Kölcsönös kizárási algoritmusok helyességének bizonyítása

Készítette: **Horányi Gergő** és **Jeszenszky Balázs**

2010.

Tartalom

1 A kölcsönös kizárási feladat

A kölcsönös kizárás a konkurens programozás és az elosztott rendszerek fejlesztésének klasszikus problémája. A lényege a következő [1]: Processzek egy csoportjának egy megosztott erőforrást kell elérnie, amelyet egyidejűleg legfeljebb egy processz használhat. A megoldásnak a következő tulajdonságokkal kell rendelkeznie.

- *Kölcsönös kizárás*: Minden processznek végre kell hajtania egy kritikus szakasznak nevezett kódszegmenst úgy, hogy bármely adott időpillanatban legfeljebb egy processz hajthatja azt végre (azaz egy processz van a kritikus szakaszban).
- *Holtpontmentesség*: Ha egy vagy több processz megkísérel belépni a kritikus szakaszba, valamikor végül egy sikerrel jár, feltéve, hogy egy processz sem tartózkodik a kritikus szakaszban örökké.
- *Kiéheztetésmentesség*: A kritikus szakaszba belépni kívánó processz valamikor végül sikerrel jár, feltéve, hogy egy processz sem tartózkodik a kritikus szakaszban örökké.

Ennek a problémának az eredeti megoldásai olyan speciális szinkronizációs eszközökre támaszkodnak, mint a szemafor vagy a monitor. A feladat keretein belül olyan elosztott algoritmusokat vizsgálunk, amelyek csak közönséges megosztott változókat használnak, tehát a kölcsönös kizárást a megosztott változókon végrehajtott írás illetve olvasás műveletekkel kell megvalósítani.

Feltesszük, hogy egy processz programja az alábbi szakaszokra bomlik:

- Belépő (próbálkozó): a kritikus szakaszba való belépés előkészítése.
- Kritikus: az egyidejű végrehajtástól megvédendő szakasz.
- Kilépő: a kritikus szakasz elhagyásakor végrehajtott kód.
- Fennmaradó: a kód többi része.

Egy-egy processz rendre a Belépő – Kritikus szakasz – Kilépő – Fennmaradó kódrészleteket hajtja végre ciklikusan. Feltesszük, hogy nincs olyan processz, ami állandóan a kritikus szakaszban tartózkodik.

A következőkben az a célunk, hogy a megadott algoritmusokat modellezzük az UPPAAL eszköz által támogatott formalizmust használva, majd megpróbáljuk a fenti elvárt tulajdonságok (mint követelmények) teljesülését az UPPAAL modellellenőrző komponense segítségével igazolni. Ha egy követelmény nem teljesül, akkor pedig elemezzük a modellellenőrző által adott ellenpéldát.

2 Hyman algoritmusának ellenőrzése

2.1 Az algoritmus

Harris Hyman 1966-ban javasolta a következő algoritmust [2]. Legyen két processz P1 és P2, a használt megosztott változók pedig a következők:

- *blocked0*: Az első processz (P1) be akar lépni;
- *blocked1*: Második processz (P2) be akar lépni;
- *turn*: Ki következik belépni (0 esetén P1, 1 esetén P2).

Az algoritmusok a két processz esetén a következők:

P1 processz	**P2 processz**

while (true) {
blocked0 = true;
while (turn!=0) {
while (blocked1==true) {
skip;
}
turn=0;
}
// Critical section here
blocked0 = false;
// Other things
}
	while (true) {
	blocked1 = true;
	while (turn!=1) {
	while (blocked0==true) {
	skip;
	}
	turn=1;
	}
	// Critical section here
	blocked1 = false;
	// Other things
	}

Az a célunk, hogy megvizsgáljuk, az algoritmus teljesíti-e az alapvető követelményeket (holtpontmenteség, kölcsönös kizárás, kiéheztetésmentesség). Ha valamelyik követelmény nem teljesül, megpróbáljuk megmutatni, hogy miért (milyen példa adható a követelmény megsértésére).

