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

Aszinkron rendszerek modellellenőrzése párhuzamos technikákkal

TDK dolgozat

Készítette:
Jámbor Attila és Szabó Tamás

Konzulensek:
Horváth Ákos és Vörös András

2010.

Tartalomjegyzék

Aszinkron rendszerek modellellenőrzése párhuzamos technikákkal ... 1

1 Bevezetés ... 4

2 Háttérismeretek ... 6

2.1 Aszinkron rendszerek modellellenőrzése ... 6

2.1.1 A formális módszerek szerepe az informatikai rendszerek tervezésében ... 6

2.1.2 Korábbi állapottér-felderítő algoritmusok ... 8

2.2 Petri hálók ... 9

2.2.1 A Petri hálók eredete ... 9

2.2.2 A Petri hálókról általában ... 9

2.2.3 A Petri hálók struktúrája, definíció ... 10

2.2.4 A Petri hálók dinamikus viselkedése ... 11

2.2.5 Tüzelési szekvencia ... 12

2.2.6 Petri hálók invariánsai ... 12

2.3 Döntési diagramok és alkalmazásaik a modellellenőrzésben ... 13

2.3.1 Bináris döntési diagramok ... 13

2.3.2 Többértékű döntési diagramok ... 16

2.4 Állapottér-bejárás ... 17

2.4.1 Explicit módszerek az állapottér felderítésre ... 18

2.4.2 Szimbolikus állapottér-felderítés ... 18

2.5 A szaturációs algoritmus ... 19

2.5.1 Az állapottér dekompozíciója ... 20

2.5.2 A következő állapot függvény ... 21

2.5.3 A helyben frissítés módszere (in-place update) ... 21

2.5.4 Rekurzív mélységi állapottér-felderítés ... 22

2.5.5 Szaturáció ... 22

3 Párhuzamos modellellenőrzés ... 23

3.1 Korábbi párhuzamos szimbolikus algoritmusok ... 23

3.2 Párhuzamos szaturációs algoritmus ... 24

3.2.1 Az algoritmus áttekintése ... 24

3.2.2 Az algoritmus bemutatása, megvalósítása ... 25

3.2.3 Az algoritmus főbb függvényei ... 28

3.2.4 Példa ... 30

3.2.5 Értékelés ... 33

4 Fejlesztéseink a párhuzamos algoritmuson ... 34

4.1 Az MDD szinkronizálása ... 34

4.1.1 Részgráfok zárolása ... 35
4.1.2 Read-write lock megvalósítás ... 36
4.1.3 Lokális szinkronizációs mechanizmus ... 36
4.2 Heurisztikán alapuló tranzíció-előretüzelés ... 38
4.2.1 Az előretüzelés működésének bemutatása ... 38
4.2.2 Az eltüzelendő tranzíció kiválasztásához használt heurisztikák ... 40
4.2.3 Implementációs részletek ... 40
5 Mérési eredmények ... 42
5.1 A mérési környezet ... 42
5.2 Slotted Ring ... 43
5.3 FMS ... 44
5.4 Kanban ... 45
5.5 A futási idő skálázódása ... 46
6 Összefoglalás, értékelés ... 49
7 Irodalomjegyzék ... 50
Függelék A ... 52
Függelék B ... 58

1 Bevezetés

Az informatikai rendszerek elterjedésével párhuzamosan egyre inkább nő az általuk nyújtott szolgáltatások minőségével szemben támasztott igény. Elsősorban itt a magas fokú tervezési helyesség igazolhatóságára kell gondolni. A mai ipari alkalmazások komplexitása mellett már nem lehetséges az, hogy a kvázi ad-hoc módon megírt tesztesetek alapján végezzük el a hibakeresést. Korábban ez csak a speciális, missziókritikus rendszerek (például vasúti forgalomirányító berendezések, repülőgép vezérlés, orvosi elektronika, nukleáris erőművek vezérlőberendezése) esetén volt követelmény, de az internet elterjedésével ez már kiterjed a mindennapok informatikai szolgáltatásaira is (például banki szolgáltatások, online vásárlás).

Az *aszinkron rendszerek* helyes működésének igazolhatósága az elosztottság miatt még nehezebb, mivel az egymástól függetlenül működő komponensek közötti időviszony nemdeterminisztikus. Ezen felül a klasszikus kézi hibakeresés további hátránya, hogy a rendszerben nagy valószínűséggel maradnak fel nem derített részek, a kimerítő hibakeresés csak valamilyen automatizált módszerrel lehetséges.

A rendszer verifikációja és validációja

Az imént említett problémák orvoslására sokféle szoftver- és rendszerfejlesztési metodika született már, melyek többé-kevésbé garantálni tudják a kiinduló specifikáció és az implementációs modellek ekvivalenciáját. Napjainkban azonban egyre inkább nő a szolgáltatásbiztonság iránti igény, amely magával vonta a formális módszerek alkalmazását. Ilyen esetben a rendszer működését valamilyen precíz matematikai formulával írjuk le (Petri háló [5][6][7][8], formálisan helyes UML szekvencia diagram [29], állapotgépek [31]). Ezen alapulva célunk a rendszer helyes működésének igazolása. Az ellenőrzés során két alapvető megközelítést illetve azok kombinációját lehet alkalmazni: *verifikációt* és *validációt*. A verifikáció elvégzéséhez többféle módszer terjedt el, többek között ilyen a tesztelés, szimuláció, modellellenőrzés és a tételbizonyítás.

Jelen dolgozat a modellellenőrzés témakörével foglalkozik bővebben. A *modellellenőrzés* során a rendszer egy véges modelljén bizonyítjuk be, hogy a megkívánt tulajdonság teljesül-e, például párhuzamosan dolgozó komponensek esetén előfordulhat-e, hogy a rendszer holtpontra jut, közösen használt erőforrások esetén felléphet-e kiéheztetés. Ehhez szükség van először a rendszer elérhető állapotainak felderítésére illetve tárolására, csak ez után lehetséges a rendszerrel szemben támasztott követelmények ellenőrzése. A jelenlegi iparban használt alkalmazások állapottere azonban olyan nagy, hogy rendkívül kifinomult és komplex algoritmusokra van szükség a feladatok megoldásához [30].

A rendszer állapotterének felderítése

A formális módszereken alapuló állapottér-felderítés két alapvető nehézsége:

- belátható időn belül eredményre jusson lehetetlen méretű erőforrások hiányában is
- a tárigényre is elérhető legyen valamilyen felső korlát

A futási időre megoldást jelenthetnek a párhuzamos modellellenőrző megoldások. Ez abból a szempontból is érthető igény, hogy a mai hardverek nagy része már több processzorral vagy többmagos processzorral rendelkezik. A munkánk során mi is egy párhuzamos algoritmust implementáltunk. Az elkészült algoritmust integráltuk a Budapesti Műszaki és Gazdaságtudományi Egyetem Méréstechnika és Információs Rendszerek Tanszékén fejlesztett *PetriDotNet* [1] modellellenőrző keretrendszerbe. Ez az eszköz *Petri háló*kat használ a rendszer működésének modellezésére.

A tárigény orvoslására alapvetően két módszer terjedt el az évek során:

- Az *explicit technikák* a rendszer állapotait és az átmeneti 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. Ehhez valamilyen kompakt, tárterület szempontjából hatékonyabb adatstruktúrát használnak, például *döntési diagramokat* (decision diagram) [5], hash táblát.

Az általunk implementált szaturációs algoritmus is egy szimbolikus technika, amely kvázi redukált döntési diagramokat alkalmaz az elérhető állapotok tárolására. Alkalmazásával például sikeresen felderítettük a *Slotted Ring* hálózati protokoll [23] állapotterét több különböző résztvevőszám esetén is. Említésre méltó, hogy 15 résztvevő esetén az állapottér mérete már eléri a 10^{15} állapotot, amelynek tárolása explicit módon nem lehetséges a jelenlegi memóriakorlátok miatt, azonban szimbolikus technikákkal ez a probléma kezelhető.

A kutatási célok, a dolgozat felépítése

Az irodalom áttekintése és az elvégzett munka a dolgozatban az alábbiak szerint tagolódik:

- A 2. fejezetben rövid áttekintés található mindazokról a formális módszerekhez kapcsolódó ismeretekről, amelyek szükségesek a később leírt szaturációs algoritmus megértéséhez; döntési diagramok, Petri hálók és állapottér-felderítés.
- A 3. fejezetben található a szaturációs algoritmus részletes leírása, amely alapjául szolgált a párhuzamos változatnak.
- A párhuzamos algoritmushoz kapcsolódó fejlesztésekről és egy kísérleti jelleggel implementált tranzíció-előretüzelési algoritmusról szól a 4. fejezet.
- Eredményeinket mérésekkel igazoltuk, amelyben az algoritmus szekvenciális változatát összehasonlítottunk a párhuzamos változattal. Ezek az eredmények az 5. fejezetben megtekinthetők.
- A 6. fejezetben egy összefoglalás után szót ejtünk a munka lehetséges folytatási irányairól is.

2 Háttérismeretek

A mai szoftver és hardver rendszerek már messze meghaladják azt a komplexitásbeli határt, amelyet a fejlesztő mérnökök támogatás nélkül képesek átlátni. A szolgáltatásbiztonság iránti igény növekedésével párhuzamosan a formális módszerek iránti igény is jelentősen megnőtt az elmúlt években. A matematikai módszerek alkalmazásának előnye éppen abban rejlik, hogy – legalábbis az alkalmazott modellek érvényességi körén belül – ezt a helyességet 100% valószínűséggel bizonyítják. A tesztelés és szimuláció által nyújtott bizonyosság ettől jelentősen elmarad. Az internet elterjedésével egyre több elosztott alkalmazás jelenik meg, gondolhatunk itt például egy banki alkalmazásra, ahol az egyes fiókokat össze kell kapcsolni egymással és az egyes ügyfelek adatai például csak a hozzájuk legközelebb eső fiók adatbázisában vannak tárolva.

A 2.1 fejezet áttekintést nyújt a különböző verifikációs technikákról illetve az állapottér-felderítésre szolgáló korábbi megoldásokról (2.4 fejezet). A *PetriDotNet* modellellenőrző keretrendszer a Petri hálókat alkalmazza a rendszer modellezésére, ezért a 2.2 fejezet a szükséges háttérismereteket tartalmazza a Petri hálókról.

A 2.3 fejezet mutatja be a döntési diagramokat, amelynek a 2.5 fejezetben ismertetett szaturációs algoritmusban is fontos szerepe van.

2.1 Aszinkron rendszerek modellellenőrzése

Az aszinkron rendszerek több független komponens együttműködéséből állnak össze. A rendszer egyes komponensei egymással valamilyen meghatározott protokoll szerint, csatornákon keresztül kommunikálnak. A kommunikáció kettős célt szolgál, egyrészt szinkronizációt az egyes komponensek között, másrészt magának a tényleges adatnak az átvitelét. Ilyen rendszerekre példa egy repülőgép vezérlőszoftvere vagy egy dialízis gép, amelyek a környezetüktől többféle mérőműszeren keresztül kapnak információkat, és az egyes komponensek egymástól függetlenül működnek. Ezen rendszerek hibáinak felderítése az elosztottság miatt különösen nehéz feladat.

2.1.1 A formális módszerek szerepe az informatikai rendszerek tervezésében

Az informatikai rendszerek tervezése során az egyik végcél a szolgáltatások helyességének bizonyítása. Az ellenőrzés során két alapvető megközelítés, illetve a kettő kombinációja terjedt el:

- *Validáció* esetén a rendszer külső körülményeknek való megfelelőségét vizsgáljuk, azaz arra kérdésre adjuk meg a választ, hogy „jó rendszert építünk, építettünk-e?”. Más szavakkal a felhasznált eszközkészlet által nem garantált tulajdonság teljesülését ellenőrizzük ilyenkor. Tipikus példa erre az esetre annak eldöntése, hogy a program nem juthat-e holtpontra a futása során.
- Ezzel szemben a *verifikáció* valamilyen matematikai formalizmuson alapuló bizonyítása annak, hogy a kiindulási specifikáció és az egyes implementációs fázisok után keletkező modellek ekvivalensek egymással. A folyamat a „jól építjük-e a rendszert” kérdésre válaszol képletesen.

A komplex rendszerek validációjának és verifikációjának eszközei többek között:

- tesztelés
- szimuláció

- tételbizonyítás
- modellellenőrzés.

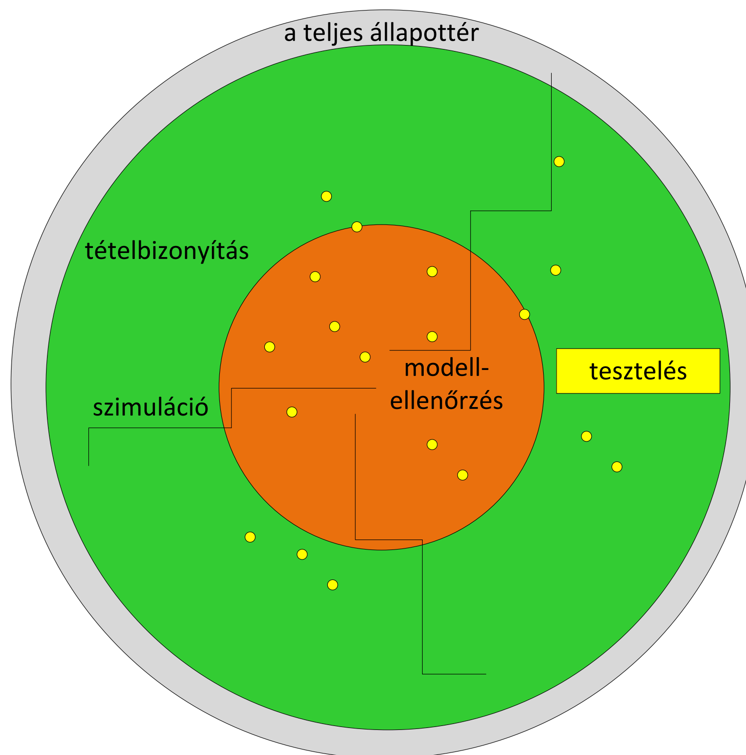
A *tesztelés*t a konkrét rendszeren végzik, ennek következménye, hogy sok esetben költséges a megtalált tervezési hibák utólagos javítása.

Ezzel szemben a *szimulációt* általában magasabb absztrakciós szinten, a rendszer valamilyen modelljén végzik. Ezen módszerek alkalmazása a rendszer működésének kimerítő ellenőrzésére meglehetősen drága, így korlátozott a használhatóságuk a biztonságkritikus alkalmazások, protokollok esetén.

A *tételbizonyítás*sal [5] axiómákból kiindulva valamilyenfajta matematikai logika alkalmazásával lehetséges a rendszer helyes működésének igazolása. Ez az eljárás egy rendkívül időigényes folyamat, amely nagy szaktudást is igényel, így csak olyan rendszerek esetén alkalmazzák, ahol ez igazán szükséges. Jellemzően ilyen alkalmazási területek a biztonsági protokollok helyes működésének igazolása. Egy ilyen folyamat hetekig, hónapokig is eltarthat. Fontos azonban megemlíteni azt, hogy a rendszerrel szemben támasztott, automatikusan verifikálható követelmények tere véges, például nem létezik olyan algoritmus, mely képes bizonyítani, hogy egy program futása során valamikor biztosan fog-e terminálódni. A tételbizonyítással végtelen állapotterű rendszerek esetén is lehet érvelni valamely tulajdonság igazságértékével kapcsolatban, illetve ezen a területen is léteznek már automatizált megoldások, például a FOL (First Order Logic) megoldók [32].

A *modellellenőrzés* egy olyan technika, amely a véges állapotterű rendszerek esetén tudja vizsgálni a követelmények teljesülését vagy nem teljesülését. A modell végessége miatt a folyamat automatizálható és kellő méretű erőforrások biztosítása mellett a modellellenőrző algoritmus biztosan terminálódni fog. A problémát az jelenti, hogy a mai ipari alkalmazások olyan méretű állapottérrel rendelkeznek, melyek kimerítő bejárása nehéz feladat, mivel az erőforrások szükséges mérete nem, vagy csak nehezen biztosítható. Ugyanakkor a modellellenőrzés hasznunkra lehet hibák felderítésére végtelen állapotterű rendszerek esetén is, ilyen esetekben az állapottér méretét kell valamilyen módon végesre korlátozni. Például egy végtelen üzenetsort tartalmazó alkalmazás esetén célravezető lehet az is, ha csak valamilyen nagy, de véges számú üzenetsort alkalmazunk a hibakeresés során, tehát ebben az esetben a modell egy korlátos megfelelőjén vizsgáljuk a követelményeket.

A négy ismertetett módszert szemlélteti az 1. ábra. A rendszer lehetséges állapotai közül a tesztelés csak 1-1 jól elkülönített esetet fed le (az ábrán a kis pontoknak felel meg). A rendszer helyes működésének bizonyíthatósága így jól látható módon meglehetősen időigényes és költséges feladat. A szimuláció egy állapotból kiindulva valamilyen görbe mentén végigmegy az elérhető állapotokon, így alkalmazhatósága ennek is korlátozott (az ábrán törött vonal szemlélteti). A modellellenőrzés ezekkel szemben a rendszer állapotterének egy korlátos részén vizsgálja meg a követelmények teljesülését vagy nem teljesülését, habár igaz, hogy csak egy korlátos részét ellenőrzi, de manapság egyre többen a small scope hypothesisre [27] alapozva azt mondják, hogy így is jól lehet következtetni a teljes rendszer hibamentességére (az ábrán narancssárga színű kör szemlélteti). A tételbizonyítás módszere még ennél is nagyobb állapotteret fed le, de alkalmazása meglehetősen költséges és időigényes feladat, amely speciális szaktudást is igényel (az ábrán zöld színű kör szemlélteti) és általában csak nagyon indokolt esetben használják nagyméretű rendszereknél (lásd a biztonsági szabványnak számító *common criteria* legmagasabb EAL6 és EAL7 szintű megfelelőségnek a bizonyítására [28]).



1. ábra: tesztelés, szimuláció, modellellenőrzés és tételbizonyítás ellenőrzése

Mindezek alapján elmondható, hogy a legoptimálisabb megoldás a különféle technikák együttes alkalmazása, de ez a legtöbb esetben nem megoldható. A modellellenőrzés erőssége pont abban rejlik, hogy automatizált módon képes elfogadható bizonyossággal ellenőrizni a rendszer működését.

2.1.2 Korábbi állapottér-felderítő algoritmusok

A legtöbb formális verifikációs eszköz azon alapul, hogy a rendszer modellje valamilyen magas szintű formalizmussal le legyen írva; Petri háló, UML modell, szekvenciadiagram, időzítési diagram. Az elérhető állapotok halmazát minden esetben először fel kell deríteni, hiszen csak ez után van lehetőség „kérdéseket feltenni" a rendszer működésével kapcsolatban. A *PetriDotNet* modellellenőrző a rendszer Petri háló alapú reprezentációján tudja ellenőrizni a követelményeket, így a 2.2 fejezetben rövid áttekintés található a Petri hálókról.

Sajnos az állapottér-felderítő algoritmusok használhatóságának gátat szab az állapottér robbanás problémája (*state explosion* problem). Ahogy nő a rendszer komplexitása, úgy válik kezelhetetlenné az állapottér felderítését végző algoritmus futási ideje és az állapottér tárolásához szükséges tárterület. Ezen problémák orvoslására számos irányban indultak el a kutatók. Az egyes megoldások az állapottér tárolásának módjában különíthetők el leginkább. Ezek alapján megkülönböztetünk:

- explicit
- szimbolikus

technikákat.

Explicit technikák esetén volt olyan kutatási irány [24], melyben azt használták ki, hogy az állapottérben fellelhető bizonyos fokú szimmetria esetén az eredeti állapotok halmazának

csak valamilyen valódi részhalmazát kellett eltárolni, azonban a kimerítő keresés lehetősége továbbra is megmaradt.

A szimbolikus technikák a döntési diagramokkal [3][4][5] a hatékonyabb tárolás miatt már nagy állapottér mérettel is meg tudnak birkózni, de a memória és futási időbeli korlátok itt is jelentkeznek. Ezen a területen is számos módszerrel próbáltak javítani az algoritmusokon a kutatók. Említésre méltó például az az ötlet, hogy az iterációk számát próbáljuk meg csökkenteni, melyen belül elérhető a rendszer összes állapota. Ehhez egy statikus elemzés alapján a tranzíciók sorrendezhetőek, így növelve az esélyét annak, hogy új állapotot derítünk fel az egymást követő iterációk során.

