
BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM
VILLAMOSMÉRNÖKI ÉS INFORMATIKAI KAR
MÉRÉSTECHNIKAI ÉS INFORMÁCIÓS RENDSZEREK TANSZÉK

# Aszinkron rendszerek modellellenőrzése párhuzamos technikákkal

SZAKDOLGOZAT

Készítette:
Jámbor Attila

Konzulensek:
Vörös András és Horváth Ákos

2010.

MŰEGYETEM 1782

# SZAKDOLGOZAT-FELADAT

**Jámbor Attila**
mérnök informatika szakos hallgató részére

**Aszinkron rendszerek modellellenőrzése párhuzamos technikákkal**

(A feladat szövege a mellékletben)

A szakdolgozat-feladatot összeállította és tanszéki konzulense:

Vörös András
doktorandusz

A záróvizsga tárgya: Objektumorientált szoftvertervezés: (bmeviii371)

A szakdolgozat-feladat kiadásának napja: 2010. szeptember 6.
A szakdolgozat beadásának határideje: 2010. december 10.

dr. Majzik István
egyetemi docens,
diplomaterv-felelős

dr. Horváth Gábor
egyetemi docens,
tanszékvezető

BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM
Méréstechnika és Információs Rendszerek Tanszék

A szakdolgozatot bevette:
A szakdolgozat beadásának dátuma:
A szakdolgozat bírálója:

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

# Aszinkron rendszerek modellellenőrzése párhuzamos technikákkal

Szakdolgozat-feladat melléklete

Napjainkban a beágyazott és ezen belül is az elosztott, aszinkron, misszió kritikus rendszerek tervezésénél egyre nagyobb szerepet kapnak a különböző formális módszereken alapuló technikák. Legnagyobb előnyük, hogy már a tervezés kezdeti fázisától képesek a rendszer helyes működésének a vizsgálatára és verifikálására.

Ezen a téren az egyik legszélesebb körben elterjed formális módszer a modellellenőrzés. A modellellenőrzés során a rendszer egy véges állapotterű modelljén vizsgáljuk a követelmények teljesülését. Ez azonban nagyon komplex, összetett feladat, amely megoldásához bonyolult algoritmusok szükségesek. Ilyen algoritmus a szimbolikus – azaz döntési diagramokon alapuló – szaturációs algoritmus, melyet kifejezetten nagy, elosztott, aszinkron rendszerek modellellenőrzésére fejlesztettek ki.

A Méréstechnika és Információs Rendszerek tanszéken fejlesztett *PetriDotNet* keretrendszer Petri hálók szerkesztésére, szimulációjára és analízisére szolgáló program, amely rendelkezik a szaturációs algoritmust megvalósító modullal. Ez azonban az algoritmust szekvenciálisan hajtja végre, nem használja ki a ma már elterjedt többprocesszoros gépek és többmagos processzorok jelentette többleterőforrásokat.

A hallgató feladata a szaturációs állapottérgeneráló és CTL modellellenőrző algoritmus vizsgálata, és párhuzamosításának kidolgozása, megvalósítása, ezáltal lehetővé téve a többszálú végrehajtást, növelve a verifikáció sebességét.

A hallgató feladatának a következőkre kell kiterjednie:

- Mutassa be a szaturációs állapottérgeneráló és modellellenőrző algoritmust
- Elemezze a szaturációs algoritmus párhuzamosításának lehetőségeit
- Implementálja a megtervezett CTL alapú modellellenőrző algoritmust a *PetriDotNet* keretrendszerbe
- Mérésekkel igazolja az elkészült modul hatékonyságát

Vörös András
doktorandusz

# HALLGATÓI NYILATKOZAT

Alulírott Jámbor Attila szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök, stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. A teljes szöveg közzététele dékáni engedéllyel titkosított diplomatervekre nem vonatkozik.

Budapest, 2010. december 9.

_____

Jámbor Attila

szigorló hallgató

# Tartalomjegyzék

## Kivonat

Napjainkban a biztonságkritikus rendszerekkel szemben támasztott fontos követelmény a tervezési helyesség magas fokú verifikálhatósága, azaz igazolhatósága. Erre gyakran formális módszereket használnak, amelyek közül a modellellenőrzés az egyik leginkább elterjedt. A modellellenőrzés folyamán kimerítő állapottér bejárással vizsgáljuk a rendszerrel szemben támasztott követelményeket. Aszinkron rendszerek esetén a formális viselkedést gyakran Petri-hálók segítségével, a rendszerrel szemben támasztott követelményeket pedig CTL temporális logikai kifejezésekkel fogalmazzák meg.

A formális verifikáció, és különösen a modellellenőrzés során gyakran probléma az állapottér drasztikus mérete. Ennek leküzdésére az irodalomban leginkább szimbolikus technikákat használnak. A legújabb kutatási eredmények azt mutatják, hogy aszinkron rendszerek esetén lehetőség van olyan fejlett algoritmusokat [1] használni, amelyek nagy hatékonysággal tudják tárolni, illetve bejárni az aszinkron rendszerek által kifeszített óriási állapottereket. Ezen algoritmusok hatékonyan felhasználhatóak a modellellenőrzés során is [2].

Ahogy egyre gyakoribbá válnak a többmagos processzorok, többprocesszoros gépek, jogos igényként merül fel, hogy a különböző szimbolikus technikák kihasználják ezeket a plusz erőforrásokat. A szimbolikus technikákon alapuló modellellenőrzést több munkában is próbálták párhuzamosítani [3], azonban a próbálkozások döntő többsége csak részsikereket ért el a felhasznált algoritmusok és adatstruktúrák szekvenciális jellemvonásai következtében. Munkám során célul tűztem ki, hogy egy már meglévő algoritmust hatékonyan megvalósítsak és további gyorsítási lépésekkel egészítsek ki. Ennek keretében a [3]-ban ismertetett párhuzamos szimbolikus állapottér bejáró algoritmushoz dolgoztam ki új zárolási mechanizmusokat és integráltam a Méréstechnika és Információs Rendszerek Tanszéken fejlesztett PetriDotNet [4] Petri háló modellező keretrendszerbe. Ennek eredményeképp nagyobb fokú párhuzamos futást tudtam elérni, amely további gyorsulásokhoz vezetett. A PetriDotNet keretrendszer tartalmaz továbbá egy, a CTL nyelven megfogalmazott specifikációs követelményeket ellenőrző modult. Munkám során ezt a modult is továbbfejlesztettem oly módon, hogy bizonyos logikai kifejezések kiértékelését párhuzamossá alakítottam.

Eredményeimet mérésekkel igazolom, amelyekben összehasonlítottam az általam megvalósított párhuzamos algoritmust annak szekvenciális változatával mind az állapottér-felderítés, mind a modellellenőrzés tekintetében.

## Abstract

Safety-critical systems require a high degree of assurance of design correctness. The verification of these systems is often based on formal methods, and especially on model checking, which is a widely used approach to investigate the behavior of discrete state models. Model checking examines the requirements of a system based on its mathematical representation using an exhaustive state-space exploration. Petri nets are often used as the high-level formalism to describe the behavior of asynchronous systems and requirements are expressed as CTL temporal logic predicates. Within my thesis work I focused only on system models and requirements captured as Petri nets and CTL predicates, respectively.

Unfortunately, most of the techniques are limited by the state explosion problem: as the complexity of a system increases, the memory and time required to store the combinatorially expanding state space can easily become excessive. To overcome this problem research in this area mainly focuses on symbolic techniques [1]. These approaches store the state space in compact data structures, such as decision diagrams. With this representation it became achievable to store state space over $10^{20}$ states [12].

Furthermore, the period of the model checking is as important as storage requirement. Nevertheless, most of today's hardware are built with multi-core processors so the need for parallel symbolic techniques is growing to utilize these additional resources. Several promising directions have been investigated to build such algorithms that work on shared memory architectures [3], but most of them ended with limited speed-ups due to the highly sequential manner of symbolic model checking algorithms.

The main aspects of my work is to apply novel synchronization techniques for the parallel symbolic model checking algorithm described in [3] and integrate it into the PetriDotNet [4] framework developed at the Department of Measurement and Information Systems. Besides this I turnd the CTL module of the PetriDotNet framework into parallel so I made a complete parallel model checking module. As part of my work, I measured and compared the different synchronization mechanisms, the parallel algorithm against its sequential implementation.

# 1 Bevezetés

Jelenünket az informatikai rendszerek egyre szélesebb körű terjedése határozza meg, így a velük szemben támasztott követelmények egyre inkább előtérbe kerülnek. A legtöbb rendszer esetében nem tolerálható a hibás működés ugyanis az sok pénzt, vagy ami még fontosabb, emberéleteket követelhet. Az úgynevezett missziókritikus rendszerek, mint például az atomerőművek vagy a légi forgalom irányító berendezései mellett napjainkban már a teljesen hétköznapi rendszerekkel szemben is megköveteljük az általuk nyújtott szolgáltatások biztonságát. Jó példa ezekre a telekommunikációs hálózatok, vagy az online banki rendszerek, amelyek már senki előtt sem ismeretlenek.

Az informatikai rendszerek hibás működése legtöbb esetben a tervezési hibákra vezethető vissza, ezért rendkívül fontossá vált a tervezés egyes fázisainak helyességét biztosítani és bizonyítani. A tervezési hibák elkerülhetők validáció, verifikáció vagy a kettő kombinációjának alkalmazásával [5][6][7]. Verifikáció során többféle metodika közül választhatunk: használhatjuk a tesztelést, a szimulációt, a modellellenőrzést vagy a tételbizonyítást. A felsorolt technikák közül azonban csak a modellellenőrzés és a tételbizonyítás biztosít kimerítő, azaz teljes körű vizsgálatot, ugyanis a tesztelés és a szimuláció során a hibák többsége kiszűrhető, de a hibamentesség nem garantálható. A tételbizonyítás [8] alkalmazhatóságát korlátozza, hogy használatához magasan képzett emberekre van szükség, akik gyakran nem állnak rendelkezésre. Ezzel szemben a modellellenőrzés automatikus ellenőrzést tesz lehetővé [8], így a gyakorlatban is hatékonyan használható.

Mindazonáltal a modellellenőrzés is rendelkezik gyenge pontokkal, név szerint a jelentős idő- és tárigénnyel. A tárigény csökkentésére már léteznek jól használható megoldások, az úgynevezett szimbolikus technikák, amelyek a rendszer állapotait nem egyenként, explicit módon tárolják, hanem kódolt formában. Ilyen szimbolikus technika a szaturáció is, amely döntési diagramokat használ a rendszer állapotainak kódolására [1].

A manapság használatos számítógépek döntő többsége több processzormaggal szerelt, amelyek révén képesek párhuzamos feldolgozásra. Ezt kihasználva szakdolgozatom keretein belül a szaturációs algoritmus párhuzamosíthatóságát vizsgáltam a rendelkezésre álló erőforrásokkal való hatékonyabb gazdálkodás érdekében. Célom az volt, hogy a modellellenőrzés tár- és időigényét kezelhető méretűvé csökkentsem. Az előbbit a szimbolikus szaturációs algoritmus alkalmazásával, utóbbit pedig a párhuzamosítás révén kívántam megvalósítani.

Szakdolgozatom keretein belül az alábbi célokat tűztem ki magam elé:

- A modellellenőrzés alapjainak megismerése.
- A szaturációs állapottér-generáló és modellellenőrző algoritmus és alkalmazásának megismerése.
- A párhuzamos szaturációs algoritmus implementációjának elkészítése.
- Az elkészített algoritmus optimalizációja és továbbfejlesztése hatékonyabb erőforrás-kihasználás érdekében.
- A modellellenőrzés párhuzamosítása.

Jelen dolgozatom a következő részekre tagolható:

- A 2. fejezet áttekintést nyújt a dolgozat és a fejlesztési folyamat megértéséhez szükséges előismeretekről, úgymint a modellellenőrzésről, a Petri-hálókról, a döntési diagramokról és a CTL temporális logikáról.
- A 3. fejezet részletesen bemutatja a munkám alapját képező szaturációt. Az algoritmus bemutatása mellett kitérek arra is, hogyan alkalmazható a szaturáció az állapottér-felderítés és a modellellenőrzés során.
- A 4. fejezet a korábbi részek ismereteit felhasználva mutatja be a szaturáció párhuzamosításának mikéntjét. Ebben a fejezetben számolok be a megvalósított kutatási eredményekről, megvalósított fejlesztésekről is.
- Az 5. fejezet szolgál az eredményeim alátámasztására. A kitűzött cél elérése után méréseket végeztem az elkészített modul hatékonyságának vizsgálatára. Ezen mérések eredményei kerülnek bemutatásra ebben a fejezetben.
- Végezetül a 6., záró fejezetben értékelem munkámat és további fejlesztési irányokat is megjelölök kutatásom lehetséges folytatásaként.

# 2 Háttérismeretek

A most következő fejezet a dolgozat olvasása kapcsán fontos előismereteket rendszerezi. A 2.1 fejezet rövid áttekintést nyújt a megcélzott, aszinkron rendszerekről. Ezt követően a 2.3 és 2.4 fejezetek bemutatják a rendszerek leírására, modellezésére alkalmas Petri-hálókat, illetve a rendszerek állapotterének tárolását szolgáló döntési diagramokat. A modellellenőrzésről általánosságban a 2.2 fejezet szól, míg a 2.6 fejezetben bemutatásra kerül a PetriDotNet keretrendszer, amelybe az elkészített munkámat integráltam.

## 2.1 Aszinkron rendszerek

Dolgozatomban aszinkron rendszerek modellellenőrzését tűztem ki célul, ugyanis ezen rendszerek igen fontos szerepet játszanak a környezetünkben. Aszinkronnak nevezünk egy olyan elosztott rendszert, amely több, egymástól független komponensből épül fel. Mivel a rendszer egyes részei függetlenek egymástól, ezért a rendszer egészét tekintve a működés nem determinisztikus. Mindazonáltal fontos, hogy az egyes részek közös cél elérésére törekednek, aminek elérése érdekében együttműködnek. Az együttműködés minden esetben egy meghatározott protokoll szerinti kommunikáció, ami biztosítja az egyes részek közötti szinkronizációt és adatátvitelt.

Az aszinkron rendszerek egyik tipikus változata az elosztott, beágyazott rendszerek. Ezekre példaként említhetők az autókban megtalálható elektronikák vagy a szenzorhálózatok, amelyek jól tükrözik az aszinkron rendszerek elterjedtségét.

## 2.2 Modellellenőrzés általában

Ha az informatikai rendszerek tervezése során biztosítani szeretnénk a majdani szolgáltatások helyességét, akkor kétféle ellenőrzés áll rendelkezésünkre, a validáció és a verifikáció.

- A *validáció* azt mondja meg számunkra, hogy a rendszerünk megfelel-e a felhasználó követelményeinek, azaz a létrehozandó rendszer teljesíti-e a vele szemben támasztott specifikációt.
- Ezzel szemben a *verifikáció* során arra keressük a választ, hogy valóban a specifikációnak megfelelő rendszert építettük, azaz az implementáció egyes fázisai végén a rendszerről alkotott modell ekvivalens-e a specifikációs modellel.

Talán jobban érzékelhető a kétféle ellenőrzés közötti különbség, ha abból a szemszögből vizsgáljuk őket, milyen kérdésre is adnak választ. A validáció során feltehetjük azt a

kérdést, hogy "Jó rendszert építünk-e?", míg a verifikáció során arra kapunk választ, hogy "Jól építjük-e a rendszert?".

A dolgozatom tárgyát képező *modellellenőrzés* egy formális módszereken alapuló verifikációs technika. A modellellenőrzés egy véges állapotterű rendszer esetén képes ellenőrizni a kívánt tulajdonságok teljesülését. A technika alkalmazása napjainkban már széles körben elterjedt, ugyanis segítségével automatizálható a verifikáció.

Más verifikációs technikákhoz hasonlóan a modellellenőrzés során is problémát jelent az ipari rendszerek óriási, esetleg végtelen mérete, amely az állapottér tárolásának és a verifikáció idejének kezelhetetlenségét vonja maga után. Ezt úgy szokták orvosolni, hogy a verifikáció szempontjából kevésbé fontos részek elhagyásával, a rendszer egyszerűsítésével a rendszerről alkotott modell méretét csökkentik. A valóságban léteznek ugyan végtelen méretű állapottérrel rendelkező modellek is, azonban ezekkel a dolgozat keretein belül nem foglalkozom.

A modellellenőrzésen belül több megközelítés is elterjedt, amelyek az állapottárolást tekintve 2 csoportba sorolhatóak:

- Az *explicit technikák* a rendszer állapotait és az állapotátmeneti információkat egyenként, explicit módon tárolják el, amelyet szélességi vagy mélységi bejárás során derítenek fel.
- Ezzel szemben a *szimbolikus technikák* implicit módon tárolják a rendszer állapotait [1][3][8]. Ehhez valamilyen fejlett, tárterület szempontjából hatékonyabb adatstruktúrát használnak, például döntési diagramokat [9][10], hash táblát.

A modellellenőrzés alkalmazása során a rendszerrel szemben támasztott követelmények többféle, különböző kifejezőerejű formalizmussal is megadhatóak, azonban ahhoz, hogy ezeket megismerjük, még további ismeretekre van szükségünk, így ezeket a 2.5 fejezet fogja részletesebben kifejteni.

## 2.3 Petri-hálók

A Petri-hálók alapjait Carl Adam Petri, német matematikus fektette le, amelyet eredetileg kémiai folyamatok modellezésére szánt [11]. A Petri-hálók azonban olyan jól használhatók egyéb folyamatok, rendszerek modellezésére is, hogy napjainkban többek között a modellellenőrzés kedvelt rendszerleíró eszközévé váltak.

A most következő fejezetek a Petri-hálók bemutatását célozzák azon aspektusból, amely a rendszermodellezés és -analízis során fontos lehet.

### 2.3.1 Petri-hálók általában

A Petri-háló konkurens, aszinkron, elosztott rendszerek modellezésére alkalmas magas-szintű formalizmus. Más formális eszközökkel szemben a Petri-hálók a matematikai reprezentáción túl grafikus megjeleníthetőséget is nyújtanak. Ennek köszönhetően a precíz matematikai pontosság mellett a könnyű kezelhetőség és átláthatóság tulajdonságával is bírnak. További előnyük más modellezési nyelvekkel szemben, hogy strukturális tulajdonságok vizsgálatára is lehetőséget nyújtanak.

A Petri-hálóknak széles körű használatuk következtében többféle változata is használatos annak függvényében, hogy a vizsgált rendszer mekkora kifejezőerejű modellezési nyelvvel írható le. Én csak a legegyszerűbb esetet vizsgálom, ugyanis már annak kifejező ereje is elegendő az általam vizsgált aszinkron rendszerek modellezésére.

A Petri-hálók statikus és dinamikus tulajdonságokkal is rendelkeznek, ezekről bővebben a 2.3.2 és 2.3.3 fejezetekben lesz szó.

### 2.3.2 Struktúra, definíció

A Petri-hálók struktúrája adja meg az általa leírt modell statikus komponenseit és azok kapcsolatát.

