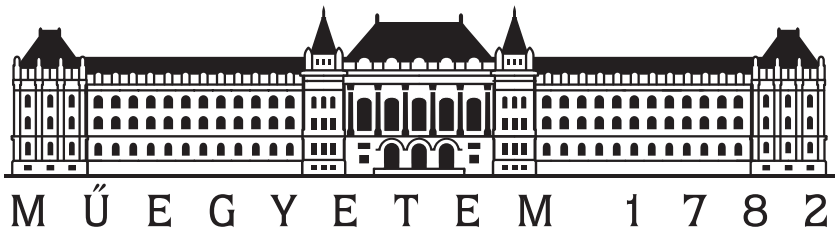
TDK-dolgozat

Darvas Dániel
2010.



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

Szaturáció alapú automatikus modellellenőrző fejlesztése aszinkron rendszerekhez

TDK-dolgozat

Készítette:
Darvas Dániel
III. évfolyam

Konzulensek:
dr. Bartha Tamás,
egyetemi docens

Vörös András,
doktorandusz

2010.

Tartalomjegyzék

1. fejezet

Bevezető

Manapság számos olyan helyen alkalmaznak hardver- és szoftverrendszereket, ahol a hibás működés nem engedhető meg, mivel jelentős anyagi veszteségeket, baleseteket okozhat vagy akár emberéletet is követelhet. Ilyenek például a telekommunikációs hálózatok, a légiforgalmi irányítás, a repülőgépipar, a vasúti biztosítóberendezések, az orvosi eszközök, az erőművek, az űrkutatás és számos egyéb felhasználási terület. A hibás működés gyakran tervezési hibákra vezethető vissza. Ezek kijavítását a tervezőasztalon kell elvégezni, hiszen sok esetben erre később nincs mód vagy óriási költséggel járna.

Komplex rendszereknél a tervezési hibák kiszűrésére alkalmazható főbb módszerek a következők:

- szimuláció,
- tesztelés,
- deduktív verifikáció,
- modellellenőrzés [11].

A szimuláció és a tesztelés a hibák jelentős részét relatíve költséghatékony módon feltárja, viszont az összes hiba kiküszöbölése ezzel a módszerrel szinte lehetetlen [11]. A deduktív verifikáció, azaz az axiómákkal és bizonyításokkal történő helyességbizonyítás hatásos módszer, viszont rendkívül hosszú ideig tart és kiváló szakértőkre van hozzá szükség. A modellellenőrzés a véges állapotú rendszerek helyességbizonyításának egy technikája. Előnye, hogy az előző módszerrel szemben nagyrészt automatikusan végrehajtható, hiszen tipikusan a feltérképezett állapottéren alapul, az állapottér pedig algoritmus segítségével feltérképezhető.

Ahhoz, hogy modellellenőrzés legyen végezhető, szükség van a rendszer modelljére. Ennek leírására számos megoldás ismert, ezek közül az egyik elterjedt modellezési eszköz a Petri-háló.
A modellellenőrzés két fő problémával küzd:

- Egyelőre főként véges állapotterű modellekre alkalmazható, azonban ez a problémák jelentős részénél nem okoz gondot, illetve korlátos modellellenőrzéssel vagy megfelelő absztrakcióval elkerülhető.
- Jelentős erőforrásigénye van, hiszen már egy kis modellnek is óriási állapottere lehet, ezért nem triviális az állapottér felderítésének, tárolásának és a modellellenőrzés megvalósítása.

Munkám során a BME Méréstechnika és Információs Rendszerek Tanszékén fejlesztett *PetriDotNet* keretrendszerhez [23] készítettem egy modellellenőrző modult a friss kutatási

eredményeket felhasználva. Ezen új, ún. szaturációs megközelítéssel igen nagy állapotterű modellek is hatékonyan ellenőrizhetők.

A dolgozatom felépítése a következő: a 2. fejezetben bemutatom a munkámhoz szükséges háttérismereteket és a modellellenőrzés alapjául szolgáló állapottér-generálási algoritmusok fejlődését. Ez után a 3. fejezetben leírom a jelenleg ismert egyik leghatékonyabb algoritmust aszinkron rendszerek modellellenőrzésére. A 4. fejezetben a 3. fejezetben leírt módszeren alapuló megvalósításom részleteit mutatom be. Ezt követően az 5. fejezetben mérésekkel támasztom alá az elért eredményeket, végül a 6. fejezetben összegzem a munkámat.

2. fejezet

Háttérismeretek

E fejezetben bemutatom a vizsgálandó modellek leírására használt Petri-hálók főbb tulajdonságait (2.1), a megvalósításomban használt bináris és többértékű döntési diagramokat (2.3), valamint a modellellenőrzés alapjait (2.4) és az utóbbi időkben ehhez használt főbb megközelítéseket. Emellett kitérek a munkámban felhasznált *PetriDotNet* keretrendszer rövid ismertetésére is (2.2).

2.1. Petri-hálók

A Petri-háló a rendszermodellezés és rendszeranalízis egyik elterjedt modellezési eszköze. Egyidejűleg grafikus és matematikai reprezentációt is definiálnak, így könnyen kezelhetők, jól áttekinthetők és vizsgálhatók. A Petri-hálók fő alkalmazási területe a konkurens aszinkron elosztott rendszerek modellezése [18].

2.1.1. Alapstruktúra

A Petri-háló egy irányított, súlyozott, páros gráf, melyben az egyik pontosztály (P) elemeit *helyeknek* (place), a másik pontosztály (T) elemeit pedig *tranzícióknak* (transition) nevezzük. Grafikusan a tranzíciókat téglalappal, a helyeket körökkel reprezentáljuk. A gráfban egy irányított él egy tranzíciót köt össze egy hellyel vagy egy helyet egy tranzícióval. Az élekhez egy-egy pozitív egész számot rendelünk, ezeket *élsúlyoknak* nevezzük. A Petri-háló állapotát helyeken lévő *tokenek* segítségével fejezzük ki. A helyekhez rendelt tokenszámot a helyeknek megfelelő körökben elhelyezett pontokkal vagy számokkal jelöljük. A háló tokeneloszlása (állapota) egy $M : P \to \mathbb{N}$ függvény, amely minden helyhez egy nemnegatív egész számot rendel. A tokeneloszlás (állapot) kifejezhető egy $\mathbf{m} = \begin{bmatrix} m_1 & \dots & m_\pi \end{bmatrix}^T$ oszlopvektorral is, ahol $m_i = M(p_i)$.

Ezek alapján a Petri-háló formálisan egy olyan $PN = (P, T, E, W, M_0)$ struktúra [17], ahol:

$P = p_1, p_2, \dots, p_\pi$ a helyek véges halmaza,
$T = t_1, t_2, \dots, t_\tau$ a tranzíciók véges halmaza,
$E \subseteq (P \times T) \cup (T \times P)$ az élek halmaza,
$W : E \to \mathbb{Z}^+$ az élekhez súlyokat rendelő súlyfüggvény,
$M_0 : P \to \mathbb{N}$ a kezdeti tokeneloszlás.

A Petri-háló strukturális tulajdonságait a kezdeti tokeneloszlás nem befolyásolja, így a Petri-háló *struktúrája* az alábbi szerint adható meg: $N = (P, T, E, W)$.

A következőkben jelölje $\bullet t$ a $t \in T$ tranzíció bemeneti helyeit, $t\bullet$ pedig a $t \in T$ tranzíció kimeneti helyeit, azaz $\bullet t = \{p|(p, t) \in E\}$ és $t\bullet = \{p|(t, p) \in E\}$.

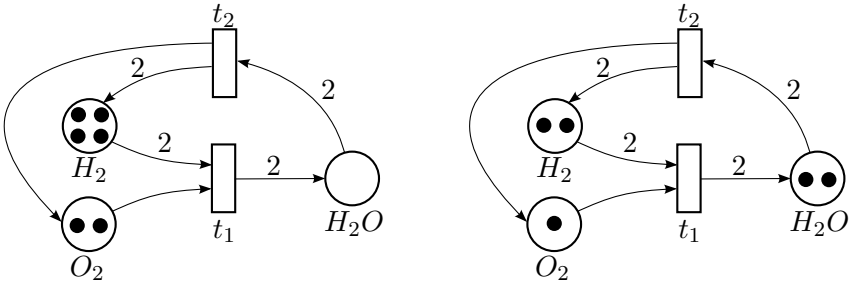
A Petri-háló dinamikus viselkedését az egyes állapotokban értelmezett *tüzelések* határozzák meg. A tüzelések szabályai a következők:

1. Egy $t \in T$ tranzíció *engedélyezett*, ha t-nek minden egyes $p \in \bullet t$ bemenő helyén legalább $w^-(p,t)$ token van, ahol $w^-(p,t) = W(p,t)$, azaz a (p,t) él súlya.

2. Egy engedélyezett tranzíció tetszése szerint tüzelhet vagy nem tüzelhet, tehát működése nemdeterminisztikus.

3. Egy engedélyezett t tranzíció tüzelése $w^-(p,t)$ darab tokent vesz el t minden egyes $p \in \bullet t$ bemenő helyéről és $w^+(p,t)$ tokent helyez el a t tranzíció minden egyes $p \in t\bullet$ kimenő helyére, ahol $w^+(p,t) = W(t,p)$, azaz a t-ből p-be vezető (t,p) él súlya.

1. példa. *A 2.1. ábrán egy Petri-háló látható, mely egy egyszerű kémiai folyamatot modellez. A hálóban két tranzíció (t_1, t_2) és három hely (H_2, O_2, H_2O) található. A 2.1(a) ábrán csak a t_1 tranzíció engedélyezett. A t_1 tranzíció tüzelésével a 2.1(a) ábrán látható állapotból a 2.1(b) állapotba kerül a háló. Ebben az állapotban a t_1 és a t_2 tranzíció egyaránt engedélyezett.*



(a) Kezdő tokeneloszlás (b) Tokeneloszlás t_1 tüzelése után

2.1. ábra. Példa Petri-háló: vízbontás és hidrogén oxidációja

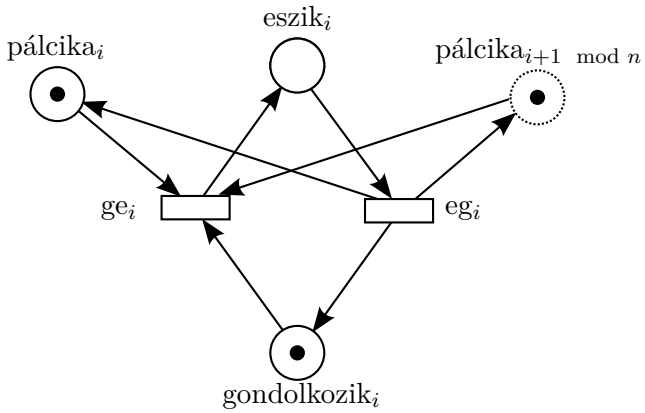
2. példa. *A későbbiekben egy másik modellt is gyakran fogok példaként felhozni: ez az étkező filozófusok modellje. Az étkező filozófusok egy klasszikus szinkronizációs probléma. Lényege az, hogy egy asztal körül n filozófus ül, mindegyikük rizst eszik vagy gondolkozik, egyszerre viszont két egymás mellett ülő filozófus nem ehet [13], mivel két tányér között csak egyetlen pálcika található. A 2.2. ábra az ezt leíró háló i. filozófusra vonatkozó részét mutatja.*

Ez az étkező filozófusok egyszerűsített modellje, mivel az egyes pálcikák felvételének problémáját elhanyagoljuk, azokat a filozófusok külön-külön nem vehetik fel. Az eredeti modellt a C.1. fejezetben ismertetem.

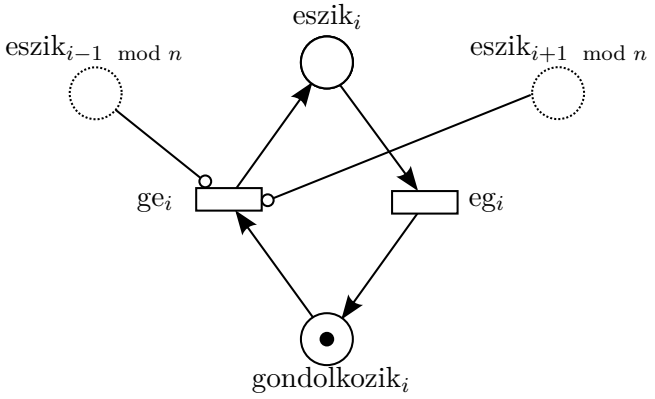
2.1.2. Tiltó élek

A Petri-hálók gyakran használt kiegészítései a *tiltó élek*. A tiltó élekkel kifejezhető, hogy bizonyos feltételek teljesülésekor egy adott működés ne hajtódjon végre. A tiltó élek halmaza az élek egy $I \subseteq (P \times T)$ részhalmaza. A szokványos élekhez hasonlóan a tiltó élekhez is rendelhetők élsúlyok. Jelölje a tiltó élek súlyfüggvényét: $W_I : I \to \mathbb{Z}^+$. Így a tiltó élekkel kiegészített Petri-hálók formálisan egy $PN_I = (PN, I, W_I)$ struktúrával írhatók le.

A tiltó élekkel kiegészített Petri-hálókban a tüzelési szabály megváltozik: egy $t \in T$ tranzíció engedélyezettségéhez az eddigi feltételeken túl az is szükséges, hogy a hozzá tiltó éllel kapcsolódó $p \in P$ helyen a tiltó él súlyánál $(W_I(p,t))$ kevesebb token legyen.



2.2. ábra. Az étkező filozófusok egyszerűsített modelljének i. filozófusra vonatkozó része



2.3. ábra. Az étkező filozófusok egyszerűsített modelljének i. filozófusra vonatkozó része tiltó élekkel

A tiltó élek bevezetésével a Petri-háló kifejezőereje megnő, viszont számos analízis módszer ezekre a hálókra már nem (vagy csak korlátozott mértékben) használható [18].

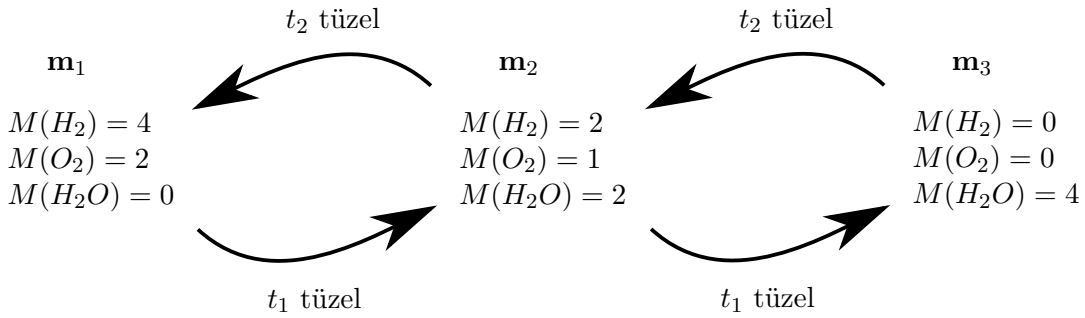
3. példa. *A 2. példában leírt étkező filozófusok probléma megfogalmazható tiltó élek felhasználásával is: az i. filozófus akkor kezdhet el enni, ha a két oldalán ülő filozófusok nem esznek, azaz bármelyik mellette ülő filozófus evő állapota meggátolja az i. filozófust, hogy enni kezdjen.*

E modell i. filozófusra vonatkozó részét mutatja be a 2.3. ábra. Ahogyan az ábrán is látható, a tiltó élek végeire nem nyilat, hanem üres kört rajzolunk.

2.1.3. Állapottér

Gyakran felmerülő kérdés, hogy egy adott $\mathbf{m}$ állapotból egy tüzeléssel milyen állapotokba kerülhet a háló. Ezért bevezetjük az ún. *next-state függvényt* ($\mathcal{N}(\mathbf{m})$), mely megadja az $\mathbf{m}$ állapotból egyetlen tüzeléssel elérhető állapotok halmazát. Jelölje $\mathcal{N}_t(\mathbf{m})$ azt a t tranzícióra szorított next-state függvényt (azaz következő állapotot), mely azt adja meg, hogy $\mathbf{m}$ állapotból t tranzíció egyetlen tüzelésével milyen állapot érhető el. Értelmezzük a $\mathcal{N}$ függvényt állapothalmaz argumentummal is: ha $\mathcal{A}$ az állapotok egy halmaza, $\mathcal{N}(\mathcal{A}) = \bigcup_{\mathbf{a}\in\mathcal{A}} \mathcal{N}(\mathbf{a})$.

