$. Feltesszük, hogy minden f_i részfüggvény az X_i input partíció minden elemére pontosan ugyanazt a műveletsort hajtja végre. Amennyiben ez nem áll fenn, akkor tovább kell bontani (partícionálni) az inputot. Így ahhoz, hogy a tesztelés validálja a teljes függvény helyességét, minden részfüggvényt érintenünk kell legalább egyszer a tesztelés során. Azt mondjuk, hogy két teszteset *lazán kapcsolódik* egymáshoz, ha két különböző részfüggvényt tesztelnek. A nyilvánvaló cél, hogy olyan teszteset halmazt állítsunk össze, ahol a tesztesetek csak lazán kapcsolódnak egymáshoz. Így minden egyes teszteset a program egy kis komponensét teszteli, a kis részek uniója pedig megadja a teljes tesztelést. Ezt a módszert részfüggvény-tesztelésnek nevezzük.

Az előzőek szerint, ha adott két input, ami ugyanabban az X_i input partícióban van, akkor az ezt a partíciót lefedő f_i részfüggvény pontosan ugyanazt a műveletsort fogja végrehajtani az inputokon. Ekkor azt mondjuk, hogy a két input *szorosan összefüggő*. Ehhez hasonlóan, ha két különböző partícióból választunk inputokat (X_i, X_j), akkor két különböző részfüggvény (f_i, f_j) fogja ezeket végrehajtani (különböző operációkkal), így a két input *lazán összefüggő*. Egy, az összes partíciót legalább egyszeresen lefedő lazán összefüggő teszteset-halmaz tehát megfelelő, hiszen minden részfüggvényt letesztel, és nem redundáns.

Az X input halmaz partícionálását úgy végezzük el, hogy minden X_i input partícióra teljesüljenek a következők:

$$X_i = \{ x \mid x \in X \wedge C_i(x) \},$$

ahol $C_i(x)$ a program specifikációból származtatott feltétel, ami egyértelműen meghatározza az adott input partíciót.

3.2.1. Példák

Tekintsük a bevezetőben megismert program-specifikációt (háromszög). A teljes input halmazt felírhatjuk a következőképpen:

$$X = \{<x_1, x_2, x_3> \mid x_i \text{ integer}\}$$

(Az egyszerűség kedvéért az `integer` típuson kívül eső értékeket most nem vesszük figyelembe.)

A bevezetőben megadott specifikációból látható, hogy a fenti `X` input halmazt az alábbi 4 partícióra bonthatjuk:

$$X = X_1 \cup X_2 \cup X_3 \cup X_4,$$

ahol

$$X_1 = \{<x_1,x_2,x_3> \mid <x_1,x_2,x_3> \in X \wedge \text{HÁROMSZÖG}(x_1,x_2,x_3) \wedge \text{ÁLTALÁNOS}(x_1,x_2,x_3)\}$$

$$X_2 = \{<x_1,x_2,x_3> \mid <x_1,x_2,x_3> \in X \wedge \text{HÁROMSZÖG}(x_1,x_2,x_3) \wedge \text{EGYENLŐ}(x_1,x_2,x_3)\}$$

$$X_3 = \{<x_1,x_2,x_3> \mid <x_1,x_2,x_3> \in X \wedge \text{HÁROMSZÖG}(x_1,x_2,x_3) \wedge \text{SZABÁLYOS}(x_1,x_2,x_3)\}$$

$$X_4 = \{<x_1,x_2,x_3> \mid <x_1,x_2,x_3> \in X \wedge \neg\ \text{HÁROMSZÖG}(x_1,x_2,x_3)\}$$

ahol a $\text{HÁROMSZÖG}(x_1,x_2,x_3)$ predikátum jelentése: „$x_1, x_2,$ és x_3 egy háromszög oldalait alkotják”; a $\text{ÁLTALÁNOS}(x_1,x_2,x_3)$ predikátum jelentése: „$x_1, x_2,$ és x_3 egy általános háromszög oldalait alkotják”; az $\text{EGYENLŐ}(x_1,x_2,x_3)$ predikátum jelentése: „$x_1, x_2,$ és x_3 egy egyenlő szárú háromszög oldalait alkotják”; a $\text{SZABÁLYOS}(x_1,x_2,x_3)$ predikátum jelentése: „$x_1, x_2,$ és x_3 egy szabályos háromszög oldalait alkotják”.

Így látható, hogy erre a példára négy tesztesetből álló teszteset-halmazt kell lefuttatnunk, mégpedig úgy, hogy egy-egy teszteset egy-egy input partíciót fedjen le. A feltételnek megfelelő teszt-halmaz például:

$$T = \{<3,4,5>, <3,3,4>, <7,7,7>, <2,3,6>\}$$

Ez a kiválasztott teszteset-halmaz akkor és csak akkor tekinthető helyesnek, ha a `háromszög` program implementációja során pontosan négy különböző program-végrehajtási útvonal keletkezik, és mindegyik megfelel a specifikáció alapján létrehozott részfüggvényeknek. Felmerül a kérdés: vajon a programot így is fogják megvalósítani?

Most egy kicsit komplexebb példa következik:

Írjunk egy programot, ami újraformáz egy szöveget a következőképpen: adott egy SZÖVEGVÉGE karakterrel befejezett szöveg, aminek a szavai SZÓKÖZ, vagy ÚJSOR karakterrel vannak elválasztva. A feladat, hogy ezt a szöveget átformázzuk az alábbi szabályok szerint:

1. *Csak ott lehet sortörést alkalmazni, ahol a szövegben SZÓKÖZ vagy ÚJSOR karakter van.*
2. *Minden sort töltsünk ki addig, ameddig csak lehet.*
3. *Egyik sor sem tartalmazhat egy konstans MAX értéknél több karaktert.*

Az első feladat most is az input halmaz meghatározása. Tegyük fel, hogy a program karakterenként olvassa a kapott szöveget, így a teljes input halmaz a lehetséges ASCII karakterek.

Következő feladat az input halmaz partícionálása. Ennek első lépéseként az alábbi részhalmazokat definiáljuk:

- X_{EW}: Szó végét jelző (End-of-the-Word) karakterek halmaza, amibe beleértendő a SZÓKÖZ és az ÚJSOR karakter is.
- X_{ET}: Szöveg végét jelző (End-of-the-Text) karakterek halmaza.
- X_{AN}: Az összes lehetséges alfanumerikus és írásjel karakterek halmaza.

Tegyük fel, hogy az így definiált részhalmazok az input halmaz egy helyes partícionálását képezik, vagyis: $X = X_{EW} \cup X_{ET} \cup X_{AN}$ ahol X_{EW}, X_{ET} , X_{AN} páronként diszjunktak, továbbá a program által megvalósított funkcionalitást három részfüggvényre lehet bontani: f_{EW}, f_{ET} , f_{AN} . Az így kapott részfüggvények akkor alkalmasak részfüggvény-tesztelésre, ha a megvalósíthatóak szekvenciális utasításokkal (elágazás nélkül). Ellenkező esetben tovább kell bontani az input halmazt. A fent említett három részhalmaz implementációjára láthatunk egy pszeudokódot az alábbiakban:

1. $x \in X_{EW}$ – Az input karakter SZÓKÖZ vagy ÚJSOR karakter

```
if szohossz > 0 then
     begin
          if sorhossz + szohossz ≥ MAX then
               begin
                    kiir(ujsor);
                    sorhossz := szohossz
               end
          else
               begin
                    kiir(szokoz);
                    sorhossz := sorhossz + szohossz
               end
          kiir(szo);
          szohossz := 0;
     end;
beolvas(char);
```

2. $x \in X_{ET}$ – Az input karakter egy SZÖVEGVÉGE karakter

```
if szohossz + sorhossz ≥ MAX then
    kiir(ujsor)
else
    kiir(szokoz)
kiir(szo);
kiir(karakter);
exit;
```

3. $x \in X_{AN}$ – Az input karakter egy alfanumerikus karakter

```
hozzafuz(karakter, szo);
szohossz := szohossz + 1;
if szohossz > MAX then
    begin
         kiir(jelzes);
         exit
else
    beolvas(char);
```

A fenti implementációból látszik, hogy minden részfüggvény tartalmaz elágazást (`if` vezérlési szerkezet), ami azt jelzi, hogy az ugyanazon részfüggvényhez tartozó input adatok különböző műveleteken keresztül hajtódhatnak végre.

Ez a megfigyelés azért helytálló, mert a példa programot fel kell készíteni arra, hogy az egyes karakterek feldolgozása függ a korábbi karakterektől, vagyis az aktuális input környezetétől. Például ha egy SZÖVEGVÉGE karaktert olvasunk be, akkor attól függően, hogy van-e még hely az aktuális sorban, vagy oda, vagy a következő sorba kell kiírnunk az aktuális szót. Ez a feltétel további részhalmazokra bontja az aktuális input-részhalmazt. Ezt a következőképpen formalizáljuk: (az előző pszeudo-kód 2-es pontját bontjuk tovább).

2.1. `x ∈ XET és szohossz + sorhossz ≥ MAX` – Az input karakter egy SZÖVEGVÉGE karakter, és az aktuális szó hosszabb, mint az aktuális sorban lévő hely.

```
kiir(ujsor);
kiir(szo);
kiir(karakter);
exit
```

2.2. x ∈ XET és szohossz + sorhossz < MAX – Az input karakter egy SZÖVEGVÉGE karakter, és az aktuális szó elfér az aktuális sorban.

```
kiir(szokoz);
kiir(szo);
kiir(karakter);
exit
```

Látható, hogy azáltal, hogy a 2-es feltételhez hozzávettünk 1-1 plusz predikátumot, az összes olyan inputra, ami kielégíti az adott predikátumot ugyanúgy fog viselkedni a program.

Alkalmazva ezt a módszert az 1-es és 3-as input-részhalmazokra, a következő input-felosztást kapjuk:

1.1.1. x ∈ XEW és szohossz > 0 és szohossz + sorhossz ≥ MAX – Az input karakter SZÓKÖZ vagy ÚJSOR karakter, és az aktuális szó hosszabb, mint az aktuális sorban lévő hely.

```
kiir(ujsor);
sorhossz := szohossz;
kiir(szo);
szohossz := 0;
beolvas(karakter);
```

1.1.2. x ∈ XEW és szohossz > 0 és szohossz + sorhossz < MAX – Az input karakter SZÓKÖZ vagy ÚJSOR karakter, és az aktuális szó elfér az aktuális sorban.

```
kiir(szokoz);
sorhossz := sorhossz + szohossz;
kiir(szo);
szohossz := 0;
beolvas(karakter);
```

1.2. x Є XEW és szohossz = 0 – Az input karakter SZÓKÖZ vagy ÚJSOR karakter, amit szintén egy ilyen jellegű karakter előzött meg.

```
beolvas(karakter);
```

2.1. x Є XET és szohossz + sorhossz ≥ MAX – Az input karakter egy SZÖVEGVÉGE karakter, és az aktuális szó hosszabb, mint az aktuális sorban lévő hely.

```
kiir(ujsor);
kiir(szo);
kiir(karakter);
exit
```

2.2. x Є XET és szohossz + sorhossz < MAX – Az input karakter egy SZÖVEGVÉGE karakter, és az aktuális szó elfér az aktuális sorban.

```
kiir(szokoz);
kiir(szo);
kiir(karakter);
exit
```

3.1. x Є XAN és szohossz > MAX – Az input karakter egy alfanumerikus karakter, és az aktuális szó túl hosszú.

```
hozzafuz(karakter, szo);
szohossz := szohossz + 1;
kiir(jelzes);
exit;
```

3.2. x Є XAN és szohossz ≤ MAX – Az input karakter egy alfanumerikus karakter, és az aktuális szó nem túl hosszú.

```
hozzafuz(karakter, szo);
szohossz := szohossz + 1;
beolvas(karakter);
```

Az így kapott input-részhalmazoknak megfeleltetett részfüggvények már alkalmasak tesztelésre. Olyan teszteset-halmazt kell összeállítanunk, ami lefedi az összes fent leírt input halmazt.

Egy másik módszer a példában szereplő input részhalmazok további felosztására input-karakterpárok definiálása, vagyis annak a megadása, hogy két egymás után következő karakter esetén hogyan kell viselkednie a programnak. Formálisan: Mivel X-et, az összes lehetséges input halmazát három további részhalmazra bontottuk (lásd korábban), ezért két egymás után következő karakter összesen 3*3 = 9 részhalmazt határoz meg, a következők szerint:

$$X = X_{EW} \cup X_{ET} \cup X_{AN}$$

$$XX = (X_{EW} \cup X_{ET} \cup X_{AN})(X_{EW} \cup X_{ET} \cup X_{AN})$$

Ezt a felírást követve az alábbi input részhalmazokat definiálhatjuk (a továbbiakban x_{i-1} az aktuális karaktert megelőző karakter, míg x_i az aktuális karakter jelölése):

1 $x_{i-1}x_i \in X_{AN}X_{AN}$:

1.1. $x_{i-1}x_i \in X_{AN}X_{AN}$ és `szohossz > MAX` – Az aktuális karakter egy túl hosszú új szó része.

```
hozzafuz(karakter, szo);
szohossz := szohossz + 1;
kiir(jelzes);
exit;
```

1.2. $x_{i-1}x_i \in X_{AN}X_{AN}$ és `szohossz ≤ MAX` – Az aktuális karakter egy megfelelő hosszúságú szó része

```
hozzafuz(karakter, szo);
szohossz := szohossz + 1;
beolvas(karakter);
```

2 $x_{i-1}x_i \in X_{AN}X_{ET}$:

2.1. $x_{i-1}x_i \in X_{AN}X_{ET}$ és `sorhossz + szohossz ≥ MAX` – Az aktuális karakter a szöveg végét jelöli, és nincs elég hely az aktuális sorban.

```
hozzafuz(karakter, szo);
kiir(ujsor);
kiir(szo);
exit;
```

2.2. $x_{i-1}x_i \in X_{AN}X_{ET}$ és `sorhossz + szohossz < MAX` – Az aktuális karakter a szöveg végét jelöli, és van elég hely az aktuális sorban.

```
hozzafuz(karakter, szo);
kiir(szo);
exit;
```

3 $x_{i-1}x_i \in X_{AN}X_{EW}$:

3.1. $x_{i-1}x_i \in X_{AN}X_{EW}$ és `sorhossz + szohossz ≥ MAX` – Az aktuális karakter az új szó végét jelöli, amit ki kell írni egy új sorba.

```
kiir(ujsor);
kiir(szo);
sorhossz := szohossz;
szohossz := 0;
beolvas(karakter);
```

3.2. $x_{i-1}x_i \in X_{AN}X_{EW}$ és `sorhossz + szohossz < MAX` – Az aktuális karakter az új szó végét jelöli, amit ki kell írni az adott sorba.

```
kiir(szokoz);
kiir(szo);
sorhossz := sorhossz + szohossz;
szohossz := 0;
beolvas(karakter);
```

4 $x_{i-1}x_i \in X_{ET}X_{AN}$ – Az aktuális karakter felesleges.

Az elvárás az, hogy ne történjen semmi.

5 $x_{i-1}x_i \in X_{ET}X_{ET}$ – Az aktuális karakter felesleges.
Az elvárás az, hogy ne történjen semmi.

6 $x_{i-1}x_i \in X_{ET}X_{EW}$ – Az aktuális karakter felesleges.
Az elvárás az, hogy ne történjen semmi.

7 $x_{i-1}x_i \in X_{EW}X_{AN}$ – Az aktuális karakter felesleges.
```
hozzafuz(karakter, szo);
szohossz := szohossz + 1;
beolvas(karakter);
```

8 $x_{i-1}x_i \in X_{EW}X_{ET}$ – Az aktuális karakter egy SZÖVEGVÉGE karakter.
```
kiir(karakter);
exit;
```

9 $x_{i-1}x_i \in X_{EW}X_{EW}$ – Az aktuális karakter felesleges.
```
beolvas(karakter);
```

Így tehát a specifikáció alapján elvégezett részfüggvény teszteléshez olyan input-halmazt (a példára levetítve: olyan szövegrészeket) kell megadnunk, ami lefedi a fenti 12 eset mindegyikét.

3.2.2. Gyakorlati megközelítés

A részfüggvény tesztelés egy gyakorlati megközelítése a kategória-partícionálás módszere (Category-Partition Method – CPM). A technika lényege, hogy a specifikációból kiindulva egyre kisebb – önállóan is tesztelhető – egységekre bontja a specifikációt. A CPM konkrét lépései az alábbiak:

1. A specifikáció elemzése – a tesztelő önállóan tesztelhető funkcionális egységeket keres a specifikációban. Minden egyes ilyen egységhez meghatározza az alábbiakat:
   a. A funkcionális egység paraméterei.
   b. A paraméterek jellemzői.
   c. A környezet olyan objektumai, amik befolyásolhatják a funkcionális egység működését.
   d. A környezeti objektumok jellemzői.

   Ezután a tesztelő a fenti jellemzőket kategóriákba sorolja annak megfelelően, hogy milyen hatással van az adott funkcionális egységre.
2. A kategóriák lehetőségekre osztása – a tesztelő meghatározza azokat a lehetséges eseteket, amik bekövetkezhetnek egy paraméter / környezeti objektum kiértékelése során.
3. Feltételek, megszorítások megadása – a tesztelő a lehetőségek halmazán megszorításokat és feltételeket határoz meg az alapján, hogy hogyan viszonyulnak egymáshoz az egyes lehetőségek.

4. Teszt specifikáció írása és feldolgozása – A kategóriákat, választásokat és megszorításokat (feltételeket) egy formális teszt specifikációba írjuk, ezután pedig egy feldolgozónak adjuk, ami egy teszt keretet készít.
5. A generált kimenet értékelése – A tesztelő eldönti, hogy a generátor által adott kimenet megfelelő-e, vagy szükséges a teszt specifikáció újragondolása és újraírása. Ha változtatás szükséges, akkor visszamegyünk az előző ponthoz.
6. Amikor a teszt specifikáció stabil, akkor a tesztelő a teszt generátor által készített kimenetből teszteseteket, teszt eljárásokat készít.

3.3. Predikátum tesztelés

A specifikációban a teljes program részkomponenseit egyértelműen meghatározó predikátumokat azonosíthatunk, ahogy azt a részfüggvény tesztelésnél láthattuk is. Ilyenkor, ha a specifikációból azonosítható predikátumok C_1 és C_2, akkor a részfüggvény tesztelés a $C_1 \wedge C_2$, $C_1 \wedge \neg C_2$, $\neg C_1 \wedge C_2$, $\neg C_1 \wedge \neg C_2$ predikátumok által meghatározott teszteseteket veszi figyelembe (ezeket a kombinációkat résztartományoknak nevezzük). Az résztartományok a feltételek által meghatározott partíciókat tovább finomítják. A predikátum tesztelés a részfüggvény tesztelés egy változata, a vizsgált komponenseket pusztán a predikátumok igaz és hamis kiértékelésére adott tesztesetek határozzák meg. Ez azt jelenti, hogy nem szükséges minden egyes predikátum kombinációihoz teszteket megadni.

