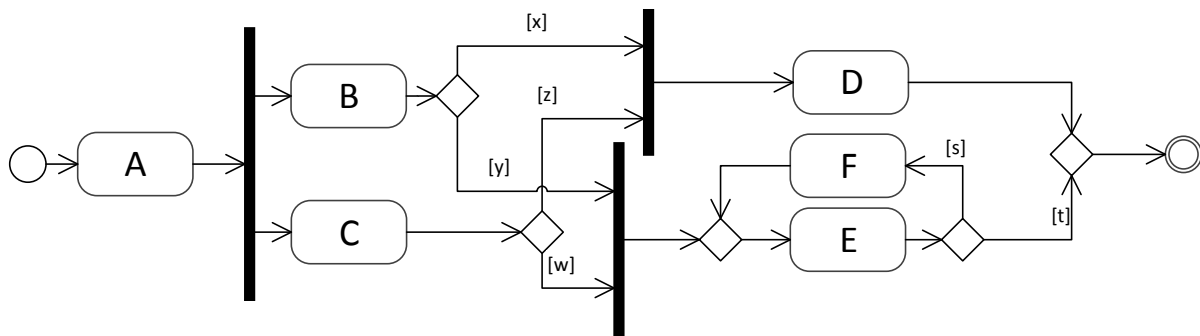


4. gyakorlat – Modellek ellenőrzése és tesztelése – Megoldások

1. Folyamat statikus analízise

Ellenőrizzük az alábbi folyamatmodellt.



- Milyen feltételek mellett teljesen specifikált a folyamat?
- Milyen feltételek mellett determinisztikus is a folyamat?
- Milyen feltételek mellett holtpontmentes is a folyamat?
- Milyen további feltételek mellett termináló a folyamat?
- Jólstrukturált-e a folyamat? Ha nem, hogyan lehetne azzá tenni? Segít-e ez a problémákon?

Megoldás

- Az állapotgépekkel analóg módon a folyamat akkor teljesen specifikált, ha minden elágazáshoz érkezéskor (decision) a kimenő élek őrfeltételei közül legalább az egyik igaz – magyarul mindig járható legalább az egyik kimenő él. Ehhez elégséges feltétel, hogy teljes feltételrendszert alkossanak az őrfeltételek, de igazából elég annyit megkövetelni, hogy feltételesen teljes rendszer legyen, tehát ha odakerülhet a vezérlés, akkor álljon fenn, hogy legalább az egyik kimenő igaz (a harmadik decisionnél ez számít).
 - Következésképp:
 - $x \vee y$
 - $z \vee w$
 - $w \wedge y \rightarrow s \vee t$ (a jobb oldal a ciklus bárhány végrehajtása után igaz).
- A folyamat akkor determinisztikus, ha minden elágazáshoz érkezéskor (decision) a kimenő élek őrfeltételei közül legfeljebb az egyik igaz – magyarul mindig csak az egyik kimenő él járható. Ehhez elégséges feltétel, hogy kizárólagos feltételrendszert alkossanak az őrfeltételek, de igazából elég annyit megkövetelni, hogy feltételesen kizárólagos rendszer legyen, tehát ha odakerülhet a vezérlés, akkor álljon fenn, hogy legfeljebb az egyik kimenő igaz (a harmadik decisionnél ez számít).
 - Következésképp:
 - $\neg(x \wedge y)$
 - $\neg(z \wedge w)$
 - $w \wedge y \rightarrow \neg(s \wedge t)$ (a jobb oldal a ciklus bárhány végrehajtása után igaz)
 - Az a) feladatrésszel összesítve némileg egyszerűsíthetjük az első két kritériumot:
 - $y = \neg x$ (ugyanis x és y nem lehetnek egyszerre igazak és nem lehetnek egyszerre hamisak, tehát x XOR y)
 - $w = \neg z$
- Holtpont (deadlock): örök várakozás. Először is nyilván feltesszük, hogy maguk az elemi tevékenységek holtpontmentesek – ha nem így lenne, akkor nem a folyamatmodell a holtpont forrása. Holtpont itt úgy fordulhat elő, hogy a fork után az egyik ág a fenti, a másik ág a lenti joint választja, és örökké várnak egymásra. Baj van, ha az egyik joinba csak az egyik ág fut be.
 - Következésképp:
 - $x = z$
 - $y = w$
- Először is nyilván feltesszük, hogy maguk az elemi tevékenységek terminálnak – ha nem így lenne, akkor nem a folyamatmodell a nemterminálás forrása. A folyamatmodellben probléma lehet, ha

a join örökké vár – de a holtpontra már kizártuk. Utolsó lehetőségként marad a livelock, vagyis a végtelen ciklus. Ebben akkor ragadunk bele, ha ráfutunk, és a kilépési feltétel sose válik igazzá.