Hasonlóan fontos kérdés, hogy egy bizonyos állapotból mi az összes elérhető állapot. Egy $\mathbf{s}$ állapotból az elérhető állapotok halmaza $\{\mathbf{s}\} \cup \mathcal{N}(\mathbf{s}) \cup \mathcal{N}(\mathcal{N}(\mathbf{s})) \cup \cdots = \mathcal{N}^*(\mathbf{s})$, ahol $\mathcal{N}^*(\mathbf{s})$ a tranzitív lezártat jelöli. A Petri-háló kezdőállapotából elérhető állapotok halmazát a háló *állapotterének* nevezzük.



2.4. ábra. A vízbontás modelljének állapottere

Mivel jellemzően a hálók állapottere exponenciálisan nő a háló méretének növekedésével, már relatíve kis modellek esetén is óriásira nő az állapotok száma, ezt a jelenséget állapottér-robbanásnak nevezik. Emiatt az állapottér felderítése nem egyszerű feladat.

4. példa. *A 2.1. ábrán látható Petri-háló állapottere látható a 2.4. ábrán. Látható, hogy a hálónak három lehetséges tokeneloszlása (állapota) lehet:* $\boldsymbol{m}_1$, $\boldsymbol{m}_2$ *és* $\boldsymbol{m}_3$. *Az* $\boldsymbol{m}_1$ *állapotban a* t_1 *tranzíció engedélyezett, ennek tüzelése* $\boldsymbol{m}_2$ *állapotba vezet át. Az* $\boldsymbol{m}_2$ *állapotban mindkét tranzíció engedélyezett, a* t_1 *tranzíció* $\boldsymbol{m}_1$ *állapotba vezet, a* t_2 *pedig* $\boldsymbol{m}_3$*-ba. Az* $\boldsymbol{m}_3$ *állapotban csak* t_2 *engedélyezett, ennek tüzelése* $\boldsymbol{m}_2$*-be vezet.*

2.1.4. Strukturális tulajdonságok

A Petri-hálók bizonyos tulajdonságai függetlenek a tokeneloszlástól, ezek pusztán a háló felépítéséből leszűrhetők. Ezeket *strukturális tulajdonságoknak* nevezzük. Egy Petri-háló strukturális tulajdonságai közül a legfontosabbak az *invariánsok*. Jellegzetes kérdés, hogy mely tüzelési sorozatok azok, melyek eltüzelése a háló tokeneloszlását nem változtatja meg. Az ilyen tüzelési sorozatot *T-invariánsnak* nevezzük.

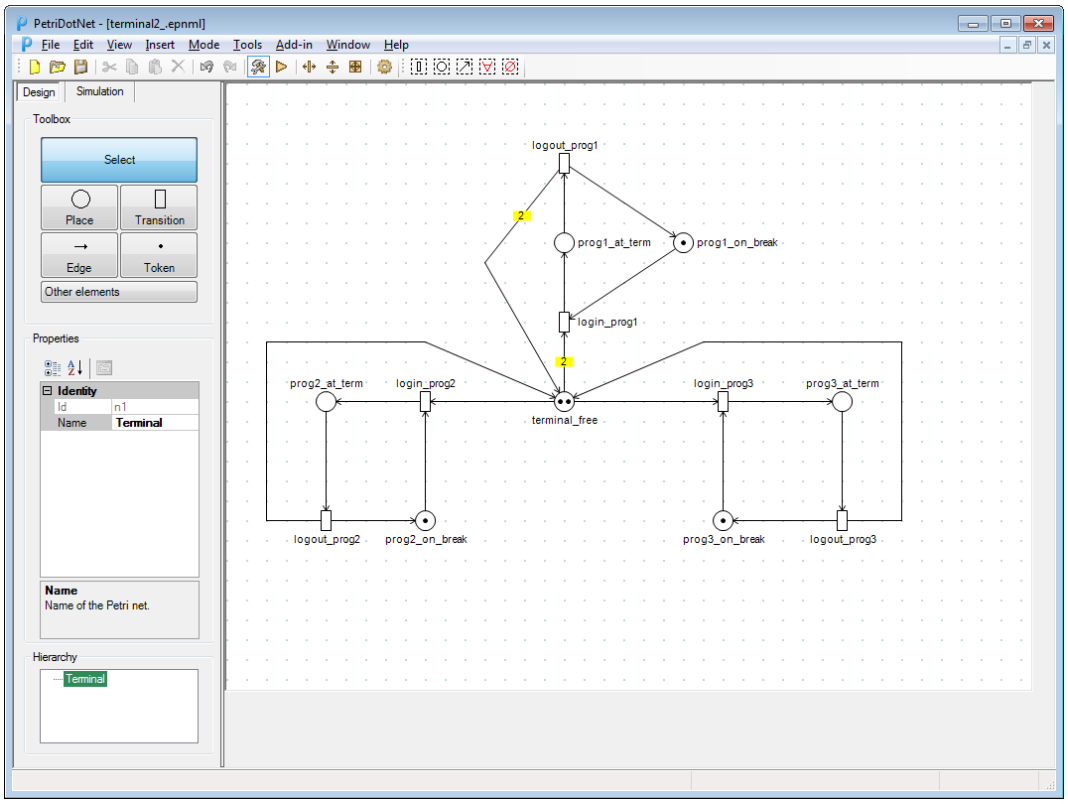
Másik fontos tulajdonsága egy hálónak, hogy a tokenek, melyek gyakran erőforrásokat reprezentálnak, ne fogyjanak és ne szaporodjanak. Azonban egy erőforrás részerőforrásokra bomolhat, így a kívánalom a úgy fogalmazható meg, hogy a tokenek számának a helyek valamely részhalmaza felett képzett súlyozott összege állandó maradjon. Ezt *P-invariánsnak* nevezzük.

5. példa. *A 2.1. ábrán látható Petri-hálóban a* t_1, t_2 *tranzíciók tüzelési szekvenciája T-invariánst alkot, hiszen ezek egymás utáni tüzelése a szekvencia kiinduló tokeneloszlását állítja vissza.*

Ebben a hálóban P-invariáns is található, bár az jól látható, hogy a tokenek összege nem állandó, hiszen az egyik ábrán 6, a másikon 5 token szerepel a helyeken. Az viszont tetszőleges tüzelések után igaz lesz, hogy $2 \cdot M(H_2O) + 2 \cdot M(O_2) + M(H_2) = 8$*, azaz a tokenek egy súlyozott összege állandó marad. A Petri-hálóban más P-invariáns is van,* $M(H_2) + M(H_2O) = 4$*, illetve* $2 \cdot M(O_2) + M(H_2O) = 4$*. Ez összhangban van a valósággal, hiszen a modellezett kémiai folyamatban atomok nem tűnnek el és újak sem keletkeznek.*

2.2. A PetriDotNet keretrendszer

A *PetriDotNet* egy Petri-háló szerkesztő, szimuláló és analizáló eszköz, mely fejlesztése a BME Méréstechnika és Információs Rendszerek Tanszéken folyik 2008 óta. A program képes kezelni az alap Petri-hálókon kívül tiltó élekkel kiegészített vagy kapacitáskorlátokkal



2.5. ábra. A *PetriDotNet* keretrendszer főablaka

ellátott hálókat és időzített, sztochasztikus modelleket is. A keretrendszer .NET alapokon nyugszik, így Windows, Linux és Mac OS alatt is futtatható (.NET vagy Mono framework-kel). Előnye a manapság elérhető többi hasonló alkalmazáshoz képest, hogy bárki könnyen fejleszthet hozzá beépülő modulokat, illetve felhasználóbarát, könnyen használható.

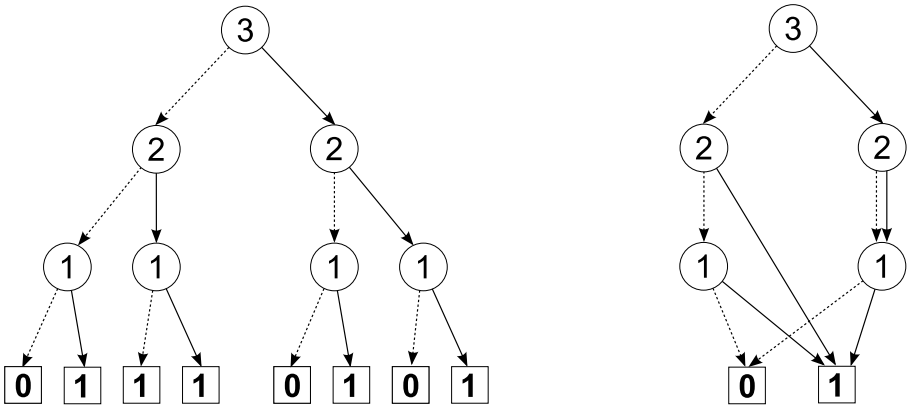
Részletesen a *PetriDotNet* keretrendszerről a keretrendszer honlapján [23] olvashat. Ugyanitt a keretrendszer le is tölthető, kipróbálható.

2.3. Döntési diagramok

A bináris függvények hatékony tárolása nem újkeletű probléma. A kiindulást a döntési fák szolgáltatták, melyek képesek jól reprezentálni egy bináris függvényt, azonban nem kompaktak. Sok különböző optimalizációs módszerrel próbálták csökkenteni a döntési fák méretét. Ezek közül a legsikeresebb, leghatékonyabb változatot a döntési diagramok képviselik.

Bináris döntési diagramok

Bryant 1986-ban írt cikkében [1] hatékony reprezentációt mutatott be az $f(x_n, \ldots, x_1) = b$ bináris függvények ábrázolására (ahol $x_1, \ldots, x_n, b \in \{0, 1\}$). Egy *bináris döntési diagram* (binary decision diagram, BDD) egy irányított aciklikus gráffal ábrázolja a függvényt. A gráf csúcshalmazában (V) két típusú csomópont található: nemterminális és terminális csomópont. Egy $v \in V$ nemterminális csomópontból két él indul ki, mely két gyermekcsomópontra mutat, $low(v) \in V$-re és $high(v) \in V$-re. A kompaktabb formalizmus kedvéért egy alternatív jelölést is bevezetünk: $low(v) = v[0]$ és $high(v) = v[1]$. Ezen kívül tartozik minden nemterminális csomóponthoz egy szint is: $level(v) \in \mathbb{Z}^+$. A szintet terminális



(a) Döntési fa

(b) Bináris döntési diagram

2.6. ábra. Bináris függvények kódolásai

csomópontra is értelmezzük, minden terminális w csomópontra $level(w) = 0$. Egy terminális $w \in V$ csomóponthoz tartozik egy bináris érték: $value(w) \in \{0, 1\}$. A terminális csomópontokra helyenként értékükkel hivatkozunk, valamint logikai kifejezésekben is ezt vesszük figyelembe. Szövegben a $value(w) = 0$ terminális csomópontra terminális nulla, a $value(w) = 1$ terminális csomópontra terminális egy néven hivatkozok.

Egy v gyökerű gráf az alábbi f_v függvényt reprezentálja:

- Ha v terminális csomópont, $f_v = value(v)$.
- Ha v nemterminális csomópont és $level(v) = i$, akkor

$$f_v(x_n, \ldots, x_1) = \overline{x}_i \cdot f_{low(v)}(x_n, \ldots, x_1) + x_i \cdot f_{high(v)}(x_n, \ldots, x_1)$$

Szemléletesen: ha az $f(x_n, \ldots, x_1)$ függvény értékét szeretnénk meghatározni a gráf alapján adott $x_1, \ldots, x_n$ értékek mellett, el kell indulnunk a gráf v gyökeréből (a döntési fák bejárásához hasonlóan). Amennyiben v nemterminális csomópont, $x_{level(v)} = 0$ esetén a $low(v)$ csomópont felé haladunk tovább, különben $high(v)$ felé. Ezt a csomópontot v-nek tekintve folytassuk így tovább, amíg terminális csomóponthoz nem érünk. A függvény értéke az elért terminális csomópont értéke lesz.

A gráf jelölésekor a nemterminális csomópontokat körrel, a terminális csomópontokat négyzettel szokás jelölni. A nemterminális csomópontoknál körökbe a csomópont szintje, a terminális csomópontoknál a négyzetbe a csomópont értéke kerül. A *low*-nak megfelelő éleket szaggatott, a *high*-nak megfelelő éleket folytonos vonalú nyíllal jelölik.

6. példa. *Egy bináris döntési diagram látható a 2.6(b) ábrán. Az általa kódolt függvény az alábbi (x_3, x_2, x_1) értékekre ad igaz (1) eredményt:* $(0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1)$. *Tehát például* $f(1, 0, 1) = 1$, *de* $f(0, 0, 0) = 0$.

Ugyanezt a függvényt írja le a 2.6(a) ábrán látható döntési fa. Jól érzékelhető a két leírás közti eltérés tömörségben. Megjegyzendő, hogy a például szolgáló BDD kompaktabb formában is leírható lenne.

2.3.1. Többértékű döntési diagramok

A *többértékű döntési diagramok* (multi-valued decision diagram, MDD) a BDD-k kiterjesztései. Az MDD-vel kódolt $f(x_n, \ldots, x_1)$ függvényben az egyes x_i-k tetszőleges véges

D_i tartományból kerülhetnek ki, tehát $x_i \in D_i = \{d_{i,0}, d_{i,1}, \ldots, d_{i,|D_i|-1}\}$. Az egyszerűség kedvéért a következőkben úgy tekintjük, hogy $D_i = \{0, 1, \ldots, |D_i| - 1\}$. Mivel az egyes $d_{i,j}$ elemek kölcsönösen egyértelműen megfeleltethetők $j \in \mathbb{N}$ értékeknek, ezért ez az általánosságot nem csökkenti.

Az MDD csúcshalmazában a BDD-hez hasonlóan található nemterminális és terminális csomópont. Ugyanakkor MDD esetén egy $v \in V$ nemterminális csomópontnak $|D_{level(v)}|$ gyermeke van: $child(v, 0), \ldots, child(v, |D_{level(v)}| - 1) \in V$. Viszont a terminális csomópontok értékei MDD-k esetén is kizárólag a $\{0, 1\}$ halmazból kerülhetnek ki. Többi tulajdonsága a BDD-vel azonos. Megjegyzendő, hogy a BDD az MDD speciális esete, ahol $D_i = \{0, 1\}$ minden i esetén.

A könnyebb hivatkozás érdekében tekintsük a következő jelöléseket. Jelölje $v[i]$ a v csomópont i. gyerekét, azaz $v[i] \equiv child(v, i)$. Legyen továbbá $v[i_k, i_{k-1}, \ldots, i_l] \equiv (v[i_k])[i_{k-1}, \ldots, i_l] \equiv v[i_k][i_{k-1}] \ldots [i_l]$.

Az MDD-k ábrázolása a BDD-khez hasonló, de az egyes gyerekcsomópontokba mutató élek sorszámát az élre ráírjuk, nem pedig szaggatott és folytonos vonallal jelöljük a megfelelő értéket. Az átláthatóbb ábra érdekében gyakran csak azokat az éleket és csomópontokat ábrázoljuk, melyek a gyökér és a terminális egy csomópont közti utak valamelyikén található, a terminális nullába vezető utakat nem.

2.3.2. Döntési diagramok típusai

Egy döntési diagram *rendezett* (OMDD), ha minden nemterminális v csomópontjára, minden $d \in D_{level(v)}$-re igaz, hogy $level(v) > level(v[d])$. A továbbiakban kizárólag rendezett döntési diagramokkal foglalkozunk.

Ha egy döntési diagram *kváziredukált* is (QMDD vagy QOMDD), akkor nincsen benne szintugrás, azaz p összes q gyerekére igaz, hogy $level(q) = level(p) - 1$, és nincsenek benne duplikált csomópontok, azaz ha $level(u) = level(v)$, valamint u és v csomópontok minden gyereke páronként megegyezik ($u[0] = v[0], u[1] = v[1], \ldots$), akkor $u = v$.

Ha egy döntési diagram *teljesen redukált* (RMDD vagy ROMDD) [4], akkor rendezett, nincsenek benne olyan p csomópontok, melyekre $p[0] = p[1] = \ldots$, és nincsenek olyan csomópontok, melyek minden gyereke páronként megegyezik.