A predikátumok lefedéséhez választott tesztesetek a predikátum kombinációk egy részhalmazából kerülnek ki. Például legyen x_1 és x_2 ami a C_1 és C_1 predikátumokat elégíti ki, x_3 és x_4 ami C_2-t és C_2-t. Ha mind a négy fenti résztartományt kielégítené $\{x_1, x_2, x_3, x_4\}$, akkor részfüggvény tesztelésről beszélünk. x_1-x_2 kiválasztása függetlenül történik x_3-x_4 választástól, így nem garantált, hogy a meghatározott tesztesetek az összes résztartományt kielégítik. Ugyanakkor a tesztesetek meghatározásához szükséges analízis így jóval könnyebb lehet, emiatt a predikátum tesztelés kevésbé költséges, mint a részfüggvény tesztelés.

A tesztelési módszer hiba felfedezési képessége javítható úgy, hogy a predikátumokat a kiválasztott teszteseteknek megfelelően vizsgáljuk. Teszteset szelekciókor megjegyezzük az résztartományt, amihez tartozik, és ha lehetséges újabb tesztesetnek olyat veszünk, amelyik résztartományjához még nem volt teszteset rendelve. Meg kell jegyezni, hogy mivel nem függetlenek a predikátumok, ezért lehet olyan résztartomány, amelyhez nem lehet kielégítő tesztesetet meghatározni.

A bemutatott technika a kód alapú tesztelési módszerek közül az elágazás tesztelésnek feleltethető meg, hiszen a specifikációból felírható predikátum azonosítható a forráskódban található elágazás fogalmával.

3.4. Ekvivalencia partícionálás

Az ekvivalencia partícionálás a részfüggvény tesztelés egy verziója. A motiváció ugyanaz: azok közül a bemenetek közül, amelyekre a program ugyanúgy viselkedik, elegendő egyet vizsgálni. Ekvivalencia partíciókat a specifikációban található bemenő vagy kimenő adatokra, esetleg belső program logikára vonatkozó információk alapján készíthetünk. A

módszer alkalmazásával az inputot egymással ekvivalens elemek valid és invalid partícióira bontjuk, melyek uniója a teljes input tér. Invalid az a partíció, amelynek az értékei a specifikáció által implicit vagy explicit módon meghatározva nem számítanak a program helyes inputjának; például ha a specifikáció életkorról beszél, akkor a negatív értékek. Valid pedig a specifikációból meghatározott azon partíció, amelyre a programnak „normál" működést kell produkálnia. Az ekvivalencia partíciós tesztelés előírásai szerint ezek után partíciónként egyetlen teszteset elegendő, hiszen a partíció többi elemére (a feltételezések szerint) a program ugyanúgy fog működni.

3.5. Határérték analízis

A határérték analízis a részfüggvény tesztelésen, ekvivalencia partíciókon alapszik. Az említett módszerek által meghatározott bemeneti partíciók, pontosabban azok értékhatárai segítségével definiál különböző teszteseteket. A módszer azon elgondolásra épít, hogy a programhibák meglepően nagy arányban az ilyen értéktartományok határain következnek be, vagyis olyan helyek környezetében, ahol a program logika az egyik vagy másikfajta működés között dönt. Tipikus hiba, hogy a több, kevesebb, kisebb, nagyobb, előtt, után szavak tartalmazzák-e a megadott értéket vagy sem; kódolás közben a megfelelő reláció helyett annak egyenlősséggel kiegészített, vagy éppen egyenlőség mentes alakját használjuk; a kódban a határértéket tévesen ±1-gyel torzítva adjuk meg; stb.

Egy programhoz meghatározott ekvivalencia partíciók között megkülönböztetünk valid és invalid partíciókat. A határérték-analízis a valid partíciók közötti, illetve a valid-invalid partíciók közötti határok közvetlen környezetét vizsgálja.

A határérték tesztelésnek megkülönböztetünk két- illetve hárompontos változatát. A kétpontos határérték tesztelés a határt, mint értéket nem reprezentáló választóvonalat tekinti, és ennek két oldalát, vagyis az elhatárolt partíciók szomszédos elemeit tekinti tesztelendő esetnek. Ez két teszteset partíció-határonként. (Egy másik értelmezés szerint egyazon partíció két elemét, a minimális és maximális elemet definiálja. Ez a nem korlátos partíciók esetének végiggondolásával és helyes kezelésével ugyanazt az eredményt adhatja, mint az előző verzió.) A hárompontos változat a specifikációban meghatározott határértékeket veszi alapul, és ezekre, valamint ezek egy-egy külső (nem a határérték partíciójához tartozó) és belső (a határértékkel egy partícióban lévő) szomszédjára definiál teszteseteket. A szomszédok meghatározásánál nagyon fontos a pontosság meghatározása: az ezzel az egységgel megadott szomszédos értékeket vesszük fel a teszteléshez.

Az imént meghatározott módszer és a részfüggvény tesztelés között több különbség is megfigyelhető:

- A határérték analízis a program által definiált partíciók helyes megkülönböztetését teszteli – az alterek határai mentén. Ezzel szemben a predikátum tesztelés során az input alterek egy-egy reprezentatív elemét vizsgáljuk abból a szempontból, hogy a végrehajtás során helyes működést tapasztalunk-e.
- Ezen túlmenően további háttérismeret szükséges a programozási környezetről, valamint az alkalmazás működéséről is.

A határérték analízis jól használható olyan esetekben, amikor a program több független változót használ, amelyek valamilyen jól behatárolható fizikai mennyiséget reprezentálnak. Először nézzük a függetlenségi tényezőt. Annak érdekében, hogy megmutassuk a kijelentés

jogosságát, ellenpéldának hozhatunk egy dátumválasztó alkalmazást, amivel év, hónap, és nap input adható meg. A három érték nem független egymástól, különböző hónapok különböző számú napokat határoznak meg, az évek pedig szökőnapokat is. A határértékek megadása problémát okozhat, mivel független határértékeket tudunk könnyen meghatározni, és hozzáadott információk nélkül az analízis nem veszi figyelembe a funkcionális megfontolásokat, vagy a változók szemantikus jelentését.

Ugyancsak legalább ekkora gondot okozhat, ha egy érték nem felel meg a fizikai mennyiségi kritériumnak. Fizikai mennyiségeknél, mint hőmérséklet, nyomás, sebesség, terhelés, különösen fontos a határértékek vizsgálata. Amennyiben nem fizikai mennyiséget, hanem valamilyen logikai változó értékét kell tesztelnünk, a határérték analízis könnyen elvesztheti hatékonyságát: például egy telefon PIN kód bekérésénél nem sok értelme van a 00000-99999 határértékek vizsgálatának, mert semmivel sincs nagyobb hibafelderítő képessége, mint egy másik inputnak.

A következő fejezetekben a határérték analízis változatai, kiegészítései kerülnek bemutatásra.

3.6. Speciális érték tesztelés

Az előző módszerek közül ez a legkevésbé formális módszer. Az input értékek kiválasztásakor a módszer nagyban támaszkodik a tesztelő előismeretére, tehát részben tapasztalati alapú technika. A lényege, hogy a specifikációból a tesztelő meghatároz bizonyos speciális értékeket, amik a tapasztalata alapján problémásak lehetnek. Különösen fontos technika lehet olyan esetekben, amikor egyszerű határérték analízissel nem lehet a határértékeket függetlenül meghatározni, mert valamilyen függőség található a változók között.

3.7. Hibasejtés (error guessing)

A hibasejtés egy teljesen a tesztelő képességein, tapasztalatain, intuícióján alapuló tesztelési módszer. Ehhez a tesztelési formához semmilyen általános leírás nem adható meg, a tesztelt program jellegéből, környezetéből adódóan kell speciálisan elvégezni. A program fejlesztőjének képessége, szokásai, a programozási környezet jellemzői mind olyan információk, amelyek alapot adnak a hibák megsejtéséhez. Ez a tesztelési forma akkor a leghasznosabb, amikor más formális technikák, például az ekvivalencia partícionálás, vagy a határérték analízis nem képesek lefedni a tesztelés bizonyos területeit.

Amikor teszteseteket választunk ki, akkor a meglévő teszthalmazhoz egy gyengén kapcsolódó elemet kell választanunk, azaz olyan inputokkal megadott tesztesetet, ami az eddigiektől eltérő végrehajtási szekvenciát hoz létre. Hogy megfeleljünk ennek az elvárásnak, kerüljük a hasonló input partícióból származó teszteseteket, mert azok gyakran ugyanazokat az utasításokat érintik. Ennek ellenére lehetséges olyan speciális input, ami bár egy csoportba tartozik más már tesztelt inputtal, mégis létezik olyan feltétel, vagy programállapot, ami kiemeli, és gyengén kapcsolódóvá teszi a többi inputtal. Például egy programnak lehet olyan fájl az inputja, ami üres tartalom mellett más végrehajtást idéz elő, mint egyébként, vagy ha tudjuk egy fejlesztőről, hogy el szokta felejteni a lokális változók inicializálását, akkor ez is kiemeli a hozzá kapcsolódó inputokat. Ezek a speciális

körülmények sokszor nem deríthetők fel az előzetes analízis során, megtalálásuk automatizált módszerekkel nehéz.

A specifikációt vizsgálva a felkészült tesztelő ki tudja szűrni a tisztázatlan részleteket, amik a végleges programban potenciálisan hibás működéshez vezethetnek. Tekintsük például a következő specifikációt, amiből egy függvény készül: egy csomagküldő szolgálatnál 10.000 Ft összköltség alatt számítanak fel házhozszállítási díjat, valamint a 15 kg-nál nehezebb termékek költségesebb szállítása miatt 1000 Ft többletet kell megfizetni. A specifikációban nincs tisztázva, hogy milyen módon számítsuk a házhozszállítási díjat, hogyha a rendelt termékek ára összesen átlépi a 10.000 Ft-os határt, de az áruk között szerepel egy 15 kg-nál nehezebb termék is. További fejtörést okozhat, hogy a 10.000 Ft-os határt a szállítási költséggel, vagy anélkül kell figyelembe venni.

A módszerhez nagy segítség lehet a tapasztalat összegyűjtése. Érdemes lehet például úgynevezett „defect taxonomy”–t készíteni, ami tulajdonképpen egy hiba-katalógus. Ezt a tesztelők tapasztalata alapján, az általuk korábban tesztelt projektekből össze lehet állítani, és az újabb projektek alapján folyamatosan lehet frissíteni.

3.7.1. A módszer gyakorlati alkalmazása

A teszt körülmények ismeretében és a tesztelő saját tapasztalata, tudása alapján felállít egy listát a lehetséges hibaforrásokról, és ezen szempontok alapján próbál meg meghibásodást keresni a programban. Ha a kód nélkül, csak a specifikáció ismeretében kell elvégezni a tesztelést, akkor tapasztaltabb tesztelőkre érdemes bízni a feladatot. A hibasejtés a kód vizsgálata mellett hatékonyan használható, kiegészítve az egyéb módszerekkel. A kód gyanús pontjainak vizsgálatával könnyebben beazonosíthatók a meghibásodások, mintha a megfelelő input kitalálásával próbálnánk előhozni a hibát. A felülvizsgálatok során megtalált meghibásodások azonosítása és megszüntetése ráadásul felfedi a hiba helyét és természetét is, ami költségkímélő, mivel a hiba lokalizálása általában nehéz feladat.

A tesztelő korábbi tapasztalat alapján összegyűjti a programra jellemző hibafajtákat. Ennek a listának az összeállítása támogatható eszközzel, segítve ezzel a hibasejtés folyamatát. Készíthetünk adatbázist, amiben a speciális hibafajtákat írjuk le, és amelyet folyamatosan bővítünk az újabb és újabb felmerülő hibákkal. Ezek a listák a későbbi hasonló projekteknél eredményesen újrafelhasználhatók. Koncentrálhatunk a fejlesztők által jellemzően elkövetett hibákra is, melyeket naplózva könnyebben rájöhetünk, hogy mik a problémásabb pontok az egyes fejlesztők kódjaiban, amikre később jobban oda kell figyelni.

A legjellemzőbb hibafajtákra közlünk néhány példát. Ezek általánosan előforduló, programozási nyelvtől független hibaforrások. Speciális szakterületek, fejlesztési környezetek, programozási nyelvek szerint egyedi listák is létrehozhatók a tesztelés támogatására.

Tipikusan hibákat magukban hordozó szituációk lehetnek:

- adatok inicializálása
- adat rossz típus szerinti használata: például negatív számok miatt keletkezik meghibásodás, vagy mert nem numerikus adatot használunk numerikus helyett és fordítva
- hibát rejthet el az olyan tesztbázis generálás, ami inkább a rendszert szolgálja ki, nem pedig a valós környezetet próbálja meg minél pontosabban szimulálni

- hibakezelés: az együttesen előforduló hibaüzenetek priorizálása, érthető üzenetek a hiba bekövetkeztének körülményeiről, mi történik az adatokkal a hiba bekövetkezése után, jogosan folytatódik-e a vezérlés hiba esetén, stb.
- számítások: fizikai mennyiségeken végzett matematikai műveletek, összehasonlítások
- újraindítás / helyreállítás: a program állapotait vizsgálhatjuk megfelelő inputok esetén, hogy kiderítsük megfelelően működik-e az újból futtatott, vagy megszakított és újraindított program végrehajtása
- párhuzamos, konkurens végrehajtás: eseményvezérelt programok esetén például, vagy több processz egyidejű futtatásával végrehajtott tesztekkel

3.8. Tesztelési stratégia

Az előzőekben megismert tesztelési módszerek jól kombinálhatók egymással, mivel mindegyik módszer más megközelítéssel ad értékes, egymást kiegészítő teszteseteket. Összeállíthatunk segítségükkel egy stratégiát, ami segíti a hibák minél nagyobb számú felderítését:

- Ha a specifikáció tartalmaz döntéseket, predikátumokat, inputokra vagy outputokra megszorításokat, akkor használjuk a részfüggvény vagy predikátum tesztelést.
- Használhatjuk ezután a határérték analízist, ami az input és output változók értékhatárait vizsgálja.
- Teszteljük az input és output értékek valid és invalid ekvivalencia osztályait is, ha szükséges.
- A hibasejtés technikájával határozzunk meg újabb teszteseteket.
- Számoljunk lefedettséget a kódon a tesztesetek alapján. Ha az előre felállított kritérium nem teljesül, akkor egészítsük ki a teszteset halmazt a kritérium kielégítéséhez.

3.9. Egyéb specifikáció alapú módszerek

Említés szintjén érintünk néhány egyéb specifikáció alapú módszert:

3.9.1. Döntési tábla teszt

A predikátum teszteléshez szorosan kapcsolódó módszer, ami szintén a specifikációból indul ki, és az üzleti logikát meghatározó egyedi döntések kombinációit veszi sorra, és kapcsolja össze a lehetséges üzleti kimenetekkel. Mindezt egy táblázatban ábrázolva, ahol a táblázat felső sorai a specifikációból kinyerhető feltételeknek, az alsó sorai a specifikációból kinyerhető kimeneteknek, az oszlopok pedig az egyes teszteseteknek felelnek meg. A táblázat celláiban igaz/hamis értékek szerepelnek. A táblázat egy oszlopához tartozó logikai kombináció felső sorai azt mutatják meg, hogy mely feltétel volt igaz, az alsó sorok pedig azt, hogy ebben az esetben mely kimenetek feltételei teljesültek. A táblázat felső része egy teljesen kitöltött logikai táblázat, így biztosítható, hogy semmilyen feltétel-kombináció tesztelése nem marad el.

3.9.2. Használati eset teszt

A használati esetek olyan forgatókönyvek, amik a rendszer funkcionalitását írják le különböző felhasználók szemszögéből. Egy használati eset egy adott típusú felhasználó és a rendszer kölcsönhatását írja le. A használati esetek (scenario) jó kiindulási alapot jelentenek a teszteléshez. Minden használati esetnek vannak elő-, és utófeltételei, melyek a helyes

végrehajtás előfeltételét, valamint a helyes végrehajtás után tapasztalható eredményeket, és a rendszer végső állapotát írják le. A használati esetnek általában van fő ága (a legvalószínűbb), és alternatív ágai.

3.9.3. Állapotátmenet teszt

Az állapotátmenet teszt a döntési táblákhoz hasonló technika. Olyan rendszerekhez használatos, ahol a kimenetet a bemeneti feltételekben vagy a rendszer állapotában beálló változás váltja ki. Ez a fajta tesztelés ún. állapot-átmeneti diagramokat, vagy állapottáblákat használ.

3.9.4. Osztályozási fa módszer

A módszer lényege, hogy minden inputhoz rendelünk egy fa-pontot, majd ezt alábontjuk az adott input minden lehetséges értékével (érték-kategóriájával), és ezt egy táblázat fölé helyezzük. Egy-egy levélhez egy oszlopot rendelünk, a sorokhoz pedig a teszteseteink lesznek. Ezután úgy kell meghatározni a teszteseteket, hogy egy tesztesetnél minden input-csomópont oszlopai közül pontosan egy legyen kiválasztva, és az összes teszteset az összes oszlopot lefedje. Az inputok lehetséges értékeinek alábontásakor a módszer kombinálható az említett módszerekkel, például az ekvivalencia partícionálással vagy a határérték analízissel.

3.9.5. Összes-pár tesztelés

A módszer az előzőhöz hasonlóan szintén több input esetén használható, és kombinálható más módszerekkel. Míg az osztályozási fa módszer minden input független „teljes” tesztelését biztosította, addig ez a módszer minden input-pár összes lehetséges kombinációját leteszteli. Ez még mindig lényegesen kisebb szám, mint amennyit az összes input kombináció igényelne, ugyanakkor biztosan felderíti azokat a hibákat is, amik csak két input bizonyos érték-kombinációjában lépnek fel. A módszer lényege, hogy a tesztesetek ügyes megválasztásával az összes pár-kombináció letesztelhető annyi tesztesettel, amennyi a két legszámosabb input kombinációinak teszteléséhez kell.

3.10. Feladatok

1. Adott a következő program specifikáció:

   Írjunk olyan programot, ami két egész számról eldönti, hogy a beolvasás sorrendjében első szám kisebb, nagyobb, vagy egyenlő a másodikkal. Minden számtól különböző input esetén adjon hibaüzenetet a program.

   Írjuk fel a program specifikációhoz tartozó lehetséges inputok halmazát, majd partícionáljuk az inputokat megfelelő részhalmazokra. Adjunk meg egy lehetséges teszteset-halmazt, amivel elvégezve a részfüggvény-tesztelést az összes input-részhalmazt érintjük!

2. Adott a következő program specifikáció:

   Egy italautomata hideg és meleg italokat árul. Ha a meleg italokat választjuk, akkor megkérdezi, hogy kérünk-e bele tejet, majd a választól függetlenül megkérdezi, hogy kérünk-e bele cukrot? Ezután kiadja az italt. Ha hideg italt választunk, akkor csak ki kell választani a megfelelő italt, és az automata kiadja.

   Írjuk fel a program specifikációhoz tartozó lehetséges inputok halmazát, majd partícionáljuk az inputokat megfelelő részhalmazokra. Adjunk meg egy lehetséges teszteset-halmazt, amivel elvégezve a részfüggvény-tesztelést az összes input-részhalmazt érintjük!

