

# Modellezés UPPAAL-ban

## Házi feladat minta és megoldása

dr. Bartha Tamás

BME Méréstechnika és Információs Rendszerek Tanszék

### Tartalom

- Az előadás egy „tipikus” félévközi házi feladat megoldásának módját és menetét mutatja be.
- Ezen felül demonstrál néhány hasznos UPPAAL modellezési fogást:
  - véletlen érték generálása és felhasználása
  - atomi műveletek modellezése
  - szinkron kommunikáció modellezése
    - globális osztott változó használatával
    - dedikált csatornatömbök alkalmazásával
  - állapottér csökkentése ideiglenes változók törlésével
  - adatstruktúrák és függvények használata
  - temporális kifejezések írása modellellenőrzéshez

# Bemelegítés

Egy egyszerű feladat megoldásával

3

## Bemelegítő feladat

### A feladat

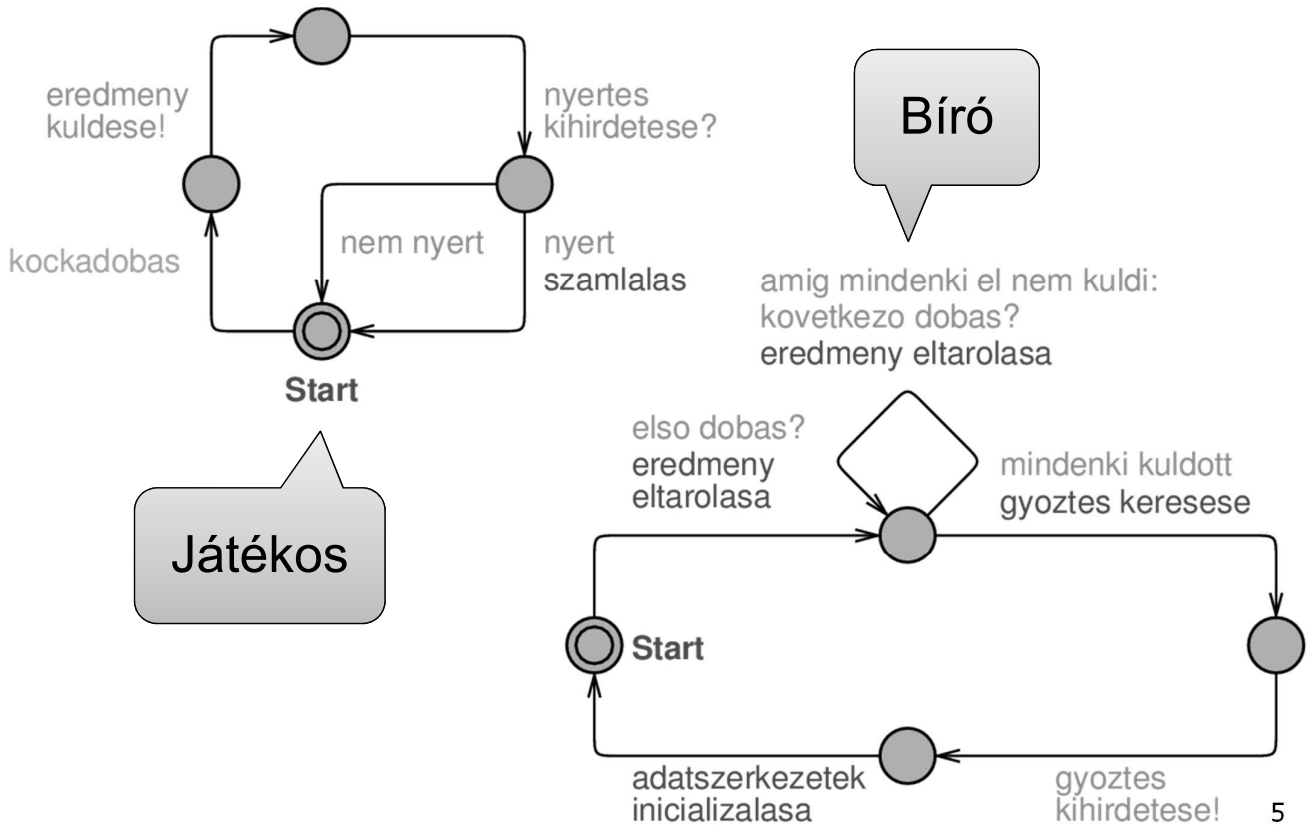
- Kockadobálós játék
  - $n$  játékos, 1 bíró
  - minden játékos egy kockával egyszer dob
  - közlik az eredményt a bíróval
  - a bíró
    - feljegyzi az eredményeket
    - megkeresi a legnagyobbat
    - kihirdeti a nyertest
  - a játékosok számlálják a nyeréseket

### Mit kell megoldanunk?

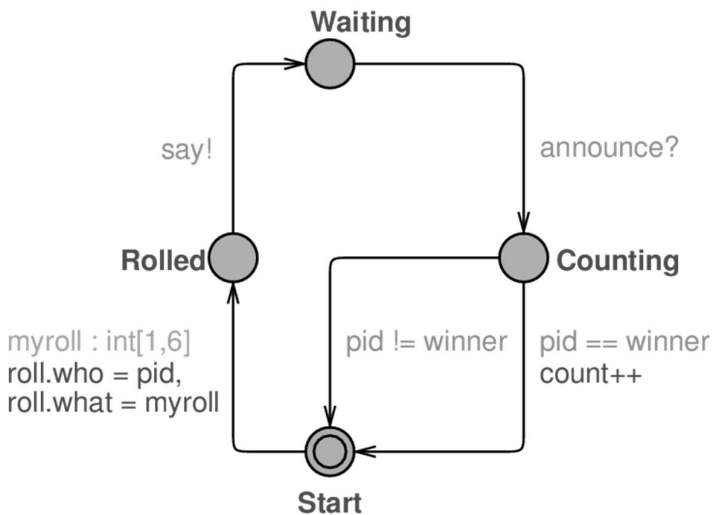
- Véletlen érték generálása
- Kommunikáció
  - értékek „továbbítása”
  - „broadcast” kommunikáció
  - csatornatömbök kezelése
  - Update szekciók sorrendje
- Adatszerkezetek
- Függvények
- Konkurencia és időzítés
- Modell ellenőrzés

4

# Megoldás alapgondolata



# Megoldás: a rendszer és a játékos



Játékos:

Player(id\_t pid)

int[0,wins] count = 0;  
clock x;

Rendszer:

system Player, Referee;

const int players = 3;  
const int wins = 10;

```
typedef int[0,players-1] id_t;
typedef int[0,6] dice_t;
struct {
    id_t who;
    dice_t what;
} roll;
```

id\_t winner;  
chan say;  
broadcast chan announce;

# Megoldás: bíró

Bíró:

```
int [0,players] ans = 0;
dice_t rolls[id_t];
dice_t best = 0;

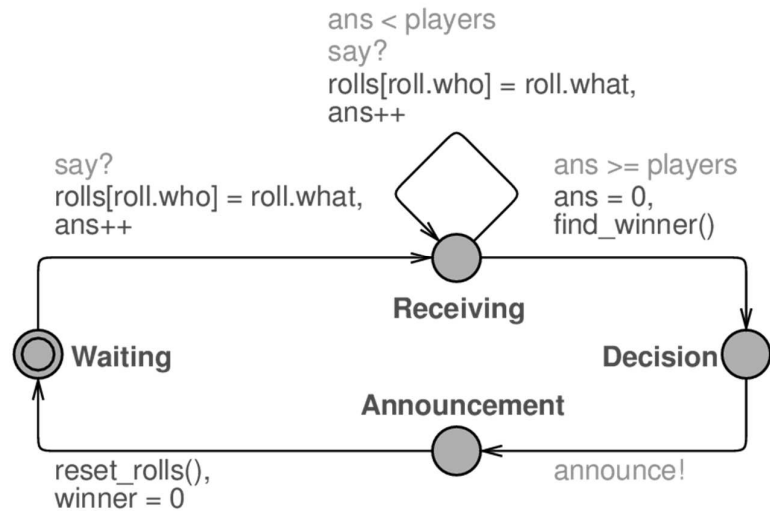
clock x;
```

```
void reset_rolls() {
    int[0,players] i;

    for (i = 0; i < players; i++) rolls[i] = 0;
}
```

```
void find_winner() {
    int[0,players] i;

    for (i = 0; i < players; i++) {
        if (rolls[i] > best) {
            best = rolls[i];
            winner = i;
        }
    }
    best = 0;
}
```



7

## Ellenőrizzük a működést!

Overview

```
A<> exists (i : id_t) (Player(i).count == wins)
A<> Referee.Decision && forall (i : id_t) Referee.rolls[i] > 0
E<> Referee.Decision && forall (i : id_t) Referee.rolls[i] > 0
A[] not deadlock
```

Query

```
A<> exists (i : id_t) (Player(i).count == wins)
```

Comment

Status

```
A[] not deadlock
Established direct connection to local server.
(Academic) UPPAAL version 4.0.13 (rev. 4577), September 2010 -- server.
The verification was aborted due to an error. Most likely, this is caused by an out-of-range assignment or out-of-range array lookup.
E<> Referee.Decision && forall (i : id_t) Referee.rolls[i] > 0
Property is satisfied.
A<> Referee.Decision && forall (i : id_t) Referee.rolls[i] > 0
Property is not satisfied.
A<> exists (i : id_t) (Player(i).count == wins)
Property is not satisfied.
```