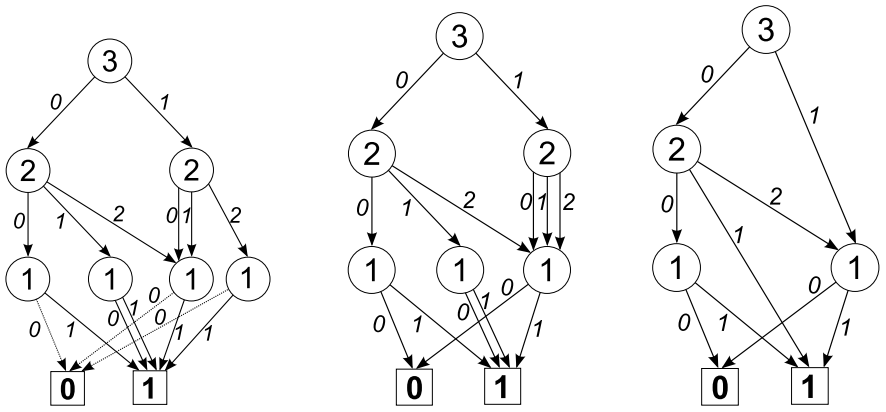
7. példa. *Az MDD-k típusaira láthatók példák a 2.7. ábrán. A 2.7(a) ábrán egy rendezett MDD, a 2.7(b) ábrán egy kváziredukált MDD, a 2.7(c) ábrán pedig egy teljesen redukált MDD látható. Mindhárom MDD azonos függvényt kódol. A függvény az alábbi* (x_3, x_2, x_1) *értékekre ad igaz eredményt:* $\{(0, 1, 0), (0, 1, 1), (0, 0, 1), (1, 0, 1), (1, 1, 1), (1, 2, 1)\}$.

2.3.3. Műveletek csomópontokon

A későbbiekben részletesen tárgyalt megközelítésekben és az én megoldásomban is kváziredukált többértékű döntési diagramok szerepelnek, ezért a műveleteket is QMDD-kre vezetem be az egyszerűség kedvéért.

A QMDD-ken végezhető műveletek gyakorlatilag halmazműveletek, melyeket a QMDD-ben kódolt n-esek halmazán végzünk el. (A QMDD-ben kódolt n-esek alatt olyan $(x_n, x_{n-1}, \ldots, x_1)$ struktúrákat értünk, melyekre $f(x_n, x_{n-1}, \ldots, x_1) = 1$.)
Egy QMDD-ben p és q egy szinten lévő csomópontok *uniója* a következő:

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



(a) Rendezett MDD (b) Kváziredukált MDD (QMDD) (c) Teljesen redukált MDD (RMDD)

2.7. ábra. Többértékű döntési diagramok

Egy QMDD-ben p és q egy szinten lévő csomópontok *metszete* a következő:

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

Két QMDD uniója/metszete alatt a gyökércsomópontjainak unióját/metszetét értjük.

2.4. Modellellenőrzés

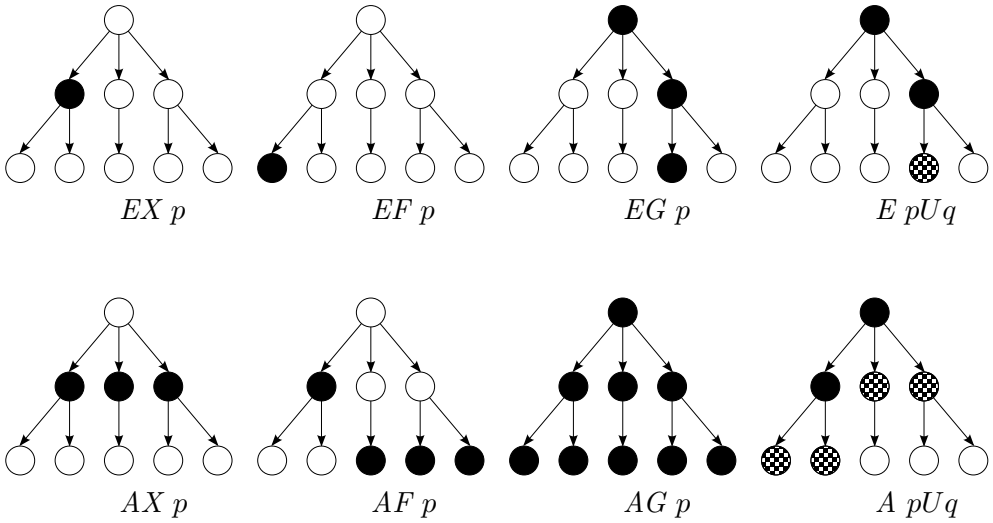
A modellellenőrzés egy olyan verifikációs technika, amely a rendszer egy véges modelljén vizsgálja meg, hogy bizonyos adott tulajdonságok teljesülnek-e [18]. Formálisan: azt vizsgáljuk, hogy egy M modellre egy adott r követelmény igaz-e [3]. Az r követelmény többféle, különböző kifejezőerejű formalizmussal is megadható, melyekkel eltérő típusú kifejezések értékelhetőek ki. A következőkben az elágazó idejű temporális logikával foglalkozom.

A CTL (Computational Tree Logic, elágazó idejű temporális logika) széles körben használt formalizmus egyszerűsége és kifejezőereje miatt. Segítségével kijelentések igazságának logikai időbeli változását vizsgálhatjuk [18]. A kezdőállapotból kiindulva az elágazó idővonalak mentén az egymás utáni állapotokat egy fa-szerű struktúrában ábrázolhatjuk, ahol minden állapotnak legalább egy utódja (rákövetkező állapota) lehet, amennyiben a rendszer nem jut holtpontra. A következőkben bemutatott kifejezéseket ebben a számítási fában fogjuk kiértékelni.

A CTL-kifejezések szintaxisát az alábbi szabályok adják:

- Minden P atomi kijelentés egy állapotkifejezés.
- Ha p és q állapotkifejezések, akkor $\neg p$ és $p \wedge q$ is állapotkifejezés.
- Ha s útvonalkifejezés, akkor $E\, s$ és $A\, s$ állapotkifejezések.
- Ha p és q állapotkifejezések, akkor $X\, p$ és $p\, U\, q$ útvonalkifejezések.

Az egyszerűbb írásmód kedvéért további operátorokat is definiálni szokás, így az utolsó kijelentés az alábbira változik: Ha p és q állapotkifejezések, akkor $F\, p$, $G\, p$, $X\, p$ és $p\, U\, q$ útvonalkifejezések.



2.8. ábra. Minták a CTL operátorokra

A CTL segítségével állapotkifejezéseket vizsgálhatunk. A fenti szabályokból következően az útvonalkifejezések elé mindenképp kerül egy útvonalkvantor (E vagy A), így az állapotoperátorok (F, G, X, U) és az útvonalkvantorok mindenképp párban állnak.
Az útvonalkvantorok szemléletes jelentései a következők:

- A: az adott állapotból kiindulva minden lehetséges útra (for all futures)
- E: az adott állapotból kiindulva legalább egy útra (for some future)

Az állapotoperátorok szemléletes jelentései a következők:

- $F\,p$: az útvonal egy tetszőleges állapotán p igaz (future)
- $G\,p$: az útvonal összes állapotán p igaz (globally)
- $X\,p$: a következő állapotban p igaz (next)
- $p\,U\,q$: az útvonal egy állapotán igaz lesz q, és addig minden állapotban igaz p (p until q)

A lehetséges operátorok *szemléletes jelentése* egy-egy mintán keresztül a 2.8 ábrán látható [11]. Az ábrán az egyes körök állapotokat jelölnek. Egy állapotból a lehetséges következő állapotokba nyilak mutatnak. A sötétre színezett állapotokban p feltétel igaz. A fehér színű állapotokban p igazságtartalma nem lényeges. A pepita színezésű állapotokban q feltétel igaz.

Az itt leírt állapotoperátorokon kívül még két, ritkábban használt operátor is létezik:

- $p\,W\,q$: minden állapotban igaz p vagy addig igaz, amíg q igazzá nem válik (weak until)
- $p\,R\,q$: q egészen addig fennáll, amíg p igaz nem lesz (release)

8. példa. *Tekintsünk néhány példát a CTL-kifejezésekre. Ezeket a 2. példában leírt étkező filozófusok modellre írom fel. Az alábbi kifejezések mindegyike igaz az étkező filozófusok modelljére.*

Informális kifejezés	Formális (CTL) kifejezés
Mindig igaz lesz, hogy az 1. és 2. filozófus egyszerre nem eszik.	$AG[\neg(eszik_1 = 1 \wedge eszik_2 = 1)]$
Létezik olyan tüzelési sorozat, hogy az 1. filozófus enni fog.	$EF[eszik_1 = 1]$
Minden esetben amíg az 1. filozófus nem kezd el enni, addig gondolkozik.	$A[gondolkozik_1 = 1 \, U \, eszik_1 = 1]$

A nyolc lehetséges főbb operátor (EX, EF, EU, EG, AX, AF, AU, AG) egymástól nem független. Például $AGp \equiv \neg EF \neg p$. Az $\{EX, EU, EG\}$ operátorhalmaz elemeivel az összes lehetséges operátor kifejezhető [9].

Bár a korábban leírt formális szintaxisból következik, külön kiemelendő, hogy a CTL-kifejezések egymásba is ágyazhatók.

2.5. Az állapottér-generálás módszerei

Az 1. és 2.4. fejezetben bemutatott modellellenőrzéshez szükség van a modell állapotterének feltérképezésére, amely a modellellenőrzés egyik kritikus pontja. Azért, hogy a munkám motivációja érthető legyen, a következőkben áttekintem az állapottér-generálás főbb módszereit.

2.5.1. Explicit állapottér-generálás

Egy Petri-háló állapotterének generálására a legegyszerűbb módszer a következő: kiindulunk a kezdeti tokeneloszlásból, majd tüzelésekkel újabb állapotokat derítünk fel. Ezt mindaddig folytatjuk, amíg már egyik állapotból sem lehetséges tüzeléssel új állapotba eljutni. Egy ilyen megvalósítást mutat be az 1. algoritmus (*BfSSGen*).

Algoritmus 1 BfSSGen

Input: $\mathbf{m}$: állapot, $\mathcal{N}$: next-state függvény
Output: elérhető állapotok halmaza

```
1. S,U,X : állapothalmaz
2. S = {m}                          // ismert állapotok halmaza
3. U = {m}                          // ismert, felderítetlen állapotok halmaza
4. while U ≠ ∅ do
5.     X = N(U)                     // U-ba csak a tényleges új állapotok kerülnek
6.     U = X\S
7.     S = S ∪ U
8. end while
9. return S
```

Mivel minden egyes állapotot külön-külön megvizsgálunk, ezért a fenti algoritmus futásideje legalább $O(|\mathcal{S}|)$ nagyságrendű, az állapotok száma viszont gyakran a háló méretének exponenciális függvénye. Vizsgáljuk a 2. példában ismertetett étkező filozófusok modellt 100 filozófussal! Ennek állapottere 10^{20} nagyságrendű. Képzeljünk egy olyan processzort, mely 10 GHz-en üzemel és egy órajelütem alatt ki tudja számítani $\mathcal{N}(\mathbf{i})$ értékét. Az algoritmus futásideje ebben az esetben is 10^{10} $s = 316$ év lenne. Eközben ha egy felderített állapotot egy biten el tudnánk tárolni, a teljes állapottér memóriaigénye körülbelül $10^{10} GB$ lenne. Nyilvánvaló, hogy ez a megközelítés csak egészen kis állapotterű modellekre működhet, nagyobb modellekre az állapottér-robbanás ellehetetleníti a vizsgálatot.

Előnye viszont ennek a megoldásnak, hogy lehetőség van a feltérképezett struktúrán viszonylag egyszerűen modellellenőrzés végrehajtására, hiszen explicit módon rendelkezésre állnak az állapotátmenet-információk.

2.5.2. Szimbolikus modellellenőrzés

[3]-ban bemutattak egy újabb megközelítést, immár binárisan kódolja az állapotokat és a tranzíciókat, amelyek tárolását BDD-k segítségével oldották meg. A cikkben megmutatták, hogy legtöbb esetben megfelelő kódolással és kompakt reprezentációval az állapottérgenerálás és a modellellenőrzés felgyorsítható. Esettanulmányukban egy $1,5 \cdot 10^{29}$ állapotterű modellen 22 perc alatt sikerült modellellenőrzést végrehajtaniuk. Ez a megoldás azonban nem működik jól aszinkron rendszerek esetén [5]. Emellett a módszer hátránya, hogy az explicit állapottér-generálással szemben a kapott szimbolikus állapottéren közvetlenül nem végezhető modellellenőrzés.

2.5.3. Láncolás

Egy láncolás alapú (chaining alapú) algoritmust mutat be a 2. algoritmus (*ChainSSGen*). A fő különbség a tisztán szélességi bejárást használó 1. algoritmussal (*BfSSGen*) szemben, hogy a ChainSSGen algoritmus nem követ szigorú szélességi sorrendet, kombinálja a mélységi és szélességi lépéseket. Míg a *BfSSGen* egy iterációs lépése során kizárólag az $\mathcal{U}$ halmazból egyetlen tüzeléssel elérhető állapotok kerülnek felderítésre, addig a ChainSSGen algoritmussal egy iterációs lépés során több állapot is felderíthető. Ha például az α_1 eseménnyel egy $\mathbf{u}_1$ esemény került felderítésre, akkor az ezt követő $\alpha_2, \alpha_3, \ldots$ tranzíciók már ebből az $\mathbf{u}_1$ állapotból is tüzelhetnek [19][8].

Algoritmus 2 ChainSSGen

Input: $\mathbf{m}$: állapot, $\mathcal{N}$: next-state függvény
Output: elérhető állapotok halmaza

```
1.  S,U : állapothalmaz
2.  S = {m}                                   // ismert állapotok halmaza
3.  U = {m}                                   // ismert, felderítetlen állapotok halmaza
4.  while U ≠ ∅ do
5.     for each α eseményre do
6.        U = U ∪ N_α(U)                      // hozzávétel a lehetséges új állapotokhoz
7.     end for
8.     U = U\S                                // a nem valóban új állapotok eltávolítása
9.     S = S ∪ U
10. end while
11. return S
```

2.5.4. Szaturáció

A 2000-es évek elején egy újabb módszert javasoltak a hatékony állapottér-generálásra: az ún. *szaturációs algoritmust* [5][7]. Ennek alapja, hogy MDD-ket használnak BDD-k helyett a lokális állapottér közvetlenebb, hatékonyabb kódolása és tárolása érdekében. Emellett egy új iterációs sorrendet is bemutattak, mely jobban alkalmazkodik a döntési diagramok tulajdonságaihoz és az általuk kódolt részinformációkhoz, így jelentős gyorsulást értek el. A szaturációs algoritmus részletes bemutatása a 3. fejezetben található meg.

Ciardo és társai a szaturációs algoritmus segítségével bizonyos optimális modelleknél sikeresen derítettek fel 10^{6000} méretű állapotteret néhány perc alatt egy közönséges sze-

mélyi számítógéppel [8]. A szaturáció általában jelentősen kisebb idő- és memóriaigényű aszinkron rendszerek esetén, mint a többi algoritmus.

3. fejezet

A szaturációs algoritmusok működése

A következőkben részletesen kifejtem az általam megvalósított szaturációs algoritmus főbb jellegzetességeit. A fejezet első részében leírom, hogy a szaturációs algoritmus segítségével hogyan deríthető fel egy modell állapottere. Bemutatom, hogy az [5][7][8]-ban leírt technikák a modellek milyen típusú dekompozíciójára építenek (3.1.1), hogyan kezelik az eseményeket (3.1.2) és hogyan kódolják újszerűen a next-state függvényt (3.1.3). Bemutatom továbbá egy Petri-háló állapotterének leírását MDD-vel (3.1.4), majd ismertetem a szaturációs algoritmust és ennek tulajdonságait (3.1.5–3.1.7). Ez után a 3.2. fejezetben leírom, hogy a [9]-ben hogyan alkalmazták a szaturációs algoritmust modellellenőrzésre, konkrétan az *EX* és *EU* operátorokra.

3.1. Állapottér-generálás szaturációs algoritmussal

3.1.1. A modell dekomponálása

Az állapottér generálása előtt a Petri-hálót K darab részmodellre bontjuk (K értéke befolyásolja az algoritmus hatékonyságát, erről bővebben később lesz szó). Így az állapotteret kisebb részekre oszthatjuk. Ezek a részek lokálisan is bejárhatók, ezáltal globális lépésekre sokkal ritkábban lesz szükség aszinkron rendszerek megfelelő felbontása esetén. A dekomponálás eredményeképp a háló globális állapota $(i_K, \ldots, i_1)$ lesz, ahol i_j a j. részmodell *lokális* állapota. A j. részmodell összes lehetséges lokális állapotát a $\mathcal{S}_j$ halmaz reprezentálja.