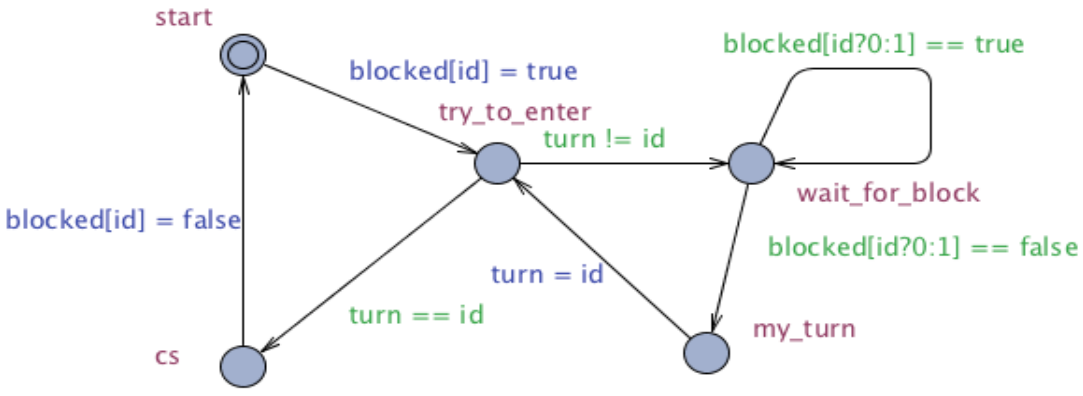
2.2 A modell elkészítése

A megosztott változókat az UPPAAL modellben két globális változóként jelenítem meg:

```
bool blocked[2] = {false, false};

int[0,1] turn = 0;
```

A processzeket modellező template egy paraméterrel is rendelkezik, amely a processz sorszámát adja meg (*id*).



A kölcsönös kizáráson kívüli részt a *start* hely jeleníti meg. Amikor egy processz jelzi a belépési szándékát, akkor belép a *try_to_enter* állapotba (és a *blocked* tömb megfelelő változóját módosítja). Ezek után, ha épp a *turn* változó nem rá mutat, akkor a *wait_for_block* állapotba kerül, ahol addig vár, amíg a másik processz hamissá nem állítja a blokkolását. Ezután a *my_turn* állapoton keresztül a *try_to_enter*-be visszatérve a *turn* változót a saját sorszámára állítja. Ez a három hely valósítja meg az algoritmus belépő (próbálkozó) szakaszát. Ha a *turn* változó a saját sorszámát tartalmazza, akkor beléphet a kritikus szakaszba, amiből a kilépés során a blokkolást „le kell vennie” magáról.

Az elkészült modellt a csatolt *Hyman.xml* fájl tartalmazza.

2.3 A követelmények ellenőrzése

✓ Holtpontmentesség

A[] (! deadlock) - A modell holtpontmentes.

X Kölcsönös kizárás

A[] (! (P0.cs && P1.cs)) - A modell nem felel meg a kölcsönös kizárás elvárásának.

Ellenpélda:

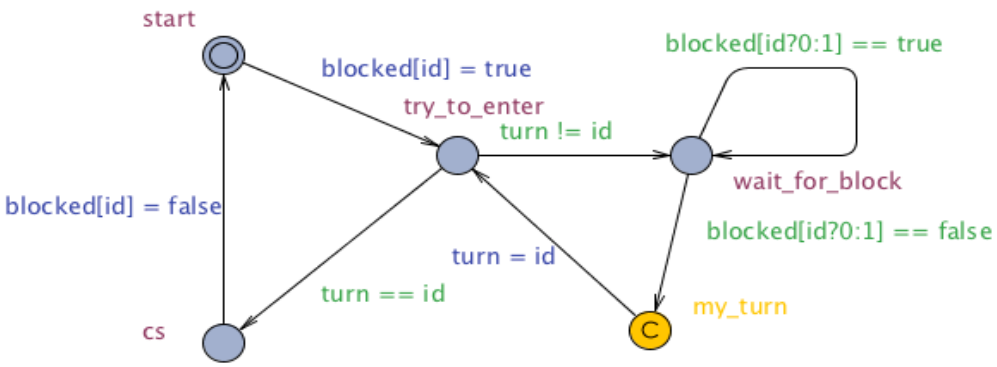
- P1 processz megpróbál belépni (blokkolás).
- Mivel épp nem P1 jön (P1 alapból 0 értékű), ezért P1 várakozó állásba kerül.
- P1 a várakozó állásból azonnal kilép, átkerül a *my_turn* állapotba.
- P0 megpróbál belépni (blokkol).
- P0 beléphet a kritikus szakaszba.
- P1 visszatér a belépési ponthoz, és a *turn* változót átállítja.
- P1 belép a kritikus szakaszba.