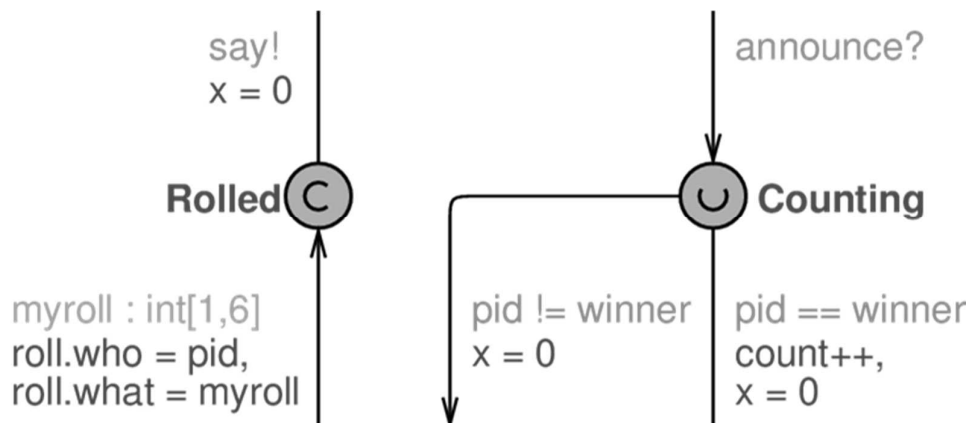
De van olyan útvonal, ahol nem tudunk ilyen állapotba eljutni!

- Miért?
- Két oka: a konkurencia és az időzítés helytelen kezelése!

8



# Nem kívánt konkurencia kiküszöbölése

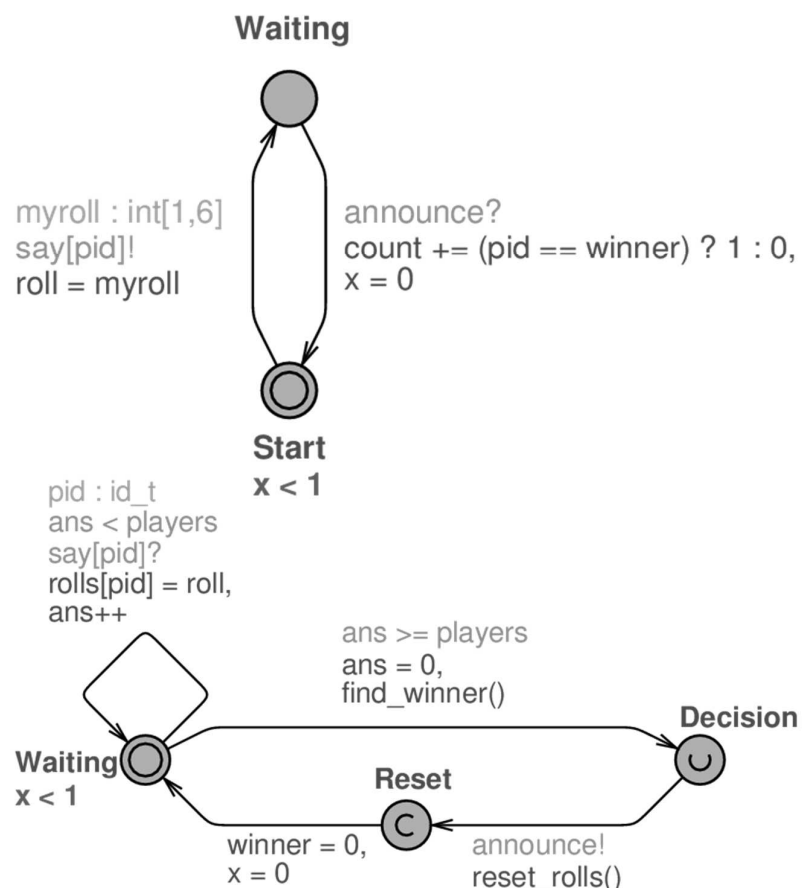


- A probléma az, hogy a **Waiting** és **Rolled** állapotok konkurenssek, a tüzelések nemdeterminisztikusak
- **Megoldás:**
  - konkurencia megszüntetése: „committed” állapot bevezetése
    - a „committed” állapotból azonnal tovább kell lépniünk

11

## További egyszerűsítési lehetőségek

- Csatornatömbök használata
- „?” operátor alkalmazása
- Válaszok gyűjtése egy állapotban
- Iterátorok alkalmazása
- Reset állapot elhagyása

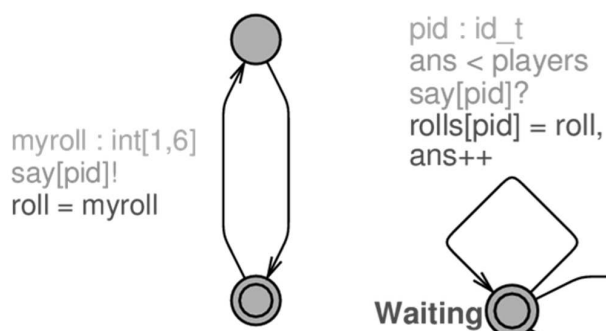


12

# Speciális lehetőségek

- Csatornatömbök használata

- A fogadó folyamat egy Select konstrukcióval „egyszerre” az összes csatornát figyeli
- A csatorna azonosító felhasználható az Update szekcióban!



- Iterátorok alkalmazása

```
void reset_rolls() {  
    for (i : id_t) rolls[i] = 0;  
}
```

```
void find_winner() {  
    for (i : id_t) {  
        if (rolls[i] > best) {  
            best = rolls[i];  
            winner = i;  
        }  
    }  
    best = 0;  
}
```

13

## További modellezési tippek, jó tanácsok

- Az élek esetén a szekciók kiértékelési sorrendje:  
Select » Sync » Guard » Update
  - Szinkronizáló élek esetén a küldő Update-ja a fogadóé előtt fut le!
  - Nem tesztelhetünk szinkronizáló él által beállított globális változót
  - Nem „védhetünk meg” Guard-dal egy Sync-ben levő változót!
- A függvények működésének ellenőrzése nehézkes. Nincs lehetőség nyomkövetésre. Próbáljunk lépésenként haladni!
- Az „eventually”  $A \langle \rangle q$  tulajdonság használata esetén óra változókat kell használnunk a triviális ellenpélda kizárására.
  - Emlékezzünk a „leads to”  $p \rightarrow q$  szemantikájára:  $A[] (p \text{ imply } A \langle \rangle q)$
- Ne felejtsük el az óra változókat a megfelelően inicializálni!
- A csatorna-, vagy automataszintű prioritások használata esetén UPPAAL modellellenőrzője nem tudja a holtpontot kezelni. Ezek a modellezési elemek lehetőleg kerülendők.

14

# A feladat megoldása

Az eddig tanultak alkalmazásával

15

## A „házi feladat”

- Egy egyszerű operációs rendszer feladatainak és futtatási szálainak kezelését kell modellezni
  - Periodikus feladatvégrehajtás, kötött idejű intervallumok
  - Intervallum elején a feladatok véletlen választással döntenek a futási igényről
  - Minden feladat adott processzorigénnyel rendelkezik
  - Véges számú futtatási szál, egy feladathoz egy szál
  - Intervallum végén a feladatokat leállítják, egy futási periódus lezárul, az op. rendszer „alaphelyzetbe” kerül
  - A folyamat innentől előlről kezdődik

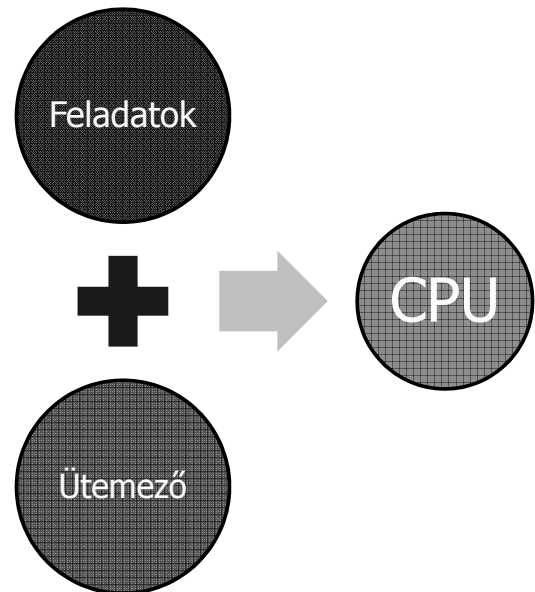
16



# A rendszert három fő komponens alkotja

- Feladatok

- Affinitás: ekkora valószínűséggel „kíván futni” az adott feladat
- Igény: az adott feladat ( $10 \cdot \text{Igény}$ ) százaléknyi processzorterhelést igényel
- Prioritás: a feladatok adott számú prioritási szinttel rendelkeznek



- csak  $\leq 100\%$  lehet a kiválasztott feladatok igényelte összes processzorterhelés
- az igények szabta korláton belül a feladatokat prioritási sorrend szerint kell kiválasztani

