

A modell-ellenőrzés gyakorlata

SPIN modell-ellenőrző rendszer

Gerard J. Holzmann – 1991-től a rendszer elérhető
2002-ben ACM Szoftver Rendszer Díj;

- ipari fejlesztésekben sikeres alkalmazás (pl. PathStar telefonközpont feldolgozó szoftver ellenőrzése, NASA űrkutatási projekteken missziókritikus szoftverek algoritmusainak ellenőrzése);
- konkurens rendszerek (kommunikációs protokollok) ellenőrzésére készült;

Promela (Process Meta Language)

- specifikációs nyelv véges állapotú rendszerek modellezésére;
- folyamatok közötti kommunikáció:
szinkron (randevú) vagy aszinkron (pufferelt);
- folyamatok dinamikus létrehozása;
- szintaxisában a C programozási nyelvre hasonlít
(pl. boole és aritmetikai operátorok szintaxisa,
értékadás (=), egyenlőség tesztelése (==), változó és
paraméter deklarációk, változók kezdeti értékadása,
megjegyzések, utasítás blokkok { } között);
- nem-determinisztikus vezérlési szerkezetek;

- nem támogatja a számításokat;
nincsenek pl.:
 - karakter típus (karaktert byte típusú változóhoz lehet rendelni, nyomtatása %c specifikációval);
 - lebegőpontos változók;
 - pointererek;
 - függvényhívások;
 - a kifejezések kiértékelésének mellékhatásai;

Alap objektum típusok:

- processzusok (folyamatok, mindig globális);
- adat objektumok (pl. változók, lehet globális és lokális);
- üzenet csatornák ;

Processzusok

- létrehozás processzus típusok példányosításával;
- proctype: processzus típus kulcsszava, pl.

```
proctype producer()
```

- active [*darabszám*] proctype: processzus deklarálás és példányosítás, pl.

```
active [2] proctype producer ()
```

- run *név*: dinamikusan processzus példányosítás;
- init { }: kezdeti processzus, ha nincs active kulcsszóval deklarált processzus, pl.

```
init { print ("hello \n") }
```

- processzus törzse adat deklarációkból és egy vagy több utasításból áll;

```
pl.      proctype main() {printf("Hello world\n")}
        init { run main() }
```

- processzus azonosító (pid): 0-254 közötti szám, kiosztás a létrehozás szerint növekvő sorrendben (első létrehozott processzusé 0);
- létrehozott processzus azonnal megkezdheti, de nem feltétlenül kezdi meg a futását;
- processzus terminál az utolsó utasítást követően;
- processzusok keletkezésükkel ellentétes sorrendben halhatnak meg (pid szám felszabadul);
- proctype deklarációban provided kulcsszó szerepel, a processzus csak akkor futhat, ha a provided után szereplő feltétel igaz;

run operátor

- run *név*() utasítás kifejezés, vagyis nem végrehajtódik, hanem kiértékelődik;
- kiértékelés mellékhatása: értéke a létrehozott folyamat pid-je, illetve hibát jelez, ha több folyamat már nem hozható létre (már 255 van);
- run kifejezések csak egyetlen run operátort tartalmazhatnak és más kifejezésekkel nem kombinálhatók;
pl. run P() & run Q(), !run R() hibásak;

Adat objektumok

alap adattípusok függenek az operációs rendszertől és a C fordítótól;

Típus	tipikus értelmezési tartomány
bit vagy bool	0, 1
byte	0...255
short	$-2^{15} \dots 2^{15} - 1$
int	$-2^{31} \dots 2^{31} - 1$
unsigned :n	$0 \dots 2^n - 1$
chan	1...255
mtype	1...255
pid	0...254

$1 < n \leq 32$

5.1. táblázat. A SPIN alap adattípusai

bit és bool szinonímák (0, 1 vagy false, true);

chan, mtype, pid: bájt értékekkel reprezentált típusok;

chan: csatornák megkülönböztetésére használható;
csatorna mindig globális objektum, neve lehet lokális
vagy globális változó;

```
pl.: chan qcsat = [8] of (byte, int); /*puffer mérete 8*/  
chan pcsat = [0] of (byte, int); /*randevú csatorna*/
```

mtype: szimbólumok használatát teszi lehetővé;
szimbolikus név kiíratás: %e-vel vagy printm
függvénnyel;

```
pl.: mtype = {red, yellow, green};  
mtype light = green;
```

szimbolikus nevek alkalmazhatók szám helyett:

```
pl. # define N 10;
```


tömbök

pl. `short b[4] = 6; /*4 elemű tömb, elemek értéke 6 kezdetben*/`

saját adatstruktúra definiálás `typedef` kulcsszóval;

```
pl. typedef Record { byte a = 3;  
                    short b };
```

```
Record r;
```

```
byte t[10] = 5;
```

```
init{ r.b = t[2] + r.a; printf("r.b=%d",r.b) }; /* r.b = 8*/
```

több dimenziós tömb

```
pl. typedef tomb {short x[3]}; /*egydimenziós tömb típus*/  
tomb y[5]; /*5 db tomb típusú változóból álló tömb*/  
init { y[2].x[3] = 4};
```

Operátorok (a C nyelvből öröklődnek)

- (), [] zárójelek, tömb zárójelpár (legmagasabb precedencia)
- !, ~, ++, -- tagadás, bitenkénti komplementum, 1-el növelés, ill. 1-el csökkentés (pl. a++)
- *, /, % szorzás, osztás, maradék képzés
- +, - összeadás, kivonás
- <<, >> bitenkénti balra, jobbra léptetés
- <, <=, >, >= relációk
- ==, != egyenlő, nem egyenlő reláció
- &, ^, | bitenkénti és, kizáró vagy, megengedő vagy
- &&, || logikai és művelet és a logikai vagy művelet
- (-> :) feltételes kifejezés: (*kif*-> *kif_igaz*: *kif_hamis*)
- = értékadás (legalacsonyabb precedencia)

(operátorok precedenciája a legmagasabbtól csökkenve)

Kezdeti értékadás:

- ha nincs kezdeti értékadás, a változó értéke 0;
pl. `unsigned a : 3 = 5;` /*3 biten tárol, kezdetben értéke 5*/
`short b[3] = 5;` /*a b tömb elemeinek értéke 5*/
- azonos típusok deklarációja összevonható;
pl. `byte a, b[2] = 1, c = 3;`
- kifejezéseket `int` típusú egész számokra értékeli ki;
- értékadáskor és üzenetátadáskor a célváltozó típusára konvertálás (csonkolás) megy végbe, a kifejezés teljes kiértékelése után;
- szimulátor figyelmeztet a lehetséges információvesztésre;

Előre definiált változók

- `else` : értéke igaz, ha a processzusnak nincs végrehajtható utasítása;
- `timeout` : értéke igaz, ha a modellnek (rendszernek) nincs végrehajtható utasítása;
- `_` : csak írható változó, nem tárolja az értéket, üzenet fogadásnál lehet hasznos;
- `_pid` : a processzus azonosítóját adja vissza;
- `_nr_pr` : az aktív processzusok száma;
- `np_` : igaz, ha a modell nincs *progress* (előrehaladási) állapotban;
- `_last` : az utoljára végrehajtott processzus azonosítója;
- `STIND` : előre definiált csatorna (csak szimulációs módban);

példa:

```
active [2] proctype szia()
```

```
{ printf("Az azonosítóm: %d\n", __pid)}
```

másként

```
proctype you_run(byte x)
```

```
{ printf("x = %d, pid = %d\n", x, __pid)}
```

```
init { pid p0, p1;
```

```
p0 = run you_run(0);
```

```
p1 = run you_run(1);
```

```
printf("pids: %d and %d\n", p0, p1)}
```

Üzenet csatornák

- deklaráció:

`chan csat = [méret] of {kompl, ..., kompi};`

- aszinkron: méret $\neq 0$

- szinkron: méret = 0

- üzenetküldés:

`csat! kif1, ..., kifn;`

- kifejezések értékei a csatorna definíciójában megadott típusokra konvertálódnak;

- végrehajtható, ha van hely a pufferben, egyébként blokkol;

- **üzenet fogadás:**

csat? var1, ..., vari;

- végrehajtható, ha nem üres a *csat* nevű csatorna;
- legrégebbi üzenet kerül a változókba;
- ha konstans szerepel változó helyett, akkor csak az adott helyen megadott konstans értéket tartalmazó üzenet olvasható, egyébként blokkol az utasítás;
- *eval (var)* kifejezést kell írni, ha a *var* változó értékét használjuk konstansként;

- **rövidítés:**

csat! kif1(kif2, ..., kifn);

csat? var1(var2, ..., vari);

(általában *kif1* és *var1* az üzenet típusát jelölő konstans)

- szinkron csatorna esetén a küldés és fogadás egyszerre megy végbe, az üzenet nem tárolódik;
- küldő / fogadó blokkol, amíg egy az üzenetet fogadni/ küldeni képes párt nem talál;

példa:

```
mtype {ker, kuld};
```

```
chan csat = [8] of {mtype, int};
```

```
int valasz;
```

```
active proctype P() {
```

```
csat ! ker(0);
```

```
csat ? kuld(valasz); }
```


- egyéb csatorna műveletek:

- $\text{len}(q)$ értéke a q -ban levő üzenetek száma
- $\text{empty}(q)$ értéke igaz, ha q üres
- $\text{nempty}(q)$ értéke igaz, ha q nem üres
- $\text{full}(q)$ értéke igaz, ha q tele van
- $\text{nfull}(q)$ értéke igaz, ha van szabad hely q -ban

- rendezett küldés:

$q!! n, m, p$

ha q -ban az üzenetek rendezettek elsőként n , majd m és végül p értéke (üzenet komponensek sorrendje) szerint, akkor az új üzenet a legelső nála nagyobb értékeket tartalmazó üzenet elé kerül;

példa: (fel_rend_ch1.pml)

```
init {chan q = [3] of {int};
```

```
int x;
```

```
q!! 5;
```

```
q!! 2;
```

```
q? x -> printf ("%d\n", x);
```

```
q? x -> printf ("%d\n", x);} 
```

(elsőként a 2 majd az 5 érték kerül kiírásra)

- közvetlen fogadás:

csat?? n, m, p

- a *csat* csatorna bármelyik üzenete kiolvasható vele (nem feltétlenül az első), ha a konstansok által meghatározott feltételeket teljesíti;
- ha *n*, *m* és *p* között nincsenek konstansok, akkor alkalmazása felesleges (ugyanaz mint a normál fogadó utasításnak);
- mindig a legelső, a feltételeknek eleget tevő üzenetet olvassa ki, ami nem biztos, hogy a csatorna fejénél van;
- a normál, a rendezett küldés, és a közvetlen fogadás utasítások egymással szabadon kombinálhatók;

- egyéb csatorna utasítások

csat ? [*n*, *m*, *p*]

- mellékhatás mentes Boole-kifejezés;
- értéke igaz, ha *csat* ? *n*, *m*, *p* végrehajtható;
- kiértékelése nem változtatja meg *csat* csatornát;
- a változók nem veszik fel az üzenet értékét;

csat ? < *n*, *m*, *p* >

- ugyanaz, mint *csat* ? *n*, *m*, *p*, vagyis a változók felveszik az üzenet értékét, ha az utasítás végrehajtható, de
- az üzenet nem törlődik a csatornából, *csat* változatlan marad (észrevétlenül megnézhető a legrégebbi üzenetet a *csat* csatornában);

Alap utasítástípusok

- értékadás pl. `x++`, `x--`, `x = x+1`, `x = run P()` (`b = a++` hibás)
- nyomtatás pl. `printf ("x=%d\n", x)`, `printm (x)`
- kifejezések pl. `x`, `1`, `run P()`, `else`, `timeout`, `true`, `skip`
akkor hajthatók végre, ha az értékük igaz (nem 0);
- `assert` `assert (kifejezés)`
mindig végrehajtható, a kifejezés szimulációkor kiértékelődik, ha az érték igaz, akkor az adott processzusban az ezt követő utasítás lesz a következő végrehajtható, egyébként hibaüzenettel fejeződik be;
pl. `init {assert(false)}` a program hibaüzenettel befejeződik

- üzenetküldés végrehajtható, ha a célcsatorna nincs tele;
- üzenetfogadás végrehajtható, ha nem üres a csatorna és az első üzenet a konstansoknak megfelelő értéket tartalmaz;