Egy modell ilyen formájú dekompozícióját *Kronecker-konzisztensnek* nevezzük, ha teljesíti az alábbi feltételeket:

- $\mathbf{i} = (i_K, \ldots, i_1)$, azaz a modell globális állapotát meghatározza a részmodellek lokális állapotainak összessége,
- $\mathcal{N}(\mathbf{i}) = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha(\mathbf{i})$, azaz egy $\mathbf{i}$ állapotból elérhető állapotok halmaza az egyes események tüzelésével külön-külön elérhető állapotok halmazának uniója, ahol $\mathcal{E}$ a modell eseményeinek halmaza,
- a modellben minden α eseményre teljesül, hogy $\mathcal{N}_\alpha(\mathbf{i}) = \mathcal{N}_{1,\alpha}(i_1) \times \cdots \times \mathcal{N}_{K,\alpha}(i_K)$, azaz a következő állapotok halmaza formálisan a lokális következő állapotok Descartes-szorzataként adódik.

A közönséges Petri-hálók tetszőleges partícionálása Kronecker-konzisztens felbontáshoz vezet [8].

3.1.2. Események

A globálisan aszinkron – lokálisan szinkron rendszereknél (GALS rendszereknél) a legtöbb esemény (tranzíció) csupán a részmodellek egy halmazát érinti. Ezt ki lehet használni az állapottér-generálás során, elkerülhetők ennek segítségével olyan tranzíciók tüzelési próbálkozásai, melyek a felbontásból következően az adott részmodellt nem befolyásolják. Így a lokalitás kihasználásával az algoritmus hatékonysága javítható.

Ahhoz, hogy ezt vizsgálni tudjuk, bevezetünk néhány jelölést. Jelölje $\mathcal{E}$ az összes eseményt. Minden $\alpha \in \mathcal{E}$ eseményre meghatározható egy $Top(\alpha) = k$ érték, amely azt adja meg, hogy melyik az a legnagyobb k. szint, melynek lokális állapotát az esemény tüzelése befolyásolhatja. Hasonlóképp $Bot(\alpha) = l$ jelöli azt az l. legkisebb szintet, melynek lokális állapotát az esemény tüzelése befolyásolhatja. Nyilvánvaló, hogy minden α eseményre $K \geq Top(\alpha) \geq Bot(\alpha) \geq 1$.

Ezek alapján az eseményeket K halmazba oszthatjuk:

$$\mathcal{E}_k = \{\alpha \in \mathcal{E} | \, Top(\alpha) = k\}, \qquad \forall k \in \{1, \ldots, K\}$$

3.1.3. A next-state függvény Kronecker-kódolása

Az $\mathcal{N}$ next-state függvényt globális tárolás helyett dekomponáljuk események és részmodellek szerint. Így $K \cdot |\mathcal{E}|$ függvényre bontható az eredeti $\mathcal{N}$ függvény. A kapott $\mathcal{N}_{k,\alpha}$ lokális next-state függvények megadják, hogy adott k. szinten lokális állapotból α esemény hatására milyen lokális állapotok érhetők el.

Az állapotátmeneti relációk dekompozíciója nem új ötlet, már korábban is használták [2]. Aszinkron rendszerek esetén azonban különösen hatékony módszernek bizonyult ez a módszer.

Ezek az $\mathcal{N}_{k,\alpha}$ lokális next-state függvények könnyen kódolhatók $\mathbf{N}_{k,\alpha} \in \{0,1\}^{|\mathcal{S}_k| \times |\mathcal{S}_k|}$ mátrixokkal a következő alapján:

$$\mathbf{N}_{k,\alpha}[i,j] = 1 \Leftrightarrow j \in \mathcal{N}_{k,\alpha}(i)$$

A fenti mátrixokat az egyszerűség kedvéért a következőkben *Kronecker-mátrixoknak* nevezem. Petri-hálók esetén ezek a Kronecker-mátrixok kevés 1-es értéket tartalmaznak. Mivel a tranzíciók egyértelműen meghatározzák, hogy mely lokális állapotból mely lokális állapotba vezetnek át, ezért a Kronecker-mátrixokban soronként legfeljebb egyetlen 1-es érték szerepelhet, a többi 0 lesz. [8]-ban megmutatták, hogy a next-state függvényt aszinkron rendszerek esetén hatékonyabb mátrixokkal ábrázolni, mint MDD-vel.

9. példa. *Tekintsük a 2.1. ábrán látható Petri-hálót úgy, hogy a hálót nem dekomponáljuk. Legyen az állapotkódolás a következő:*
$0 : M(H_2) = 4, M(O_2) = 2, M(H_2O) = 0$
$1 : M(H_2) = 2, M(O_2) = 1, M(H_2O) = 2$
$2 : M(H_2) = 0, M(O_2) = 0, M(H_2O) = 4$

Ahogyan az a 2.4. ábrán is látható, a t_1 esemény tüzelésével a 0 állapotból az 1 állapotba kerülhet a háló, az 1 állapotból pedig 2 állapotba. A t_2 esemény tüzelése a hálót 2 állapotból 1 állapotba, 1 állapotból 0 állapotba viszi.

Ez Kronecker-mátrixokkal a 3.1. ábrán látható módon reprezentálható.

Abban az esetben, ha egy α esemény tüzelése egy bizonyos k. szintet nem befolyásol, akkor $\mathbf{N}_{k,\alpha} = \mathbf{I}$ identitásmátrix. Ebben az esetben az α esemény *független* a k. szinttől. Nyilvánvaló, hogy $\mathbf{N}_{k,\alpha} = \mathbf{I}$, ha $k > Top(\alpha)$ vagy $k < Bot(\alpha)$. Az is előfordulhat ugyanakkor, hogy egy $Top(\alpha) > k > Bot(\alpha)$ tulajdonságú k. szintre is igaz lesz ez. Ezek az információk az állapottér generálása előtt is megismerhetők a Petri-háló struktúrájából, tehát a továbbiakban a priori információként tekinthetünk rá.

$$\begin{matrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{matrix}$$

(a) $\mathbf{N}_{1,t_1}$ Kronecker-mátrix

$$\begin{matrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{matrix}$$

(b) $\mathbf{N}_{1,t_2}$ Kronecker-mátrix

3.1. ábra. Next-state függvény reprezentálása Kronecker-mátrixokkal

3.1.4. Állapottér leírása többértékű döntési diagrammal

[5][7][8]-ban a K részre bontott modell állapotterét egy $K+1$ szintű MDD-vel írják le, melyben minden (nemterminális) szinthez a Petri-háló helyeinek egy-egy diszjunkt halmaza tartozik. Ez eltér az addigi megközelítésektől, mivel a korábbi MDD-t használó algoritmusok $2K+1$ szintű MDD-ket használtak, amelyekben az állapotátmeneti relációt is kódolták. Jelen megoldásban azonban az állapotátmenetek kódolása külön, Kronecker-mátrixokban történik, így helyhalmazonként egy szint használata elegendő.

A könnyebb kezelhetőség érdekében a szaturációs algoritmusok kváziredukált többértékű döntési diagramokat (QMDD-ket) használnak.

A lokális állapotokat nemnegatív egész számokkal szintenként folytonosan indexeljük. Ennek megfelelően minden részmodell lokális kezdőállapota 0. Az állapotteret leíró MDD az alábbi szerint alakul:

- Az MDD csomópontjait $K+1$ darab szinthez rendeljük, a szinteket 0-tól K-ig indexeljük.
- A terminális csomópontok a 0. szintre kerülnek. A többi szinten csak nemterminális csomópont lehet.
- Egy j. szinten lévő nemterminális csomópont összes gyermeke a $j-1$. szinten van.
- A K. szinten egyetlen csomópont található, ez a gyökér.
- A csomópontok gyermekeit a lokális állapotokkal indexeljük.

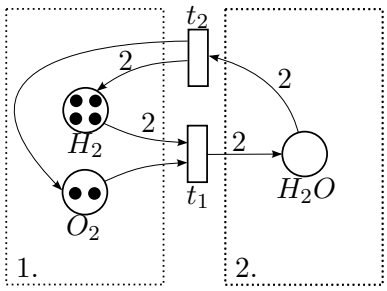
Az egyszerűbb kezelés érdekében minden $k \in \{1, \ldots, K\}$ szinten felveszünk egy-egy *nullcsomópontot*. Ezek összes éle az alsóbb szint nullcsomópontjába mutat. Kivétel ez alól az 1. szint nullcsomópontja, melynek minden éle a terminális nulla csomópontba mutat.

10. példa. *Példaképp a 2.1(a) ábrán látható modell állapotterét mutatom be MDD-vel leírva. Bár a modell mérete nem kívánja meg feltétlenül, a modellt partícionálom: az első részmodellbe a H_2 és O_2 helyek, a második részmodellbe pedig a H_2O hely fog tartozni. Ezt mutatja a 3.2(a) ábra.*

Ez után kódolom az egyes szinteken lehetséges állapotokat a 3.2(b) és a 3.2(c) ábra alapján. A táblázatban i_1 az 1. szint lokális állapotait, i_2 a 2. szint lokális állapotait jelöli. A lokális állapotok sorrendje tetszőleges lehet, mindössze annyi megkötés van, hogy a kezdőállapotnak a 0-s sorszámot kell kapnia.

A modell felbontása és a lokális állapotok számozása után az állapottér MDD-je értelmezhető. A modell lehetséges (i_1, i_2) globális állapotai a következő halmazból kerülhetnek ki: $(0,0), (1,2), (2,1)$. *Jól látható, hogy a 3.2(d) ábrán látható MDD által kódolt függvény is pontosan ezen értékekre fog 1 értéket adni.*

Természetesen a 3.2(d) ábrán látható MDD ugyanazon állapotokat kódolja, mint amelyek a 2.4 ábrán láthatók, csak más kódolás alapján.



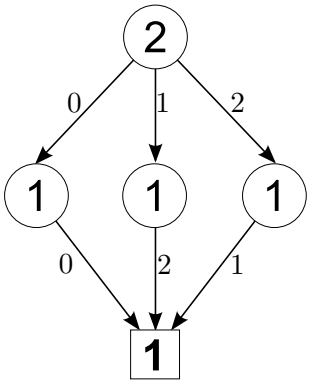
(a) Partícionált modell

i_1	$M(H_2)$	$M(O_2)$
0	4	2
1	2	1
2	0	0

(b) Állapotok kódolása az 1. részmodellben

i_2	$M(H_2O)$
0	0
1	4
2	2

(c) Állapotok kódolása a 2. részmodellben



(d) Az állapottér MDD-vel reprezentálva

3.2. ábra. Állapottér kódolása MDD-vel

3.1.5. A szaturációs algoritmus működése

Az előző munkákhoz képest [5][7]-ben a már felderített állapotteret leíró csomópontok bejárására egy speciális iterációs stratégiát használnak, melyet *szaturációnak* neveznek. Ennek az iterációs stratégiának a lényege, hogy mélységi bejárást hajtunk végre az MDD-n. A bejárás során az egyes eseményeket lokalitásukat kihasználva, megfelelő sorrendben és kimerítően tüzeljük el. Ezt a szaturációs stratégiát kiegészítve a hatékony állapottér-reprezentációval gyors algoritmust kapunk.

Szaturálás során egy i. szinten lévő csomópont esetén az összes olyan α eseményt eltüzeljük, melyre $Top(\alpha) = k$. Egy csomópont *szaturált*, ha az általa reprezentált állapotból tüzeléssel újabb állapotok már nem elérhetők. A csomópontok szaturációja mélységi bejárás szerint történik, azaz egy csomópont feldolgozására akkor kerül sor, ha az összes gyermeke már szaturált [5].

Szemléletesen az algoritmus az alábbi módon épül fel:

1. Kezdőállapotot kódoló K szintű MDD felépítése.

2. Minden 1. szinten lévő csomópont szaturálása (összes olyan α esemény eltüzelése a szint csomópontjaira, melyekre $Top(\alpha) = 1$).

3. Minden 2. szinten lévő csomópont szaturálása (összes olyan α esemény eltüzelése a szint csomópontjaira, melyekre $Top(\alpha) = 2$).
   Ha ez új csomópontot hozott létre az 1. szinten, a létrehozáskor azonnal szaturálni kell ezeket.

4. Minden 3. szinten lévő csomópont szaturálása (összes olyan α esemény eltüzelése a szint csomópontjaira, melyekre $Top(\alpha) = 3$).
   Ha ez új csomópontot hozott létre az 1. vagy 2. szinten, a létrehozáskor azonnal szaturálni kell ezeket.

5. ...

6. Minden K. szinten lévő csomópont szaturálása (összes olyan α esemény eltüzelése a szint csomópontjaira, melyekre $Top(\alpha) = K$).
   Ha ez új csomópontot hozott létre az 1., 2., ..., $K-1$. szinten, a létrehozáskor azonnal szaturálni kell ezeket.

7. Ha az utolsó csomópont szaturációja is elkészül, az algoritmusnak vége, az eredmény az állapotteret leíró MDD.

3.1.6. A szaturáció tulajdonságai

A szaturációs algoritmus számos előnyös tulajdonsággal rendelkezik:

- kihasználja az események lokalitását,
- rekurzívan épülő részleges rendezés jellegű cache-elést alkalmaz,
- az előzőekből adódóan kis tárigénye van akár hatalmas modellek esetén is.

Ezeket a tulajdonságokat fejtem ki bővebben a következőkben.

Lokalitás kihasználása Annak érdekében, hogy a felesleges munkát csökkentsük, egy α eseménynél a tüzelést nem az MDD gyökerénél kezdjük, hanem a $Top(\alpha)$ sorszámú szinten, hiszen ismert, hogy α esemény nem befolyásolja a K és $Top(\alpha)+1$ közti szinteket. Hasonló okokból a tüzelést nem folytatjuk lefelé az 1. szintig, csak a $Bot(\alpha)$ sorszámú szintig. Tehát a tüzelést nem a teljes, addig felderített állapottérre hajtjuk végre, hanem ennél egy optimális esetben sokkal kisebb rész-állapotteret reprezentáló MDD-részen.

Ha egy p csomópont szaturált, többé nem tüzelünk semmilyen α eseményt p csomópontra, melyre $Top(\alpha) = level(p)$.

Ha egy k. szinten lévő v csomópontot szaturálunk egy α eseménnyel, mely tüzelésével i_k lokális állapotból j_k állapotba vezet, akkor $v[j_k]$ csomópont $Fire(\alpha, v[i_j]) \cup v[j_k]$-ra változik, ahol $Fire(\alpha, v[i_j])$ a $v[i_j]$-ben kódolt állapotokból α tüzelésével elérhető állapotokat jelöli. Amennyiben az aktuális α esemény viszont csak a k. szintet befolyásolja (azaz $Top(\alpha) = Bot(\alpha) = k$), akkor biztos, hogy α lokális tüzelése alsóbb szinten új állapotot nem hoz be, így $v[j_k]$-t $v[i_k] \cup v[j_k]$-ra elegendő frissíteni. Ez további futásidő-megtakarítást jelent.

Csomópontok tárolása és cache-elés A csomópontok tárolása szintenként történik, egyedi tárolókban (unique table), mivel kváziredukált MDD-ket használ a szaturációs algoritmus. Azaz ezekben a tárolókban nem lehet két olyan csomópont, melyek összes gyermeke megegyezik. E feltétel biztosításáról a *CheckIn* függvény gondoskodik, amikor egy újabb csomópontot helyezünk el a tárolóban (lásd a 3.1.7. fejezetben). Fontos megjegyezni, hogy az egyedi tárolókba kizárólag szaturált csomópontok kerülhetnek, illetve a tárolóba bekerült csomópontokból kiinduló élek nem módosulhatnak.

Annak érdekében, hogy elkerüljük a felesleges, redundáns számításokat, cache-elést is alkalmazunk. A cache-ekben a tüzelések és az unióképzések eredményei találhatók.

Az állapottérben a hasonló részállapotok bejárását kerüli el a *tüzelési cache*. Ebben (α, p, s) hármasok találhatók, melynek jelentése: a p csomóponton (mely állapotok egy

halmazát reprezentálja) α eltüzelése s csomópontot eredményezte. A tüzelési cache-be kizárólag szaturált s csomópontok kerülhetnek.

Az uniócache annak elkerülésére szolgál, hogy egy p, q csomópontpárra a $p \cup q$ értékét többször ki kelljen számolni. A cache-ben (p, q, u) hármasok találhatók, ahol $u = p \cup q$. Az uniócache-be szintén csak szaturált csomópontok kerülhetnek, hiszen ezek a továbbiakban nem fognak megváltozni, így uniójuk változatlan marad. Fontos azt is figyelembe venni az implementációnál, hogy $p \cup q = q \cup p$.