17

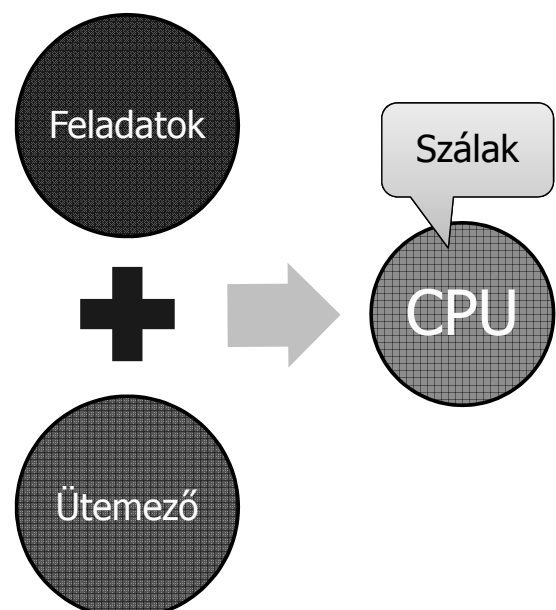
# A rendszert három fő komponens alkotja

- Ütemező

- kiválasztja a „futásra jelentkező” feladatok közül a futókat
- adott számú futtatási szál van
- minden feladatot adott szálhoz
- max. annyi futó feladat lehet, ahány szál van

- CPU

- erőforrás, a feladatok futtatásához szükséges
- két állapota van: aktív és inaktív
- aktív állapotában futhatnak a feladatok
- aktív állapot közben érkezhethet megszakítás, ami preemptív



18

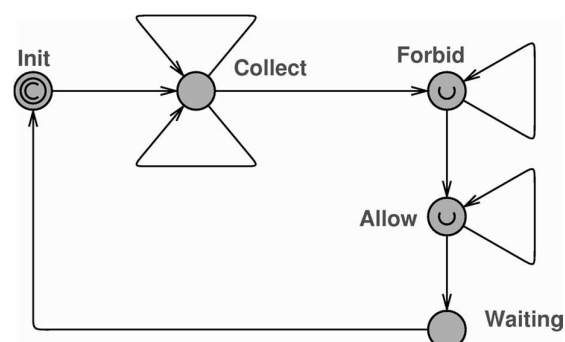
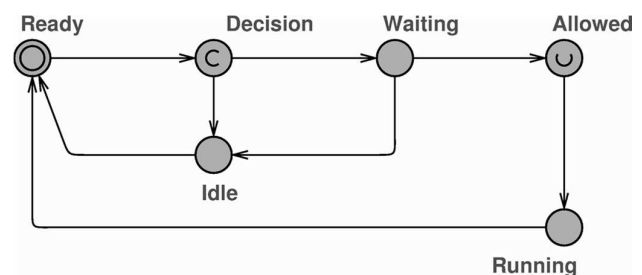
# A rendszer alapvető működése

- A feladatok
  - az alapállapotból kilépve egy 0 és 10 közötti  $p$  véletlen számot sorsolnak
  - ezt hasonlítják össze az Affinitás paraméterükkel, ha  $p \geq \text{Affinitás}$ , akkor futásra jelentkeznek, ha kisebb, akkor lemondanak a futásról és inaktívvá válnak
- Az ütemező
  - regisztrálja a futásra jelentkezéseket és lemondásokat
  - feldolgozza a jelentkezéseket: a korlátok betartása mellett prioritási szintben és az igényelt terhelés szerint is lefele haladva sorrendezi a feladatokat
  - a futásra kiválasztott feladatokat szálakhoz rendeli és ezt egy globális adatstruktúrában adminisztrálja

19

## Kezdjük el modellezni!

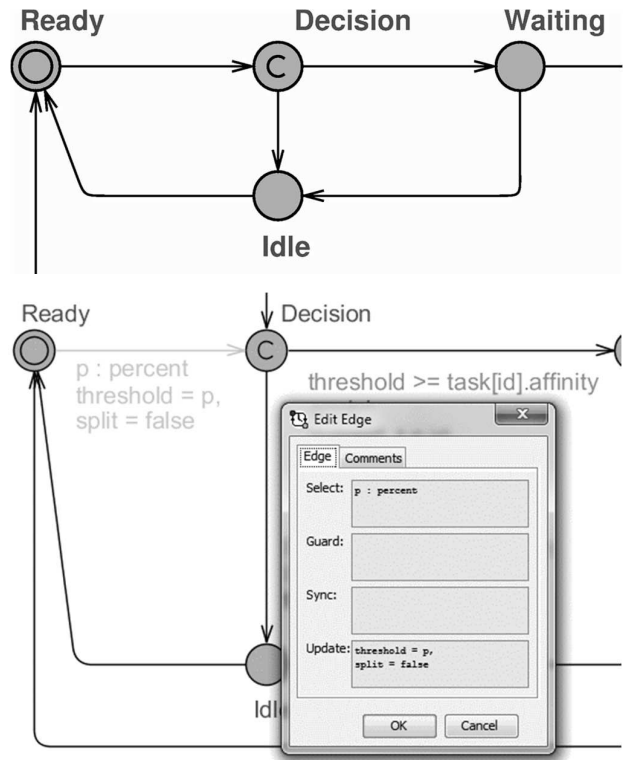
- Feladat
  - Ready: kezdőállapot
  - Decision: futási igényről dönt
  - Idle: lemondott és inaktív
  - Allowed: futásra kiválasztva
  - Running: fut
- Ütemező
  - Init: kezdőállapot
  - Collect: futásra jelentkezések és lemondások összegyűjtése
  - Forbid: értesíti a visszautasított feladatokat
  - Allow: értesíti a kiválasztott feladatokat
  - Waiting: periódus végére vár



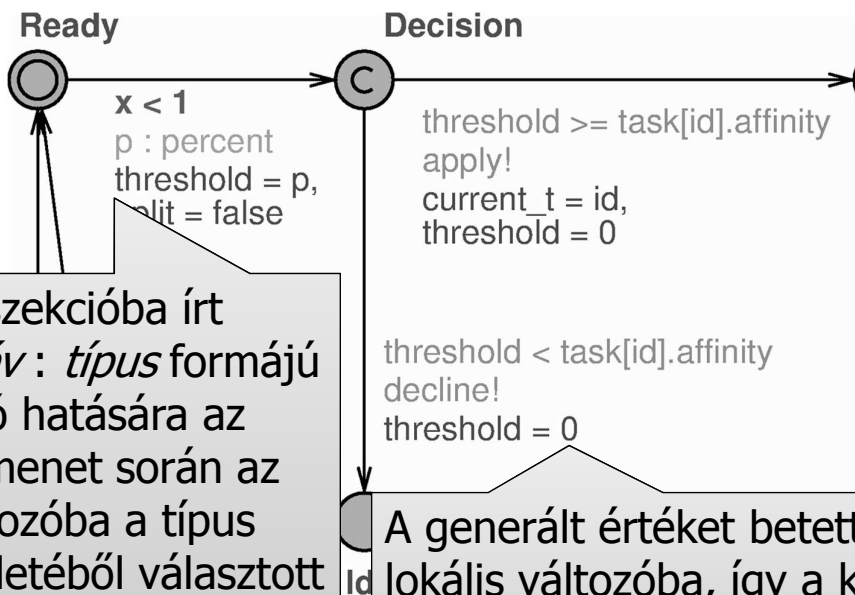
20

# Első probléma: véletlen választás modellezése

- Egyszerű megoldás
  - Működik? Igen, mert az UPPAAL az engedélyezett állapotátmenetek közül véletlenszerűen választ
  - Megfelelő? Nem, mert a valószínűségnek arányosnak kellene lennie az Affinitással
- Helyes megoldás
  - Véletlen érték generálása az UPPAAL Select konstrukciójával



## Véletlen választás modellezése



A Select szekcióba írt *változónév* : *típus* formájú deklaráció hatására az állapotátmenet során az adott változóba a típus értékészletéből választott véletlen érték kerül. Ez csak az adott állapotátmenet többi szekciójában használható!

A generált értéket betettük egy lokális változóba, így a következő lépésben össze tudjuk hasonlítani. A Committed állapot célja, hogy a két egymást követő lépést ne lehessen megszakítani.