*Struktúráját* tekintve a Petri-háló egy irányított, súlyozott, páros gráf. A két pontosztálynak a háló helyei és tranzíciói felelnek meg. A élek kötik össze, amelyek vezethetnek helyből tranzícióba vagy fordítva. Az élek élsúlyokkal rendelkeznek, amelyek pozitív egész számok. Az egyes helyek állapotát a helyeken található tokenek határozzák meg, míg a rendszer globális állapotának az egyes helyek állapotainak összessége felel meg.

Ezeknek megfelelően a Petri-háló egy olyan $PN = (P, T, E, W)$ struktúrának felel meg, ahol:

- $P = \{p_1, p_2, \dots, p_n\}$a helyek véges számú halmaza,
- $T = \{t_1, t_2, \dots, t_m\}$ a tranzíciók véges számú halmaza,
- $E \subseteq (P \times T) \cup (T \times P)$ az élek véges számú halmaza,
- $W: E \to \mathbb{Z}^+$ az élek súlyfüggvénye.

A Petri-háló állapotát a *tokeneloszlás* határozza meg, amely leírható az $M: P \to \mathbb{N}$ függvénnyel. Ez alapján az aktuális tokeneloszlás minden esetben egy $M = [m_1, m_2, \dots, m_n]^T$ oszlopvektorral adható meg, ahol a vektor komponenseinek száma megegyezik a hálóban található helyek számával. Így a Petri-háló megadható egy $PN = (P, T, E, W, M_0)$

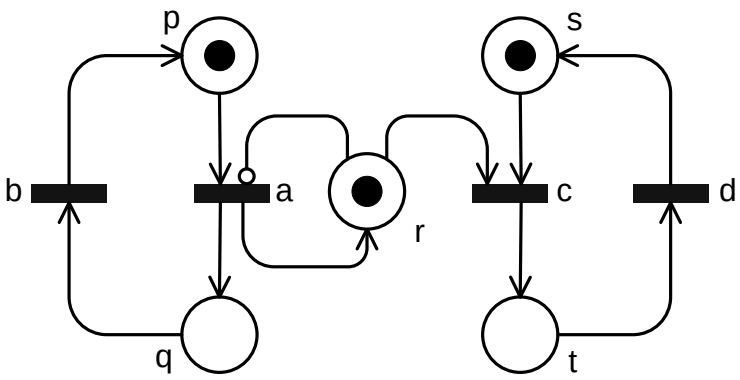
struktúrával is, ahol $M_0$ a kezdeti tokeneloszlásnak felel meg. A kétféle struktúra alak onnan ered, hogy a kezdeti tokeneloszlás nem befolyásolja a háló struktúra tulajdonságait, így ezt gyakorta elhagyják.

Az élek mentén haladva meghatározhatóak a $n \in (P \cup T)$ csomópont ősei és utódai, amelyek a következőképpen definiálhatóak:

- egy $p \in P$ hely ősei a bemenő tranzíciói: $\bullet p = \{t|(t,p) \in E\}$
- egy $p \in P$ hely utódai a kimenő tranzíciói: $p\bullet = \{t|(p,t) \in E\}$
- egy $t \in T$ tranzíció ősei a bemenő helyei: $\bullet t = \{p|(p,t) \in E\}$
- egy $t \in T$ tranzíció utódai a kimenő helyei: $t\bullet = \{p|(t,p) \in E\}$

A Petri-hálók az említettek alapján grafikusan is megjeleníthetők. Ezen reprezentációban a helyeknek körök, a tranzícióknak pedig téglalapok felelnek meg. Az egyes tokeneket a Petri-háló helyeken ponttal jelölhetjük.

A Petri-hálóra és annak grafikus reprezentációjára az 1. ábra mutat példát.




**1. ábra Petri-háló - példa**

Az 1. ábra Petri-hálóján a helyeknek $P = \{p, q, r, s, t\}$, míg a tranzícióknak $T = \{a, b, c, d\}$ halmazok felelnek meg. A háló jelenlegi állapotában a p, r és s helyek tartalmaznak tokent, így a háló állapota leírható a következőképpen: $m: [p, q, r, s, t] = [1,0,1,1,0]$. Az r hely és a tranzíció között látható egy, a többitől eltérő jelölésű él. Ezt tiltó élnek nevezik, amely a következő fejezetben kerül bemutatásra.

### 2.3.3 Dinamikus viselkedés

A Petri-háló dinamikus viselkedése az állapotok változását jelenti. A háló állapotváltozása minden esetben egy esemény tüzelésének következménye.

A *tüzelések* az alábbi szabályoknak megfelelően történhetnek:

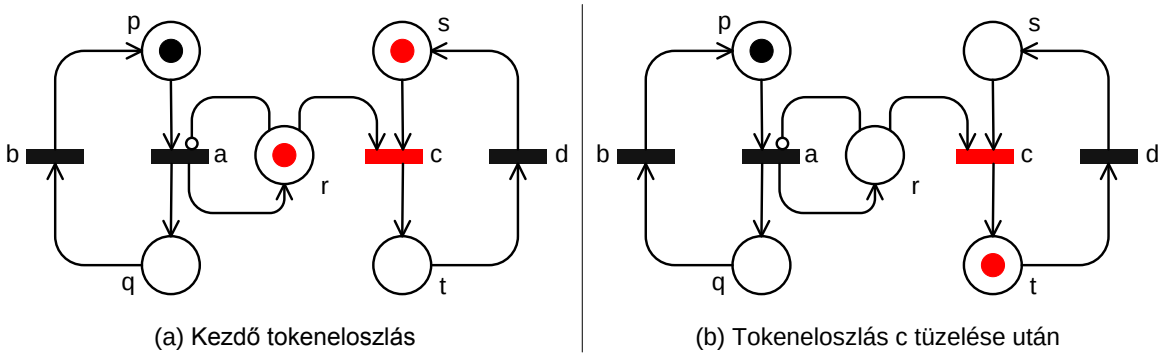
- Egy $t \in T$ tranzíció engedélyezett, ha t-nek minden egyes $p \in \bullet t$ bemenő helyén legalább $w^-(p,t)$ token van, ahol $w^-(p,t) = W(p,t)$, azaz a $(p,t)$ él súlya.
- Csak engedélyezett tranzíció tüzelhet.
- A működés nem determinisztikus, ugyanis előre nem meghatározható, hogy a következő időpillantban melyik engedélyezett tranzíció fog tüzelni. Az is előfordulhat, hogy egy tranzíció sem tüzel.
- Egy engedélyezett t tranzíció tüzelése $w^-(p,t)$ darab tokent vesz el a t minden egyes $p \in \bullet t$ bemenő helyéről és $w^+(p,t)$ darab tokent helyez el a t tranzíció minden egyes $p \in t\bullet$ kimenő helyére, ahol $w^+(p,t) = W(t,p)$, azaz a (t,p) él súlya.

Abban az esetben, ha a vizsgált Petri-háló *tiltó él*e(ke)t is tartalmaz, akkor a tüzelésre vonatkozó szabályok a következővel egészülnek ki:

- Egy $t \in T$ tranzíció nem lehet engedélyezett, ha t-nek létezik legalább egy olyan $p \in \bullet t$ bemenő helye, hogy azon legalább $w_i(p,t)$ token van, ahol $w_i(p,t)$ a $(p,t)$ tiltó él súlya.

Egy tranzíció tüzelése után a Petri-háló új állapotba kerül, amely egy új tokeneloszlással írható le.

A tranzíciók tüzelésére a 2. ábra mutat példát.

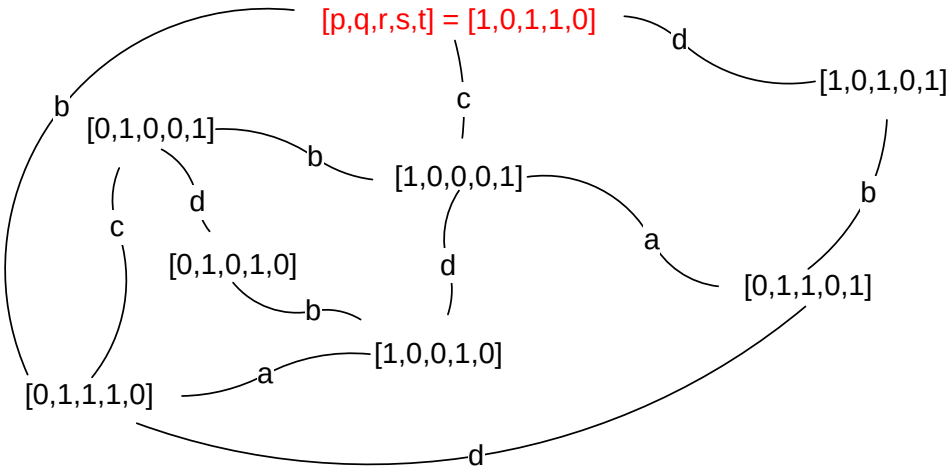



**2. ábra Tranzíció tüzelés - példa**

A 2. ábra Petri-hálója a következő állapotban van az (a) ábrán: $m: [p,q,r,s,t] = [1,0,1,1,0]$. Ebben az állapotban csak a *c* tranzíció engedélyezett, hiszen a *b* és *d* tranzíciók bemeneti helyein nincsen elegendő számú token az engedélyezettséghez, míg az *a* tranzíciót az (r, a) tiltó él nem engedélyezi. A *c* tranzíció tüzelésének hatására a bemeneti helyekről, azaz *r* és *s* helyről elvesszük a tokeneket, majd *c* kimeneti helyére, azaz a *t* helyre generálunk egy

tokent. A tranzíció tüzelésének hatására a Petri-háló új állapotba kerül, amely a következőképpen írható le: m′: [p, q, r, s, t] = [1,0,0,0,1]. Ezt az új tokeneloszlást mutatja a (b) ábra. Érdemes még megfigyelni, hogy a kialakult új állapotban a *d* és az *a* tranzíciók lesznek engedélyezettek.

Az 1. ábrán látható Petri-háló által kifeszített állapotteret a 3. ábra szemlélteti.




**3. ábra Állapottér – példa**

A 3. ábrán egy állapot-átmeneti gráf látható. A csomópontoknak az egyes állapotok felelnek meg, belőlük pedig az adott állapotokban engedélyezett tranzícióknak megfelelő élek vezetnek ki. Az élek végpontja abba az állapotba mutat, amelybe a rendszer az esemény tüzelése után kerül. Az ábrának megfelelően az 1. ábra Petri-hálója egy 9 állapotú teret feszít ki. Az ábrán segítségképpen pirossal jelöltem a kezdő állapotot.

## 2.4 Döntési diagramok

A modellellenőrzés egyik kulcsfontosságú lépése a rendszer állapotterének felderítése és tárolása, azonban a nagyméretű állapothalmazok kezelése explicit technikák segítségével bonyolult és költséges. Ennek elkerülése végett hatékony az állapotok explicit helyett kódolt tárolása [8]. Az ilyen kódolt tárolás megvalósítható karakterisztikus függvények használatával [9][10]. Az így kódolt állapothalmazon a halmazműveleteket is karakterisztikus műveletek segítségével értelmezzük. A *karakterisztikus függvény* egy olyan állapotváltozókon értelmezett logikai függvény, amely akkor és csak akkor igaz, ha az állapotváltozók behelyettesítési értékének megfelelő állapot része a karakterisztikus függvény által kódolt állapothalmaznak. A kódoláshoz szükséges állapotváltozók számát a kódolandó állapotok mennyisége határozza meg.

### 2.4.1 Bináris döntési diagramok

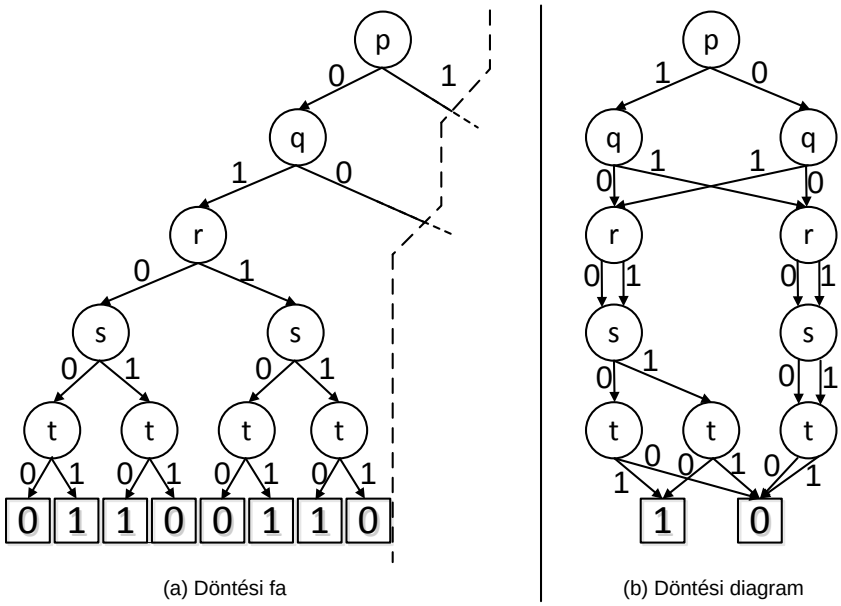
A modellellenőrzés során a karakterisztikus függvénnyel kódolt állapottér tárolására a bináris döntési diagramok (binary decision diagram, BDD) hatékonyan alkalmazhatók [5][8]. A döntési diagramok megismeréséhez azonban először be kell vezetni a döntési fákat.

A *döntési fák* teljes, bináris fák kétféle csomóponttal. A legalsó szinten terminális, míg a magasabban levő szinteken nemterminális csomópontok foglalnak helyet. Egy $v$ csomópont szintjét a level($v$) $\in \mathbb{Z}^+$ függvény határozza meg. A terminális csomópontok a 0 vagy 1 értéket vehetik fel, míg a nemterminális csomópontok egy-egy logikai állapotváltozónak felelnek meg. Minden nemterminális csomópontnak pontosan 2 éle van. A v csomópont élei a logikai igaz és hamis értékekkel vannak címezve, és egy level($v$)-1 szinten levő csomópontba mutatnak.

Döntési fák alkalmazásakor n különböző állapot tárolásához $\lceil \log_2 n \rceil$ darab állapotváltozóra van szükség. Ennek megfelelően a fa magassága is $\lceil \log_2 n \rceil$ lesz. Döntési fák segítségével egy állapotról a következőképpen dönthető el, hogy bele tartozik-e a kódolt állapothalmazba: A döntési fa gyökér csomópontjából kiindulva az aktuális csomópont állapotváltozójának megfelelő értéket behelyettesítve haladunk az élek mentén lefelé a fában, mígnem egy terminális csomóponthoz érünk. Ha az elért terminális csomópont értéke 1, akkor a vizsgált állapot része a karakterisztikus függvény által reprezentált állapothalmaznak; különben pedig nem része.

Abban az esetben, ha egy döntési fában az azonos csomópontokat illetve részfákat összevonjuk, akkor egy olyan döntési diagramot kapunk, amely a döntési fával megegyező állapothalmazt kódol. (Két csomópont akkor tekinthető azonosnak, ha ugyan azon a szinten vannak, és éleik ugyan azon csomópontokba mutatnak.) A döntési diagramok megfelelnek egy körmentes, irányított, két levéllel rendelkező gráfnak. Egy állapotról ellenőrizni, hogy a tárolt állapothalmazba tartozik-e hasonló módon történik, mint a döntési fáknál. Mivel az állapotváltozók itt is 2 értéket vehetnek fel, ezért egy $n$ állapotot tartalmazó halmaz kódolásához $\lceil \log_2 n \rceil$ db állapotváltozóra van szükség, azaz ilyen magas lesz a döntési diagram.

A döntési fákra és döntési diagramokra a 4. ábra mutat példát.




**4. ábra Döntési fa és döntési diagram - példa**

Mind a 4. ábra (a) részén látható döntési fa, mind a (b) részén látható döntési diagram az 1. ábra Petri-hálójának állapotterét kódolja. Mivel a Petri-háló állapottere olyan, hogy minden lehetséges állapotban legfeljebb 1 token található egy helyen, ezért a Petri-hálóban található helyek állapotának kódolására elegendő 1-1 állapotváltozó mind a döntési fában, mind a döntési diagramban. A változó értéke 1, ha az adott állapotban van token a helyen, egyébként pedig 0. Az éleken szereplő számok megfelelnek az állapotváltozók lehetséges értékeinek. A gyökér csomópontból indulva 0 és 1 élek mentén jutunk el a terminális csomópontokig. A példa állapottere 8 lehetséges állapotot tartalmaz. Ez úgy látható a döntési fában, hogy a legalsó szinten 8 terminális 1 csomópont található, míg a döntési diagramon 8 különböző út vezet az egyetlen terminális 1 csomópontba. Jól látszik az ábrán, hogy a döntési diagram sokkal kompaktabb ábrázolást nyújt, hiszen a döntési fa akkora, hogy ábrámon annak csak negyedét tudtam ábrázolni.

### 2.4.2 Többértékű döntési diagramok

A többértékű döntési diagramok (multi-valued decision diagram, MDD) a bináris döntési diagramok kiterjesztéseinek tekinthetők [5].

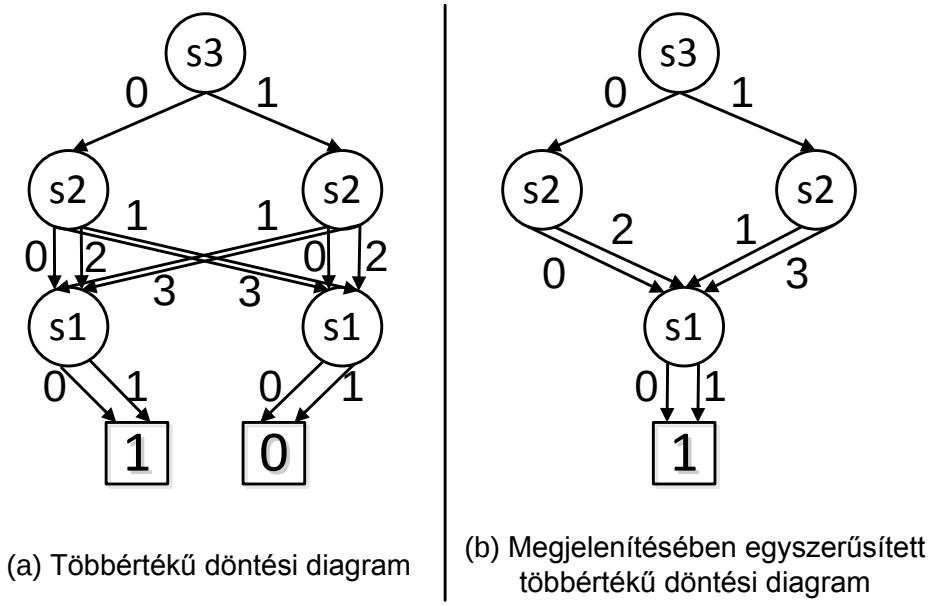
A többértékű döntési diagramok is egy körmentes, irányított gráfnak tekinthetők, azonban a bináris döntési diagramokkal szemben az MDD csomópontjaiban levő állapotváltozók nem binárisak, hanem egy $D = \{d_0, d_1, \dots, d_{|D|-1}\}$ véges halmazból vehetnek fel értékeket. A könnyebb kezelhetőség érdekében a következő egyszerűsítéssel élek: $D = \{0, 1, \dots, |D| - 1\}$. Mivel a kétféle jelölés közötti megfeleltetés egyértelmű, ezért ez az általánosságot nem

csökkenti. Más tekintetben az MDD-k teljesen hasonlatosak a BDD-khez. Szintén terminális és nemterminális csomópontokat tartalmaznak. A terminális csomópontok itt is a 0 és 1 értéket vehetik fel. A nemterminális *v* csomópontnak itt viszont nem 2, hanem $\left|D_{level(v)}\right|$ db éle van. Az élek szintén egy level(*v*)-1 szinten levő csomópontba mutatnak.