3. Adott a következő program specifikáció:

   Írjunk olyan programot, ami egy másodfokú egyenlet együtthatóit megkapva kiszámítja az egyenlet gyökeit. Rossz formátumú input esetén adjon hibaüzenetet a program.

   Írjuk fel a program specifikációhoz tartozó lehetséges inputok halmazát, majd partícionáljuk az inputokat megfelelő részhalmazokra. Adjunk meg egy lehetséges teszteset-halmazt, amivel elvégezve a részfüggvény-tesztelést az összes input-részhalmazt érintjük!

4. Adott a következő program specifikáció:

   Írjunk olyan programot, ami egy 0 és 23 közötti egész számot vár inputként, kimenetként pedig a következőt adja:

   - *Ha a bemenet 0 és 3 között van, akkor a kimenet: „éjszaka”*
   - *Ha a bemenet 4 és 6 között van, akkor a kimenet: „hajnal”*
   - *Ha a bemenet 7 és 9 között van, akkor a kimenet: „reggel”*
   - *Ha a bemenet 10 és 12 között van, akkor a kimenet: „délelőtt”*
   - *Ha a bemenet 13 és 17 között van, akkor a kimenet: „délután”*
   - *Ha a bemenet 18 és 21 között van, akkor a kimenet: „este”*
   - *Ha a bemenet 22 és 23 között van, akkor a kimenet: „éjszaka”*
   - *Minden más input esetén adjon hibaüzenetet.*

   Írjuk fel a program specifikációhoz tartozó lehetséges inputok halmazát, majd partícionáljuk az inputot megfelelő részhalmazokra. Adjunk meg egy lehetséges teszteset-halmazt, amivel elvégezve a részfüggvény-tesztelést az összes input-részhalmazt érintjük!

5. Rajzoljunk fel két változó által meghatározott input teret koordinátarendszerben (a tengelyek a változók értékkészletét jelentsék), és jelöljünk be rajta a határértékeket. Készítsünk külön ábrát a két- és hárompontos változathoz. Jelöljük be a kitüntettet értékek közül azokat, amelyek részt vesznek az összes értéket lefedő tesztelésben illetve az összes-pár tesztelésben.

6. A határérték analízis használatával adjunk meg teszteseteket a következő program specifikáció alapján:

   Írjunk egy programot, ami a Newton-módszer szerint kiszámítja egy valós szám köbgyökét. Legyen q kezdetben 1.0, a számításhoz pedig a (2.0 * q + r / (q * q)) / 3.0 képletet használjuk, ahol r a valós input. Ha a kifejezés értéke egy meghatározott pontosság szerint majdnem egyenlő q-val, akkor a kiszámított érték lesz r köbgyöke. Különben legyen q a kifejezés értéke, és számítsuk ki újra a képletet.

7. Egy ország vasútjain a következő kedvezményeket lehet érvényesíteni életkor szerint:
   - 0-6 éves korig 100%
   - 6-14 éves korig 50%
   - 14-26 éves korig: 33%
   - 55-65 éves korig: 20%
   - 65-99 éves korig: 100%

   Ezen felül 14-30 éves kor között diák kedvezmény is igénybe vehető, ami 50% kedvezményt jelent.
   Határérték analízis segítségével határozzunk meg teszteseteket a fenti specifikáció alapján.

4. Egyéb módszerek

Ebben a fejezetben néhány egyéb módszert ismerhetünk meg. Ezek nem mindegyike kifejezetten teszt tervezési technika, így nem sorolható be az előzőekhez. Vannak közöttük segédtechnikák programmegértéshez (szeletelés), de teszt szervezési módszerek is (priorizálás).

4.1. Statikus analízis

Statikus analízis során a számítástudományban gyakran alkalmazott „absztrakciót” használunk. Ez valamilyen közös tulajdonság alapján egy magasabb szintre történő elvonatkoztatást jelent az adott problémakörtől (Például amikor ciklusokat használunk, akkor egy ismétlődő szekvenciális lépéssorozaton alkalmazunk absztrakciót, vagy amikor osztályokat használunk, akkor egy közös tulajdonság halmaz alapján „emeljük ki” az osztályt). Az egyik legnehezebb feladat, hogy megtaláljuk a helyes egyensúlyt a megfelelő mértékű absztrakcióhoz.

A statikus analizátorok absztrakció használatával, a programok forráskódjából, de azok futtatása nélkül elemezik a forráskódot, és olyan potenciális programozási hibákat keresnek bennük, amiket futási időben nagyon nehéz lenne észrevenni illetve pontosan beazonosítani. Ráadásul az így megtalált hibák a korábbi detektálás révén összességében olcsóbban javíthatók.

A korai időszakban a statikus eszközök alig tudtak többet egy szimpla mintaillesztésnél, ami arra volt jó, hogy a programozó rákereshetett olyan függvényekre, amik köztudottan kerülendők voltak. Ezt követően megjelentek, és alkalmazásra kerültek a kóddal összekapcsolható metrikák, például a kódméret, komplexitás, ciklomatikus komplexitás, stb. Ezek segítségével már jobban felmérhetővé vált a kód bonyolultsága, adott esetben a kód átírására is késztethette a programozókat. A következő lépés az volt, hogy a statikus eszközök kontextus-függő keresést (vagyis egyfajta absztrakciót) kezdtek el alkalmazni. Manapság a legkifinomultabb eszközök absztrakt szintaxis-fákkal támogatják a programozókat.

4.1.1. Mikor használjuk?

A statikus eszközök nagy előnye, hogy a hibát a fejlesztési folyamat korai fázisában találják meg, amikor még alacsony a javítás költsége (minél később találunk meg egy hibát a fejlesztési folyamatban, annál költségesebb a javítása). A kifinomultabb eszközök azt is képesek megmutatni, hogy a kapott hiba milyen útvonalon következik be. Statikus kódelemzést lehet statikus tesztelésre is használni, de máshol is alkalmazzák azokat.

A statikus eszközök használata minden fejlesztési területen javasolt: a fejlesztők egy egyszerűbb statikus analízissel már korán feltérképezhetik a lehetséges hibaforrásokat, és könnyebben betarthatják a kódolási standardokat; a buildelésért felelős csapat már komplexebb hibákat is kereshet az eszközökkel (pl. integrációs hibák); a tesztelők a

bonyolult kódrészletek felderítésével és a kód lefedettség mérésével tudnak hasznot kovácsolni a statikus eszközökből.

A mai statikus eszközök a szabálysértések bő halmazát képesek felismerni, ezért mindig körültekintően érdemes választanunk, annak függvényében, hogy az adott eszköz mennyire felel meg a mi céljainknak. Projekttől függően a választásnál mérlegelendő szempontok lehetnek például, hogy olyan szabálysértések vannak-e az eszközben, amik hasznosak lehetnek számunkra; testreszabhatóak-e a szabályok; fel lehet-e venni új szabálysértéseket; stb.

A statikus eszközök az alábbi tipikus ellenőrzéseket végzik (könnyítik meg):

- Típus ellenőrzés
- Stílus ellenőrzés
- Program érthetőségi elemzés
- Program verifikáció
- Hibakeresés
- Biztonsági felülvizsgálat

Ezek közül példának tekintsük a program verifikációt, amely során a program specifikáció alapján készítünk, majd ellenőrzünk szabályokat. Az ilyen jellegű vizsgálatoknál olyan szabályokat definiálunk, ami azt határozza meg, hogy milyen lépéssorozatot *nem* szabad a programnak végrehajtania. Például: „Egy memória címre nem lehet hivatkozni, ha azt már korábban felszabadítottuk”. Vannak olyan eszközök, amelyek ha találnak egy ilyen jellegű hibát, akkor szemléltetik is, hogy milyen potenciális lépéssorozat vezethet el az adott szabály megsértéséhez. Az alábbiakban erre láthatunk egy példát. Tekintsük az alábbi C kódot:

```
1       inBuf = (char*) malloc(bufSz);
2       if (inBuf == NULL)
3           return -1;
4       outBuf = (char*) malloc(bufSz);
5       if (outBuf == NULL)
6           return -1;
```

Látható, hogy ha az első memóriafoglalás sikeres, de a második nem, akkor memóriaszivárgás („memory leak”) keletkezik. Ha a statikus eszköz ezt észleli, az alábbi üzenettel segíthet a probléma megértésében:

```
Violation of property "allocated memory should always be freed":
line 2: inBuf != NULL
line 5: outBuf == NULL
line 6: function returns (-1) without freeing inBuf
```

4.1.2. Hátrányok

A statikus eszközök legnagyobb hátránya, hogy rossz konfigurálás esetén nagyon sok lehet a hamis riasztás (false positive), vagyis azon esetek száma, amikor az eszköz lehetséges hibát jelez – tévesen. A nagyszámú hamis riasztásnak nem csak az lehet a hátránya, hogy megkérdőjelezi az eszköz létjogosultságát, hanem az is, hogy a validálás közben a programozó figyelme elsiklik fontosabb (jogos) hibák felett.

A tévedések másik fajtája az, amikor a statikus eszköz nem vesz észre egy létező hibát (false negative). Az ilyen jellegű hibák sokkal költségesebbek, hiszen egyrészt hamis biztonságérzetet kelthetnek bennünk, másrészt, ha a hiba a fejlesztési folyamat későbbi szakaszában mégis előjön, akkor sokkal költségesebb lesz a javítása.

Az ilyen esetek kiküszöbölésére napjaink statikus eszközei már rendelkeznek azzal a képességgel, hogy lehet finomítani a szabályokat (ami alapján észlelik a hibákat). Ha úgy gondoljuk, hogy egy adott típusú hibára az eszköz 99%-ban tévesen jelez, akkor azt a típusú figyelmeztetést ki lehet venni az elemzésből. Fontos megtalálni az egyensúlyt, mert ha indokolatlanul magas számú hibatípust veszünk ki, akkor könnyen megnövekedhet a false negative jelzések száma. Ebben segíthet a hibák kategorizálása. A hibákat többféle szempont alapján csoportosíthatjuk: léteznek kódméretre, tervezési hiányosságokra, elnevezési konvenciókra, nem használt kódméretre, és számos egyéb területre specializált szabályok. Ezek után első körben elegendő azt meghatározni, hogy mely típusú hibákat szeretnénk statikus elemzéssel felderíteni.

4.1.3. Példa

Az alábbiakban a PMD elemzőt fogjuk használni. A PMD egy statikus elemző eszköz (lásd 8. Felhasznált irodalom és további olvasmány), ami számos JAVA kódolási problémára hívja fel a figyelmünket. Több száz szabálysértés típus közül válogathatunk, melyek kategóriákba vannak osztva. Tekintsünk először egy egyszerű példát (a most következő példákat a PMD honlapjáról idézzük):

```
public void doSomething() {
   try {
      FileInputStream fis = new FileInputStream("/tmp/bugger");
   } catch (IOException ioe) {

   }
}
```

Erre a PMD elemzője egy ún. „Empty Catch Block” figyelmeztetést fog adni, ami azt jelzi, hogy valahol egy kivételt elkapunk, de azután nem csinálunk vele semmit. Lehet, hogy ez volt az eredeti szándékunk, de legtöbbször az ilyen eseteket kerülni kell, mert ez nem megfelelő kivételkezeléshez vezethet.

Nézzünk példát egy másik kategóriából:

```
public class OneReturnOnly1 {
   public void foo(int x) {
      if (x > 0) {
              return "hey";
      }
      return "hi";
   }
}
```

Látható, hogy a foo függvénynek két kilépési pontja van, ami potenciálisan nem helyes, ezért a PMD elemző erre egy „Only One Return” jelzést fog adni.

A következő példa bemutatja, hogy a PMD elemző képes az egyszerűbb data-flow anomáliák jelzésére is:

```
public class Foo {
   public void foo() {
      int buz = 5;
      buz = 6;
   }
}
```

A fenti kódrészletben könnyen azonosítható egy dd-anomália a buz változóra nézve. A PMD elemző ezt észre is veszi, és „Dataflow Anomaly Analysis” szabálysértést fog jelezni ezen a kódrészleten.

4.2. Szeletelés

A szeletelés célja meghatározni a program azon utasításait, amelyek valamilyen tesztelési kritérium alapján kapcsolatban állnak a kritériumban meghatározott változó értékével. A szeleteket a CFG alapján számolt program reprezentáció segítségével tudjuk meghatározni.

Térjünk tehát vissza egy kicsit ismét a program vezérlési folyamatához. A korábban megismert módszerek alapján készítsünk vezérlési folyam gráfot a kód alapján, ahol minden csomópont egy program-utasításnak felel meg.

Miután a vezérlési gráfot felrajzoltuk, következtetéseket vonhatunk le a program utasításokra vonatkozóan. Hogyan befolyásolhatja a program menetét egy utasítás?

- Megváltoztathatja a program állapotát (például értéket ad egy változónak).
- Meghatározhatja, hogy melyik utasítást hajtsuk végre a következő lépésben.

Az utasításokat befolyásolhatják korábbi utasítások, ezáltal függőséget hoznak létre. Ha egy utasítás kiolvassa egy változó értékét, akkor a változó korábbi értékadásai befolyással vannak az olvasás eredményére. Hasonlóképpen, egy utasítás lefutása függhet egy korábbi utasítástól. Ezek alapján az utasítások között megkülönböztetünk adat-függőséget és vezérlési függőséget.

- Adat függőség: Egy B utasítás adat függőségben áll A-val, ha A módosítja valamely V változó értékét, amelyet B kiolvas, és a vezérlési gráfban van legalább egy olyan útvonal A-ból B-be, melynek során V-t nem módosítja semmilyen másik utasítás.
- Vezérlési függőség: Egy B utasítás vezérlési függőségben áll A-val, ha B lefutását A vezérli (vagyis A határozza meg, hogy B lefut-e).

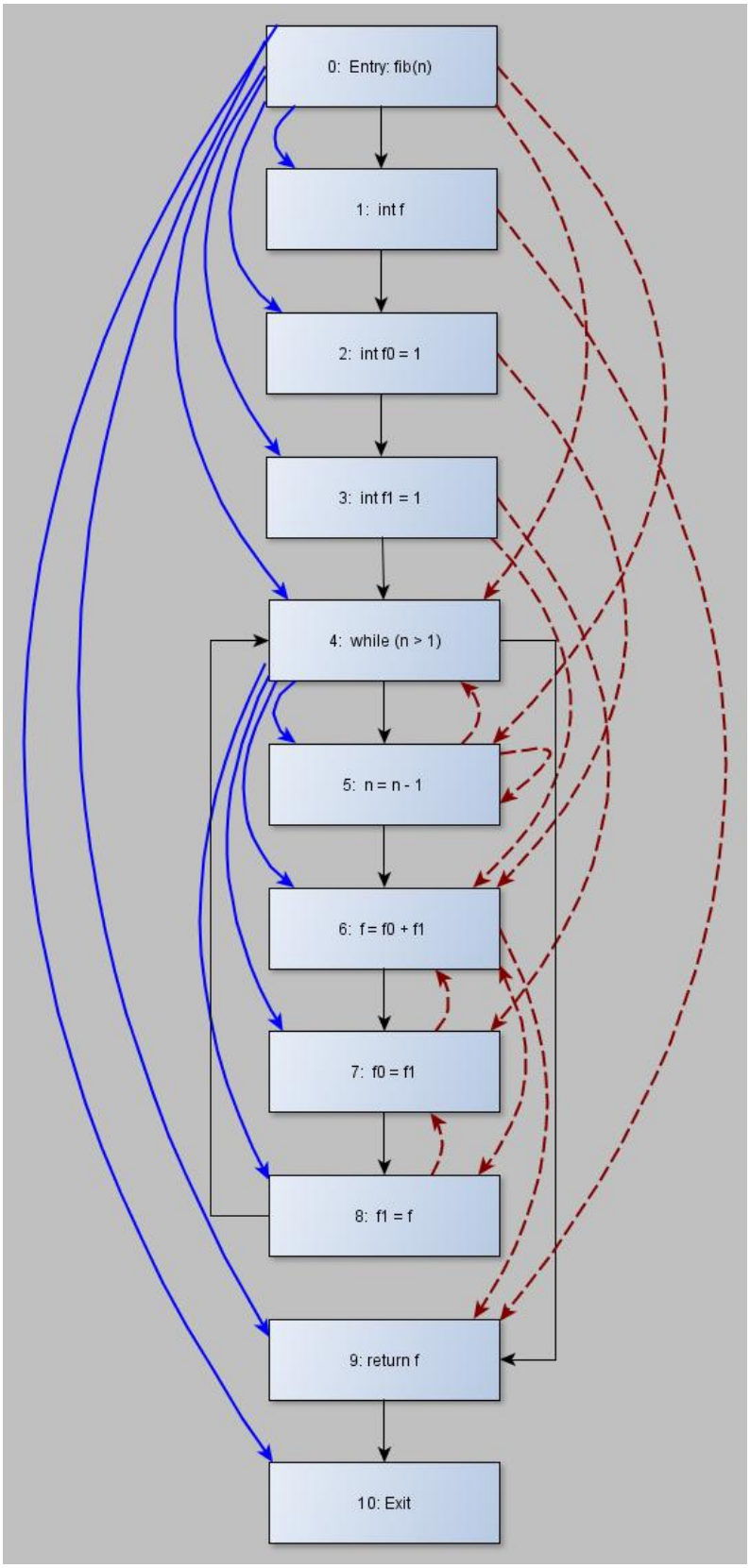
Ezek alapján megkonstruálható egy függőségi gráf (Program Dependency Graph – PDG). Az alábbiakban erre láthatunk példát.

4.2.1. Példa

Az alábbiakban láthatunk egy rövid C nyelven megírt függvényt:

```
int fib(int n)
{
     int f, f0 = 1, f1 = 1;
     while (n > 1) {
          n = n - 1;
          f = f0 + f1;
          f0 = f1;
          f1 = f;
     }
     return f;
}
```

A fenti kódhoz tartozó vezérlési gráf látható az alábbi ábrán, melybe már berajzoltuk a függőségeket is (12. ábra).



12. ábra: Függőségek - piros: adat függőség; kék: vezérlési függőség

4.2.2. Szeletelés

A szeletelés nem más, mint egy gráfbejárás a függőségi gráfon. Bár más módszerek is léteznek (lásd 8. Felhasznált irodalom és további olvasmány), ez a legelterjedtebb. Az előző függőségek követésével program-szeleteket definiálhatunk, a következőképpen:

$$S^F(A) = \{B \mid A \rightarrow^+ B\}$$

$$S^B(B) = \{A \mid A \rightarrow^* B\}$$

A fenti képletek jelentése a következő:

$S^F(A)$ - Induljunk ki az A utasításból, és jegyezzünk fel minden olyan utasítást, amit A befolyásolhat. Ezt hívjuk előre-szeletelésnek.

$S^B(B)$ – Induljunk ki egy B utasításból, és visszafele haladva határozzuk meg azokat az utasításokat, amik befolyásolhatják B-t. Ezt hátra szeletelésnek hívjuk.