- Következésképp:

- Ha $\neg x$ (ilyenkor y és w igaz), akkor *előbb-utóbb* t -nek igazzá kell válnia. Érdekeség: ahogy a fenti állításokat, úgy ezt is le lehet írni logikai formulaként (van *előbb-utóbb* szimbólum, a *future* operátor (F)), de az ehhez szükséges ún. temporális logikákat nem tanultuk.
- $\neg x \rightarrow Ft$

e) A tanult módszerrel megvizsgálva kiderül, hogy nem jólstrukturált. Azzá lehet tenni, ha B és C után van egy join, és utána egyetlen decision. Azzal a feltételezéssel tudjuk jólstrukturálttá tenni, hogy a c) feladatban megállapított összefüggések igazak. Ez a deadlockot automatikusan kiküszöböli, a többi hibalehetőséget viszont nem – legfeljebb a modell átláthatóbbá tételével segít –, így pl. livelock megmaradhat, determinizmust nem garantál.

2. Dinamikus analízis teszteléssel

Az $f()$ függvénnyel szemben a következő *követelményeink* vannak:

R1 Az $f()$ függvénynek minden végrehajtása során legalább egyszer outputot kell kiadnia.

R2 Az $f()$ függvénynek tetszőleges inputsorozat esetén terminálnia kell.

R3 Az $f()$ függvény végrehajtása során kiadott legutolsó output értéke kötelezően 0.

A függvény egy lehetséges megvalósítását adja meg az alábbi C nyelvű kódrészlet:

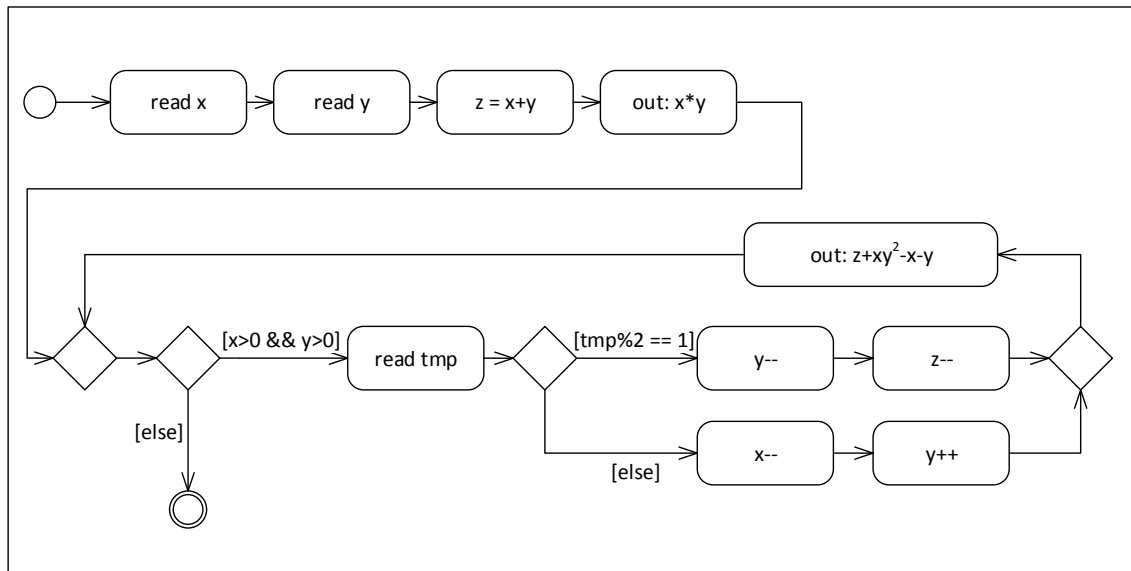
```

1 int readInput();
2 void writeOutput(int out);
3
4 void f() {
5     int x = readInput();
6     int y = readInput();
7     int z = x + y;
8     writeOutput(x * y);
9     while (x > 0 && y > 0) {
10         if (1 == readInput() % 2) {
11             y--;
12             z--;
13         } else {
14             x--;
15             y++;
16         }
17         writeOutput(z + x * y * y - x - y);
18     }
19 }
```

A következő lépések során ellenőrizzük a függvény működését!

a) Ábrázoljuk folyamatmodellként $f()$ vezérlési folyamatát!