X Kiéheztetésmentesség

Mivel a modell nem egy helyes kölcsönös kizárás, így a kiéheztetésmentességet nem feltétlenül érdemes vizsgálni. Természetesen kiéheztetésről nem is beszélhetünk, hiszen mindkét processz egyszerre tartózkodhat a kritikus szakaszban.

2.4 Konklúzió

Úgy vélem, hogy az algoritmus hibája, hogy a blokkolás ellenőrzése és a *turn* változó beállítása nem atomi jellegű, ennek következtében az algoritmus nem helyes. Amennyiben a *my_turn* helyet *committed*-nek állítanám be, akkor a probléma megoldódna. Ennek bizonyítására elkészítettem a következő modellt:



Ezen ellenőriztem a korábbi feltételeket, és a várt eredményt kaptam. A gond ezzel a megoldással az, hogy nem minden platformon megvalósítható, hiszen speciális atomi végrehajtására van szükség.

3 Peterson algoritmusának ellenőrzése

3.1 Az algoritmus

Peterson 1981-ben adott egy módosított algoritmust a következők szerint [3]:

P1 processz	**P2 processz**

while (true) {
blocked0 = true;
turn=1;
while (blocked1==true &&
turn!=0) {
skip;
}
 	
// Critical section here
blocked0 = false;
// Other things
}
	while (true) {
	blocked1 = true;
	turn=0;
	while (blocked0==true &&
	turn!=1) {
	skip;
	}
	
	// Critical section here
	blocked1 = false;
	// Other things
	}

A modellezés során ügyelnünk kell arra, hogy a belső *while* ciklus feltételvizsgálata nem atomi jellegű.

Az algoritmus tulajdonságait a következők szerint vizsgáljuk meg:

- Ellenőrizzük, hogy ez a módosított algoritmus teljesíti-e az alapvető követelményeket (holtpontmenteség, kölcsönös kizárás, kiéheztetésmentesség).
- Ellenőrizzük, hogy az algoritmus teljesíti-e a kölcsönös kizárás követelményét, ha a belső *while* ciklus feltételvizsgálat részében felcseréljük az *&&* operátorral összekötött két vizsgálat sorrendjét.
- Az algoritmus wikipedia lapján[1] a következő olvasható (a változók neveit értelemszerűen átírva): *If P1 is in its critical section, then blocked0 is true and either blocked1 is false or turn is 0.* Megpróbáljuk bebizonyítani ezt az állítást.
- Peterson algoritmusának jellemzője, hogy egy processz esetén a kritikus szakaszba való belépéshez szükséges várakozási idő korlátos, amennyiben feltesszük, hogy minden processz korlátos időt tölt el a kritikus szakaszban. Megpróbáljuk ezt is bizonyítani az UPPAAL segítségével.

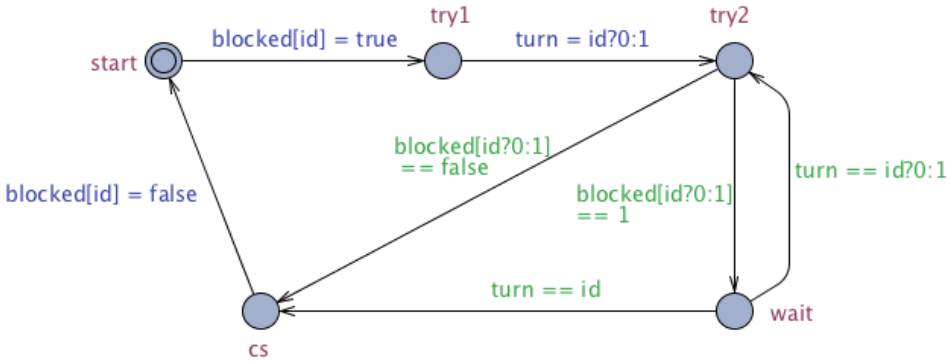
3.2 A modell elkészítése

A modellezés rendkívül sokban hasonlít a Hyman algoritmusnál korábban leírtakhoz. Ugyanazokat a globális változókat hoztam létre, azonban némileg más állapotokat kellett létrehoznom. Az algoritmus belépő, próbálkozó szakaszáért itt is 3 állapot felelős: *try1*, *try2* és *wait*. A két *try* állapot csak a belépéshez szükséges változókat állítja be. Ezután az algoritmus várakozó *while* ciklusát valósítom meg, amelynél nagyon fontos, hogy a ciklusfeltételek ellenőrzése nem atomi művelet. Ha a feltétel első fele nem teljesül, akkor máris átléphetünk a kritikus szakaszba, ha azonban az első fele még teljesül, akkor külön meg kell nézni a második felét is. Ha az nem teljesül, akkor szintén

