

A modellellenőrzési feladat (bevezető)

dr. Majzik István

BME Méréstechnika és Információs Rendszerek Tanszék

Ismétlés: Mit szeretnénk elérni?

Alacsony szintű modellek:

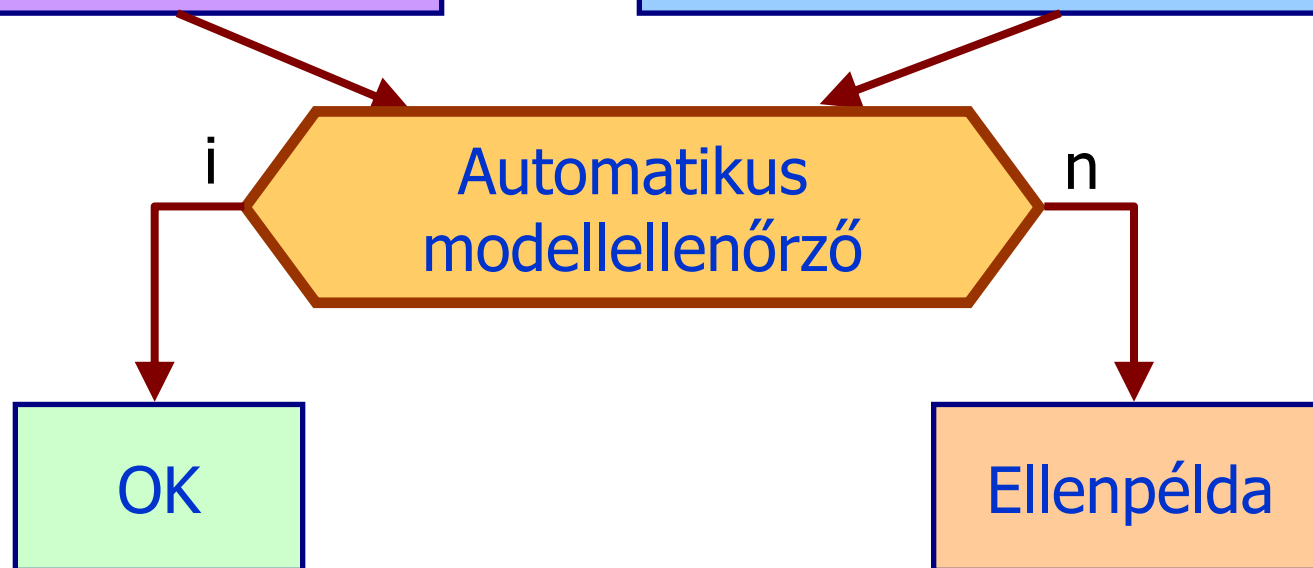
- KS, LTS, KTS
- Időzített automata

Állapot elérhetőségi követelmények:

- Temporális logikák:
lineáris / elágazó idejű

Rendszer modellje

Követelmény megadása

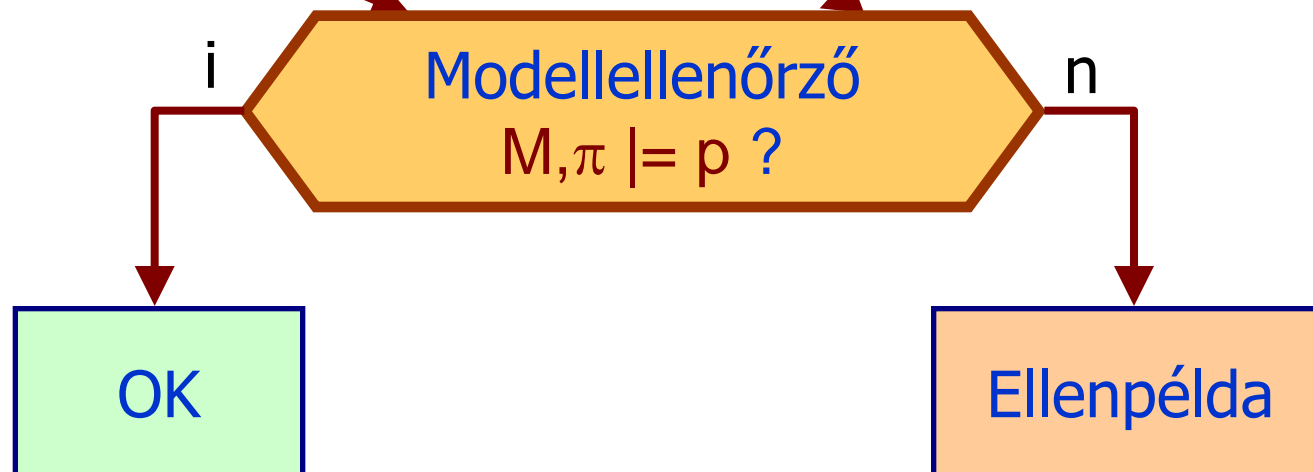


PLTL modellellenőrzés

Ha nincs útvonal megadva, akkor a kezdőállapotból induló minden útra ellenőriz!

Kripke struktúra M

PLTL kifejezés p



A SPIN modellellenőrző (régi felület)

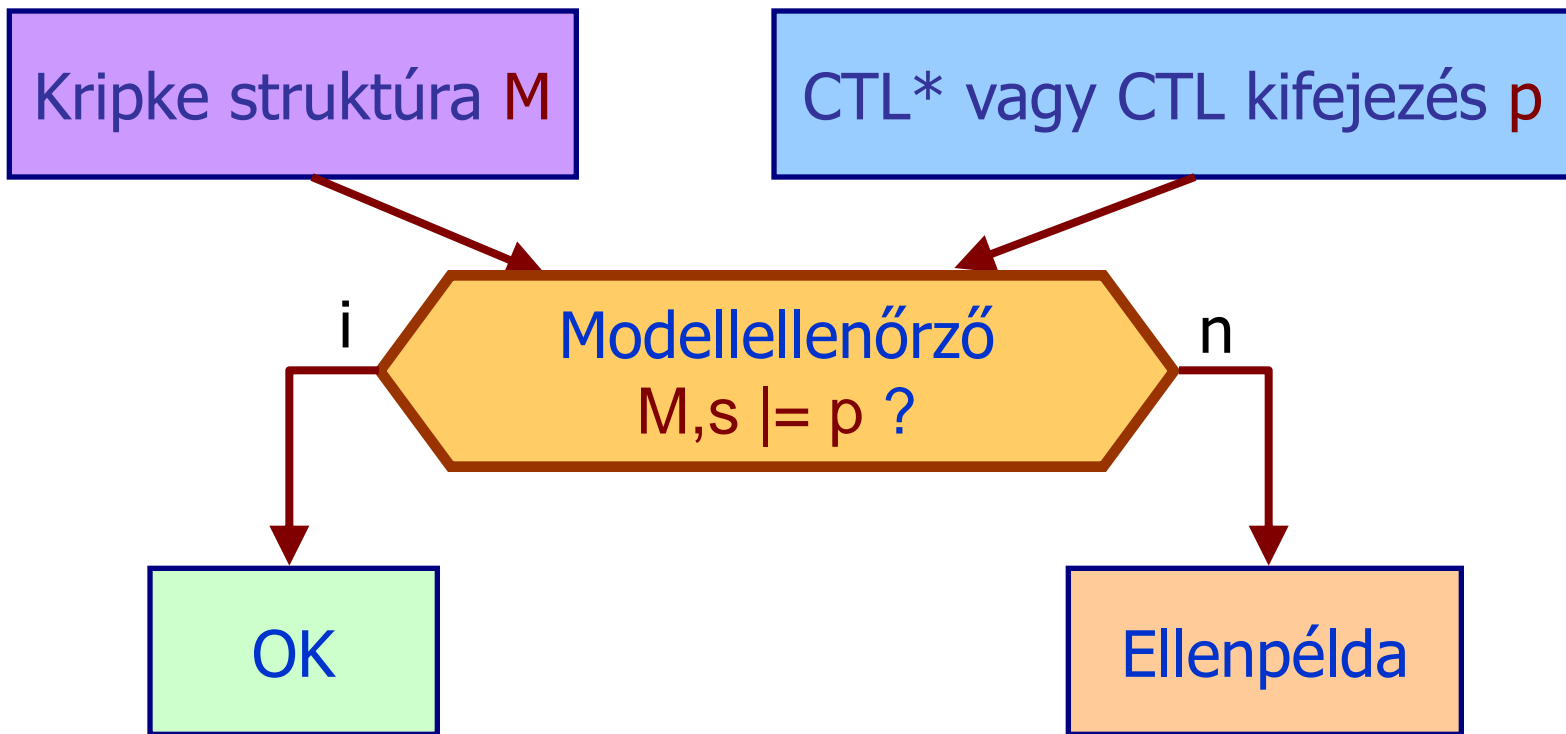
The screenshot shows the SPIN model checker interface. The title bar reads "Linear Time Temporal Logic Formulae". The main window contains a "Formula:" field with the text "<>[]oneLeader" and a "Load..." button. Below this is an "Operators:" field with buttons for [], <>, U, ->, and or/and. A "Property hold:" section has two radio buttons: "All Executions (desired behavior)" (selected) and "No Executions (error behavior)". The bottom section, labeled "Symbolic", contains a code block with the following content:

```
#define elected (nr_leaders > 0)
#define noLeader (nr_leaders == 0)
#define oneLeader (nr_leaders == 1)
```