Összetett utasítások

Ezekkel lehet szabályozni a végrehajtás sorrendjét, normál esetben az utasítások egymás után hajtódnak végre kivéve a goto-t de az igazából nem önálló utasítás, csak a következő vezérlési pontot jelzi, mint a break.

nem-determinisztikus választás

```
if
    :: őrfeltétel1 -> ut11; ut12;...
    :: őrfeltétel2 -> ut21; ut22;...
    ...
    :: őrfeltételn -> utn1; utn2;...
fi
```

- végrehajtható, ha van igaz értékű őrfeltétel;
- ha több igaz értékű őrfeltétel is van, akkor nem-determinisztikusan választ;
- ha egy igaz értékű őrfeltétel sincs, az utasítás blokkol;
- őrfeltétel bármilyen alap- vagy összetett utasítás lehet;
- else őrfeltétel akkor igaz, ha a többi őrfeltétel értéke hamis;

- őrfeltételt követő utasítás sorozat lehet üres is, ekkor az őrfeltétel igaz értéke esetén kilép az if utasításból;
- skip, true és 1 értéke mindig igaz ;

példa: (fel_max.pml)

```
active proctype P() {  
int a = 5, b = 5, max, branch;  
if  
:: a >= b -> max = a; branch = 1  
:: b >= a -> max = b; branch = 2  
fi;  
printf("szamok: %d és %d max: %d ag: %d\n", a, b, max, branch)}
```


ciklikus nem-determinisztikus választás

do

::őrfeltétel₁ -> ut₁₁; ut₁₂;...

::őrfeltétel₂ -> ut₂₁; ut₂₂;...

...

::őrfeltétel_n -> ut_{n1}; ut_{n2};...

od

- végtelen sokszor ismétlődik, egy menet olyan, mint az if .. fi utasítás ;
- csak break vagy goto utasítással lehet kilépni belőle;
- a break utasítás a ciklus utáni utasításra ugrik;
- őrfeltétel bármilyen alap- vagy összetett utasítás lehet;

példa: x és y legnagyobb közös osztója (fel_Ink.pml)

```
active proctype P() {  
    int x = 15, y = 20;  
    int a = x, b = y;  
    do  
        :: a > b -> a = a - b  
        :: b > a -> b = b - a  
        :: a == b -> break  
    od;  
    printf("szamok: %d és %d Ink: %d\n", x, y, a)}  
}
```

Nincs ciklus számlálóval.

példa: első N szám összege (fel_szamokosszege.pml)

```
# define N 10
active proctype P() {
    int sum = 0; byte i = 1;
    do
        :: i > N -> break
        :: else -> sum = sum + i; i++
    od;
    printf("első %d szám összege: %d\n", N, sum)}
```

példa: szerver és kliens közötti kommunikáció (Fel_13_ch2.pml)

```
chan request = [4] of {byte};
chan reply = [4] of {byte, byte};
active [2] proctype Server()
{
  byte client;
  do
  :: request ? client ->
    printf ("Client %d processed by server
           %d\n", client, __pid);
    reply ! __pid, client
  od
}
```

```
active [4] proctype Client()
{
  byte server;
  request ! __pid;
  reply ?? server, eval(__pid);
  printf("Reply received from
        server %d  by client %d\n",
        server, __pid)
}
```

megszakíthatatlan utasítás (atomic)

`atomic { őrfeltétel -> ut1; ut2; ...; utn }`

- akkor hajtható végre, ha az őrfeltétel igaz;
- amíg az összes utasítás végre nem hajtódik más folyamat nem léphet;
- bármely utasítás lehet őrfeltétel;
- lehetnek benne nem-determinisztikus utasítások;
- ha valamelyik belső utasítása blokkol, akkor a vezérlést megkaphatja más folyamat, ha közben a blokkolás megszűnik, akkor (nem feltétlenül azonnal) megszakítás nélkül folytatódik a blokk;

példa:

- változók értékének cseréje:

```
atomic { tmp = b, b = a, a = tmp };
```

- két folyamat elindítása egyszerre:

```
init { atomic { run A(1,2); run B(3,4) } };
```

- felesleges használat:

```
nfull(qname) -> qname!msg0 vagy qname?[msg0] -> qname?msg0 ;
```

utasítások nem biztos, hogy nem blokkolnak, mert más folyamat telítheti közben a csatornát, vagy ellophatja az üzenetet;

pl. `atomic { qname?[msg0] -> qname?msg0 }` felesleges, mert `qname?msg0` hatása ugyanaz;

példa: (Provided.pml)

```
byte n = 0;
```

```
bool interrupt = false;
```

```
proctype Compute() provided (!interrupt) {n = n + 1}
```

```
proctype Interrupt() {
```

```
    byte temp;
```

```
    interrupt = true;
```

```
    temp = n + 1;
```

```
    n = temp;
```

```
    interrupt = false}
```

```
init { atomic {run Interrupt(); run Compute()}
```

```
    (_nr_pr == 1);
```

```
    assert (n == 2)}
```

determinisztikus lépések

`d_step { őrfeltétel-> ut1; ut2; ...; utn }`

- ugyanaz, mint az `atomic {...}`, de determinisztikusnak kell lennie;
- biztosan megszakítás nélkül hajtódik végre;
- nem lehetnek az őrfeltételen kívül blokkoló utasításai;
- tilos bele vagy belőle `goto`-val be- vagy kiugrani;
- lehetnek benne nem-determinisztikus utasítások, de ha nem-determinisztikusság áll elő, akkor ezt fix módon oldja fel a rendszer, pl. mindig a legelső lehetőséget választja (a választás módja nem ismert)

menekülő utasítás sorozat (escape sequences)

$\{ P \}$ unless $\{ Q \}$, ahol P és Q utasítás sorozatok

- kivételkezelést lehet vele megvalósítani;
- a rendszer elkezdi P utasításait végrehajtani,
- minden P -beli utasítás végrehajtása előtt ellenőrzi a Q őrfeltételt (annak az első utasítását), hogy igaz-e
 - ha igaz, akkor P további része helyett Q -t hajtja végre
 - ha Q őrfeltétel P egész futása alatt hamis, akkor Q egyáltalán nem hajtódik végre.