A szeleteléseknek további típusait különböztethetjük meg:

- chop: Egy előre-, és egy hátra szeletelés metszete.
- intersection: Két tetszőleges irányú szeletelés metszete. Gyakori viselkedés megfigyelésére használható.
- dice: Két szeletelés közti eltérést mutatja meg. Különböző viselkedés megfigyelésére.

A szeletelésnek sok alkalmazása van, mint például a programmegértés, dekompozíció vagy hibakeresés. A teszteléshez kapcsolódóan a szeletelés segítségével különféle programhibákat találhatunk meg.

- Nem inicializált változóból olvasás.
- Nem használt változó. (Ez a függőségi gráfban úgy jelenhet meg, hogy a változóba való írást végző utasításoktól nem függ egyik további utasítás sem – adatfüggés szerint.)
- Nem elérhető („halott”) kód. (Olyan utasítások, amik nem függnek semmilyen korábbi utasítástól – vezérlési függés szerint.)
- Memória szivárgás.
- Rosszul használt interfészek.
- Null pointerek.

4.2.2.1. Példa

Tekintsük példaként a korábbi C függvényből készült vezérlési gráfot (12. ábra). Határozzuk meg, hogy a 2-es utasítás előre szeletelésekor mely utasítások kerülnek bele az eredményhalmazba!

1. A 2-es utasítás: `f0 = 1` először a 6-os utasítást éri el: `f = f0 + f1`
2. Az `f`-en keresztül elérjük a 8-as és a 9-es utasításokat: `f1 = f`, valamint `return f`
3. Az `f1`-en keresztül a 7-es utasítást érjük el: `f0 = f1`
4. A teljes előre szelet tehát: $S^F(2) = \{2, 6, 7, 8, 9\}$

Hasonlóképpen kiszámítható, hogy $S^B(9) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

4.2.3. Dinamikus szeletelés

Láttuk tehát, hogy képesek vagyunk a forráskódból statikusan meghatározni, hogy mely utasítások függnek korábbi utasításoktól. A szeletelési módszernek azonban létezik dinamikus változata is, amely egy konkrét futás szeletelését végzi. A dinamikus

szeleteléshez először elő kell állítani az ún. „trace”-t, vagyis azt az utasításokat tartalmazó listát, ami a végrehajtás sorrendjében tartalmazza az utasításokat.

A dinamikus szeletelés algoritmusa ezután a következő:

1. Haladjunk a trace-n. Minden egyes *w* változóba íráskor definiáljunk egy üres dinamikus szeletet:

$$\mathtt{DynSlice}(w) = \emptyset$$

2. Folytassuk a trace feldolgozását. Ha egy *w* értékbe írnak, nézzük meg azokat a változókat (r_i), amiket abban az utasításban olvasnak. Minden egyes r_i-re nézzük meg azt a sort, ahol r_i-be legutoljára írtunk, legyen ez line(r_i)-nek, valamint a DynSlice(r_i) halmazt. Vegyük az így kapott sorok és szeletek unióját, és ami a DynSlice(*w*) értéke lesz:

$$\mathtt{DynSlice}(w) = \bigcup_i (\mathtt{DynSlice}(r_i) \cup \{\mathtt{line}(r_i)\})$$

Általánosságban elmondható, hogy a dinamikus szeletelés sokkal pontosabb, mint a statikus szeletelés, bár a trace-t előállítani sokszor körülményes lehet. A pontosság abból adódik, hogy míg a statikus szeletelés a program összes lehetséges lefutásából adódó függőséget figyelembe kell, hogy vegye, addig a dinamikus szeletelés mindig pontosan egy lefutás éppen aktuálisan megvalósuló függőségeivel dolgozik. Ennek kapcsán szokás uniós szeletelésről és realizálható szeletről beszélni. Az uniós szeletelés elve, hogy az ugyanarra a programsorra több eltérő futás eredményeként kapott dinamikus szeletek unióját vesszük. Elméletben, ha az utasításra az összes lehetséges lefutás dinamikus szeletét uniózzuk, akkor megkapjuk a realizálhatónak nevezett szeletet. A statikus szeletelés ezt „felülről”, konzervatív módon közelíti (azaz inkább bővebb, de nem hagy ki semmit).

A szeletelés segítségével hibákat (fertőzött helyeket) lehet visszakeresni a kódban, az alábbi módszer szerint:

1. A kiindulópont legyen az a hibás érték, ami a meghibásodás (failure) során előjött.
2. Kövessük végig a függőségeket, lehetséges ősök után kutatva. Ezt mind statikus, mind dinamikus szeleteléssel megtehetjük.
3. Vizsgáljuk meg az ősöket, és döntsük el, hogy ezek fertőzöttek-e vagy sem.
4. Ha a vizsgált ős fertőzött, akkor ismételjük meg a 2-es és 3-as lépéseket erre az értékre.
5. Ha találunk egy olyan fertőzött értéket, melynek minden őse tiszta (sane), akkor találtunk egy fertőzési helyet – vagyis egy defektust.
6. Javítsuk ki a defektust, és ellenőrizzük, hogy a hiba előjön-e. Ezzel megbizonyosodhatunk arról, hogy valóban a kijavított defektus okozta-e a hibát.

Gyakorlatban a vizsgálandó adat nagy mennyisége miatt a megfigyelés egyes fázisait automatizáljuk. Az egyik ilyen mód ellenőrzések (assertion-ök) beépítése a kódba, amik segítenek eldönteni egy állapotról (értékről), hogy helyes-e.

Az ilyen jellegű ellenőrzések többféle vizsgálatot is lehetővé tehetnek:

- konstansok: olyan vizsgálatok, melyek arról bizonyosodnak meg, hogy egy érték végig állandó marad a futás során. Ez magában foglalja azt is, hogy egy érték végig egy adott korlát között marad.
- elő- és utófeltételek: olyan ellenőrzések, amik arra vonatkoznak, hogy egy függvényhívás előtt/után milyen feltételeknek kell teljesülni.

A fent bemutatott dinamikus szeletelő algoritmus futására láthatunk egy példát az alábbiakban.

4.2.3.1. Példa

Nézzük meg az alábbi C programkódot, és határozzuk meg a DynSlice(4) értéket.

```
1       n = read();
2       a = read();
3       x = 1;
4       b = a + x;
```

A 4. sorban az *a* és *x* változókból olvasunk ki. Ezen változókat korábban rendre a 2. illetve a 3. sorban módosítottuk, ezért a 4. sor dinamikus szelete az alábbi három sor uniója:

- a 2. sorban lévő *a* változó dinamikus szelete (üres)
- a 3. sorban lévő *x* változó dinamikus szelete (üres)
- a 2. és a 3. sor

A fentiekből következik, hogy:

$$DynSlice(4) = \{2, 3\}$$

4.3. Teszt priorizálás, teszt szelekció

A most következő két technika inkább szervezési módszerként használatos, több elemi módszert magukban foglalhatnak.

4.3.1. Teszt priorizálás

A teszt-eset priorizálás technikája egy olyan módszer, amit akkor érdemes használni, ha az a célunk, hogy egy bizonyos szempontból fontosabb tesztesetek fussanak le először egy tesztkör (vagy egy regressziós tesztelés) során. Ilyen szempontok lehetnek például:

- Minél korábban hozzuk elő a hibákat.
- Minél gyorsabban érjünk el egy előre megadott kód lefedettségi szintet.
- Minél gyorsabban alakuljon ki megbízhatóság-érzet a rendszer iránt.
- Minél gyorsabban találjunk meg magas kockázatú hibákat.
- Minél hamarabb találjuk meg a kódban bekövetkezett változások okozta esetleges hibákat.

A tesztesetek priorizálása során nem marad ki egyetlen egy teszteset sem a tesztelésből (vesd össze: teszt szelekció – lásd később), így a teszt szelekció esetleges hátrányai kiküszöbölhetőek. (Mindazonáltal ha a tesztelés során megengedett bizonyos tesztesetek elhagyása, akkor a teszteset priorizálás jól használható közösen a teszt szelekciós módszerekkel). A priorizálás egyik nagy előnye, hogy ha valami miatt nem jut elegendő erőforrás a tesztelésre, vagy nem annyi jut, mint amennyit előre terveztünk, akkor azok a tesztesetek fognak először lefutni, amik fontosak. (Ellenkező esetben, ha nem priorizáljuk őket, akkor könnyen teszteletlen maradhat a rendszer egy-két kulcsfontosságú része).

Formálisan, a teszteset priorizációt a következőképpen közelíthetjük meg:

> Legyen T egy teszteset halmaz, PT pedig T-beli permutációk egy halmaza. Legyen ezen felül f egy a PT halmazból a valós számokra képzett függvény. Keressük meg azt a $T' \in PT$ úgy, hogy $(\forall T'')\ (T'' \in PT)\ (T'' \neq T')\ [f(T') \geq f(T'')]$

A fenti definícióban PT-t úgy értelmezhetjük, mint T összes lehetséges priorizálásának halmazát, f-et pedig felfoghatjuk úgy, hogy minden priorizáláshoz hozzárendel egy értéket, attól függően, hogy mennyire „jó” az adott priorizálás. Itt a „jóságot” olyan értelemben definiáljuk, hogy mennyire felel meg a priorizálás céljának. Feltesszük, hogy minél nagyobb ez az érték, annál „jobb” az adott priorizálás.

A továbbiakban a priorizálás céljaként a minél korábbi hibafelfedezést fogjuk tekinteni. (Ez megegyezik a fejezet elején ismertetett célok közül az elsővel. Megjegyezzük, hogy gyakorlatban általában ennél egyszerűbb céljaink vannak, pl: minél nagyobb lefedettség minél korábbi elérése.) Másképp megfogalmazva a célunk az, hogy növeljük a teszteset halmaz hiba-felderítő képességét abból a szempontból, hogy minél hamarabb fedezzük fel a hibákat. Ennek a megközelítésnek az az előnye, hogy a fejlesztők hamarabb elkezdhetik a felderített hibák javítását, valamint hamarabb kapunk visszajelzést a szoftver állapotáról.

Felmerülhet a kérdés, hogy hogyan tudjuk mérni egy priorizálás hatékonyságát? Erre a következő mértéket találták ki:

APFD – weighted Average of the Percentage of Faults Detected

Vagyis a megtalált hibák százalékának súlyozott átlaga. Ezt az algoritmus futása közben számolják, és minél magasabb egy ilyen érték, annál jobb egy priorizálási algoritmus.

A metrika illusztrálására tekintsünk egy 10 hibát tartalmazó programot, melyhez 5 darab teszteset tartozik. A tesztesetek az alábbi táblázat szerint hozzák elő a hibákat:

teszteset/hiba	1	2	3	4	5	6	7	8	9	10
A	X				X					
B	X				X	X	X			
C	X	X	X	X	X	X	X			
D					X					
E								X	X	X

Tegyük fel, hogy először **A-B-C-D-E** sorrendbe rakjuk a teszteseteket. Ekkor a teszt halmaz futtatása során az egyes tesztesetek végrehajtását követően az alábbi hiba lefedettségi értékeket kapjuk:

[20, 40, 70, 70, 100] => Ebből az APFD érték: 50.

(Egy kis magyarázat: az első értéket úgy kapjuk, hogy megnézzük, hogy az összes hiba hány százalékát fedte le az első lefuttatott teszteset: 2/10 = 20%. A következő lépésben azt vizsgáljuk, hogy a második lefuttatott teszteset hány új hibát érintett, és ezt a százalékot hozzáadjuk az eddigi értékhez, így kapjuk a 40%-os második értéket. Ezt a módszert követve kapjuk meg a fenti számsorozatot.)

Változtassuk most meg a tesztesetek priorizálását úgy, hogy a teszt eseteket **E-D-C-B-A** sorrendben futtatjuk le. Ekkor így módosulnak a számok:

[30, 30, 100, 100, 100] => Ebből az APFD érték: 64.

Nyilvánvaló cél, hogy minél hamarabb érjük el a 100-as határt, hiszen ekkor teljesül az, hogy viszonylag rövid idő alatt felfedeztük a hibák 100%-át.

Látható, hogy a fenti feladatban az optimális teszteset priorizálás: **C-E-B-A-D,** hiszen ekkor két lépés alatt elérjük a 100%-os hiba felderítettséget:

[70, 100, 100, 100, 100] => Ebből az APFD érték: 84.

A fenti célok alapján több különböző priorizálási algoritmust különböztethetünk meg:

1. **Nincs priorizálás** – A teljesség kedvéért belevesszük azt az esetet, amikor nem priorizálunk, de külön nem foglalkozunk ezzel az eshetőséggel.
2. **Véletlenszerű priorizálás** – Az összehasonlítás miatt vesszük fel azt az esetet, amikor véletlenszerűen rakjuk sorba a teszteseteket.
3. **Optimális priorizálás** – Annak érdekében, hogy mérni tudjuk a priorizálások hatékonyságát olyan programokat fogunk venni, amikben tudjuk, hogy hol és hány darab hiba van: egy adott P program esetén, ha tudjuk a benne lévő hibák egy halmazát, és meg tudjuk határozni, hogy a T teszteset halmazból melyik teszteset melyik hibát „fedi le" (hozza elő), akkor meg tudjuk határozni egy optimális priorizálását T-nek.

   Természetesen gyakorlatban ez a módszer nem kivitelezhető, mert „a priori" tudást nem tudunk a hibák létéről. Ennek ellenére érdemes összehasonlítani a többi heurisztikus algoritmus eredményességével.
   Probléma, hogy legrosszabb esetben exponenciális idejű futási időt eredményez a legjobb olyan algoritmus, ami bármelyik esetben meghatározza az optimális priorizálást. Emiatt létezik ennek a megközelítésnek egy „mohó" változata, amikor is mindig azt a tesztesetet választjuk ki, ami a legtöbb – még „le nem fedett" – hibát hozza elő. Ezt a lépést iteráljuk addig, amíg az összes hibát le nem fedtük (a maradék teszteseteket „tetszés szerint" rendezzük). Könnyen belátható, hogy előfordulhat olyan eset, amikor ez a „mohó" algoritmus nem az optimális megoldást találja meg. Ennek ellenére ez a módszer megfelelő felső korlát lehet abban a tekintetben, hogy egy versenyképes algoritmusnak ennél a megoldásnál nem szabad rosszabbat találnia.
4. **Teljes utasítás lefedettséget eredményező priorizálás** – Korábban a lefedettségi módszerek vizsgálatánál láttuk, hogy ha instrumentáljuk a kódot, akkor mérhetjük az egy-egy teszteset által lefedett utasításokat. Ezek alapján a teszt-eseteket priorizálhatjuk aszerint, hogy hány utasítást fednek le; amelyik többet, az kerül előrébb a prioritási sorban.

   Példaként tekintsük az alábbi pszeudo kódrészletet:

```
1  s1
2  while (c1) do
3  if (c2) then
4        exit
         else
5        s3
         endif
6  s4
   endwhile
7  s5
8  s6
9  s7
```


A fenti pszeudo-kódban `s1, s2, …, s7` az 1-es, 2-es, …, 7-es számú utasításokat (statement), míg `c1` és `c2` pedig feltételek (condition) jelöl. Tegyük fel, hogy van 3 tesztesetünk, amik rendre az {1; 2; 7; 8; 9} – {1; 2; 3; 4} – {1; 2; 3; 5; 6; 7; 8; 9} sorokat fedik le. Ez esetben a tesztesetek priorizálási sorrendje: 3 – 1 – 2

Ami a fenti eljárás hatékonyságát illeti: Vegyünk egy teszt eset halmazt, aminek az elemszáma `m`, és egy programot, ami `n` darab utasításból áll. Ekkor az imént ismertetett priorizálási eljárás `O(m*n + m*logm)` idő alatt elvégezhető. Mivel gyakorlatban `n` értéke jóval nagyobb, mint `m`, ezért a priorizálás elvégezhető `O(m*n)` idő alatt.

5. **Kiegészítő utasítás lefedettséget eredményező priorizálás** – Az előző módszer hátránya, hogy a priorizálási folyamat során könnyen kiválaszthatunk olyan tesztesetet, ami már korábbi tesztesetekkel lefedett utasításokat tesztel le. (Az előző pontban bemutatott példánál látható, hogy az 1-es számú teszt-eset beválasztása semmivel nem javította a lefedettséget, mert a 3-as számú teszt-eset már lefedett minden olyan utasítást, amit az 1-es számú teszteset érintett.)

   Ezért az előző algoritmust módosítsuk úgy, hogy mindig azt a teszt-esetet választjuk a prioritási sor következő elemének, ami a legnagyobb mértékben növeli a lefedettséget. Az előző példában így a 3-as teszteset kiválasztása után a 2-es fog következni, hiszen így 100%-os utasítás lefedettség érhető el.
6. **Teljes branch lefedettséget eredményező priorizálás** – Ez a fajta módszer megegyezik a 4-es pontban említett technikával, azzal a különbséggel, hogy utasítás-szintű lefedettség helyett branch lefedettség szerint rendezi sorba a teszteseteket.
7. **Kiegészítő branch lefedettséget eredményező priorizálás** – Megegyezik az 5-ös pontban említett technikával, azzal a különbséggel, hogy utasítás-szintű lefedettség helyett branch-lefedettség szerint rendezi sorba a teszteseteket.

A fent bemutatott módszerek hiba felfedező képességéről empirikus tanulmányok megmutatták, hogy még a véletlenszerű priorizálás is jobban teljesít, mint ha egyáltalán nem priorizálunk. A branch lefedettségen alapuló priorizálás pedig majdnem minden esetben legalább olyan jól teljesít, mint az utasítás lefedettségen alapuló priorizálás.

4.3.2. Teszt-szelekció

A teszt-szelekciós módszerek létjogosultságát az adja, hogy különösen regressziós tesztelés esetében nagyon költséges lehet az összes tesztesetet mindig lefuttatni. Az sem túl nagy segítség, ha véletlenszerűen kiválasztjuk egy részhalmazát a lefuttatandó teszteseteknek, mert ez gyakran nem megbízható. Ezáltal eljutunk a következő problémához:

Adott tesztesetek egy T halmaza, ahol $T = \{t_1, t_2, t_3, \ldots t_N\}$, N darab tesztesetből álló halmaz. Ezt az N darab tesztesetet használjuk a fejlesztési folyamat során regressziós tesztelés céljából. A kérdés az, hogy a program megváltozása után milyen módon válasszunk ki egy R részhalmazt a T halmazból annak érdekében, hogy az így kiválasztott tesztesetek lefuttatása után nagy biztonsággal meggyőződjünk, hogy a változtatás nem okozott nem várt működést a programban.

A fenti problémára több vázlatos algoritmust is ismertetünk.

1. **Szimulált hűtés („Simulated Annealing”) módszere** – Ennél a megközelítésnél minden egyes lehetséges megoldás egy konfiguráció formájában kerül felírásra a következő módon: $[X_1, X_2, X_3, \ldots, X_N]$, ahol $X_i = 0$, ha az adott tesztesetet nem választjuk be, míg $X_i = 1$, ha beválasztjuk a tesztelendő halmazba. Minden konfigurációhoz definiálunk egy ún. „energia értéket”, amit Z-vel jelölünk: $Z = c_1X_1 + c_2X_2 + c_3X_3 \ldots + c_NX_N$, ahol c_i egy súly, amit a tesztelő határoz meg annak tükrében, hogy mennyire fontos egy teszteset.