Mindkét ismertetett cache jelentősen, több nagyságrenddel gyorsítja a végrehajtást.

3.1.7. A szaturációs állapottér-generálás algoritmusa

Az állapottér-generálás során adottnak vesszük a modell partícionálását, illetve feltételezzük, hogy ismertek az egyes α események $Top(\alpha)$ és $Bot(\alpha)$ értékei és az is, hogy mely k szintekre és α eseményekre lesz $\mathbf{N}_{k,\alpha}$ identitásmátrix. Ezek oka az, hogy ezen információk az állapottér-generálás előtt a Peri-háló struktúrájából kiolvashatóak.

Az alább leírt algoritmusok pszeudókódjai megtalálhatók a B. függelékben.

Az állapottér-generálást a ***GenerateStateSpace*** függvény végzi (12. algoritmus). Ez az 1. szinttől a K. szintig felfelé haladva létrehozza a kezdőállapotot kódoló *Nr* csomópontokat, majd szaturálja is ezeket. A szaturáció elvégzése után a *CheckIn* függvénnyel a csomópontok tárolójában elhelyezi a szaturált *Nr* csomópontot.

A ***Saturate*** függvény (13. algoritmus) végzi egy paraméterül kapott *Np* csomópont szaturálását, azaz eltüzel rá minden $\alpha \in \mathcal{E}_k$ eseményt a *SatFire* függvény segítségével. Ezt mindaddig végzi, amíg sikerül újabb állapotokat felderíteni. Ezzel legenerálja a csomópont és gyerekei által kódolt állapotokból elérhető összes állapotot.

A ***SatFire*** függvény (14. algoritmus) feladata egy paraméterül kapott *Np* csomóponton egy szintén paraméterül kapott α esemény eltüzelése. Ennek érdekében egy $\mathcal{L}$ halmazba kigyűjti azon lokális állapotokat, melyekbe az állapottér aktuális helyzete alapján el lehet jutni és ahol az α esemény tüzelése engedélyezett. Minden egyes $i \in \mathcal{L}$ értékkel tüzelést hajt végre $Np[i]$ csomóponton α eseménnyel. Ennek eredménye egy *Nf* csomópont, mely kódolja az aktuális $level(Np) = k$ szint alatti szinteken a frissített állapottér megfelelő részét kódoló MDD gyökerét. Amennyiben ez az *Nf* csomópont legalább egy állapotot kódol, ennek képezzük az unióját az összes olyan $Np[j]$ csomóponttal, ahol α tüzelés hatására i lokális állapotból j lokális állapot elérhető. (Petri-hálók esetén legfeljebb egy ilyen j lokális állapot lehetséges.) Amennyiben az unió eredménye nem a régi $Np[j]$ csomópont, a változás tényét feljegyezzük. Ez után, amennyiben a j lokális állapotból α tüzelésével új k. szintű lokális állapot elérhető, erre az új j állapotra is végrehajtjuk a tüzelést a chaining eljáráshoz hasonló módon. A függvény visszatérési értéke egy logikai érték, mely azt jelzi, a tüzelés okozott-e módosulást az MDD-ben.

A ***SatRecFire*** egy rekurzív függvény (15. algoritmus), mely egy *Np* csomópont által kódolt állapothalmazon végrehajt egy tüzelést a kapott α eseménnyel. Ennek során létrehoz egy új *Ns* csomópontot, majd minden i lokális állapot esetén, melyekben az α esemény engedélyezett, meghívja a *SatRecFire* függvényt az *Ns* csomópontra. Így rekurzívan az *Ns* csomópontra és az alatta lévő összes csomópontra meghívja a tüzelést, ahol az változást okozhat. A SatFire függvényhez hasonlóan amennyiben alsóbb szintről visszatérő *Nf* csomópont legalább egy állapotot kódol, képezi az unióját a jelenlegi értékkel. Amennyiben változás történt, az *Ns* csomópontot a *Saturate* függvény segítségével szaturálja, majd ezzel visszatér. Fontos megjegyezni, hogy mivel az *Ns* csomópont módosul, a szaturált *Np* csomópontot nem módosítja a függvény. A függvény végére *Ns* fogja tárolni azokat az állapotokat, melyek újonnan kerültek felderítésre.

Az algoritmus lényegét ezek a függvények adják. A következőkben ismertetésre kerülnek a további függvények, melyek részt vesznek az algoritmusban.

A ***Confirm*** függvény egy paraméterül kapott k szinten az i lokális eseményt hozzáveszi a globálisan elérhető állapotok ($\mathcal{S}_k$) közé, majd ezt felhasználva új állapotokat keres. Az i állapotból α esemény egyetlen tüzelésével elérhető j állapotokra beállítja $\mathbf{N}_{k,\alpha}[i,j]$ értékét 1-re. Amennyiben ez az állapot eddig felfedezetlen volt, természetesen a Kronecker-mátrix kitöltése előtt kódot is rendel hozzá. Erre a függvényre azért van szükség, mert a priori nem ismertek a lokális állapotok, így futás közben kell azokat felderíteni.

A ***Union*** függvény visszaadja egy p és egy q csomópont unióját. A 2.3 fejezetben leírtakhoz képest több esetet különböztetünk meg benne, ezzel az algoritmus jelentősen gyorsul.

A ***CheckIn*** függvény biztosítja, hogy az állapotteret kódoló MDD-be egy szintre két egyforma csomópont ne kerülhessen. Amennyiben paraméterül egy olyan p csomópontot kap, mely nem szerepel még a csomópontok tárolójában, elhelyezi ott és visszatér p-vel. Amennyiben p megegyezik egy, már a tárolóban szereplő r csomóponttal, visszatér r-rel.

3.2. Modellellenőrzés szaturációs algoritmussal

[9]-ben megmutatták, hogy a többértékű döntési diagramok, a szaturációs technikák, a Kronecker-mátrixos kódolás használata a CTL modellellenőrzés során is jelentős gyorsulást és memóriaigény-csökkenést okoznak.

A következőkben a 2.4 fejezetben bevezetett EX, EU és EG CTL-operátorok megvalósítását írom le. Ezek közül az EU operátor megvalósítása a leghangsúlyosabb.

3.2.1. Strukturális modellellenőrzés megközelítése

A strukturális modellellenőrzés alapműködése az, hogy vesszük azt az állapothalmazt, amelyre a kívánt feltétel teljesül, majd az operátornak megfelelően a logikai időben visszafelé lépkedünk, azaz a tranzíciók inverzeit tüzeljük el. A megfelelő teljes állapottér felderítése után azt vizsgáljuk, hogy szerepel-e benne a háló kezdőállapota. A következőkben leírt algoritmusok azt a állapothalmazt fogják visszaadni, amelyekből kiindulva a feltétel teljesül.

Ennek megfelelően a Kronecker-mátrixok transzponáltjait és a next-state függvények inverzét fogjuk használni. A könnyebb érthetőség végett ezeket újra definiálom. Ha egy t tranzíció tüzelésével egy $\mathbf{a}$ állapotból egy $\mathbf{b}$ állapotba vezet, akkor $\mathcal{N}_t(\mathbf{a}) = \mathbf{b}$ és $\mathcal{N}_t^{-1}(\mathbf{b}) = \mathbf{a}$. Hasonlóan $\mathbf{N}_{k,t}[i,j] = 1 \Leftrightarrow \mathbf{N}_{k,t}^T[j,i] = 1 \Leftrightarrow j \in \mathcal{N}_{k,t}(i)$.

Feltételezzük, hogy a modellellenőrzés előtt állapottér-generálás is történt (mint később látható lesz, erre szükség is van). Ennek megfelelően ismertnek tekintjük az állapotteret és a Kronecker-mátrixok tartalmát.

A későbbiek tárgyalásához bevezetésre kerül néhány jelölés. Jelölje $\mathcal{N}_{\mathcal{A}}(\mathcal{X})$ azon állapotok halmazát, amelyek az $\mathcal{X}$-beli állapotokból $\mathcal{A}$-beli eseményekkel elérhetők, azaz $\mathcal{N}_{\mathcal{A}}(\mathcal{X}) = \bigcup_{\alpha\in\mathcal{A}} \mathcal{N}_\alpha(\mathcal{X})$.

3.2.2. *EX* operátor algoritmusa

Formálisan: egy $\mathbf{i}$ állapot pontosan akkor elégíti ki EXp-t, ha létezik olyan $\mathbf{j} \in \mathcal{N}(\mathbf{i})$, mely kielégíti p-t.

Jelen megközelítésben a p feltételnek megfelelő állapotokat kapja az algoritmus paraméterül egy $\mathcal{P}$ MDD-vel leírva, azaz $\mathcal{P}$-ben az állapottérnek azon része található meg, melyekre p feltétel teljesül. Ennek megfelelően azok az $\mathbf{x}$ állapotok fogják kielégíteni a feltételt, melyekre $\mathbf{x} \in \mathcal{N}^{-1}(\mathcal{P})$. Az EX operátor megvalósításához nincs szükség szaturációra, csak az állapottér-generálás során feltöltött Kronecker-mátrixokat kell felhasználni.

3.2.3. *EU* operátor algoritmusa

Tradicionális megoldás

A 2.4. fejezetben ismertetett $E[p\,U\,q]$ kifejezés értéke igaz, ha létezik legalább egy olyan útvonal, melyen q feltétel valamikor igazzá válik és a p feltétel egészen addig igaz, míg q igazzá nem válik. Ennek megfelelően ki kell indulni $\mathcal{Q}$-ból (mely azokat az állapotokat tartalmazza, ahol q igaz) és visszafelé kell lépkedni olyan állapotokon keresztül, melyekre p igaz. Gyakorlatilag egy $\mathcal{Q}$-ból kiindulva felderítjük visszafelé az állapottér azon részét, melyre p teljesül (azaz $\mathcal{P}$-ben benne van).

Ebből a tradicionális *EU* algoritmus (*EUtrad*, 3. algoritmus) könnyen felírható. Az algoritmus működését mutatja be a 11. példa.

Algoritmus 3 EUtrad

Input: $\mathcal{P}$, $\mathcal{Q}$: állapothalmaz
Output: $\mathcal{X}$: állapothalmaz
1. $\mathcal{X} = \mathcal{Q}$ // $\mathcal{X}$ inicializálása $\mathcal{Q}$-val
2. **repeat**
3. $\quad\mathcal{Y} = \mathcal{X}$
4. $\quad\mathcal{X} = \mathcal{X} \cup (\mathcal{N}^{-1}(\mathcal{X}) \cap \mathcal{P})$ // $\mathcal{X}$-hez olyan előző állapotok hozzávétele, melyekre p igaz
5. **until** $\mathcal{X} = \mathcal{Y}$
6. **return** $\mathcal{X}$

11. példa. *A 3.3. ábra egy modell állapotterét ábrázolja. A korábbi jelölésekhez hasonlóan a fekete csomópontok jelölik azon állapotokat, melyekre p kifejezés igaz, a pepita csomópontokban q kifejezés igaz, a fehér csomópontok által reprezentált állapotokra pedig sem p, sem q kifejezés nem igaz.*

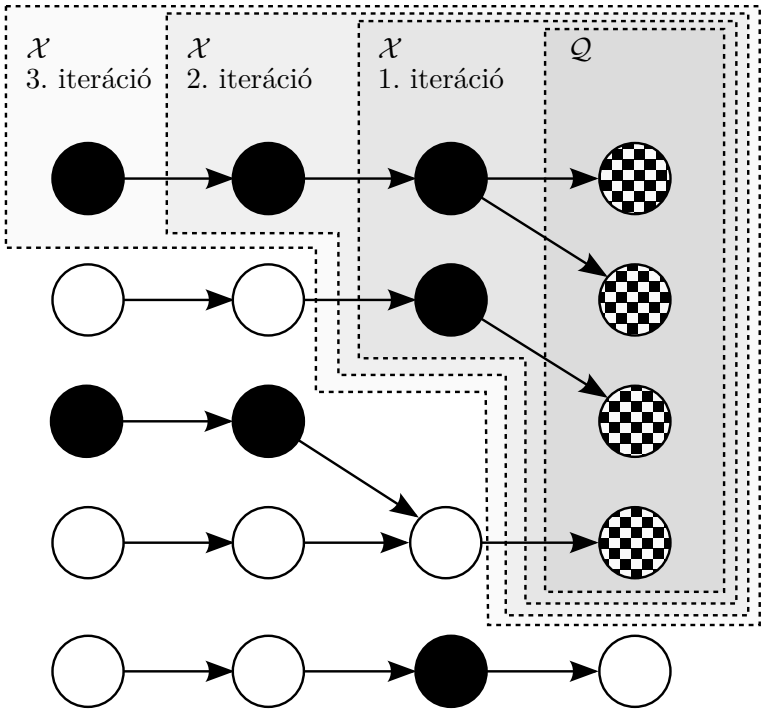
Az $E[p\,U\,q]$ kiszámítása a tradicionális EU algoritmussal a következőképp alakul:

1. *Az algoritmus az összes olyan csomóponttal indul, melyekre q igaz, ez lesz $\mathcal{Q} = \mathcal{X}$ állapothalmaz.*
2. *Az $\mathcal{X}$ halmazhoz hozzávesszük az összes olyan állapotot, melyekre p igaz és amelyekből egy tüzeléssel $\mathcal{X}$ egyik állapota elérhető.*
3. *A 2. lépést ismételjük addig, amíg $\mathcal{X}$ elemszáma növekszik.*

Szaturáció alkalmazása *EU* operátorra

Az EU operátor szaturációs megvalósításának fő kihívása az, hogy az állapottér-generálásnál ismertetett metódushoz képest ki kell szűrni azokat az állapotokat, amelyekre p nem teljesül, azaz amelyek nincsenek benne $\mathcal{P}$-ben (vagy $\mathcal{P} \cup \mathcal{Q}$-ban, a két állítás a temporális logikai kifejezés szempontjából egyenértékű [9]). Fontos, hogy egyetlen olyan állapot se kerülhessen az eredménybe, melyre p akár csak átmenetileg nem teljesül, mielőtt q teljesülne.

Ha viszont minden egyes tüzelés után a kapott állapotokat elmetszenénk $\mathcal{P}$-vel, bár helyes eredményhez vezetne, óriási terhelést okozna, mivel a tüzelésekhez képest a metszet igen költséges művelet. Emiatt [9]-ben részleges szaturációt használnak: az eseményeknek csak azon részhalmazára hajtanak végre szaturációt, amelyeknél nincs szükség a $\mathcal{P}$-vel metszésre. Ezek olyan események, melyek garantáltan megtartják p teljesülését. A többi eseményre a tradicionális megoldást használják. Mivel itt a szaturációt egy korlátozott eseményhalmazra kell elvégezni, valamint az összes szinten szaturálni kell, ezért az



3.3. ábra. Példa a tradicionális EU algoritmus működésére

állapottér-generálásnál bemutatott speciális *Saturate* függvény itt nem használható. Erre az általánosabb, visszafelé szaturáló függvényre *GeneralBackSaturate* néven hivatkozom. Mivel a [9] cikkben az algoritmus pszeudókódja nem szerepel, erre csak a saját megvalósításomnál térek ki.

A szaturáció alkalmazhatóságához az algoritmus elején csoportokba kell sorolni az $\mathcal{E}$ eseményeket. Három kategóriát definiáltak:

1. $\mathcal{X}$-re nézve *halott események*: olyan események, melyek semmilyen $\mathcal{X}$-beli állapotból nem vezetnek $\mathcal{X}$-beli állapotba,

2. $\mathcal{X}$-re nézve *biztonságos (safe) események*: olyan események, melyek nem halottak és semelyik $\mathcal{X}$-en kívüli állapotból nem vezetnek $\mathcal{X}$-beli állapotba, tehát az inverzük nem vezet ki $\mathcal{X}$-ből,

3. $\mathcal{X}$-re nézve *nem biztonságos (unsafe) események*: olyan események, melyek nem halottak és nem biztonságosak.

Ezt a besorolást a *ClassifyEvents* (4. algoritmus) végzi el. A fenti definíciókban szereplő $\mathcal{X}$ halmaznak $\mathcal{P} \cup \mathcal{Q}$-t választjuk. Valójában tetszőlegesen választható itt $\mathcal{P}$ vagy $\mathcal{P} \cup \mathcal{Q}$ a korábban leírtak alapján.

A halott események az inverz állapottér-generálás során nem játszanak szerepet, hiszen a $\mathcal{P} \cup \mathcal{Q}$ állapothalmazból egyik inverz tüzeléssel sem szabad kilépnünk, a halott események pedig $\mathcal{P} \cup \mathcal{Q}$ eseményből sohasem engedélyezettek.