Az egyszerűbb hivatkozás érdekében bizonyos jelöléseket célszerű bevezetni. A *v* csomópontot jelöljük ⟨k|p⟩-vel, ahol k=level(*v*), p pedig a szinten belül egyedi azonosító. Ennek megfelelően a terminális csomópontot a ⟨0|0⟩, míg a terminális 1 csomópontot a ⟨0|1⟩ jelöli. A ⟨k|p⟩ csomópont i. éle által mutatott csomópontot pedig mostantól jelöljük ⟨k|p⟩[i]-vel.

Az MDD-k grafikus ábrázolásukat tekintve is hasonlóak a BDD-khez, az éleket itt is az állapotváltozó megfelelő értékével címkézzük. A különbség csupán annyi, hogy a jobb áttekinthetőség érdekében csak azokat az éleket és csomópontokat szokás feltüntetni, amelyek a gyökér és a terminális 1 csomópont közti utak valamelyikén találhatók, a terminális nullába vezető utakat nem.

A többértékű döntési diagramokra és azok egyszerűsített megjelenítésére az 5. ábra mutat példát.




**5. ábra Többértékű döntési diagram - példa**

A 5. ábra újra az 1. ábra Petri-hálójának állapotterét kódolja. Az (a) ábrán látható a teljes MDD, míg a (b) ábrán nem tüntettem fel a terminális 0 csomópontba vezető utakat és magát a terminális 0 csomópontot. Az *s1* állapotváltozó a *p* hely állapotát, az *s2*

állapotváltozó a *q* és *s* hely állapotát együttesen, az *s3* állapotváltozó pedig az *r* és *t* helyek állapotát együttesen kódolja. Az *s2* állapotváltozót jelentő csomópontoknak azért van 4 éle, mivel a *q* és *r* helyek állapotainak 4 különböző kombinációja fordulhat elő a teljes állapottérben. Ezek rendre [*q*,*r*]=[0,0], [*q*,*r*]=[1,1], [*q*,*r*]=[0,1] és [*q*,*r*]=[1,0]. A rendszer egy elérhető globális állapota itt is a gyökértől a terminális 1 csomópontig vezető úton található lokális állapotok összessége. Megfigyelhető, hogy a többértékű döntési diagramok még a BDD-knél is tömörebb ábrázolást és tárolást nyújthatnak; azonban ez nagyban függ attól, hogy az egyes állapotváltozókat miként alakítjuk ki a Petri-háló helyeiből.

### 2.4.3 Műveletek csomópontokon

A MDD-ken végezhető műveletek gyakorlatilag megfelelnek az MDD által kódolt állapothalmazon végzett művelteknek.

Egy MDD-ben p és q csomópont *uniója* a következő, ha $\text{level}(p) = \text{level}(q)$:

$$p \cup q = \begin{cases} p \vee q & \text{ha } (\text{level}(p) = \text{level}(q) = 0 \\ r & \text{különben, ahol } r[i] = p[i] \cup q[i] \text{ minden } i - \text{re} \end{cases}$$

Egy MDD-ben p és q csomópont *metszete* a következő, ha $\text{level}(p) = \text{level}(q)$:

$$p \cap q = \begin{cases} p \wedge q & \text{ha } (\text{level}(p) = \text{level}(q) = 0 \\ r & \text{különben, ahol } r[i] = p[i] \cap q[i] \text{ minden } i - \text{re} \end{cases}$$

Két MDD uniója/metszete alatt a gyökércsomópontok unióját/metszetét értjük [13].

A döntési diagramokban lehetőség adódik *útvonal*ak megadására is. A $\langle k|p \rangle$ csomópontból indulva egy l. szinten levő csomópont megadható egy $(i_k, i_{k-1}, \dots, i_{l+1}) \in D_k \times D_{k-1} \times \dots \times D_{l+1}$ sorozattal, ahol $K \geq k > l + 1 > 1$ és K a döntési diagram szintjeinek a száma. Az így megadott útvonal jelentése a következő: $\langle k|p \rangle[i_k, i_{k-1}, \dots, i_{l+1}] = \langle k - 1|\langle k|p \rangle[i_k]\rangle[i_{k-1}, .., i_{l+1}]$.

A $\langle k|p \rangle$ csomópont által tárolt útvonalakt összességében a következőképpen adhatjuk meg: $\mathcal{B}(\langle k|p \rangle) = \{\beta \in D_k \times D_{k-1} \times \dots \times D_1 : \langle k|p \rangle[\beta] = 1\}$

## 2.5 CTL temporális kijelentéslogika

A 2.2 fejezetben bevezetett modellellenőrzés során a rendszerrel szemben támasztott követelmények teljesülését vizsgálom [13]. Ahhoz, hogy a követelményeinket meg tudjuk fogalmazni, temporális logikát szokás használni. A CTL (Computational Tree Logic, elágazó

idejű temporális logika) kellően egyszerű, azonban kifejező ereje mégis megfelelő, így széles körben elterjedt.

A CTL alkalmas kijelentések igazságának logikai időbeli változásának vizsgálatára [13], ahol a logikai idő szemantikája elágazó, az időpillanatok fa-struktúrában elágazó idővonalak mentén követik egymást. A fa struktúrában minden állapotnak legfeljebb egy utódja, rákövetkező állapota van. (Ha nincs rákövetkező állapot, akkor a rendszer holtpontra jutott.)

A CTL kifejezéseket temporális operátorok segítségével fogalmazhatjuk meg, amelyeknek 2 csoportja létezik, útvonalkvantorok és állapotkvantorok.

A CTL kifejezések *útvonalkvantorai* a következők:

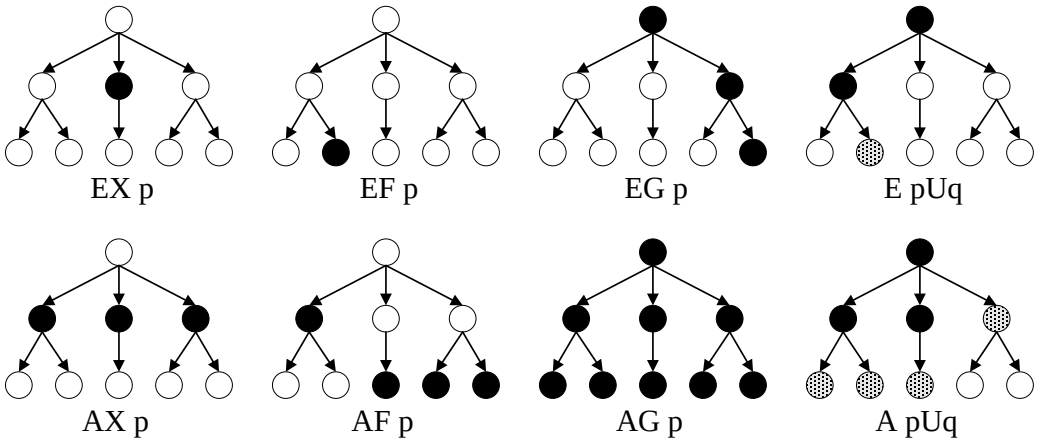
- A: minden lehetséges útra az adott állapotból indulva
- E: legalább egy útra az adott állapotból indulva

A CTL kifejezések *állapotkvantorai* a következők:

- G p: az adott útvonal összes állapotán igaz lesz p
- F p: az adott útvonal legalább 1 állapotán igaz lesz p
- X p: az adott útvonal következő állapotán igaz lesz p
- p U q: az adott útvonal p mindaddig igaz, amíg q igazzá nem válik

A CTL kifejezések mindig egy útvonalkvantorból és egy állapotkvantorból tevődnek össze, és minden esetben az aktuális állapottól értelmezzük őket. A definíciókban a p és q betűk állapotkifejezéseket jelölnek.

A bemutatott útvonal- és állapotkvantorok kombinációjaként 8-féle CTL kifejezést tudunk létrehozni. Ezek definíciója helyett jelentésüket a 6. ábra adja meg.




**6. ábra CTL operátorok szemléletes jelentése**

A 6. ábrán a lehetséges CTL kifejezéseket szemléltetem. Az ábrán a körök állapotokat jelölnek, közöttük a nyilak pedig lehetséges állapotátmeneteket. A fekete színű csomópontokban a p kifejezés teljesül, a halványabb színű csomópontokban a q állapotkifejezés teljesül, míg a fehér színű csomópontok igazságtartalma nem lényeges.

Az 1. ábra Petri-hálójára példaként megfogalmazhatjuk a következő kifejezéseket:

- A termelő és fogyasztó közötti bufferbe nem teszünk sosem egynél több tokent: AG(r<2), ahol *r* a Petri-háló *r* jelű helyének tokenszámát jelöli.
- A termelő komponens mindig visszajuthat a kiinduló állapotba: AG(EF p=1).

## 2.6 PetriDotNet keretrendszer

A PetriDotNet egy modellellenőrző keretrendszer. A modellezendő rendszert Petri-hálók segítségével adhatjuk meg. A PetriDotNet képes a Petri-hálók szerkesztésére, szimulálására és ellenőrzésére. Az eszköz fejlesztése a Budapesti Műszaki és Gazdaságtudományi Egyetem Méréstechnikai és Információs Rendszerek Tanszékén folyik .Net alapokon.

Az eszköz jellegzetessége, hogy moduláris felépítésű, így lehetőség nyílik a PetriDotNet funkcióinak kibővítése saját fejlesztésű bővítményekkel. Ezt a tulajdonságot kihasználva fejlesztettem a párhuzamos modellellenőrzőmet és integráltam a rendszerbe.

A PetriDotNet keretrendszerről az eszköz honlapján [4] lehet bővebb információt találni. Itt nyílik lehetőség az eszköz letöltésére és kipróbálására is.

# 3 Szaturáció

A szaturáció egy szimbolikus technika aszinkron rendszerek állapotterének felderítésére és modellellenőrzésére [1][14][15][17]. Alkalmazásával elkerülhető az ismert állapottér robbanás problémája, ugyanis a tradicionális megoldásokhoz képest jelentősen növeli az idő- és tárhatékonyságot.

A 3.1 fejezet áttekintést nyújt az algoritmus megismeréséhez szükséges további ismeretekről. Az algoritmusnak 2 nagyobb felhasználási területe van, az állapottér-felderítés és a modellellenőrzés. Ennek megfelelően a 3.2 és a 3.3 fejezetekben mutatom be a konkrét szaturációs algoritmust.

## 3.1 Bevezetés

A szaturáció során az aszinkron rendszereket Petri-hálójukkal adhatjuk meg, amelyek formálisan a következő hármassal írhatók le:

$(\hat{\mathcal{S}}, \mathrm{s}, \mathcal{N})$, ahol:

- $\hat{\mathcal{S}}$ a modell lehetséges állapotainak halmaza
- $\mathrm{s} \in \hat{\mathcal{S}}$ a kezdeti állapot
- $\mathcal{N}: \hat{\mathcal{S}} \to 2^{\hat{\mathcal{S}}}$ pedig az úgynevezett következő-állapot (Next-State) függvény. A következő-állapot függvény megadja az egyes állapotokra, hogy belőlük egy lépésben milyen állapotok érhetők el.

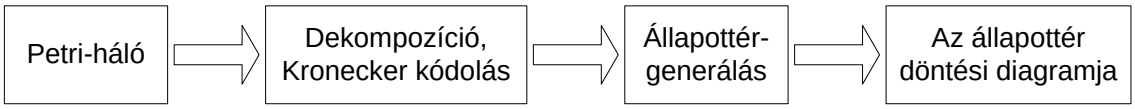
A modellben az állapotváltozások úgynevezett *események* hatására valósulnak meg. Mivel a rendszerünket Petri-hálóval adtuk meg, ezért a háló tranzícióinak tüzelése megfeleltethető egy-egy eseménynek. Aszinkron rendszerek esetén a következő-állapot függvényt lehetőségünk van több részre bontani, amelyek uniója megegyezik az eredeti függvénnyel: $\mathcal{N} = \bigcup_{\alpha \in \varepsilon} \mathcal{N}_\alpha$, ahol $\varepsilon$ az események véges halmaza, $\mathcal{N}_\alpha$ pedig a következő-állapot függvény az $\alpha$ eseményre nézve. Azt mondjuk, hogy egy $\alpha$ esemény engedélyezve van egy *i* állapotban, ha $\mathcal{N}_\alpha(\mathrm{i}) \neq \{\ \}$, különben nincs engedélyezve.

Állapottér-felderítés során a szaturáció eredménye az elérhető állapotok halmaza, amelyet $\mathcal{S}$-sel szokás jelölni. $\mathcal{S} \subseteq \hat{\mathcal{S}}$ az a legszűkebb halmaz, amely tartalmazza az $s$ kiinduló állapotot és már tovább nem bővíthető $\mathcal{N}$ iteratív alkalmazásával: $\mathcal{S} = \{\mathrm{s}\} \cup \mathcal{N}(\mathrm{s}) \cup \mathcal{N}\big(\mathcal{N}(\mathrm{s})\big) \cup \ldots = \mathcal{N}^*(\mathrm{s})$, ahol * a tranzitív lezártat jelöli.

## 3.2 Állapottér-felderítés szaturációs algoritmussal

Szaturáció alkalmazásával az eddigi megoldásokhoz képest kedvező feltételek mellet deríthető fel a vizsgált modell állapottere. A szaturáció az explicit, sőt más szimbolikus technikákhoz képest is idő- és tárhatékony. Ennek eléréséhez többértékű döntési diagramokat használ a felderített állapottér tárolására.

A szaturáció működését a 7. ábra tekinti át.




**7. ábra Szaturáció - áttekintő ábra**

A szatuáció bemenete egy rendszer Petri-háló alapú modellje, kimenete pedig a rendszer állapotterét kódoló döntési diagram. A megvalósított állapottér-felderítés több lépésből áll:

- A modell dekomponálása
- A következő-állapotfüggvény Kronecker mátrix alapú leképzése
- Állapottér-generálás, azaz a döntési diagram csomópontjain való speciális iteráció

A modell szükséges dekomponálását a 3.2.1 fejezet ismerteti, a Kronecker alapú megoldás a 3.2.3 fejezetben ismerhető meg, míg az algoritmus átfogó működése a 3.2.5 fejezetben kerül bemutatásra.

### 3.2.1 Dekompozíció

A szaturáció megkezdése előtt a bemeneti Petri-hálónkat K db részmodellre bontjuk. Az így kapott részmodellek lokálisan is bejárhatók. Ennek következménye, hogy a rendszer egy globális *i* állapota K darab lokális állapottal adható meg: $i = (i_1, i_2, \dots, i_K)$, ahol $i_j$ a j. részmodell egy lokális állapota. Ezek alapján a rendszer lehetséges állapotainak $\hat{S}$ halmazát a K darab részmodell lehetséges állapotainak halmazának Descartes szorzata adja meg: $\hat{S} = S_1 \times S_2 \times \dots \times S_K$, ahol $S_j$ a j. részmodell lehetséges lokális állapotainak halmaza.

Egy modell dekompozíciója *Kronecker-konzisztens* [14] az alábbi esetben:

- $i = (i_1, i_2, \dots, i_K)$, azaz a modell globális állapotát meghatározza a részmodellek lokális állapotaink az összessége,

- $\mathcal{N}(i) = \bigcup_{\alpha\in\varepsilon} \mathcal{N}_\alpha(i)$, azaz az i állapotból elérhető állapotok halmaza megegyezik az események külön-külön történő tüzelésével elérhető állapothalmazok uniójával,
- végül pedig $\mathcal{N}_\alpha(i) = \mathcal{N}_{1,\alpha}(i_1) \times \ldots \times \mathcal{N}_{K,\alpha}(i_K)$, vagyis a következő-állapot függvény felbontható lokális következő-állapot függvények Descartes szorzatára.

Petri-hálók dekompozíciójára oly módon van lehetőség, hogy azt részhálókra bontjuk és így a token-eloszlás a K darab részeloszlásból képzett vektorként áll elő. A Petri-hálók ilyen típusú felbontása Kronecker-konzisztens lesz.

Fontos tulajdonság, hogy a modellellenőrzés során a Petri-háló dekompozíciója meghatározza az állapotokat kódoló MDD szerkezetét. Ez úgy lehetséges, hogy a dekompozíció során kialakított részmodell lokális állapotait az MDD egy-egy szintje fogja kódolni. Ennek megfelelően különböző felbontások esetén különböző szerkezetű - de azonos állapothalmazt kódoló - MDD-ket fogunk eredményül kapni. Itt megjegyezhető az is, hogy a különböző felbontások eltérő mértékben befolyásolják a modellellenőrzés hatékonyságát, azonban a legoptimálisabb szintezés megtalálása NP-teljes feladat [27].

### 3.2.2 Események lokalitása

A modell dekompozíciója akkor nyer értelmet, ha felismerjük, hogy így az egyes események a részmodelleknek csak egy csoportjára vannak hatással.

Aszinkron rendszerek esetén sok esemény független sok részmodelltől, azaz az esemény tüzelése nincs hatással a részmodell állapotára.

Az események lokalitásának vizsgálatához új jelöléseket vezetünk be:

- $\mathrm{Top}(\alpha) = k$, amely megadja, hogy a k. szint a legmagasabb azok közül, amit az $\alpha$ esemény befolyásol,
- $\mathrm{Bot}(\alpha) = k$, amely megadja, hogy a k. szint a legalacsonyabb azok közül, amit az $\alpha$ esemény befolyásol.

A Top és Bot függvényekre igaz, hogy $K \geq \mathrm{Top}(\alpha) \geq \mathrm{Bot}(\alpha) \geq 1$ minden $\alpha$ esemény esetén.

Az események lokalitását kihasználva szignifikánsan csökkenthető a szükséges tüzelések száma. Ez abból következik, hogy az eseményeket nem szükséges a döntési diagram teljes magasságában eltüzelni, hanem elegendő az események Top és Bot szintjei között.

Ezek alapján az események halmaza felbontható: $\varepsilon_k = \{\alpha \in \varepsilon | \mathrm{Top}(\alpha) = k\}$, minden k-ra, ahol $1 \leq k \leq K$. Erre majd a későbbiekben lesz szükségünk.

### 3.2.3 Következő-állapot függvény, Kronecker-kódolás

A következő-állapot függvényt gyakorta 2K magasságú döntési diagrammal tárolják. Mivel a Petri-hálók Kronecker konzisztensek, ezért lehetőség van arra, hogy a részmodellek és események alapján felbontott következő-állapot függvényt mátrixok segítségével tároljuk.