A másik említésre méltó irány az algoritmusok párhuzamosítására vonatkozik. Ez abból a szempontból is jogos igény, hogy a ma kapható hardverek nagy része már amúgy is több processzorral vagy többmagos processzorral rendelkezik, így érdemes kihasználni a többlet erőforrásokat. Az ötlet hátránya, hogy az állapottér felderítés meglehetősen szekvenciális feladat, így rendkívül összetett algoritmusokat kell kidolgozni a feladat megoldásához. Külön nehézséget jelent a döntési diagramokon való párhuzamos műveletvégzés miatt a szálak közötti szinkronizáció megoldása.

2.2 Petri hálók

Jelen fejezet a Petri hálók bemutatásával foglalkozik, azonban csak azokra a definíciókra és fogalmakra szorítkozik, melyek a dolgozat témaköréhez szorosan kapcsolódnak.

2.2.1 A Petri hálók eredete

A Petri hálók alapjait Carl Adam Petri nevű, német matematikus dolgozta ki 1939-ben. Eredetileg kémiai folyamatok leírására szánta, a matematikai alapjait doktori disszertációjában dolgozta ki 1962-ben [9].

2.2.2 A Petri hálókról általában

A Petri hálók a rendszermodellezés és –analízis széles körében használható leíró eszközök. Legfontosabb előnyük más formális módszerekkel szemben, hogy egyidejűleg grafikus és matematikai reprezentációt is definiálnak. Ez magában hordozza a könnyű kezelhetőséget, átláthatóságot a formális modellek matematikai korrektségével együtt.

Petri hálók használatával

- konkurens
- aszinkron
- elosztott
- párhuzamos
- nemdeterminisztikus és/vagy sztochasztikus

rendszerek viselkedését lehet modellezni. A Petri hálók lehetőséget nyújtanak a rendszer strukturális tulajdonságainak vizsgálatára is, ami szintén előny az alacsony szintű modellezési nyelvekhez képest.

A Petri hálók a kiforrott matematikai háttér miatt igen hatékony analízis eszközöket kínálnak, fontos azonban megjegyezni, hogy a dominánsan mindent struktúrával kifejező szemlélet miatt egy egyszerű feladat leírása is nagyméretű Petri hálót eredményezhet. Erre megoldást nyújthat valamilyen kiterjesztett Petri háló alkalmazása, többek között ilyen a színezett, időzített vagy tiltóélekkel kiegészített Petri háló.

2.2.3 A Petri hálók struktúrája, definíció

A Petri háló struktúráját tekintve egy irányított, súlyozott élű páros gráf. A gráfban két fajta csomópont lehet: hely *($p \in P$)* és tranzíció *($t \in T$)*. A helyeket a Petri hálóban körrel ábrázoljuk, a tranzíciókat téglalappal. Az irányított élek mehetnek helyből tranzícióba vagy tranzícióból helybe (páros gráf). Ezek alapján megadjuk a Petri hálók formális definícióját:

Definíció: A Petri háló egy $PN = (P, T, F)$ 3-asnak felel meg, ahol:

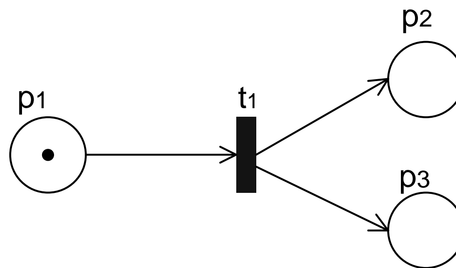
$P = \{p_1, p_2, \ldots, p_n\}$ az állapotok véges számú halmaza,

$T = \{t_1, t_2, \ldots, t_n\}$ a tranzíciók véges számú halmaza,

$F \subseteq (P \times T) \cup (T \times P)$, itt F az élek véges számú halmaza, az × művelet a halmazok közötti Descartes szorzatot jelöli,

$P \cap T = \emptyset$ *és* $P \cup T \neq \emptyset$.

Az 2. ábrán látható egy egyszerű Petri háló, melyben p_1, p_2 és p_3 helyek, t_1 pedig tranzíció.



2. ábra: egyszerű Petri háló

Definíció: Egy $n \in (P \cup T)$ csomópont $\bullet n$ ősei és $n\bullet$ utódai a következőképpen definiálhatók:

- 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: $t\bullet = \{p|(t, p) \in E\}$

Az 1-es ábrán látható példára $\bullet p_1 = \{\}$, $\bullet p_2 = \{t_1\}$ és $\bullet p_3 = \{t_1\}$.

Definíció: Egy $t \in T$ forrás tranzíció egy bemenő hely nélküli tranzíció, azaz $\bullet t = \emptyset$. Egy $t \in T$ nyelő tranzíció egy kimenő hely nélküli tranzíció, azaz $t\bullet = \emptyset$.

Az állapotváltozók szerepét az ún. token tölti be (grafikusan ezt a hely körébe rajzolt ponttal jelezzük). Egy hely állapota a benne levő tokenek számával egyezik meg. A háló állapota egy token eloszlással jellemezhető, $\overline{M_0}$*:* $P \rightarrow N$ leképezés (innentől $\bar{A}$ fogja jelenteni az A vektort), ahol N a természetes számok halmazát jelöli. $\overline{M_0}$ tehát egy $|P|$ elemű vektor.

Az élekhez ugyanakkor súlyt is rendelhetünk, mely egy pozitív természetes szám lehet. Ez annak felel meg mintha annyi egyszeres él futna párhuzamosan az adott két csomópont között a hálóban. Az egyszeres súlyokat nem szokás feltüntetni, a többszörös súlyokat az élre írjuk rá. Formálisan a súlyfüggvény a $W: E \rightarrow N^+$ leképezés.

2.2.4 A Petri hálók dinamikus viselkedése

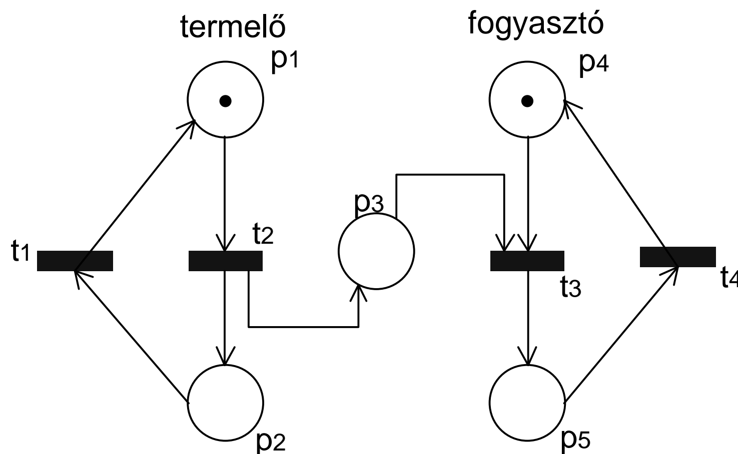
A háló állapotváltozása tranzíciók tüzelésével történhet. Egy tranzíció akkor engedélyezett, ha minden bemeneti helyen elégséges számú token van. Minden egyszeres él egy tokent tud szállítani. Formálisan a tranzíció engedélyezettségének feltétele:

$\forall p \in \bullet t : m_p \geq w^-(p, t)$, ahol •t jelöli a t tranzíció bemenő helyeit, m_p a tokeneloszlás vektor p-edik komponense, $w^-(p, t)$ a p-ből t-be vezető él súlya.

Ha a tranzíció engedélyezett, akkor lehet (nemdeterminisztikus működés, „fire at will" tulajdonság), hogy eltüzelődik, ebben az esetben a bemeneti helyekről elvesszük a tokeneket, a kimeneti helyekre pedig kirakjuk azokat. Ilyenkor egy új tokeneloszlás alakul ki, azaz a rendszer egy új állapotba kerül. Vegyük észre azt, hogy a tüzelés a $[0,\infty)$ időintervallumban bármikor bekövetkezhet, a következő tüzelés valamilyen logikai idővel később fog csak megtörténni. Formálisan egy tranzíció tüzelésének menete:

- elveszünk $w^-(p, t)$ tokent a $p \in \bullet t$ bemeneti helyekről
- kiteszünk $w^+(t, p)$ tokent a $p \in t\bullet$ kimeneti helyekre

A 3. ábra egy termelő-fogyasztó rendszer működését mutatja Petri hálóval modellezve (t_1 – t_4 tranzíciók, p_1 – p_4 helyeket jelölnek). A termelés itt t_2 tranzíció tüzelésében mutatkozik meg, ilyenkor p_2 és p_3 helyekre is kerül 1-1 token, a fogyasztás pedig t_3 tüzelésével történik meg, nyilván ehhez mind a p_3, mind a p_4 helyeken kell lennie legalább 1-1 tokennek. A kezdeti tokeneloszlás a $[p_1,p_2,p_3,p_4,p_5] = [1,0,0,1,0]$ vektorral írható le.



3. ábra: termelő-fogyasztó rendszer modellje Petri hálóval

Abban az esetben, ha több tranzíció is engedélyezett egy időben, nem definiált az, hogy melyik fog tüzelni (ha egyáltalán tüzel bármelyik is). E miatt a logikai működés szempontjából kiemelendő, hogy az egy-egy tüzelés-sorozatnak az állapottérben egy-egy trajektória felel meg. A tüzelések között fennálló versenyhelyzet eredménye a lehetséges trajektóriák közötti véletlen választás, így a Petri hálók jellegzetesen nemdeterminisztikus automaták.

Egy tüzelés hatására kialakuló új állapotot leírhatunk az $\overline{M}' = \overline{M} + \overline{\overline{W}}^T \cdot \bar{e}_t$ *módon is* ($\overline{\overline{A}}$ jelenti innentől A mátrixot), ahol $\bar{e}_t$ a t tranzíciónak megfelelő egységvektor. $\overline{\overline{W}}$ súlyozott szomszédossági mátrixot jelöl: $\overline{\overline{W}} = [w(p, t)]$, melynek dimenziója $|P| \times |T|$, ahol

$$w(p,t) = \begin{cases} w^+(p,t) - w^-(p,t), ha\ (p,t) \in E \\ 0, ha\ (p,t) \notin E \end{cases}$$

2.2.5 Tüzelési szekvencia

Az állapotátmenetek egymást követő tüzelések útján valósulnak meg. A tüzelések ilyen sorozatát tüzelési szekvenciának hívjuk. A $\vec{\sigma} = \langle M_{i0} t_{i1} M_{i1} \dots t_{in} M_{in} \rangle$ állapotátmenet sorozatot (melyet szokás $\vec{\sigma} = \langle t_{i1} \dots t_{in} \rangle$ formában is jelölni) tüzelési szekvenciának nevezzük, ha az abban szereplő összes tranzíció kielégíti a megfelelő tüzelési szabályt. Az $\langle M_{i0} M_{i1} \dots M_{in} \rangle$ állapotsorozatot az M_{i0} állapotból az M_{in} állapotba vezető trajektóriának hívjuk és az M_{in} állapotot az M_{i0}-ból elérhetőnek mondjuk a $\vec{\sigma}$ tüzelési szekvencia által, amit $M_{i0}[\vec{\sigma} > M_{in}$ formában jelölünk.

2.2.6 Petri hálók invariánsai

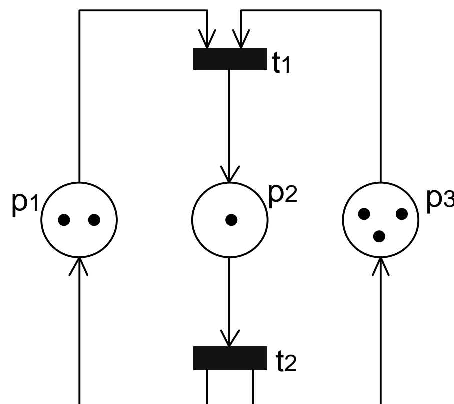
Az invariáns tulajdonságok azt fogalmazzák meg, hogy a rendszer állapottere helyes működés esetén nem alakul tetszőlegesen, ezek alapján megkülönböztetünk P- és T-invariánsokat.

2.2.6.1 T-invariánsok

Definíció: A t-invariánsok működését egy $M_0[\vec{\sigma}_T > M_0$ sorozattal írhatjuk le. Az olyan $\vec{\sigma}_T$ tüzelési szekvenciákat, melyek a tüzelési szekvencia végrehajtása után visszajuttatják a rendszert a kezdeti M_0 állapotba, azaz az állapottérben egy ciklust írnak le, T-invariánsnak nevezzük.

Más szavakkal a T-invariáns egy $T \rightarrow N$ leképezés, ahol T a tranzíciók halmaza, N a természetes számok halmaza, azaz súlyokat rendelünk az egyes tranzíciókhoz. A súlyvektor elemei fogják meghatározni, hogy az egyes tranzíciókat hányszor kell végrehajtani, így juttatva vissza a rendszert a kezdeti állapotába.

A 4. ábrán látható Petri háló T-invariánsai az $\bar{v} = (t_1, t_2) = (n, n)$ alakban adhatók meg, ahol $n \geq 0$. Ha például vesszük a $(t_1, t_2) = (1, 1)$ T-invariánst, akkor az azt jelenti, hogy a t_1 és t_2 tranzíciót is egyszer kell eltüzelni és ennek hatására a rendszer visszajut az eredeti állapotába.



4. ábra: Petri háló T-invariánsainak szemléltetése

2.2.6.2 P-invariánsok

Jelen esetben azt követeljük meg, hogy a tokenek számának a helyek valamely részhalmaza felett egy súlyvektorral képzett súlyozott összege állandó maradjon, azaz a $\vec{\sigma}$ tüzelési

szekvenciára σ_P^TM = állandó. A $\vec{\sigma}$ tüzelési szekvencia a $M_0[\vec{\sigma} > M$ állapotátmenetet hajtja végre, így az állapotátmenet:

$$M = M_0 + W^T\sigma$$

A súlyozott tokenösszeget kiszámítva:

$$\sigma_P^T M = \sigma_P^T M_0 + \sigma_P^T W^T\sigma$$

Mivel $\sigma_P^T M = \sigma_P^T M_0$ = állandó adódik, hogy $\sigma_P^T W^T\sigma = 0$. Ez csak akkor teljesülhet minden σ tüzelési szekvenciára, ha $\sigma_P^T W^T = 0$, azaz

(1) $W\sigma_P = 0$.

Definíció: Az olyan σ_P súlyvektorokat, melyek az (1) egyenlet teljesülését garantálják hely- vagy röviden P-invariánsoknak nevezzük. Fontos megjegyezni, hogy P-invariánsok lineáris kombinációja is P-invariáns.

A 4. ábrán látható Petri háló P-invariánsai például a (p_1, p_2, p_3) = (1,1,0), (1,2,1) súlyvektorokkal jellemezhetőek. A vektor adott sorszámú elemével kell súlyozni az adott sorszámú helyen levő tokenek számát.

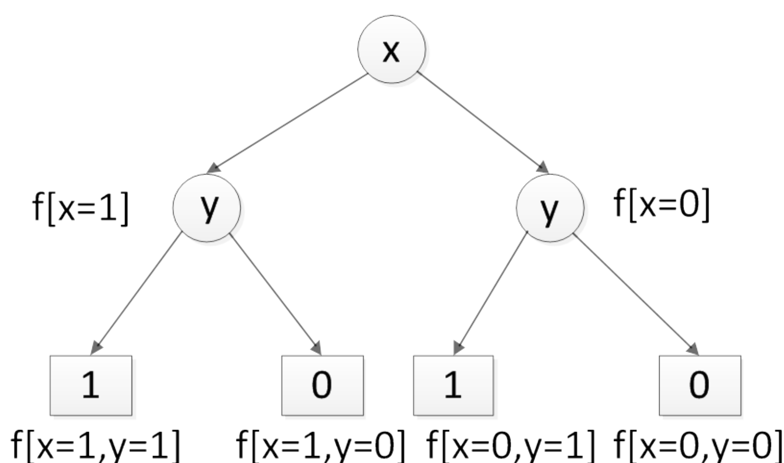
A kezdeti tokeneloszlás esetén a súlyozott összeg az (1,1,0) P-invariánssal számolva $2 \cdot 1 + 1 \cdot 1 = 3$. A t_1 tranzíció tüzelésével a tokeneloszlás az [1, 2, 2] vektorral írható le, a súlyozott összeg $1 \cdot 2 + 1 \cdot 1 = 3$, tehát azonos az előbbivel.

2.3 Döntési diagramok és alkalmazásaik a modellellenőrzésben

A nagy méretű állapothalmazok tárolási és kezelési problémájának megoldására nyújtanak alternatívát a szimbolikus technikák [3][4][5]. Itt az állapothalmazok explicit tárolása helyett azok karakterisztikus függvényét (bináris függvény) tároljuk [18][19] és a halmazműveleteket is a karakterisztikus műveletek segítségével értelmezzük. Egy állapothalmaz karakterisztikus függvénye az az állapotváltozókon értelmezett logikai függvény, mely akkor és csakis akkor igaz, ha az állapot az adott halmazba tartozik. Az állapotváltozók számát ezek alapján a kódolandó állapotok száma határozza meg. A karakterisztikus függvények használatának matematikai alapját a Stone-tétel (Boole-algebrák reprezentációs tétele) adja: minden véges Boole-algebra izomorf egy véges *S* halmaz részhalmazainak algebrájával.

2.3.1 Bináris döntési diagramok

Szimbolikus modellellenőrzés során a karakterisztikus függvénnyel kódolt állapottér tárolására a redukált sorrendezett bináris döntési diagramok (reduced ordered binary decision diagram, ROBDD) hatékonyan alkalmazhatók [5][6]. A döntési diagramok bevezetéséhez meg kell ismerkedni az *if-then-else* operátorral; $x \to x_1, x_0 = (x \wedge x_1) \vee (\neg x \wedge x_0)$. A kifejezés az x_1 értékét veszi fel, ha x igaz, x_0 értékét veszi fel, ha x hamis. Amikor a karakterisztikus függvényt ki akarjuk értékelni az előbbi operátor segítségével akkor az iteratív alkalmazás során, a jobb oldalon egyre kevesebb változó marad, mindezt úgy képzelhetjük el, hogy egy döntési fa szintjein megyünk lefelé. A folyamatot szemlélteti az 5. ábra: a példa az $f = (x \wedge y) \vee (\neg x \wedge y)$ karakterisztikus függvényhez tartozó döntési fát mutatja. Először az *x* változó értékét kell behelyettesíteni, ezzel a fában egy szinttel lejjebb jutunk, ahol az *y* változó értékét kell behelyettesíteni. A levél csomópontokban a 0 vagy 1 logikai értékek szerepelnek, látható, hogy igaz értéket akkor kapunk, ha $x = 1$ és $y = 1$ vagy $x = 0$ és $y = 1$.



5. ábra: Az $f = (x \wedge y) \vee (\neg x \wedge y)$ karakterisztikus függvényhez tartozó döntési fa

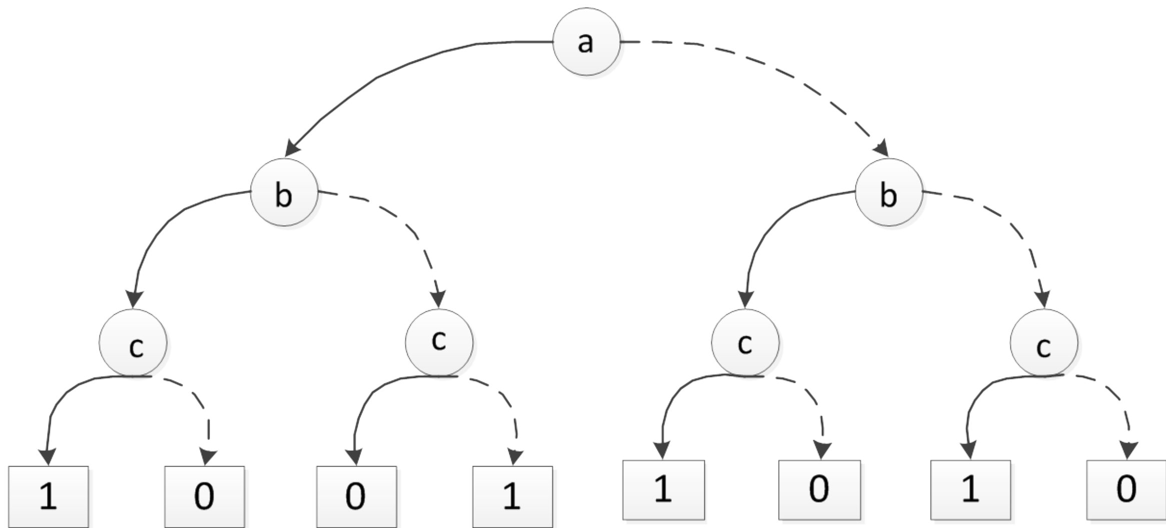
A fa levél csomópontjaiban már az igaz vagy hamis logikai értékek maradnak csak. A feladatunk tehát a döntési fán a gyökértől kiindulva a karakterisztikus függvénynek megfelelő változó értékét behelyettesíteni így egy szinttel lejjebb jutunk. A bináris döntési fát többféleképpen egyszerűsíthetjük:

- Bináris döntési diagramot (binary decision diagram, BDD) kapunk, ha az azonos csomópontokat illetve részfákat összevonjuk.
- Rendezett bináris döntési diagramot (ordered binary decision diagram, OBDD) kapunk, ha a felbontás során minden ágon azonos sorrendben vesszük fel a változókat.
- Redukált rendezett bináris döntési diagramot (reduced ordered binary decision diagram, ROBDD) kapunk, ha a döntési fában a szükségtelen csomópontokat (amelyekből a felbontás során azonos csomóponthoz vezet mindkét ág) redukáljuk: a csomópontot töröljük és a bemenő élet a kimenő élek által elért csomóponthoz irányítjuk.