[1] http://en.wikipedia.org/wiki/Peterson's_algorithm

beléphetünk a kritikus szakaszba, ha azonban az is teljesül, akkor az algoritmus tovább várakozik, és újra ellenőriz. A *wait* hely tulajdonképpen azt a köztes állapotot jeleníti meg, amely a feltétel első és második felének ellenőrzése között van.

Az elkészült modellt a *Peterson.xml* fájl tartalmazza.



3.3 A követelmények ellenőrzése

✓ Holtpontmentesség

A[] (! deadlock) - A modell holtpontmentes.

✓ Kölcsönös kizárás

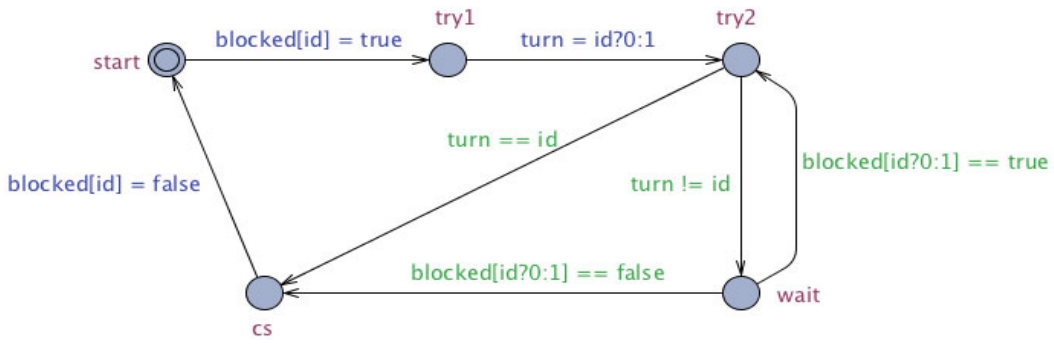
A[] (! (P0.cs && P1.cs)) - A kölcsönös kizárás teljesül.

✓ Kiéheztetésmentesség

A kiéheztetésmentesség azt jelenti, hogy abban az esetben, hogy ha egy processz belép a *try* részbe, akkor egy idő után beléphet a kritikus szakaszba. UPPAAL segítségével ez nehezen ellenőrizhető, mert véges lefutású modelleket alkotunk, azaz ha a *(P0.try1 || P0.try2 || P0.wait) --> P0.cs* temporális logikai kifejezéssel szeretnénk az algoritmus kiéheztetésmentességét ellenőrizni, egy triviális ellenpéldát kapunk, ahol a P0 processz a *try1* állapotba belép, majd a lefutás megáll. A konkrét esetben azonban ismerjük, hogy az algoritmus várakoztatására van egy felső korlát (lásd később), így az algoritmus kiéheztetésmentes.

✓ Felcserélt *while* feltételek

A módosított UPPAAL modell:



A felcserélt feltételekkel a holtpontmentesség és a kölcsönös kizárás továbbra is teljesül.

X A wikipedia állítása:

If P0 is in its critical section, then blocked[0] is true and either blocked[1] is false or turn is 0.

Az állítás nem igaz és ezt a következő temporális logikai kifejezéssel bizonyítom:

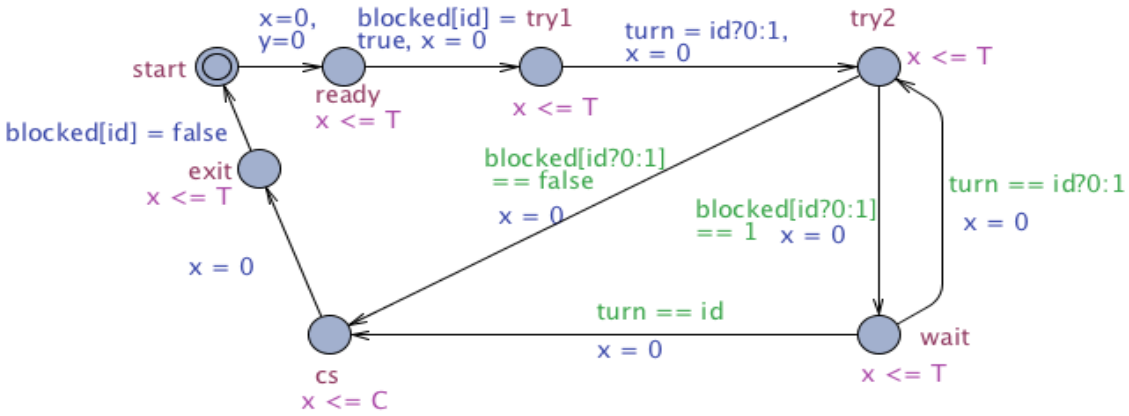
A[] (P0.cs imply (blocked[0] == true && (blocked[1] == false || turn==0)))