A Kronecker-kódolás pontosan $K \cdot |\varepsilon|$ mátrixszal tárolja a következő-állapot függvényt [16]. Minden $\mathcal{N}_{k,\alpha}$ lokális függvény kódolható egy $N_{k,\alpha} \in \{0,1\}^{|\mathcal{S}_k| \times |\mathcal{S}_k|}$ mátrixszal, ahol $N_{k,\alpha}(i,j) = 1 \Leftrightarrow j \in \mathcal{N}_{k,\alpha}(i)$, azaz az $N_{k,\alpha}$ mátrix i. sor és j. oszlopbeli eleme 1, ha az $\alpha$ esemény tüzelésének hatására a k. részmodell i állapotból j állapotba jut.

Mivel Petri-hálók esetén az események tüzelése a kiinduló állapot ismeretében egyértelműen meghatározza a tüzelés utáni állapotot, ezért a Kronecker-mátrixok soronként csak 1 darab nem-nulla értéket fognak tartalmazni. Ezt a tulajdonságot figyelembe vevő megvalósítás esetén azonban a Kronecker-kódolás tárhatékonysága jobb, mint az azonos állapotátmeneteket tároló döntési diagramoké.

Ha az események lokalitását figyelembe vesszük, akkor elmondható, hogy azon részmodell-esemény párosoknál, ahogy a részmodell független az eseménytől, a Kronecker-mátrix egy egységmátrix lesz. Az előzőleg megismert definíciókat alkalmazva: $N_{k,\alpha} = \mathcal{I} \Leftarrow k > \mathrm{Top}(\alpha) \vee k < \mathrm{Bot}(\alpha)$. Ha ezt kihasználjuk, akkor még hatékonyabban tárolhatók a Kronecker-mátrixok.

### 3.2.4 Helyben frissítés

A szaturáció során MDD-t építünk a felderített állapotokból. Az MDD csomópontokból épül fel, amelyek globális állapotok halmazait kódolják. A csomópontokon értelmezzük a tüzelés műveletét, amikor is először az eltüzelendő eseményre megvizsgáljuk, hogy engedélyezett-e a csomópont által reprezentált lokális állapotban. Ha engedélyezett, akkor azt az állapotot is hozzáadjuk a csomópont által kódolt állapothalmazhoz, amelybe az esemény tüzelésének hatására kerül a rendszer.

A szaturáció végrehajtása során az eseményeket kimerítően tüzeljük a döntési diagram csomópontjain. A tüzelések során lehetőségünk van arra, hogy a tüzelést eredményét ne új csomópont létrehozásával tároljuk, hanem a meglévő csomópont éllistáját inkrementálisan bővítsük. Ezt akkor tehetjük meg, ha a $\langle k|p\rangle$ csomóponton egy olyan $\alpha$ eseményt tüzelünk el, amire $\mathrm{Top}(\alpha) = k$. Az események lokalitását ily módon kihasználva sok döntési diagram műveletet spórolunk meg, illetve jelentősen csökken a memória-felhasználás is.

### 3.2.5 Szaturáció

A szaturáció újdonsága abban rejlik, hogy a rekurzív bejárást nem egyenként az állapotokra, hanem csomópontokra alkalmazza, majd az esetleges változásokat rekurzívan lefelé érvényesítve számítja ki az elérhető állapotok halmazát. A szaturáció továbbá az eseményeket mohó módon, azaz az egyes csomópontokra kimerítően tüzeli.

A teljes szaturációs algoritmus megadása előtt szükségünk van még egy jelölés bevezetésére: $\mathcal{N}_{\leq k} = \bigcup_{1\leq l\leq k} \mathcal{N}_{\varepsilon_l} = \bigcup_{\alpha:\mathrm{Top}(\alpha)\leq k} \mathcal{N}_\alpha$ . Azaz $\mathcal{N}_{\leq k}$ azt a következő-állapot függvényt jelenti, amely azon α eseményekre értelmezett, amelyekre a $\mathrm{Top}(\alpha)$ legfeljebb k.

Ezek alapján egy $\langle k|p\rangle$ csomópontot *szaturált*nak mondunk [17], ha kódolja az összes olyan állapotot, ami $\alpha: \mathrm{Top}(\alpha) \leq k$ események hatására elérhetők, azaz $\mathcal{B}(\langle k|p\rangle) = \mathcal{N}^*_{\leq k}\big(\mathcal{B}(\langle k|p\rangle)\big)$.

A teljes állapottér akkor tekinthető teljesen felderítettnek, ha a döntési diagram $\langle K|r\rangle$ gyökér csomópontja szaturálttá vált. Ha a kiindulási döntési diagramot úgy inicializáltuk, hogy az pontosan a kezdeti állapotot reprezentálja, azaz $\mathcal{B}(\langle K|r\rangle) = \{s\}$, akkor a szaturáció végén a döntési diagram a modell teljes állapotterét fogja tárolni ($\mathcal{B}(\langle K|r\rangle) = \mathcal{S}$).

A fentiek függvényében a szaturációs algoritmus futása a következő lépésekből áll:

- Felépítünk egy döntési diagramot, amely a kezdő állapotot kódolja.
- Szaturáljuk az összes csomópontot az 1. szinten.
- Szaturáljuk az összes csomópontot a 2. szinten. Ha menet közben új csomópont jött létre az 1. szinten, akkor azt azonnal szaturáljuk.
- Minden $2 < k \leq K$ szinten szaturáljuk az összes k. szinten levő csomópontot. Ha menet közben új csomópont jött létre k-tól alacsonyabb szinten, akkor azt azonnal szaturáljuk.
- Ha a legfölső szinten levő, gyökér csomópontot is szaturáltuk, akkor készen vagyunk az állapottér bejárásával. Ekkor az épített döntési diagram a modell teljes állapotterét kódolja.

### 3.2.6 Az algoritmus megvalósítása

A 2.6 fejezetben bemutatott PetriDotNet keretrendszer munkám megkezdésekor már tartalmazott egy szekvenciális szaturációs modult. A keretrendszerhez hasonlóan ezt is a BME Méréstechnikai és Információs Rendszerek Tanszékén fejlesztették.

A szaturáció eredményének tárolását megvalósító MDD modul szintén a tanszéken készült, mivel a fejlesztés folyamán nem állt rendelkezésre megfelelő tudású, és

hatékonyan integrálható külső könyvtár. A helyi fejlesztés előnye, hogy az MDD könyvtár kódjához én is hozzáférhettem, ugyanis saját munkám implementálása közben azt néhol módosítani, kiegészíteni kényszerültem, hogy többszálú környezetben is konzisztens működést mutasson.

Dolgozatom szempontjából a szekvenciális modul felépítése és implementációja tekintetében csupán csak annyi a releváns, hogy a sebességnövekedés érdekében több helyen gyorsítótár került alkalmazásra. Az első gyorsítótár az úgynevezett UnionCache. Az UnionCache lehetővé teszi, hogy a döntési diagram csomópontjain végrehajtott unió műveletek eredményét eltároljuk. Ha a későbbiekben olyan éllistájú csomópontoknak szeretnénk az unióját képezni, amelyekre egyszer már megtettük, akkor annak újbóli kiszámítása helyett az eredményt elővesszük az UnionCache-ből. A másik említendő gyorsítótár az úgynevezett FireCache. A FireCache-ben az események tüzelésének eredményét helyezzük el. A megfontolás hasonló az UnionCahce-hez, ugyanis elképzelhető, hogy egy adott csomóponton egy esemény tüzelését többször is kérvényezik magasabb szintekről. A gyorsítótárak alkalmazásával szignifikánsan csökken a futási idő. A 4.2.1 fejezetben még vissza fogok térni a gyorsítótárak kérdésére, ugyanis ezeken is változtatásokat kellett végrehajtanom a párhuzamosítás miatt.

A szekvenciális modul részletesebb szerkezete, megvalósítása az eszköz honlapján olvasható [4].

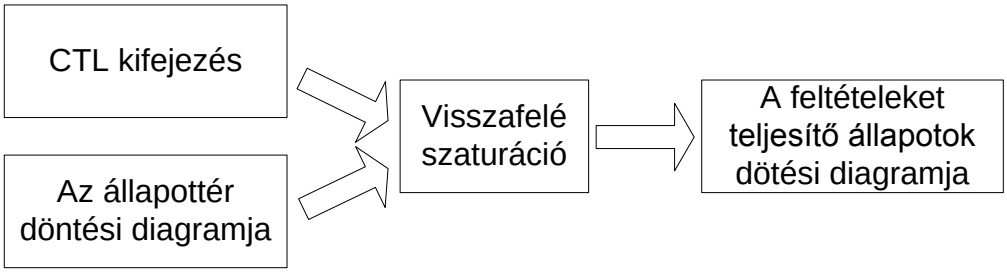
## 3.3 Modellellenőrzés szaturációs algoritmussal

A 2.5 fejezetben bemutattam a lehetséges CTL kifejezéseket. Az útvonalkvantorok és állapotkvantorok kombinációjaként keletkezett 8 kifejezés közül kiemelendő az EX, EU és az EG operátor. Ha ezt a 3 operátort megvalósítjuk, akkor lényegében az összes operátort leképeztük, ugyanis ezen 3 operátor segítségével a maradék 5 kifejezhető [2].

A PetriDotNet rendszerben a szekvenciális állapottér-felderítő modulon kívül található egy szekvenciális modellellenörző modul is. Én ebben a modulban az EU operátor implementációját egészítettem ki párhuzamos működésűvé. Azért az EU operátort választottam, mivel annak megvalósítása is szaturációval történik.

A szaturációt alkalmazó modellellenőrzés áttekintése a 8. ábrán látható. A modellellenőrzés bemenete a CTL kifejezés (2.5 fejezet), amelynek az igazságtartalmát szeretnénk vizsgálni, és az ellenőrizendő rendszer állapottere döntési diagrammal kódolva. A modellellenőrzés kimenete szintén egy döntési diagram, amely a vizsgált

feltételt teljesítő állapotokat kódolja. A folyamat során úgynevezett visszafelé szaturációt alkalmazunk, amelyet a 3.3.2 fejezetben mutatok be.




**8. ábra Modellellenőrzés szaturációval - áttekintő ábra**

### 3.3.1 Strukturális modellellenőrzés megközelítése

A strukturális modellellenőrzés alapötlete az, hogy abból az állapothalmazból indulunk ki, amelyre az ellenőrizendő feltétel teljesül, majd ebből az állapothalmazból lépkedünk visszafelé a logikai időnek megfelelően. A működés úgy is elképzelhető, hogy az egyes tranzíciók inverzeit tüzeljük el. Ha az eredményül kapott halmazban szerepel a kiindulási állapot, akkor azt mondhatjuk, hogy a vizsgált kifejezés igaz.

A strukturális modellellenőrzés megvalósításához a következő-állapot függvény inverzére, illetve a Kronecker mátrixok transzformáltjára van szükség. Ez formálisan azt jelenti, hogy ha egy α esemény tüzelésével *i* állapotból *j* állapotba jutunk, akkor $\mathcal{N}_\alpha(\mathrm{i}) = \mathrm{j}$ és $\mathcal{N}_\alpha^{-1}(\mathrm{j}) = \mathrm{i}$. Hasonlóan $\mathrm{N}_\alpha[\mathrm{i},\mathrm{j}] = 1 \Leftrightarrow \mathrm{N}_\alpha^{\mathrm{T}}[\mathrm{j},\mathrm{i}] = 1$.

Továbbá a modellellenőrzés előtt ismerni kell a teljes állapotteret, hogy abból meghatározhassuk azt a részhalmazt, ahol a vizsgált feltétel teljesül. Emiatt a modellellenőrzés előtt mindenképp állapottér-felderítést is kell végezni, aminek következménye, hogy a Kronecker mátrixok is ismertek lesznek a modellellenőrzés kezdetén.

### 3.3.2 EU operátor algoritmusa

**Tradicionális megvalósítás**

A 2.5 fejezetben megismert E(p U q) kifejezés jelentése a következő: létezik olyan útvonal, amelyen p mindaddig igaz, amíg Q igazzá nem válik. A probléma közelítésének legegyszerűbb módja, ha kiindulunk azoknak az állapotoknak a halmazából, amelyekben q teljesül. Ekkor a logikai időben visszafelé haladva tüzeljük az események inverzét. Azonban figyelni kell arra, hogy csak olyan állapotok kerülhetnek be a megoldáshalmazba, amelyek mindegyikében teljesül p. Ez úgy oldható meg, hogy a következő-állapot függvény

inverzének alkalmazása után a kapott állapothalmazt metszeni kell, és csak azokat megtartani, ahol p teljesül.

A tradícionális megvalósítás pszeudó kódja az alábbi:

```
EUtrad (in: P,Q: állapothalmaz, out: X: állapothalmaz)

1 X=Q
2 repeat
3      Y=X
4      X=X ∪ (𝒩⁻¹(X) ∩ P)
5 until X=Y
```

**Megvalósítás szaturáció alkalmazásával**

A bemutatott tradicionális megvalósítás fő hátránya, hogy a gyakori metszés igen költséges. Mindazonáltal a szaturáció alkalmazásával a metszések elkerülhetők [2].

Az alapötlet a szaturáció alkalmazásakor is hasonló a tradicionális megvalósításhoz, azaz abból a halmazból indulunk ki, ahol q teljesül (ezeknek az állapotoknak a halmazát jelöljük Q-val). Innen pedig visszafelé lépkedve keressük meg azokat az állapotokat, ahol p teljesül (ezeknek az állapotoknak a halmazát pedig jelöljük P-vel). A fő nehézséget azonban az egyszerű állapottér-generáláshoz képest az jelenti, hogy a megoldáshalmazba csak olyan állapotokat vehetünk fel, amelyekben p teljesül. Ha az eredményhalmazban akár csak egy olyan állapot is előfordulna, ahol p ideiglenesen nem teljesül, akkor hamis eredményt kapnánk.

Ha a tradicionális megvalósításhoz hasonlóan minden tüzelés után elmetszenénk az eredményt P-vel, akkor az hatalmas többletköltséget jelentene, ugyanis a tüzelésekhez képest a metszés sokkal költségesebb művelet. Ehelyett úgynevezett részleges szaturációt alkalmazunk [2], azaz a szaturáció során csak olyan eseményeket tüzelünk, amelyek után nincs szükség a P-vel való metszésre. Az olyan eseményeket, amelyek nem tartják meg garantáltan p teljesülését továbbra is a tradicionális algoritmussal vizsgáljuk. Mivel nem kell az összes eseményre tüzelni, ezért az általános szaturációs függvény itt nem használható. Helyette egy *BackSaturate* nevű függvényt hoztam létre.

Mielőtt a szaturációt megkezdenénk az U operátor vizsgálatához, az ε-beli eseményeket csoportokba kell osztani:

- X halmazra nézve *halott események*: olyan események, amelyek semmilyen X-beli állapotból nem vezetnek X-beli állapotba.

- X halmazra nézve *biztonságos események*: olyan események, amelyek nem halottak és semelyik X-en kívüli állapotból nem vezetnek X-beli állapotba, azaz inverzük nem vezet ki X-ből.
- X halmazra nézve *nem biztonságos események*: olyan események, amelyek nem halottak és nem biztonságosak.

Az események besorolását a fenti 3 csoportba a *ClassifyEvents* függvény végzi. A definíciókban szereplő X halmaz az EU operátor vizsgálatakor megfelel a P halmaznak.

Az inverz állapottér-generálás során a halott események elhagyhatók, ugyanis P-ből sosem engedélyezettek.

A biztonságos események sosem vezetnek ki P-ből, így a tüzelés után nincs szükség metszet műveletre. E tulajdonság miatt alkalmazható a szaturáció a P-re nézve biztonságos események halmazára.

A nem biztonságos események tüzelése után szükséges a metszet képzése, ugyanis nem tudjuk, hogy bekerültek-e olyan állapotok is a megoldásba, ahol p nem teljesül. A nem biztonságos eseményekre a tradicionális algoritmust kell alkalmazni.

A fentieknek megfelelően a szaturációt alkalmazó algoritmus pszeudó kódja:

```
ClassifyEvents (in: X: állapothalmaz, out: ε_U, ε_S: eseményhalmaz)

1 ε_U = ε_S = { }
2 for each α in ε do
3     if N_α^-1(X) ≠ { } ∧ N_α^-1(X) ⊆ X then
4         ε_S = ε_S ∪ {α}
5     else if N_α^-1(X) ∩ X ≠ { }
6         ε_U = ε_U ∪ {α}
7     end if
8 end for
```

```
EUsat (in: P,Q:állapothalmaz, out: X: állapothalmaz)

1 ClassifyEvents(P, ε_U, ε_S)
2 X=Q
3 BackSaturate(X, ε_S)
4 repeat
5     Y = X
6     X = X ∪ (N_U^-1(X) ∩ (P ∪ Q))
7     if X ≠ Y then
8         BackSaturate(X, ε_S)
9     end if
10 until X = Y
```

# 4 Párhuzamos szaturáció és fejlesztéseim

A most következő fejezet a szaturáció párhuzamos változatát mutatja be. A 4.1 rész egy rövid áttekintést nyújt a szaturációtól különböző, de szintén párhuzamos technikákról. Ezt követően a 4.2 fejezet azokat a változtatásokat mutatja be, amelyeket a [3]-ban publikáltak, és ahhoz szükségesek, hogy a szaturáció párhuzamosításra alkalmas környezetben is a lehető legjobban kihasználja a rendelkezésre álló erőforrásokat. A 4.3 és a 4.4 fejezetekben pedig azt mutatom be, hogy milyen fejlesztéseket végeztem el az állapottér-felderítő és modellellenőrző algoritmusokon.

## 4.1 Korábbi párhuzamos szimbolikus algoritmusok

Az utóbbi években számos kísérlet történt párhuzamos modellellenőrző algoritmusok implementálására. A jelenleg elérhető modellellenőrző keretrendszerek legnagyobb része valamely explicit technikán alapulnak. Ebből a szempontból a PetriDotNet párhuzamos állapottér-felderítő modulja hiánypótló, hiszen itt szimbolikus technikán alapul az elérhető állapotok kiszámítása.

A PREACH (Parallel Reachability) [18] nevű eszköz egy Erlang és C++ nyelveken implementált párhuzamos modellellenőrző, amely a Stern és Dill által közölt DEMC (Distributed Explicit-state Model Checking) [19] algoritmuson alapul. Itt az állapotok egy elosztott mélységi bejárásáról van szó. A program egy megfelelő hash függvénnyel az egyes állapotokat csomópontokhoz rendeli hozzá, amelyek itt processzoroknak felelnek meg. Minden egyes csomópont csak a hozzá tartozó állapotot fejti ki, azaz nézi meg az abból elérhető állapotokat, a nem hozzá tartozó csomópontokat pedig a tulajdonos processzornak küldi át. Az Erlang programozási nyelv az üzenetváltáson alapuló konkurencia modelljével rendkívül hasznosnak bizonyult, a program magjának megírása 1000 kódsoron belül sikerült. A hivatkozott dokumentum arról számol be, hogy ipari méretű feladatok végrehajtásához alkották meg az eszközt, így említést tesznek 30 milliárd állapotot tartalmazó állapottér ellenőrzéséről is.