# Deklarációk

## Globális

```
typedef int[0,10] percent;  
  
const int Levels = 3;  
typedef int[0,Levels-1] p_level;  
  
const int Tasks = 5;  
typedef int[0,Tasks-1] t_id;  
t_id current_t;  
  
typedef struct {  
    percent affinity;  
    percent demand;  
    p_level pri;  
} task_t;  
  
// affinity, demand, priority  
const task_t task[Tasks] = {  
    ...  
};
```

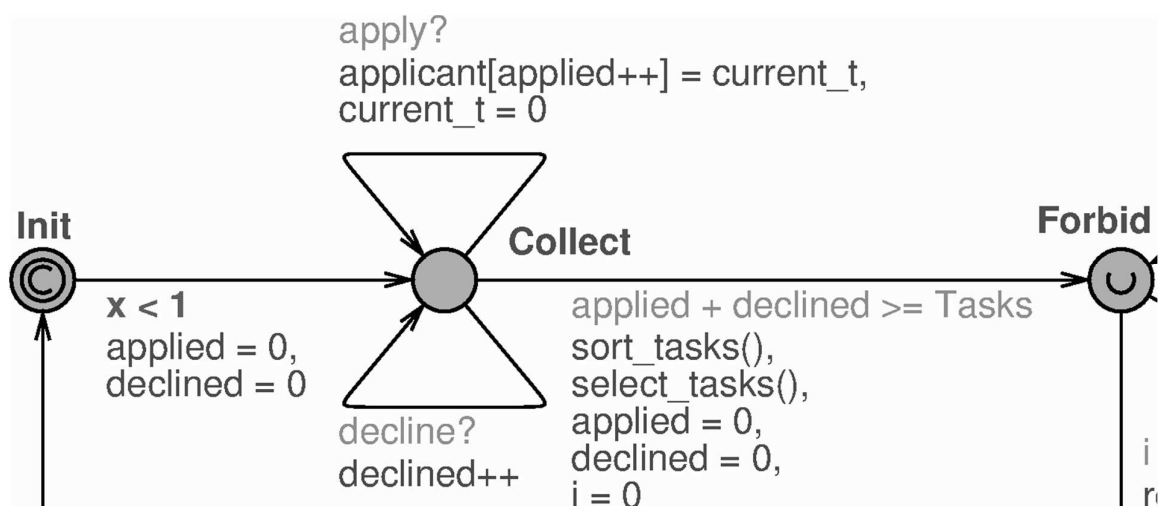
## Lokális (Task)

Name:  Parameters:

```
clock x;  
meta bool split = false;  
percent threshold;
```

23

## Hogyan működik a jelentkezések összeszámolása?

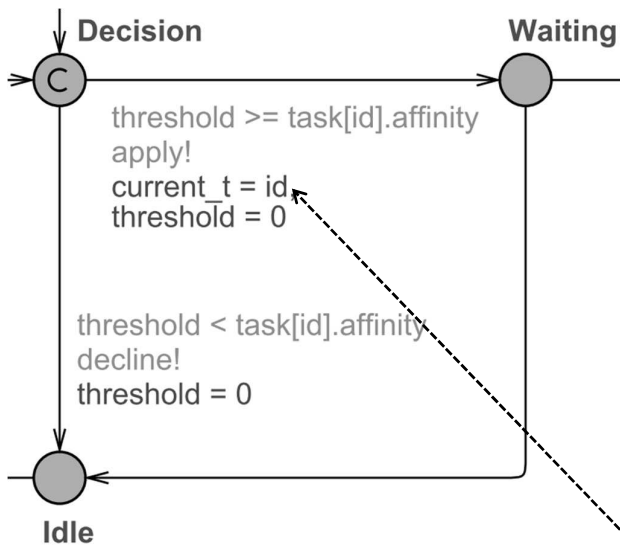


- Addig maradunk a Collect állapotban, amíg minden feladat vagy futásra jelentkezett, vagy a futásról lemondott
- A futásra jelentkezőket megjegyezzük egy lokális tömbben
- A Forbid állapotba lépéskor végrehajtott `sort_tasks()`, `select_tasks()` függvények végzik a feladatok kiválasztását

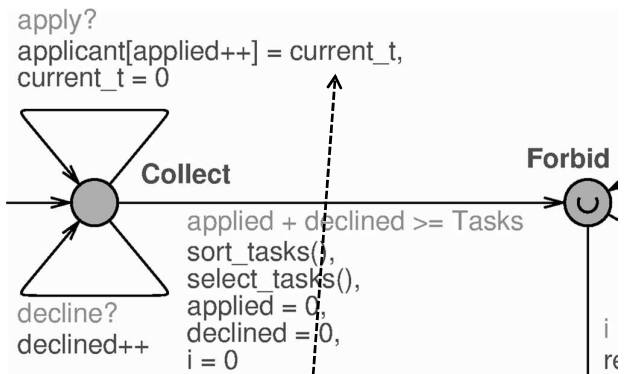
24

# Jelentkezések és lemondások gyűjtése

## Feladat



## Ütemező

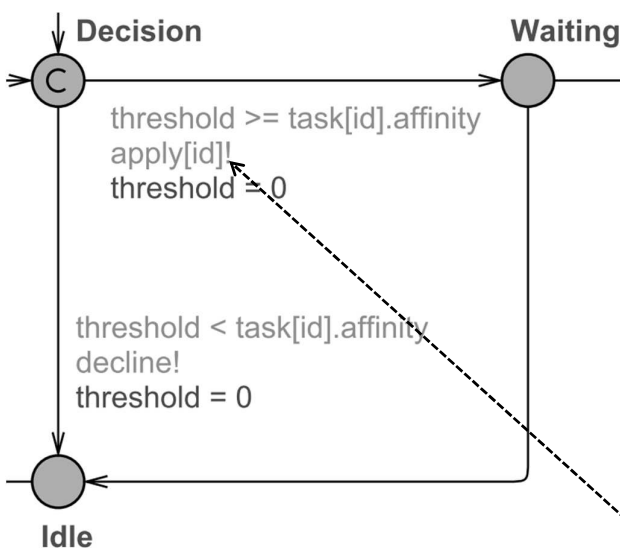


Szinkron kommunikáció modellezése globális változó segítségével.

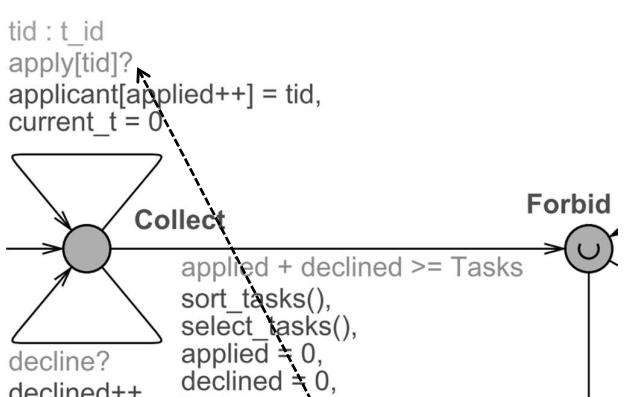
Garantált, hogy a „küldő” automata Update szekciója előbb hajtódik végre!

## Szinkron kommunikáció modellezése 2.

### Feladat



### Ütemező

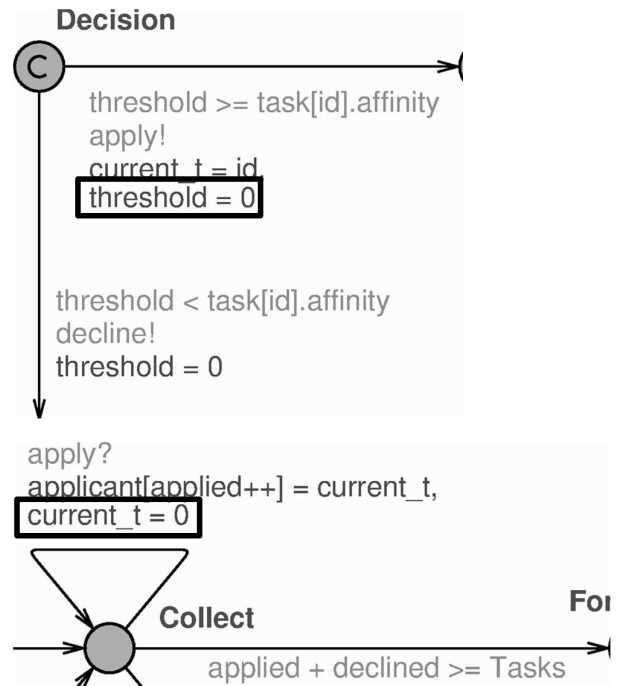


Szinkron kommunikáció modellezése csatorna tömbök segítségével.

Csak kis értékű változók esetén használható!

## Miért töröljük azonnal az ideiglenes változókat?

- Ideiglenes változó
  - minden értéke után új trajektóriasereg indul
  - megszozorozza az állapotteret
  - csökkenti: visszaállítás
- Átlapolás
  - aszinkron automaták lépései több sorrendben
  - azonos eredmény más-más úton
  - csökkenti: szinkronizáció, Committed állapot



27

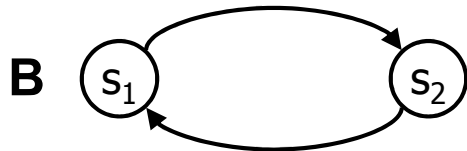
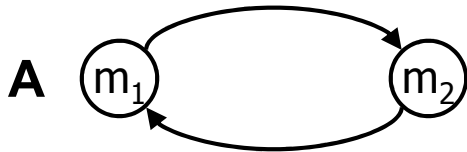
## Kitérő: két automata együttes működése

Automaták direkt szorzata, átlapolás, szinkronizáció

28

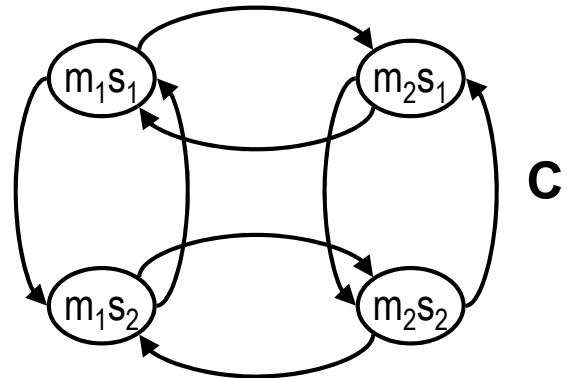
# Aszinkron automaták működése: átlapolás