Ellenpélda:

- P0 belép a *try1* állapotba, és blokkol.
- P0 belép a *try2*-be.
- P0 behalad a kritikus szakaszba (*blocked[0]* igaz, *turn == 1*).
- P1 belép a *try1* állapotba, amivel a *blocked[1]* értéke *true* lesz, tehát az állítás megdől.

✓ Korlátos várakozási idő

Ehhez az elkészített modellemet módosítottam (ld. a *Peterson_timed.xml* fájlban):



A modellben létrehoztam minden processzhez két-két lokális órát (x és y) valamint globálisan létrehoztam a C és T konstansokat (értékük tetszőleges lehet). Invariánsok segítségével beállítottam, hogy minden helyen korlátos ideig tartózkodjon. A várakozó szakaszokban T ideig, a kritikus szakaszban C ideig tartózkodhat.

Ezek után a következő kifejezéssel ellenőrizve bizonyítom, hogy a várakozás idő korlátos:

*A[] (P1.ready || P1.try1 || P1.try2 || P1.wait) imply P1.y <= C+10*T*

valamint

*A[] (P0.ready || P0.try1 || P0.try2 || P0.wait) imply P0.y <= C+10*T*

Az is belátható, hogy a C+10*T a várakozási idő legkisebb felső korlátja. Ennek belátásához a kifejezéseket az alábbi módon módosítottam:

*A[] (P1.ready || P1.try1 || P1.try2 || P1.wait) imply P1.y <= C+10*T-1*

valamint a P0 processzre is ugyanez. Míg az első kettő állítás az UPPAAL szerint teljesül, a második kettő már nem. Az is megállapítható, hogy nem feltétlenül kell ennyi idő az algoritmusnak. Ezt az alábbi kifejezéssel ellenőrzöm:

E[](P0.y <= 0)

azaz van olyan lefutás, hogy 0 idő alatt körbeér egy processz.

4 Lamport algoritmusának ellenőrzése

4.1 Az algoritmus

Leslie Lamport 1987-ben javasolt egy algoritmust a kölcsönös kizárásra [4]. Ennek egyik előnye volt, hogy lehetőséget adott versenyhelyzet nélkül a kritikus szakaszba való „gyorsabb" belépésre. (Egy kölcsönös kizárás algoritmust „gyorsnak" nevezünk, ha egy processz egy konstans számon belüli lépésszámmal lép be a kritikus szakaszba, amikor ez az egyetlen, a kritikus szakaszba belépni próbáló processz.) Ezt az algoritmust először 2 processz esetére vizsgáljuk, majd megpróbáljuk több processz esetére is kiterjeszteni.

A két processz legyen P1 és P2, azonosítójuk értelemszerűen 1 és 2. A megosztott változók legyenek a következők:

- *Igény1* illetve *Igény2* jelzik, ha P1 illetve P2 processz be akar lépni a kritikus szakaszba.
- A *Rögtön* változó segítségével vizsgálható a gyorsabb belépés lehetősége, míg a *Lassabban* változó segítségével a lassabb belépés lehetősége.