A biztonságos események egyetlen tüzeléssel sem vezetnek ki $\mathcal{P} \cup \mathcal{Q}$-ból, így tüzelésük után nem szükséges a kapott állapottéren költséges metszet műveletet végrehajtani, emiatt ezen eseményekre használhatók a szaturációs algoritmusok. A biztonságos események halmazát $\mathcal{E}_S$ jelöli.

A nem biztonságos események kivezethetnek a $\mathcal{P} \cup \mathcal{Q}$ állapotok köréből, így minden tüzelés után az esetlegesen belekerülő, nem $\mathcal{P} \cup \mathcal{Q}$-beli állapotokat ki kell metszeni. A nem biztonságos események halmazát $\mathcal{E}_U$ jelöli.

Az itt leírtakból gyakorlatilag következik az algoritmus és az ezt megvalósító *EUsat* függvény (5. algoritmus).

Algoritmus 4 ClassifyEvents

Input: $\mathcal{X}$: állapothalmaz
Output: $\mathcal{E}_U$: eseményhalmaz, $\mathcal{E}_S$: eseményhalmaz
1. $\mathcal{E}_S = \mathcal{E}_U = \emptyset$
2. **for each** α in $\mathcal{E}$ **do**
3. **if** $\mathcal{N}_\alpha^{-1}(\mathcal{X}) \neq \emptyset \wedge \mathcal{N}_\alpha^{-1}(\mathcal{X}) \subseteq \mathcal{X}$ **then**
4. $\mathcal{E}_S = \mathcal{E}_S \cup \{\alpha\}$ // $\mathcal{X}$-re nézve biztonságos esemény
5. **else if** $\mathcal{N}_\alpha^{-1}(\mathcal{X}) \cap \mathcal{X} \neq \emptyset$ **then**
6. $\mathcal{E}_U = \mathcal{E}_U \cup \{\alpha\}$ // $\mathcal{X}$-re nézve nem biztonságos esemény
7. **end if** // különben $\mathcal{X}$-re nézve halott esemény
8. **end for**

Algoritmus 5 EUsat

Input: $\mathcal{P}$, $\mathcal{Q}$: állapothalmaz
Output: $\mathcal{X}$: állapothalmaz
1. ClassifyEvents($\mathcal{P} \cup \mathcal{Q}$, out $\mathcal{E}_U$, out $\mathcal{E}_S$)
2. $\mathcal{X} = \mathcal{Q}$ // $\mathcal{X}$ inicializálása $\mathcal{Q}$-val
3. GeneralBackSaturate($\mathcal{X}$, $\mathcal{E}_S$)
4. **repeat**
5. $\mathcal{Y} = \mathcal{X}$
6. $\mathcal{X} = \mathcal{X} \cup (\mathcal{N}_U^{-1}(\mathcal{X}) \cap (\mathcal{P} \cup \mathcal{Q}))$ // $\mathcal{X}$-hez olyan előző állapotok hozzávétele, melyek $\mathcal{E}_U$-beli eseményekkel érhetők el és melyekre $p \vee q$ igaz
7. **if** $\mathcal{X} \neq \mathcal{Y}$ **then**
8. GeneralBackSaturate($\mathcal{X}$, $\mathcal{E}_S$)
9. **end if**
10. **until** $\mathcal{X} = \mathcal{Y}$
11. **return** $\mathcal{X}$

3.2.4. *EG* operátor algoritmusa

Az *EG* p kifejezés igaz, ha létezik legalább egy olyan útvonal, melyen p végig igaz. Ez alapján az algoritmusnak $\mathcal{P}$-ből kiindulva visszafelé kell lépkednie olyan állapotok felé, amelyekre p szintén igaz. Másképpen fogalmazva $\mathcal{P}$-ből ki kell törölni azokat az állapotokat, melyeknek csak nem $\mathcal{P}$-beli megelőző állapota van.

Ez alapján a tradicionális *EGtrad* algoritmus felírható (6. algoritmus).

Algoritmus 6 EGtrad

Input: $\mathcal{P}$: állapothalmaz
Output: $\mathcal{X}$: állapothalmaz
1. $\mathcal{X} = \mathcal{P}$
2. **repeat**
3. $\mathcal{Y} = \mathcal{X}$
4. $\mathcal{X} = \mathcal{N}^{-1}(\mathcal{X}) \cap \mathcal{P}$
5. **until** $\mathcal{X} = \mathcal{Y}$
6. **return** $\mathcal{X}$

Látható, hogy ebben az algoritmusban is a költséges metszés minden iterációban megjelenik. Sajnos azonban az *EU* operátornál ismertetett módszer ennek elkerülésére itt nem alkalmazható jól [9]. Az EG operátorra [9]-ben adnak egy szaturációs megoldást az *EUsat* és *ESsat* algoritmusokon alapulva (az *ESsat* az *EUsat* algoritmus nemtranszponált

Kronecker-mátrixokkal). Azonban mint azt írják, ez az algoritmus az *EGtrad*-hoz képest csak speciális esetekben hoz gyorsulást, más esetekben viszont sokkal rosszabb eredményt ér el, mint az *EGtrad*. A mérési eredményeikben nem is szerepeltetik az *EGsat* algoritmust. Ezek miatt az *EGsat* algoritmus ismertetésétől és implementálásától is eltekintettem, a programomban a tradicionális változat került megvalósításra.

Kifejezések egymásba ágyazhatósága

A fent említett három algoritmus mindegyike előállítja az összes olyan kezdőállapotot kódoló MDD-t, melyekből kiindulva a CTL-kifejezés igaz lesz. Amennyiben ebbe a halmazba bekerül a tényleges kezdőállapot, a kifejezést le lehetne állítani, hiszen már biztosan igaz lesz az eredménye. Azonban ha így tennénk, elveszítenénk a kifejezések egymásba ágyazhatóságának lehetőségét. A jelenlegi megoldásban minden kiértékelés előállít egyben egy állapotkifejezést is. Részben ennek köszönhetően az általam elkészített program az egymásba ágyazott kifejezéseket is képes kiértékelni.

4. fejezet

A szaturációs algoritmusok implementációja

A következőkben bemutatom az implementáció legfontosabb elemeit, melyek jelentősen befolyásolták az elkészült program eredményeit. Röviden leírom azon részek megvalósítását is, melyekről az [5][7][9][8] cikkekben nem szóltak. A fejezet végén kifejtem az egyes heurisztikák mögött álló ötleteket és ezek implementálását.

A következő fejezetekben tárgyalt megoldások könnyebb megértését segíti a 4.1. ábra, mely áttekintést nyújt az architektúráról.

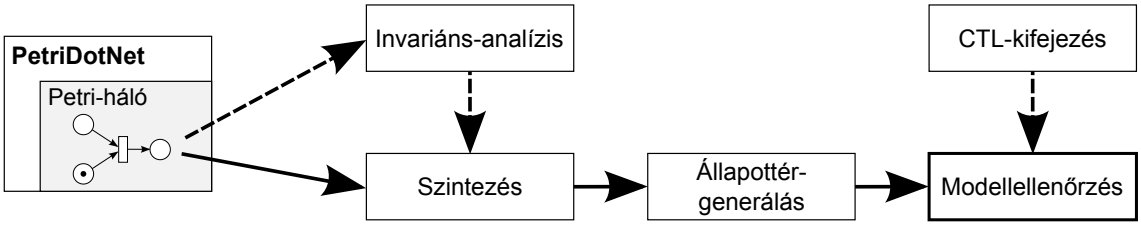
4.1. Döntési diagramok megvalósítása

Az állapottér 3.1.4 fejezetben leírt leképezéséhez szükséges a többértékű döntési diagramok kezelése. Ehhez implementáltam egy döntési diagram csomagot, mely hatékony tárolást és műveletvégzést tesz lehetővé. Erre azért volt szükség, mert jelenleg nincs olyan elérhető MDD csomag, mely megbízható működésű, kellő tudású és a megfelelő környezetbe illeszthető [15][14]. Helyhiány miatt ezzel részletesen nem foglalkozom, a következőkben mindössze egy rövid áttekintést adok a használt struktúrákról és azok céljairól. A felépítés részleteiről a *PetriDotNet* keretrendszer honlapján [23] olvashat bővebben.

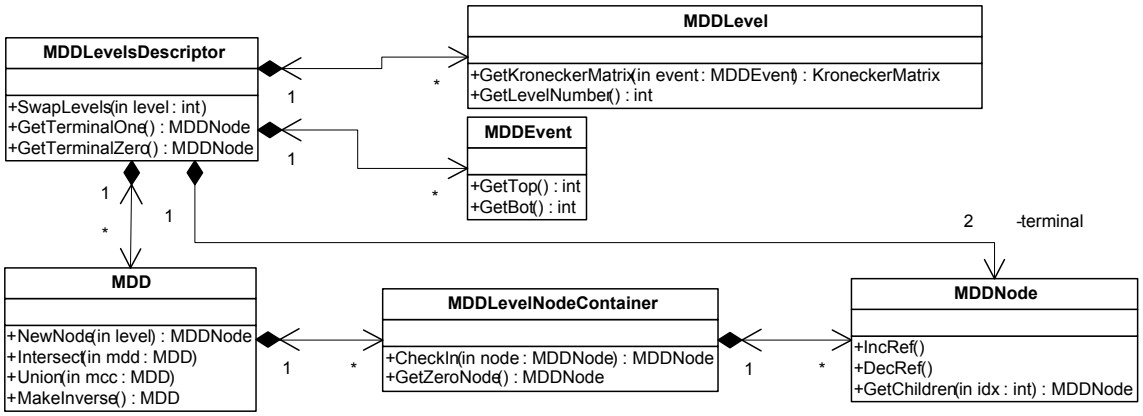
Az eligazodást segíti a 4.2 ábra, mely a fontos osztályok kapcsolatát és az egyes osztályok legfontosabb metódusait ismerteti a teljesség igénye nélkül.

Az osztályok az *MDD* osztály köré szervezhetők. Ez az osztály reprezentál egy kvázi-redukált többértékű döntési diagramot. A diagramban szereplő csomópontok a hatékony használat érdekében szintenként vannak tárolva, egy-egy ilyen szint tárolója az *MDDLevelNodeContainer* osztály példánya. Ez tartalmazza az egyes csomópontokat leíró *MDDNode* objektumokat a 4.2.2. fejezetben ismertetett módon.

Az azonos Petri-hálókhoz azonos szintezéssel kapcsolódó *MDD*-ket fogja össze egy *MDDLevelsDescriptor* objektum, mely tárolja a szintezést (azaz hogy melyik hálóbeli he-



4.1. ábra. A program egyszerűsített szerkezete



4.2. ábra. A döntési diagramok egyszerűsített szerkezete

lyek tartoznak egy-egy szinthez) és az eseményeket, továbbá a terminális csomópontokat. Minden eseményt az *MDDEvent* osztály segítségével írunk le, melyből lekérdezhető az eseményobjektum által leírt tranzíció és az, hogy az adott eseményre *Top* és *Bot* értéke mennyi. Az egyes MDD szinteket az *MDDLevel* osztály írja le, mely tárolja, hogy az adott szint melyik helyek által alkotott részmodellt jelképezi, illetve itt kerül eltárolásra az adott szint Kronecker-mátrixa és az egyes szimbolikus állapotok Petri-hálóbeli megfeleltetése.

A [9]-ben leírtakkal szemben ebben a megvalósításban az egyes MDD csomópontok egymásra referenciákkal hivatkoznak, míg náluk a kapcsolat alapja a csomópontok indexe. A csomópontok indexelése és az ilyen módú hivatkozás egyszerűbb implementációt tesz lehetővé, viszont a futást lassíthatja.

4.2. Állapottér-generálás megvalósítása

Egy modell állapotterének generálása szaturációs algoritmussal komplex feladat, sok lépésből tevődik össze. Ennek ismertetése a 3.1. fejezetben olvasható. A következőkben röviden leírom az algoritmusok megvalósításához szükséges fontosabb implementációs fejlesztéseket.

4.2.1. Szinthozzárendelés

Ahhoz, hogy a a [7][8]-ban leírt algoritmusok használhatók legyenek, először a Petri-hálót dekomponálni szükséges, azaz meg kell határozni, hogy mely Petri-hálóbeli helyek mely MDD szintekhez tartoznak. Egy hely pontosan egy szinthez tartozhat.

Ez a szinthozzárendelés kulcsfontosságú az algoritmus sebessége szempontjából. Amennyiben egy szinthez túl kevés hely tartozik, akkor az MDD nagyon sok szintből fog állni, így a rekurzív hívások nagyon mélyek lehetnek. Ha viszont egy szinten sok hely van és így ahhoz a szinthez túl sok lokális állapot fog tartozni, a szaturálás nagyon memória- és időigényessé válik. Amennyiben az összes helyet egy szinthez rendeljük, visszakapjuk az explicit állapottér-generálást.

Az sem mindegy, hogy az egyes szintek milyen sorrendben követik egymást. Amennyiben egy α eseményre $Top(\alpha)$ nagy, $Bot(\alpha)$ kicsi és ezek közt sok szintet α nem befolyásol, akkor nagyon sok felesleges *SatRecFire* hívás lesz a futás során.

A szinthozzárendelésre az elkészített beépülő modul az alábbi lehetőségeket biztosítja:

- Manuális szinthozzárendelés. A felhasználó minden egyes szintre meghatározhatja, hogy mely helyeket kívánja hozzárendelni. Ennek használatához a program és a

szaturációs algoritmusok működésével tisztában kell lenni, így a felhasználók számára kevésbé használható.

- P-invariáns alapú heurisztika. A program igyekszik olyan szinthozzárendelést készíteni, hogy lehetőleg egy P-invariáns egy szinthez tartozzon. Ezzel részletesen a 4.5.1 fejezet foglalkozik.

- Összetett heurisztika. Ezzel részletesen a 4.5.2 fejezet foglalkozik.

A szinthozzárendelések után a programom meghatározza az egyes α eseményekre a $Top(\alpha)$ és $Bot(\alpha)$ értékeket, majd ezek alapján az eseményeket $\mathcal{E}_k$ diszjunkt halmazokba osztja úgy, hogy az $\mathcal{E}_k$ halmazba azon α események kerüljenek, melyekre $Top(\alpha) = k$.

4.2.2. Hashelés, csomópontok tárolása

A csomópontok tárolóiban gyakran kell vizsgálni, hogy egy adott elemet tartalmaz-e. Emiatt a tárolókat célszerű hashelt halmazként implementálni. Ehhez szükséges egy megfelelő hash-függvényt alkalmazni. Jelen megoldás alapvetően egy csomópont hash-értékét a gyermekeinek egyedi azonosítóiból számítja [21]. A nullcsomópontok egyedi azonosítóját itt nullának tekintem.

Fontos azonban figyelembe venni, hogy az algoritmus elején még nem ismertek az állapotteret leíró MDD-re az egyes D_i értékek, tehát a futás során változhat, hogy egy csomópontnak hány gyermekcsomópontja van. Új állapot felderítésekor elvileg az adott szinten minden csomópontba egy-egy új lefelé mutató él kerül, de mivel ezek felderítetlen állapotot reprezentálnak, mind az alsóbb szint nullcsomópontjába mutatnak. Ez egy szaturált csomóponttal is megtörténhet, melyek már szerepelhetnek az egyes szintek tárolóiban. Ennek azonban nem szabad a csomópont hash-értékét befolyásolnia, másképp fogalmazva két csomópont azonos, ha kizárólag olyan élekben térnek el, melyek csak az egyik csomópontban léteznek és ezek az élek nullcsomópontokba mutatnak. A hash-értékek számításának alapötlete a [21]-ben került leírásra, de ehhez képest a hash-érték generálásakor először megkeressük a legnagyobb sorszámú élet, mely nem nullcsomópontba mutat, majd ettől a gyermektől a nullás sorszámú gyermekig haladva generáljuk a hash-értéket.