Three callout boxes provide additional information:

- PLTL operátorok jelölése:**
F megfelelője <>
G megfelelője []
(X operátor nem található)
- Útvonalak kezelése**
- Címkék a modell állapotváltozói segítségével definiálhatók**

CTL* vagy CTL modellellenőrzés



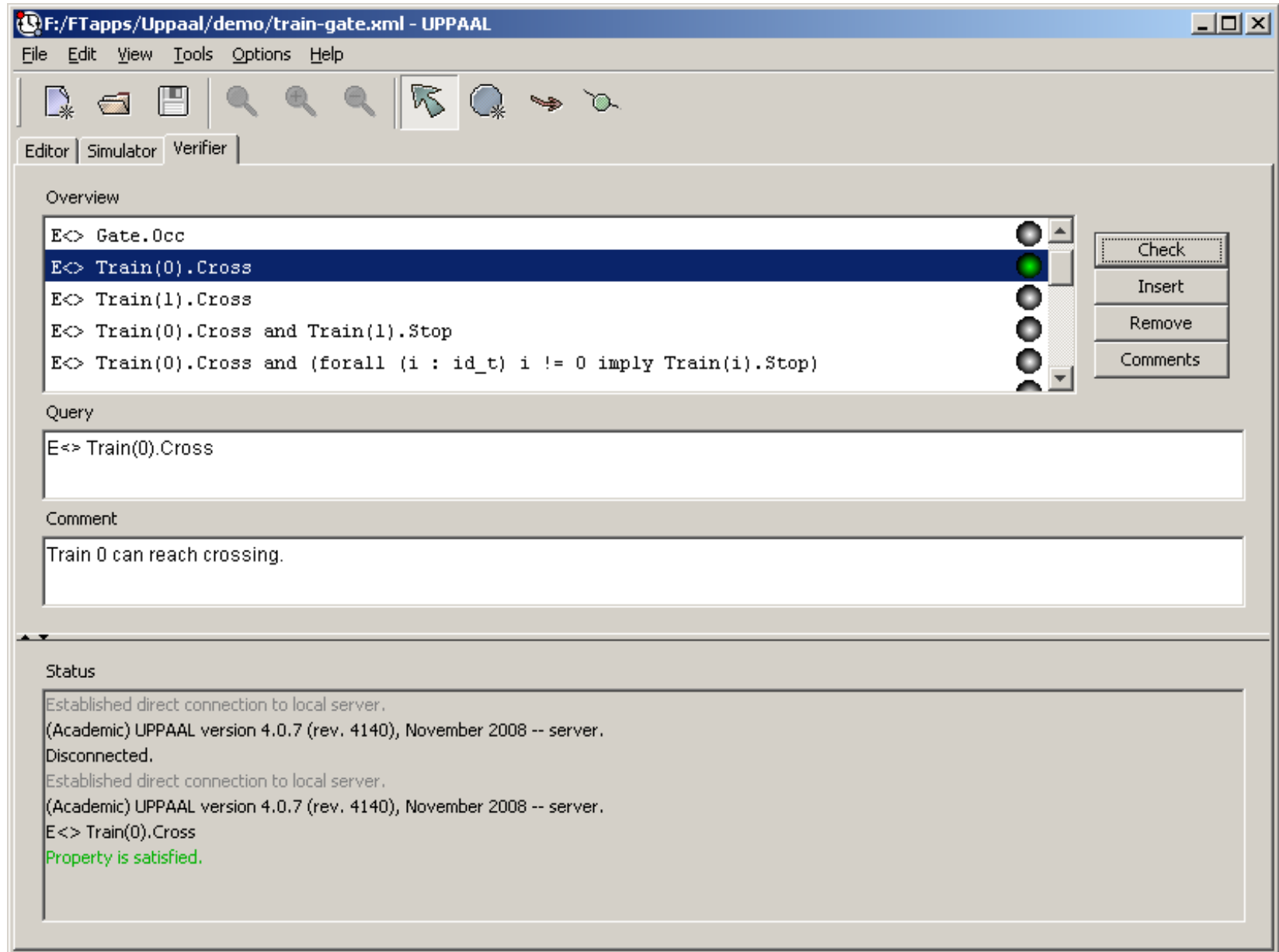
Az UPPAAL modellellenőrző lehetőségei

- **Atomi kijelentések:**
 - Változók értéke hivatkozható: pl. $a \neq 1$
 - Egész aritmetika és bitenkénti műveletek használhatók
 - Állapot hivatkozható: pl. $\text{Train}(0).\text{cross}$
 - Paraméterezett processzekre: forall , exists operátorok
 - Holtpont: **deadlock** kijelentéssel megadható (nincs akció)
- **Boole logikai operátorok:**
 - and , or , imply , not , $?$: (ez utóbbi az if-then-else)
- **Temporális operátorok: Korlátozott CTL**
 - Kifejezés elején szerepelhet egy temporális operátor
 - Speciális lehetőség: $p \rightarrow q$, jelentése $\text{AG}(p \text{ imply } \text{AF } q)$
 - Jelölés: $[]$ szerepel G helyett, $\langle \rangle$ szerepel F helyett
 - Így lesz: $A[], A\langle \rangle, E[], E\langle \rangle$
 - $E[]$ esetén véges útvonalon is értelmezett (végállapotig)

Követelmények ellenőrzése az UPPAAL-ban

- Követelmények halmaza szerkeszthető
- Modellellenőrzés egyenként is indítható
- Ellenpélda generálható
 - Legrövidebb, leggyorsabb, vagy akármi
 - Betölthető a szimulátorba (végigjátszható)