Az ROBDD az alábbi tulajdonságokkal rendelkezik:

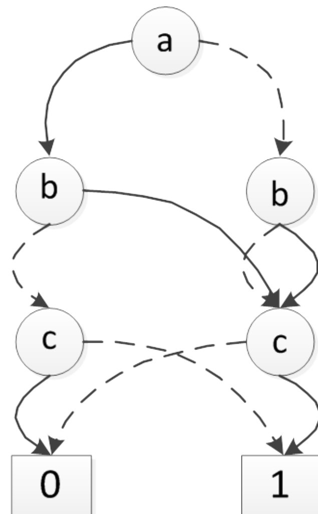
- Irányított, aciklikus gráf egy gyökérrel és két levéllel. A levelekben tehát csak az igaz és hamis logikai konstansok vannak, a csomópontokat egy teszt változóval címkézzük.
- Minden csomópontból két él indul ki, a teszt változóba való 0 és 1 logikai értékek behelyettesítését reprezentálják.
- Az izomorf részfák egyetlen részfává vannak összevonva.
- Azok a csomópontok, ahonnan kimenő él ugyanahhoz a csomóponthoz vezet, redukálva vannak.
- A változók azonos sorrendben fordulnak elő minden útvonal mentén.

Az alábbi ábrák szemléltetik a döntési fa típusait, az ábrázolt karakterisztikus függvény az $f = (b \wedge c) \vee (a \wedge \neg b \wedge \neg c) \vee (\neg a \wedge c)$. A 6. ábrán a döntési fa alapú reprezentáció látható.



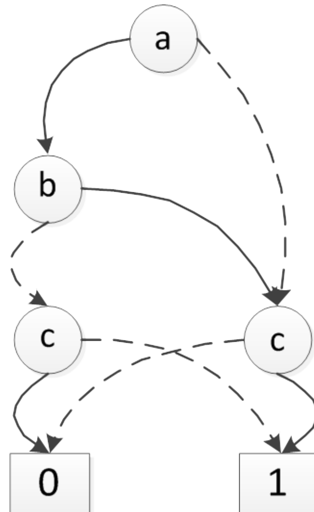
6. ábra: karakterisztikus függvény döntési fa alapú reprezentációja

A 7. ábra a 6. ábrán látható döntési fához tartozó BDD reprezentáció. Itt az azonos részfák és csomópontok össze vannak vonva.



7. ábra: karakterisztikus függvény BDD alapú reprezentációja

A 8. ábrán pedig az ROBDD látható, ahol már a fölösleges csomópontok törölve is vannak és az élek is át vannak irányítva a megfelelő csomópontokba.



8. ábra: karakterisztikus függvény ROBDD alapú reprezentációja

Fontos megjegyezni: a döntési diagram mérete szempontjából lényeges, hogy mi a teszt változók sorrendje. Az optimális sorrendezés NP teljes probléma, de sok esetben heurisztikus módszerekkel jó eredményt lehet találni [20]. A *PetriDotNet* modellellenőrző program a döntési fa teszt változóinak és ezzel együtt a szintek kialakításához többféle módszert biztosít. Lehetőség van

- a Petri háló P-invariánsai
- manuális szintezés
- összetett, a P- és T-invariánsokat is figyelembe vevő módszer [2]

alapján kialakítani a szintezést.

2.3.2 Többértékű döntési diagramok

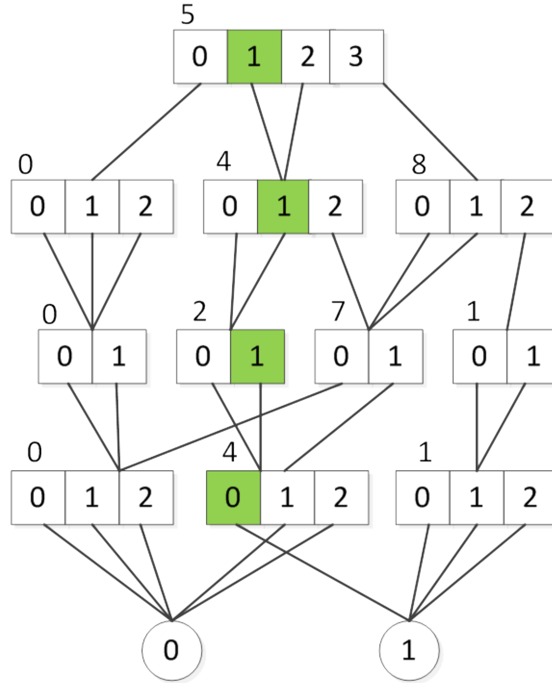
A bináris döntési diagramokkal ellentétben a *többértékű döntési diagram* [21][22] nem bináris típusú változókat tartalmaz, egy változó egy véges halmaz elemei közül tetszőleges értéket felvehet. A szaturációs algoritmus az úgynevezett *kvázi-redukált többértékű döntési diagramokat* [3] (quasi-reduced multi-way decision diagram, quasi-reduced MDD) használja. A kvázi-redukált jelző arra utal, hogy a duplikált csomópontok itt sem megengedettek a döntési diagramban, de az olyan csomópontok igen, melyek minden éle azonos csomópontba mutat, ugyanis az algoritmus során ez is lényegi információt fog hordozni, így ezeket nem redukáljuk.

A kvázi-redukált MDD főbb tulajdonságai:

- A fa csomópontjait $\langle k|p\rangle$ alakkal jelöljük, ahol k a csomópont szintje, p pedig valamilyen elsődleges index a csomópont azonosítására.
- A fa K+1 szintből áll, a nulladik szinten található a két terminális levél csomópont ($\langle 0|0\rangle$ *és* $\langle 0|1\rangle$). A gyökér csomópont a K-ik szinten található, jelölése $\langle K|r\rangle$.
- Irányított, aciklikus gráf.
- Egy nem terminális $\langle k|p\rangle$ csomópontnak az élei a k-1 szinten található csomópontokba mutatnak. Ha a $\langle k|p\rangle$ csomópont i-edik éle a $\langle k-1|q\rangle$-ba mutat, akkor ezt a továbbiakban a $\langle k|p\rangle[i] = q$ formában jelöljük.

A 9. ábra mutat példát a kvázi-redukált MDD-re. Az itt látható MDD 4+1 szintből áll. A korábbi jelöléseket alkalmazva írhatjuk például, hogy $\langle 4|5\rangle\,[0] = \langle 3|0\rangle$. Az egyes

csomópontokon belül a lokális állapotok találhatók a négyzetekben. A rendszer elérhető állapotai a gyökér csomópontból kiindulva az 1-es terminális csomópontig vezető utakon olvashatók le a lokális állapotok mentén kódolva, így például előfordulhat az 1-1-1-0 állapot a rendszer működése során (zöld színnel).



9. ábra: Kvázi-redukált MDD

Ezen jelölések felhasználásával útvonalakat is le tudunk írni. Ha egy $\langle k|p\rangle$ csomópontból kiindulva el lehet érni egy m-ik szinten levő csomópontot egy $(i_k, \ldots, i_m) \in \mathcal{S}_k \times \ldots \times \mathcal{S}_m$ m-esen keresztül, ahol $K \geq k > m \geq 1$, akkor azt rekurzívan az $\langle k|p\rangle[\mathrm{i_k}, \mathrm{i_{k-1}}, \ldots, \mathrm{i_m}] = \langle \mathrm{k}-1|\langle k|p\rangle[\mathrm{i_k}]\rangle[\mathrm{i_{k-1}}, \ldots, \mathrm{i_m}]$ formában írjuk le.

A $\langle k|p\rangle$ által vagy az alatt kódolt állapotokat a $\mathcal{B}(\langle k|p\rangle) = \{\beta \in \mathcal{S}_k \times \ldots \times \mathcal{S}_1 : \langle k|p\rangle[\beta] = 1\}$ formában jelöljük.

Definíció: Tekintsük a $\langle k|p\rangle$ illetve $\langle k|q\rangle$ gyökér csomóponttal rendelkező MDD-ket. A két MDD uniója egy olyan $\langle k|u\rangle$ gyökerű MDD lesz, melyre teljesül, hogy $\mathcal{B}(\langle k|u\rangle) = \mathcal{B}(\langle k|p\rangle) \cup \mathcal{B}(\langle k|q\rangle)$.

2.4 Állapottér-bejárás

Definíció: Egy rendszer diszkrét állapotainak modellje a $(\hat{S}, \mathrm{s}, \mathcal{N})$ hármassal írható le, ahol:

- $\hat{S}$ a rendszer lehetséges állapotainak véges halmaza, az egyes állapotokat vastagított kis betűvel írjuk
- $\mathrm{s} \in \hat{S}$ a rendszer kezdeti állapotát jelenti
- $\mathcal{N}: \hat{S} \rightarrow 2^{\hat{S}}$ a következő állapot (next state) függvény, mely azt hivatott megmutatni, hogy egy adott állapotból melyik állapot(ok) érhetők el egy lépésben.

Definíció: A rendszer elérhető, véges állapotainak halmaza az az $S \subseteq \hat{S}$ legszűkebb halmaz, amely tartalmazza a kiindulási s állapotot és mindazokat, amelyek ebből kiindulva az $\mathcal{N}$ iteratív alkalmazásával elérhetőek.

Formálisan ez az $S = \{s\} \cup \mathcal{N}(s) \cup \mathcal{N}(\mathcal{N}(s)) \cup \ldots = \mathcal{N}^*(s)$ felel meg, ahol a „*” szimbólum a tranzitív és reflektív lezártat jelöli.

Fontos tehát megjegyezni, hogy S az elérhető, míg $\hat{S}$ a lehetséges állapotok halmazát. A kettő nem feltétlenül azonos halmaz, mert előfordulhat az, hogy például Petri hálóknál olyan a kezdeti token-eloszlás, hogy az kizárja egyes állapotok elérhetőségét. Ilyenkor S valódi részhalmaza $\hat{S}$-nak.

2.4.1 Explicit módszerek az állapottér felderítésre

A hagyományos explicit technikák a rendszer elérhető állapotait lépésenként, egyenként derítik fel és tárolják el. Általában az algoritmus magját a szélességi vagy mélységi bejárás adja. Egy ilyen algoritmusra mutat példát az alábbi pszeudokód:

```
ExplicitStateSpaceGeneration(s: állapot, 𝒩 : következő állapot függvény):
állapotok halmaza

1 𝒮,𝒰: állapotok halmaza, ψ: állapotok halmaza → 𝒩 ∪{null} függvény, i,j:
állapot
2 𝒮 = {s} //elért állapotok
3 𝒰= {s} //felderítésre váró állapotok
4 ψ(s) = 0 //a hash-tábla kulcsául szolgáló értékeket megadó függvény
5 while 𝒰 ≠ ∅ do
6    válasszunk ki egy i állapotot 𝒰-ból és azt távolítsuk is el
7    foreach j∈ 𝒩(i)
8          if j ∉𝒮 then
9                ψ(j) = |𝒮| //a következő kiosztott érték a felderített
                            állapot sorszáma
10               𝒮 = 𝒮 ∪{j}
11               𝒰 = 𝒰 ∪{j}
12 return 𝒮
```

Jól látható, hogy az algoritmus tár- és futási idő igénye a felderített állapotok számával arányos, tehát $\mathcal{S}$ mérete gátat szab az alkalmazhatóságnak a jelenlegi hardver erőforrásokat is figyelembe véve.

2.4.2 Szimbolikus állapottér-felderítés

A következő rész néhány, az állapottér felderítésére szolgáló szimbolikus technikát vesz szemügyre, melyekben közös, hogy a szélességi bejáráson alapulnak. Ezen algoritmusok a rendszer elérhető állapotainak halmazát a következő állapot (*Next State,* $\mathcal{N}(\mathcal{S})$) függvény iteratív alkalmazásával állítják elő.

```
BreadthFirstStateSpaceGeneration(s: állapot, 𝒩 : következő állapot
függvény): állapotok halmaza
1 𝒮,𝒰,𝒳: állapotok halmaza
2 𝒮 = {s} //elért állapotok
3 𝒰= {s} //felderítésre váró állapotok
4 while 𝒰 ≠ ∅ do
5    𝒳 = 𝒩(𝒰) //lehetséges új állapotok
6    𝒰 = 𝒳 \ 𝒮 //ténylegesen új állapotok
7    𝒮 = 𝒮 ∪ 𝒰
8 return 𝒮
```

```
AllBreadthFirstStateSpaceGeneration(s: állapot, 𝒩 : következő állapot
függvény): állapotok halmaza
1 𝒮,𝒰,𝒳: állapotok halmaza
2 𝒮 = {s} //elért állapotok
3 𝒰 = {} //régi állapotok tárolására szolgál
4 while 𝒰 ≠ 𝒮 do
5    𝒰 = 𝒮
6    𝒮 = 𝒮 ∪ 𝒩(𝒮) //az egész eddig felderített állapottérre alkalmazzuk a
függvényt
7 return 𝒮
```

A két algoritmus abban különbözik egymástól, hogy míg az elsőben a következő-állapot függvényt csak a ténylegesen új állapotokra alkalmazzuk, addig a másodikban az egész addig a lépésig felderített állapottérre. Vegyük észre, hogy a *k*-ik lépésben az első algoritmus azokra az állapotokra fogja meghívni a következő állapot függvényt melyek pontosan *k* lépésben érhetők el a kiindulási állapotból. Ezzel szemben a második algoritmus a legfeljebb *k* távolságra levő állapotok egész halmazára meg fogja hívni a függvényt.

Az algoritmus egy továbbfejlesztett változata a láncolásos (chaining) algoritmus [3], melyben a következő állapot függvényt külön specifikusan egy eseményhez kapcsolódóan hajtjuk végre és a felderített állapotokat még a ciklusmagon belül hozzáadjuk a korábban felderítettekhez. Ez után folytatjuk más esemény eltüzelését, majd ha mind el lett tüzelve, akkor fog megtörténni a ciklusmag újbóli meghívása. Ennek előnye, hogy ha az események tüzelésének sorrendje olyan, hogy az egy egész útvonalat határoz meg az állapotok között (tegyük fel hogy *k* hosszúságú), akkor az algoritmus a ciklusmag egyszeri lefuttatásával megtalálja ezt, szemben a korábban közölt két algoritmussal, melyben *k* iterációra lenne ehhez szükség. Ez az algoritmus tehát úgy képzelhető el, hogy a szélességi bejárás közben ezeket az útvonalakat a mélységi bejáráshoz hasonlóan végig tudja követni.

2.5 A szaturációs algoritmus

A szaturációs algoritmus [3][15][16] egy szimbolikus technika a rendszer elérhető állapotainak bejárásához, mely az állapotok tárolásához kvázi-redukált döntési diagramokat használ.

Az algoritmus futása a következő lépésekből áll:

1. A modell dekompozíciója
2. Szaturáció, amely a következő fontos algoritmusokat tartalmazza:
    a. in-place update, azaz helyben frissítés
    b. az állapotátmeneti függvény *Kronecker mátrix* alapú tárolása
    c. rekurzív mélységi állapottér-bejárás és döntési diagram építés

A modell dekompozíciója során a bemenetet a 2.5.1 fejezetben leírtaknak megfelelően részekre bontjuk, amelyek alapján majd a szimbolikus kódolást menet közben elvégezzük. A dekompozíciót követően, a szaturáció során felhasznált helyben frissítés előnye, hogy jelentősen csökkenti az algoritmus futása során létrehozott csomópontok számát, ezért lehet hatékony a memória felhasználás szempontjából. A helyben frissítés módszere a 2.5.3 fejezetben kerül bemutatásra.

A Kronecker mátrixos állapotátmenet-tárolás segítségével elkerüljük a következő állapot függvény explicit, döntési diagramokkal történő tárolását (2.5.2 fejezet).

A 2.5.4 fejezetben megismerhető a szaturáció alapját képező rekurzív mélységi bejárás algoritmusa. Ennek jelentős szerepe van mind az állapottér-felderítése, mind a döntési diagram építése során. Alkalmazásával jelentős sebességnövekedés és tárhatékonyság érhető el a korábban említett explicit és szimbolikus technikákhoz képest. A szaturációs algoritmus bemutatása a 2.5.5 fejezetben található, mely formailag az alábbiak szerint jellemezhető:

- bemenete a rendszer Petri háló alapú modellje
- kiemente egy MDD
- eredményeként kapott MDD-ben a gyökér csomópontból kiinduló és terminális 1-es csomópontba vezető élsorozatok reprezentálják a rendszer elérhető állapotait.

2.5.1 Az állapottér dekompozíciója

Aszinkron rendszerek esetén lehetőség nyílik arra, hogy $\mathcal{N}$ -et felbontsuk diszjunkt következő állapot függvények uniójára, ezen elgondoláson alapulva $\mathcal{N}(i) = \bigcup_{s \in \varepsilon} \mathcal{N}_\alpha(i)$. Itt $\mathcal{N}_\alpha$ a következő állapot függvény, mely az α eseményen alapul, ε az események véges számú halmaza. $\mathcal{N}_\alpha(i)$ azon állapotok véges halmazát jelöli, melyen az i állapotból kiindulva, α esemény tüzelésének hatására elérhetőek.

Sok magas szintű modellnél megjelenő tulajdonság az, hogy a modellt kisebb, egymással kölcsönhatásban levő részmodellekre lehet partícionálni. Ennek következménye, hogy egy globális i állapotot ezután egy M-essel tudunk leírni az (i_1, i_2, …, i_M) formában. Ezzel a jelöléssel i_K egy lokális állapot a K-ik részmodellben. Ezek alapján a rendszer lehetséges állapotainak halmazát $(\hat{S})$ az M darab részmodell lehetséges állapotai halmazának Descartes szorzata adja meg.

Petri hálóknál lehetőség van a modell dekompozíciójára oly módon, hogy azt részhálókra bontjuk és így a token-eloszlás az N darab részeloszlásból képzett vektorként áll elő.