- Két (független) automatából álló rendszer



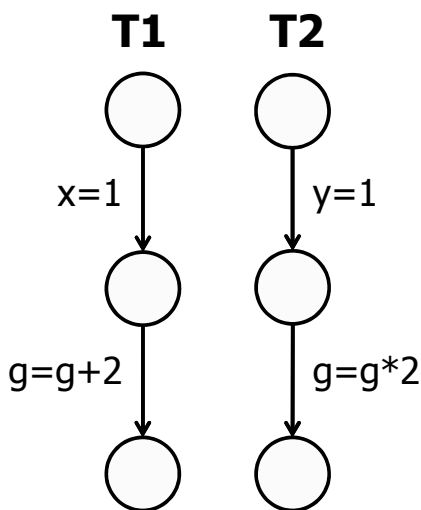
- Az automaták állapotai:  
 $A = \{m_1, m_2\}, B = \{s_1, s_2\}$

- (Direkt) szorzatautomata: a rendszer állapottere

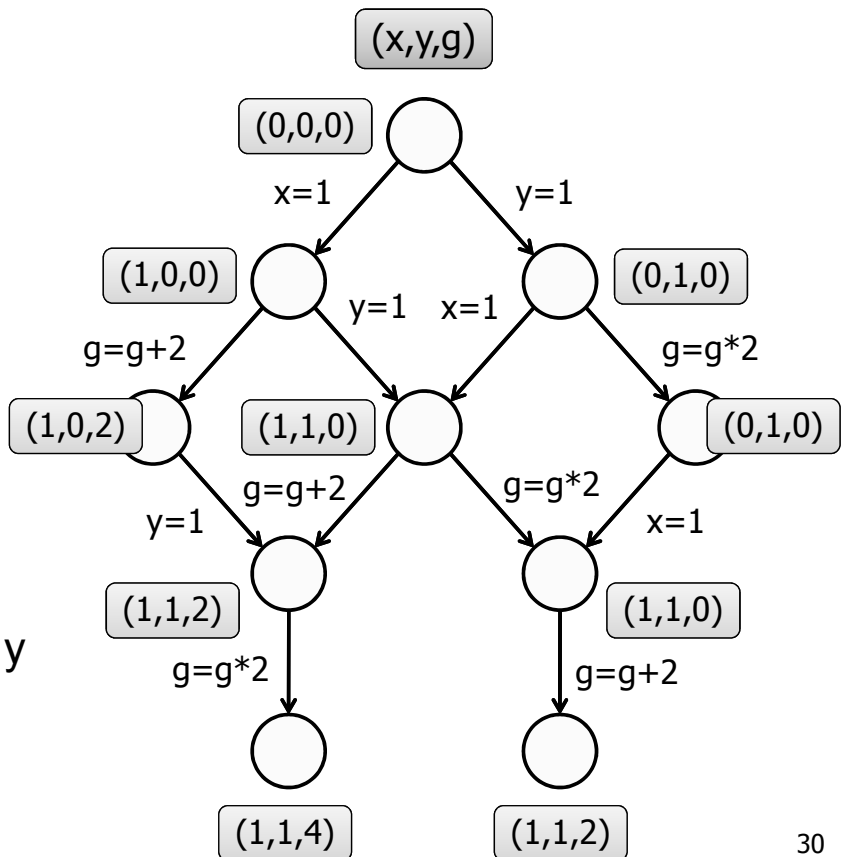


- Az állapotok halmaza:  
 $C = A \times B$   
 $C = \{m_1s_1, m_1s_2, m_2s_1, m_2s_2\}$

## Példa: alternatív utak

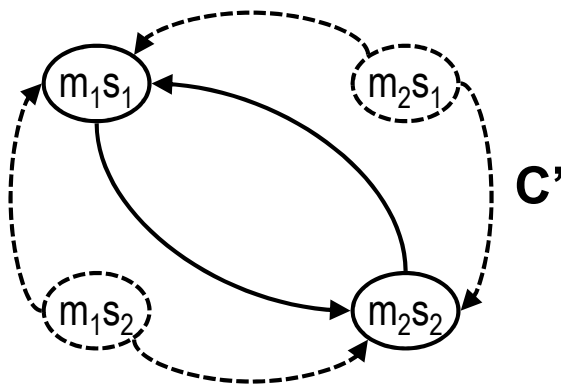


Lokális változók:  $x$  és  $y$   
 Globális változó:  $g$



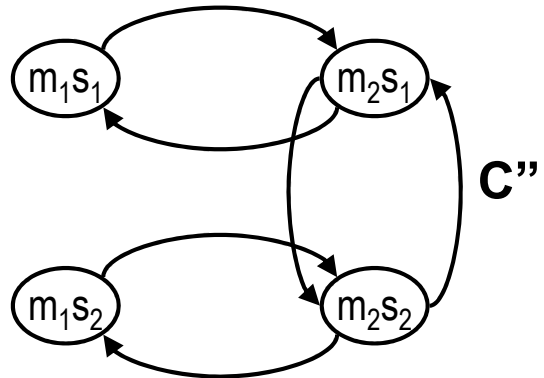
# Egyszerűsíti a modellt: szinkronizáció, feltétel

- Egyidejű állapotváltás: szinkronizáció



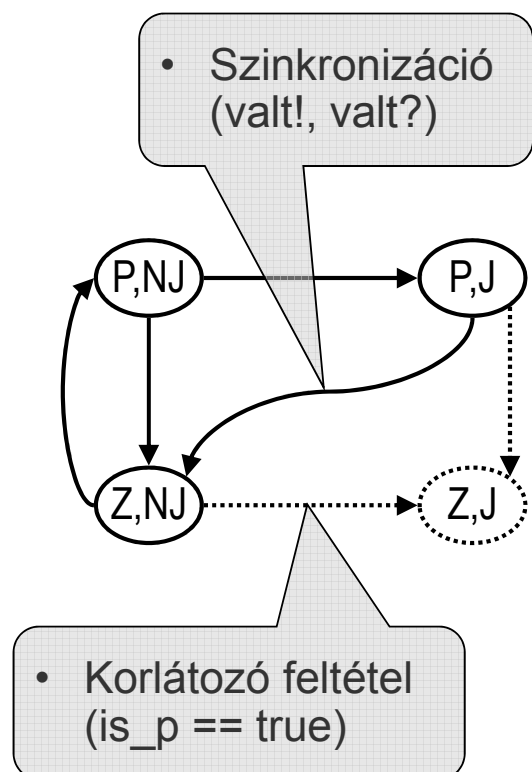
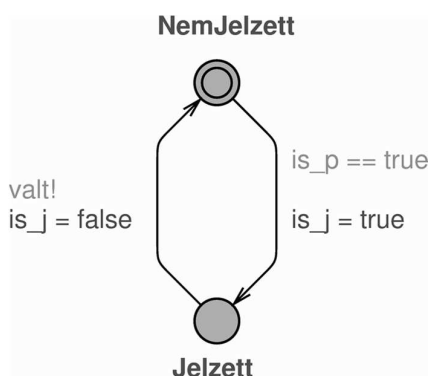
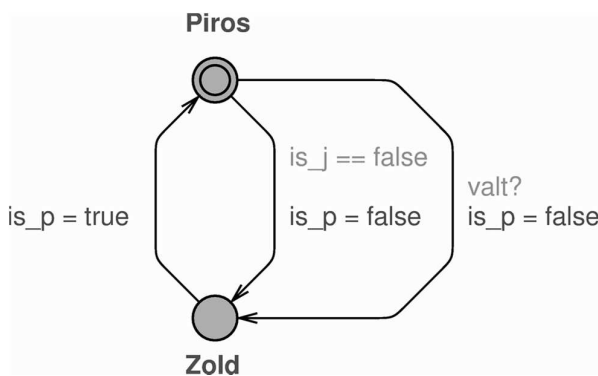
- pl. „A és B egyszerre vált állapotot, ha állapot indexük azonos”

- Korlátozó feltételek: állapotátmenetek tiltása



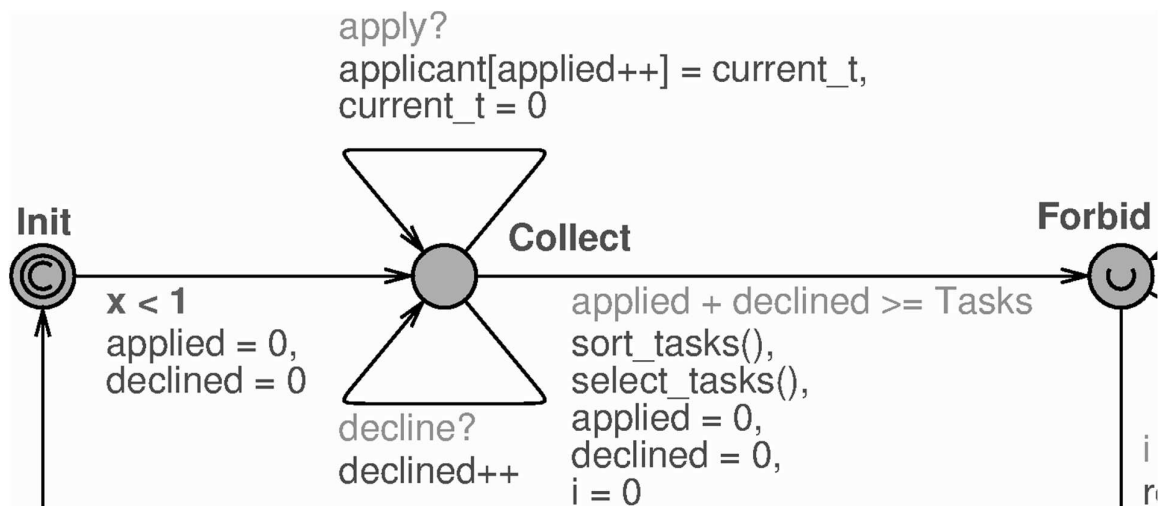
- pl. „B csak akkor vált állapotot, ha A m<sub>2</sub> állapotban van”