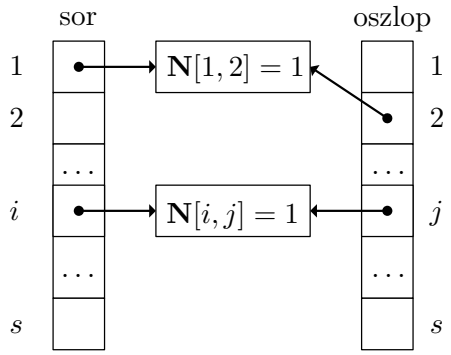
4.2.3. Kronecker-mátrix reprezentációi

A 3.1.3 fejezetben részletesen ismertetett, a next-state függvényt szintenként és eseményenként kódoló Kronecker-mátrixok megvalósítása számos módon lehetséges. A korábban leírtak alapján Petri-hálók esetén a Kronecker-mátrixok minden sorában és oszlopában összesen 1-1 nemnulla (azaz egyes) elem található. A k. szint Kronecker-mátrixának $|S_k| = s$ sora és oszlopa van (ahol a korábbiak alapján $|S_k|$ a k. szinten érvényes állapotok számát jelöli). A mátrix elemeinek tárolására több lehetőség is adódok az alábbiak szerint:

- Tárolható a teljes $s \times s$-es mátrix. Ennek memóriaigénye $O(s^2)$, viszont $O(1)$ időben kereshető benne egy adott elem, $O(s)$ időben pedig egy sor vagy oszlop összes eleme megkapható.

- Tárolhatók egy listában azok a sor- és oszlopindex párok, melyek megadják, hogy a mátrixban hol találhatók egyesek. Mivel legfeljebb s bejegyzés lehet, így a memóriaigény $O(s)$, viszont $O(s)$ idő szükséges egy adott elem megkereséséhez is.

- Tárolhatók azonban csak az 1-eseknek megfelelő elemeket is, melyekre sorok és oszlopok szerint is hivatkozhatunk. Amennyiben egy sorban vagy oszlopban több egyes



4.3. ábra. Kronecker-mátrix implementációja

is található, akkor láncolt listát is alkalmazni kell. Így a memóriaigény $O(s)$, viszont $O(1)$ idő alatt megkereshető benne egy adott elem még akkor is, ha csak egyik indexével adott.

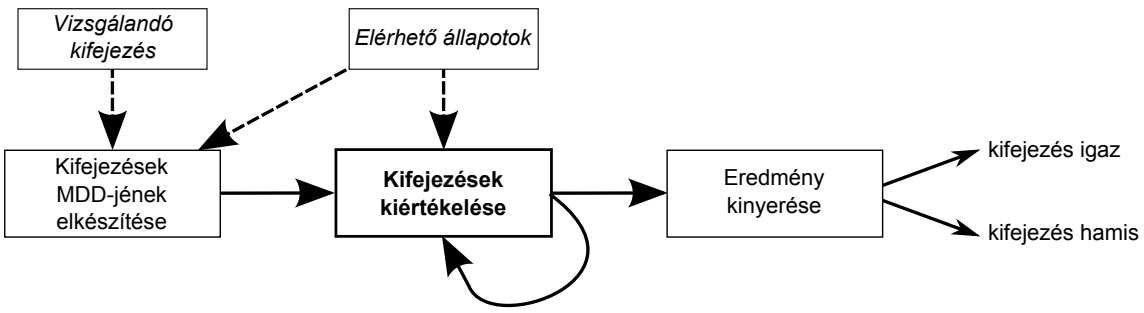
Megjegyzendő, hogy bár a két utóbbi megoldás egyaránt $O(s)$ memóriaigényű, az utóbbi nagyobb helyet igényel. Ennek ellenére végül az utolsó megoldást választottam, mivel a program összes memóriaigényéhez képest elhanyagolható e téren a növekedés, de bizonyos nagy méretű modelleknél a számítási idő szignifikánsan lecsökken. Ezt a megoldást mutatja be a 4.3 ábra.

A megvalósítás során kihasználtam, hogy Petri-hálók analízisét végzi a program, így a Kronecker-mátrixok egy sorban vagy egy oszlopban pontosan nulla vagy egy darab nemnulla elem található csak. Így az [5]-ben bemutatott pszeudókódban bizonyos ciklusokat elágazásra tudtam egyszerűsíteni, hiszen adott i mellett $\mathbf{N}_{k,\alpha}[i,j] = 1$ legfeljebb egy j-re teljesülhet.

4.3. Modellellenőrzés megvalósítása

Ebben a részben sorra veszem azokat a fejlesztéseket, melyek a modellellenőrző algoritmusok implementálásához szükségesek voltak.

A modellellenőrzés egyszerűsített folyamatát mutatja be sematikusan a 4.4. ábra. A folyamat első lépése, hogy a felhasználó által megadott CTL-kifejezés alapján előállítjuk a feltételeket leíró MDD-ket, majd ezek alapján kiértékeljük a kifejezéseket, tehát meghatározzuk, hogy mely kezdő tokeneloszlásokra fognak teljesülni. Amennyiben egymásba ágyazott kifejezések vannak, ezt többször is el kell végezni. Végül meg kell vizsgálni, hogy a tényleges kezdeti tokeneloszlásra a feltétel teljesül-e, így meghatározható, hogy a kifejezés értéke igaz vagy hamis.



4.4. ábra. A modellellenőrzés egyszerűsített folyamata

4.3.1. CTL atomi kifejezés MDD-jének elkészítése

A [9]-ben adottnak veszik az egyes modellellenőrző algoritmusokban az MDD-vel leírt bemenetet. A felhasználótól azonban nem várható el, hogy a kiértékelendő kifejezést MDD-kkel írja le, ehelyett az atomi kifejezéseket helyek tokenszámára teheti meg a program használója. Ezekből a tokenszámokra felírt egyenlőségekből (vagy egyenlőtlenségekből) kell felírni az algoritmusok bemenetéül szolgáló, feltételeket leíró MDD-ket.

Ehhez először készítek egy *maszk-MDD*-t. Szintjeinek száma megegyezik az állapottér szintjeinek számával és minden nemterminális szinten a nullcsomóponton kívül egyetlen csomópont található. Amennyiben egy k szinten v csomópont az egyetlen nem nullcsomópont és $k-1$ szinten w az ugyanilyen tulajdonságú csomópont, valamint P jelöli a Petri-hálóbeli helyet, melyre a kifejezést felírták, akkor:

- minden v-ből kiinduló él w-be mutat, ha P nem a k. szinthez tartozik
- $v[i]$ a w csomópontba mutat, ha P a k. szinthez tartozik és az i állapotra teljesül az atomi kifejezés
- $v[i]$ a $k-1$. szinten lévő nullcsomópontba mutat, ha P a k. szinthez tartozik és az i állapotra *nem* teljesül az atomi kifejezés

Ez a maszk-MDD már MDD formájában tárolja az atomi kifejezés általi megszorítást. Ebben még szerepelhetnek olyan állapotok, melyek valójában nem elérhetők, így ennek képezzük metszetét az előzőekben kiszámított állapottér-MDD-vel. Így már megkapjuk azt a $\mathcal{P}$ MDD-t, mely az adott hálóra vonatkoztatva kódolja az atomi kifejezést.

4.3.2. Invertálás megvalósítása

A CTL-kifejezésekben az atomi kifejezések negáltjai is szerepelhetnek, ezért a p kifejezést kódoló MDD-ből szükséges volt a $\neg p$ kifejezést kódoló MDD előállítása. Ez egy olyan művelet, mely után az új MDD által kódolt f' függvényre igaz lesz, hogy $f'(x_K, \ldots, x_1) = 1 \Leftrightarrow f(x_K, \ldots, x_1) = 0$ minden lehetséges $(x_K, \ldots, x_1)$-re.

Ennek megvalósításához elvileg elegendő a terminális egy és terminális nulla csomópontokat felcserélni. Konkrét megvalósítás esetén azonban arra is ügyelni kell, hogy minden szinten megmaradjanak a nullcsomópontok, illetve hogy az eddigi nullcsomópontok ezen tulajdonságukat elvesztik. Mivel a csomópontok és így az egyedi azonosítóik változnak, ezért természetesen a hash-értéküket is változtatni kell.

4.3.3. Inverz next-state függvény implementálása

A [9]-ben leírt modellellenőrző algoritmusokhoz szükség van az $\mathcal{N}_{\mathcal{A}}^{-1}(\mathcal{X})$ inverz next-state függvény (azaz previous-state vagy előző állapot függvény, ahol $\mathcal{A}$ események egy halmaza) megvalósítására. Az állapottér-generálás során a *SatRecFire* függvény pontosan ilyen működést végzett előre-irányban. Néhány módosítást azonban végre kellett hajtanom rajta, hogy ezt a feladatot is ellássa:

- A Kronecker-mátrixok transzponáltjaival kell számolni benne, hiszen ezek fogják a tüzelések inverz hatásait leírni.
- Mivel csak egy lépést tesz visszafelé, ezért nincs szükség benne szaturációra, azaz a *Saturate* hívásra.
- Az eredeti *SatRecFire* függvénnyel szemben az újonnan létrehozott csomópontokat egy új MDD-be kell helyezni.

- Minden egyes $\alpha \in \mathcal{A}$ eseményre meg kell hívni a módosított *SatRecFire* függvényt.

A létrehozott csomópontok új MDD-je fogja kódolni $\mathcal{N}_{\mathcal{A}}^{-1}(\mathcal{X})$-et, a futás pedig nem fogja $\mathcal{X}$-et befolyásolni. Ez viszont pazarló lehet, mivel új MDD-t kell létrehozni, illetve új csomópontok jönnek akkor is létre, ha a csomópont által leírt állapot mind $\mathcal{X}$-ben, mind $\mathcal{N}_{\mathcal{A}}^{-1}(\mathcal{X})$-ben szerepel. Amennyiben a szükséges új csomópontokat lokálisan hozzuk létre és a legfelső szinten esetlegesen létrehozott új csomópontot tekintjük az MDD gyökerének, a kapott MDD szintúgy kódolja $\mathcal{N}_{\mathcal{A}}^{-1}(\mathcal{X})$-et. Így hatékonyan képezhető $\mathcal{X} \cup \mathcal{N}_{\mathcal{A}}^{-1}(\mathcal{X})$ is az MDD-n belül is, ahogyan ezt a 4.3.7. fejezet a későbbiekben leírja.

4.3.4. Csomópontok egyezésvizsgálata

A szaturációs algoritmus során gyakran van szükség csomópontok összehasonlítására. Egyrészt bizonyos műveleteknél így derül ki, hogy a művelet okozott-e változást, másrészt jelentős mennyiségű összehasonlítást végez a hash-elt halmazos tárolás is. Ezt az összehasonlítást implementálja az *Equals* függvény.

Két, *azonos szinten* lévő csomópontot *egyenlőnek* tekintünk akkor, ha azonos függvényt kódol részfájuk. Másképp kifejezve két csomópont megegyezik, ha minden gyermekcsomópontjuk megegyezik.[1]

A csomópontok összehasonlítása során két eset lehetséges: a két csomópont azonos vagy különböző MDD-ben szerepel. Azonos MDD-ben szereplő csomópontok esetén könnyen implementálható az összehasonlítást végző *Equals* függvény, mindössze a gyermekcsomópontok referenciáit kell összehasonlítani, mivel a szaturációs algoritmus QMDD-ket használ, amiben egy szinten nem lehet azonos függvényt kódoló két csomópont. Ez egy gyors művelet, mindössze a csomópontok éllistáján kell végigiterálni.

Abban az esetben, ha a két csomópont különböző MDD-ben található, a helyzet nem ilyen egyszerű. Hiába kódolnak azonos függvényt két csomópont gyermekei, a különböző MDD miatt referenciájuk is különböző lesz. Ezért ilyen esetben az alacsonyabb szinten ugyanezt az *Equals* függvényt kell a csomópontok összehasonlítására meghívni, nem vezethető vissza referenciák egyenlőségvizsgálatára a probléma. A rekurzív hívás végül eljut a terminális csomópontokig, melyek már referencia szerint összehasonlíthatók. Ez jelentős komplexitást hoz be, amikor különböző MDD-k csomópontjain kell egyenlőséget vizsgálni. Ezért ez a módszer kerülendő, lehetőség szerint a műveleteket egy MDD-n belül kell végezni. Ahol lehetett, úgy módosítottam az algoritmusokat, hogy az egyenlőség lokálisan eldönthető legyen. Erről a 4.3.7. fejezetben olvashat bővebben.

A függvény egyszerűsített pszeudókódját mutatja a 7. algoritmus. A *RefEquals*(p, q) függvény jelöli p és q csomópont referencia szerinti egyezésvizsgálatát.

4.3.5. Általános szaturálás

Ahogyan azt a 3.2.3. fejezetben leírtam, az állapottér-generálásnál használt szaturáló algoritmus (*Saturate* függvény) a szaturálás egy speciális esetét valósítja meg, hiszen feltételezi, hogy üres MDD-ből indul az algoritmus. Emellett arra sincs felkészítve, hogy korlátozott eseményhalmazon végezze el a szaturálást. A modellellenőrzéshez a későbbiekben leírtak szerint szükség lesz az általános jellegű szaturálás megvalósítására. A következőkben a speciális algoritmuson végrehajtott módosításaimat mutatom be.

Általános szaturálás esetén az 1. szinttől a K.-ig minden k. szinten *az összes csomópontra* el kell tüzelni azon $\alpha \in \mathcal{A}$ eseményeket, melyekre $Top(\alpha) = k$. Néhány további

[1] Megjegyzendő, hogy QMDD esetén két egyenlő csomópont a tárolókban nem szerepelhet. Szintén fontos szem előtt tartani, hogy egyenlőségvizsgálat történhet egy tárolóbeli és egy nem tárolóbeli, aktuálisan manipulált csomópontot között is. Emiatt ideiglenesen két különböző objektum is leírhat egyenlő csomópontokat, ezért nem elég a csomópontok referenciáit vizsgálni.

Algoritmus 7 Equals

Input: p, q : csomópont
Output: logikai
1. **if** level(p) = 0 **then**
2. **return** RefEquals(p, q) // a terminális csomópontok referencia szerint összevethetők
3. **end if**
4. **if** p és q azonos MDD-ben szerepel **then**
5. **for** $i = 0$ to $|D_{level(p)}| - 1$ **do**
6. **if** RefEquals($p[i]$, $q[i]$) = **false then**
7. **return false**
8. **end if**
9. **end for**
10. **else**
11. **for** $i = 0$ to $|D_{level(p)}| - 1$ **do**
12. **if** Equals($p[i]$, $q[i]$) = **false then**
13. **return false**
14. **end if**
15. **end for**
16. **end if**
17. **return true**

járulékos módosítást is eszközölni kell. A speciális *Saturate* függvényben mindig új csomópontra hívtuk meg a Saturate függvényt és ez a csomópont sosem volt nullcsomópont. Itt viszont vizsgálni kell már, hogy nullcsomópontra az algoritmus ne hívódjon meg, mert ennek szaturálása értelmetlen.

Az általános szaturálást megvalósító *GeneralSaturate* függvény pszeudókódja a 8. algoritmus. A kódban $\mathcal{X}.SatFire_{\mathcal{A}}(\alpha, v)$ jelöli a korábban ismertetett *SatFire* függvény hívását $\mathcal{X}$ MDD-re úgy, hogy amennyiben *Saturate* függvényt hív, azt az $\mathcal{A}$ eseményhalmazra szűkítve teszi.

Algoritmus 8 GeneralSaturate

Input: $\mathcal{X}$: MDD, $\mathcal{A}$: eseményhalmaz
1. **for** $k = 1$ to K **do**
2. **for each** v csomópontra i. szinten, amely nem nullcsomópont **do**
3. $chng$ = **true**
4. **while** $chng$ = **true do**
5. $chng$ = **false**
6. **for each** $\alpha \in \mathcal{A} : Top(\alpha) = k$ **do**
7. $chng = \mathcal{X}$.SatFire$_{\mathcal{A}}(\alpha, v) \vee chng$
8. **end for**
9. **end while**
10. **end for**
11. **end for**

Fontos megjegyezni, hogy mivel a szaturáció $\mathcal{X}$-ben már elhelyezett csomópontokat módosít, ezért azokat először ki kell venni a tárolóból, majd módosítás után vissza kell rakni. Amennyiben a *SatFire* hívás eredményeként a v csomópont egy másik $\mathcal{X}$-beli w csomóponttal megegyezik, úgy a továbbiakban minden v-t w-vel kell helyettesíteni. Ezeket a műveleteket a pszeudókód az átláthatóság kedvéért nem tartalmazza, a valódi megvalósítás azonban természetesen igen.

Az itt leírt függvényből már készíthető egy visszafelé szaturáló függvény, ehhez a Kronecker-mátrixok transzponáltjaival kell számolni a *SatFire* függvényben (és az általa hívott függvényekben), ezt jelöltem a korábbiakban *GeneralBackSaturate* névvel.