Az események lokalitása

Fontos tulajdonsága a Petri hálóknak az, hogy nem minden tranzíció befolyásol minden szintet, azaz a tüzelésével nem változik meg az adott szinten az elérhető állapotok halmaza. Ennek tudatában az állapottér felderítése során felesleges minden szinten minden eseményt eltüzelni, ilyen módon jelentősen növelhető az algoritmus teljesítménye. Jelölje Top(α) azt a legmagasabb szintet, Bot(α) pedig azt a legalacsonyabbat, amelyet az α még befolyásol. Formálisan ez a $Top(\alpha) = max\{k : \mathcal{N}_{k,\alpha} \neq \boldsymbol{I}\}$ illetve $Bot(\alpha) = min\{k : \mathcal{N}_{k,\alpha} \neq \boldsymbol{I}\}$ formában adható meg. Az események ε halmazát felbonthatjuk K osztályra, $\varepsilon_k = \{\alpha \in \varepsilon, Top(\alpha) = k\}$, minden $K \geq k \geq 1$ -re. Lényeges tehát észrevenni, hogy egy esemény tüzelésekor nem szükséges a gyökér csomóponttól elindulni és onnan az alsóbb szintek felé végig eltüzelni azt. Egy α esemény, melyre $Top(\alpha) =$ k, független a K és k+1 közötti szintektől, így ilyenkor a tüzelést csak a k-ik és az alatti szintekre kell végrehajtani. Másik fontos észrevétel, hogy az előbb leírt jelenséget ki nem használó algoritmusok egy α esemény tüzelését a gyökér csomópontnál elkezdik és egy $\langle k|p \rangle$ csomópontra annyiszor fogják végrehajtani a tüzelést, ahány él vezet ebbe a csomópontba. Ezzel szemben, ha kihasználjuk azt, hogy egyből a k-ik szintre ugorhatunk a tüzelés végrehajtásával, akkor a $\langle k|p \rangle$ csomópontra csak egyszer fog ez megtörténni. Annyiban jelent ez nehézséget, hogy ha egy $\langle k|p \rangle$ csomópontot helyettesíteni akarunk egy $\langle k|q \rangle$ csomóponttal, akkor ahhoz az kell, hogy az összes $\langle k|p \rangle$-be menő élet át kell irányítani a $\langle k|q \rangle$ csomópontba. Ezen problémára ad majd megoldást a helyben frissítés módszere.

2.5.2 A következő állapot függvény

A következő-állapot függvény hatékonyan és szemléletesen kódolható a Kronecker mátrixok [33] alkalmazásával, így a következő állapot függvény egy kétdimenziós mátrixként való reprezentációját kapjuk. A mátrixos forma leggyakoribb felírásában a sorok a részmodelleknek felelnek meg, az oszlopok pedig az eseményeknek. Erre akkor van lehetőség, ha a modell dekompozíciója *Kronecker konzisztens* [15], ez alatt azt értjük, hogy létezik $K \cdot |\varepsilon|$ darab függvény – ahol az egyes függvények $\mathcal{N}_{k,\alpha} : \mathcal{S}_k \to 2^{\mathcal{S}_k}$ alakúak – melyek egy adott esemény hatását írják le egy adott részmodellen (itt K az részmodellek száma, $|\varepsilon|$ pedig az események halmazának számossága). Más szavakkal annak kell teljesülnie, hogy egy globális állapoton való esemény tüzelés hatása meg kell, hogy egyezzen az egyes részmodelleken való tüzelések eredményének Descartes szorzatával. A lokális következő állapot függvény egy 0 és 1 elemeket tartalmazó mátrixként írható fel, $\mathcal{N}_{k,\alpha} \in \{0,1\}^{n_k \times n_k}$ (itt n_k a k-ik részmodell állapotainak száma). $\mathcal{N}_{k,\alpha}[i_k, j_k] = 1 \Leftrightarrow j_k \in \mathcal{N}_{k,\alpha}(i_k)$, azaz a mátrixelem csak akkor nem nulla, ha az i_k állapotból a j_k állapot elérhető valamilyen tüzelés végrehajtásával.

Ezek alapján a teljes következő állapot függvény az alábbi képlet szerint áll elő:

$$\mathcal{N} = \sum_{\alpha \in \varepsilon} \bigotimes_{K \geq k \geq 1} \mathcal{N}_{k,\alpha}$$

Ebben a képletben a $\otimes$ szimbólum a mátrixok Kronecker szorzatának felel meg. Ha A egy $m \times n$-es mátrix, B pedig egy $p \times q$ méretű akkor a $C = A \otimes B$ mátrix egy $mp \times nq$ méretű mátrix lesz, ahol:

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}$$

Az előálló mátrix rendkívül ritka, Petri hálók esetén egy sor legfeljebb egy nemnulla elemet tartalmaz. Színezetlen Petri hálókra a Kronecker konzisztencia feltétel mindig teljesül.

Előfordulhat az, hogy ha egy részmodell bármely lokális állapotában az α eseményt eltüzelve az nem változik meg. Tegyük fel, hogy a részmodell a k-ik a partícionálás szerint. Ekkor azt mondhatjuk, hogy az α esemény független a döntési diagram k-ik szintjétől. A tulajdonság szemléletes jelentése az is, hogy ilyenkor $\mathcal{N}_{k,\alpha} = \boldsymbol{I}$, ahol $\boldsymbol{I}$ jelöli az identitásmátrixot (egységmátrix), tehát ilyenkor bármely tranzíció tüzelésének hatására az adott állapot nem változik meg.

2.5.3 A helyben frissítés módszere (in-place update)

A helyben frissítés alapgondolata az, hogy egy $\langle k|p \rangle$ csomóponton a lehetséges eseményeket kimerítően eltüzeljük és a tüzelések eredményének megfelelően az elérhető állapotok halmazát inkrementálisan bővítjük. A módszer lényege az, hogy amikor egy olyan α eseményt tüzelünk el a $\langle k|p \rangle$ csomóponton, amire $Top(\alpha) \leq k$, akkor az új, elérhető állapotokat a $\langle k|p \rangle$ éllistájának folyamatos bővítésével kódoljuk el ahelyett, hogy minden tüzelés eredményét egy új csomópontban tárolnánk. Azáltal, hogy helyben frissítjük az éllistát, sok költséges döntési diagram műveletet tudunk megspórolni, továbbá jelentősen csökken a memória felhasználás, különösen aszinkron modellek esetén.

2.5.4 Rekurzív mélységi állapottér-felderítés

Jellemzően a modellellenőrző algoritmusok szélességi, vagy kombinált szélességi-mélységi (pl. chaining [3]) állapottér bejárást alkalmaznak. Ilyenkor az állapottér egy részéből megvizsgálják az elérhető következő állapotokat, és azokkal bővítik az állapotok aktuális halmazát. A szaturáció újdonsága ezzel szemben abban rejlik, hogy a rekurzív bejárást nem egyenként az állapotokra, hanem csomópontokra alkalmazza, majd az esetleges változásokat rekurzívan lefelé érvényesítve számítja ki az elérhető állapothalmazt. Mindeközben a tüzelést mohó módon, kimerítően alkalmazza, amíg egy lokális fixpontot el nem érünk. Lokális fix pontról akkor beszélünk, ha az állapotok részhalmaza lokális tüzelésekkel (azaz olyan tüzelés, amely az adott állapothalmazban tüzelhető) nem bővíthető tovább.

A rekurzív mélységi állapottér-felderítést 2 függvény, a *SatFire* és a *SatRecFire* végzi. A *SatFire* egy $\langle k|p\rangle$ csomóponton tüzel el egy α eseményt, ahol k = Top(α). A *SatRecFire* függvény paraméterül kapott α esemény tüzelését végzi el mindazokon az $\langle l|q\rangle$ csomópontokon, amelyekre teljesül, hogy $Top(\alpha) \geq l \geq Bot(\alpha)$, de nem módosítja ezeket a csomópontokat. E helyett létrehoz egy új csomópontot és azon tüzeli el az eseményt, amely így a tüzelés hatását reprezentálja $\langle l|q\rangle$-n és gyerekein. A metódusban használt *Cached* és *PutInCache* metódusok teljesen hasonlóak az unió műveletben használtakhoz, azzal a különbséggel, hogy ezek a tüzelési gyorsítótáron (*Fire Cache*, továbbiakban FC) végeznek műveleteket, nem pedig az uniók eredményét eltároló gyorsítótáron (union cache). A hash tábla kulcsa ebben az esetben az unióban használt két csomópont helyett (melyeken az unió műveletet végeztük el) egy csomópont és egy esemény, az érték pedig a tüzelés eredményét reprezentáló csomópont.

A helyben frissítés előnye továbbá, hogy az algoritmus mindaddig tüzeli az α eseményt a csomóponton - mely így már állapothalmazt reprezentál - míg az új állapot felderítését eredményezi. Ennek következtében a *SatFire*(α, k, p) hívás a $\langle k|p\rangle$ csomópontot helyben frissíti úgy, hogy a végén az a $\mathcal{N}_\alpha^*(\mathcal{B}(\langle k|p\rangle))$ állapotok egész halmazát kódolja.

2.5.5 Szaturáció

A szaturáció definíciójának megadása előtt bevezetünk egy jelölést a következő állapot függvényre vonatkozóan, ahol azoknak az eseményeknek az unióját vesszük, melyekre a befolyásolt szint a *k*-ik vagy az alatti:

$$\mathcal{N}_{\leq k} = \bigcup_{1 \leq l \leq k} \mathcal{N}_{\varepsilon_l} = \bigcup_{\alpha : Top(\alpha) \leq k} \mathcal{N}_\alpha$$

Ezek alapján a *szaturáció definíciója*: azt mondjuk hogy az MDD egy $\langle k|p\rangle$ csomópontja a *k*-ik szinten szaturált, ha az mindazokat az állapotokat reprezentálja melyek minden olyan *e* esemény tüzelésének hatására elérhetőek, ahol *Top(e)* ≤ *e*, azaz $\mathcal{B}(\langle k|p\rangle) = \mathcal{N}_{\leq k}^*(\mathcal{B}(\langle k|p\rangle))$.

Fontos megjegyezni, hogy a szaturáció szükséges, de nem elégséges feltétel arra vonatkozóan, hogy az adott csomópont szerepelni fog a végleges MDD-ben. Mivel $\mathcal{N}_{\leq K} = \mathcal{N}$ (K a szintek számát jelöli) ezért igaz az alábbi állítás is: $\mathcal{B}(\langle K|\mathrm{r}\rangle) = \mathcal{N}^*(\mathcal{B}(\langle \mathrm{K}|\mathrm{r}\rangle))$, ha a gyökér csomópont is szaturált. Ha adott egy s ∈ $\mathcal{B}(\langle \mathrm{K}|\mathrm{r}\rangle)$ állapot, akkor teljesül $\mathcal{N}^*(s) \subseteq \mathcal{B}(\langle \mathrm{K}|\mathrm{r}\rangle)$. Ennek következménye, hogyha kezdetben $\mathcal{B}(\langle \mathrm{K}|\mathrm{r}\rangle) = \{\mathrm{s}\}$ és a szaturáció során csak elérhető állapotokat adunk hozzá a $\mathcal{B}(\langle \mathrm{K}|\mathrm{r}\rangle)$ állapothalmazhoz, akkor az algoritmus futása végén igaz lesz, hogy $\mathcal{B}(\langle \mathrm{K}|\mathrm{r}\rangle) = \mathrm{S}$. Tehát a szaturáció végén a $\mathcal{B}(\langle \mathrm{K}|\mathrm{r}\rangle)$ kódolja a rendszer elérhető állapotainak teljes halmazát.

A szaturáció tehát egy rekurzív algoritmus, amely az MDD gyökér csomópontjából kiindulva építi fel a döntési diagramot. Az események tüzelését rekurzívan hajtja végre a szinteken lefelé haladva. Ha menet közben a *SatFire* által meghívott *SatRecFire* metódus létrehoz egy új csomópontot, akkor az azonnal szaturálva lesz. Ennek következménye, hogy amikor egy felsőbb szinten levő csomóponton hajtjuk végre a szaturációt, akkor biztos az, hogy az alsóbb szinten levő gyerek-csomópontok - melyek elérhetőek a csomópontból - már szaturáltak. A végső MDD-be csak szaturált csomópontok fognak bekerülni. A szaturációs algoritmus erőssége pont abban rejlik, hogy a többi szimbolikus technikához képest a futás közben létrehozott csomópontok száma nagyságrendekkel kevesebb. Fontos megjegyezni, hogy a létrejövő csomópontok száma nagymértékben függ az alkalmazott szintezéstől.

3 Párhuzamos modellellenőrzés

A hagyományos szimbolikus modellellenőrző algoritmusok hatékonyan működnek a gyakorlatban [3][15][16]. Felmerül a kérdés, hogy a napjaink számítógépei által kínált többszálú feladatvégzést fel tudjuk-e használni arra, hogy még kedvezőbb futási időeredményeket érjünk el a modellellenőrzés során akár a 2.5 fejezetben ismertetett szaturációt, akár más szimbolikus technikát felhasználva.

E fejezet célja a párhuzamos modellellenőrző algoritmusok bemutatása. A 3.1 részben olyan párhuzamos szimbolikus technikák kerülnek bemutatásra, amelyek alapját nem a szaturáció képzi. A 3.2 rész a kutatásunk alapjául szolgáló szaturációs algoritmus párhuzamos megvalósításával [4] foglalkozik.

3.1 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) [11] nevű eszköz egy Erlang és C++ nyelveken implementált párhuzamos modellellenőrző, mely a Stern és Dill által közölt DEMC (Distributed Explicit-state Model Checking) [12] algoritmuson alapul. Itt az állapotok egy elosztott mélységi bejárásáról van szó. A program egy megfelelő hash függvénnyel az egyes állapotokat csomópontokhoz rendeli hozzá, melyek 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, említést tesznek 30 milliárd állapotot tartalmazó állapottér ellenőrzéséről is.

A DIVINE [13] egy párhuzamos LTL modellellenőrző keretrendszer, mely hálózatban levő 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 [14] használják fel a kommunikációhoz. Az eszköz LTL formula vagy Büchi automata [5] formájában

megfogalmazott követelmények teljesülését tudja ellenőrizni. Az implementáció alapjául a [ ]-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 fejt ki, ehhez saját hash táblában tárolja el az elért állapotokat. A szálak közötti kommunikáció itt is üzenetváltáson alapul, ezzel csökkentve a kommunikáció többletköltségét.

A [10]-ben közölt módszer megkülönböztet dolgozó és koordinátor processzorokat, koordinátorból egy lehet a futás során. A csomópontok itt is hozzá vannak rendelve az egyes processzorokhoz, de ezen felül hatékony terheléskiegyenlítés is része a megoldásnak, melyet munkaállomások csoportján történő végrehajtásra optimalizáltak. Ha egy munkaállomás az állapottér bejárása során a felhasználható memória szűkösségét érzékeli, akkor a BDD-t szétvágja k részre. Egy részt megtart magának, a többi k-1 részt 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.

3.2 Párhuzamos szaturációs algoritmus

A 2.5 fejezetben egy szimbolikus modellellenőrzési technika került bemutatásra, az úgynevezett szaturáció. Jelen fejezetben ezen algoritmus párhuzamos változata kerül bemutatásra, amelynek alapjául a [4]-ben közölt publikáció szolgált. Mi ezt az algoritmust implementáltuk, és több iterációs lépésen keresztül fejlesztettük tovább.

3.2.1 Az algoritmus áttekintése

A szaturációs algoritmus [4]-ben közölt párhuzamos megvalósítása lehetővé teszi, hogy a 2.5 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.

A [4]-ben 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 [4]-ben bevezették az úgynevezett felfelé mutató éleket 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.

3.2.2 Az algoritmus bemutatása, megvalósítása

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

A szekvenciális szaturációs algoritmus párhuzamossá alakításához szükséges változtatások:

- a work pool [25] tervezési minta bevezetése
- 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 való többszörös hozzáférés biztosítása.

3.2.2.1 Work pool tervezési minta

A 2.5.1 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 [25] 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.

3.2.2.2 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 3.2.1 fejezetben bemutatott felfelé mutató élek nyilvántartására szolgál az élek kiinduló pontjában.
- `Ops`: Szinkronizációs okokból fontos, hogy a szaturáció során végig nyilvántartsuk, 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.

3.2.2.3 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 párost 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.

3.2.2.4 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 készítettük azt el. Az MDDNode osztályt kiegészítettük egy locked attribútummal, mely 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.

3.2.2.5 Az egyéb adatstruktúrák változásai

A szaturáció során több olyan adatot is szükséges tárolni, amelyek értéke a teljes állapottér felderítése 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 adatszerkezetin dolgoznak. Hogy ennek fényében az overhead-et tovább csökkentsük, a *read-write* zárakból szintenként hoztunk létre példányokat.

3.2.3 Az algoritmus főbb függvényei

Az alábbiakban a párhuzamos szaturációs algoritmus főbb függvényei kerülnek bemutatásra (a téglalapban az egyes függvények fejlécei találhatók), amelyhez még szükséges a *ZeroNode* fogalmának bevezetése.

Definíció: A *ZeroNode* szintenként azt a csomópontot jelenti, amelynek fölsőbb szintek esetén minden éle az alatta levő szint *ZeroNode*-jába mutat, míg a legalsó szinten minden éle a terminális 0 csomópontba mutat. A *ZeroNode* megfelel az üres állapothalmaznak. A *k*. szinten levő *ZeroNode*-t szokás $\langle k|0\rangle$-val is jelölni.

Saturate

```
Saturate(in: k: szint, p: index)
```

A függvény a $\langle k|p\rangle$ csomópont szaturációját végzi. Lépései:

- A $\langle k|p\rangle$ csomópont saturating értékét *true*-ra állítja, így jelzi a többi szál számára, hogy a csomópont szaturációja elkezdődött
- A $\langle k|p\rangle$ csomópont *ops* értékét megnöveli, hogy jelezze, jelenleg dolgozik a csomóponton

- Az összes elérhető lokális állapotra kimerítően eltüzelteti az eseményeket a *SatFire* függvény meghívásával
- A $\langle k|p\rangle$ csomópont *ops* értékét csökkenti, mivel a továbbiakban ez a szál már nem változtat a csomóponton
- Ha az *ops* 0-ra csökkent, akkor az azt jelenti, hogy más szál nem dolgozik a csomóponton, illetve felfelé élek sem mutatnak rá, amik késleltetnék a szaturáció befejezését. Ekkor a *NodeSaturated* függvényt kell meghívni a csomópontra.

SatFire

```
SatFire(in: k: szint, p: index, i:lokális állapot)
```

A függvény eltüzeli a $\langle k|p\rangle$ csomóponton az *i*. lokális állapotban engedélyezett eseményeket. Lépései:

- A $\langle k|p\rangle$ csomóponton engedélyezett *e* eseményekre végrehajtja az alábbi lépéseket:
- A $\langle k|p\rangle[i]$ csomóponton eltüzeli az *e* eseményt a *SatRecFire* meghívásával.
- Ha a *SatRecFire* visszatérése nem *ZeroNode*, akkor zároljuk a $\langle k|p\rangle$ csomópont alatti részgráfot. Majd a *SatRecFire*-től kapott csomópontot uniózza a $\langle k|p\rangle$ csomópont *j*. élének korábbi értékével, ahol *j* az a lokális állapot, amelyikbe az *e* esemény *i*. állapotban való eltüzelésének hatására kerül a rendszer.
- Ha az unió eredménye különbözik az él korábbi értékétől, akkor azt elmentjük, és feloldjuk az elhelyezett zárat.
- Ha történt változás, azaz új részét értük el az állapottérnek, akkor a *j*. állapotra meghívjuk a *Confirm* függvényt, majd a *j*. állapotban is eltüzeltetjük a lehetséges eseményeket a *SatFire* rekurzív meghívásával, különben pedig feloldjuk az elhelyezett zárat.

SatRecFire

```
SatRecFire(in: e: esemény, l: szint, q: index, p: index, i: lokális
állapot): index
```

Az $\langle l|q\rangle$ csomóponton eltüzeli az *e* eseményt. Lépései:

- Megvizsgálja, hogy az *e* esemény még hatással van-e az *l*. szinten. Ha nincs, akkor visszatér az $\langle l|q\rangle$ csomóponttal, mivel nem kellett rajta változtatást végezni.
- Zárolja a tüzelési gyorsítótárat.
- Megvizsgálja a gyorsítótárban, hogy az $\langle l|q\rangle$ csomóponton el lett-e már tüzelve az *e* esemény.
- Ha volt találat, de az még nem teljesen szaturált, akkor a talált csomópontból felfelé mutató éleket állít be az $\langle l+1|p\rangle$ csomópontra, ahonnan kezdeményezték az esemény tüzelését. Így az $\langle \mathrm{l}+1|\mathrm{p}\rangle$ csomópont szaturációja nem fejeződhet be, míg a gyorsítótárban talált csomópont nem lett szaturált. Ekkor a függvény *ZeroNode*-dal tér vissza, miután feloldotta a zárat a gyorsítótárról.
- Ha volt találat, és az már szaturált, akkor a függvény feloldja a gyorsítótáron elhelyezett zárat, és visszatér a megtalált csomóponttal.
- Ha nem volt találat, akkor létrehozza az $\langle l|s\rangle$ csomópontot a tüzelés eredményének tárolására, majd felfelé mutató éleket állít az $\langle l+1|p\rangle$ csomópontba.
- Megnöveli az $\langle l|s\rangle$ csomópont *ops* értékét, hogy jelezze a szál, hogy dolgozik a csomóponton.
- Elhelyezi a csomópontot a tüzelési gyorsítótárban, de jelzi, hogy az még nem szaturált. Az $\langle l|s\rangle$ csomópont key változóját *Key*(*l*+1, *p*)-re állítja.