példa: 0-val való osztás elkerülése az unless alkalmazásával

```
active proctype divide() {
```

```
int n = 1;
```

```
end: do
```

```
  :: { ch? n;
```

```
    printf("%d\n", 100/ n)}
```

```
    unless {n == 0 -> printf("attempt to divide by zero\n")}
```

```
  od
```

```
}
```

ugró utasítás

goto label

- hatására a következő végrehajtható utasítás a *label* címkéjű utasítása a folyamatnak ;

Címkék

- bármely utasítás elé írható (egy vagy több) címke;
- a folyamattípuson belül egyedieknek kell lenniük;
- az utasításokból a folyamat automatájának átmenetei lesznek, így az utasítás elé írt címke, azt az állapotot jelöli, ahonnan az átmenet kiindul;
- ciklusnál az elágazás előtti állapotot lehet megjelölni, vagyis a `do` utasítás elé lehet tenni a címkét (őrfeltétel elé nem);

Helyességi kritériumok megadása

állapot tulajdonság:

- alapfeltételezések (basic assertions)
- végállapot-címkék (end-state labels)

út tulajdonság:

- előrehaladási címkék (progress-state labels)
- elfogadási címkék (accept-state labels)
- soha-állítások (never claims) (ezek automatikusan generálhatók LTL formulákból)
- út-feltételezések (trace assertions)

- **helyességi ellenőrzés** címke típusok:
 - end... kezdetű *végállapot-címkék*
 - progress... kezdetű *előrehaladási címkék*
 - accept... kezdetű *elfogadási címkék*

végállapot-címkék

end[a-zA-Z0-9_]*: utasítás

- a *rendszer végállapothoz ér*, ha már nincs végrehajtható utasítása egyik folyamatnak sem (mindegyik véget ért vagy blokkolt);
- alapértelmezés szerint a rendszernek csak azok a *végállapotai megengedettek*, melyben minden folyamat befejeződött (elérte a záró kapcsos zárójelet), különben *holtpont* (deadlock) állt elő;

- a végállapot címkével megadhatjuk, hogy mely állapotban való várakozás megengedett, azaz nem okoz holtpontot;

példa: Dijkstra algoritmus (fel_dijkstra.pml)

```
mtype { p, v };
chan sema = [0] of { mtype };
active [3] proctype user()
{sema?p; /* enter */
 critical: skip; /* leave */
 sema!v}
active proctype Dijkstra()
{ byte count = 1;
 do
 :: (count == 1) -> end: sema!p; count = 0
 :: (count == 0) -> sema?v; count = 1
 od }
```

előrehaladási címkék

`progress[a-zA-Z0-9_]*`: utasítás

- azt jelzi, hogy az általa jelölt állapotban előrehaladás történt (pl. belépett a kritikus állapotba);
- azt nézi, hogy van-e olyan ciklusa a rendszernek, melyben sohasem történik előrehaladás, azaz az egész futás alatt csak véges sokszor (a ciklusba lépés előtt);
- non-progress módban ciklusok keresése, mikor egy nem kívánatos esemény rendelkezik progress címkével;

példa: Dijkstra algoritmus progress címkével

```
mtype { p, v };
chan sema = [0] of { mtype };
active proctype user_1()
{do
:: sema?p; /* enter */
  progress_1: skip; /* leave */
  sema!v
od}
active proctype user_2()
{do
:: sema?p; /* enter */
  critical: skip; /* leave */
  sema!v
od}
```

```
active proctype Dijkstra()
{ byte count = 1;
  do
    :: (count == 1) -> sema!p;
  count = 0
  :: (count == 0) -> sema?v;
  count = 1
  od }
```


elfogadási címkék

`accept[a-zA-Z0-9_]*`: utasítás

- az előrehaladási címkék „szimmetrikus párja”;
- vizsgálja, hogy van-e olyan ciklusa a rendszernek, melyben végtelen sokszor halad át elfogadási címkével jelölt állapoton (vagy végtelen sokáig ilyen állapotban tartózkodik);
- ilyen „rossz” vagy csak számunkra érdekes ciklusok létezése kérdezhető;

példa: (fel_accept.pml)

```
byte x = 2;  
active proctype A()  
{do  
  :: x = 3 - x  
  od}  
active proctype B()  
{do  
  :: x = 3 - x;  
  accept: skip  
  od}
```

(verifikálás acceptance módban)

Beillesztett definíció

példa:

```
inline request (x, y, z) {atomic { x == y -> x = z; who = _pid }} ...  
request(turn, P, N) ;...
```

- minden hívásnál a beillesztett definíció törzse a paraméterek kért értékeivel a hívás helyére helyettesítődik még a fordítás megkezdése előtt;

pl. a `request(turn, P, N)` helyére az alábbi kerül

```
atomic { turn == P -> turn = N; who = _pid }
```

Fair ciklusok

Gyenge fair tulajdonság:

Ha egy folyamat egy olyan utasításánál áll, amely végtelen sokáig (megszakítás nélkül) végrehajtható, akkor azt az utasítást a folyamat előbb utóbb végre is tudja hajtani.

Erős fair tulajdonság:

Ha egy folyamat egy olyan utasításánál áll, amely végtelen sokszor (esetleg megszakításokkal) végrehajtható, akkor azt az utasítást a folyamat előbb utóbb végre is tudja hajtani.

A SPIN közvetlenül csak a gyenge fair tulajdonságot támogatja, az erős fair tulajdonság soha-állításokkal valósítható meg.