## Példa: gyalogos lámpa jelzőgombbal





## Térjünk vissza a feladat megoldásához



- A futásra jelentkezőket megjegyezzük egy lokális tömbben
- A Forbid állapotba lépéskor végrehajtott `sort_tasks()`, `select_tasks()` függvények végzik a feladatok kiválasztását
  - a korlátok betartása mellett prioritási szintben és az igényelt terhelés szerint is lefele haladva sorrendezi a feladatokat

33

## Feladatok kiválasztása és visszautasítása

- `sort_tasks()`
  - kétdimenziós tömböt használ a sorrendezéshez:

```
typedef struct {
    int[0,Tasks] length;
    t_id task[Tasks];
} buffer_t;
buffer_t buffer;
```
  - Legyen a paraméterezés:

```
// affinity, demand, priority
const task_t task[Tasks] = {
    {0, 2, 0},
    {3, 3, 1},
    {3, 4, 1},
    {3, 1, 1},
    {3, 5, 2}
};
```
  - Ha a jelentkezők: 0, 2, 3, 4
  - Akkor a sorrend:

```
buffer[0] = [2]
buffer[1] = [2, 3]
buffer[2] = [4]
```
  - Kiválasztva: 0, 2, 3
  - Visszautasítva: 4
- `select_tasks()`
  - prioritási szint szerint felfele haladva addig gyűjti a választott feladatokat, amíg valamelyik korlátba nem ütközik

34

# Feladatok sorrendezése CPU igény szerint

```
void insert_at(int[0,Tasks] pos, t_id tid) {
    int i;
    for (i = buffer.length; i > pos; i--) {
        buffer.task[i] = buffer.task[i - 1];
    }
    buffer.task[pos] = tid;
    buffer.length++;
}

void sort_tasks() {
    int i, j, pri, pos;
    for (i = 0; i < applied; i++) {
        pri = task[applicant[i]].pri;
        for (j = 0, pos = -1; j < buffer[pri].length && pos < 0; j++) {
            if (task[applicant[i]].demand > task[buffer[pri].task[j]].demand)
                pos = j;
        }
        insert_at(pri, pos < 0 ? buffer[pri].length : pos, applicant[i]);
        applicant[i] = 0;
    }
}
```

35

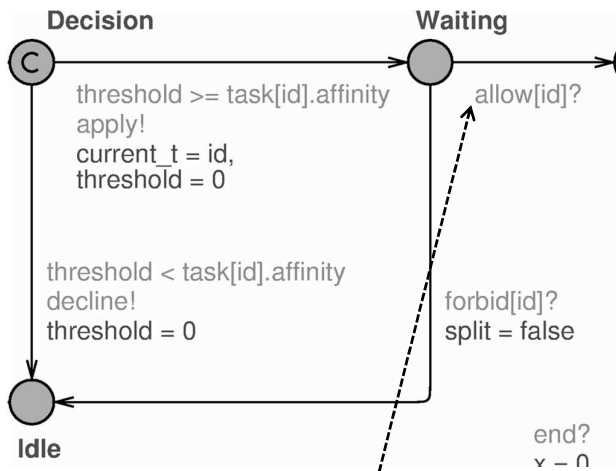
# Feladatok kiválasztása a korlátok betartásával

```
void select_tasks() {
    int i, pri;
    percent p = 0;
    rejected = 0;
    thread.num = 0;
    for (pri = 0; pri < Levels; pri++) {
        for (i = 0; i < buffer[pri].length; i++) {
            if (p + task[buffer[pri].task[i]].demand <= 10 &&
                thread.num < Threads) {
                thread.task[thread.num++] = buffer[pri].task[i];
                p = p + task[buffer[pri].task[i]].demand;
            }
            else applicant[rejected++] = buffer[pri].task[i];
            buffer[pri].task[i] = 0;
        }
        buffer[pri].length = 0;
    }
}
```

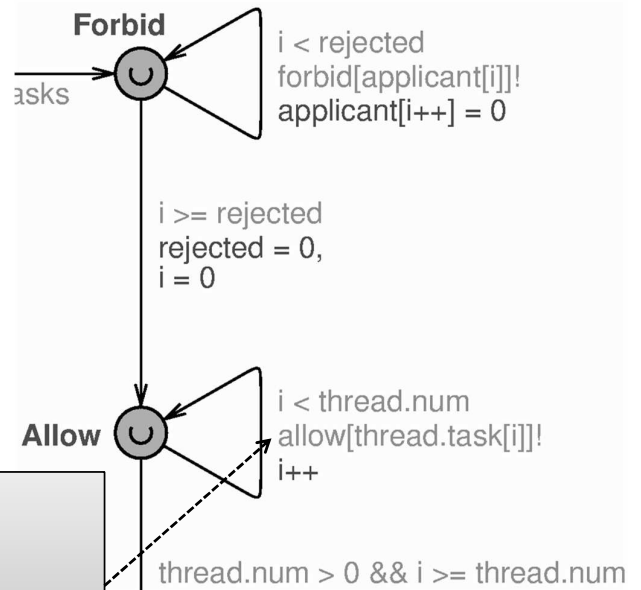
36

# Értesítés az kiválasztásról és visszautasításról

## Feladat



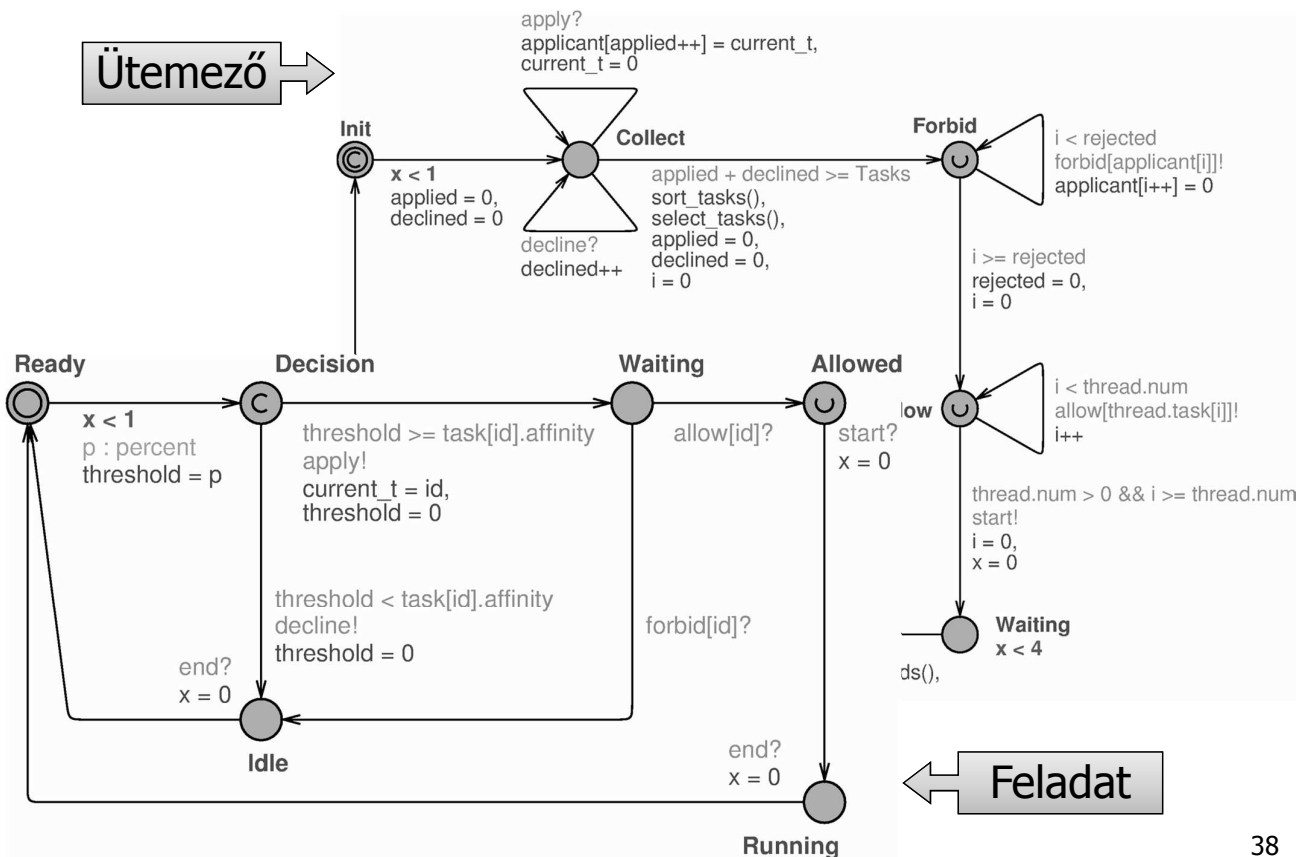
## Ütemező



A kiválasztott és visszautasított feladatokat egyenként, külön csatornán értesítjük. Az ideiglenes változókat töröljük.