- Keresés az állapottérben
 - Mélységi
 - Szélességi
- Állapottárolás különféle opciókkal
 - Redukció
 - Közelítő állapottér tárolás (alul- illetve felülbecslés)
 - Hash tábla mérete megadható

Az UPPAAL modellellenőrző ablaka



Ellenpélda az UPPAAL szimulátorban

F:/FTapps/Uppaal/demo/train-gate.xml - UPPAAL

File Edit View Tools Options Help

Editor Simulator Verifier

Drag out

Enabled Transitions

- appr[2]: Train(2) --> Gate
- appr[3]: Train(3) --> Gate
- appr[4]: Train(4) --> Gate
- appr[5]: Train(5) --> Gate
- leave[0]: Train(0) --> Gate

Next Reset

Simulation Trace

```
(Safe, Safe, Safe, Safe, Safe, Safe, Free)
appr[0]: Train(0) --> Gate
(Appr, Safe, Safe, Safe, Safe, Safe, Occ)
Train(0)
(Cross, Safe, Safe, Safe, Safe, Safe, Occ)
appr[1]: Train(1) --> Gate
(Cross, Appr, Safe, Safe, Safe, Safe, -)
stop[tail():] Gate --> Train(1)
(Cross, Stop, Safe, Safe, Safe, Safe, Occ)
```

Trace File:

Prev Next Replay

Open Save Auto

Slow Fast

Drag out

```
Gate.list[0] = 0
Gate.list[1] = 1
Gate.list[2] = 0
Gate.list[3] = 0
Gate.list[4] = 0
Gate.list[5] = 0
Gate.list[6] = 0
Gate.len = 2
Train(0).x in [0,5]
Train(1).x in [0,5]
Train(2).x >= 10
Train(3).x >= 10
Train(4).x >= 10
Train(5).x >= 10
Train(0).x - Train(2).x <= -10
Train(1).x - Train(0).x in [-5,0]
Train(2).x = Train(3).x
Train(3).x = Train(4).x
Train(4).x = Train(5).x
Train(5).x = Train(2).x
```

Train(0)

Train(1)