A DIVINE [20] egy párhuzamos LTL modellellenőrző keretrendszer, amely hálózatba kapcsolt multiprocesszoros gépek erőforrásainak együttes kihasználására is képes. A program az egyes munkaállomásokon belül futó szálak között osztott memórián alapuló konkurenciát valósít meg, míg az egyes munkaállomások az MPI-t [21] használják fel a kommunikációhoz. Az eszköz LTL formula vagy Büchi automata [8] formájában megfogalmazott követelmények teljesülését tudja ellenőrizni. Az implementáció alapjául a

[3]-ben közölt algoritmus szolgált. Az algoritmus feldarabolja a rendszer állapotterét és az egyes részeket rendeli hozzá processzorokhoz, minden processzor csak a saját részébe tartozó állapotot fejti ki, ehhez saját hash táblában tárolja el az elért állapotokat. A szálak közötti kommunikáció itt is üzenetváltáson alapul, ezzel csökkentve a kommunikáció többletköltségét.

A [22]-ben közölt módszer megkülönböztet dolgozó és koordinátor processzorokat, amely utóbbiból csak egy lehet a futás során. A csomópontok itt is hozzá vannak rendelve az egyes processzorokhoz, de ezen felül hatékony terheléskiegyenlítés is része a megoldásnak, amelyet munkaállomások csoportján történő végrehajtásra optimalizáltak. Ha egy munkaállomás az állapottér bejárása során a felhasználható memória szűkösségét érzékeli, akkor a BDD-t szétvágja több részre. Egy részt megtart magának, a maradékot pedig a koordinátor processzor hozzárendeli a többi szabad dolgozó processzorhoz bejárásra. A másik véglet az, amikor egy munkaállomás kihasználtsága nagyon alacsony. Ebben az esetben több alacsony kihasználtságú munkaállomás feladata összevonásra kerül és csak egy munkaállomás fog foglalkozni a megnövekedett feladat végrehajtásával, a többi dolgozó processzor szabad állapotú lesz. A terheléskiegyenlítés lebonyolításáért minden esetben a koordinátor munkaállomás felelős.

## 4.2 Párhuzamos szaturáció áttekintése

A szaturációs algoritmus [3]-ban közölt párhuzamos megvalósítása lehetővé teszi, hogy a 3. fejezetben megismert szaturáció során a Petri hálóval megfogalmazott rendszer állapotterének felderítése több szálon hajtódhasson végre. Megfelelő hardver eszközök esetén ez azt is jelenti, hogy az állapottér felderítés kisebb részfeladatai közül több akár egyidejűleg is feldolgozásra kerülhet.

Hogy a szaturáció párhuzamosan legyen végrehajtható, az állapottér felderítést részekre kell bontani, amely részeket az egyes szálak külön-külön tudnak feldolgozni. Mivel a szálak közötti kommunikáció és szinkronizáció költséges és időigényes, ideális esetben a képzett részfeladatok egymástól függetlenek. Mindazonáltal ezt nem könnyű biztosítani, mivel a szaturáció egyes lépéseinek sorrendje jellemzően meghatározott, közöttük erős függőségek vannak, így az egyes részfeladatok méretének kiválasztása igen nehéz. Ezen felül a modellek is eléggé korlátozzák a független részfeladatok kialakítását, hiszen ritka az olyan modell, ami teljesen független komponensekből áll.

A [3]-ban leírtak alapján az egyes feladatokat az események tüzeléséből célszerű kialakítani. Ez lehetővé teszi, hogy a vizsgált modell állapotterét reprezentáló MDD-t az

egyes szálak egymástól függetlenül kisebb részekből építhessék fel, ugyanis az egyes tüzelések eredményei azonos rész-MDD-kben helyezkednek el.

Mivel az egyes részfeladatok még így is függnek kis mértékben egymástól, a [3]-ban bevezették az úgynevezett *felfelé mutató él*eket a szekvenciális algoritmus további kiterjesztéseként. A felfelé mutató élek az egyes részfeladatok közötti függőségeket hivatottak kifejezni.

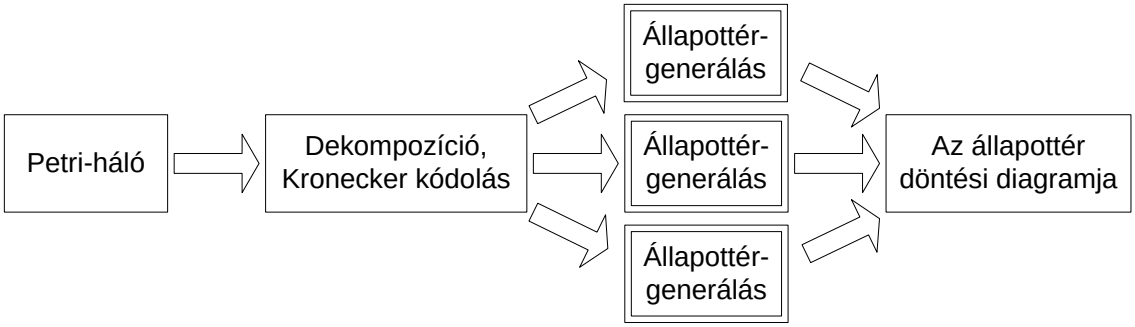
Az egyes szálak közötti függőségek az algoritmus futása során úgy mutatkoznak meg, hogy a megoldást reprezentáló MDD-ben a szálakhoz tartozó rész-MDD-k csomópontjai között kellene új (lefelé mutató) élet behúzni. Az él behúzásának feltétele, hogy az él végpontjában levő csomópont szaturált állapotban legyen. Mivel a párhuzamos feldolgozás miatt a szálak egymáshoz képest aszinkron módon működnek, ezért ez a feltétel nem minden esetben teljesül abban a pillanatban, amikor az élet be szeretnénk húzni. Hogy ilyenkor, az él létrehozását kezdeményező szálnak ne kelljen várakoznia a feltétel teljesülésére - hanem feladatának végrehajtását folytathassa – a lefelé mutató él behúzásának igényét egy felfelé mutató éllel helyettesítjük. Ekkor a felfelé mutató él kezdőpontja a még szaturálatlan csomópont, végpontja pedig az eggyel magasabb szinten levő, kezdeményező csomópont azon lokális állapota lesz, amelyikből a lefelé mutató indult volna. Ha az újonnan behúzott felfelé mutató él kezdőpontjában lévő csomópont szaturációja befejeződött, akkor az azon dolgozó szál a felfelé mutató élet egy lefelé mutatóra cseréli, és az él behúzásából eredő feladatokat innentől ő hajtja végre.

Összefoglalva a fejezetben leírtakat, a szekvenciális szaturációs algoritmus párhuzamossá alakításához a részfeladatok fogalmának, és a részfeladatok közötti függőségeket kifejező felfelé mutató élek bevezetése szükséges.

### 4.2.1 Az algoritmus bemutatása, megvalósítása

A párhuzamos szaturációs algoritmus elkészítéséhez megfelelő alapot nyújt a szekvenciális szaturációs algoritmus [1][3][17]. A most következő fejezet azokat a konkrét változtatásokat mutatja be, amelyek a szekvenciális algoritmus párhuzamossá alakításához szükségesek.

A párhuzamos algoritmus felépítését a 9. ábra mutatja be. A szekvenciális szaturációhoz hasonlóan (7. ábra) a párhuzamos esetben egy Petri-háló a bemenet, egy döntési diagram a kimenet valamint szükséges a rendszer dekompozíciója. A szignifikáns különbség abban rejlik, hogy magát az állapottér felderítését több (az ábrán 3) részre osztjuk, és ezeket lehetőleg egymástól függetlenül hajtjuk végre.




**9. ábra Párhuzamos szaturáció - áttekintő ábra**

A szekvenciális szaturációs algoritmuson az alábbi változtatások szükségesek az állapottér párhuzamos generálásához:

- a work pool [23] tervezési minta bevezetése a feladatok elosztására
- a csomópontokat reprezentáló adatszerkezet kiegészítése új attribútumokkal
- a tüzelési gyorsítótárban tárolt értékek bővítése, a többszörös hozzáférés biztosítása
- az MDD-tárolóhoz való többszörös hozzáférés biztosítása
- az egyéb adatstruktúrákhoz (pl. állapotátmenetekhez tartozó adatszerkezetekhez) való többszörös hozzáférés biztosítása.

#### 4.2.1.1 Work pool tervezési minta

A 3.2.1 fejezetnek megfelelően a szaturáció folyamatát részfeladatokra kell bontani, majd ezeket az egyes szálakhoz feldolgozásra hozzárendelni. Nem megfelelő hozzárendelés, illetve ütemezés esetén előfordulhat, hogy a terhelés eloszlás a szálak között egyenetlen lesz.

Mivel az események tüzeléséből kialakított feladatok mérete változó és előre nehezen becsülhető, ezért a legegyszerűbb ütemezési technikák, mint például a Round Robin alkalmazása nagyon könnyen azt eredményezné, hogy a különböző szálak által elvégzett feladatmennyiség egyenetlen lenne. Ilyen egyenetlenség esetén azt mondhatjuk, hogy erőforrásokkal pazarlóan bánunk, hiszen egyes szálak tétlenül várakoznak, míg mások a nekik kiosztott feladatokat végzik.

A [3]-ban ismertetett párhuzamos megvalósítás az úgynevezett work pool tervezési mintát [23] használja a feladatok minél egyenletesebb elosztására. A work pool tervezési minta lényege, hogy a részfeladatok dinamikusan, futási időben kerülnek az egyes szálakhoz hozzárendelésre, ezáltal az algoritmus futása során végig biztosítható egy optimálisnak tekinthető terheléselosztás.

#### 4.2.1.2 Csomópont adatszerkezet kiterjesztése

Mint ismeretes, a párhuzamos feldolgozás többletköltséggel jár. Ide sorolható többek között, hogy egyes adatszerkezetek megváltoztatása, kibővítése válhat szükségessé. A szaturáció párhuzamosítása során a csomópontokat reprezentáló adatszerkezetet kell új mezőkkel ellátnunk.

A szaturáció párhuzamosítása során az MDD csomópontjait reprezentáló adatszerkezeten bevezetendő új mezők:

- upward edges: {szint, index, lokális állapot} hármasok halmaza
- ops: integer változó
- saturating: bool változó
- key: Key típusú változó

A bevezetendő változók szükségességét az alábbiak indokolják.

- *Upward edges*: A 4.2.1 fejezetben bemutatott felfelé mutató élek nyilvántartására szolgál az élek kiinduló pontjában.
- *Ops*: Szinkronizációs okokból fontos, hogy a szaturáció során végig nyilvántartsuk, hogy az egyes csomópontokon egy bizonyos időpillanatban hány szál végez valamilyen műveletet. Az ops nevű számláló értékének növelése akkor szükséges, amikor egy szál elkezdi a csomópont szaturációját, vagy egy eseményt kíván eltüzelni a vizsgált csomóponton. A változó értékét pedig akkor csökkentjük, mikor ezek közül bármelyik befejeződik. A számláló ilyen típusú használatával elkerülhető, hogy szinkronizáció hiányában egy csomópont szaturációját több szál is végrehajtsa. Az ops változó bevezetésének másik aspektusa, hogy míg egy csomópontra léteznek felfelé mutató élek, addig a csomópont szaturációja nem fejeződhet be. Mivel a felfelé mutató élek feldolgozása azok kezdőpontjában történik, ezért az élek végpontjában elegendő azt nyilvántartani, hogy van-e a csomópontra mutató felfelé él. E megvalósítására szintén használható az ops változó, ha új él létrehozásakor az ops értéket növeljük, az él törlésekor pedig csökkentjük.
- *Saturating*: Szintén szinkronizációs célokat szolgál a saturating nevű bool változó bevezetése. Az algoritmus során előfordul, hogy bizonyos csomópontok szaturációja megszakad ideiglenesen fennálló függőségek miatt, majd azok megszűntével folytatódik. Más esetekben a függőségek feloldása még az érintett csomópont szaturációja előtt megtörténik. Ebben az esetben a függőség megszűnésekor a csomópont szaturációját el lehet kezdeni. A saturating változó arra szolgál, hogy a függőséget feloldó szál tudja, hogy az érintett csomópont szaturációját folytatnia vagy

éppen kezdeményeznie kell. A változó alapértéke hamis, míg a szaturáció megkezdésekor értékét igazra kell állítani.

- *Key*: Mivel az a szaturáció során az egyes tüzelések eredményét nem csak elhelyezzük a tüzelési tárolóban, hanem olykor frissítjük is a hozzájuk tartozó értéket, ezért ismerni kell, hogy az adott csomópont milyen kulccsal megcímezve került a tárolóba. Ennek a kulcsnak a feljegyzésére szolgál a key nevű változó.

#### 4.2.1.3 Tüzelési gyorsítótár változásai

A szaturáció során a többször előforduló tüzeléseket fölösleges többször elvégezni. Ezt a szekvenciális algoritmusban a tüzelési gyorsítótár bevezetésével oldották meg. A tüzelési gyorsítótár tartalmazza a már elvégzett tüzelések eredményét a hívó csomópont és az eltüzelt esemény párosából alkotott kulccsal megcímezve. Párhuzamos esetben több változtatást is végezni kell a tüzelési gyorsítótár megfelelő működéséhez.

A tüzelési gyorsítótár szerkezetében és működésében a következő változtatások szükségesek:

- szinkronizációs okokból a tüzelés elkezdését is jelezni kell a gyorsítótárban; nem elegendő csak az eredményt elhelyezni benne
- megoldandó a tüzelési gyorsítótárhoz való többszörös hozzáférés

A tüzelési gyorsítótár felhasználható arra, hogy az egyes események eltüzelését az egyes csomópontokra legfeljebb egyszer végezzük el. A szekvenciális esetben, ha a tüzelés eredménye még nem található meg a tüzelési gyorsítótárban, akkor az egyetlen szál elvégzi azt, az eredményt pedig elhelyezi a gyorsítótárban. Párhuzamos esetben ez a módszer nem elégséges, ugyanis előfordulhatna, hogy több szál egy olyan esemény tüzelését kezdi meg, amelyiket már egy másik szál is elkezdett, de még nem fejezett be, így annak eredménye még nincs benne a tüzelési gyorsítótárban. Az ilyen esetek elkerülésére, a szálakat szinkronizálni kell oly módon, hogy a szálak már a tüzelés megkezdésekor elhelyeznek egy olyan *kulcs-érték páros*t a tüzelési gyorsítótárban, amelynek:

- a kulcsa egy csomópont, és a rajta eltüzelendő esemény párosa
- az értéke pedig a tüzelés eredményét tároló csomópont, és egy bool érték, amely jelzi, hogy az értékben tárolt csomópont szaturált-e már.

Mivel a tüzelés eredményének új csomópontot hoz létre az algoritmus, ezért az a tüzelés megkezdésekor még nem lehet szaturált, így az eredmény szaturáltságát jelző érték először mindig false. A csomópont szaturációjának végén természetesen a gyorsítótárban is jelezni kell, hogy immáron szaturált a csomópont. A korábban említett key változót azért

tároltuk el az egyes csomópontokban, hogy most a tüzelési gyorsítótárban megtaláljuk, melyik kulcs-érték párost kell frissítenünk. Az is előfordulhat, hogy a megoldást tároló csomópont szaturációjának végén, az MDD-tárolóban való elhelyezéskor nem ezt a csomópontot kapjuk eredményül. Ekkor a kulcshoz tartozó értéket is frissíteni kell a tüzelési gyorsítótárban.

A tüzelési gyorsítótárban való keresés, beszúrás és frissítés nem atomi művelet, ezért a gyorsítótár konzisztenciájának megőrzése érdekében az egyes műveletek elvégzésének idejére a tüzelési gyorsítótárat zárral kell ellátni, amely biztosítja a szálak közötti kölcsönös kizárást.

#### 4.2.1.4 MDD tároló változásai

Mivel az egyes részfeladatok eredményeit tároló csomópontok egy közös MDD-tárolóba kerülnek, ezért a tárolón végzett műveletek idejére is biztosítani kell, hogy egyidejűleg csak egy szál férhessen ahhoz hozzá. Ehhez a [3]-ban az MDD részgráfjainak zárolását vezetik be.

A [4] nem tér ki a tényleges zárolás implementációjának részleteire, így saját ötlet alapján készítettem azt el. Az MDDNode osztályt kiegészítettem egy locked attribútummal, amely bool típusú. A *true* érték jelzi azt, hogy a csomópont zárolva van, a *false* érték pedig azt, hogy zárolható. Ha egy csomópont zárolva van, akkor más szálak nem férhetnek hozzá. A művelet elvégzésének idejére az MDD-t lockolni kell, hogy atomikus legyen az attribútumok beállítása.

#### 4.2.1.5 Egyéb adatszerkezetek szinkronizálása

A szaturáció során több olyan adatot is szükséges tárolni, amelyek megteremtik a kapcsolatot a Petri-háló és az annak állapotterét kódoló MDD között. Ezen adatok a teljes állapottér-felderítés során szükségesek, értékük pedig a folyamat során többször is megváltozik. Ennek következtében az érintett adatstruktúrákat is módosítani kell a szekvenciális változathoz képest, hogy az adatok konzisztens voltát megőrizzük.

Az érintett adatstruktúrák:

- Kronecker mátrixok
- lokális állapotok halmaza
- globálisan elérhető állapotok halmaza

A fenti adatstruktúrák az alábbi helyeken vannak felhasználva:

- egy állapot megjelölése globálisként (Confirm függvény)

- új csomópont elhelyezése az MDD-tárolóban (CheckIn függvény)
- lokális állapotok lekérése (Locals függvény)
- annak lekérdezése, hogy a rendszer egy tüzelés hatására milyen állapotba kerül (GetTargetState függvény)

Figyelembe véve, hogy a 4 eset közül mindössze akkor kell módosítást is végezni az adatszerkezeteken, mikor új globális állapotot fedeztünk fel, ezért a hagyományos zárak helyett hatékony lehet a read-write típusúak használata az alábbi módon: a Confirm függvényben írási zárat, míg a többi 3-ban olvasási zárat helyezünk el. Ily módon a CheckIn, Locals és GetTargetState függvények párhuzamosan is futhatnak, ezzel is csökkentve a szálak költséges várakozását.

Felismerve azt, hogy ezek az adatok szorosan csak egy-egy szinthez tartoznak, a kölcsönös kizárást csak azon szálak között kell biztosítani, amelyek azonos szint adatszerkezetein dolgoznak. Hogy ennek fényében az overhead-et tovább csökkentsük, a read-write zárakból szintenként hoztunk létre példányokat.

### 4.2.2 Értékelés

Az ismertetett szaturációs algoritmus implementálása közben magam is megbizonyosodhattam arról, amit oly sokszor hallani, azaz nincs könnyű dolga annak a fejlesztőnek, aki párhuzamos program megírásába kezd. Ha szeretnénk a többprocesszoros, többmagos számítógépek rendelkezésre álló teljesítményét kihasználni meglévő algoritmusaink, programjaink párhuzamosításával, akkor rendkívül körültekintően kell eljárni.

A szaturáció párhuzamos megvalósításakor én is és a [3] szerzői is több nehézségbe és problémába ütköztünk.

A párhuzamos szaturációs algoritmus kapcsán felmerülő problémák:

- Az adatverseny elkerülése végett gyakran kell zárolni.
- A zárakat általában az MDD-tároló nagy részére helyezzük el.
- A zárak miatt a szálak ritkán futnak ténylegesen párhuzamosan, sokszor csupán a zárak feloldására várakoznak.
- Részgráf zárolás holtponthoz vezethet.
- A szaturáció ideje nagyban függ az alkalmazott architektúrától.