# A modell már működőképes (CPU nélkül is)

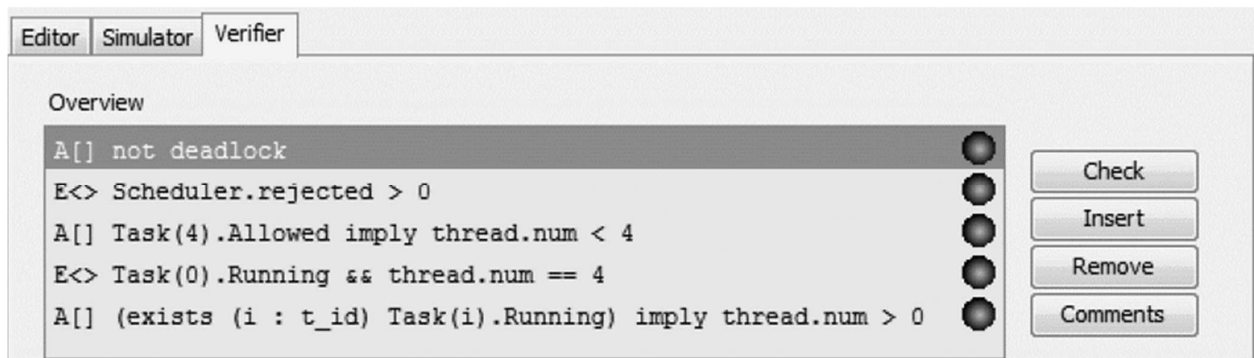
Ütemező



Feladat

# Köztes ellenőrzés

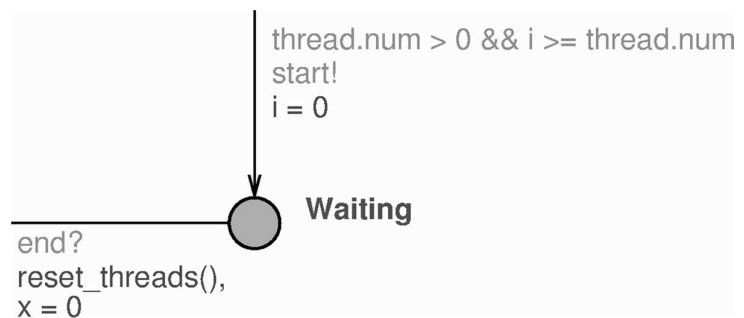
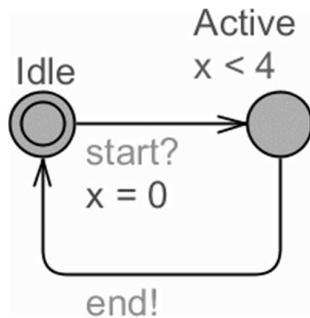
- Most már van egy működőképes rendszerünk
  - érdemes ellenőrizni az eddig elkészültek helyességét
- Néhány elvárt tulajdonság és annak ellenőrzése:
  1. A rendszerben nincs holtpon (deadlock).
  2. Lehetséges, hogy az ütemező visszautasít egy feladatot.
  3. A 4-es feladat kiválasztása esetén nem lehet minden szál foglalt.
  4. Lehetséges, hogy minden szál foglalt.
  5. Nem lehetséges, hogy ha van futó feladat, ne legyen foglalt szál.



## A modell kiegészítése a CPU-val

- A startjelet az ütemező adja ki egy broadcast csatornán keresztül. Ennek hatására:
  - a futásra kiválasztott feladatok futó állapotba kerülnek,
  - az ütemező várakozó állapotba kerül, amiből majd csak a végjel hatására fog továbblépni,
  - a CPU aktív állapotba kerül, a rajta futó szálak és feladatok listáját egy globális adatstruktúra tartalmazza.
- Az aktív állapot elhagyásakor a CPU egy broadcast csatornán egy végjelet küld. Ennek hatására:
  - a CPU inaktív állapotba kerül,
  - az ütemező alapállapotba kerül, a futó szálak és feladatok listáját alaphelyzetbe (üres) állítja,
  - a feladatok is alapállapotba kerülnek.

## Indítás és leállítás a CPU és az ütemező esetén



```
const int Threads = 4;
```

```
typedef struct {  
    int[0,Threads] num;  
    t_id task[Threads];  
} thread_t;
```

```
thread_t thread;
```

```
chan apply, decline;
```

```
urgent chan allow[Tasks], forbid[Tasks];
```

```
chan suspend[Tasks];
```

```
broadcast chan start, end;
```

```
void reset_threads() {
```

```
    while (thread.num > 0)
```

```
        thread.task[thread.num-- - 1] = 0;
```

```
}
```

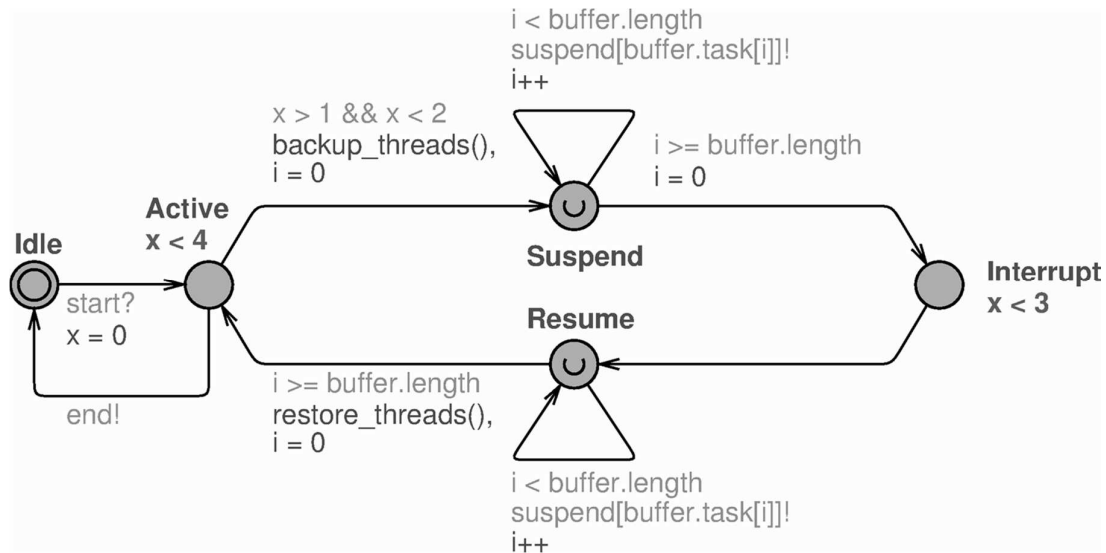
41

## Tovább bonyolítjuk a modellt: megszakítás!

- CPU aktív állapotában érkezhethet egy NMI megszakítás
  - preemptív módon félbeszakítja egyes feladatok futását
  - a megszakítás CPU igénye dönti el, hogy mely feladatok kerülnek felfüggesztésre
  - annyi feladatot kell felfüggeszteni (a legkisebb prioritásúaktól felfelé haladva), amennyi elég CPU kapacitást szabadít fel ahhoz, hogy a megmaradó feladatok és a megszakítás-kezelő rutin CPU igényeinek összege  $\leq 100\%$  legyen
    - a felfüggesztendő feladatok kiválasztását a CPU végzi
    - értesíti is a felfüggesztésre kiválasztott feladatokat
    - ezek futó állapotból felfüggesztett állapotba kerülnek
  - a megszakítás-kezelő rutin lefutása után
    - a CPU értesíti a korábban felfüggesztett feladatokat
    - ezek ennek hatására újra futó állapotba kerülnek
    - ezután a CPU is újra futó állapotba kerül

42

# Megszakítás megvalósítása



- A feladatok felfüggesztését a `backup_threads()`, az újra futó állapotba helyezést a `restore_threads()` függvények végzik
- A felfüggesztésről és az újra futó állapotba helyezésről a feladatokat egyenként, külön csatornán értesítjük

43

## Felfüggesztendő feladatok kiválasztása

```
void backup_threads() {
    int i, p;
    t_id tid;
    for (i = 0, p = 0; i < thread.num; i++)
        p += task[thread.task[i]].demand;
    buffer.length = 0;
    for (i = 0; i < thread.num; i++) {
        if (p + i_demand > 10) {
            tid = thread.task[thread.num - i - 1];
            buffer.task[buffer.length++] = tid;
            thread.task[thread.num - i - 1] = 0;
            p -= task[tid].demand;
        }
    }
    thread.num -= buffer.length;
}
```