Train(0) Train(1) Train(2) Train(3) Train(4) Train(5) Gate

A mintapélda befejezése

Mintapélda: Kölcsönös kizárás

- 2 résztvevőre, 3 megosztott változóval (H. Hyman, 1966)
 - **blocked0**: Első résztvevő (P0) be akar lépni
 - **blocked1**: Második résztvevő (P1) be akar lépni
 - **turn**: Ki következik belépni (0 esetén P0, 1 esetén P1)

```
while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```

P0

```
while (true) {
    blocked1 = true;
    while (turn!=1) {
        while (blocked0==true) {
            skip;
        }
        turn=1;
    }
    // Critical section
    blocked1 = false;
    // Do other things
}
```

P1

Helyes-e ez az algoritmus?

A modell UPPAAL-ban

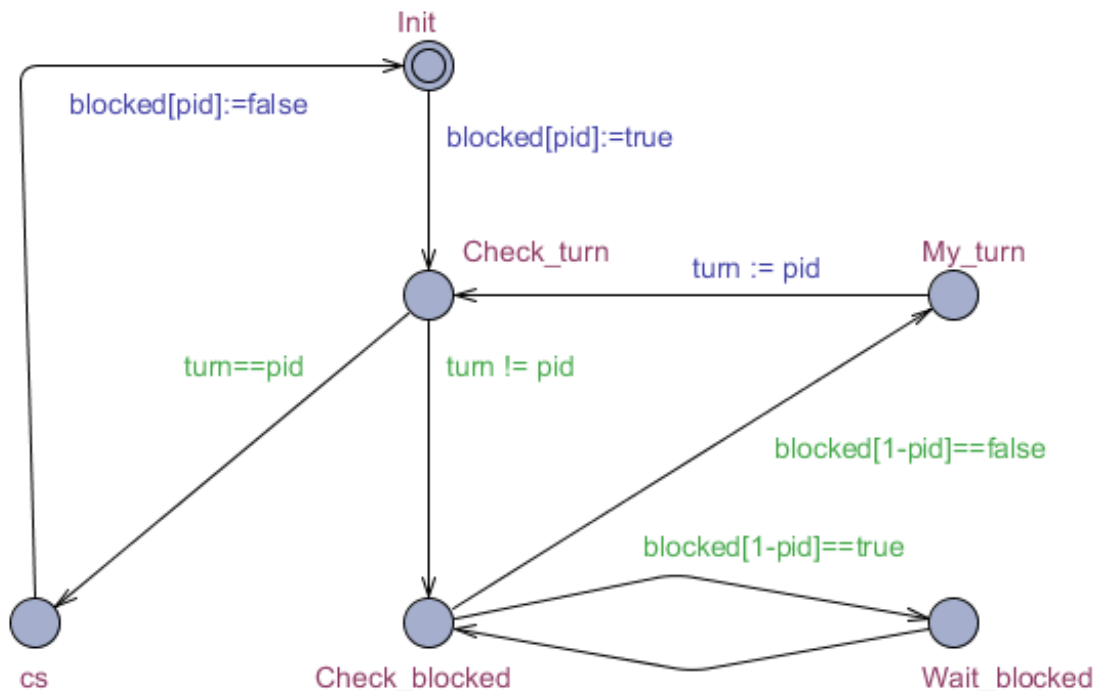
Deklarációk:

```
bool blocked[2];  
int[0,1] turn;  
P0 = P(0);  
P1 = P(1);  
system P0,P1;
```

A P automata template
pid paraméterrel:

Kihasztnált modellezési lehetőségek:

- Közös változók rendszerszintű deklarációja
- Korlátozott értékészletű változók
- Azonos viselkedésű résztvevők azonos automata template alapján
- Példányosítás paraméterezéssel
- Változó tömbök (résztvevőkhöz)