- Feloldja a zárat a tüzelési gyorsítótárról
- Az L listába lekéri $\langle l|q\rangle$ azon állapotait, amelyekben az *e* esemény engedélyezve van.
- Amíg a listából tud kivenni egy lokális *g* állapotot, addig a következőket hajtja végre:
- A g lokális állapoton is eltüzeli az e eseményt önmaga rekurzív meghívásával. A rekurzió eredményét a *SatFire*-höz hasonlóan feldolgozza.
- Ha az *ops* érték csökkentése után az 0-ra csökkent, akkor megvizsgálja, hogy vannak-e a csomópontnak lefelé mutató élei.
- Ha nincsenek, azaz egyetlen állapotában sem volt eltüzelhető az esemény az alsóbb szintek miatt, akkor helyettesíthető *ZeroNode*-dal. Ekkor töröljük a csomópontot.
- Ha van lefelé mutató él, akkor kezdeményezi a csomópont szaturációját a *QSaturate* meghívásával.
- A függvény visszatér *ZeroNode*-dal.

NodeSaturated

```
NodeSaturated(i:n k: szint, p: index)
```

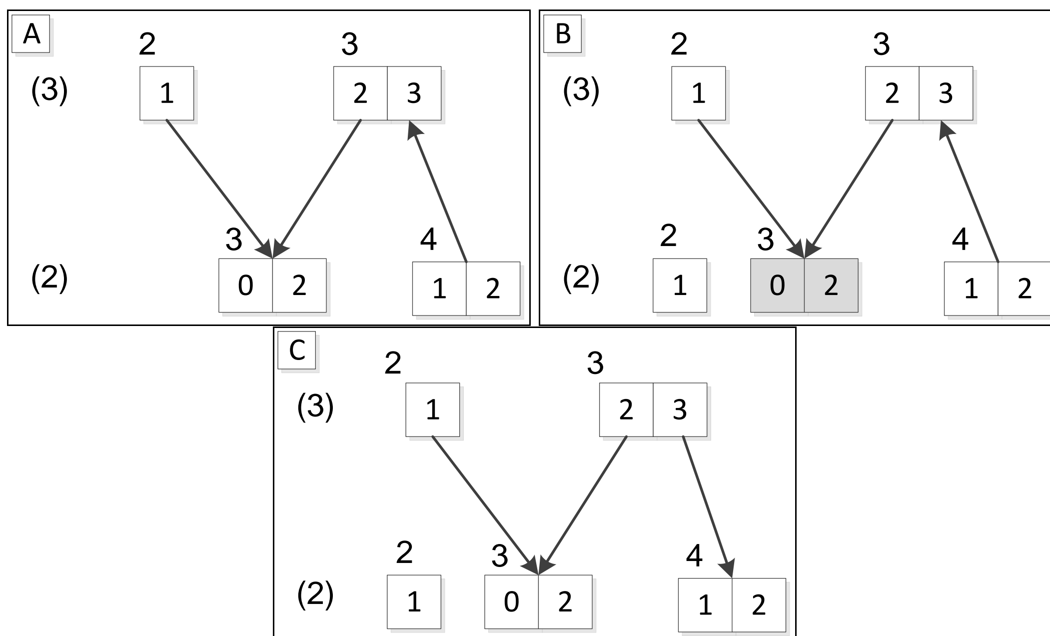
A $\langle k|p\rangle$ csomópont szaturációjának befejezése. Lépései:

- A $\langle k|p\rangle$ csomópontot elhelyezzük az MDD-tárolóban.
- Ha k megfelel a legfölsőbb szintnek, akkor elkészült a modell teljes szatuárciója. Ekkor meghívja a *Terminate* függvényt, és véget ér a szaturáció.
- Zárat helyez el a tüzelési gyorsítótáron. Frissíti a gyorsítótárat arra a csomópontra, amelyiket a *CheckIn* függvény visszaadott, illetve igazra állítja, hogy a csomópont már szaturált.
- Feloldja az elhelyezett zárat.
- Amíg van a csomópontból felfelé mutató él, amelynek végpontja $\langle k+1|r\rangle[i]$, addig
- Zárolja a $\langle k+1|r\rangle$ alatti részgráfot
- Uniózza $\langle k+1|r\rangle[i]$-t $\langle k|p\rangle$-vel.
- Ha az unió eredménye különbözik az él korábbi értékétől, akkor azt elmenti, és a feloldja az elhelyezett zárat.
- Ha történt élállítás és $\langle k+1|r\rangle$ szaturációja már korábban elkezdődött, akkor az *i* állapotot is bejárjuk a *SatFire* meghívásával.
- Csökkentjük $\langle k+1|r\rangle$ *ops* értékét, mivel feldolgozta a csomópontba mutató felfelé élet.
- Ha az *ops* 0-ra csökkent, akkor megvizsgálja, hogy korábban elkezdték-e már szaturálni a csomópontot. Ha igen, akkor ezzel befejeződött a csomópont szaturációja, és meghívja rá a *NodeSaturated* függvényt. Ha még nem szaturálták a csomópontot, akkor pedig kezdeményezi azt a *QSaturate* meghívásával.

A vizsgált Petri háló szaturációját az *Initialize* [A8] függvény meghívásával lehet elindítani, míg a szaturáció végét a *Terminate* függvény lefutása jelzi.

3.2.4 Példa

A 3.2.1 fejezetben leírtak könnyebb megértését segítve az alábbiakban egy példán keresztül bemutatásra kerül a szaturációs algoritmus párhuzamos futása.



10. ábra: példa MDD részlete

Az 10. ábra A része egy MDD részletet tartalmaz. Az ábrán az MDD-nek csak a 2. és 3. szintjei láthatók, a többi szint nincs megjelenítve. A szaturáció jelenlegi állapota szerint 3 él van behúzva a megjelenített csomópontok között. A példa kontextusa legyen a következő:

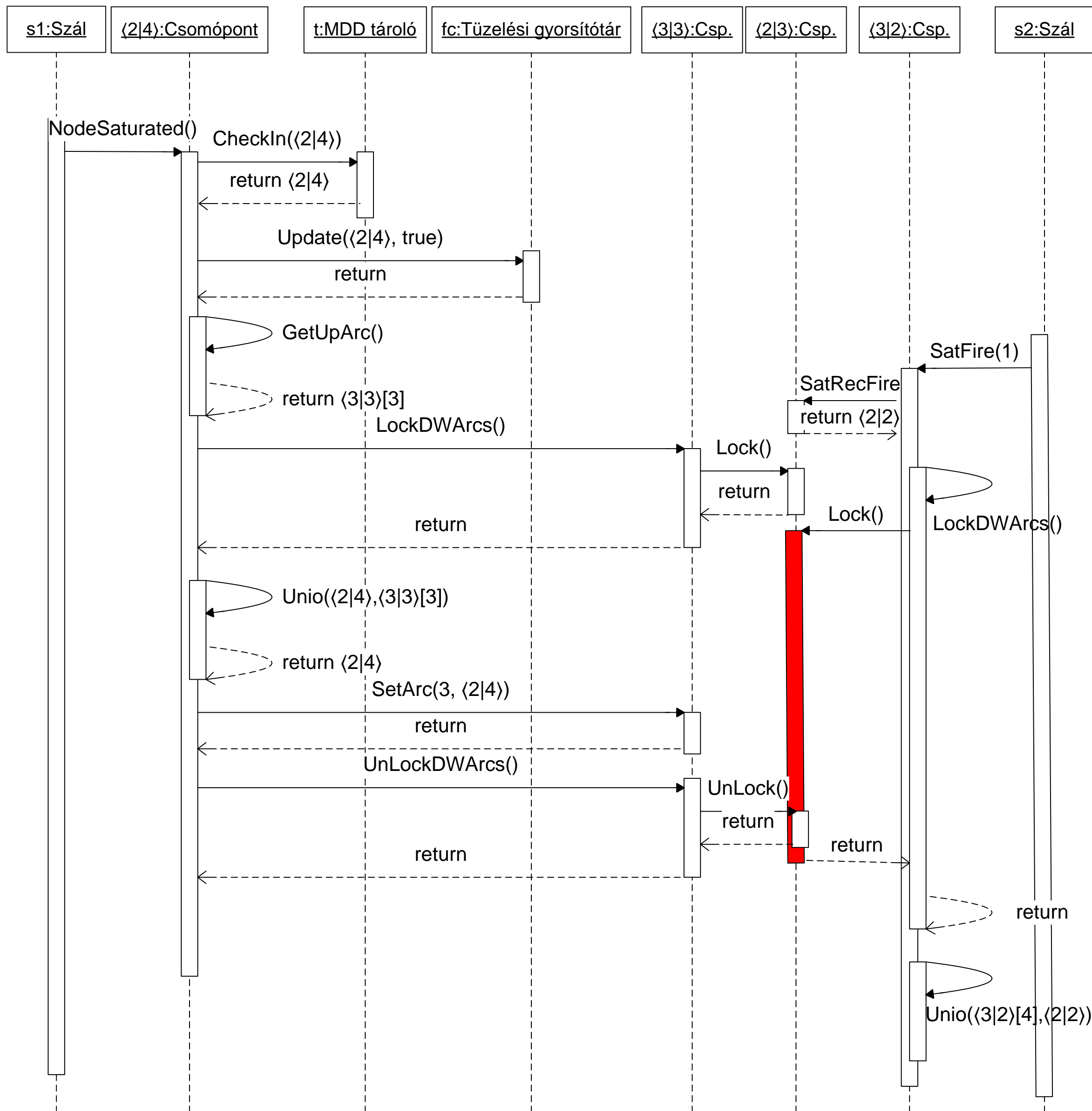
- A ⟨2|3⟩ csomópont már szaturált, a ⟨2|4⟩ csomópont szaturációja éppen elkészült, a ⟨3|2⟩ csomópont szaturációja még tart, a ⟨3|3⟩ csomópont szaturációja pedig félbe szakadt, mivel a ⟨2|4⟩ még nem készült el.
- Mivel a ⟨2|4⟩ csomópont szaturációja éppen elkészült, ezért rajta a *NodeSaturated* függvényt futtatja az 1. szál.
- Egy 2. szál a ⟨3|2⟩ csomóponton szeretne egy állapotot felderíteni, mivel azon jelenleg egy SatFire függvény hajtódik végre.
- Létezik egy e_1 esemény, ami a 3. szintet 1-ből 4 állapotba viszi

Ekkor a szaturáció folytatásának lépései:

1. szál	2. szál
A ⟨2|4⟩ csomópontra meghívja a NodeSaturated függvényt. Elhelyezi a csomópontot az MDD-tárolóban a CheckIn függvény meghívásával, amely a ⟨2|4⟩ csomóponttal tér vissza. A tüzelési gyorsítótárban frissíti a csomóponthoz tartozó bool értéket, hiszen a ⟨2|4⟩ csomópont most már szaturált.	Meghívja a ⟨3|2⟩ csomópont 1 állapotára a *SatFire* függvényt.
Lekéri a ⟨2|4⟩ felfelé mutató éleit. Mivel csak egy ilyen éle van, ezért visszakapja a ⟨3|3⟩[3]-ba mutatót.	Kiválasztja az engedélyezett események közül az e_1-t.
Zárat helyez el a ⟨3|3⟩ csomópont alatti részgráfon, azaz az ábra szerint a ⟨2|3⟩ csomóponton és annak gyerekein (amelyeket azonban mi nem tüntettünk fel ábránkon).	Eltüzeli az e_1 eseményt a ⟨3|2⟩[1] csomóponton, azaz a ⟨2|3⟩-n. A *SatRecFire* a tüzelési gyorsítótárban talált ⟨2|2⟩ csomópontot adja eredményül.

11.B ábra	
Meghívja az Unio függvényt a ⟨2|4⟩ és a ⟨3|3⟩[3] csomópontokra.	Kezdeményezi a ⟨3|2⟩ csomópont alatti részgráf zárolását, azonban azon az 1. szál tart fenn zárat. Ekkor ez a szál várakozásra kényszerül.
Befejezi az unio végrehajtását. Mivel a ⟨3|3⟩[3] korábbi értéke ZeroNode volt, ezért az unio eredménye ⟨2|4⟩ lett. Ezt behúzza a ⟨3|3⟩ 3. élébe.	
Feloldja az elhelyezett zárat.	
11.C ábra	
Mivel a ⟨3|3⟩ csomópont szaturációja már korábban elkezdődött, ezért az új, 3. állapotot is bejárja, a SatFire meghívásával.	Most már elhelyezheti saját zárát a ⟨3|2⟩ alatti részgráfon. Innentől folytatódhat a szál futása a ⟨3|2⟩[4] -ben levő csomópont és a ⟨2|2⟩ csomópont uniójával.

A szaturáció folyamatát szekvencia diagramon (11. ábra) is ábrázoltuk a könnyebb érthetőség érdekében. A diagramon pirossal kiemelve látható, amikor a 2. szál a szinkronizáció következtében várakozik.



11. ábra: a példa szekvencia diagramja

3.2.5 Értékelés

Az ismertetett szaturációs algoritmus implementálása közben magunk is megbizonyosodhattunk 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 mi is és a [4] 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 csomópontok alatti részgráfok zárolásakor további 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 következő fejezetben megvizsgáljuk 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. Tesztjeink során azt találtuk, hogy a futási idő jelentős részét a szálak közötti szinkronizáció tölti ki.

A [4] 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 [4] 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 kutatásunkat éppen ez indokolta, hogy megvizsgáljuk, képesek vagyunk-e előnyt kovácsolni a párhuzamos feldolgozás kínálta lehetőségekből.

4 Fejlesztéseink a párhuzamos algoritmuson

A 3. fejezetben közölt algoritmus és az alapjául szolgáló cikk sok helyen kiegészítésre szorult, illetve voltak olyan részek, melyre nem adtak implementációs részleteket. Ilyen volt például az MDD-hez történő párhuzamos hozzáférés szinkronizálása, melynek hatékony implementálása különösen fontos a futási idő szempontjából. A saját fejlesztéseink során megvizsgáltunk többféle szinkronizációs mechanizmust is. Jelen fejezetben az iteratív fejlesztési fázisok követhetők végig és megtalálható a kísérleti jelleggel implementált tranzíció-előretüzelési algoritmus leírása is.

4.1 Az MDD szinkronizálása

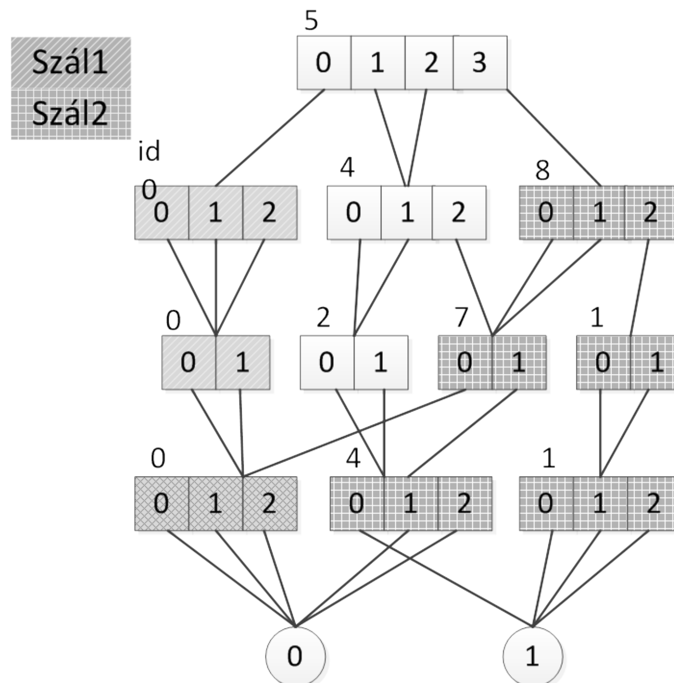
A szaturáció során MDD-ket használunk a felderített állapottér tárolására. Az MDD-khez való többszörös hozzáférést nehéz biztosítani, különösen az olyan bonyolult algoritmusok esetén, mint amilyen a párhuzamos szaturáció. A [4]-ben a részgráfok zárolását javasolják az MDD konzisztenciájának megőrzésére, azonban ez költséges és időigényes művelet, ezért mi ennek leváltására újfajta zárolást dolgoztunk ki.

4.1.1 Részgráfok zárolása

A [4]-ben javasolt, 3.2.2.4-ben 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 közlünk az implementációval kapcsolatban, a 12. 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.



12. 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 vetettük alá a problémát és a további fejezetekben láthatók a javított iterációs fázisok.

4.1.2 Read-write lock megvalósítás

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 fejlesztések alapját az úgynevezett írási-olvasási zárakon alapulva végeztük el. Az ilyen zár lehető 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 13. ábrán látható).

	olvasási zár	írási zár
olvasási zár	kompatibilis	nem kompatibilis
írási zár	nem kompatibilis	nem kompatibilis

13. ábra: Az írási-olvasási zár kompatibilitási mátrixa

A munkánk során felismertük 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 alkalmaztuk: a korábbi zárolással megegyező hatáskörben olvasási zárakat helyeztünk el, míg közvetlenül az élek állítását írási zárakkal láttuk el. A zárak effajta elhelyezésével biztosítottuk, 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 hoztunk létre, melyet globálissá tettünk a teljes állapottér felderítésére nézve. A szálak függetlenül attól, hogy az MDD melyik szintjén dolgoznak éppen, ugyan azt az objektumot használták szinkronizálásra. Ez igen pazarló működésnek tűnhet, azonban csak így tudtuk 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.

4.1.3 Lokális szinkronizációs mechanizmus

Az 4.1.1 és az 4.1.2 fejezetekben ismertetett megoldásoknak közös hátránya, hogy a zárak használata rendkívül lecsökkenti a szálak párhuzamos futásának arányát. Az előbbi esetben, mikor 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 mi végül egy 3. algoritmust fejlesztettünk. Az új technika lényege, hogy teljes részgráfok zárolása helyett csupán azokat a csomópontokat zároljuk, amelyeknek valamelyik élét éppen át szeretnénk állítani.

Az eddigiekben azért alkalmaztuk 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 jutottunk, 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[4], 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 műveletek azok nem vezetnek ki a valós állapottérből.

A konzisztens állapottereken végrehajtva a szaturációs algoritmust, a szaturáció konvergens tulajdonsága miatt [3][4][15] 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 általunk megvalósított lokális szinkronizációs algoritmus helyességét.

Az általunk 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 általunk 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.

4.2 Heurisztikán alapuló tranzíció-előretüzelés

A továbbiakban bemutatunk egy kísérleti jelleggel implementált technikát. A cél az volt, hogy a többprocesszoros rendszerek megnövekedett erőforrásait még hatékonyabban ki tudjuk használni és további sebességnövekedést tudjunk elérni a párhuzamos algoritmushoz képest.

4.2.1 Az előretüzelés működésének bemutatása

A párhuzamos algoritmus hatékonyságának elemzése során arra lettünk figyelmesek, hogy az egyes processzorok kihasználtságának összege az egész állapottér felderítése során csak a 70-80%-ot érte el, azaz a hardverben még voltak ki nem használt erőforrások. Célunk az volt, hogy a maradék erőforrásokat kihasználva tovább gyorsítsuk a párhuzamos algoritmust. Ennek kapcsán született meg a tranzíció-előretüzelési algoritmus (pre-firing algorithm) ötlete, melynek motivációját egy munkaállomások hálózatára optimalizált, összetett mintaillesztésen alapuló algoritmus adta [26]. Az itt közölt algoritmussal ellentétben a saját implementációnk egyedülálló multiprocesszoros gépekre lett kifejlesztve, az előretüzelés pedig heurisztikákon alapulva történik.

Ahogy azt már korábban leírtuk, az algoritmus az állapottér felderítése során a tüzelési gyorsítótárhoz fordul a csomópontokon eltüzelt tranzíciók eredményének lekérdezéséhez. Ha a gyorsítótárban már benne van a kérdéses elem, akkor nem szükséges a tranzíciót még egyszer eltüzelni a csomóponton, hiszen a gyorsítótárban (mely egy hash-tábla) levő érték már ezen tüzelés eredményét reprezentáló csomópont. Ebből látható, hogy az algoritmus akkor érheti el a legjobb futási időt, ha a gyorsítótárban minden szükséges érték megtalálható. Az ötletet pont ez az észrevétel adta; ha a többlet erőforrásokat arra használjuk ki, hogy a gyorsítótárba előre berakjuk ezeket az értékeket, akkor tovább növelhetjük a párhuzamos algoritmus sebességét.