   A szimulált hűtés algoritmus egy kezdeti magasabb hőmérséklet fokozatos csökkentésével egy lokális optimum megoldást talál.
2. **Redukciós módszer** – Az algoritmus előfeltétele, hogy a program követelményei össze legyenek rendelve a tesztesetekkel, vagyis adott egy $\{r_1, r_2, r_3, \ldots r_n\}$ követelményhalmaz, és $T_1, T_2, T_3, \ldots T_n$ részhalmazai T-nek úgy, hogy a T_i-ben lévő tesztesetek lefedik az r_i követelményt. Ezek után, ha egy R követelményhez szeretnénk a teszteseteket meghatározni, akkor első körben azokat a teszteseteket vesszük be, amik egyelemű T_i részhalmazba tartoznak. Ezeket a T_i-ket megjelöljük, majd a még meg nem jelölt T_i-kre tovább folytatjuk az algoritmust sorban a két-, három-, stb. elemű T_i-kre.
3. **Szeletelés módszere** – A korábbiakban ismertetett szeletelés módszerét is használhatjuk teszt szelekcióra, mégpedig a következő megfigyelések alapján:
   - Egy teszteset nem érinti az összes utasítását a programnak.
   - Ha egy teszteset nem érint egy utasítást, akkor a teszteset lefuttatása során az az utasítás nem befolyásolhatja a program kimenetét.
   - Ha egy teszteset érint egy utasítást, még akkor sem biztos, hogy az adott utasítás befolyásolja a program kimenetét a tényleges futtatás során.
   - Egy utasítás nem biztos, hogy befolyásolja a program teljes kimenetét (lehet, hogy csak egy részét).

   Ezen megfigyelések segítségével dinamikus szeleteket határozhatunk meg a tesztesetekhez, mégpedig úgy, hogy egy szelet olyan utasításokból fog állni, amelyeket az adott teszteset a futtatása során érint, és amelyek befolyásolják a program kimenetét. Így ez az algoritmus olyan teszteseteket fog kiválasztani, amelyek dinamikus szelete tartalmazza a módosított programrészt.
4. **Adatfolyam módszere** – Ez az algoritmus szintén szeletelést használ, de első lépésként meghatározza a definíció-használat párokat, mégpedig azokat, amiket a megváltozott program befolyásol. Egy program szelet itt olyan utasításokból áll, amik befolyásolhatják bizonyos változók értékét a megváltoztatott utasításokban. A kiválasztott tesztesetek itt azok lesznek, amelyek lefedik ezeket a szeleteket, vagyis amik letesztelik ezeket a megváltozott definíció-használat párokat.
5. **„Firewall” módszer** – A firewall módszer a hatásanalízisen alapszik. A hatásanalízis fő kérdése az, hogy egy programelem megváltoztatása esetén mely más programelemeket kell megváltoztatni (de legalábbis átnézni) ahhoz, hogy a program továbbra is helyesen működjön. A hatásanalízis annyiban hasonlít a szeletelésre, hogy az egymásra kiható programrészeket kapcsolja

össze. Viszont a szeleteléssel szemben általában magasabb szintű programelemekkel dolgozik (metódusok, osztályok) ezáltal kevésbé pontos, viszont könnyebben számolható, továbbá nem csak adat és vezérlési, hanem egyéb potenciális függőségeket is figyelembe vehet, és mivel a változás hatása kölcsönös, nincs kitüntetett iránya.

A firewall módszer lényege, hogy meghatározzuk a megváltoztatott programelemeket, majd hatásanalízissel ezek közvetlen szomszédait. Ezek után azokat a teszteseteket választjuk ki, amelyek a futás során érintik az előbb meghatározott programelemeket.

4.4. Programhelyesség-bizonyítás

A mindennapjainkban használt programok többsége általában tartalmaz hibákat. Vagy rejtett hibákat, vagy olyanokat, amik csak kellemetlenséget okoznak, esetleg olyanokat, amelyek napvilágra kerülése súlyosabb következményekkel járnak, például egy támadható biztonsági rés. Számítógépek vezérlik többek között az olyan kritikus rendszereket is, melyek leállása, meghibásodása még az előzőekben említett problémáknál is súlyosabb helyzeteket eredményeznének, emiatt nem is szabad ilyen eseménynek bekövetkeznie. Gondoljunk például az atomerőművekre, repülőgépekre, forgalomirányító rendszerekre. Az ilyen alkalmazási területeken szigorú keretek közt kell bizonyítani, hogy a megírt program minden körülmények között helyes. Ennek bizonyítása nem könnyű feladat, a következőkben megismerhetjük a programhelyesség igazolásához szükséges alapokat.

4.4.1. A verifikáció feladata

Először is azt kell meghatározni, hogy mikor értünk egy programot helyesnek. Ennek bevezetéséhez elő- és utófeltételeket fogunk használni. A verifikáció egy általános feladata megmutatni egy adott S programról, hogyha egy bizonyos Q előfeltétel igaz az S végrehajtása előtt, akkor egy bizonyos R utófeltétel is igaz lesz miután lefutott a program, bizonyítva ezzel, hogy S végrehajtása befejeződött. Jelölésre a {Q}S{R} formát alkalmazzuk.

Vegyünk egy szemléletes példát. Tegyük fel, hogy süteményt kívánunk és sütni akarunk, valamint jóllakottak és elégedettek leszünk, miután megsütjük a kedvenc tortánkat. Ebben az esetben a „süteményt kívánunk” jelenti az előfeltételt, a „jóllakottak és elégedettek leszünk” pedig az utófeltételt. A feltételekben használhatóak a matematikai logika műveletei (konjunkció, diszjunkció) az egyszerű kezelhetőségért.

Formalizált jelöléssel a következőképpen írható le a fenti példa:

{süteményt kívánunk ÉS sütni akarunk}
megsütjük a kedvenc tortánkat
{jóllakottak ÉS elégedettek leszünk}

vagy akár részletezhetjük is a végrehajtást:

{süteményt kívánunk ÉS sütni akarunk}
megvásároljuk a hozzávalókat;
előkészítjük a szükséges eszközöket;
kikeverjük a tésztát és a krémet;

összeállítjuk a tortát;
megsütjük sütőben;
hagyjuk kihűlni;
jóízűt eszünk belőle;
{jóllakottak ÉS elégedettek leszünk}

A helyességbizonyítást egyszerűbb elvégezni, ha külön vizsgáljuk meg a végesség kérdését. Ehhez definiáljuk a részleges helyesség fogalmát. Elnevezésével ellentétben ez nem azt jelenti, hogy a program részben helyes, vagy részben helytelen is, hanem hogy olyan, mint a normál helyesség, csak a végessége nem garantált, azaz lehetséges, hogy a végrehajtása végtelen ciklusba fut. Ez önmagában lényegesen kevésbé hasznos, mint a normál helyesség, de külön megvizsgálva a részleges helyességet, majd bebizonyítani, hogy terminál a program, egyszerűbbé teszi a probléma megoldását.

Egy részlegesen helyes programra mutatunk egy példát:

{nincs egy árva garasunk sem}
Utcazenéléssel gyűjtsünk össze egy lottóra valót;
Tegyünk meg egy lottószelvényt;
Ha a szelvénnyel nem nyertük meg a főnyereményt, ugorjunk vissza a program elejére;
Ha a szelvény nyert, akkor ünnepeljünk;
{milliomosok vagyunk}

Ha valaha befejeződik a program, akkor gazdagok leszünk, a baj az, hogy a véletlenen múlik ennek a reménytelenül kis valószínűségű eseménynek a bekövetkezése.

A helyzetet bonyolítja, ha rosszul specifikáltuk a feltételeket, ennek az eredménye rosszabb is lehet annál, minthogy pusztán csak haszontalan. A tortás példa feltételeit például a következő program is kielégítené:

{süteményre vagyunk éhesek ÉS sütni akarunk}
megvásároljuk a hozzávalókat;
előkészítjük a szükséges eszközöket;
kikeverjük a tésztát és a krémet;
összeállítjuk a tortát;
megsütjük sütőben;
odaégetjük véletlenül, emiatt ki kell dobni az egészet;
rendelünk egy tortát a közeli cukrászdából, ahonnan még ajándékot is kapunk a torta mellé;
jóízűt eszünk belőle;
{jóllakottak ÉS elégedettek leszünk}

Látható, hogy ez esetben nem egészen az történt a program futása során, mint amit az előző esetben is vártunk, a feltételek mégis teljesülnek. Ez azért lehet, mert nem elég erős feltételeket alkottunk meg a specifikációból. A példa esetén ki lehetne még kötni utófeltételként, hogy a torta jól sikerül. Valós programoknál nyilván sokkal formálisabban és egyértelműen meg tudjuk adni ezeket a feltételeket.

Most egy-két számítógép közelibb példát is nézzünk meg a helyességbizonyításra. Megmutatjuk, hogy a

```
y := 2;
z :=  x + y;
```

kódrészlet helyes a Q = {`x = 1`} előfeltételre, és az R = {`z = 3`} utófeltételre nézve.

Feltesszük, hogy Q igaz, így `x = 1`. Az program utasításainak megfelelően ekkor `y = 2`, valamint `z` változó az `x + y` azaz `1 + 2` értéket veszi fel, tehát az R utófeltétel is teljesül.

4.4.2. Floyd-Hoare logika

Amikor programhelyesség igazolásról beszélünk, akkor a legtöbb ember tulajdonképpen a Floyd-Hoare logikára gondol. Ez a logika egymást követő állapotleírásokon, a korábban bemutatott {Q}S{R} hármasokon alapul. Minden ilyen szerkezet az őt megelőzőekből épül fel néhány meghatározott szabály szerint. A különböző utasítás típusokhoz létezik legalább egy-egy ilyen szabály. A szabályoknak két fajtáját különböztetjük meg: axiómák és következtetési szabályok. Az axióma olyan kiindulási feltételt jelent, ami mindig érvényes, formula esetén a változók bármely hozzárendelése esetén kielégíthető. Az axiómák kiindulási alapul szolgálnak más utasítások logikai levezetéséhez. A következtetési szabály egy szintaktikai szabály, vagy függvény, ami premisszából, azaz kiindulópontból, előzményből, valamint konklúzióból, azaz következményből áll.

Az imperatív programozási nyelvek konstrukcióit a következő elemi szabályokkal képes leírni a Floyd-Hoare logika. Az elemi szabályok kombinálásával felépíthetőek komplex szabályok, amelyek azután egy levezetést adnak meg a programvégrehajtás bizonyításához.

4.4.2.1. Üres utasítás axióma séma

$$\overline{\{P\}\ \mathbf{skip}\ \{P\}}$$

Az üres utasítás szabály kiköti, hogy a skip utasítás nem változtat a program állapotán, így az bármi is a skip előtt, ugyanaz marad utána is.

4.4.2.2. Hozzárendelési axióma séma

$$\overline{\{P[E/x]\}\ x := E\ \{P\}}$$

Ez a szabály azt jelenti, hogy bármely predikátum, amelyik tartalmaz a hozzárendelés jobb oldalán korábban igaz változót, az ezután is igaz. P[E/x] jelöli azt a kifejezést, amit úgy kapunk, hogy P-ben lecseréljük az x változó szabad előfordulásait az E kifejezéssel. Az axióma jelentése, hogy a {P[E/x]} igazságértékei ekvivalensek {P} helyettesítés után igazságértékeivel.

A következő példák érvényes struktúrák:

```
{ x + 1 = 43 } y := x + 1 { y = 43 }
{ x + 1 ≤ N } x := x + 1 { x ≤ N }
```

A hozzárendelési axióma nem alkalmazható olyankor, ha két címke ugyanarra a tárolt értékre hivatkozhat. Például a `{ y = 3 } x := 2 { y = 3 }` nem lehet állítás, ha `x` és `y` ugyanarra a változóra hivatkozik, mivel nincs olyan előfeltétel, ami az `x` 2.vel történő értékadása után 3-ra változtatná `y` értékét.

4.4.2.3. Kompozíciós szabály

Szekvenciálisan végrehajtott programokra alkalmazható szabály. S és T legyen két program, és az S előbb hajtódik végre, mint T, ekkor:

$$\frac{\{P\}\ S\ \{Q\}\ ,\ \{Q\}\ T\ \{R\}}{\{P\}\ S;T\ \{R\}}$$

Például vegyünk két hozzárendelési axiómát:

```
{ x + 1 = 43 } y := x + 1 { y = 43 } és
{ y = 43 } z := y { z = 43 }
```

Alkalmazva a szabályt a következőt kapjuk:

```
{ x + 1 = 43 } y := x + 1; z := y { z = 43 }
```

4.4.2.4. Feltételes szabály

$$\frac{\{B \wedge P\}\ S\ \{Q\}\ ,\ \{\neg B \wedge P\}\ T\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{endif}\ \{Q\}}$$

Ennél a szabálynál a B feltételt használjuk az elágazás felírására, a {B ∧ P} S {Q}, és {¬B ∧ P} T {Q} a két végrehajtási ág felírásai.

4.4.2.5. Következtetési szabály

$$\frac{P' \rightarrow P\ ,\ \{P\}\ S\ \{Q\}\ ,\ Q \rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

A következtetési szabállyal meglévő utasítások felhasználásával egy új, következmény utasítást határozhatunk meg.

4.4.2.6. While szabály

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{\neg B \wedge P\}}$$

A feltételes szabályhoz hasonlóan értelmezhető a While szabály is, ami egy ciklus konstrukciót valósít meg.

4.4.3. Modellellenőrzés

Egy modellellenőrző feladata, hogy megvizsgálja, hogy a szoftver kielégíti-e a megadott interfész által megfogalmazott viselkedési követelményeket. Ennek kivitelezésére, a követelmények leírására általában valamilyen formális eszközt szoktak használni. Automatikus és költséghatékony módon végezhető el az ilyen eszközök segítségével verifikáció, valamint a tervezési hibák feltárása. Ez a program logikailag felírt összefüggéseiből végezhető el tételbizonyítással, ami akár komplex rendszerek esetén is használható. Egyes eszközök matematikai modellek – például automaták, hálózaton kommunikáló automaták, petri hálók, bináris döntési diagramok – segítségével adnak megoldást a feladatra, míg vannak absztraktabb szinten dolgozó megvalósítások is. A gyakorlatban ezek a magasabb szintű modellek jobban használhatóak közvetlenül komplex rendszerekre, processz algebrát, processz kalkulust alkalmaznak a verifikáció során.

A modellellenőrzés egy véges állapotú verifikációs technika. Használata vonzó lehet, mert képes felderíteni bonyolultabb hibákat is szekvenciális és konkurens végrehajtású rendszerek logikájában is. A legfőbb nehézsége a gyakorlati alkalmazásnak a forráskóddal kifejezett nem véges állapotú rendszerekből a verifikációhoz is használható modell automatizált kinyerése.

A fellelhető eszközök már eljutottak arra a fejlettségi szintre, hogy hardver közeli területekkel foglalkozó és telekommunikációs cégek elkezdték beépíteni fejlesztési folyamataikba a modellellenőrzésen alapuló programhelyesség ellenőrzést is.

4.5. Szimbolikus végrehajtás

Dinamikus programanalízis segítségével inputról inputra kiértékelhetjük a kapott outputokat, végrehajtási utakat. Ha meghibásodást találunk a program működésében, akkor elmondhatjuk, hogy az alkalmazás hibát tartalmaz, minden inputra helyes eredménnyel lefutó programról viszont csak annyit tudunk biztosan, hogy a lehetséges inputok egy kiválasztott részhalmazára helyesen működik. Az azonos utakat bejáró tesztesetek végrehajtásával kevésbé lesz hatékony a hibakeresés, ezért fontos az inputok körültekintő megválasztása. A helyes tesztesetek megválasztásához fontos megérteni egy eredmény kiszámításának menetét: a megválasztott inputok függvényében hogyan keletkezik az adott érték. Ez történhetne akár a kód tanulmányozásával is, de még ez sem lenne garancia arra, hogy a kód minden részét felderítettük, így a program megértését támogató technikáknak a gyakorlatban kiemelt fontossága van. A szimbolikus végrehajtás elősegíti az input és output értékek kapcsolatának meghatározását, jelzi a kódban lévő lehetséges hibák előfordulását, és segíti a bejárható tesztelési utak felderítését.

4.5.1. Példa szimbolikus végrehajtásra

A szimbolikus végrehajtás működését egy egyszerű kódrészlet segítségével szemléltetjük. Előtte viszont egy kis magyarázatra szorul az alkalmazott módszer elméleti háttere. A szimbolikus végrehajtást modellellenőrzéssel és kielégíthetőség vizsgálatával kombináltan is szokták alkalmazzák. Minden utasításhoz rendelhető egy predikátum, pontosabban az inputokon meghatározott feltétel. Ez a predikátum alapesetben igaz, elágazásnál pedig a vezérlési utasításban szereplő predikátum. Kombinálásukkal útvonal feltételek képezhetők, így visszavezethető a probléma a kielégíthetőségi problémára: ha a kapott formula kielégíthető, akkor egy lehetséges végrehajtási útról van szó. A kielégíthető feltételekből megfelelő inputokkal meg tudunk határozni olyan tesztet, amely az adott végrehajtási utat fedi le.

A példa kódrészlet legyen a következő:

```
int x, y, z;
read(x, y);
if (x < y) {
     if (y > 0) {
          z = y;
     }
} else {
     if (x > 0) {
          z = x;
     }
}
```

Ezen egy konkrét végrehajtási út lehet:

```
x = 10, y = 20
10 < 20 ? true
20 > 0 ? true
z = 20
```

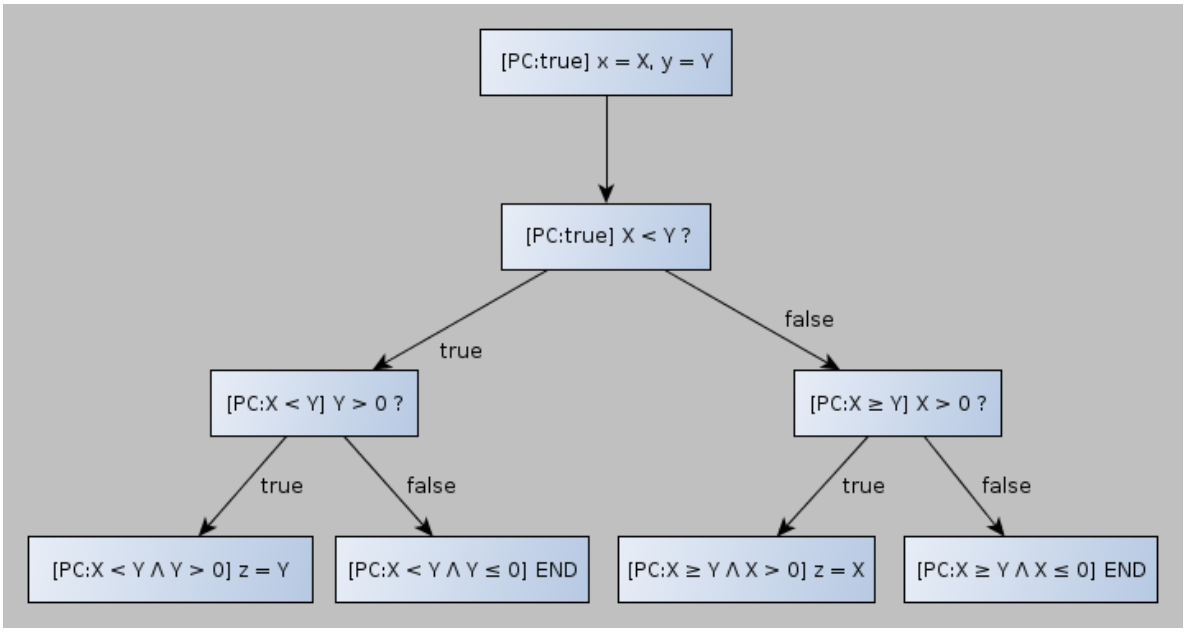
Szimbolikus végrehajtás során pedig a következő, predikátumokkal megcímkézett trace-t kapnánk az előző útvonalon (PC jelentése útvonal feltétel / path condition):

```
[PC:true] x = X, y = Y
[PC:true] X < Y ?
[PC:X < Y] Y > 0 ?
[PC:X < Y ∧ Y > 0] z = Y
```

Mint ahogy látható a példában is, amikor szimbolikusan kiértékelünk egy kifejezést, akkor a változók szimbolikus értékei behelyettesítődnek a kifejezésbe. Ha egy értékadó utasítás jobboldalán áll ilyen kifejezés, akkor a kiértékeléssel meghatározott új szimbolikus kifejezés lesz a baloldalon álló változó új szimbolikus értéke.