Megoldás. Készítsük el a folyamatmodellt.



b) Miért lehetünk biztosak az R1 teljesülésében?

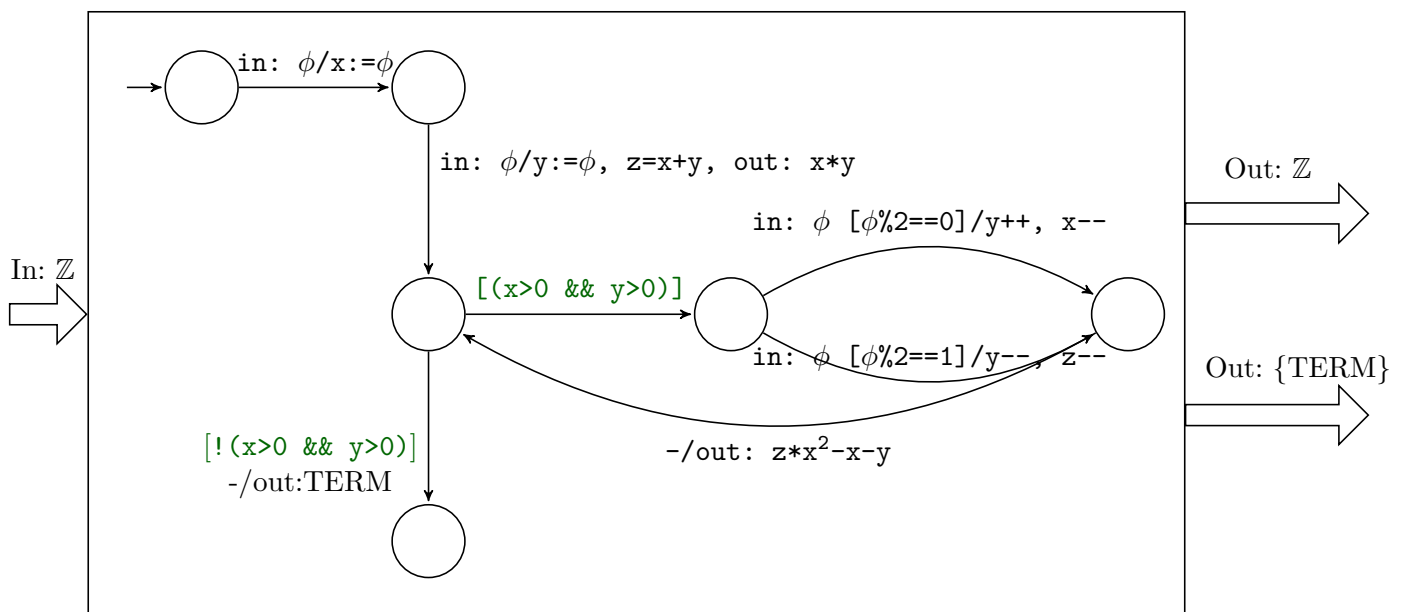
Megoldás. A folyamatmodellen jól látszik: minden út átmegy az első output adási tevékenységen.

c) Miért lehetünk biztosak az R2 teljesülésében?

Megoldás. Pozitív x és y esetén vagyunk a ciklusban; itt x nem nőhet, ezért a második ág csak véges sokszor hajtható végre; így pedig egy idő után csak az első ágat hajthatnánk végre, ahol y előbb-utóbb elfogy. (Van még az az eset is, hogy y negatívba túlsordul, de akkor is rögtön leáll.)

d) (Kiegészítő feladat.) Építsünk olyan állapotgépet, amely az $f()$ függvénnyel ekvivalens módon működik. Modellezzük a `readInput()` hívásokat input csatornaként, valamint a `writeOutput()` hívást output csatornaként. Az $f()$ függvény terminálását modellezzük úgy, hogy az automata ad egy speciális outputot, és átmegy egy nyelő (kimenő átmenet nélküli) állapotba.