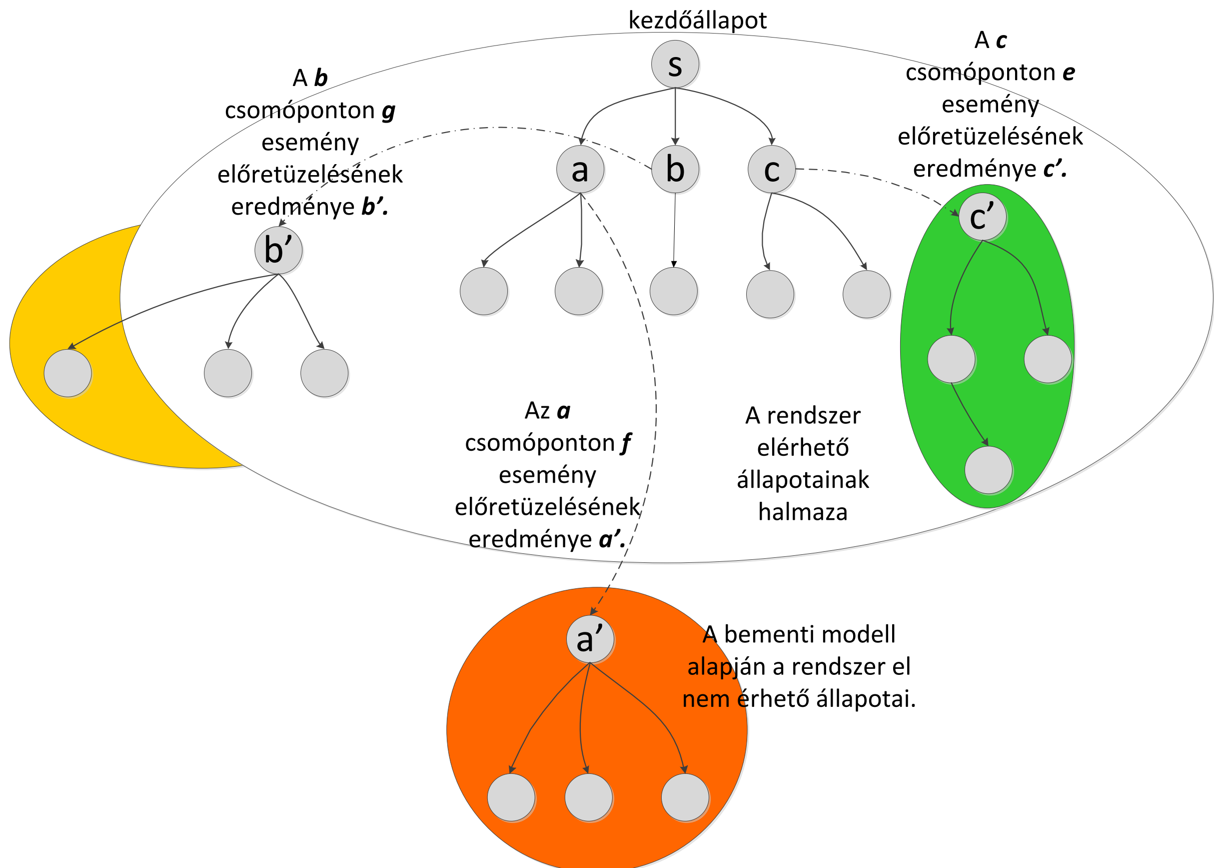
Az előretüzelés elvégzéséhez tehát szükség van egy csomópontra és a rajta eltüzelendő tranzícióra. Ezek kiválasztása azonban nem triviális feladat, ugyanis előfordulhat, hogy a kiszámított csomópont teljesen fölösleges olyan szempontból, hogy arra soha nem lesz szüksége az algoritmusnak a futása során. A szaturációs algoritmus sem hajtja végre minden csomóponton minden tranzíció tüzelését, csak azokat, melyek ténylegesen hatással vannak az adott csomópontra. Az előretüzeléshez tehát nem megfelelő az, hogy tetszőlegesen kiválasztunk egy csomópont-tranzíció párt, mert így nagy valószínűséggel sok lesz a fölösleges műveletvégzés, ezek számát pedig minimalizálni kell, mivel az ez idő alatt felhasznált erőforrásokat esetleg a szaturációs algoritmus futására lehetett volna felhasználni. Szélsőséges esetben ez akár az algoritmus lassulását is okozhatja. A leírtak alapján az előretüzelés hasznosságát három csoportba sorolhatjuk be:

1. az előretüzelés teljesen felesleges, mert a tüzelési gyorsítótárba bekerült értéket az algoritmus nem fogja felhasználni

2. az előretüzelés olyan csomópontot eredményezett, melynek részfája által reprezentált állapotok halmaza és a rendszer elérhető állapotainak halmaza metszete nem üres halmaz
3. az előretüzelés olyan csomópontot eredményezett, melynek részfája által reprezentált állapotok halmaza részhalmaza a rendszer elérhető állapotai halmazának. Ez az az eset, amelynek eredményét a szaturációs algoritmus fel tudja használni.

A megfelelő tranzíció kiválasztására az implementáció során különféle heurisztikákat alkalmaztunk, amelyek a csomópont tulajdonságait használják fel, ezekről bővebb információ az 4.2.2 fejezetben található.

A 14. ábra szemlélteti az előretüzelés működését. A szaturációs algoritmus a kezdőállapotból (*s*) kiindulva elkezdi felderíteni az elérhető állapotokat, ilyen például az *a,b* és *c* állapotok. Ezzel párhuzamosan végrehajtjuk a *c* csomóponton az *e* esemény (előre)tüzelését és az eredményt eltároljuk a tüzelési gyorsítótárban. A szaturációs algoritmus, amikor végre akarja hajtani ugyanezt a tüzelést, akkor erre már nem lesz szükség, mivel előre megcsináltuk azt, és a gyorsítótárból egyszerűen lekérhető az eredmény (3. eset). Ezzel szemben az *a* csomóponton az *f* esemény előretüzelése fölösleges volt (1. eset), mivel a szaturációs algoritmus sosem fogja megkísérelni végrehajtani ezt a tüzelést, mert az *f* esemény nem befolyásolja az *a* csomópontot. Az előretüzelés eredményeként létrejött *b'* csomópont a korábban említett 2. esetnek felel meg, így nem volt teljesen haszontalan a végrehajtása.



14. ábra: A tranzíció-előretüzelés szemléltetése

4.2.2 Az eltüzelendő tranzíció kiválasztásához használt heurisztikák

Fontos megjegyezni, hogy nem érdemes az összes tranzíció közül választani az előretüzelés során, sok olyan van, amely irreleváns az adott szinten. Minden szinten a tranzíciók egy szűkebb halmazából választunk, csak azokat vesszük figyelembe, melyek hatással vannak az adott szintre. Ezen tranzíciók halmazát heurisztikák alkalmazásával még tovább szűkítjük úgy, hogy az így megmaradt tranzíciókat eltüzelve az adott csomóponton az eredmény minél hasznosabbnak bizonyuljon.

A jelenlegi implementációban két heurisztikát próbáltunk ki:

- *Naiv heurisztika*: A csomópont szintjéhez tartozó tranzíciók közül egy olyat választ ki, amelynek az eltüzelése az egyszerű párhuzamos algoritmusban két szinttel feljebb történne meg. Ez azt jelenti tehát, hogy ha az eltüzelendő tranzíció az α és a csomópont szintje a k, akkor teljesülnie kell a $Top(\alpha) = k + 2$ egyenlőségnek. A heurisztika ily módon történő megválasztásának oka, hogy az előretüzelést az eredeti csomóponthoz viszonylag közel hajtjuk végre és az esemény lokalitás miatt e tüzelés eredményére nagy valószínűséggel szükség lesz a későbbiekben.

- *Uppermost (legmagasabb) heurisztika*: Azt a tranzíciót választja ki, melynek Top szintje a legnagyobb azok közül, melyek hatással vannak a csomópont szintjére is. Ezzel a módszerrel az eredeti és az előretüzelés eredményeként létrejött csomópont szintje viszonylag távol is kerülhet egymástól, a többlet erőforrásokat ilyenkor egy az állapottérben távoli rész felderítésére használjuk.

4.2.3 Implementációs részletek

Az előretüzelés végrehajtásához a korábban említett *SatRecFire* metódust vettük alapul. A lényeges különbség a kettő között, hogy míg a *SatRecFire-t* egy felsőbb szinten levő csomópontból hívjuk meg, ezzel szemben az előretüzeléshez nincs ilyen csomópont, hiszen itt csak előre szeretnénk dolgozni. E miatt az előretüzelést végrehajtó *PreFire* metódus paraméterezése is más, egy csomópont-tranzíció párt kell neki átadni. Az előretüzelést úgy implementáltuk az algoritmusban, hogy egy időben csak egy történhessen, mivel el akartuk kerülni a korábban vázolt problémát, miszerint az előretüzelésre túlzott mennyiségű erőforrást használnánk fel. Ehhez egy változóban (*actPreNodeId*) eltároljuk annak a csomópontnak az azonosítóját (*id*), melyet a *PreFire* hoz létre az előretüzelés elindításakor. Maga a tüzelési feladat egy éppen szaturálttá vált csomóponton fog meghívódni, azaz *NodeSaturated* hívás paraméterén. Az előretüzelés csak akkor indítható el, ha az *actPreNodeId* változó értéke 0. Kezdetben a változó 0-val van inicializálva, később eltároljuk benne egy csomópont azonosítóját, majd amikor befejeződik az előretüzelés – azaz a csomópont *saturated* attribútumát *true*-ra állítjuk - ismét 0 lesz az értéke.

Az eltüzelendő tranzíció kiválasztásához a *Strategy* [17] tervezési mintát használtuk fel, ezzel könnyen változtatható az, hogy éppen melyik heurisztikát szeretnénk alkalmazni. Érdekes folytatási lehetőség lehet olyan adaptív algoritmus kipróbálása, mely a csomópont tulajdonságai alapján automatikusan meg tudja állapítani a megfelelő heurisztikát a tranzíció kiválasztására.

Az előretüzelési feladatokat a szaturációt megvalósító *Saturate* függvényhívásokhoz hasonlóan a *ThreadPool*-nak átadott metódusreferenciákon keresztül hajtjuk végre (*PreQSaturate* metódus). A metódusnak két paramétert kell átadni:

- Azt a csomópontot, melyen az előretüzelést végre kell hajtani. A tranzíciót a csomópont tulajdonságai alapján valamely heurisztika segítségével választjuk ki.
- Egy logikai típusú változót, melynek értéke *true*, ha előretüzelést indítunk, *false,* ha a csomóponton *Saturate* hívást kell végrehajtani.

A *PreFire* metódus kódja az A függelékben található [A26]. Fontos megjegyezni, hogy az előretüzelés elindítása a *PreFire* metódus meghívásával történik, de a továbbiakban már *SatRecFire* hívások sorozatán keresztül szaturáljuk a csomópontot.

Mivel a *PreFire* hívásnál nem állt rendelkezésre felsőbb szinten levő csomópont, ezért minden olyan részt elhagytunk a *SatRecFire* kódjából, mely felfelé mutató élek beállítására vonatkozik. Megmaradt azonban itt is az a rész, mely a tüzelési gyorsítótárban keresést hajt végre, annak elkerülésére, hogy ha esetleg az eredeti szaturációs algoritmus már elkezdte végrehajtani az adott csomóponton az adott tranzíció eltüzelését, akkor azt ne hajtsuk végre még egyszer. Az *MDDNode* osztályt kiegészítettük egy *preFiring* nevű attribútummal, mely bool típusú, és azt jelöli, hogy a csomópont előretüzelés eredményeként jött-e létre vagy sem. A *PreFire* függvényben a létrehozott csomópontra ezt az attribútumot beállítjuk *true* értékre, ezt a későbbiekben felhasználjuk a *RecFire* és a *Remove* metódusokban is.

Az előretüzelés értékelése és a hozzá kapcsolódó mérési eredmények a 5. fejezetben találhatók.

5 Mérési eredmények

A szaturáció és az eredmények tárolására használt döntési diagramok szekvenciális tulajdonságai miatt nehéz feladat a párhuzamosítás, amely tulajdonsággal a [4]-ben is szembesültek. Az ott szereplő, C nyelven implementált párhuzamos szaturációs algoritmus a szekvenciális változathoz képest csak a véletlenszerűen generált modellekre hozott gyorsulást a szekvenciális változathoz képest, míg a valós problémák egyszerűsített modelljeire rendre lassabb futást produkált. Munkánk során ezen próbáltunk javítani, azonban a probléma nehézsége miatt célunk az volt, hogy a cikkben szereplő párhuzamos futási sebességeket elérjük, és némely modellre felülmúljuk.

Eredményeink bemutatásához méréseket végeztünk. Az ezek során használt szoftver és hardver komponensek a 5.1 fejezetben kerültek bemutatásra. A 5.2-5.5 fejezetekben találhatóak a méréseink eredményei.

5.1 A mérési környezet

A mérési infrastruktúrát két számítógép alkotta:

- Intel Quad Core 4 magos számítógép, 4GB memóriával. A méréseket Windows 7 Professional operációs rendszeren végeztük el. A továbbiakban erre desktop gépként fogunk hivatkozni.
- Sun SunFire X4600 szerver 8 db duplamagos AMD processzorral, egyenként 2,5GHz és 32 GB memóriával. A hardveren ESXi virutalizációs szoftvert futtattunk, ez lekorlátozta a felhasználható processzorok és memória méretét, így a processzorok közül 4-et (8 mag), a memóriából 24GB-ot használtunk fel a mérések elvégzéséhez. A host operációs rendszer Windows Server 2008 volt. A továbbiakban erre szerverként fogunk hivatkozni.

Mindkét számítógépen a .NET 4.0 keretrendszer 64 bites verziója fut. A szálak ütemezéséhez a .NET által biztosított ThreadPool osztályt használtuk fel.

A mérésekhez felhasznált modellek

A méréseket többféle modell Petri háló alapú reprezentációján végeztük el:

- **Slotted Ring**: egy hálózati protokoll, melyben a résztvevő felek gyűrű topológián keresztül kommunikálnak. A mérések során a Slotted Ring N arra utal, hogy hány résztvevőt tartalmaz a gyűrű. További információk a [B1]-ben olvashatók.

- **Flexible Manufacturing System (FMS)**: termelési folyamatot ellátó rendszert modellez. A rendszer képes alkalmazkodni a véletlenül bekövetkezett vagy az előre tervezett változásokhoz és ehhez alakítani az előállított termékek számát. A mérések során az FMS N, arra utal, hogy mennyi a termelés alatt álló termékek száma [B2].

- **Kanban modell**: egy termelési folyamat ütemezésének modellje, hatékonyan alkalmazható olyan kérdésekben, hogy mely termékből mikor és mennyit kell előállítani. A modell paraméterezhető, erre utal a később látható Kanban N jelölés, részletes leírás a [B3]-ben található.

A szekvenciális változattal összehasonlító méréseinket a desktop számítógépen végeztük el, míg szerveren a Flexible Manifacturing System és Slotted Ring modellekkel olyan méréseket is elvégeztünk, amelyek azt vizsgálták, hogyan skálázódik a párhuzamos algoritmusunk, ha különböző számú processzort biztosítunk a működéséhez.

5.2 Slotted Ring

Az alábbiakban a Slotted Ring hálózati kommunikációs protokoll állapottér felderítésének idejét vizsgáltuk különböző beállításokkal.

A méréseket elvégeztük a *PetriDotNet* által kínált többféle szintezési mód segítségével (Szintezés). A méréshez kétféle szintezési beállítást használtunk: először megvizsgáltuk a P-invariáns alapú szintezést, mely a Petri hálót kis részekre bontja, minden egyes komponenst pontosan két részre, melyeknek a lokális állapottere szűk. A második esetben a kézenfekvő minden komponenst külön szintre elhelyező felbontást választottuk (manuális, szintenként 8 hely). Ilyenkor a lokális állapottér mérete nagyobb, azonban a modell kevesebb szintre bomlik, növelve a lehetséges szinkronizációs többletköltséget.

Slotted Ring N				
Szintezés	N	Típus	Futási idő [s]	Arány [%]
P-invariáns	50	Szekvenciális	1,642	100,000
		Párhuzamos	1,688	102,801
		Párhuzamos előret. (uppermost)	1,902	115,834
		Párhuzamos előret. (naiv)	2,090	127,284
	100	Szekvenciális	14,656	100,000
		Párhuzamos	13,206	90,106
		Párhuzamos előret. (uppermost)	13,930	95,046
		Párhuzamos előret. (naiv)	15,656	106,823
	150	Szekvenciális	70,350	100,000
		Párhuzamos	42,246	60,051
		Párhuzamos előret. (uppermost)	44,547	63,322
		Párhuzamos előret. (naiv)	48,640	69,140
Manuális (szintenként 8 hely)	50	Szekvenciális	2,612	100,000
		Párhuzamos	4,200	160,796
		Párhuzamos előret. (uppermost)	5,038	192,879
		Párhuzamos előret. (naiv)	5,298	202,833
	100	Szekvenciális	22,496	100,000
		Párhuzamos	32,410	144,070
		Párhuzamos előret. (uppermost)	39,096	173,791
		Párhuzamos előret. (naiv)	40,320	179,232
	150	Szekvenciális	97,726	100,000
		Párhuzamos	105,026	107,470
		Párhuzamos előret. (uppermost)	135,033	138,175
		Párhuzamos előret. (naiv)	131,580	134,642

***15*. ábra: mérési eredmények a Slotted Ring modellre**

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

Vizsgáltuk továbbá, hogy milyen hatással van a modell mérete (N) a futási időkre nézve a program egyes változataiban (Típus). Az azonos mérési beállítások eredményeit egymáshoz hasonlítottuk (Arány), ahol a 100%-nak a szekvenciális változat mindenkori futását tekintettük. Eredményeinket a 15. ábra tartalmazza.

A táblázatban szereplő mérési eredmények alapján megfigyelhető, hogy a modellek méretének növekedésével P-invariánsos szintezésnél a párhuzamos algoritmus előnye egyre inkább nő. Slotted Ring 50-nél még nem tapasztalható gyorsulás, de Slotted Ring 100-ra már 10%, 150-re pedig már 40% körüli a futási időbeli gyorsulás. Manuális szintezés esetén azonban csak Slotted Ring 150-nél kezdi megközelíteni a párhuzamos változat a szekvenciálisat. Jól látható, hogy a feladatmennyiség növekedésével mindkét esetben a párhuzamos algoritmus által jelentett szinkronizációs költségek hatása csökken, a párhuzamosítás hatékonysága pedig növekszik.

A jelenlegi előretüzelési algoritmus még nem gyorsabb, mint a párhuzamos változat, de itt is megfigyelhető az a tendencia, hogy a szekvenciális változattal szemben a P-invariánsos szintezés esetén nő az előny, míg manuális szintezés esetén csökken a hátrány a futási időt tekintve.

5.3 FMS

Az FMS modellre vonatkozó mérési eredmények a 16. ábrán láthatók, amely az előretüzelésre vonatkozó stratégiákkal is ki van egészítve. Az előretüzeléses esetekben a hatékonyág vizsgálatához feltüntettünk két új mérőszámot:

- Előretüzelés találati arány: megmutatja, hogy az előretüzelés következményeként létrejött csomópontok hány százalékát használja fel ténylegesen a szaturációs algoritmus.
- Előretüzelés csomópont referencia átlag: azt mutatja meg, hogy a szaturációs algoritmus által felhasznált, előretüzelés következményeként létrejött csomópontokra hány referencia mutat az algoritmus végén. A referencia itt azt jelenti, hogy a felsőbb szinten hány csomópontból mutat él az adott csomópontra a szaturáció végén.

A táblázat alapján jól látható, hogy az FMS modell itt megvizsgált minden esetére gyorsabb volt a párhuzamos algoritmus, mint a szekvenciális. Kiemelendő, hogy az FMS 300-ra a párhuzamos algoritmussal már 80% körüli gyorsulást értünk el. Megfigyelhető az is, hogy az előretüzelés hatása nem elhanyagolható: hatékonysága ugyan változó – mely főként a heurisztikáknak tudható be - de olykor további sebességnövekedés is elérhető alkalmazásával (lásd FMS 8).