Soha-állítások

- soha-állítások speciális folyamatok, csak kifejezéseket és vezérlő utasításokat tartalmaznak;
- automatákat definiálnak, mint a többi folyamat, de nem változtathatják meg a rendszer állapotát, hanem lépésenként megfigyelik (ellenőrzik) azt;
- a rendszer minden lépése előtt a soha állítások egy-egy lépése hajtódik végre (felváltva léphetnek, de ha a rendszer működése véget ért a soha állítás még lépegethet;
- minden modellben csak egy soha-állítás szerepelhet;

- az ellenőrzés azt jelenti, hogy keresi a rendszernek olyan végrehajtási sorozatait, melyek a soha állításban megfogalmazott tulajdonságot teljesítik;
- ezekre a SPIN hibát jelez;
- soha-állítással pontosan az általa lehetetlennek definiált működés kereshető, vagyis hiba, ha a soha-állítás véget ér (azaz eléri a záró kapcsos zárójelet) vagy elfogadó állapotot is tartalmazó végtelen ciklusba jut;
- soha-állítás nem blokkol, ha nem tud lépni, akkor nincs hibás működés, a rendszer ezen a végrehajtási ágon nem keres tovább hibát, hanem másik ágot próbál;

példa: soha-állítások

(biztonságos és élősegi tulajdonságokra)

```
never { /* !([] mutex) */
T0_init:
  if
    :: (! ((mutex))) -> goto accept_all
    :: (1) -> goto T0_init
  fi;
accept_all: skip
}
```

```
never { /* !((<>csp) */
accept_init:
T0_init:
  if
    :: (! ((csp))) -> goto T0_init
  fi;
}
```

LTl formulák

- formula lehet egy kifejezést jelölő szimbolikus név (kisbetűvel kezdődő);

példa:

- `#define p (a > b)` : p értéke igaz, ha a zárójelben levő kifejezés értéke igaz;
- `#define q (root@label)` : értéke igaz, ha a root folyamat következő végrehajtható utasítása a label címkéjű utasítás;
- `#define p (len(q) < 5)` : értéke igaz, ha a q csatornában 5-nél kevesebb üzenet van;

Operátorok:

- *egyváltozós operátorok:*

[] : always (a G jelölés helyett)

<> : eventually (a F jelölés helyett)

! : not (negáció)

X : next

- *kétváltozós operátorok:*

U : until

V : release (az until duálisa vagyis $p \vee q \equiv !(!p \wedge !q)$)

&& : and (és)

|| : or (vagy)

-> : implikáció

<-> : ekvivalencia

(operátorok jele helyett az angol név is használható)

példa: tulajdonság formális megadása

k: közelít **v**: áthalad, **s**: sorompó leengedve

- Ha egy vonat áthaladt a sorompót fel fogják emelni:

$$[] ((v \text{ U } !v) \rightarrow (\langle \rangle (!v \ \&\& \ !s)))$$

- A sorompó sosem maradhat örökké lezárva:

$$! (\langle \rangle [] s) = [] \langle \rangle !s$$

- Minden vonat folyamatosan közelít az áthaladás kezdetéig:

$$[] ((k \rightarrow (k \text{ U } v))$$

- Minden közelítő vonat át is halad a kereszteződésen, még mielőtt a következő vonat közelítene:

$$[] (k \rightarrow [(k \ \&\& \ !v) \text{ U } (v \ \&\& \ ((v \ \&\& \ !k) \text{ U } !v))])$$

A modell-ellenőrzés gyakorlata

UPPAAL

Uppsalai Egyetem + Aalborgi Egyetem közös fejlesztése;
1995. első verzió megjelenése;

részei:

- grafikus modellt leíró eszköz (*System editor*)
- szimulátor (*Simulator*)
- modell-ellenőrző (*Verifier*)
- ingyenesen letölthető: <http://www.uppaal.org/>
- [Uppaal rövid bemutatása](#)

Modell megadása

kiterjesztett időzített automata hálózat ;

- *kiterjesztett* szó arra utal, hogy korlátos egész értékű változók is használhatók;
- *időzített* arra utal, hogy az automatákban valós idejű órák alkalmazhatók (újraindíthatók, tesztelhetők);
- több *automata* alkothat *hálózatot*, melyek csatornákon keresztül szinkronizálnak (üzenet! címkéjű (küldő) átmenet csak egy üzenet? (fogadó) átmenettel egyszerre hajtható végre);

Modell elemei:

- **Globális és lokális deklarációk** (C nyelvhez hasonlóak)
 - konstansok pl. `const int a=1;`
 - változók pl. `int a;` értékkészlet [-32768, 32767]
 - korlátozott egész értékű változók pl. `int [0, 100] b=5;`
 - csatornák pl. `chan d;` `d?` és `d!` használható szinkronizációhoz
 - órák pl. `clock x, y;`
 - tömbök pl. `bool b[8], c[4];`
 - rekordok pl. `struct {int a; bool b} s1={2, true};`
 - felhasználói típusok
pl. `typedef struct { int a; bool b; clock c} S;`
 - skalárok pl. `typedef scalar[3] set; set s;`
`int a[set]` (a tömb, set típus elemeivel indexelhető)

- **Sablonok** (*template*):
 - processzus típusok megadása;
 - minden sablonnak van neve;
 - rendelkezhet lokális deklarációkkal;
 - paraméterezhetők;
 - paraméterátadás történhet:
 - a konkrét paraméter megadásával (ekkor érték szerint (*call by value*), vagy
 - a formális paraméter azonosítója előtti & jel használata esetén hivatkozás szerinti (*call by reference*);
 - óra, csatorna, tömb csak hivatkozás szerinti paraméter lehet;

- időzített automata (processzus típus)

megadásához grafikus felület:

hely paraméterek:

- kezdő (*initial*), sürgős (*urgent*), elkötelezett (*committed*),

- hely-invariáns (invariant);

átmenet paraméterek:

- szelekció (select),

- őrfeltétel (guard),

- szinkronizáció (sync.),

- értékadás (update) megadás;