```
while (true) { P0  
  blocked0 = true;  
  while (turn!=0) {  
    while (blocked1==true) {  
      skip;  
    }  
    turn=0;  
  }  
  // Critical section  
  blocked0 = false;  
  // Do other things  
}
```

UPPAAL: A követelmények formalizálása

- Kölcsönös kizárás:
 - Egyszerre csak az egyik résztvevő lehet a kritikus szakaszban: $A[] \text{ not } (P0.cs \text{ and } P1.cs)$
- Holtpontmentesség:
 - Nem alakul ki leállás (amikor nincs továbblépés): $A[] \text{ not deadlock}$
- Lehetséges az elvárt viselkedés:
 - P0 beléphet a kritikus szakaszba: $E<> (P0.cs)$
 - P1 beléphet a kritikus szakaszba: $E<> (P1.cs)$
- Nincs kiéheztetés:
 - P0 mindenképpen be fog lépni a kritikus szakaszba: $A<> (P0.cs)$
 - P1 mindenképpen be fog lépni a kritikus szakaszba: $A<> (P1.cs)$

UPPAAL: A modellellenőrzés eredménye

- A kölcsönös kizárás nem teljesül!
 - Ellenpélda: Átlapolódás a két résztvevő között (végigjátszható a szimulátorban)
 - Javítás: Pl. Peterson, Dekker algoritmus

Hyman:

```
while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```

Peterson:

```
while (true) {
    blocked0 = true;
    turn=1;
    while (blocked1==true &&
        turn!=0) {
        skip;
    }

    // Critical section
    blocked0 = false;
    // Do other things
}
```

UPPAAL: A modellellenőrzés eredménye

- A kölcsönös kizárás nem teljesül!
 - Ellenpélda: Átlapolódás a két résztvevő között (végigjátszható a szimulátorban)
 - Javítás: Pl. Peterson, Dekker algoritmus
- Nincs holtpont
- Lehetséges az elvárt viselkedés
- A kiéheztetés elkerülése nem teljesül
 - Triviális ellenpélda: A kiinduló állapotban marad
 - Ez megengedett (ld. időfüggő viselkedés modellezése)
 - Korlátozni kell egy-egy vezérlési helyen tölthető időt
 - Ezután is lehet kiéheztetés?
 - Van rá példa (az egyik processz ciklikus működése)
 - Az algoritmus önmagában nem garantálja a kiéheztetés elkerülését, fair ütemezés is szükséges

Dekker algoritmus (demo)

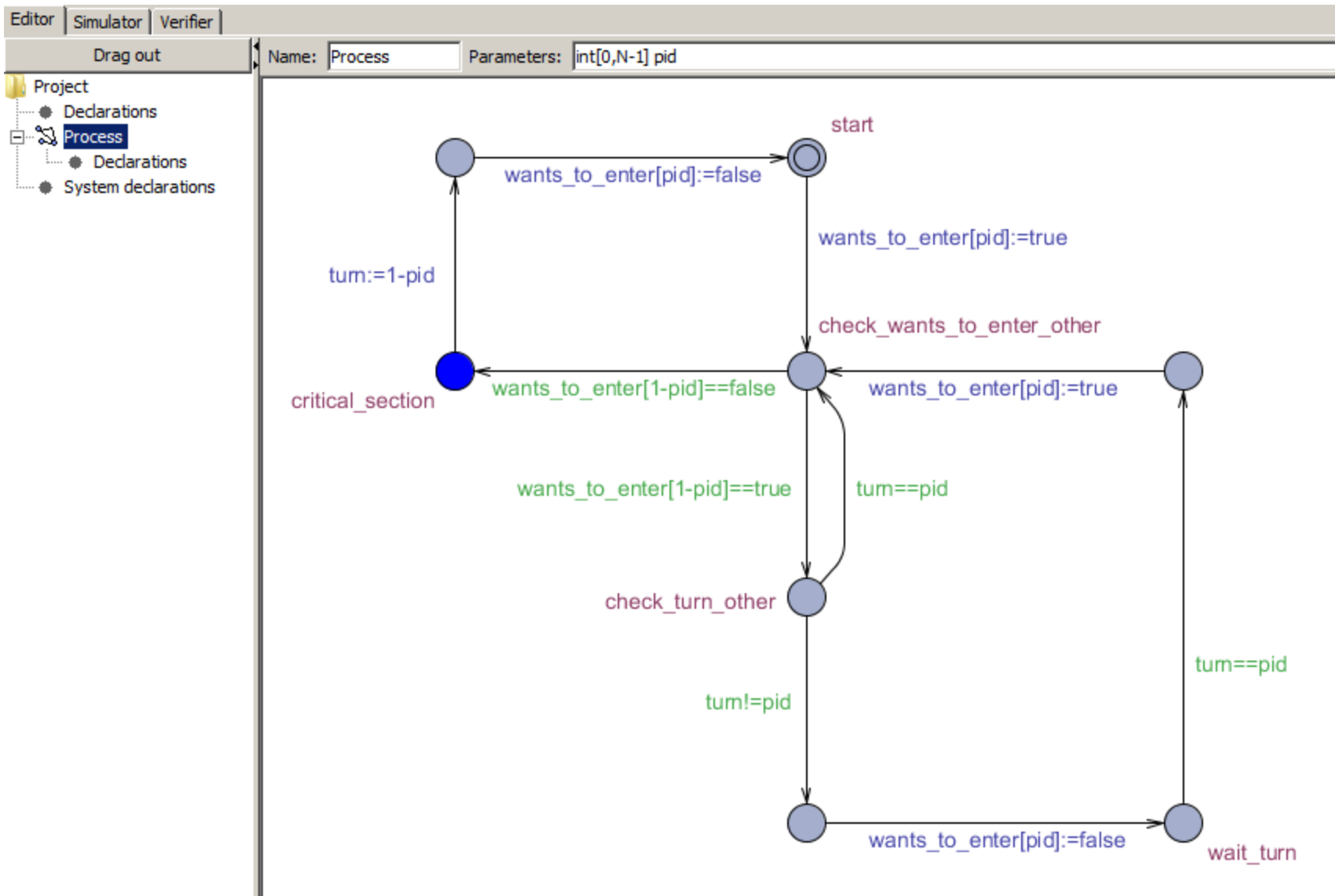
Algoritmus két processz közötti kölcsönös kizárásra megosztott változókkal:

- `bool wants_to_enter[0] = false;` // P0 processz akar-e belépni
- `bool wants_to_enter[1] = false;` // P1 processz akar-e belépni
- `int [0,1] turn = 0;` // P0 fog belépni (0 esetén) vagy P1 (1 esetén)

```
while (true) { P0
  wants_to_enter[0] := true
  while (wants_to_enter[1]) {
    if (turn != 0) {
      wants_to_enter[0] := false
      while (turn != 0) {
        // Busy wait
      }
      wants_to_enter[0] := true
    }
  }
  // Critical section
  turn := 1
  wants_to_enter[0] := false
  // Do other things
}
```

```
while (true) { P1
  wants_to_enter[1] := true
  while (wants_to_enter[0]) {
    if (turn != 1) {
      wants_to_enter[1] := false
      while (turn != 1) {
        // Busy wait
      }
      wants_to_enter[1] := true
    }
  }
  // Critical section
  turn := 0
  wants_to_enter[1] := false
  // Do other things
}
```


Dekker algoritmusának modellje (demo)



Dekker algoritmusának ellenőrzése (demo)

```
Editor | Simulator | Verifier |
Overview
E<> Process(0).critical_section
E<> Process(1).critical_section
A<> Process(0).critical_section
A<> Process(1).critical_section
A[] not (Process(0).critical_section and Process(1).critical_section)
A[] not deadlock
```

- **Teljesülő követelmények:**
 - Belépés lehetősége: $E \langle \rangle \text{Process}(i).\text{critical_section}$
 - Kölcsönös kizárás: $A[] \text{ not } (\text{Process}(0).\text{critical_section} \text{ and } \text{Process}(1).\text{critical_section})$
 - Holtpontmentesség: $A[] \text{ not deadlock}$
- **Nem teljesülő követelmények:**
 - Kiéheztetés-mentesség: $A \langle \rangle \text{Process}(i).\text{critical_section}$

Dekker algoritmus modelljének módosítása (demo)

- Ellenpéldák a kiéheztes-mentességhez
 - A processzek végtelen ideig várnak egy állapotban
 - Az egyik processz ciklikusan működik, míg a másik processz nem lép előre
- Modellezési lehetőségek az ellenpéldák elkerülésére
 - Ha nem fontos az időfüggő viselkedés:
 - Adott helyen való **várakozás korlátozása** Urgent helyekkel és
 - **ütemező** (scheduler) modellezése, ami garantálja a fair végrehajtást (egyszerű példa: szinkronizációval)
 - Ha fontos az időfüggő viselkedés:
 - Adott helyen **eltöltött idő korlátozása** hely invariánsokkal és
 - **késleltetés** („időtöltés”) a ciklusokban, ami esélyt ad a másik processz haladására, vagy **ütemező** (scheduler) modellezése, ami garantálja a fair végrehajtást (egyszerű példa: szinkronizációval)