Az adatverseny jelenségének elkerüléséhez az MDD tárolót és a tüzelési gyorsítótárat zárolni kell. A futási idő szempontjából az az optimális, ha a szálak minél kevesebbet várakoznak, ezért a zárolást hatékonyan kell megvalósítani.

A 4.2.1.4 bevezetett csomópontok alatti részgráfok zárolásakor probléma, hogy viszonylag nagy részét is lefedhetik az MDD-nek. Mivel igen gyakori, hogy egy-egy csomópont alatti részgráfok tartalmaznak közös csomópontot egy alacsonyabb szinten, ezért ilyen esetekben a szálak kizárják egymást az élek állításának idejére. A 4.3 fejezetben megvizsgálom annak lehetőségét, hogy miként lehet a zárolt adatok körét szűkíteni.

Mivel ilyen gyakran helyezünk el zárakat, és az elhelyezett zárak gyakran fedik egymást a szálak között, ezért az egyes szálak csupán ritkán futnak valóban párhuzamosan. Tesztjeim során azt találtam, hogy a futási idő jelentős részét a szálak közötti szinkronizáció tölti ki.

A [3] szerzői az algoritmus ismertetése mellett kitérnek arra is, hogy az általuk C nyelven implementált programmal milyen sebességeredményeket tudtak elérni. Méréseiket többprocesszoros architektúrán végezték, melyekhez Slotted Ring, Round Robin, Kanban és további véletlenszerűen generált modelleket használtak fel. Azt tapasztalták, hogy a párhuzamos megvalósításuk sikeres abban az értelemben, hogy a párhuzamos program 4 processzort használva gyorsabban lefutott, mint amikor ugyanannak a programnak csupán 1 processzort biztosítottak. Párhuzamos algoritmusuk azonban mégsem tudta beteljesíteni az elvárásokat, ugyanis minden tesztelt modellre jelentősen gyorsabb maradt a szekvenciális változata a programnak. Jelen kutatásomat éppen ez indokolta, hogy megvizsgáljam, képesek vagyunk-e előnyt kovácsolni a párhuzamos feldolgozás kínálta lehetőségekből.

## 4.3 MDD szinkronizálása

A szaturáció során MDD-ket használunk a felderített állapottér tárolására. Az MDD-khez való többszörös hozzáférést nehéz biztosítani, különösen az olyan bonyolult algoritmusok esetén, mint amilyen a párhuzamos szaturáció. A [3]-ban a részgráfok zárolását javasolják az MDD konzisztenciájának megőrzésére, azonban ez költséges és időigényes művelet, ezért én ennek leváltására újfajta zárolást dolgoztam ki. A fejlesztési folyamatot, és a zárolás fejlődését a következő fejezetek mutatják be.

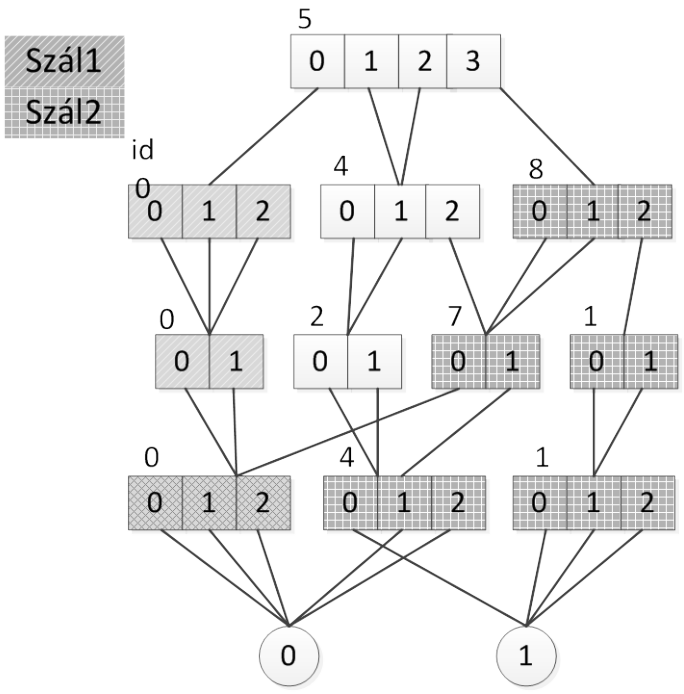
### 4.3.1 Részgráfok zárolása

A [3]-ban javasolt és a 4.2.1.4 fejezetben ismertetett részgráfzárolás megvalósításakor a nehézséget az jelenti, hogy hogyan tudjuk jelezni egy szál számára, hogy sikeresen zárolta-

e az adott csomóponttól kezdve az összes gyerek csomópontot. A zárolási mechanizmusnak mindenképp meg kell hiúsulnia, ha a levél csomópontok felé vezető út során bármely zárolandó csomópont már más szál által zárolva lett.

További részleteket nem közlök az implementációval kapcsolatban, a 10. ábra szemlélteti az algoritmus működését. Az ábrán egy MDD látható, melyet két szál manipulál párhuzamosan. Mindkét szál az MDD egy-egy csomópontja alatt elhelyezkedő részgráfot szeretné zárolni. Az egyes szálak által zárolt csomópontok különböző mintával vannak szemléltetve.

Az 1-es szál zárolja 3-as szint 0. sorszámú csomópontját és a gyerek csomópontokat. A 2-es szál zárolja a 3-as szint 8. sorszámú csomópontját és a gyerek csomópontokat. A probléma abból adódik, hogy a 2-es szál a zárolásban eljut egy darabig, majd észreveszi, hogy az 1-es szinten a 0. sorszámú csomópont már zárolva van, így azt nem tudja zárolni. Ekkor fel kell szabadítania az összes addig lefoglalt csomópontot, hiszen a zárolás meghiúsult. A hatékonyság szempontjából ez nagyon hátrányos, mivel tárolni kell, hogy mely csomópontok lettek zárolva addig, míg meg nem hiúsult és azokra visszamenőleg vissza kell állítani a locked attribútum értékét false-ra.




**10. ábra A részgráf tárolás problémájának szemléltetése**

Jól látható, hogy egy zárolási művelet elvégzéséhez az egész részfát be kell járni. A nagy adminisztrációs költség miatt az implementáció rendkívül lassúnak bizonyult, ezért

további vizsgálatnak vetettem alá a problémát és a további fejezetekben láthatók a javított iterációs fázisok.

### 4.3.2 Read-write zárolás

A részgráf-zárolás a fentebb leírt okok miatt nagy overhead-el jár, hatékonysága nem megfelelő, mivel a párhuzamos algoritmus jelentős lassulását okozta a szekvenciális változathoz képest.

A további fejlesztéseket az úgynevezett írási-olvasási zárakon alapulva végeztem el. Az ilyen zár lehetővé teszi több szál számára az egyidejű olvasást, de csak egy kizárólagos írója lehet az erőforrásnak bármely időpillanatban. Az olvasáshoz RLOCK-ot (olvasási zár), az íráshoz WLOCK-ot (írási zár) kell elhelyezni az erőforráson (a zár kompatibilitási mátrix a 11. ábrán látható).

| | **Olvasási zár** | **Írási zár** |
|---|---|---|
| **Olvasási zár** | Kompatibilis | Nem kompatibilis |
| **Írási zár** | Nem kompatibilis | Nem kompatibilis |

**11. ábra Az írási-olvasási zár kompatibilitási mátrixa**

A munkám során felismertem azt, hogy a kritikus rész az unió művelet elvégzése utáni él-átállítás, tehát ezekben az esetekben mindenképp írási zárral kell rendelkeznie a végrehajtó szálnak. Ilyen műveleteket a *SatFire*, *SatRecFire* és a *NodeSaturated* metódusokban végzünk, azokon a helyeken ahol korábban a részfát zárolni kellett, most az írási zárat kell megszereznie a szálnak. Fontos megjegyezni, hogy ezek rövid műveletek, de az algoritmus nagyobb részére hatással bírhatnak.

Az új típusú zárakat a következőképpen alkalmaztam: a korábbi zárolással megegyező hatáskörben olvasási zárakat helyeztem el, míg közvetlenül az élek állítását írási zárakkal láttam el. A zárak effajta elhelyezésével biztosítottam, hogy az MDD-hez annak módosítása pillanatában kizárólag csak 1 szál férhessen hozzá, míg módosítás híján többen is használhassák azt. Mivel a cikkben ismertetett zárolás legnagyobb problémája az volt, hogy hogyan kerülhető el a holtpont kialakulása a zárak elhelyezése közben, ezért read-write zárból csupán egyetlen egyet hoztam létre, amelyet globálissá tettem a teljes állapottér felderítésére nézve. A szálak függetlenül attól, hogy az MDD melyik szintjén

dolgoznak éppen, ugyanazt az objektumot használták szinkronizálásra. Ez igen pazarló működésnek tűnhet, azonban csak így tudtam egyszerűen elkerülni, hogy a read-write típusú zárak használatakor is szembe kelljen nézni a holtpontelkerülés jelentette hatalmas többletköltséggel. Azonban az effajta globálisan közös zárolás is hatékonyabb lehet, mint a részgráfok zárolása, mivel azokat csak egy él átállítás idejére kell alkalmazni.

### 4.3.3 Lokális szinkronizáció

A 4.3.1 és a 4.3.2 fejezetekben ismertetett megoldásoknak közös hátránya, hogy a használt zárak az adatok széles körét zárolják, ezáltal rendkívül lecsökkentik a szálak párhuzamos futásának arányát. Az előbbi esetben, amikor is teljes részgráfokat zárolunk, már a zárak elhelyezése is költséges. Az utóbbi esetben pedig az jelenti a problémát, hogy az elhelyezett zárak hatásköre nagy, így egy írási művelet, az összes többi szálat kizárja az MDD-tárolóhoz való hozzáférésből.

A korábbi 2 szinkronizációs mechanizmus helyett én végül egy 3. algoritmust fejlesztettem. Az új technika lényege, hogy teljes részgráfok zárolása helyett csupán azokat a csomópontokat zároljuk, amelyeknek valamelyik élét éppen át szeretnénk állítani.

Az eddigiekben azért alkalmaztam a részgráfok zárolását, mivel annak hiányában, ha az unió művelet végrehajtása közben egy másik szál megváltoztatja valamelyik, az unió során felhasznált csomópontot, akkor az unió műveletünk eredménye helytelen lenne az adatverseny következtében. Ez most sincs másképp, azonban a szaturációs algoritmus vizsgálata során arra a következtetésekre jutottam, hogy:

- ha megfelelő zárolási mechanizmust használunk, akkor rekurzív zárolások nélkül is biztosítani tudjuk az inkonzisztens állapotok elkerülését
- jól megtervezett szinkronizációs mechanizmusok segítségével elkerülhető, hogy nem teljesen befejezett részgráfokon végezzünk MDD műveleteket
- a szinkronizációs mechanizmusok minél inkább az adatstruktúrák mélyére süllyesztésével minimalizálni tudjuk a szinkronizációval elveszített időt.

**Helyesség**

A korábban látott részgráfzárolási algoritmus helyes [3], mivel teljesíti a helyesség következő feltételeit:

- konkurens csomópont-manipuláció megakadályozása
- a számítások konzisztenciájának megőrzése
- csomópontok szaturált állapotának tényleges elérése

Az 1. feltételt a teljes részgráfzárolási algoritmus teljesíti, mivel a csomópont manipulálás során zárat helyez el a csomóponton és az alatta levő részgráfon is, ezáltal megakadályozva nemcsak a csomópont, hanem a belőle elérhető részgráf általa kódolt információ konkurens módosítását.

Az előbbiekből következik a 2. feltétel teljesülése is, így minden csomópont egy valós részhalmazát kódolja az állapottérnek, amelyen végrehajtva a szaturációs műveletet, azok nem vezetnek ki a valós állapottérből.

A konzisztens állapottereken végrehajtva a szaturációs algoritmust, a szaturáció konvergens tulajdonsága miatt [1][3][14] a legenerált állapotteret reprezentáló döntési diagram a teljes elérhető állapotteret fogja kódolni. Ennek értelmében a részgráfzárolási technika biztosítja a 3. feltétel teljesülését is.

A read-write lock zárolás újítása, hogy részgráfok helyett a teljes döntési diagramot zárolja a módosítások idejére, viszont ezt sokkal gyorsabban teszi, mint ahogy a korábbi technika során a részgráfokat zároltuk. Mivel a teljes döntési diagramnak részhalmaza a korábbi technikában zárolt részgráf, ezért a módszer is teljesíti a helyességhez szükséges és elégséges 3 feltételt.

Ezek alapján vizsgáljuk az általam megvalósított lokális szinkronizációs algoritmus helyességét.

Az általam javasolt megoldás szakít azzal a korábbi gyakorlattal, hogy az MDD részgráfjai a szinkronizációs pontok. Helyette elegendő csak az egyes csomópontokat lokálisan zárolni a műveletvégzés idejére. Az 1. feltétel akkor teljesül, ha a csomóponton nem tud több szál módosításokat végezni (éllista módosítás). A csomóponton való zár elhelyezésével éppen az ilyen konkurens módosításokat akadályozzuk meg.

A 2. feltétel megköveteli a konzisztenciát a döntési diagramban, aminek teljesüléséhez szükséges, hogy műveletet csak konzisztens állapotot reprezentáló csomópontokon végezzünk. Mivel az általam javasolt zárolási technika a csomópontot zárolja, amíg az inkonzisztens állapotban van, ezért a szaturációt végző függvényeknek mindenképpen meg kell várniuk a konzisztens lokális állapot bekövetkeztét. Csomópontokon manipulációs műveletet a szaturációs függvényeken kívül még az unió függvénye végez, ezért meg kell akadályozni, hogy az unió műveletvégzés inkonzisztens csomóponton hajtódjon végre. Az általunk javasolt szinkronizációs mechanizmus biztosítja, hogy unió műveletet csak a csomópont-tárolóban elhelyezett csomóponton végzünk, amely elégséges feltétele, hogy az argumentum csomópontok nem módosulnak már többet. Az unió

műveleteként így kapott csomópont és az általa reprezentált megoldáshalmaz konzisztens lesz.

Tehát az algoritmusunk az új zárolási technikával is helyesen működik, ezért biztosított, hogy a modell helyes állapotterét generáljuk le.

## 4.4 CTL kiértékelő párhuzamosítása

A 3.3 fejezetnek megfelelően a szaturáció felhasználható bizonyos CTL kifejezések kiértékelésére is. Ez főként az EU operátorra igaz, ahol a teljes állapottérnek viszonylag nagy részét kell bejárnunk ahhoz, hogy eldöntsük a vizsgált kifejezés igazságtartalmát. EX operátor esetén csak az egy lépésben elérhető állapotokat vizsgáljuk, így itt nem érdemes szaturációt alkalmazni. Azt pedig a [2]-ben mutatták meg, hogy az EG esetén nem hatékony a szaturáció.

Az állapotér-felderítés során szerzett ismereteimet felhasználtam arra, hogy az EU operátor működését is párhuzamossá alakítsam, így javítva a modellellenőrzés erőforrás-kihasználását. Azért az EU operátort valósítottam meg, mivel ennek kiértékelése lehetséges szaturáció segítségével, ami a dolgozatom fő kontribúciója. Fontos viszont kiemelni, hogy az EU operátoron keresztül az EF és AG operátorok párhuzamos feldolgozása is lehetővé vált [5].

### 4.4.1 EU kifejezés kiértékelésének szinkronizálása

Az EU operátor megvalósításának bemutatása előtt megismétlem az EUSat függvény pszeudó kódját, amelyet már a 3.3.2 fejezetben bemutattam:

```
EUsat (in: P,Q:állapothalmaz, out: X: állapothalmaz)

1 ClassifyEvents(P, εU, εS)
2 X=Q
3 BackSaturate(X, εS)
4 repeat
5      Y = X
6      X = X ∪ (N_U^-1(X) ∩ (P ∪ Q))
7      if X ≠ Y then
8            BackSaturate(X, εS)
9      end if
10 until X = Y
```

A *ClassifyEvent* függvény az események csoportosítását, míg a *BackSaturate* az úgynevezett visszafelé szaturálást végzi a 3.3.2 fejezetnek megfelelően. A *BackSaturate* algoritmusa igen hasonlatos az állapottér-felderítő algoritmushoz, így csak azokat a változtatásokat mutatom be, amelyek a *BackSaturate* megvalósításához szükségesek voltak.

A párhuzamos állapottér-felderítő szaturációs algoritmuson a következő változtatások voltak szükségesek, hogy alkalmazható legyen az EU operátor kiértékelésére is:

- tüzelések csak olyan eseményekre, amelyek biztonságosak
- Kronecker mátrixok transzponáltjainak kiszámítása
- új lokális állapotok felderítésének elhagyása
- inicializáló MDD több útvonalat is tartalmazhat.

Állapottér-felderítés során az összes eseményt eltüzeljük egy adott csomóponton, hiszen a teljes állapottérre kíváncsiak vagyunk. Ezzel szemben modellellenőrzéskor csak a p-re biztonságos eseményeket engedjük meg eltüzelni, ha a vizsgált kifejezés E(pUq) alakú. Ez azért fontos, mivel így elkerülhető a metszetképzés költséges műveletének elvégzése.

A *visszafelé szaturálás* során az események inverzeit tüzeljük el. Az állapottér-felderítéskor a tüzelések lehetséges eredményeit, az állapotátmeneteket a Kronecker mátrixok tárolják. Visszafelé szaturációkor a következő-állapot függvény inverzét a Kronecker mátrixok transzponáltjai adják.

Állapottér-felderítés esetén minden tüzelés végrehajtása után ellenőrizzük, hogy vannak-e újonnan elérhető lokális állapotok az egyes szinteken. A *BackSaturate* függvényben erre nincs szükség, ugyanis a modellellenőrzés előtt minden esetben végzünk állapottér-felderítést is, így az ott felderített lokális állapotok felhasználhatóak a modellellenőrzés során is. Ennek következtében a *BackSaturate* függvényből elhagytam azt a részt, amely az addig ismeretlen állapotok felderítését végezte.

Végezetül fontos különbség mutatkozik az inicializálást illetően az állapottér-felderítő és EU kiértékelő szaturáció között. Állapottér-felderítés során az inicializált MDD csupán egy globális állapotot tartalmaz, amely megfelel a kiindulási állapotnak. Ezzel ellentétben az E(pUq) kifejezés kiértékelésekor azokból az állapotokból építjük fel a kiindulási MDD-t, ahol a q feltétel teljesül. Nyilvánvaló, hogy utóbbi esetben több globális állapotot is kódolhat az MDD. Ezt figyelembe véve és kihasználva a program párhuzamos futása már rövidebb idő után is növekedhet, ugyanis a függvények hívási lánca hamarabb szerteágazik, ezáltal hamarabb hozhatóak létre a független feladatok.

# 5 Eredmények

A szaturáció és az eredmények tárolására használt döntési diagramok szekvenciális tulajdonságai miatt nehéz feladat a párhuzamosítás, amely tulajdonsággal a [3]-ban is szembesültek. Az ott szereplő, C nyelven implementált párhuzamos szaturációs algoritmus csak a véletlenszerűen generált modellekre hozott gyorsulást a szekvenciális változathoz képest, míg a valós problémák egyszerűsített modelljeire rendre lassabb futást produkált. Munkám során ezen próbáltam javítani, azonban a probléma nehézsége miatt célom az volt, hogy a cikkben szereplő párhuzamos futási sebességeket elérjem, és némely modellre felülmúljam.