A példából világosan látszik, hogy milyen feltételek kellenek egy utasítás végrehajtásához, így látható az is, hogy a *z* változó értékadásához két feltételnek is teljesülnie kell az inputokra, valamint hogy ilyen esetben a változó a második beolvasott értéket fogja felvenni.

A 13. ábra a teljes szimbolikus végrehajtási fát mutatja be:



13. ábra: A példakód teljes szimbolikus végrehajtási fája

4.5.2. Felhasználási területei

Szimbolikus végrehajtás (analízis) során az aktuális értékek helyett szimbólumok segítségével követjük a program logikáját. Ez az analízis egy absztrakt interpretációt, a program szemantikájának egy közelítését jelenti. A végrehajtási utak extrém nagy mennyisége miatt a gyakorlati szimbolikus végrehajtás nehéz feladat, a probléma egy önálló kutatási területté nőtte ki magát.

A programot nem egy konkrét értéken futtatjuk, hanem input szimbólumon végezzük el a kiválasztott programúton meghatározott műveleteket. Ha egy – a lehetséges értékek közül

kiválasztott – inputérték mellett vizsgáljuk a programunk végrehajtását, akkor csak azt látjuk, hogy hogyan működik a program egy különleges esetre. Hogyha képesek vagyunk formulákként kezelni a változók értékeit, akkor a programhelyességet egy sokkal általánosabb módon is vizsgálhatjuk.

A szimbolikus végrehajtásnak egyik fő felhasználási területe a magas lefedettséget (például útlefedettséget) biztosító teszt inputok generálása. Jól használható még ezen kívül nem behatárolt értékű változókkal dolgozó párhuzamosított alkalmazások tesztelésére is.

Általánosságban egy szimbolikus végrehajtás eredménye szimbólumok egy halmaza, amik reprezentálják a változók egy értékét a program egy adott állapotában. Ezek a szimbólumokból álló karakterláncok a végrehajtás során nagyon dinamikusan nőnek, ezért a szimbolikus végrehajtó rendszer egyszerűsítései nélkül nem lennének hasznosíthatók önmagukban.

Mint említettük, a szimbolikus végrehajtást modellellenőrzéssel és kielégíthetőség vizsgálatával kiegészítve alkalmazzák. A számítástudományban a modellellenőrzés egy adott rendszer modelljét teszteli le, hogy megfelel-e egy adott specifikációnak. Tipikusan olyan hardver vagy szoftver rendszereknél használatosak, ahol a specifikációban valamilyen biztonsági követelmény is található, például nem tartalmazhat halott kódot, vagy más olyan hasonló kritikus állapotot, ami a rendszer összeomlásához vezetne. A modellellenőrzés automatikusan és szisztematikusan utakat kombinál össze, amivel meghatározza a lehetséges útvonalakat a modellben. A szimbolikus végrehajtás ezeken az utakon egy kimerítő végrehajtást eredményez. Bár egy egyszerű tesztelés, vagy szimuláció gyorsabb, és könnyebben végrehajtható, csak néhány végrehajtási utat vizsgál a rendszerben, ezért elkerülhet hibákat. A modellellenőrzés során megtalált hibákat a modell útvonalai alapján képzett trace-ek jelölik ki. A teljes végrehajtás emberi erővel szinte lehetetlen volna, hiszen nagyszámú végrehajtási útról beszélhetünk már a legegyszerűbb programok esetében is, ezért a teljes végrehajtást automatizált eszközök segítségével szokták elvégezni.

4.5.3. Gyakorlati alkalmazása

A szimbolikus végrehajtó rendszerek a nagyszámú végrehajtási út miatt is számításigényes szoftverrendszerek, és nem is egyszerű megalkotni őket számos leküzdendő nehézség miatt. Nézzünk meg közelebbről egy konkrét megvalósítását a tárgyalt technikának.

A NASA szakemberei szimbolikus végrehajtó eszközt készítettek a szoftvereik teszteléséhez: Java byte-kódon végeztek el szimbolikus végrehajtást. A céljuk az volt, hogy olyan hibakereső technikát fejlesszenek ki, ami jól használható emberes küldetések űrhajós repülésirányító szoftverének tesztelésekor. Egy kimerítő megoldást találtak a szimbolikus végrehajtás, a modell ellenőrzés, valamint a kielégíthetőség problématerület kutatási eredményeinek felhasználásával. Java PathFinder eszközük segítségével teszteseteket tudnak generálni meglévő szoftverekhez úgy, hogy ezek a tesztesetek flexibilis lefedettség metrikák szerinti magas lefedettséget is képesek elérni. Az eszköz megengedi a vegyes használatot: a kódot valós és szimbolikus módon is végre lehet hajtani. A szimbolikus végrehajtás elkezdhető a program bármely pontján, és a futtatás bármely időpontjában. A teljes megvalósítás Java virtuális gép fölött épül be a végrehajtásba.

Számos nehézségre megoldást kell találni egy ilyen eszköz kifejlesztésekor. Bizonyos vezérlési konstrukciók általában is problémát jelenthetnek, például a ciklusok, amik

lehetséges végtelen futást eredményezhetnek a szimbolikus végrehajtás során. A szimbolikus végrehajtást használó eszközök úgy védik ki az ilyen eshetőségeket, hogy limitálják a szimbolikus keresési állapotteret, vagy korlátot határoznak meg a modellellenőrzés keresési mélységére, vagy a predikátumok számára az útvonalfeltételben. A ciklusok mellett a tömbök, pointerek, eljáráshívások is gondot jelenthetnek, a gyakorlatban használható megvalósításoknak ezekkel a problémákkal is meg kell küzdeni.

A megalkotott Java PathFinder a következő hibák felderítésére összpontosít:

- konkurencia kezeléssel kapcsolatos hibák
  - holtpont felderítése
  - versenyhelyzetek felismerése
  - értesítési jel elvesztése
- kifejezetten Java specifikus hibák
  - kezeletlen kivételek
  - dinamikus adatterületek használatának problémái
- egyéb, alkalmazás specifikus megszorítások, input megkötések hibái

4.6. Feladatok

1. Látogassunk el a PMD statikus elemző honlapjára (http://pmd.sourceforge.net/), és telepítsük fel a rendszerünkre. A honlapon található példakódok közül elemezzünk le párat (a PMD-vel), és figyeljük meg, hogy milyen PMD-üzeneteket kapunk.
2. Tekintsük az alábbi kódrészletet:

```
1  int i;
2  int sum = 0;
3  int product = 1;
4  for(i = 0; i < N; ++i) {
5  sum = sum + i;
6  product = product *i;
7  }
8  write(sum);
9  write(product);
```

   A statikus szeletelés módszerét alkalmazva határozzuk meg SF(2) és SB(9) értékét!

3. Tekintsük az alábbi kódrészletet:

```
1  read (n)
2  for I := 1 to n do
3  a := 2
4        if c1==1 then
5              if c2==1 then
6                    a := 4
7              else
8                    a := 6
9              z := a
10 write (z)
```

   A dinamikus szeletelés módszerét alkalmazva határozzuk meg `DynSlice(10)` értékét, ha tudjuk, hogy `n = 1, c1 = 1` és `c2 = 1`!

4. Mutassuk meg, hogy a következő függvény helyes a meghatározott elő- és utófeltételek mellett:

```
int identity (int x) {
   x = x * 1;
   return x;
}
```

Q = {a függvény paraméterül 0-100 közötti egész számot kaphat}, R = {a fgv. visszatérési értéke megegyezik a paraméterül kapott értékkel}

5. Keressünk két példát, amelyek szintaktikusan helyesek, de szemantikusan hibásak. Végezzünk el szimbolikus analízist, és a végrehajtási trace-ek segítségével jelöljük ki a hibát hordozó utasításokat a kódban. Használjuk a példában bemutatott jelöléseket a feltételek megadására. Határozzunk meg teszteseteket a megkapott feltételek segítségével a lehetséges kimenetek szerint.
6. Melyik válasz a helyes? A program szimbolikus végrehajtási fája végtelenné válik
   a. akkor és csak akkor, ha van végtelen ciklus a programban.
   b. ha egy output változó értéke végtelen lesz.
   c. ha végtelen ciklus, vagy rekurzív függvényhívás található a programban.
   d. akkor és csak akkor, ha a ciklusok nem megfelelően egymásba ágyazottak.
   e. akkor és csak akkor, ha egy elágazás feltételének igaz kiértékelése helytelenül történik.
7. A következő végrehajtási úton akarunk tesztelni egy programban:

```
cin >> x >> y;
z = 1;
y != 0 true
y % 2 == 1 true
z = z * x;
y = y / 2;
x = x * x;
!(y != 0) true
cout << z << end;
```

Szimbolikus végrehajtással az x változónak A-t, y változónak B-t rendelve a következő három eredmény közül melyik helyes?

   i. `x = A * A`
   ii. `y = B / 2`
   iii. `z = A`

   a. mind
   b. mind, kivéve az (i)
   c. mind, kivéve az (ii)
   d. mind, kivéve az (iii)
   e. egyik sem

8. Az előző feladat végrehajtási útjához határozzunk meg teszteseteket a szimbolikus végrehajtás eredményeinek felhasználásával.

5. Módszertani megközelítés

Az előző fejezetekben ismertetett technikák megfelelő módszertani segédlet, folyamatbeli világos elhelyezés nélkül olyanok, mint a hangzatok harmónia nélkül a zenében. A technikák önmagukban tárgyalhatók és elsajátíthatók, példákon keresztül gyakorolhatók. Azonban, ahhoz hogy a gyakorlatban, valós projektekben is sikeresen alkalmazhatók legyenek, számos egyéb feltételnek is teljesülni kell, de a legfontosabb, hogy lássuk, mikor, milyen körülmények között lehet azokat leginkább alkalmazni.

Jelen fejezetben megadunk néhány lehetséges osztályozási módot, ami támpontot nyújthat a módszertani alkalmazásukhoz. Azonban, hangsúlyozzuk, hogy mindezt a teljesség igénye nélkül tesszük, nem adunk meg folyamatleírást vagy egyéb részletesebb módszertani alkalmazási módot.

A tesztelési módszereket különböző szempontok szerint tudjuk csoportosítani, kategorizálni. Ilyenek lehetnek az alábbiak:

- Az életciklus mely részében tesztelünk?
- A rendszer mely szintjét érinti a teszt?
- Tipikusan ki végzi a tesztet?
- Mi a tesztelés célja?
- Milyen típusú a tesztelés?
- Hogyan tesztelünk (alapelv)?
- Milyen megközelítést alkalmazunk?
- Mi a tesztesetek elvárt eredményét meghatározó orákulum?
- Milyen hibákat talál meg a módszer?

Az alábbiakban egyenként megvizsgáljuk a fenti szempontokat, ami segíthet az egyes módszereket elhelyezni a kategorizálásban, és az aktuális projektünkhöz kiválasztani a legmegfelelőbbeket. Az ismertetett módszerek mindegyik elhelyezhető az alábbi sokdimenziós osztályozás valamely pontjába vagy pontjaiba. Ez az elhelyezés azonban nem adható meg általános módon, legfeljebb szokásos alkalmazások mondhatók. Az egyes módszerek alkalmazása sokkal inkább projektfüggő, ezért a csoportosítás egy bizonyos példányosítását kell elképzelnünk a módszerek tekintetében egy bizonyos projekt esetében. Ehhez viszont segítséget nyújthat az osztályozás, ha nem is nyújt teljes módszertani keretet.

5.1. Életciklus szerint

Életciklus szerinti csoportosításnál azt vizsgáljuk, hogy a tesztet a szoftverfejlesztési életciklus mely lépésében hajtjuk végre (mert a tesztelés folyamatos tevékenység, nem csak a „tesztelés” fázisban tesztelünk). Ilyenek lehetnek:

- Specifikációs fázis.
- Tervezési fázis
- Fejlesztési fázis
- Tesztelési fázis
- Karbantartási fázis

5.2. A rendszer érintett szintje szerint

A tesztelés általános felosztása a jól ismert szoftverfejlesztési V-modell szerint történik. A V-modell a szoftverfejlesztési folyamatot tervezési, implementációs és tesztelési részekre bontja. A tervezési és tesztelési fázisokat pedig egymásnak megfelelő szintekre bontja. A tervezés szintjei felülről lefelé a követelmény specifikáció, funkcionális specifikáció, technikai specifikáció, melyekhez rendre az átvételi teszt, rendszer teszt, integrációs teszt tartoznak. A tervezés szintjeit nyilván felülről lefelé érintjük. A technikai specifikáció után a program specifikáció, a kódolás következik, amely magában foglalja az egység tesztelést is. Ezután a tesztelési szinteket alulról felfelé érintjük, vagyis integrációs-, rendszer-, majd átvételi tesztek következnek. A különböző szintekre más és más módszerek jellemzők. A szempont tehát itt az, hogy az elkészülő rendszer milyen szintű elemeit kívánjuk tesztelni? A jellemző szintek:

- Modul szint. Például forráskód átvizsgálás, egység tesztek, osztálydiagramok átnézése.
- Komponens integrációs szint. Például technikai specifikáció, architektúra diagramok áttekintése, modulok interfészeinek tesztelése, integrációs teszt
- Rendszer szint. Például funkcionális specifikáció ellenőrzése, funkcionális teszt, rendszerteszt, lefedettségek ellenőrzése.
- Átvételi tesztek szintje. Például teljes, éles környezetbe integrált rendszer tesztelése, kézikönyvek átnézése, nem funkcionális követelmények tesztelése

5.3. Tesztelést végző szerint

Csoportosíthatjuk a módszereket aszerint, hogy ki használja, vagyis tipikusan melyik szerepkör feladatkörébe tartozik az adott módszerrel végzett tesztelés. Az alapvető szerepkörök:

- Fejlesztő. Bármilyen módszer, ami ahhoz kell, hogy a munkaterméket érdemes legyen elkezdeni komolyabban tesztelni. Például egység teszt.
- Tesztelő. Gyakorlatilag bármilyen módszer, ami nem csak a fejlesztés során használható.
- Béta tesztelő. Főként felhasználói teszthez, validációhoz használható tesztelési módszerek.
- Végfelhasználó. A béta tesztelőhöz nagyon hasonló, de más szempontjai vannak.

5.4. Tesztelés célja szerint

A szerint is csoportosíthatunk, hogy mi a tesztelés célja. Jellemzően:

- Megértés. Megismerni a termék felépítését, logikáját, tartalmát, működését.
- Verifikálás. Ellenőrizni, hogy a termék megvalósítása a terveknek megfelelő-e.
- Validálás. Ellenőrizni, hogy a termék betölti-e rendeltetését, használható-e arra a célra, amire készítették.
- Változások követése.
    - Regresszió. A változtatás hozott-e be már ismert, de korábban nem tapasztalt vagy már kijavított hibát.
    - Konfirmáció. A változtatás kijavította-e azt a hibát, ami miatt a változtatásra egyáltalán szükség volt.

5.5. A tesztelés típusa szerint

A tesztelés nagyon sokféle típusú lehet, és minden típusnak megvannak a maga specialitásai, technikai követelményei. Típus szerint az alábbi felosztás lehetséges:

- Funkcionális
- Nem funkcionális
- Terheléses
- Stressz
- Biztonsági
- Használhatósági
- Monkey
- Telepíthetőségi
- Integrálhatósági
- Jogszabályi előírások tesztelése

5.6. Statikus/Dinamikus

Az is fontos szempont, hogy a teszt elvégzéséhez szükséges-e a program futtatása vagy sem. Ez alapján két nagy csoportba osztható az összes módszer:

- Statikus. Program futtatás nélküli tesztelés. Bármilyen dokumentáción (forráskódot is beleértve) elvégezhető. Tipikusan manuális review vagy statikus elemző eszközökkel végzett automatikus tesztelés.
- Dinamikus. A program tesztelése végrehajtással. Szükséges hozzá a végrehajtható program(részlet).

5.7. Megközelítés szerinti csoportosítás

Megközelítés szerinti csoportosításról akkor beszélünk, amikor azt vizsgájuk, hogy a tesztelés mi alapján van tervezve, mi vezérli a tesztesetek elkészítésének, végrehajtásának sorrendjét. Az alábbi fő lehetőségeink vannak:

- Kockázat, hiba alapú. Kiterjedt kockázatelemzés után a legkockázatosabbnak tartott elemek kezelésére koncentrál.
- Módszeres. Szisztematikus, teljességre törekvő.
- Ad hoc. Alkalmi, véletlenszerű, tervezetlen.
- Tapasztalati alapú. Részleteiben előre megtervezetlen, tesztelő tapasztalatára alapuló.
- Processz alapú. Valamilyen előírás által meghatározott folyamatnak megfelelő.

5.8. Teszt orákulum szerint

A teszteléshez meg kell határozni az egyes tesztesetek elvárt eredményeit. Ezt a feladatot alapvetően a teszt orákulum látja el. Ezt a teszt orákulum jellemzően az alábbi listából kerül ki:

- Specifikáció. Az elvárt kimenetet a program előzetes dokumentációja alapján a tesztelő határozza meg.
- Felhasználói kézikönyv. Az elvárt kimenetet a program utólagos dokumentációja alapján a tesztelő határozza meg.

- Előző verzió. Az adott bemenethez az elvárt kimenetet a program egy korábbi verziójával állítjuk elő.
- Szakértő. Az elvárt kimenetet a program működéséhez értő személy határozza meg.
- Működési profil. A programhoz bizonyos bemenet-kimenet párok mintaként a rendelkezésünkre állnak.