Megoldás. Tanultunk állapotváltozókat, legyen ilyen az x, y, z . Írjunk az átmeneti élekre őrfeltételeket ezek segítségével, meg akciókat (mint a Yakinduban is). Lehet minden programsorra egy állapot, de lehet sokkal kevesebb is, pl. az egész ciklusmag lehet két hurokél egyazon állapot fölött (kicsit szebb annyira széjbontani, hogy először a ciklusfeltételt teszteljük, utána a belső elágazást); ezek mind helyes megvalósítások lesznek.



e) Az R3 követelményt teszteléssel ellenőrizzük. A $t_1 = \langle 2, 3, 5, 7, 11, 13, \dots \rangle$ input szekvencia a tesztesetünk. Detektálunk-e hibát?

Megoldás. Mit csinál a program?

- $t_1 = \langle 2, 3, 5, 7, 11, 13, \dots \rangle$

- $x = 2$
- $y = 3$
- $z = 5$
- out: 6
- belépünk a ciklusba
- true ág
- $y = 2$
- $z = 4$
- out: $4 + 8 - 2 - 2 = 8$
- bent maradunk a ciklusban
- true ág
- $y = 1$
- $z = 3$
- out: $3 + 2 - 2 - 1 = 2$
- bent maradunk a ciklusban
- true ág
- $y = 0$
- $z = 2$
- out: $0 + 2 - 2 - 0 = 0$
- kilépünk a ciklusból
- futás vége

f) Számítsunk *utasításszintű tesztfedést* a programkódon, vagyis hogy az utasítások mekkora hányadát járja be a tesztelt függvény a tesztelés végrehajtása során! Hogy jelenik meg ez a mérőszám a vezérlési folyamaton?

Megoldás. 10 utasítást hajt végre és 2 utasítást kihagy $10/12 \approx 83\%$ -os utasításfedés. A vezérlési folyamaton a csomópontokat lehet karikázni, és a csomópont szintű fedettséget jelenti (kis eltéréssel / korrekcióval, mert a decision-merge pár csak egyszer számított utasításnak).

g) Az R3 követelményhez a $t_2 = \langle 1, 2, 4, 1, 2, 4, \dots \rangle$ input szekvencia a második tesztelésünk. Detektál-e hibát ez a tesztelés? Mekkora a két tesztből álló tesztkészlet együttes utasításfedése?

Megoldás. $t_2 = \langle 1, 2, 4, 1, 2, 4, \dots \rangle$ Mit csinál a program?

- $x = 1$
- $y = 2$
- $z = 3$
- out: 2
- belépünk a ciklusba
- false ág
- $x = 0$
- $y = 3$
- out: $3 + 0 - 0 - 3 = 0$
- kilépünk a ciklusból.
- futás vége

Tehát nem sérült az R3. A tesztelés utasításfedése 100%.

h) (Kiegészítő feladat.) Milyen tesztfedettségi metrika számítható a korábban megépített állapotgép alapján?

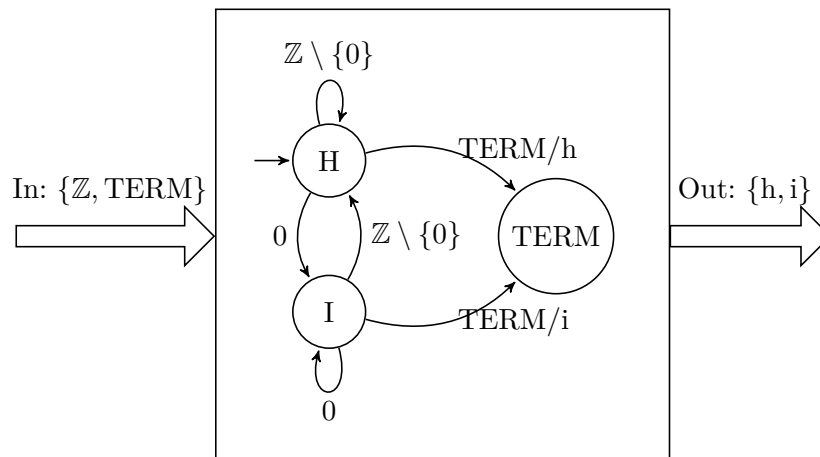
Megoldás. Az állapotgépen fedettség szempontjából hasonló a helyzet, mint a kódban, amennyiben elég közeli formában építettük fel – de a javasolt összevonások után jóval egyszerűbb, nem látszik rajta ez a fedettségi mérték (van persze állapotfedettség, de az most akár 100% is lehet; megfelelő összevonásokkal itt csak az átmenetfedettség marad el a 100%-tól).