Eredményeim bemutatásához méréseket végeztem, amelyek eredményei a most következő fejezetekben olvashatók. A mérésekhez használt hardver- és szoftverkörnyezetet az 5.1 fejezet mutatja be. Mivel az állapottér-felderítés során sokkal nagyobb modelleket is képesek vagyunk bejárni, mint amekkorákra a modellellenőrzés még hatékonyan használható, ezért az ezekre vonatkozó méréseket azonos modelleken, azonban különböző nagyságrendekkel végeztem. Az állapottér-felderítés mérési eredményei az 5.2 fejezetben szerepelnek, míg a modellellenőrzésre vonatkozó adatok az 5.3 fejezetben kaptak helyet.

## 5.1 A mérési környezet

A mérések elvégzéséhez kettő számítógép állt rendelkezésemre. Ezek specifikációja a 12. ábrán látható.

| **Név** | Desktop | Server |
|---|---|---|
| **Típus** | Desktop | Sun Sunfire x4600 szerver |
| **Processzor** | 1 db Intel Q8400, 2,6 GHz, 4 mag | 4 db AMD Opteron 8000, egyenként 2,5 GHz, 2 mag |
| **Memória** | 4 Gbyte | 24 GByte |
| **Operációs rendszer** | Windows 7 Professional | VmWare ESXi, virtualizált Windows Server 2008 |
| **Futtató környezet** | .Net 4.0 x64 | .Net 4.0 x64 |

**12. ábra Méréshez használt hardverek**

Az elkészített modulokban kihasználtam a .Net nyújtotta lehetőségeket, így a szálak ütemezését a .Net rendszerbe épített ThreadPool osztály használatával valósítottam meg. A ThreadPool-t alapbeállításokkal használtam, azaz a rendszer magonként 256 szál

egyidejű létrehozását tette lehetővé [24]. A tesztelések során azt tapasztaltam, hogy ezt a rendszer ki is használja, ugyanis többször figyeltem meg, hogy egy adott pillanatban 500-nál is több szál van létrehozva. Mindazonáltal a ThreadPool használata hatékonyabb volt így is bármelyik másik kipróbált ütemezési eszköznél.

Méréseim során a vizsgált modelleket kétféleképpen is dekomponáltam a szaturáció alatt. A vizsgált 2 szintezési mód a következő volt:

- P-invariánsos szintezés: A P-invariánsos szintezés egy automatikus dekompozíció. A PetriDotNet rendszer a Petri-háló P-invariánsai alapán osztja részmodellekre a rendszert oly módon, hogy minden egyes invariáns egy-egy szintnek felel meg az állapotteret kódoló döntési diagramban.
- Manuális szintezés: Manuális szintezés esetén a felhasználónak kell megadni a modell dekompozíciójára vonatkozó információkat. Ezt a szintezést használva a dekompozíciót minden esetben úgy adtam meg, hogy az MDD minden egyes szintje azonos számú Petri-háló hely állapotát kódolja. Más tulajdonságot nem vettem figyelembe a szintezés beállításakor.

Méréseim során minden esetben a szaturáció általam elkészített párhuzamos változatát hasonlítottam a már létező szekvenciális változattal. A más eszközökkel való összehasonlítást már elvégezték korábban a szekvenciális modullal, amelynek eredményét a [25]-ben közölték. A következő fejezetekben feltüntetett mérési eredmények minden esetben 5 mérés átlagát jelentik.

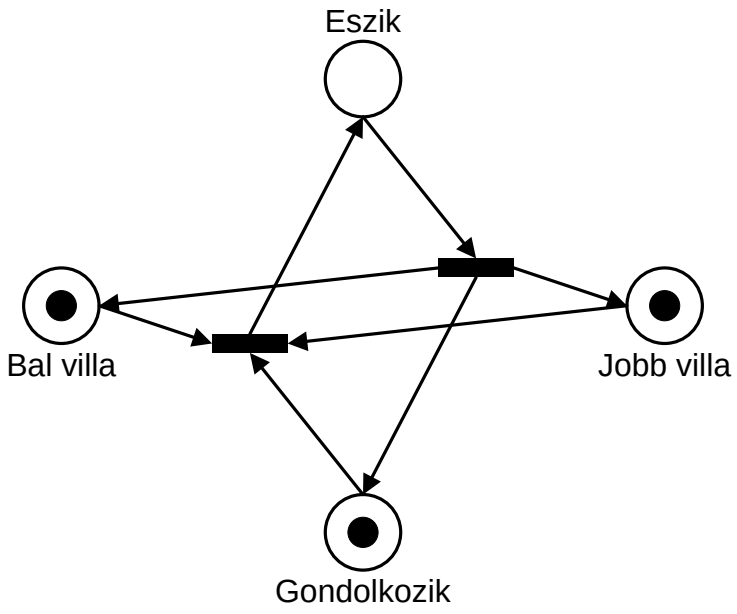
Az összehasonlításokat többféle modellre is elvégeztem, ugyanis azt találtam, hogy a modellek szerkezete szignifikánsan befolyásolja a párhuzamosítás hatékonyságát. A vizsgált modellek a következők voltak:

- Étkező filozófusok
- Réselt gyűrű
- Adaptív gyártórendszer

**Étkező filozófusok**

Az étkező filozófusok modell azt a problémát reprezentálja, amikor filozófusok ülnek egy asztal körül, és ebédelnek. A filozófusok 2 darab villát használnak evéshez, azonban az asztalon bármelyik 2 szomszédos filozófus között pontosan 1 darab villa van. Ennek következménye, hogy egymás mellett ülő filozófusok sosem ehetnek egyszerre. Ha egy filozófus nem eszik, akkor gondolkozik. Én azt az egyszerűsített változatát használom a modellnek, amikor a filozófusok egyszerre veszik fel a jobb és bal villájukat. Az étkező

filozófusok egy elterjedt benchmark jellegű modell. A rendszer egy építőeleme, egy filozófus a 13. ábrán látható.

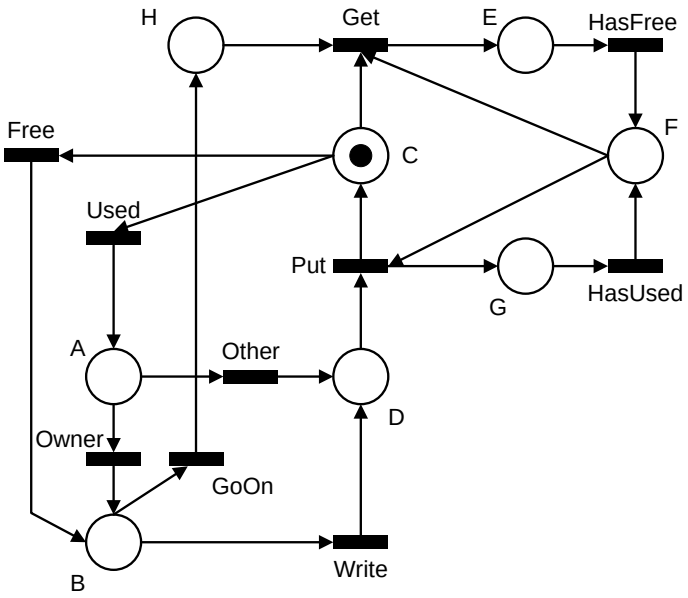



**13. ábra Étkező filozófus modell**

**Réselt gyűrű**

A réselt gyűrű egy hálózati protokoll, amelyben a résztvevő felek gyűrű topológián keresztül kommunikálnak meghatározott méretű keretek segítségével. A hálózatban keringő keretek lehetnek üresek vagy sem. Ha az egyik fél üzenetet szeretne küldeni, akkor vár egy üres keret érkezésére, és ebben helyezi el a küldendő információt. Ha az i. résztvevő az i+1. Részvevő felől egy üres keretet kap, akkor a Free tranzíció tüzel, teli keret érkezésekor pedig a Used. A protokoll egy résztvevőjének a modellje a 14. Ábrán látható, ahol az egyes helyek jelentése a következő:

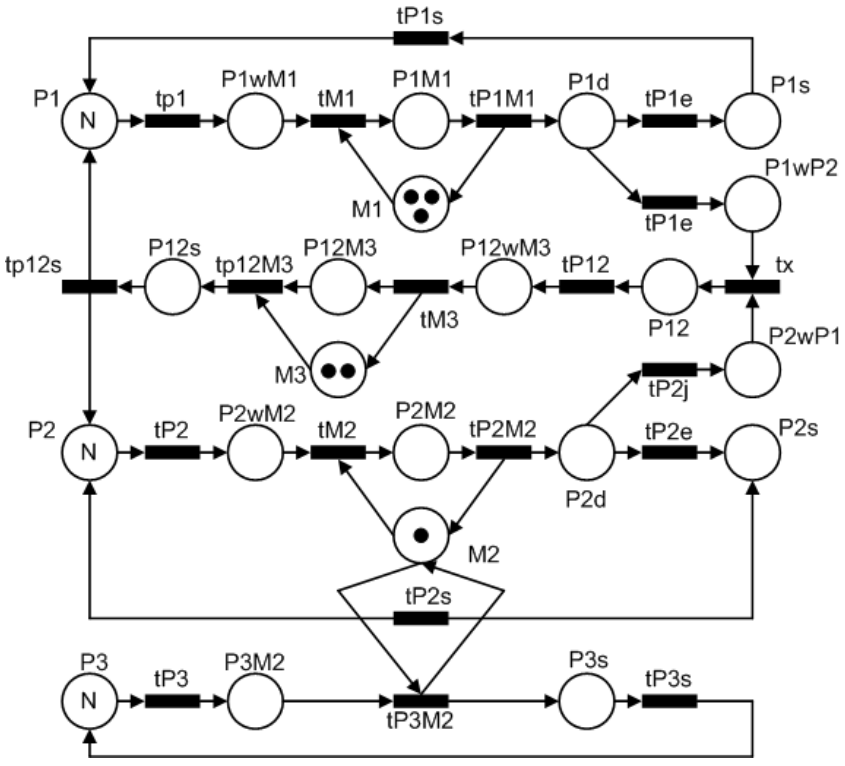
- A: foglalt keret érkezett
- B: üres keret érkezett
- C: a csomópont készen áll a következő keret fogadására
- D: a küldendő üres keret továbbítási feldolgozásra készen áll
- E: a csomópont üres keret továbbítására készen áll
- F: a küldendő keret továbbításra készen áll
- G: foglalt keret továbbításra készen áll
- H: a küldendő foglalt keret továbbítási feldolgozásra készen áll




**14. ábra A réselt gyűrű egy csomópontja**

### Adaptív gyártórendszer

Ez a modell egy termelési folyamatot ellátó rendszert modellez. A rendszer képes alkalmazkodni a bekövetkezett változásokhoz és ehhez alakítani az előállított termékek számát. A rendszer Petri-hálója a 15. ábrán látható, míg a részletes leírása a [22]-ben található meg.




**15. ábra Adaptív gyártósor modellje**

## 5.2 Állapottér-felderítés mérése

Ebben a fejezetben a párhuzamos állapottér-felderítő algoritmus hatékonyságát vizsgálom különböző modellekre.

### 5.2.1 Réselt gyűrű

Az alábbiakban a Réselt gyűrű hálózati kommunikációs protokoll állapottér-felderítésének idejét vizsgáltam különböző beállításokkal a desktop gépet használva.

A méréseket elvégeztem a PetriDotNet által kínált többféle szintezési mód segítségével. P-invariánsos esetben minden résztvevőt reprezentáló részmodellt 2 szint kódol az MDD-ben. Ekkor a lokális állapotterek szűkek lesznek. Ezzel szemben a kézi szintezést úgy állítottam be, hogy minden komponens pontosan egy szintre kerüljön, így a lokális állapotterek nagyobbak lesznek az invariáns alapú dekompozícióhoz képest.

A modell állapottere 100 komponens esetén eléri a $2{,}6 * 10^{105}$ állapotot.

A mérések során megvizsgáltam, hogy a modellek mérete (N) hogyan befolyásolja a futási sebességet. A mérések eredménye a 16. ábrán látható. Az utolsó oszlopban található Arányt a következő képlet segítségével számoltam: Arány $= \frac{\text{Párh.}}{\text{Szekv.}}$.

| Réselt gyűrű N | | | | |
|---|---|---|---|---|
| Szintezés | Méret (N) | Szekv. [s] | Párh. [s] | Arány |
| P-invariáns | 50 | 1,64 | 1,69 | 1,03 |
| | 100 | 14,66 | 13,21 | 0,90 |
| | 150 | 70,35 | 42,25 | 0,60 |
| Manuális (szintenként 8 hely) | 50 | 2,61 | 4,20 | 1,61 |
| | 100 | 22,50 | 32,41 | 1,44 |
| | 150 | 97,73 | 105,13 | 1,08 |

**16. ábra Mérési eredmények Réselt gyűrű modellre**

A táblázatban szereplő mérési eredmények alapján megfigyelhető, hogy a modell méretének növekedésével a párhuzamos algoritmus által jelentett szinkronizációs költségek hatása csökken, a párhuzamosítás hatékonysága pedig növekszik. Invariáns alapú szintezés esetén az 50 résztvevős modellnél még nem tapasztalható gyorsulás a szekvenciális változathoz képest, azonban 100-ra már 10%, 150-re pedig 40% körüli a futási időbeli gyorsulása a párhuzamos algoritmusnak. Manuálisan megadott dekompozíció esetén a párhuzamos változat nem tud gyorsabb lenni a vizsgált modellekre, csupán kezdi megközelíteni a szekvenciális változatot. Ez amiatt lehetséges,

hogy manuális szintezés esetén az állapotteret kódoló MDD kevesebb szintből épül fel, így a feldolgozást végző szálak között gyakrabban kerül sor szinkronizációra, ami mint tudjuk, igen költséges művelet. A táblázat eredményeit elemezve azt várom, hogy a résztvevők számát tovább növelve manuális szintezés esetén is gyorsabb tud lenni a párhuzamos szaturációs algoritmus.

### 5.2.2 Adaptív gyártósor

Az adaptív gyártósor modellen végzett mérések eredménye a 17. ábrán láthatók. Ennél a mérésnél is a desktop gépet használtam.

Ennél a modellnél a kézi szintezést úgy adtam meg, hogy a Petri-háló minden helye külön szintre kerüljön az MDD-ben, mivel azt találtam, hogy ez a lehető legoptimálisabb szintezés a szaturáció szempontjából.

A modell állapotterének mérete 300-as paraméterszám esetén meghaladja a $10^{25}$ állapotot.

| Adaptív gyártósor N | | | | |
|---|---|---|---|---|
| Szintezés | Méret (N) | Szekv. [s] | Párh. [s] | Arány |
| P-invariáns | 6 | 6,83 | 5,96 | 0,87 |
| | 8 | 157,32 | 140,70 | 0,89 |
| | 10 | 1879,48 | 1691,11 | 0,90 |
| Manuális (szintenként 1 hely) | 100 | 14,48 | 8,25 | 0,57 |
| | 200 | 141,38 | 71,40 | 0,50 |
| | 300 | 633,04 | 190,67 | 0,30 |

**17. ábra Mérési eredmények Adaptív gyártósor modellre**

A táblázat alapján jól látható, hogy erre a modellre minden megvizsgált esetben gyorsabb volt a párhuzamos algoritmus, mint a szekvenciális. Kiemelendő, hogy a 300-as méretű modellt manuálisan dekomponálva már 70% körüli gyorsulás érhető el a szekvenciális változathoz képest. Ez a rendkívül nagy gyorsulás azzal magyarázható, hogy az gyártósor modellje esetén az állapottér MDD-je viszonylag széles lesz, szemben a réselt gyűrűvel, ahol inkább keskenyebb lesz a döntési diagram, mivel szintenként kevesebb állapotot tárol az MDD. Az MDD ezen tulajdonsága lehetővé teszi, hogy nagyobb részfeladatokat lehessen kialakítani a párhuzamos feldolgozáshoz, aminek következtében a szálak függetlenebbül tudnak működni.

### 5.2.3 A futási idő skálázódása

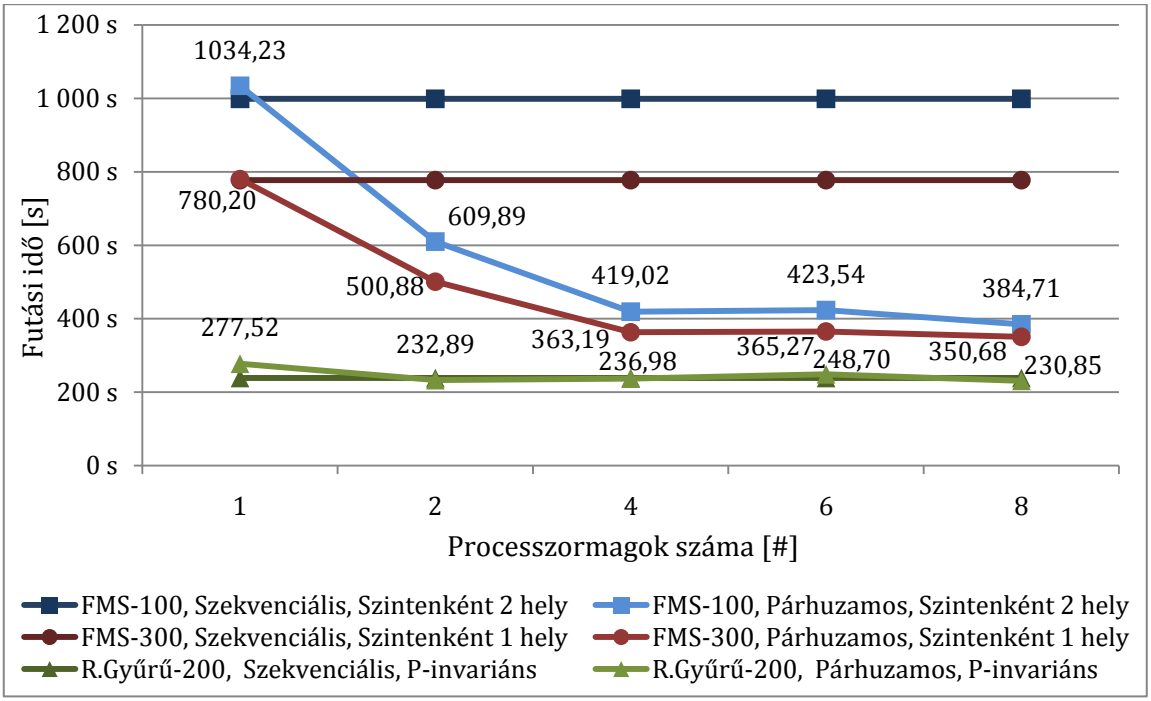
A Sun szerver lehetőségeit kihasználva méréseket végeztem az állapottér-felderítés skálázódásának vizsgálatára. A mérés célja a párhuzamos skálázódás vizsgálata volt a különböző modelleken a processzormagok számának tekintetében. A mérést a szerveren futtatott ESXi virtualizáció tette lehetővé, ugyanis annak segítségével beállítható volt, hogy a virtualizált Windows Server 2003 hány processzort használhasson.