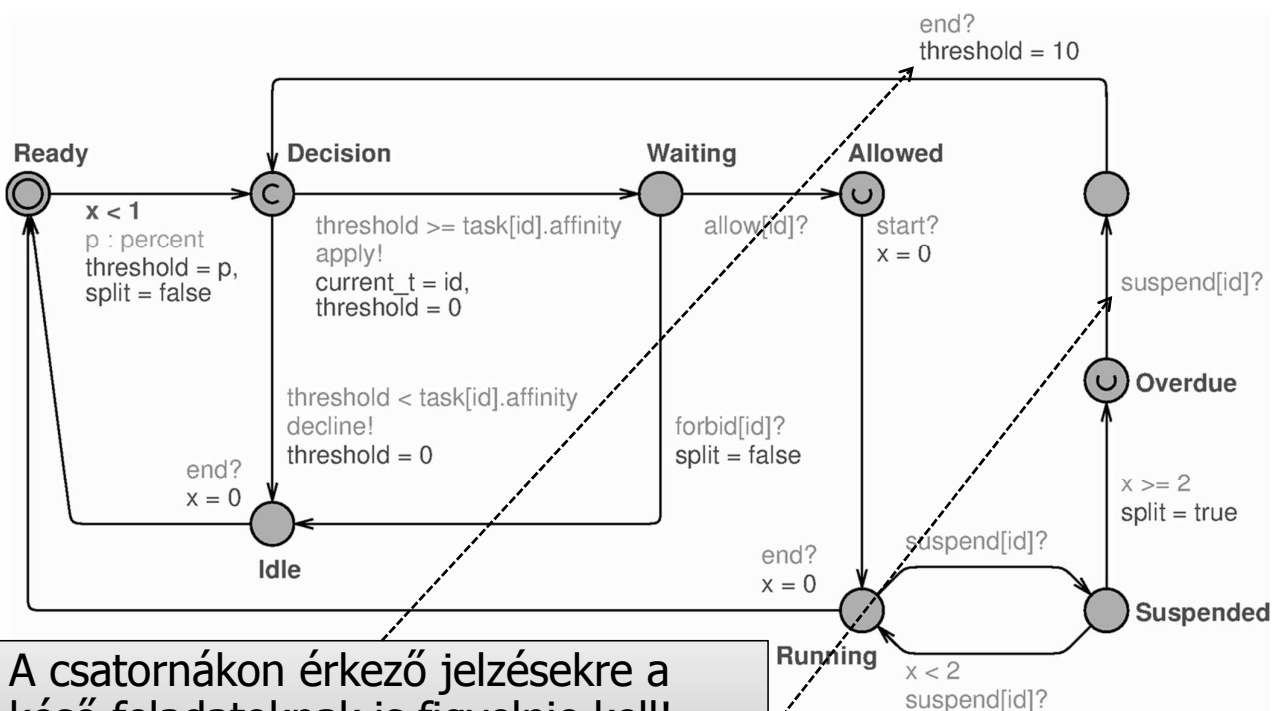
44

## Tovább bonyolítjuk a modellt: késő feladatok

- Ha a felfüggesztett feladatok túl sok időt töltenek felfüggesztett állapotban, akkor annyi „késést” szednek össze, hogy nem tudják befejezni a teendőiket az adott futási periódusban
- A késő feladatok „megkísérlik” a következő futási periódusban folytatni a végrehajtást
- Ezt úgy modellezzük, hogy a végjel hatására nem a kezdőállapotba kerülnek, hanem abba az állapotba mennek át, amiben a futásra fognak jelentkezni a következő futási periódusban
  - (tehát átugorják a véletlen döntési fázist)

45

## A feladatok modelljének kiegészítése a késéssel



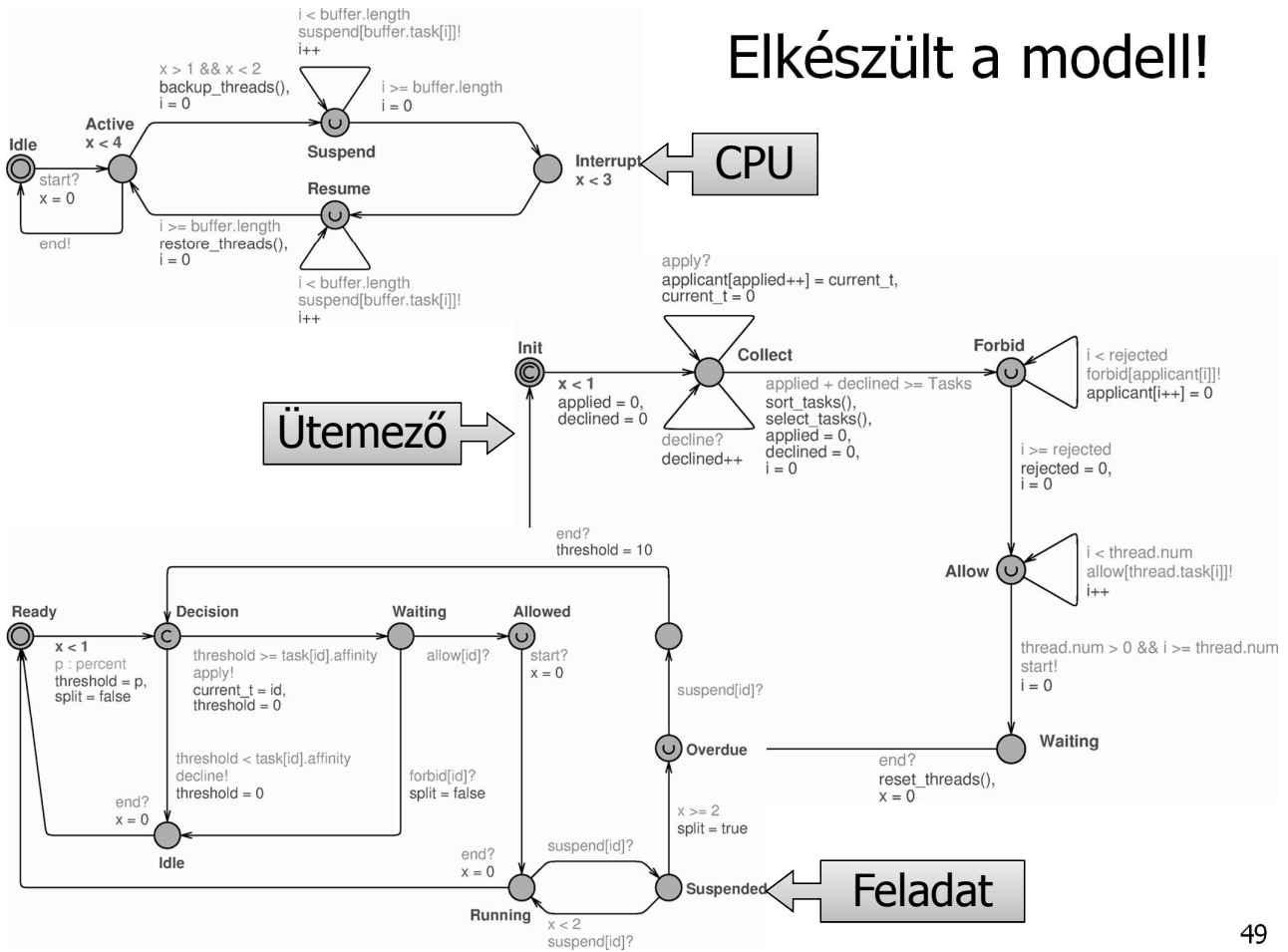
A csatornákon érkező jelzésekre a késő feladatoknak is figyelnie kell!  
A 10-es érték biztosítja a jelentkezést.

46





# Elkészült a modell!



49

## Az ellenőrzendő követelmények

1. A modellben nincs deadlock.
2. Lehetséges, hogy a futásra jelentkezett feladatok közül legalább egyet vissza kell utasítani.
3. Lehetséges, hogy összes szál foglalt, azaz maximális számú feladat fut.
4. Ha van futó feladat, akkor a globális adatstruktúrában a foglalt szálak száma  $> 0$ .
5. Lehetséges, hogy a CPU megszakítás bekövetkezte esetén  $> 2$  szál felfüggeszt.
6. Lehetséges olyan lefutás, hogy egyetlen feladatot sem függesztenek fel egyetlen futási periódusban sem, de nem lehetséges, hogy minden lefutás ilyen: azaz van legalább egy olyan lefutás, amiben legalább egyszer legalább egy feladatot felfüggesztenek.
7. Nem lehetséges, hogy egy feladat a 3. időegység után felfüggesztett állapotban legyen.
8. Ha egy feladatot felfüggesztenek, akkor lehetséges, hogy valamikor be tudja fejezni a feladatát.

50

## Az ellenőrzéshez használt temporális kifejezések

### Overview

1. `A[] not deadlock`
2. `E<> Scheduler.rejected > 0`  
`A[] Task(4).Allowed imply thread.num < 4`
3. `E<> CPU.Active && thread.num == 4`  
`E<> CPU.Interrupt && CPU.buffer.length > 1`
4. `A[] (exists (i : t_id) Task(i).Running) imply thread.num > 0`
5. `E<> CPU.Interrupt && CPU.buffer.length > 2`
6. `E[] (forall (i : t_id) Task(i).split == false)`  
`A[] (forall (i : t_id) Task(i).split == false)`
7. `E<> (exists (i : t_id) Task(i).Overdue && Task(i).x > 3)`
8. `Task(1).Overdue --> Task(1).split == true`  
`E<> (exists (i : t_id) Task(i).Overdue && Task(i).split == true)`