Az előretüzelés vizsgálatakor arra lettünk figyelmesek, hogy a manuális szintezés esetén sokkal nagyobb lett a találati arány, mint a P-invariánsos esetben. A nagy referencia szám azt mutatja, hogy az előretüzelés következtében az MDD-nek olyan részét sikerült előre elkészíteni, amelyet a szaturációs algoritmus is felhasznált a futása során. Ez függ a szintezéstől is; P-invariánsos szintezés esetén nagyobb a szintenkénti lokális állapottér, ezért az átlagos referencia szám is nagyobb. Mindkét esetre elmondható azonban, hogy a referencia szám az adott szintezéshez viszonyítva nem kevés, ebből a szempontból hatékony az előretüzelés.

FMS N						
Szintezés	N	Típus	Futási idő (s)	Arány (%)	Előretüzelés találati arány	Előretüzelés csomópont ref. átlag
P-invariáns	6	Szekvenciális	6,835	100,000	-	-
		Párhuzamos	5,960	87,198	-	-
		Párhuzamos előret. (uppermost)	6,305	92,246	35,897	247
		Párhuzamos előret. (naiv)	5,985	87,564	28,571	355
	8	Szekvenciális	157,320	100,000	-	-
		Párhuzamos	140,705	89,439	-	-
		Párhuzamos előret. (uppermost)	149,225	94,854	40,000	913
		Párhuzamos előret. (naiv)	145,110	92,239	36,111	1250
	10	Szekvenciális	1879,485	100,000	-	-
		Párhuzamos	1691,115	89,978	-	-
		Párhuzamos előret. (uppermost)	1678,390	89,301	19,640	3456
		Párhuzamos előret. (naiv)	1702,645	90,591	19,047	4388
Manuális (szintenként 1 hely)	100	Szekvenciális	14,483	100,000	-	-
		Párhuzamos	8,253	56,984	-	-
		Párhuzamos előret. (uppermost)	10,847	74,895	91,806	33
		Párhuzamos előret. (naiv)	9,193	63,474	53,570	34
	200	Szekvenciális	179,383	100,000	-	-
		Párhuzamos	71,406	39,806	-	-
		Párhuzamos előret. (uppermost)	79,113	44,103	91,974	64
		Párhuzamos előret. (naiv)	71,973	40,123	92,941	70
	300	Szekvenciális	939,047	100,000	-	-
		Párhuzamos	190,673	20,305	-	-
		Párhuzamos előret. (uppermost)	286,320	30,490	91,480	100
		Párhuzamos előret. (naiv)	269,890	28,741	92,426	115

***16*. ábra: Mérési eredmények az FMS modellre**

5.4 Kanban

A Kanban modellre vonatkozó mérési eredmények a 17. ábrán láthatók.

A manuális szintezések közül két esetre végeztünk méréseket; szintenként 1 (A eset) illetve 2 (B eset) hely elkódolásával. Az A esetben a kialakuló szintek száma több, egy-egy szinten azonban kevesebb a lokális állapotok száma. A cél itt a manuális szintezések hatásának vizsgálata volt a futás időre vonatkozóan. Az A esetben a modell méretének növekedésével párhuzamosan az algoritmus fokozatos gyorsulása tapasztalható, míg a B esetben a fokozatos lassulás. Ennek oka, hogy a B esetben az MDD-ben létrejövő szintek száma kevesebb, ezért az egyes szálak nagyobb valószínűséggel fognak azonos szinten dolgozni. Ennek következtében a szálak többet fognak várakozni a záraknál.

Kanban N				
Szintezés	N	Típus	Futási idő [s]	Arány [%]
Manuális (szintenként 1 hely)	100	Szekvenciális	5,890	100,000
		Párhuzamos	2,230	37,861
		Párhuzamos előret. (uppermost)	2,805	47,623
		Párhuzamos előret. (naiv)	2,870	48,727
	150	Szekvenciális	26,780	100,000
		Párhuzamos	6,785	25,336
		Párhuzamos előret. (uppermost)	8,695	32,468
		Párhuzamos előret. (naiv)	8,430	31,479
	200	Szekvenciális	71,945	100,000
		Párhuzamos	15,310	21,280
		Párhuzamos előret. (uppermost)	20,370	28,313
		Párhuzamos előret. (naiv)	19,775	27,486
Manuális (szintenként 2 hely)	60	Szekvenciális	7,320	100,000
		Párhuzamos	4,965	67,828
		Párhuzamos előret. (uppermost)	4,780	65,301
		Párhuzamos előret. (naiv)	5,060	69,126
	80	Szekvenciális	20,965	100,000
		Párhuzamos	17,830	85,047
		Párhuzamos előret. (uppermost)	17,470	83,329
		Párhuzamos előret. (naiv)	18,250	87,050
	100	Szekvenciális	49,050	100,000
		Párhuzamos	55,140	112,416
		Párhuzamos előret. (uppermost)	52,810	107,666
		Párhuzamos előret. (naiv)	56,845	115,892

17. ábra: Mérési eredmények a Kanban modellre

5.5 A futási idő skálázódása

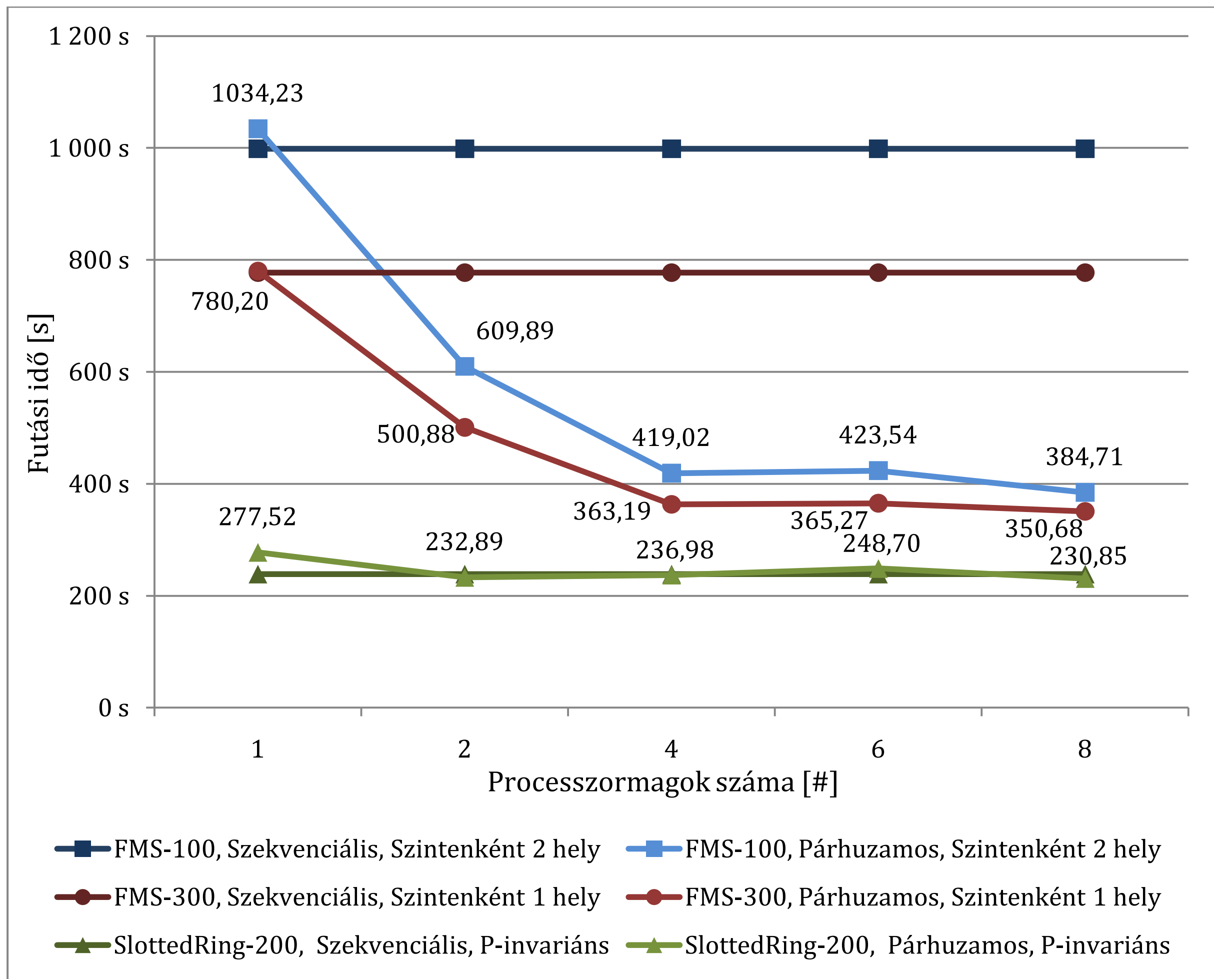
A Sun szerver gép lehetőségeit kihasználva méréseket végeztünk a skálázódás vizsgálatára. A mérés célja a párhuzamos skálázódás vizsgálata volt a különböző modelleken a processzormagok számának tekintetében.

A mérést 3 modellen futtattuk le:

- Slotted Ring 200 - állapottér mérete: $8{,}4 \cdot 10^{211}$, *p invariáns* (4 helyet tesz egy szintre)
- FMS 100 – állapottér mérete: $2{,}7 \cdot 10^{21}$, *every 2* szintezés, azaz minden szinten két hely van kódolva
- FMS 300 – állapottér mérete: $\sim 10^{25}$, *every 1* szintezés, azaz minden szinten egy hely van kódolva

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

A mérési eredmények a 18. ábrán láthatóak. A diagramon a vízszintes tengelyen az algoritmus számára biztosított processzormagok vannak feltüntetve, míg a függőleges tengelyen a futási időket jelöltük másodperces értékekben megadva. Összehasonlításként vízszintes vonalakkal feltüntettük a szekvenciális algoritmus futási idejét is az egyes modellekhez és szintezésekhez.



18. ábra: A futási idő skálázódása a processzor magok függvényében

Az ábrán látható, hogy a vizsgált modellek esetében 1 processzormagot használva nem mutatkozik jelentős különbség a szekvenciális és párhuzamos algoritmus teljesítményében, amiből azt következtettük, hogy sikeresen csökkentettük a zárolás által okozott többletköltséget. Az FMS modellek estében a magok számával folyamatosan csökken a futási idő: 2 mag esetén 10 perc, míg 8 mag esetén már csak 6 perc szükséges egy FMS100 modell teljes állapotterének generálásához. A párhuzamos szaturáció eredményeit tekintve esetünkben 4 magnál tört le az algoritmus skálázódása és onnantól nem jelentkezett jelentős gyorsulás. Ez magyarázható sok szál következtében egyre inkább növekvő mértékű overheaddel, illetve összefüggésben áll Amdahl törvényével is, amely a problémák párhuzamosíthatóságára ad egy közelítő felső korlátot.

Az FMS különböző szintezéseivel végzett mérések vizsgálatakor megfigyelhető, hogy a párhuzamosíthatóság szempontjából fontos a modellek dekomponálása is, amely hatással van a szálanként bejárható lokális állapottér méretére. Nagyobb állapottér esetén az szálak nagyobb mértékben tudnak önállóan, párhuzamosan dolgozni. A különféleképpen

dekomponált szintek továbbá befolyásolják az állapottér-bejárás eredményét tároló döntési diagram struktúráját, amelyen keresztül a dekompozíció közvetve meghatározza az MDD-hez való kölcsönös hozzáférés szinkronizációs költségét is.

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

Habár a diagramon csak a Sun szerveren végzett méréseink eredményét közöltük, méréseinket a 4 magos desktop gépen is elvégeztük. Azonos feltételeket biztosítva, azaz a szervernek is 4 magot adva, az eredmények a desktop gépen voltak alacsonyabbak. Tapasztalataink szerint az alkalmazott hardver architektúra és az abban rejlő párhuzamos képesség nem meglepő módon szignifikánsan befolyásolja a mérések eredményét.

6 Összefoglalás, értékelés

A TDK dolgozat eredményeként egy párhuzamos állapottér-felderítő modullal egészítettük ki a PetriDotNet modellellenőrző keretrendszert. Ez az algoritmus képes a multiprocesszoros hardverek többlet erőforrásait hatékonyan kihasználni, ezáltal gyorsítva a rendszer ellenőrzésére fordított időt.

Az implementáció alapjául a [4]–ben közölt algoritmus szolgált, amelyet több szempontból is továbbfejlesztettünk:

- a hivatkozott dokumentumban közölt szinkronizációs mechanizmus helyett egy kevesebb erőforrást felhasználó, hatékonyabb zárkezelést implementáltunk
- az adatszerkezeteket oly módon egészítettük ki, hogy az a párhuzamos algoritmus hatékonyságát szolgálja
- kísérleti jelleggel egy heurisztikán alapuló, tranzíció-előretüzelési algoritmust is kifejlesztettünk, melytől további sebességnövekedést várunk.

Az általunk megvizsgált irodalmakban közölt eredmények nem számolnak be a párhuzamosítás következtében elért, átütő eredményekről [4]. Ezzel szemben a mérési eredményeink alapján elmondható, hogy az implementációnkkal szignifikáns- akár 80%-os – sebességnövekedést is sikerült elérnünk, ugyanakkor azokra az esetekre, ahol nem lett gyorsabb a párhuzamos algoritmus, nem maradt el jelentősen a szekvenciális változathoz képest.

Fontos megjegyezni azonban, hogy az elérhető sebességnövekedés nagymértékben függ több tényezőtől is (5. fejezet):

- a vizsgált modell strukturális tulajdonságaitól
- a kialakuló döntési diagram szintjein kódolt állapotok számától és ez alapján a szintek számától
- az alkalmazott párhuzamos hardver architektúrától
- alkalmazunk-e előretüzelést az algoritmus futása során

Az elkezdett munka rengeteg továbblépési lehetőséget hordoz magában, amelyek közül az alábbiak tűnnek perspektivikusnak számunkra:

- Az előretüzelési algoritmus továbbfejlesztése, mind az algoritmus hatékonysága, mind a kipróbált heurisztikák száma szempontjából. Célunk olyan adaptív heurisztikák implementálása is a továbbiakban, mely képes az algoritmus futása közben alkalmazkodni és kiválasztani a legmegfelelőbb tranzíció kiválasztási módot.
- Elosztott modellellenőrző algoritmus kifejlesztése, mely munkaállomások hálózata által nyújtott erőforrásokat képes hatékonyan kihasználni.
- Párhuzamos CTL modellellenőrző modul kifejlesztése a PetriDotNet programban jelenleg megtalálható szekvenciális változat mellé, így a modellellenőrzés teljes folyamata során kihasználható válnak a multiprocesszoros hardverek által nyújtott többlet erőforrások.
- Az algoritmus strukturális módosítása, hogy alkalmas legyen az állapottér dekompozíciójával nyerhető lehetőségek kihasználására, így érve el jobb skálázódást több processzormag alkalmazása esetén.

7 Irodalomjegyzék

[1] PetriDotNet modellellenőrző keretrendszer információs oldala (BME-MIT): https://www.inf.mit.bme.hu/research/tools/petridotnet

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

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

[4] 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. 2006:331-346

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

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

[7] 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

[8] Yen, Hc. 2006. “Introduction to Petri net theory.” Recent Advances in Formal Languages and Applications.

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

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

[11] 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.

[12] U. Stern and D. L. Dill, “Parallelizing the murphi verifier,” in *International Conference on Computer Aided Verification*, 1997, pp. 256–278.

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

[14] Foster, Ian (1995) Designing and Building Parallel Programs (Online) Addison-Wesley ISBN 0201575949, chapter 8 Message Passing Interface

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

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

[17] Strategy tervezési minta: http://www.dofactory.com/Patterns/PatternStrategy.aspx

[18] 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, p.374-391, June 26-30, 1995

[19] 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

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

[21] Miller, D.M.; Drechsler, R.; , "Implementing a multiple-valued decision diagram package," *Multiple-Valued Logic, 1998. Proceedings. 1998 28th IEEE International Symposium on* , vol., no., pp.52-57, 27-29 May 1998

[22] Miller, D.M.; Drechsler, R.; , "On the construction of multiple-valued decision diagrams," *Multiple-Valued Logic, 2002. ISMVL 2002. Proceedings 32nd IEEE International Symposium on* , vol., no., pp.245-253, 2002

[23] E. Pastor, O. Roig, J. Cortadella, and R. Badia. Petri net analysis using boolean manipulation. In R. Valette, editor, Proc. 15th Int. Conf. on Applications and Theory of Petri Nets, LNCS 815, pages 416–435, Zaragoza, Spain, June 1994. Springer-Verlag.

[24] E. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in model checking. In CAV '93, LNCS 697, pages 450–462. Springer-Verlag, 1993.

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

[26] Chung M-Y, Ciardo G. Speculative Image Computation for Distributed Symbolic Reachability Analysis. *Journal of Logic and Computation*. 2009:1-19.

[27] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov. Evaluating the "Small Scope Hypothesis". Technical Report MIT-LCS-TR-921, MIT CSAIL, 2003.

[28] The Common Criteria standard documents (http://www.niap-ccevs.org/cc-scheme/cc_docs/)

[29] Z. Micskei and H. Waeselynck: The many meanings of UML 2 Sequence Diagrams: a survey Software and Systems Modeling, Springer, Online first, DOI:10.1007/s10270-010-0157-9, 2010

[30] Miller, S. 2009. "Bridging the Gap Between Model-Based Development and Model Checking." Tools and Algorithms for the Construction and Analysis of Systems: 443–453.