A mérést 3 modellen futattam:

- Réselt gyűrű-200 - állapottér mérete: $8{,}4 * 10^{211}$, P-invariáns alapú szintezés
- Adaptív gyártósor-100 (FMS-100) - állapottér mérete: $2{,}7 * 10^{21}$, MDD szintenként 2-2 Petri-háló hely
- Adaptív gyártósor-300 (FMS-300) - állapottér mérete: $\sim 10^{25}$, MDD szintenként 1-1 Petri-háló hely

A nagy állapottér és döntési diagram méretek miatt választottam ezeket a modelleket, ugyanis a célom az volt, hogy a mérést ne zavarják meg a program (és a szálak) tranziens viselkedései (szálak létrehozása a futás elején, inicializálás).

A mérési eredmények a 18. ábrán láthatóak. A diagram vízszintes tengelyén az algoritmus számára biztosított processzormagok száma van feltüntetve, míg a függőleges tengelyen a futási időket jelöltem másodperces értékekben megadva. Összehasonlításként vízszintes vonalakkal feltüntettem a szekvenciális algoritmus futási idejét is az egyes modellekhez.




**18. ábra Futási idő skálázódása a processzormagok függvényében**

Az ábrán látható, hogy a vizsgált modellek esetén 1 processzormagot használva nem mutatkozik jelentős különbség a szekvenciális és párhuzamos algoritmusok teljesítményében, amiből azt következtetem, hogy sikeresen csökkentettem a zárolás által okozott többletköltséget. A gyártósor modellek esetén a magok számával folyamatosan csökken a futási idő: 2 mag esetén 10 perc, míg 8 mag esetén már csak 6 perc szükséges egy FMS-100 modell teljes állapotterének felderítéséhez. A párhuzamos szaturáció eredményeit tekintve esetünkben 4 magnál tört le az algoritmus skálázódása és onnantól nem jelentkezett jelentős gyorsulás. Ez magyarázható a sok szál következtében egyre inkább növekvő mértékű overheaddel, illetve összefüggésben áll Amdahl törvényével is, amely a problémák párhuzamosíthatóságára ad egy közelítő felső korlátot.

Az adaptív gyártósor különböző szintezéseivel végzett mérések vizsgálatakor megfigyelhető, hogy a párhuzamosíthatóság szempontjából fontos a modellek dekomponálása is, amely hatással van a szálanként bejárható lokális állapottér méretése. Nagyobb állapottér esetén a szálak nagyobb mértékben tudnak önállóan, párhuzamosan dolgozni. A modellek különféle felosztása továbbá befolyásolja az állapottér-bejárás eredményét tároló diagram struktúráját, amelyen keresztül a dekompozíció közvetve meghatározza az MDD-hez való kölcsönös hozzáférés szinkronizációs költségét is.

A réselt gyűrű esetén hosszabb monitorozást követően azt figyeltem meg, hogy a magok hozzáadásával nem tudta a program az extra teljesítményt igénybe venni. 2 mag esetén az tapasztaltam, hogy a processzorok átlagos kihasználtsága 80% körül mozgott. E fölötti magnál a teljesítmény kihasználás lineárisan csökkent. Ez alapján elmondható, hogy a réselt gyűrű állapotterének bejárása csak korlátosan párhuzamosítható a sok kauzális függőség miatt.

Habár a diagramon csak a szerveren végzett méréseim eredményét közöltem, méréseimet a 4 magos desktop gépen is elvégeztem. Azonos feltételeket biztosítva, azaz a szervernek is 4 magot adva, az eredmények a desktop gépen voltak alacsonyabbak. Tapasztalataim szerint a használt hardver architektúra és az abban rejlő párhuzamos képesség nem meglepő módon szignifikánsan befolyásolja a mérések eredményét.

## 5.3 Modellellenőrzés mérése

Ebben a fejezetben a párhuzamos modellellenőrző hatékonyságát vizsgálom különböző modellekre.

Mérési eredményeim a 19. Ábrán láthatóak.

| Szintezés | Méret (N) | Desktop | | | | Szerver | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Állapottér-generálás [s] | Párh.1 [s] | Párh. [s] | Arány | Állapottér-generálás [s] | Párh.1 [s] | Párh. [s] | Arány |
| **Étkező filozófusok** | | | CTL kifejezés: EU(eszik1=0 u eszik2=1) | | | | | | |
| P-invariáns | 50 | ~0,10 | 0,26 | 0,97 | 3,73 | ~0,15 | 0,61 | 1,01 | 1,66 |
| | 100 | ~0,11 | 1,10 | 4,16 | 3,78 | ~0,18 | 2,10 | 3,67 | 1,75 |
| | 150 | ~0,14 | 2,90 | 9,48 | 3,27 | ~0,25 | 5,64 | 9,43 | 1,67 |
| **Réselt gyűrű** | | | CTL kifejezés: EU(B1<>1\|F1<>1 u G2=1&A2=1) | | | | | | |
| P-invariáns | 25 | ~0,25 | 8,43 | 4,11 | 0,49 | ~0,60 | 12,48 | 6,25 | 0,50 |
| | 50 | ~1,75 | 48,90 | 25,80 | 0,53 | ~7,00 | 72,60 | 47,20 | 0,65 |
| | 75 | ~5,00 | 146,07 | 78,79 | 0,54 | ~12,00 | 226,05 | 157,84 | 0,70 |
| **Adaptív gyártósor** | | | CTL kifejezés: EU(M1>0 u (P1s=N&P2s=N&P3s=N)) | | | | | | |
| Manuális (szintenként 1 hely) | 5 | ~0,05 | 1,13 | 1,47 | 1,30 | ~0,10 | 1,08 | 0,89 | 0,82 |
| | 10 | ~0,10 | 3,87 | 2,52 | 0,65 | ~0,15 | 8,47 | 5,99 | 0,71 |
| | 15 | ~0,15 | 22,48 | 9,58 | 0,43 | ~0,35 | 55,76 | 28,77 | 0,52 |

**19. ábra Mérési eredmények CTL kifejezeések kiértékelésére**

Az EU kifejezés párhuzamos kiértékelő moduljának hatékonyságát megvizsgáltam több modellen végzett mérésekkel, amelyeket mind a desktop, mind a szerver gépeken elvégeztem. A vizsgált modellek és kifejezések a következők voltak:

- Étkező filozófusok: EU(eszik1=0 u eszik2=1)
- Réselt gyűrű: EU(B1<>1|F1<>1 u G2=1&A2=1)
- Adaptív gyártósor: EU(M1>0 u (P1s=N&P2s=N&P3s=N))

A 19. Ábra táblázatában feltüntettem a használt dekompozíciós stratégiát (Szintezés), a vizsgált modellek méretét (Méret), a modellellenőrzést megelőző állapottér-felderítés idejét (Állapottér-felderítés). Ezen felül a CTL kifejezések kiértékelésének ideje kapott helyett a táblázatban, amelyet kétféleképpen is lemértem. Elsőként a párhuzamos programom számára csak 1 processzormagot engedélyeztem (Párh.1), majd elvégeztem a méréseket a mindenkori rendelkezésre álló magok összességét használva is (Párh.). Ezen felül a desktop és a szerver gép esetén is összehasonlítottam a kétféle futtatás eredményét ($\text{Arány} = \frac{\text{Párh.}}{\text{Párh.1}}$).

A táblázat eredményeit tekintve látszik, hogy a CTL kifejezések kiértékelése mindig szignifikánsan több időt vett igénybe a modellek állapotterének felderítéséhez képest. Ez érthető is, ugyanis a modellellenőrzéskor a szaturáción kívül elvégezzük még az események csoportosítását, a $q$ állapotkifejezést kielégítő állapotok összegyűjtését és egyéb további lépéseket is.

Az étkező filozófusok modellen végzett mérések eredményeiből az látszik, hogy a többmagos párhuzamos változat rendre rosszabb eredményeket produkál az 1 magon futtatott változathoz képest. Ezt a modell szerkezete indokolja, ugyanis az étkező filozófusok rendszer egyes részei szorosan összefüggnek, így igen nehéz a független részfeladatok kialakítása. Ennek következtében igen magas a szálak közötti szinkronizációs költség, amely több mag, ezáltal több szál használata esetén jelentősebb. Megfigyelhető az is, hogy a modell méretének növekedésével nem tapasztalható jelentős változás a két változat sebesség arányainak tekintetében. Az eredmények vizsgálatában előremutatóan kijelenthető az is, hogy a vizsgált modellek közül ez az egyetlen, amelynél a szerver gépen mért sebességviszonyok kedvezőbbek a desktop gép méréseinél.

A réselt gyűrű esetében a párhuzamos modellellenőrző kedvezőbben képes kihasználni a rendelkezésre álló erőforrásokat, azonban a modell méretének növelésével ez a hatékonyság egyre csökken. Érdekes megfigyelés, hogy a szerver gép ugyan kétszer több processzormaggal és hatszor több memóriával rendelkezik, mégsem tud a program az erőforrásokhoz képest arányosan gyorsabb lenni (Pl. Réselt gyűrű-75 és desktop gép

esetén 46% a gyorsulás, míg a szervernél ugyan erre a modellre csak 30% körüli.) Ezt a tulajdonságot sok tényező befolyásolhatja, de legfőképpen a számítógépek eltérő hardver- és szoftverkonfigurációjával magyarázható az eltérés. A szerver gép körülbelül 4 éves konfiguráció, több AMD processzorral szerelve. Ezzel szemben a desktop gép egy korszerűbb, Intel processzorral rendelkezik, amelynél egy tokba került beépítésre a 4 darab mag, továbbá az eltérő gyártók miatt a processzorok utasításkészlete is különböző lehet. Nem elhanyagolható az sem, hogy a szerver gépen virtualizált a Windows operációs rendszer, amely jelentősen befolyásolhatja a futtatott alkalmazások teljesítményét.

Az adaptív gyártósor modelljénél két dolgot érdemes kiemelni. Az első, hogy ennél a modellnél tapasztaltam a legjelentősebb sebességgyorsulást. Az Adaptív gyártósor-15 modellnél már 57%-os a gyorsulás a desktop gépen, de még a szerveren is 48%-os gyorsulást mértem. A meglepő gyorsulás oka itt is ugyanaz, mint az állapottér-felderítésnél, vagyis az MDD sok lokális állapotot kódoló szintjei lehetővé teszik a nagymértékben független részfeladatok kialakítását, ami kedvezően hat a párhuzamosíthatóságra megfelelő hardver, azaz több rendelkezésre álló mag esetén. A másik fontos észrevétel, hogy a jelentős gyorsulást a program csak fokozatosan tudja elérni, ugyanis Adaptív gyártósor-5 esetén még lassulást is tapasztaltam a desktop gépen való mérések során. Ezt az magyarázza, hogy ennél a rendszernél a modellellenőrzés inicializációs fázisa, amely nem párhuzamosítható, nem elhanyagolható időt vesz igénybe a bonyolult CTL kifejezés miatt, így kisebb modellek esetén jobban befolyásolja a modellellenőrzéshez szükséges időt.

Méréseim során törekedtem arra, hogy megismerjem és bemutassam a párhuzamos algoritmus teljesítményét befolyásoló tényezőket, amelyek közül a következők a legfontosabbak:

- A vizsgált modell strukturális tulajdonságai
- A kialakuló döntési diagram szintjein kódolt állapotok száma és ez alapján a szintek száma
- Az alkalmazott párhuzamosításra képes hardver architektúrája

# 6 Összefoglalás

A szakdolgozat és egy korábbi TDK dolgozat [26] eredményeként egy komplett állapottér-felderítő és modellellenőrző modullal egészítettem ki a PetriDotNet modellellenőrző keretrendszert. Az elérhető funkciók közül a TDK konferenciára az állapottér-felderítést készítettem el, míg annak továbbfejlesztéseként a szakdolgozat keretein belül párhuzamos szaturációt használva a modellellenőrzés hatékonyságát is növeltem. Az elkészített modul képes a multiprocesszoros hardverek többlet erőforrásait hatékonyan kihasználni, ezáltal gyorsítva a rendszer ellenőrzésére fordított időt.

A kitűzött célokat az alábbiak szerint teljesítettem:

- Megismertem a modellellenőrzést. Áttekintettem az alkalmazási körét, és a rendelkezésre álló technikákat.
- Részletesen tanulmányoztam egy szimbolikus modellellenőrző módszert, a szaturációt, amelyet a tárhatékonysága miatt választottam.
- A szakdolgozatot megelőző TDK dolgozat keretein belül implementáltam egy modult a PetriDotNet rendszerhez, amely párhuzamosan képes az aszinkron rendszerek állapotterét felderíteni szaturációt alkalmazva.
- Az implementáció alapját szolgáló algoritmust továbbfejlesztettem új szinkronizációs mechanizmusok fejlesztésével, amelyek kevesebb erőforrást használnak, így hatékonyabb zárkezelést biztosítanak. Ezen kívül az adatszerkezeteket oly módon egészítettem ki, hogy azok is a párhuzamos algoritmus további sebességnövekedését szolgálják.
- A továbbfejlesztett párhuzamos algoritmust végül felhasználtam az EU operátort tartalmazó CTL kifejezések kiértékelésére, ily módon egy komplett modellellenőrző eszközt alkottam meg.

Az általam megvizsgált irodalmakban közölt eredmények nem számolnak be a párhuzamosítás következtében elért, szignifikáns eredményekről [3]. Ezzel szemben a mérési eredményeim alapján elmondható, hogy az implementációmmal jelentős, akár 60% körüli sebességnövekedést is sikerült elérni, ugyanakkor azokra az esetekre, ahol nem lett gyorsabb a párhuzamos algoritmus, nem maradt el jelentősen a szekvenciális változathoz képest.

Az elkezdett munka még számos továbblépési lehetőséget hordoz magában, amelyek közül az alábbiak tűnnek perspektivikusnak számomra:

- Elosztott modellellenőrző algoritmus kifejlesztése, amely munkaállomások hálózata által nyújtott erőforrásokat képes hatékonyan kihasználni.
- A párhuzamos CTL modul továbbfejlesztése, hogy a modellellenőrzési folyamat átfogóbb része legyen párhuzamosan megvalósítható, illetve további operátorok párhuzamos feldolgozásának megvalósítása.
- Az algoritmus strukturális módosítása, hogy alkalmas legyen az állapottér dekompozíciójával nyerhető lehetőségek kihasználására, így érve el jobb skálázódást több processzormag alkalmazása esetén.

## Irodalomjegyzék

[1] Ciardo G, Marmorstein R, Siminiceanu R.: The saturation algorithm for symbolic state-space exploration. International Journal on Software Tools for Technology Transfer., 8(1):4-25., 2005.

[2] Ciardo G., Simiceanu T.: Structural symbolic CTL model checking of asynchronous systems. Computer Aided Verification, LNCS 2725, 40-50, 2003.

[3] Ezekiel J, Lüttgen G, Siminiceanu R.: Can saturation be parallelised? On the parallelisation of a symbolic state-space generator. PDMC conference on Formal methods: Applications and technology., 331-346., 2006.

[4] PetriDotNet információs oldal: https://www.inf.mit.bme.hu/research/tools/petridotnet

[5] Bartha T., Csertán Gy., Gyapay Sz., Majzik I., Pataricza A., Varró D.: Formális módszerek az informatikában. Typotex Kiadó, Budapest, 2004.

[6] Murata, T.; , Petri nets: Properties, analysis and applications, Proceedings of the IEEE , vol.77, no.4, pp.541-580, Apr 1989 doi: 10.1109/5.24143

[7] Yen, Hc.: Introduction to Petri net theory. Recent Advances in Formal Languages and Applications., 2006.

[8] Edmund M. Clarke, Orna Grumberg, Doron A. Peled: Model checking MIT Press, 1999.

[9] Oriol Roig , Jordi Cortadella , Enric Pastor: Verification of Asynchronous Circuits by BDD-based Model Checking of Petri Nets, Proceedings of the 16th International Conference on Application and Theory of Petri Nets, 374-391, June 26-30, 1995

[10] Bryant, R.E.:Graph-Based Algorithms for Boolean Function Manipulation, Computers, IEEE Transactions on , vol.C-35, no.8, pp.677-691, Aug. 1986 doi: 10.1109/TC.1986.1676819

[11] C. A. Petri Kommunikation mit Automaten. Schrift des IIM Nr. 3, Institut für Instrumentelle Mathematik, Bonn, 1962.

[12] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang: Symbolic model checking: $10^{20}$ states and beyond., Information and Computation 98(2):142-170, 1992.

[13] Pataricza A.: Formális módszerek az informatikában, Typotex, Second edition, 2005.

[14] Ciardo, G., R. Marmorstein, R. Siminiceanu.: Saturationunbound. Tools and Algorithms for the Construction and Analysis of Systems 2619/2003: 379-393., 2003

[15] Ciardo, G.: Data representation and efficient solution: a decision diagram approach. Formal Methods for Performance Evaluation: 371-394, 2007

[16] Peter Buchholz, Peter Kemper: Kronecker Based Matrix Representations for Large Markov Models , Validation of Stochastic Systems, Lecture Notes in Computer Science, Volume 2925/2004, 367-376, 2004.

[17] Ciardo, G, Lüttgen G., , Siminiceanu R.: Saturation: an efficient iteration strategy for smybolic state space generation, Tools and Algorithmy for the Construction and Analysis os Systems (TACAS), LNCS 2031, 328-342., Springer-Verlag, 2001.

[18] B. Bingham, J. Bingham, F. de Paula, J. Erickson, M. Reitblatt, and G. Singh: Industrial Strength Distributed Explicit State Model Checking, International Workshop on Parallel and Distributed Methods in Verification (PDMC), 2010.

[19] U. Stern and D. L. Dill: Parallelizing the murphi verifier, International Conference on Computer Aided Verification, pp. 256–278., 1997.

[20] Barnat, Jiří - Brim, Luboš - Rockai, Petr. DiVinE Multi-Core: A Parallel LTL Model-Checker. In Automated Technology for Verification and Analysis. Berlin / Heidelberg : Springer, ISBN 978-3-540-88386-9, pp. 234-239. 2008, Seoul.

[21] Foster, Ian: Designing and Building Parallel Programs (Online) Addison-Wesley ISBN 0201575949, chapter 8 Message Passing Interface

[22] Grumberg O, Heyman T, Schuster A.: A work-efficient distributed algorithm for reachability analysis. Formal Methods in System Design., 29(2):157-175. , 2006.

[23] A. Grama, G. Karypis, V. Kumar, A. Gupta; Introduction to parallel computing, 140-141, 2003.

[24] http://msdn.microsoft.com/hu-hu/default(en-us).aspx

[25] Darvas Dániel: Szaturáció alapú automatikus modellellenőrző fejlesztése aszinkron rendszerekhez, TDK dolgozat, 2010

[26] Jámbor Attila, Szabó Tamás: Aszinkron rendszerek párhuzamos modellellenőrzése

[27] E. Felt, G. York, R. Brayton, and A. Sangiovanni-Vincentelli: Dynamic variable reordering for BDD minimization, Proc. European Design Automation Conf. pp. 130–135., 1993.