5.9. Megtalált defektusok fajtái szerint

Az, hogy a teszt milyen hibákat képes megtalálni, rendkívül fontos osztályozás, hiszen segítségével a szükséges teszteket igazíthatjuk a tesztelési célokhoz és prioritásokhoz. A főbb lehetséges hibafajták:

- Számítási hibák.
- Logikai hibák. A program logikájában rejlő hibák, mint hibás vezérlés, rossz döntések.
- Adat kimeneti (eredmény) hibák.
- Inicializációs hibák. Nem, vagy hibásan inicializált értékekből eredő hibák.
- Adat definíciós hibák.
- Interfész hibák. Rosszul megvalósított, vagy hibásan használt interfészből adódó hibák
- Bemeneti/kimeneti hibák. A program és a környezete közötti adatáramláshoz kapcsolódó hibák.

6. Hibakeresés, debugging

A hibakeresés, vagy debugging azután alkalmazható, hogy a tesztelés megtalálta a hibát. Célja a teszteléstől eltérően nem a hiba felfedezése, hanem a már felfedezett hiba valódi okának felderítése. Bár szigorúan véve nem a teszteléshez tartozik, de sokan odaértik, hiszen a minőségjavításhoz hozzá tartozik. A tesztelés a tünetek leírása, a hibakeresés a diagnózis, a javítás a gyógyír.

Ahhoz, hogy megértsük miért van szükség a hiba felfedezése után a hibakeresésre, tisztában kell lennünk a hiba keletkezésének fázisaival, ugyanis a hiba észlelése csak a végső stádiuma egy hosszabb folyamatnak.

6.1. A hiba keletkezésének lépései

Általánosan elmondhatjuk, hogy az alábbi lépéseket különböztethetjük meg, amikor hibák keletkezéséről beszélünk:

- A programozó elkövet egy hibát (error vagy mistake) amivel létrehoz egy defektust (*defect*). A defektus a program kódjában található hiba; egy olyan kódrészlet, aminek a futtatása fertőzött állapotot (*infected state*) hozhat létre a program futása során.
- A defektus fertőzést (*infection*) okoz, ami továbbterjedhet. Ahhoz, hogy a defektus fertőzést okozzon először is végre kell hajtódnia, méghozzá olyan feltételek mellett, amik hatására a program nem az elvárt, vagyis fertőzött állapotba kerül.
- A fertőzés továbbterjed, és meghibásodást *(failure)* okozhat. A fertőzés hatására a program egyre több részállapota tér el az elvárttól. De itt sem törvényszerű a továbbterjedés, mert lehet, hogy a rossz részállapotot futás közben felülírják, elfedik, vagy épp kijavítják más futó programrészek.
- A fertőzés meghibásodást (*failure*) okoz.Ez a felhasználó által is érzékelhető, az elvárttól eltérő viselkedés, például rossz kimeneti érték vagy elszállás.

Tehát, mint látható, nem minden defektus okoz fertőzést, és nem minden fertőzés okoz meghibásodást, vagyis, ha mi nem látunk tényleges hibát a programban, az nem azt jelenti, hogy valójában nincs benne defektus. (Ami a későbbiekben aztán fertőzésen keresztül hibát okozhat majd).

6.2. A hibakeresés lépései

A hibakeresés fő lépései az alábbiak:

- A hibát, és a javításához kapcsolódó összes eredményt tároljuk el egy adatbázisban, ahol nyomon tudjuk követni az állapotát.
- Próbáljuk meg reprodukálni a hibát.
- Automatizáljuk és egyszerűsítsük a teszt esetet (amivel a hibát reprodukáljuk).
- Találjuk meg a lehetséges fertőzési pontokat.
- Fókuszáljunk a legvalószínűbb fertőzésekre.
- Keressük meg a fertőzési láncot (és a lánc végén a defektust).
- Javítsuk ki a defektust.

A hibakeresés során egy – időben és térben – történő keresést hajtunk végre, ahol is azt keressük, hogy hol kerül a program egy egészséges, helyes (*sane*) állapotból egy fertőzött (*infected*) állapotba. Ezt a folyamatot két fő elvnek kell vezérelnie:

1. Meg kell tudnunk különböztetni a helyes állapotot a fertőzött állapottól.
2. Meg kell tudnunk állapítani, hogy a fertőzés (és a későbbi defektus) megtalálása szempontjából milyen információ releváns, és milyen információ nem az.

6.3. Automatizálható módszerek

Mivel a kézzel történő hibakeresés sokszor nehézkes és unalmas feladat, számos olyan részterület létezik, amelyre már vannak automatizált eszközök. Ezen eszközök az alábbi feladatokat tudják megvalósítani:

- A teszt inputok egyszerűsítése
- Program-szeletelés
- Állapotok megfigyelése
- Összehasonlítások (*assertions*) készítése
- Ellentmondások (*anomailies*) keresése
- Az ok-hatás (*cause-effect*) lánc megtalálása

6.4. A hiba reprodukálása

A hiba reprodukálása két szempontból fontos:

1. A hiba figyelemmel kísérése. Ha nem tudjuk reprodukálni és kontrollálni a hibát, akkor maximum a programkódból találgathatunk, hogy mi történt valójában, ami behatárolja a lehetőségeinket.
2. Annak ellenőrzése, hogy a hiba eltűnt a javítás után. Reprodukálás nélkül nem tudjuk megmondani, hogy a hiba eltűnt-e vagy sem.

A reprodukálás során mind a vizsgált hiba környezetét, mind pedig a hibát okozó lépéssorozatot a lehető legpontosabban kell reprodukálni.

A környezet reprodukálására az alábbi iterációt kövessük:

a. Próbáljuk meg reprodukálni a hibát a saját lokális környezetünkben (*environment*).
b. Ha a hiba nálunk nem jön elő, változtassunk meg fokozatosan a környezetünk jellemzőit aszerint, hogy minél jobban hasonlítson arra a környezetre ahol a hiba előfordult. Azokat a jellemzőket változtassuk először, amik várhatóan okozhatják a hibát, illetve amiket a legkönnyebben lehet megváltoztatni.
c. Kövessük a b) pont szerinti jellemzők átalakítását addig, amíg:
    - reprodukálni nem tudjuk a hibát
    - a környezet teljesen meg nem egyezik a hiba környezetével. Ekkor két további eset lehetséges:
        i. Lehetséges, hogy a hibabejelentés nem teljes, vagy rossz. A leírt lépések alapján nem csak a mi környezetünkön, de a hiba környezetében sem történhetett meg a hiba.

ii. A hibabejelentés teljes, de még mindig van valami különbség a két környezet között. Próbáljunk meg még több adatot kérni a hibabejelentőtől.

A hibát okozó lépéssorozat reprodukálásának érdekében azt a program inputot kell reprodukálnunk, ami előidézi a hibát. Ez a folyamat akkor tekinthető sikeresnek, ha az inputot megfigyelni és kontrollálni is tudjuk, ekkor lesz a program futása determinisztikus.

6.5. A hibák leegyszerűsítése

A hibakeresésnek ezen fázisa azt tűzi ki célul, hogy a reprodukált hibát leegyszerűsítse egy olyan teszt-esetté, ami csak a releváns információkat tartalmazza. Ez azt jelenti, hogy a teszt-esetben csak azok, és pontosan azok az információk vannak, amik szükségesek ahhoz, hogy a hiba előforduljon.

A probléma minden egyes jellemzőjére ellenőriznünk kell, hogy releváns-e a probléma előfordulásának szempontjából. Ha nem, akkor egyszerűen kihagyjuk a probléma leírásából.

6.5.1. Módszer

Egyfajta lehetőség a Kernighan and Pike (1999) által javasolt bináris keresés, ami a következőképpen működik:

Az input felét hagyjuk el. Ha a hiba nem áll fent, akkor lépjünk egy lépést vissza, és hagyjuk el a másik felét az inputnak.

Ezt a módszert általánosítsuk a következőképpen:

Legyen egy test(c) függvényünk, ami kap egy c inputot, és eldönti, hogy a kérdéses hiba előfordul-e (✘), nem fordul elő (✓), vagy valami más történik (**?**). Tegyük fel, hogy van egy hibát generáló inputunk (c_x), amit kétfelé tudunk osztani (c_1 és c_2). Ekkor három dolog történhet:

1. A c_x input első felének eltávolítása után továbbra is fennáll a hiba, vagyis $test(c_x \backslash c_1)$ = ✘. Ekkor folytathatjuk a keresést egy új $c'_x = c_x \backslash c_1$ inputtal.
2. Az előző pont nem teljesülése esetén, ha c_x input második felének eltávolítása után továbbra is fennáll a hiba (vagyis $test(c_x \backslash c_2)$ = ✘), akkor folytassuk az új $c'_x = c_x \backslash c_2$ inputtal.
3. Az előző két pont nem teljesülése esetén osszuk kisebb részekre az eredeti c_x inputot (finomítjuk az algoritmust).

A 3-as eset miatt általánosítanunk kell az előtte lévő részt is, vagyis: ha c_x -et n részhalmazra bontjuk ($c_1, \ldots, c_n$), akkor:

- Ha valamelyik részhalmaz eltávolításakor továbbra is fennáll a hiba ($test(c_x \backslash c_i)$ = ✘, valamely $i \in \{1, \ldots, n\}$-re), akkor folytassuk $c'_x = c_x \backslash c_i$, és $n' = max(n - 1, 2)$.
- Különben folytassuk $c'_x = c_x$, és $n' = 2n$. Ha c_x-et nem lehet tovább bontani, akkor kész vagyunk.

Mivel a program futását befolyásoló tényezők nem csak inputok lehetnek (hanem pl. idő, kommunikáció, szál-kezelés, stb.), ezért az algoritmust tovább általánosíthatjuk „jellemzőkre” (*circumstances*). A jellemzők egy halmazát nevezzük konfigurációnak.

Formálisan a ddmin algoritmus a következőképpen néz ki:

- A program futását befolyásoló jellemző-halmazt nevezzük el *konfigurációnak*. Az összes jellemző halmazát jelöljük C-vel
- Legyen a *test*: $2^C \rightarrow \{\times, \checkmark, ?\}$ egy tesztelő függvény, ami egy $c \subseteq C$ konfigurációról eldönti, hogy egy adott hiba előfordul (×), nem fordul elő (✓), vagy nem mondható meg(?).
- Legyen $c_\times$ egy „elbukó” konfiguráció, vagyis $c_\times \subseteq C$, ahol *test*($c_\times$)= ×, és legyen a *test* függvény kimenete sikeres, ha egy jellemzőt se adunk meg, vagyis:
  test(Ø) =✓
- A *minimalizáló delta debuggoló algoritmus*, ddmin($c_\times$) minimalizálja a hibát generáló $c_\times$ konfigurációt. Egy olyan $c'_\times$ konfigurációt ad eredményül, amire teljesülnek az alábbiak
  - $c'_\times \subseteq c_\times$
  - *test*($c'_\times$) = ×
  - $c'_\times$ egy releváns konfiguráció, vagyis egyetlen egy jellemzőt sem lehet kivenni $c'_\times$ -ből úgy, hogy a hiba eltűnjön.
- A *ddmin* algoritmust a következőképpen definiáljuk: *ddmin*($c'_\times$) = *ddmin*'($c'_\times$, 2), ahol *ddmin*'($c'_\times$, n)-re teljesülnek az alábbiak:
  - *ddmin*'($c'_\times$, n) = $c'_\times$, ha $|c'_\times| = 1$
  - *ddmin*'($c'_\times$, n) = *ddmin*'($c'_\times \backslash c_i$, max(n − 1, 2)) különben, ha van olyan $i \in \{1..n\} \times$ *test*($c'_\times \backslash c_i$) = ×
  - *ddmin*'($c'_\times$, n) = *ddmin*'($c'_\times$, min(2n, $|c'_\times|$)) különben, ha $n < |c'_\times|$
  - *ddmin*'($c'_\times$, n) = $c'_\times$ különben, ahol

  $c'_\times = c_1 \cup c_2 \cup \ldots \cup c_n$ úgy, hogy minden $c_i, c_j \times c_i \cap c_j = \emptyset$ és $|c_i| \approx |c_j|$ teljesül.
- A *ddmin*' előfeltétele, hogy *test*($c'_\times$) = × és $n \leq |c'_\times|$.

Az algoritmus optimalizálható az alábbi opciók szerint:

- Cache használata – mivel egy konfigurációt többször is tesztelünk.
- Korai leállítás (időkorlát, darabolási egység, előrehaladás alapján)
- Szintaktikus egyszerűsítés – Nem karakterek szerint, hanem nagyobb egységek szerint történik az egyszerűsítés (lexikális szinten).

6.5.2. Példa

Az alábbiakban látható a *ddmin* algoritmusra egy példa. Egy XML alapú adatfeldolgozó program hibásan működik az egyik tesztesetre, míg egy másikra helyesen. A két teszteset között a különbség egyetlen paraméterezett tag: amelyik tesztesetben ez benne van, az bukik, amelyikből hiányzik, az helyesen lefut. A *ddmin* algoritmusunk így a következőképpen működik:

Input:

<SELECT NAME=”priority” MULTIPLE SIZE=7> - 40 karakter – Eredmény: ×

<SELECT NAME=”priority” MULTIPLE SIZE=7> **- 0 karakter – Eredmény: ✓**

Lépésszám	Input	Karakterszám	Eredmény
1	<SELECT NAME="priori**ty" MULTIPLE SIZE=7>**	20	✓
2	**<SELECT NAME="priori**ty" MULTIPLE SIZE=7>	20	✓
3	<SELECT NA**ME="priority" MULTIPLE SIZE=7>**	30	✓
4	**<SELECT NA**ME="priori**ty" MULTIPLE SIZE=7>**	30	×
5	<SELECT NAME="priori**ty" MULTIPLE SIZE=7>**	20	✓
6	**<SELECT NA**ME="priority" MULTIP**LE SIZE=7>**	20	×
7	<SELECT NAME="priority" MULTIP**LE SIZE=7>**	10	✓
8	**<SELECT NA**ME="priority" MULTIPLE SIZE=7>	10	✓
9	<SELE**CT NA**ME="priority" MULTIP**LE SIZE=7>**	15	✓
10	**<SELE**CT NAME="priority" MULTIP**LE SIZE=7>**	15	✓
11	**<SELECT NA**ME="priority" MULTIPLE SI**ZE=7>**	15	×
12	<SELE**CT NA**ME="priority" MULTIPLE SI**ZE=7>**	10	✓
13	**<SELE**CT NAME="priority" MULTIPLE SI**ZE=7>**	10	✓
14	**<SELECT NA**ME="priority" MULTIPLE SIZE=7>	10	✓
15	<SE**LECT NA**ME="priority" MULTIPLE SI**ZE=7>**	12	✓
16	**<SE**LECT **NA**ME="priority" MULTIPLE SI**ZE=7>**	13	✓
17	**<SELE**CT **NA**ME="priority" MULTIPLE SI**ZE=7>**	12	✓
18	**<SELECT** NAME="priority" MULTIPLE SI**ZE=7>**	13	×
19	<SE**LECT** NAME="priority" MULTIPLE SI**ZE=7>**	10	✓
20	**<SE**LEC**T** NAME="priority" MULTIPLE SI**ZE=7>**	10	✓
21	**<SELEC**T NAME="priority" MULTIPLE SI**ZE=7>**	11	✓
22	**<SELECT** NAME="priority" MULTIPLE SIZE=**7>**	10	×
23	<SE**LECT** NAME="priority" MULTIPLE SIZE=**7>**	7	✓
24	**<SE**LEC**T** NAME="priority" MULTIPLE SIZE=**7>**	8	✓
25	**<SELE**CT NAME="priority" MULTIPLE SIZE=**7>**	7	✓
26	**<SELECT** NAME="priority" MULTIPLE SIZE=7>	8	✓
27	<**SELECT** NAME="priority" MULTIPLE SIZE=**7>**	9	✓
28	**<**S**ELECT** NAME="priority" MULTIPLE SIZE=**7>**	9	✓
29	**<S**E**LECT** NAME="priority" MULTIPLE SIZE=**7>**	9	✓
30	**<SE**LEC**T** NAME="priority" MULTIPLE SIZE=**7>**	9	✓
31	**<SEL**EC**T** NAME="priority" MULTIPLE SIZE=**7>**	8	✓
32	**<SELEC**T NAME="priority" MULTIPLE SIZE=**7>**	9	✓
33	**<SELECT** NAME="priority" MULTIPLE SIZE=7>	8	×
34	<**SELECT** NAME="priority" MULTIPLE SIZE=**7>**	7	✓
35	**<**S**ELECT** NAME="priority" MULTIPLE SIZE=**7>**	7	✓
36	**<S**E**LECT** NAME="priority" MULTIPLE SIZE=**7>**	7	✓
37	**<SE**LEC**T** NAME="priority" MULTIPLE SIZE=**7>**	7	✓
38	**<SEL**EC**T** NAME="priority" MULTIPLE SIZE=**7>**	7	✓
39	**<SELE**CT NAME="priority" MULTIPLE SIZE=**7>**	6	✓
40	**<SELECT** NAME="priority" MULTIPLE SIZE=7>	7	✓
41	<**SELECT** NAME="priority" MULTIPLE SIZE=**7>**	7	✓
42	**<**S**ELECT** NAME="priority" MULTIPLE SIZE=**7>**	7	✓
43	**<S**E**LECT** NAME="priority" MULTIPLE SIZE=**7>**	7	✓
44	**<SE**LEC**T** NAME="priority" MULTIPLE SIZE=**7>**	7	✓
45	**<SEL**EC**T** NAME="priority" MULTIPLE SIZE=**7>**	7	✓
46	**<SELE**C**T** NAME="priority" MULTIPLE SIZE=**7>**	7	✓
47	**<SELEC**T NAME="priority" MULTIPLE SIZE=**7>**	7	✓
48	**<SELECT** NAME="priority" MULTIPLE SIZE=7>	7	✓

Az eredmény tehát az, hogy a hibát önmagában a **<SELECT>** is kiváltja.

6.6. A hibakeresés tudományos megközelítése

A több helyen alkalmazott módszer segítségünkre lehet a hibakeresésben is. Az alábbiak szerint járhatunk el:

- Figyeljük meg a hiba egy előfordulását.
- Állítsunk fel egy hipotézist a megfigyelésünk alapján. (A megfigyelésünk támassza alá a hipotézist).
- Használjuk a hipotézist arra, hogy állításokat vonjunk le belőle.
- Teszteljük a hipotézist további kísérletekkel és megfigyelésekkel.
    - Ha a kísérletek összhangban vannak az állításainkkal, akkor finomítsunk a hipotézisünkön
    - Ha a kísérletek nem támasztják alá az állításainkat, akkor új hipotézist kell készítenünk
- Ismételjük az előző két lépést addig, amíg a hipotézist nem lehet tovább finomítani.

Hipotézisek készítéséhez minél több forrásra szükségünk van. Segítségünkre lehet a probléma leírása, a forráskód a rossz futás, a többi futás. Továbbá törekedjünk arra, hogy az új hipotézis tartalmazza a korábban beigazolódott hipotéziseket, viszont ne tartalmazza a korábban elutasított (hamisnak bizonyult) hipotéziseket.