[31] SCADE (2010. október 21.): http://www.esterel-technologies.com/products/scade-suite/modeler

[32] Darwin- A Theorem Prover for the Model Evolution Calculus : http://goedel.cs.uiowa.edu/Darwin/

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

Függelék A

[A1] A szekvenciális szaturáció SatFire függvénye

```
SatFire(e : event, k :level, p : index): index
A ⟨k|p⟩ csomóponton eltüzeli az e eseményt, ahol k = Top(e). A helyben
frissítés módszerét alkalmazzuk, olyan x csomóponttal térünk vissza melyre
teljesül, hogy B(⟨k|x⟩) = N*_{k,e}(B(⟨k|p⟩)).

1 f, u : index, i,j : local, L : local-ok halmaza
2 L = {i_k ∈ S_k : ⟨k|p⟩[i_k] ≠ 0 ∧ N_{k,e}[i_k,·] ≠ 0} //azaz azokat a lokális állapotokat
     gyűjtjük össze, melyekből az e esemény tüzelésének hatására új
     állapotba jutunk
3 while L ≠ 0 do
4    vegyünk egy tetszőleges i állapotot L-ből
5    f = SatRecFire(e, k-1, ⟨k|p⟩[i]) //rekurzív hívás az alsóbb szinten levő
                                  csomópontra
6    if  (f ≠ 0) then
7          minden olyan j-re melyre N_{k,e}[i,j] = 1 //minden i-ből elérhető j
                                                        állapotra
8                u = Union(k-1, f,  ⟨k|p⟩[j])
9                if (u ≠ ⟨k|p⟩[j]) then //ha az unió új csomópontot
                                              eredményezett
10                           ⟨k|p⟩[j] = u //beállítjuk az élet
11                           if (N_{k,e}[j, ·] ≠ 0) then //mindaddig visszarakjuk
                                  a j-t amíg új állapotot érhetünk el a
                                  tüzeléssel
12                           L = L ∪ {j}
13 p = CheckIn(k, p)
14 return p
```

[A2] A szekvenciális szaturáció SatRecFire függvénye

```
SatRecFire(e : event, l :level,q : index): index
Az e esemény rekurzív tüzelése az ⟨l|q⟩ csomóponton, ahol Top(e) ≥ l ≥ Bot(e).

1 f, u, s : index, i, j : local, L : local-ok halmaza
2 if (l < Bot(e)) then return q
3 if Cached(FIRE, l, e, q, s) then return s
4 s = NewNode(l)
5 L = {i_l ∈ S_l : ⟨l|q⟩[i_l] ≠ 0 ∧ N_{l,e}[i_l,·] ≠ 0}
6 while L ≠ 0 do
7    vegyünk egy tetszőleges i állapotot L-ből
8    f = SatRecFire(e, l-1, ⟨l|q⟩[i]) //rekurzív hívás az alsóbb szinten levő
                                  csomópontra
9    if  (f ≠ 0) then
10         minden olyan j-re melyre N_{l,e}[i,j] = 1 //minden i-ből elérhető j
                                                        állapotra
11               u = Union(l-1, f,  ⟨l|s⟩[j])
12               if (u ≠ ⟨l|s⟩[j]) then //ha az unió új csomópontot
                                              eredményezett
13                     ⟨l|s⟩[j] = u //beállítjuk az élet
14 s = CheckIn(l, s)
15 PutInCache(FIRE, l, e, q, s)
16 return s
```

[A3] A párhuzamos szaturáció Saturate függvénye

```
Saturate(in: k: szint, p: index)
// Helyben frissíti a ⟨k|p⟩ csomópontot,
// amely így megfelel 𝒩*≤k(ℬ(⟨k|p⟩))-nek.

1 i: lokális állapot
2 ⟨k|p⟩.saturating = true; // jelezzük a szaturáció kezdetét
3 AddOp(k,p); // feljegyezzük, hogy egy szál éppen dolgozik a csomóponton
4 foreach i ∈ S^k do
5    if ⟨k|p⟩[i] ≠ ⟨k−1|0⟩ then  // azon lokális állapotokban,
                                  //amelyek engedélyezve vannak,
6          SatFire(k,p,i); // kimerítően eltüzeljük az összes eseményt
7 if RemoveOp(k,p) then //ez a szál befejezte a feldolgozást a csomóponton
8    NodeSaturated(k,p); // ha más sem dolgozik rajta,
                       //akkor elkészült a csomópont szaturációja
```

[A4] A párhuzamos szaturáció SatFire függvénye

```
SatFire(in: k: szint, p: index, i:lokális állapot)
// Eltüzeli azon e eseményeket a ⟨k|p⟩[i] csomóponton, amelyekre 𝒩e^k ≠ ⟨k|0⟩.

1 e: esemény; j: lokális állapot; u: index
2 foreach e ∈ ℰ^k
3    if  𝒩e^k(i) ≠ ⟨k|0⟩ then // minden engedélyezett esemény
4          f = SatRecFire(e, k-1, ⟨k|p⟩[i], p, i); // eltüzeljük az eseményt
5          if f ≠ ⟨k−1|0⟩ then
6                Lock(⟨k|p⟩.dw); //zároljuk a csomópont alatti részgráfot
7                j = GetTargetState(k, i, e);
8                u = Union(k-1, f, ⟨k|p⟩[j]);
9                if u ≠ ⟨k|p⟩[j] then // ha az unió eredménye változást hoz
10                     ⟨k|p⟩[j] = u;  // akkor frissítjük az éleket
11                     Unlock(⟨k|p⟩.dw);
12                     Confirm(k, j);
13                     SatFire(k, p, j);
14               else
15                     Unlock(⟨k|p⟩.dw);
```

[A5] A párhuzamos szaturáció SatRecFire függvénye

```
SatRecFire(in: e: esemény, l: szint, q: index, p: index, i: lokális
állapot): index
// Egy új, ⟨l|s⟩ gyökerű MDD-t hoz létre, amelyre teljesül, hogy 𝒩*≤l(𝒩e(ℬ(<
l,q >))).

1  ℒ: lokális állapotok halmaza; g,h,j: lokális állapot; f,u,s:index; sat:bool
2  if l < Bot(e) then // az esemény nincs hatással erre a szintre
3    return q;
4  Lock(FC(l)); // a módosítás erejéig zároljuk az FC-t
5  if Find(FC(l), Key(l, q, e), s, sat) then
6    if !sat then // ha a megtalálta csomópont még nem szaturált
7          j = GetTargetState(l+1, i, e); // az e esemény i-ből j
                                          // állapotba visz
8          SetUpArc(l, s, p, j);
9          s = ⟨l|0⟩; // mivel s még nem szaturált, ZeroNode a visszatérés
10   Unlock(FC(j));
11   return s;
12 s = NewNode(l); // ha nem volt az FC-ben találat, új csomópontot
```

```
                //hozunk létre
13 s.Key = Key(l, q, e);
14 j = GetTargetState(l+1, i, e); // az e esemény i-ből j állapotba visz
15 SetUpArc(l, s, p, j);
16 AddOp(l,s); // feljegyezzük, hogy egy szál éppen dolgozik a csomóponton
17 Insert(Fc(l), s.Key, s, false);
18 Unlock(FC(l));
19 ℒ = Locals(l, q, e); // azok az állapotok, amelyek e-re engedélyezettek
20 while ℒ ≠ 0 do
21   g = Pick(ℒ);
22   f = SatRecFire(e, l-1, ⟨l|q⟩[g], s,  g); // g állapotra eltüzeljük e-t
23   if f ≠ ⟨l − 1|0⟩ then // ha nem ZeroNode volt a RecFire visszatérése,
                          // azaz az FC-ben már benne volt a tüzelés
                          // eredménye
24         Lock(⟨l|s⟩.dw); // a módosítás erejéig zároljuk a csomópontot
25         j = GetTargetState(l, g, e);
26         u = Union(l-1, f, ⟨l|s⟩[j]); // unió az él korábbi értékével
27         if u ≠ ⟨l|s⟩[j] then
28               ⟨l|s⟩[j] = u;
29               Unlock(⟨l|s⟩.dw);
30               Confirm(l, j);
31         else
32               Unlock(⟨l|s⟩.dw);
33 if RemoveOp(l, s) then // ha más nem dolgozik s-n, és nem mutat rá
                            // felfelé él
34   if DWarcs(l, s) then
35         QSaturate(s); // ha volt tüzelhető esemény, akkor szaturáljuk
36   else
37         Remove(l, s); // egyetlen eseményt sem tudtuk eltüzelni, akkor
                          // fölösleges a csomópont, és törölhető
35   return ⟨l|0⟩;
```

[A6] A párhuzamos szaturáció NodeSaturated függvénye

```
NodeSaturated(i:n k: szint, p: index)
// Hozzáadja a ⟨k|p⟩ csomópontot az UT(k)-hoz, majd feldolgozza a ⟨k|p⟩
// csomópontba mutató felfelé éleket.

1  q: index, i: lokális állapot
2  q = p; // elmentünk egy hivatkozást a csomópontra
3  p = CheckIn(k, p);
4  if k = K then // ha a legfölső szintű csomópontot szaturáltuk,
               // akkor elkészült a szaturáció
5    Terminate();
6    return;
7  Lock(FC(k));
8  Update(FC(k), ⟨k|p⟩.Key, p, true); // frissítjük az FC-t
9  Unlock(FC(k));
10 while GetUpArc(k, p, r, i) do
11   Lock(⟨k + 1|r⟩.dw);
12   u = Union(k, p, ⟨k + 1|r⟩[i]); // a föntebbi csomópontokban beállítjuk
                                   // az elkészült csomópontot
13   if u ≠ ⟨k + 1|r⟩[i] then
14         ⟨k + 1|r⟩[i] = u;
15         Unlock(⟨k + 1|r⟩.dw);
16         Confirm(k+1, i);
17         if ⟨k + 1|r⟩.saturating then
18               SatFire(k+1, r, i); //az új állapotokban is eltüzeljük
                                        //az összes eseményt
```

```
19   else
20         Unlock(⟨k+1|r⟩.dw);
21   if RemoveOp(k+1, r) then // ha már más nem dolgozik a fölső cs.ponton
22         if ⟨k+1|r⟩.saturating then // és már elkezdtük szaturálni
23               NodeSaturated(k+1, r); // akkor most be is fejeződött
24         else
25               QSaturate(k+1, r); // egyébként elindítjuk a szatuációját
26 if q ≠ p then
27   delete(⟨k|p⟩);
```

[A7] A párhuzamos szaturáció Remove függvénye

```
Remove(in: k: szint, p: index)
// Törli a <k,p> csomópontot és a belőle induló felfelé mutató éleket.

1  i: lokális állapot; q: index
2  Lock(FC(k));
3  Update(FC(k), ⟨k|p⟩.Key, ⟨k|p⟩, true); // bejegyezzük az FC-be a törlést
4  Unlock(FC(k));
5  while GetUpArc(k, p, q, i) do
6    if RemoveOp(k+1, q) then // ha már más nem dolgozik a fölső cs.ponton
7          if ⟨k+1|q⟩.saturating then // és már elkezdtük szaturálni
8                NodeSaturated(k+1, q); // akkor most be is fejeződött
           else // egyébként
9                if DWarcs(k+1, q) then // ha van nem ZeroNode-ba mutató
                                        // éle
10                     QSaturate(k+1, q); // akkor elindítjuk a
                                          // szaturációját
                 else                   // különben
11                     Remove(k+1, q); // fölösleges a fölső csomópont is
12 delete(⟨k|p⟩);
```

[A8] A párhuzamos szaturáció Initialize függvényének definíciója

```
Initialize()
Létrehozza a szaturáció elindításához szükséges kiinduló MDD-t.
Szintenként létrehoz egy csomópontot a kiinduló állapot reprezentálására.
Ezeket felfelé mutató élekkel köti össze, illetve a legalsó szintű
csomópont 0. élét a terminális 1 csomópontba köti. Végül a legalsó
csomópontra hívott QSaturate függvénnyel elindítja a szaturációt.
```

[A9] A párhuzamos szaturáció Union függvényének definíciója

```
Union(in: k:szint, p: index, q: index): index
Egy új, <k,s> gyökerű MDD-t hoz létre, ahol <k,s> a <k,p> és <k,q>
csomópontok uniója. A <k,s> csomópontot azonnal el is helyezi a UT(k)-ban.
Visszatérése <k,s>.
```

[A10] A párhuzamos szaturáció DWarcs függvényének definíciója

```
DWarcs(in: k:szint, p: index): bool
Ha ∃i, <k,p>[i] ≠ ZeroNode, akkor visszatérése true, egyébként
visszatérése false.
```

[A11] A párhuzamos szaturáció SetUpArc függvényének definíciója

```
SetUpArc(in: k:szint, p: index, q: index, i: lokális állapot)
Lock(<k,p>.ua). Létrehoz egy új felfelé élet <k, p>-ből <k+1, q>[i]-be,
majd hozzáadja a <k,p>.ua –hez. AddOp(k+1, q); Unlock(<k,p>.ua);
```

[A12] A párhuzamos szaturáció GetUpArc függvényének definíciója

```
GetUpArc(in: k: szint, p: index, out: q: index, i: lokális állapot): bool
Lock(<k,p>.ua). Ha van a <k, p> csomópontból felfelé él, akkor az elsőt
betöltjük a q és i változókba, majd töröljük a <k,p>.ua listából és a
visszatérési érték true lesz. Különben a visszatérési érték false lesz.
Visszatérés előtt Unlock(<k,p>.ua).
```

[A13] A párhuzamos szaturáció AddOp függvényének definíciója

```
AddOp(in: k: szint, p: index)
Lock(<k,p>.ops). Megnöveljük a <k,p>.ops értékét. Unlock(<k,p>.ops).
```

[A14] A párhuzamos szaturáció RemoveOp definíciója

```
RemoveOp(in: k: szint, p: index): bool
Lock(<k,p>.ops). Csökkentjük a <k,p>.ops értéket. Ha a <k,p>.ops új értéke
0, akkor a visszatérési érték false, egyébként a visszatérési érték true.
Visszatérés előtt Unlock(<k,p>.ops).
```

[A15] A párhuzamos szaturáció Find függvényének definíciója

```
Find(in: FC, key, out: v: index, sat: bool): bool
Ha van key kulcsú elem az FC hashtáblában, akkor a tárolt index és bool
érték betöltődik a v és sat kimenő paraméterekbe. Ha volt találat, akkor a
visszatérés true, különben a visszatérés false.
```

[A16] A párhuzamos szaturáció Insert függvényének definíciója

```
Insert(inout: FC, in: key, v: index, sat: bool)
Az FC hashtáblában a key kulccsal elhelyezzüt a v és sat értékeket.
```

[A17] A párhuzamos szaturáció Update függvényének definíciója

```
Update(inout: FC, in: key, v: index, sat: bool)
Az FC hashtáblában a key kulccsal címzett értékekekt frissítjük a v és sat
értékekre.
```

[A18] A párhuzamos szaturáció Locals függvényének definíciója

```
Locals(in: e: esemény, k: szint, p: index): lokális állapotok halmaza
A <k,p> csomópontban az e eseményre lokálisan engedélyezett állapotok
halmazával tér vissza.
```

[A19] A párhuzamos szaturáció Pick függvényének definíciója

```
Pick(inout: ℒ: lokális állapotok halmaza): lokális állapot
Ha ℒ nem üres, kiválaszt egy lokális állapotot belőle. Törli a
kiválasztott állapotot a halmazból, és visszatér vele.
```

[A20] A párhuzamos szaturáció NewNode függvényének definíciója

```
NewNode(in: k: szint): index
A k. szinten létrehoz egy új csomópontot. Az új csomópont összes élét <k-
1, 0>-ra állítja, majd visszatér vele.
```

[A21] A párhuzamos szaturáció CheckIn függvényének definíciója

```
CheckIn(in: k: szint, inout: p: index)
Ha <k,p> minden éle a <k-1, 0>-ba vagy a <k-1, 1>-be mutat, akkor p értéke
<k, 0> illetve <k, 1> lesz, majd a függvény visszatér. Ha van <k,p>-vel
megegyező éllistájú elem az UT(k)-ban, akkor p-t beállítjuk arra, majd a
függvény visszatér. Ha ilyen nincs, akkor <k,p>-t elhelyezzük az UT(k)-
ban, és változatlan p érték mellett a függvény visszatér.
```

[A22] A párhuzamos szaturáció Key függvényének definíciója

```
Key(in: l: szint, q: index, e: esemény): key
Az <l, q> csomópontból és e eseményből képzett kulccsal tér vissza.
```

[A23] A párhuzamos szaturáció QSaturate függvényének definíciója

```
QSaturate(in: k: szint, p: index)
A szaturálandó csomópontok várakozási sorában elhelyezi a <k, p>
csomópontot.
```

[A24] A párhuzamos szaturáció Terminate függvényének definíciója

```
Terminate()
Jelzi a legfölső szintű csomópont szaturációjának elkészültét. Ha a
program jellegéből fakadóan szükséges, akkor leállítja a szálakat.
```

[A25] A párhuzamos szaturáció Confirm függvényének definíciója

```
Confirm(in: k: szint, i: lokális állapot)
Inkrementálisan bővíti az elérhető állapotok halmazát. Az i állapotot
globálisan elérhetőnek jelöli, míg az i állapotból, az egyes események
eltüzelésének hatására elérhető új állapotokkal bővíti a lokális állapotok
halmazát.
```

[A26] A PreFire metódus pszeudokódja

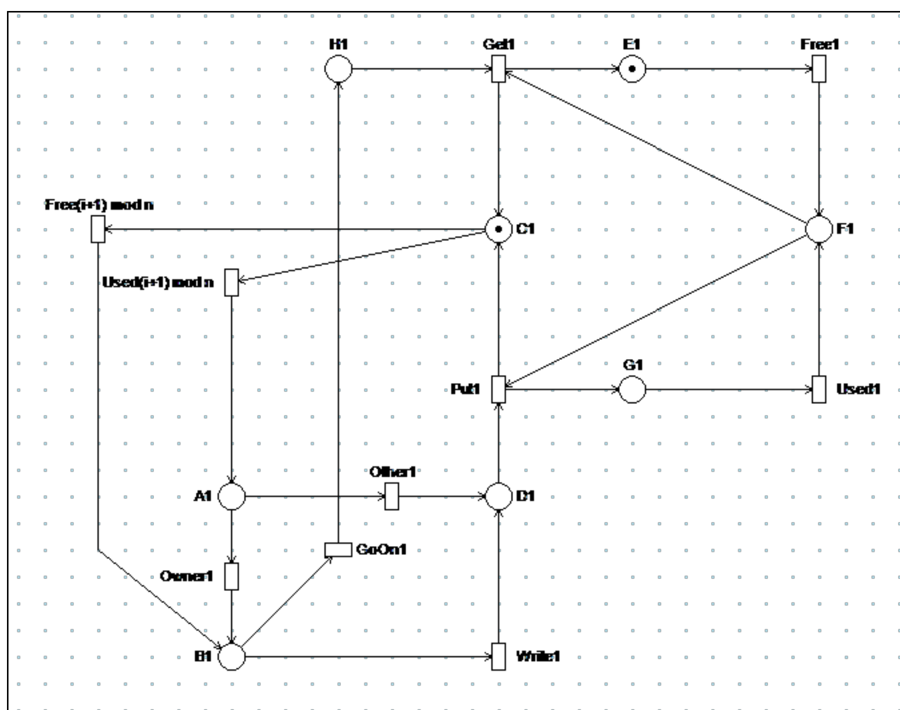
```
PreFire(in: q: csomópont, e: esemény)
Az n csomóponton az e esemény előretüzelését végzi el, az eredményt
berakja a tüzelési gyorsítótárba.

1    ℒ: lokális állapotok halmaza; l: szint; g,j: lokális állapot; f,u,s:index;
     sat:bool
2    l = q csomópont szintje
3    Lock(FC(l));
4    if Find(FC(l), Key(l, q, e), s, sat) then
5          Unlock(FC(l));
6          return;
7    s = NewNode(l);
8    actPreNodeId = s.Id
9    s.preFiring = true
10   s.Key = Key(l, q, e);
11   AddOp(l, s);
12   Insert(Fc(l), s.Key, s, false);
13   Unlock(FC(l));
```

```
14   ℒ= Locals(l, q, e);
15   while ℒ ≠ 0 do
16         g =Pick(ℒ);
17         f = RecFire(e, l-1, <l,q>[g], s,  g);
18         if f ≠ <l-1, 0> then
19               Lock(<l,s>);
20               j = GetTargetState(l, g, e);
21               u = Union(l-1, f, <l,s>[j]);
22               if u ≠ <l,s>[j] then
23                     <l,s>[j] = u;
24                     Unlock(<l,s>);
25                     Confirm(l, j);
26               else
27                     Unlock(<l,s>);
28   if RemoveOp(l, s) then
29         if DWarcs(l, s) then
30               QSaturate(s);
31         else
32               Remove(l, s);
33   return <l, 0>;
```

Függelék B

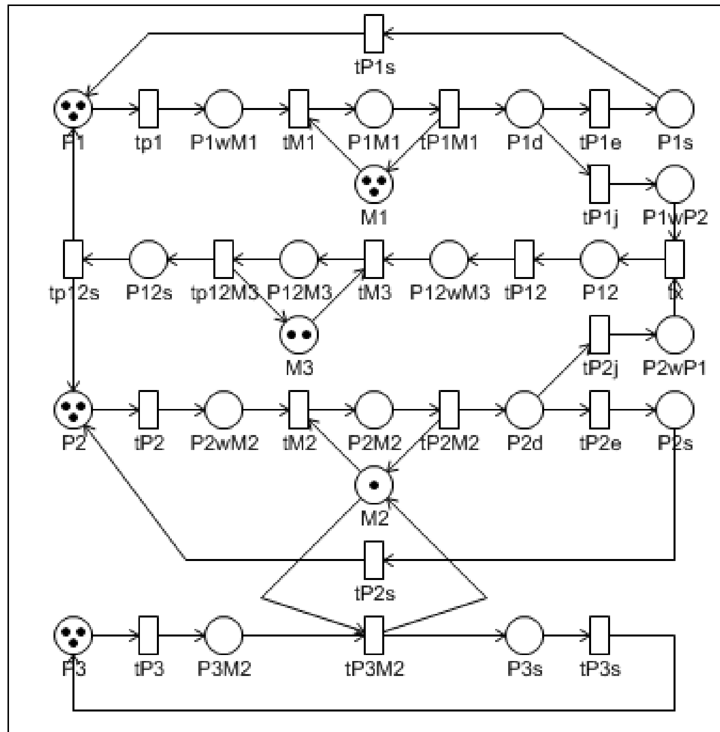
[B1] A Slotted Ring modell Petri hálója

A Petri háló a kommunikáció egy résztvevőjét reprezentálja. A Slotted Ring N jelölés arra utal, hogy a kommunikációnak N darab résztvevője van.



[B2] Az FMS modell Petri hálója

A modell paramétere a p_1, p_2 és p_3 helyeken található tokenek száma, erre utal az FMS N jelölés (jelen esetben 3).



[B3] A Kanban modell Petri hálója

A modell paramétere a p_1, p_2, p_3 és p_4 helyeken levő tokenek száma, erre utal a Kanban N jelölés (jelen esetben 3).