Kezdetben minden változónak 0 az értéke. Ezt a 4 közös változót használva az algoritmus a következő:

1. Az adott processz jelzi a neki megfelelő *Igény* változóját (*Igény1* illetve *Igény2*) 1-be állítva, hogy be akar lépni, majd a *Rögtön* változóba saját azonosítóját írja.
2. Kiolvassa a *Lassabban* változó értékét. Ha ez nem 0, akkor az *Igény* változóját 0-ba állítja, és addig vár, míg a *Lassabban* értéke 0 nem lesz. Ezután újrakezdi az algoritmust az 1. lépésben.
3. Ha a *Lassabban* változó értéke 0 volt, akkor ebbe saját azonosítóját írja.
4. Ezután megnézi, hogy mi van a *Rögtön* változóban. Ha nem a saját azonosítója, akkor a következőket teszi: Az *Igény* változóját 0-ba állítja, majd addig vár, amíg a másik *Igény* is 0 lesz. Ha ekkor a *Lassabban* változóban nem a saját azonosítója van, akkor megvárja, míg az 0 nem lesz, és akkor újrakezdi az algoritmust az 1. lépésben, máskülönben belép a kritikus szakaszba.
5. Ha a 4. pontban a *Rögtön* változóban a saját azonosítója volt, akkor belép a kritikus szakaszba.
6. A kritikus szakaszból való kilépéskor a *Lassabban* változót valamint saját *Igény* változóját is 0-ba állítja.

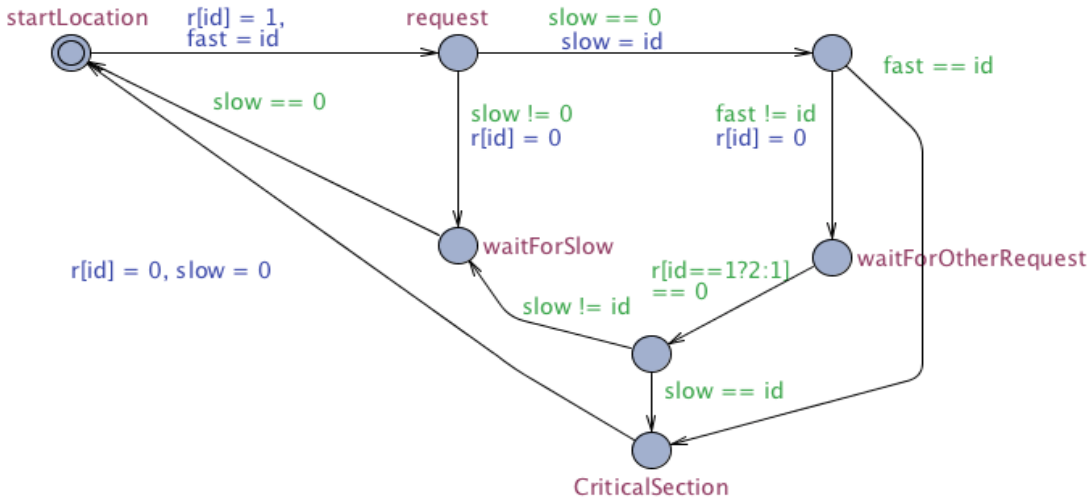
Megvizsgáljuk Lamport algoritmusa esetén is, hogy teljesíti-e az alapvető követelményeket (holtpontmenteség, kölcsönös kizárás, kiéheztetésmentesség). Ha valamelyik követelmény nem teljesül, megmutatjuk, hogy miért (milyen példa adható a követelmény megsértésére).

Ezután úgy módosítjuk az algoritmust, hogy 3 processz esetén is biztosítsa a kölcsönös kizárást. Megmutatjuk a fenti követelmények teljesülését ez esetben is.

4.2 A modell elkészítése

A modellezés során az algoritmus leírását teljes mértékben követtem. A leírtak szerint létrehoztam egy *Rögtön* (*fast*) és egy *Lassabban* (*slow*) változót, valamint egy-egy igény változót a két processznek (*r[]* tömb). Az algoritmus futása során kétszer is kerülhet olyan állapotba, hogy várakozik a *Lassabban* változó értékének 0-ra változására. Ezt a két állapotot a modellemben összevontam egy *waitForSlow* állapottá.

A modell a *Lamport.xml* fájlban megtalálható.



4.3 A követelmények ellenőrzése

✓ Holtpontmentesség

A[] (! deadlock) - A modell holtpontmentes.

✓ Kölcsönös kizárás

A[] (! (P0.cs && P1.cs)) - A kölcsönös kizárás teljesül.