6.7. A hibák megfigyelése

Az eddigiek során láthattuk, hogy mi történhet (mik a lehetséges történések), a megfigyelés során viszont arra keressük a választ, hogy valójában mi történik a hiba keletkezése során. (A konkrét tényekre vagyunk kíváncsiak.)

Fontos a szisztematikus, tudományos megközelítés (lásd korábban), mert így elkerülhetjük a „felesleges” köröket. Mindig tudnunk kell, hogy mit (melyik részét a program-állapotnak) és mikor (a program-futás mely időpillanataiban) szeretnénk megfigyelni.

Ahhoz, hogy a tényeket megfigyeljük, láthatóvá kell tennünk őket a program futása során, ezért az egyik módszer, hogy loggolási technikákat alkalmazunk, aminek segítségével kinyerjük a megfigyeléshez szükséges adatokat. Többféle loggolási technika létezik:

- „printf-loggolás”: kiíratós sorok beillesztése a kódba. Nagyon egyszerű, de sok hátránya van.
- Loggoló-függvények, loggoló-makrók (preprocesszorral rendelkező nyelvek esetén).
- Loggoló-keretrendszerek (pl. LOG4J).
- Loggolás aspektusokkal.
- Loggolás bináris szinten (instrumentálás).

A loggoláson kívül egy másik megközelítés debuggerek használata, amik a következő lehetőségeket nyújtják a hibakeresésben:

- Lefuttathatjuk a programot, és megállíthatjuk egy általunk definiált pillanatban.
- Megfigyelhetjük a megállított program adott állapotát.
- Módosíthatjuk a megállított program adott állapotát.

Az általános debuggerek és általában a hibát megfigyelő eszközök egyik nagy hátránya, hogy „előrefelé” működnek, míg ahhoz, hogy a hiba eredetét megtaláljuk, nekünk „hátrafele” kell gondolkozni. Éppen emiatt fejlesztették ki az ún. „mindentudó” („omniscient”) debuggereket, amik először lefuttatják a programot, felvételt készítenek a futásról, és ezután képesek a felvételt visszajátszani.

6.8. A hiba okának megtalálása

Egy defektus hibát okoz, hogyha az a hiba a defektus jelenléte nélkül nem jönne létre. A hibakeresés folyamatában éppen ezért kulcsfontosságú, hogy beazonosítsuk, hogy milyen események között van ok-okozati viszony. Ha megtaláljuk az okot, akkor nagy valószínűsséggel nemsokára megtaláljuk a defektust is.

Annak érdekében, hogy valamiről kiderítsük, hogy tényleg oknak tekinthető-e, meg kell ismételnünk a történéseket, de úgy, hogy a vizsgált okot kivesszük a történésből. Ha ugyanaz történik, akkor a vizsgált ok valójában nem tekinthető tényleges oknak.

Fordított megközelítést alkalmazva, egy okot felfoghatunk úgy is, mint egy különbséget két világ között:

- Egy világ, melyben az okozat előfordul.
- Egy alternatív (másik) világ, melyben az okozat nem fordul elő.

Az okok vizsgálatánál a fenti megközelítést alkalmazhatjuk. Az a világ, amiben az okozat (hiba) előfordul kész tényként tekinthető (hiszen abból indulunk ki, hogy a hiba megtörtént). A korábban ismertetett megfigyelési módszerekkel megkeressük, hogy milyen lehetséges fertőzések vezethettek a hibához, a fertőzések pedig elvezetnek egy defektushoz. Most következik az a lépés, hogy végzünk egy kísérletet egy másik világban, amiben ez defektus nincs benne, és ha a hiba sem fordul elő, akkor valóban az a defektus okozta a hibát.

Megjegyzés: Felvetődhet az a kérdés, hogy mivel egy problémára végtelen sok megoldás létezik, ezért végtelen sok lehetőség van úgy megváltoztatni, hogy egy hiba ne forduljon elő. Mivel minden változtatás egy hiba okát tünteti el, ezért végtelen sok hibaok lehet. Következésképpen nem lehet egyértelműen kijelenteni, hogy mit nevezünk a hiba okának.

Ez a gondolatmenet helytálló, de módosítsuk úgy a fent definiált elméletet, hogy ha két lehetséges ok közül kell választanunk, akkor azt válasszuk, aminek eredményeképpen az alternatív világ a legközelebb áll az eredetihez. Ezt a megközelítést „Ockham borotvája” elvnek is nevezik.

6.8.1. Példa

Nézzük a következő C kódrészletet:

```
a = compute_value();
printf("a = %d\n", a);
```

A program végrehajtásakor a konzolon mindig `a = 0` jelenik meg, pedig tudjuk, hogy `a`-nak nem szabadna 0-nak lennie. Mi az oka annak, hogy mégis megtörténik?

Gondolkodhatunk úgy, hogy megvizsgáljuk az a változó őseit, a függőségi gráfon visszafele haladva. Láthatjuk, hogy az utolsó értékadásnál a `compute_value()` függvény áll a jobb oldalon, így arra a következtetésre juthatunk, hogy lehet, hogy ennek a függvénynek a visszatérési értéke `0`. Sajnos azonban kiderül, hogy a `compute_value()` függvény visszatérési értéke nem lehetne `0`. Ekkor még mélyebbre ásunk a `compute_value()` függvényben, hogy vajon honnan jön a fertőzés.

A fentinél egy jobb megoldás, ha érvelés helyett, ténylegesen bebizonyítjuk ok-okozati összefüggésekkel, hogy honnan jön a rossz érték. Vagyis, először meg kell mutatnunk,

hogy mivel `a` értéke `0`, ezért látjuk a konzolon kiíratva az `a = 0` kifejezést. És így tovább. Ez elég kézenfekvőnek, sőt felesleges időpocsékolásnak tűnik (talán triviálisnak is), de a programnak is elvileg működnie kéne, mégsem működik.

Az alábbiakban tehát az ok-okozati folyamatot mutatjuk be követve a tudományos kísérlet módszerét. Felállítunk egy hipotézist:

Mivel `a` *értéke* `0`*, ezért ír a program a konzolra* `a = 0`*-t*

A hipotézis alátámasztására végeznünk kell egy kísérletet („alternatív világ”), melyben `a` értéke nem `0`, és `a = 0` nem jelenik meg a konzolon. Ennek érdekében írjuk át a kódot:

```
a = compute_value();
a = 1;
printf(”a = %d\n”, a);
```

Úgy gondolkozunk, hogy ha a program ezúttal `a = 1`-et ír a konzolra, akkor valóban azért írt korábban `a = 0`-t, mert `a` értéke `0` volt. Azonban, mikor lefuttatjuk a megint a programot, ismét a már ismert `a = 0` kerül a képernyőre. Új hipotézist kell felállítanunk:

`a` *értékétől függetlenül* `a = 0` *kerül a konzolra*

A hipotézis bizonyításához állítsuk be `a` értékét tetszőlegesre, azt fogjuk tapasztalni, hogy a hipotézis teljesül. Ebből az következik, hogy valami gond van a `printf()` metódussal. És valóban, az `a` változó deklarálásánál a következőt láthatjuk:

```
double a;
…
a = compute_value();
a = 1;
printf(”a = %d\n”, a);
```

Látható, hogy a `printf()` kiiratás egy integer értéket vár, de mi egy lebegőpontos értéket adunk neki át. Innen jön a hiba, vagyis a `%d` az a defektus, ami a hibát okozta.

A példán keresztül láttuk, hogy hogyan lehet a tudományos módszerrel elkerülni, hogy rossz irányba induljunk el.

6.8.2. Izoláció

Korábban láttuk, hogy a teszteseteket hogyan egyszerűsítettük. Amikor az a célunk, hogy minél jobban leszűkítsük a különbségeket (lásd a fenti zárójeles megjegyzést), hasznos lehet az ún. izolációs eljárás alkalmazása, melynek során egy teszt-eset párt készítünk. A pár egyik része egy olyan teszt-eset, ami „passed” eredményt ad, a párja pedig „failed” eredményt. További követelmény, hogy a lehető legkisebb különbség legyen a teszt-eset pár egy-egy tagja között.

Az izolációs eljárás hasonlóképpen működik, mint az egyszerűsítési eljárás. A különbség annyi, hogy amikor egyszerűsítjük a „failed” teszt-esetet, akkor a kihagyott jellemzőket hozzáadjuk az eddigi „passed” teszt-esethez, ezáltal egy nagyobb „passed” teszt-esetet kapunk.

Összefoglalva:

- Az egyszerűsítési folyamat végén egy olyan tesztesetet kapunk, aminek minden része releváns a hiba szempontjából. Bármelyik részét is hagyjuk el, a hiba eltűnik.
- Az izolációs folyamat eredménye a teszteset egy releváns részének megtalálása. Ha elhagyjuk ezt a részt, akkor a hiba eltűnik.

Az alábbiakban ismertetjük az izolációs algoritmust:

Az izolációs algoritmus tulajdonképpen a ddmin algoritmus kibővítése az alábbiakkal:

- A helyesen lefutó teszteset $c'_{✓}$, ezt kell maximalizálnunk (inicializálás: $c'_{✓} = c_{✓}=Ø$)
- Az elbukó teszteset $c'_{✗}$, ezt kell minimalizálnunk (inicializálás: $c'_{✗} = c_{✗}$)
- Számítsuk ki a Δ_i halmazokat a következőképpen: $\Delta = c'_{✗} \backslash c'_{✓}$
- Ne csak $c'_{✗} \backslash \Delta_i$ – t, hanem $c'_{✓} \cup \Delta_i$ – t is teszteljük.
- Vezessünk be új szabályokat a helyesen lefutó, illetve az elbukó tesztesetekre.

A teljes izolációs algoritmus:

- A program futását befolyásoló jellemző-halmazt nevezzük el *konfigurációnak*. Az összes megváltozott jellemző halmazát jelöljük C-vel
- Legyen a *test*: $2^C \rightarrow \{✗, ✓, ?\}$ egy tesztelő függvény, ami egy $c \subseteq C$ konfigurációról eldönti, hogy egy adott hiba előfordul (✗), nem fordul elő (✓), vagy nem mondható meg (?).
- Legyenek $c_{✓}$ és $c_{✗}$ olyan konfigurációk, melyre teljesül, hogy $c_{✓} \subseteq c_{✗} \subseteq C$ úgy, hogy test($c_{✓}$) = ✓, és test($c_{✗}$) = ✗. $c_{✓}$ az átmenő konfiguráció, míg $c_{✗}$ az elbukó konfiguráció.
- Az általános delta debuggoló algoritmus, dd($c_{✓}$, $c_{✗}$) egy hibát okozó különbséget izolál $c_{✓}$ és $c_{✗}$ között. Visszaad egy ($c'_{✓}$, $c'_{✗}$) = dd($c_{✓}$, $c_{✗}$) párt, melyre teljesülnek az alábbiak:
  1. $c_{✓} \subseteq c'_{✓} \subseteq c'_{✗} \subseteq c_{✗}$
  2. test($c'_{✓}$) = ✓
  3. test($c'_{✗}$) = ✗
  4. $c'_{✗} \backslash c'_{✓}$ releváns különbség – vagyis nincs olyan jellemző $c'_{✗}$-ben amit elhagyva $c'_{✗}$-ből eltűnik a hiba, vagy ha ezt a jellemzőt hozzáadjuk $c'_{✓}$-hoz, akkor a hiba előjön.

A dd algoritmust definiáljuk így: dd($c_{✓}$, $c_{✗}$) = dd'($c_{✓}$, $c_{✗}$, 2), ahol dd'($c'_{✓}$, $c'_{✗}$, n) =

- ($c'_{✓}$, $c'_{✗}$) ha $|\Delta| = 1$
- dd'($c'_{✗} \backslash \Delta_i$, $c'_{✗}$, 2) ha létezik $i \in \{1..n\} \times$ test($c'_{✗} \backslash \Delta_i$) = ✓
- dd'($c'_{✓}$, $c'_{✓} \cup \Delta_i$, 2) ha létezik $i \in \{1..n\} \times$ test($c'_{✓} \cup \Delta_i$) = ✗
- dd'($c'_{✓} \cup \Delta_i$, $c'_{✗}$, max(n − 1, 2)) különben, ha létezik $i \in \{1..n\} \times$ test($c'_{✓} \cup \Delta_i$) =✓
- dd'($c'_{✓}$, $c'_{✗} \backslash \Delta_i$, max(n − 1, 2)) különben, ha létezik $i \in \{1..n\} \times$ test($c'_{✗} \backslash \Delta_i$) = ✗
- dd'($c'_{✓}$, $c'_{✗}$, min(2n, |Δ|)) különben, ha $n < |\Delta|$
- ($c'_{✓}$, $c'_{✗}$) különben, ahol

  $\Delta = c'_{✗} \backslash c'_{✓} = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$, és minden $\Delta_i \times |\Delta_i| \approx (|\Delta|/n)$ teljesül.

A rekurziós invariáns: test(c'$_{✓}$) = ✓ és test(c'$_{✗}$) = ✗ és $n \leq |\Delta|$

6.9. Hogyan javítsuk ki a defektust?

A defektus megtalálásához az alábbi (korábban ismertetett technikák) lehetnek segítségünkre:

- Ellenőrzések (assertions). A nem teljesülő ellenőrzések fertőzést jelentenek, az ilyen helyekre oda kell figyelnünk.
- Gyanús viselkedések (anomalies) keresése.
- Hibaokok keresése

Milyen sorrendben alkalmazzuk ezeket a technikákat? Először a fertőzésekre koncentráljunk, azután a hibaokokra, legvégül a gyanús viselkedésekre. Mialatt a fertőzési láncon haladunk végig, minden ponton győződjünk meg arról, hogy az adott pont őse fertőzött, és ez az ős az oka az adott pont fertőzöttségének.

Javítás után további ellenőrzések szükségesek az alábbiak miatt:

- Ellenőrizzük le, hogy a hiba valóban nem jön-e elő?
    - Ha eltűnt a hiba, akkor a javítás sikeres volt
    - Ha viszont továbbra is fennáll a hiba, akkor előfordulhat, hogy a hibát több defektus okozza, vagy amit kijavítottunk defektus az valójában nem is volt defektus az adott hibára nézve (Rosszul fejtettük vissza a fertőzési láncot).
- Ellenőrizzük, hogy a javítás miatt nem keletkeztek-e újabb hibák? Nézessük át a javításunkat, illetve alkalmazzunk regressziós tesztelést.
- Ellenőrizzük, hogy nem fordul-e elő ugyanez a típusú defektus máshol a kódban? (Az egyes programozási stílusok miatt előfordulhat, hogy egy defektushoz nagyon hasonló további defektusok lehetnek a kódban)
- Ellenőrizzük, hogy a javítást megfelelően adminisztráltuk-e?

7. Összefoglalás

Ez a jegyzet csupán betekintést nyújt a tesztelés különböző fázisaiban alkalmazható technikákra, és bizonyos osztályozási szempontok szerint segítséget nyújt a megfelelő technika kiválasztásához. Nem ismerteti viszont az alapvető tesztelési folyamatot, annak részeit, hogy abban hol, és milyen szerepük van ezeknek a technikáknak. Először tehát meg kell ismerni a keretet, ahová ezek a mozaikdarabok beilleszthetők.

A jegyzetnek szintén nem célja egy teljes képet adni az alkalmazható módszerekről, csupán néhány alapvetőt villant fel közülük. Viszont az alapvető módszerek elsajátítása után a további módszerek egyre kézenfekvőbbnek, egyszerűbbnek tűnhetnek, hasonlóan ahhoz, ahogyan további programozási nyelvek elsajátításához sokkal kevesebb idő kell, mint az elsőhöz. Nem kell tehát mást tenni, mint a megemlített módszereket alaposan begyakorolni.

Bár a tesztelés mindig is kívánni fogja az emberi kreativitást, vannak olyan területek, ahol a módszerek mechanikus részeihez segédeszközök készíthetők; és ilyen eszközök az említett módszerekhez is léteznek. Meg kell keresni ezeket a segédprogramokat, és a használatuk által gyakorolhatók, elsajátíthatók a jegyzetben leírt módszerek.

8. Felhasznált irodalom és további olvasmány

1. Paul C. Jorgensen. Software Testing:A Craftsman's Approach (second edition).CRC Press, 2002. ISBN 0-8493-0809-7
2. J. C. Huang. Software Error Detection through Testing and Analysis, John Wiley and Sons, 2009. ISBN 978-0-470-40444-7
3. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Compilers – Principles, Techniques, and Tools. Addison-Wesley, 1986. ISBN 0-201-10088-6
4. Gregg Rothermel, Roland H. Untch, Chengyun Chu, Mary Jean Harrold. Prioritizing Test Cases For Regression Testing, IEEE Transactions on Software Engineering, pp. 929-948, October, 2001
5. Nashat Mansour, Rami Bahsoon, Ghinwa Baradhi. Empirical comparison of regression test selection algorithms. Journal of Systems and Software, Volume 57, Issue 1, pp. 79-90, April 2001.
6. Emelie Engström, Per Runeson, Mats Skoglund. A systematic review on regression test selection techniques. Inf. Softw. Technol., Volume 52, Issue 1, pp 14-30, January 2010.
7. Alan. M.Turing. On Computable Numbers, With an Application to the Entscheidungsproblem. Proc. London Math. Soc., Volume 42, Issue 2, pp. 230-265, 1936.
8. Paul Ammann, Jeff Offutt. Introduction to Software Testing. Cambridge University Press, 2008. ISBN 978-0-521-88038-1
9. Arthur H. Watson, Thomas J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. NIST Special Publication 500-235, Computer Systems Laboratory, National Institute of Standards and Technology, 1996.
10. Janvi Badlaney Rohit Ghatol Romit Jadhwani: An Introduction to Data-Flow Testing, NCSU CSC TR-2006-22, 2006.
11. Glenford J. Myers, Tom Badgett, Todd M. Thomas, Corey Sandler: The art of software testing, John Wiley and Sons, 2004. ISBN 0-471-46912-2
12. Dorothy Graham, Erin Van Veenendaal, Isabel Evans, Rex Black. A szoftvertesztelés alapjai. Alvicom Kft., 2010. ISBN 978-963-06-9858-0
13. Andreas Zeller. Why Programs Fail – A guide to systematic debugging (second edition). Morgan Kaufmann Publishers. 2009. ISBN 978-0-12-374515-6
14. Wikipedia. http://en.wikipedia.org Utolsó látogatás: 2011.03.25.
15. PMD statikus elemző. http://pmd.sourceforge.net/ Utolsó látogatás: 2011.02.23.
16. Mutation Testing. http://www.simple-talk.com/dotnet/.net-tools/mutation-testing/ Utolsó látogatás: 2011.02.25.
17. MuClipse. http://cs.gmu.edu/~offutt/mujava/ Utolsó látogatás: 2011.02.25.
18. Program Correctness. http://www.bbc.co.uk/dna/h2g2/A563807 Utolsó látogatás: 2011.03.16.
19. JavaPathFinder. http://javapathfinder.sourceforge.net/extensions/symbc/doc/index.html Utolsó látogatás: 2011.02.17.