i) Készítsünk olyan *tesztorákulum* állapotgépet, amely $f()$ input és output szekvenciái és terminálása alapján el tudja dönteni, hogy az adott lefutás során az R3 követelmény sérült-e! Hogy viselkedik az orákulum a fenti tesztinputra?

Megoldás.

A tesztorákulum a tesztbemenet és a valós kimenet alapján eldönti a teszt eredményét és I/H-t ad vissza. Az automata két legfontosabb állapota, hogy I („utolsó input 0”) és H („utolsó input nem 0”), ezek között értelemszerűen ugrál. A SUT terminálódásának eseményére pedig átmegegy a nyelő állapotba, és rendre igaz („i”), ill. hamis („h”) kimenetet ad. (A kezdőállapot működése

nincs teljesen specifikálva, lehet mondjuk az „utolsó input nem 0” állapot). Az orákulumon az outputok és a terminálás alapján végigjátszható, hogy (az elvárásoknak megfelelően) „i” outputot ad ki.



- j) Adjunk meg egy tesztesetet, amely kimutat egy hibát a programban! Milyen elv alapján sejtettük volna meg, hogy a korábban összeállított tesztkészletünk kiegészítésre szorul?

Megoldás. Egy lehetséges teszteset: $t_3 = \langle -1, -1, -1, \dots \rangle$. Mit csinál a program?

- $x = -1$
- $y = -1$
- $z = -2$
- out: 1
- nem lépünk be a ciklusba
- futás vége

Ez megsérti az R3 követelményt, az orákulum fölön is csípi.

Tanulság: nem elég az utasítás vagy ág szintű fedettséget nézni, a *robosztusságot* is meg kell vizsgálni a szélső vagy kritikus inputértékekre és közvetlen környezetükben (jelen esetben ilyen a 0 az x és y esetén). Ez nagyon hasznos lesz a programozásos házi feladatoknál!

Ez egy dinamikus ellenőrzési módszer, mert konkrétan *végre kellett hajtanunk* a lépéseket.

- k) (Kiegészítő feladat.) Vegyük hozzá a tesztkészlethez a $t_3 = \langle 0, 1, 2, 3, 4, 5, \dots \rangle$ és $t_4 = \langle 1, 2, 3, 4, 5, 6, \dots \rangle$ input szekvenciákat mint további teszteseteket! Detektálunk-e hibát? Hogyan változnak a tesztfedési számok?

Megoldás. Otthoni munka.

- l) (Kiegészítő feladat.) Határozzuk meg, hogy pontosan milyen input szekvenciák esetén sérül R3, és javasoljunk hibajavítást!

Megoldás. Lássuk csak!

- triviális megoldás a `void f() { writeOutput(0); }`, de inkább olyan megoldást keressünk, ami legalább nyomokban őrzi az eredeti működést.
- ha x és y közül az egyik negatív és a másik nem nulla, és nem lépünk be a ciklusba, akkor $x \cdot y$ lenne az utolsó output – ilyenkor el kell dönteni, mi a helyes viselkedés, pl. a ciklus helyett mondjuk ki lehetne adni egy 0 outputot. Vagy eleve elutasítani a negatív bemenetet, ha az érvénytelennek számít.
- $z = x + y$ végig igaz, így a ciklusmagban kiadott outputból egyedül a szorzattag marad.
- ha belépünk legalább egyszer a ciklusba, akkor a ciklus utolsó lefutása után, feltéve ha nem volt negatívba overflow, akkor x vagy y 0 lett, tehát a szorzatuk is 0, ez eleve stimmel.
- azt kell tehát még kivédeni, hogy ne csorduljon túl y MAXINT-ból negatívba eldöntendő, hogy ilyenkor mi a teendő, pl. egy magas küszöbszám feletti y értékek esetén hibát adunk és leállunk, vagy detektáljuk az overflow-t és kiadunk 0-t stb.