X Kiéheztetésmentesség

A következő temporális kifejezés fogalmazza meg a kiéheztetésmentesség kritériumát:

P0.request --> P0.CriticalSection

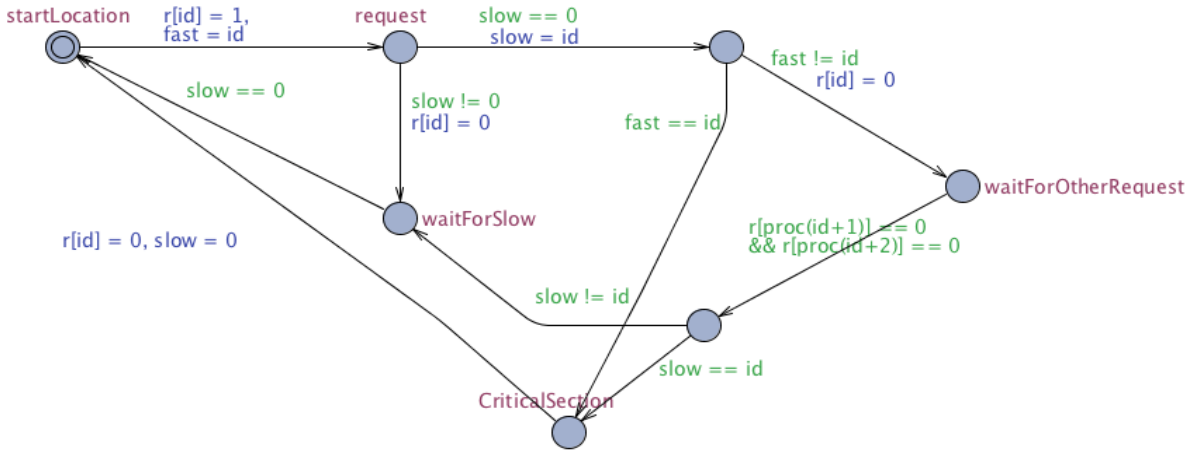
Ez sajnos az UPPAAL szerint nem teljesül, mert adódik egy triviális ellenpélda, miszerint a P0 processz belép a *request* állapotba, majd a modell futása megáll, hiszen nincs semmi, ami egy állapotban való tartózkodási időt korlátozná. Ehhez a modellt módosítani kellene.

A szimulátor használatával a következő példa található a kiéheztetésre:

- P0 belép a *request* állapotba.
- Ezután a kritikus szakasz felé haladva beállítja a *slow* változó értékét 1-be.
- Még mielőtt P0 belépne a kritikus szakaszba P1 is belép a *request* állapotba.
- P1 átlép a *waitForSlow* állapotba.
- P0 belép a *waitForOtherRequest* állapotba.
- P0 beléphet a kritikus szakaszba.
- P0 elhagyja a kritikus szakaszt, átlép a *startLocation* állapotba. Eközben a P1 processz nem mozdul el.
- P0 tovább fut és folyamatosan beléphet a kritikus szakaszba, így a másik processz kiéheztetése megtörténhet!

4.4 A követelmények ellenőrzése 3 processz esetén

Az algoritmus módosítottam 3 processzre. A változás mindössze az volt, hogy a *waitForOtherRequest* állapotból kimutató éleken mindkét processz igény változóját ellenőrzöm.



A modell megtalálható a *Lamport_3proc.xml* fájlban.

A modellben felhasznál *proc* függvény rendkívül egyszerű:

```
int proc(int id) {
        if (id == 4)
                return 1;
        else if (id == 5)
                return 2;
        else
                return id;
}
```

A függvényt azért készítettem, mert így a legegyszerűbb a kapcsolatot megteremteni az *r* tömb és az *id* változók között.

Ezek után még ellenőriztem, hogy továbbra is holtpontmentes maradt-e a modell:

A[] (!deadlock)

majd azt, hogy kölcsönös kizárás teljesül-e

A[] ((!(P0.CriticalSection & P1.CriticalSection)) & (!(P2.CriticalSection & P1.CriticalSection)) & (!(P0.CriticalSection & P2.CriticalSection)) & (!(P0.CriticalSection & P1.CriticalSection & P2.CriticalSection)))

Mindkét feltételre kimutatja a modellellenőrző, hogy teljesül.

5 Hivatkozások

1. Iványi Antal (szerkesztő): Informatikai algoritmusok II. ELTE Eötvös Kiadó, 2005.
2. H. Hyman: Comments on a problem in concurrent programming control. Communications of the ACM, 9(1):45, 1966.

3. G. L. Peterson: Myths about the mutual exclusion problem. Information Processing Letters, 12(3):115-116, 1981.
4. L. Lamport. A fast mutual exclusion algorithm. ACM Transactions on Computers, 5(1):1-11, 1987.