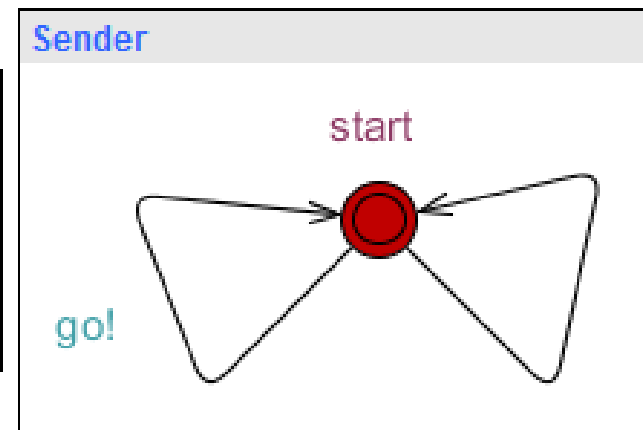
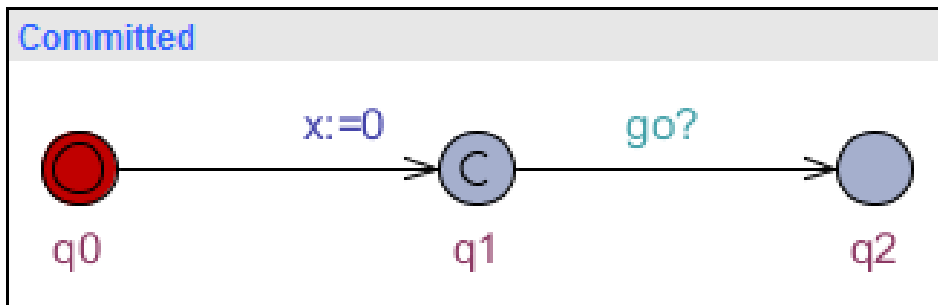
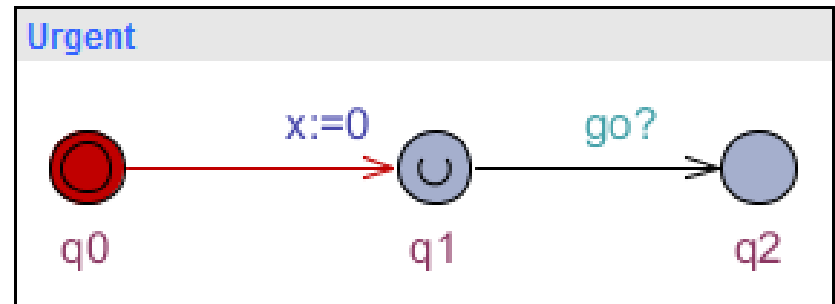
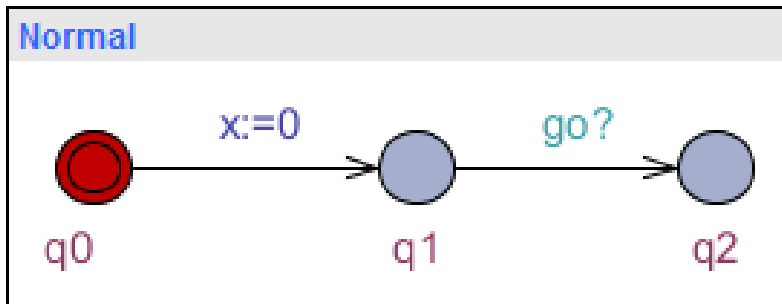
invariánsok (*invariant*):

- helyek rendelkezhetnek invariánssal;
- mellékhatás mentes, Boole értékű kifejezés;
- csak órákra, órák különbségére vonatkozó egyszerű kifejezések (csak felső korlát, ami egész értékű kifejezés) konjunkciói, illetve órákat nem tartalmazó Boole kifejezések lehetnek;
- addig tartózkodhat az automata az adott helyen míg az invariáns feltétel teljesül;
- invariáns feltétel nem válhat hamissá, vagyis ha hamissá válik, akkor nincs következő rendszer állapot és a futás véget ér;

sürgősség és elkötelezettség:

helyek tulajdonsága lehet:

- kezdő (*initial*)
 - sürgős (*urgent*): ilyen helyen nem telhet az idő, a többi automata várakozástól különböző átmeneteket csinálhat míg a sürgős helyet az automata el nem hagyja;
 - elkötelezett (*committed*): ilyen helyen nem telhet az idő, a következő átmenet innen vagy egy másik elkötelezett állapotból indulhat;
- sürgős és elkötelezett egyszerre nem lehet egy hely;



példa: normál, kezdő, sürgős, elkötelezett helyek az Uppaal-ban

őrfeltételek (*guard*):

- átmenetek engedélyezése, ha az őrfeltétel értéke igaz a rendszer aktuális állapotában;
- mellékhatás mentes, Boole értékre kiértékelhető;
- óra változókra, egészekre és konstansokra (valamint azok tömbjeinek elemeire) lehet hivatkozni bennük;
- óraváltozók, illetve azok különbsége csak egészekkel hasonlíthatók;
- különböző órákra vonatkozó feltételek csak konjunkcióval köthetők össze, diszjunkcióval nem;

példa:

$x \geq 1 \ \&\& \ x \leq 2$: x az $[1,2]$ intervallumba esik ;

$x < 100$ or $y == 50$ nem megengedett;

szinkronizációk (*sync.*):

- a szinkronizáció történhet bináris, üzenetszórásos, sürgős csatornán keresztül;
- a szinkronizáció alapját képező csatornákat globálisan kell deklarálni;

pl. *chan c*;

broadcast chan d;

urgent chan a;

- *bináris szinkronizáció*:

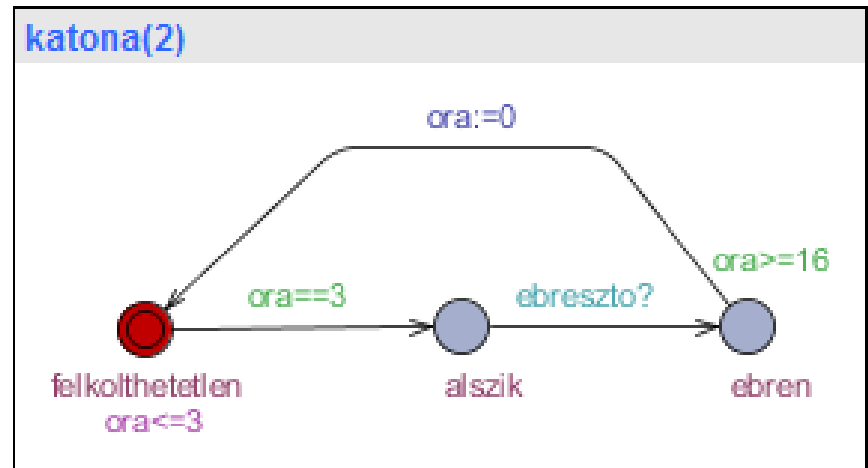
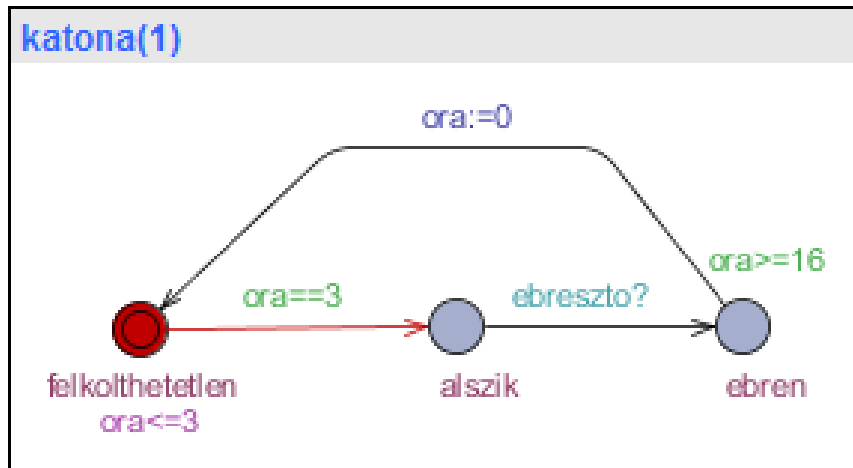
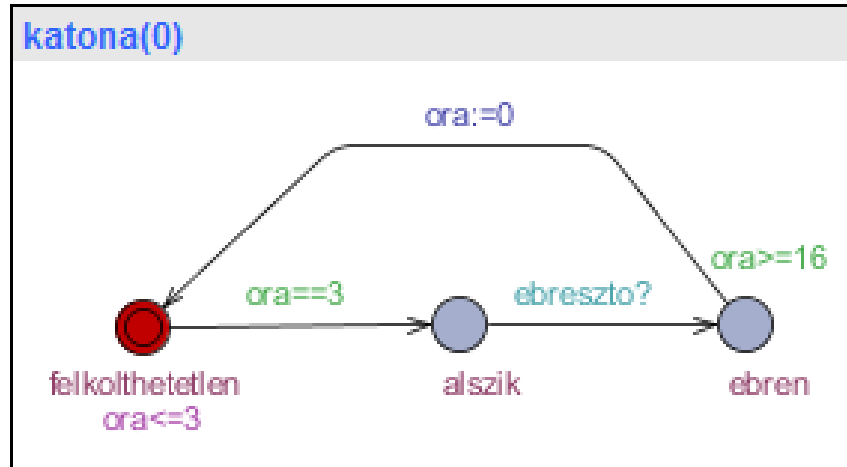
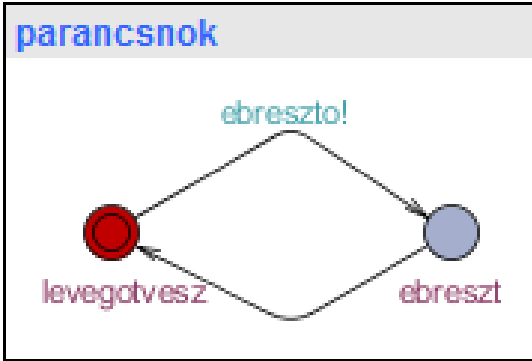
az egyik automata pl. *c!* és egy automata *c?* csatorna parancsa egyszerre hajtódik végre, tehát küldés csak akkor hajtható végre, ha van fogadó, különben az átmenet blokkol;

- *üzenetszórásos (broadcast)* csatornák:

egy küldő és tetszőleges számú (0 is lehet) fogadó fél között jön létre, a küldés mindig végrehajtható;
minden végrehajtható fogadóval (d?) szinkronizál a broadcast típusú csatornára küldő (d!);

- *sürgős (urgent)* csatornák:

ha a sürgős csatornán keresztül szinkronizálni lehet, sem a küldő sem a fogadó nem tehet várakozó átmenetet,
egymást meg kell várniuk,
órákra vonatkozó őrfeltétel nem lehet a szinkronizáló átmeneten;



példa:üzenetszórásos szinkronizáció az Uppaal-ban
 (declaration mezőben: broadcast chan ebreszto;)

értékadások (*update*):

- az átmenet végrehajtásának 3. lépése, megváltoztatja a modell állapotát;
- típusokat egyeztetni kell, csak konstansra, órára, egész változóra (ezek tömbjeire) lehet hivatkozni;
- nem csak az órák újra indítása, hanem az óraváltozóknak tetszőleges egész érték is adható;
- más változónak is lehet értéket adni;
- értékadások végrehajtása szekvenciálisan;

példa:

$j = (i[1] > i[2] ? i[1] : i[2])$: j értéke az $i[1]$ és $i[2]$ maximuma lesz;

$x = 1, y = 2 * x$: x értéke 1, y értéke 2 lesz;

szelekció (*select*):

- nem-determinisztikusan köthető érték azonosítóhoz adott értéktartományból;
- az átmenet őrfeltételében, szinkronizációjában, értékadó részében a szelekcióban megadott változó a hozzákötött értéktartományból vesz fel értéket;

példa:

select: $i : \text{int } [0,3]$

synchronization: $a[i]?$

update expression: $\text{receive_}a(i)$

(i -hez nem-determinisztikusan történik a $[0-3]$ -ba tartozó valamely egész érték kötése, mely indexként jelenik meg az a csatorna tömbnél, a $\text{receive_}a$ eljárás hívásnál pedig argumentumként);

Kifejezések szintaxisa, alkalmazható operátorok, operátorok precedenciája részletesen:

Uppaal Help / language reference / expressions

forall (id : type) Expr kifejezés:

értéke igaz, ha az Expr értéke igaz bármely type típusba eső id érték esetén, egyébként az értéke hamis;

exists (id : type) Expr kifejezés:

értéke igaz, ha van olyan type típusba eső érték, melyet id felvéve az Expr értéke igaz, egyébként hamis;
type típus korlátos egész vagy skalár halmaz lehet;

- **Rendszer definíció** (*system definition*)

- komponensek megadása az automata sablonok *példányosításával*;

példa:

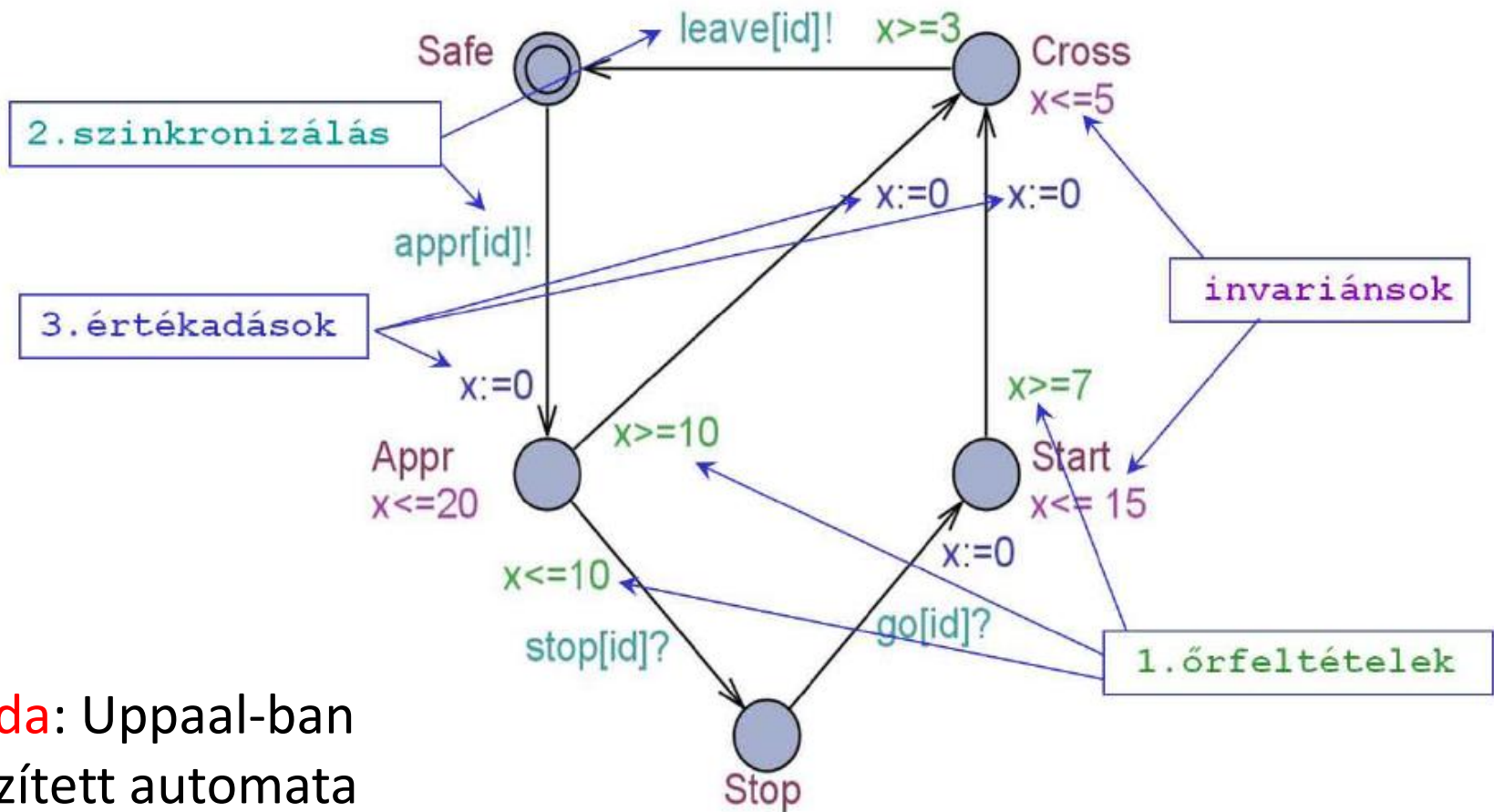
lampa és embertípus egy-egy automata sablon;

ember1 := embertípus (1000);

ember2 := embertípus (2000);

system lampa, ember1, ember2;

- ha egy sablon *korlátozott egész értékű paraméterekkel* rendelkezik, akkor aktuális paraméter megadás hiányában automatikusan létrehozza a példányokat úgy, hogy a paramétereket az összes lehetséges módon az értékkészletből megválasztja;



példa: Uppaal-ban időzített automata sablonra

(name mezőben: Train, parameters mezőben: const id_t id
 declarations mezőben:

```
const int N = 6; typedef int[0,N-1] id_t;
chan  appr[N], stop[N], leave[N]; urgent chan go[N];
```

Tulajdonság specifikáció Uppaal-ban

CTL logika megszorított változata;

- $\mathbf{E}\langle\rangle P$: van olyan futás, hogy valamikor a P igaz lesz (lehetségesség, \mathbf{EFP});
- $\mathbf{A}[]P$: minden futás minden állapotában P igaz (invariáns tulajdonság, \mathbf{AGP});
- $\mathbf{E}[]P$: van olyan futás, melynek minden állapotában P igaz (lehetséges invariáns, \mathbf{EGP});
- $\mathbf{A}\langle\rangle P$: minden futás során valamikor P igaz lesz (előbb-utóbb biztosan P igaz lesz, \mathbf{AFP});
- $P \dashrightarrow Q$: minden úton, ha P teljesül, valamikor Q is teljesülni fog;

- P és Q nem lehetnek összetett formulák, csak Boole értékű, mellékhatás mentes kifejezések;
- nem engedi a temporális operátorok egymásba ágyazását (pl. $A[] (E \leftrightarrow P)$ nem írható);
- lehet kifejezésekben konstansokra, egész értékű változókra, órák értékeire, automaták aktuális állapotaira hivatkozni;
- formulákban használhatók a *not*, *or*, *and*, *imply* Boole műveletek is;

néhány azonosság:

$$\text{not } A[] P = E \leftrightarrow \text{not } P$$

$$\text{not } A \leftrightarrow P = A[] \text{not } P$$

$$P \rightarrow Q = A[] (P \text{ imply } A \leftrightarrow Q)$$

példa:

$A[] \ 1 < 2$: ez az invariáns mindig igaz;

$E \langle \rangle (p1.cs \text{ and } p2.cs)$: igaz, ha a rendszer valamikor elérhet egy olyan állapotot, melyben $p1$ és $p2$ processzus is a cs helyen van;

$A[] (p1.cs \text{ impy not } p2.cs)$: invariáns tulajdonság, valahányszor $p1$ a cs helyen van, mindannyiszor $p2$ nincs a cs helyen (nem keverendő a $-->$ operátorral, mely megengedi az időbeli eltolódást);