4.3.6. Az A útvonalkvantorokat tartalmazó kifejezések visszavezetése

[11] alapján az A (for all futures) CTL temporális operátorok visszavezethetők E (for some future) operátorokra a következő szabályok alapján:

- $AX\ p = \neg EX(\neg p)$
- $AF\ p = \neg EG(\neg p)$
- $AG\ p = \neg EF(\neg p)$
- $A[p\ U\ q] = \neg E[\neg g\ U\ (\neg f \wedge \neg g)] \wedge \neg EG\neg g$

Mivel a szaturáció az E operátorokra alkalmazható hatékonyan [9], ezért az A operátorokat a fenti szabályok alapján visszavezettem a megfelelő E operátorokra. A gyorsabb működés érdekében az EF operátort is megvalósítottam szaturációs alapokon az EU operátor alapján.

4.3.7. Műveletek lokális megvalósítása

Számos mérési eredmény azt mutatta, hogy az amúgy is költséges műveletek nagyságrendekkel lassabbak, ha két különböző tárolóban található MDD-n végezzük, nem pedig egy MDD-tárolón belül. Ennek az a fő oka, hogy a 4.3.4. fejezetben leírtak miatt a csomópontok egyezésvizsgálata sokkal lassabb, ha a két összehasonlítandó csomópont eltérő MDD-ben van.

Ez a probléma felmerült több esetben is, például metszet- és unióképzés esetén vagy annak vizsgálatakor, hogy egy MDD által kódolt állapotok halmaza részhalmaza-e egy másik MDD által kódolt állapotok halmazának.

Több MDD tárolása egy tárolóban

A következő módszerek alapötlete az, hogy egy tárolóban több MDD is tárolható. Természetesen ezek nem feltétlenül lesznek diszjunktak, hiszen továbbra sem megengedett, hogy egy tárolóban egy szinten két azonos jelentésű csomópont legyen. Azonban minden MDD-t azonosít a gyökércsomópontja, ezért információt nem vesztünk ilyen esetben sem.

Azt a műveletet, melynek során egy $\mathcal{P}$ diagramot tartalmazó tárolóba egy $\mathcal{Q}$ diagramot is elhelyezek, *bemásolásnak* nevezem. Ennek során arra kell ügyelni, hogy két azonos csomópont ne jöjjön létre. Ezért az ezt megvalósító *CopyTo* függvény (9. algoritmus) a szinteken felfelé haladva (1-től K-ig) minden, $\mathcal{P}$-ben még nem létező csomópontot átmásol, egyezés esetén pedig eltárolja, hogy az egyező $\mathcal{Q}$-beli csomópont melyik $\mathcal{P}$-beli csomópontnak felel meg. Minden $\mathcal{Q}$-beli csomópontnál pedig megvizsgálja, hogy a gyermekei közül valamelyik cserére került-e, ebben az esetben frissíti a referenciákat. Az algoritmusban használt $Clone(q)$ függvény egy másolatot készít q csomópontról, a $\mathcal{P}.CheckIn(r)$ függvény pedig a 4.2.2 fejezetben leírt *CheckIn* függvény a $\mathcal{P}$ MDD-n meghívva.

Unió és metszet művelet lokális megvalósítása

Az eddig részletezett okok miatt $\mathcal{P} \cup \mathcal{Q}$ művelet esetén ahelyett, hogy a két MDD gyökérelemére meghívnám a csomópontok unióját képző függvényt, először $\mathcal{P}$ tárolójába bemásolom $\mathcal{Q}$ csomópontjait, majd a legfelső szinten lévő két csomópontra hívom meg az unió-függvényt. Így már kihasználható, hogy a csomópontok egy tárolóban vannak, viszont információt nem veszítünk, hiszen a gyökérelem azonosítja az MDD-t. Az eredmény egy legfelső szintű csomópont lesz, ez lesz a $\mathcal{P} \cup \mathcal{Q}$ gyökere. A többi legfelső szintű csomópontra innentől nincsen szükség, azok törölhetők. Hasonlóképp törölhetők azok a csomópontok is,

Algoritmus 9 CopyTo

Input: $\mathcal{P}, \mathcal{Q}$: MDD
1. d : (csomópont, csomópont) párok halmaza
2. **for** $i = 1$ to K **do**
3. **for each** q csomópontra i. szinten **do**
4. $r = \text{Clone}(q)$
5. **for** $j = 0$ to $|S_i|$ **do**
6. **if** d tartalmaz $(r[j], x)$ elemet tetszőleges x-re **then**
7. $r[j] = x$
8. **end if**
9. **end for**
10. **if** $\exists p \in \mathcal{P} : p = q, level(p) = level(q)$ **then**
11. $d = d \cup (q, p)$
12. **else**
13. $\mathcal{P}$.CheckIn(r)
14. $d = d \cup (q, r)$
15. **end if**
16. **end for**
17. **end for**

melyek alsóbb szinteken szerepelnek, viszont egyetlen él sem vezet beléjük. Az itt ismertetett módszert mutatja be a 12. példa.

Pontosan így megvalósítható a metszetképzés is, csak természetesen az unió hívásokat metszet hívásokkal kell helyettesíteni.

12. példa. *Vegyük a 4.5(a) ábrán látható $\mathcal{P}$ és a 4.5(b) ábrán látható $\mathcal{Q}$ bináris döntési diagramot! A lokális unióképzéshez először a $\mathcal{P}$ döntési diagram tárolójába kell másolni $\mathcal{Q}$-t. Ennek eredménye látható a 4.5(c) ábrán. Itt p jelöli $\mathcal{P}$ gyökerét, q pedig $\mathcal{Q}$ gyökércsomópontját. Ez után már p és q csomópontok uniója képezhető a döntési diagram tárolóján belül. Ennek eredménye látható a 4.5(d) ábrán.*

Az átláthatóbb ábrák érdekében a terminális nullába mutató éleket nem tüntettem fel.

Lokális részhalmaz-vizsgálat

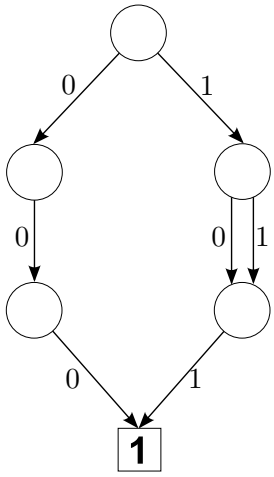
A fentiekben leírt ötlet akkor is felhasználható, amikor a $\mathcal{Q} \subseteq \mathcal{P}$ igazságtartalmát vizsgáljuk a 3.2.3. fejezetben részletezett *ClassifyEvents* függvényben.

Jelölje a $\mathcal{P}$ diagram gyökércsomópontját p, a $\mathcal{Q}$ diagram gyökerét pedig q. Vegyük $\mathcal{P} \cup \mathcal{Q}$-t a fentieknek megfelelően (azaz először $\mathcal{Q}$-t bemásoljuk $\mathcal{P}$-be), az így kapott diagram gyökerét jelölje r. Amennyiben $p = r$, azaz $\mathcal{P} \cup \mathcal{Q} = \mathcal{P}$, akkor $\mathcal{Q} \subseteq \mathcal{P}$, különben nem. A mérések itt is azt mutatják, hogy bár e módszer kicsivel nagyobb memóriaigénnyel jár, hiszen néhány új csomópont létrehozás szükséges, viszont több nagyságrenddel gyorsabb.

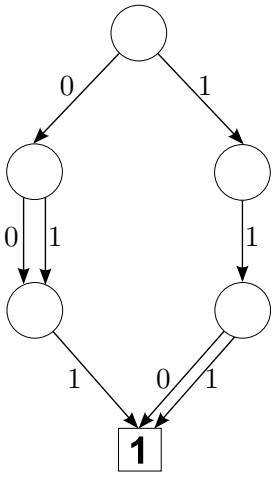
Ezzel és a fejezetben leírt többi megoldással az amúgy lassú, a teljes futásidő szempontjából meghatározó szerepű *ClassifyEvents* függvény jelentősen felgyorsítható.

4.4. Memóriaoptimalizálások

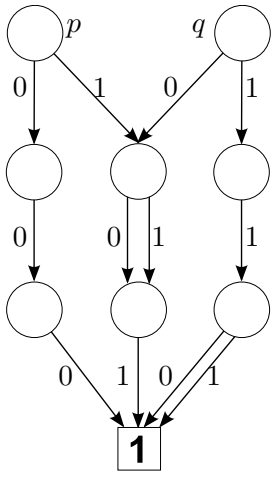
A modellellenőrzés rendszerint egy komplex, memóriaintenzív folyamat. Emiatt a teljesítményt jelentősen javíthatják azok a megoldások, melyek a memóriahasználatot akár csak kis mértékben is javítják. A fejezetben két, általam alkalmazott fejlesztést írok le, melyek a program memóriahasználatát optimalizálják.



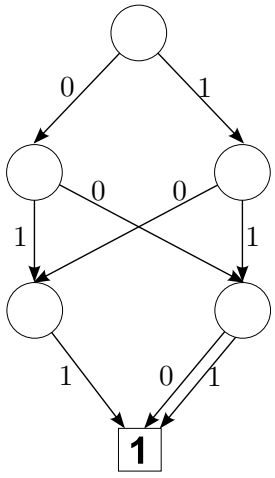
(a) $\mathcal{P}$ bináris döntési diagram



(b) $\mathcal{Q}$ bináris döntési diagram



(c) $\mathcal{P}$ és $\mathcal{Q}$ bináris döntési diagram közös tárolóban



(d) $\mathcal{P} \cup \mathcal{Q}$-t reprezentáló BDD

4.5. ábra. Lokális műveletvégzés döntési diagramokon

4.4.1. Csomópontok újrahasznosítása

Bár a szaturációs algoritmus hatékony abból a szempontból, hogy kevés felesleges MDD csomópontot hoz létre, így is keletkeznek felesleges csomópontok. Különösen igaz ez akkor, ha az MDD szintezése nem kedvező. Ha ezeket a csomópontokat kitöröljük és a garbage collectorra bízzuk, majd létrehozunk új csomópontot, kétszer is felesleges műveletet végzünk: feleslegesen terheljük a garbage collectort és egy felesleges memóriafoglalási lépés is bekerül. Ezért ehelyett a törlésre ítélt csomópontokat egy NodeTrash tárolóban eltárolom, majd új csomópont létrehozásakor ennek egy elemét veszem ki. Természetesen limitálni szükséges e tároló méretét, egy korlát fölött a törölt csomópontok nem kerülnek a tárolóba, hanem a garbage collector ténylegesen eltakarítja. A csomópontok újrahasznosításainak hatásait az 5.1. fejezetben mérem le.

4.4.2. Garbage collector manuális hívása

Különösen az EU operátort megvalósító függvények során jelentős mennyiségű adat kerül be rövid időkre a memóriába. Ennek következménye, hogy a .NET garbage collectornak sok memóriaterületet kell felszabadítania. Mérési eredmények azt mutatták (lásd az 5.1. fejezetben), hogy a garbage collector program szempontjából nemdeterminisztikus automatikus futásai nem kellőképp gyakoriak. Sőt, előfordul bizonyos esetekben, hogy az elégtelen GC-hívások miatt elfoglalja a teljes rendelkezésre álló memóriát, de a garbage collector még ilyenkor sem fut le, így a program futása nagyságrendekkel lelassul.

Emiatt időnként a garbage collectort a kódból kényszeríteni kell a futásra. Erre tipikusan olyan helyeken van szükség, ahol nagy mennyiségű felesleges adat keletkezik, ilyen a *ClassifyEvents* és az *EUsat* függvény. Amennyiben azonban túl sok kézi garbage collector hívás van, az jelentősen rontja a teljesítményt. Számos mérés után jó kompromisszumnak adódott, hogy minden *EUsat* függvénybeli iteráció után, valamint minden 30. *ClassifyEvents*-beli iteráció után hívom meg a szemétgyűjtő algoritmust.

4.5. Heurisztikák megvalósítása

A fejlesztés során cél egy olyan program implementálása volt, mely használható anélkül, hogy a felhasználó jártas lenne a szaturációs algoritmusokban, szimbolikus módszerekben és a többértékű döntési diagramok működésében. Ennek érdekében különböző heurisztikákat implementáltam.

- A 4.5.1. és 4.5.2. fejezetekben leírt heurisztikák az állapottér-generálás előtt készítik el különféle módszerek alapján a modell dekomponálását és az egyes részmodellek MDD-szintekhez rendelését.
- A 4.5.4. fejezetben egy olyan heurisztikát mutatok be, mely a modellellenőrzést hivatott gyorsítani azáltal, hogy az adott kifejezéshez próbálja optimalizálni az MDD szintjeinek sorrendjét. Ehhez meg kellett valósítani az MDD-kre a szintcsere műveletét, amit a 4.5.3. fejezetben vezetek be. A 4.5.5. fejezetben egy olyan heurisztikát írok le, mellyel egy MDD mérete (csomópontszáma) lecsökkenthető.

Utóbbi két heurisztika újdonságot jelent az eddigi módszerekhez képest, a konkrét kiértékelendő kifejezésre való optimalizálás ötlete az irodalomban eddig nem merült fel a szaturációs algoritmusokkal kapcsolatban.

Fontos megjegyezni, hogy a heurisztikák kihasználják a háló strukturális tulajdonságait a P- és T-invariánsokon keresztül. Az invariánsok kiszámítását a *PetriDotNet* keretrendszer egy általam megírt modulja végzi az ún. Martinez–Silva algoritmussal [18], mely mátrixszámításokkal állítja elő az invariánsokat. Ez okozza az algoritmus implementálásának

nehézségét is, mivel a modell méretének növekedésével körülbelül négyzetesen növekszik a használt mátrix mérete is. Az invariánsok számítási módja, ennek fejlesztése, gyorsítása azonban helyhiány miatt nem szerepel ebben a dolgozatban.

4.5.1. P-invariáns alapú heurisztika

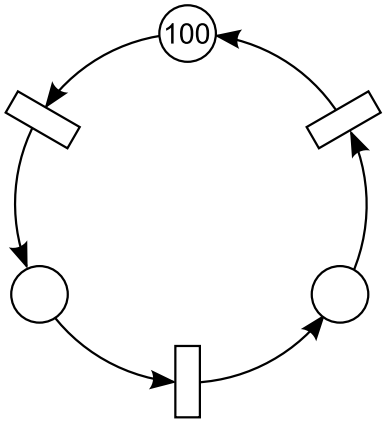
A P-invariáns heurisztika alapötlete az, hogy a szorosan összefüggő helyek egy szintre kerüljenek. Így valószínűbb az is, hogy lesznek olyan t tranzíciók, melyek csak egyetlen szintet befolyásolnak, azaz $Top(t) = Bot(t)$. Ezzel növelhető a felbontásban lévő lokalitás, ami gyorsulást okoz.

A heurisztika a P-invariánsokból indul ki, majd ezekből sorban készíti el az MDD egyes szintjeit. Fontos megjegyezni, hogy amikor egy invariánsból szintet hozunk létre, az ebben szereplő helyeket ki kell törölni a maradék invariánsokból, ugyanis egy hely több invariánsban is szerepelhet, de egy hely pontosan egy szinthez tartozhat csak, mivel a felbontás nem lenne Kronecker-konzisztens, ha a szintekhez tartozó helyek halmaza nem lenne diszjunkt.

Kis részmodellekből álló Petri-hálók esetén a P-invariáns heurisztika hatékonynak mutatkozik. Azonban észre kell venni, hogy a P-invariánsok egyben tartása bizonyos modelleknél problémát is okozhat. Ha egy P-invariánsban sok token kering, az óriási lokális állapottérhez vezethet, ami nagy éllistákat, lassú iterációkat okoz és így teljesítménycsökkenést okoz. A lokális állapottér nagy méretét szemlélteti a 13. példa.

13. példa. *Tekintsük a 4.6. ábrán látható 3 hosszúságú kört. Ebben kezdőállapotban 100 token található és a tokenek összege minden esetben állandó lesz. Bár ez az egész modell egy invariáns, mégsem célszerű dekomponálás nélkül generálni az állapotteret. Az ábrán látható háló elérhető állapotainak száma 5151, tehát ennyi él indulna ki a kört leíró csomópont(ok)ból. Egyetlen újabb token behelyezésével ez a szám már 5253-ra növekszik. C token esetén az állapotok száma $O(C^2)$.*

Egy n hosszúságú kör esetén T kezdő össztokenszám mellett az állapotok száma már $O(C^{n-1})$-re nő. Ilyen esetekben (nagy C és n értékek esetén) nyilvánvalóan káros egyben hagyni az invariánsokat.



4.6. ábra. Három helyből álló P-invariáns

4.5.2. Eseménylokalitásra és rekurzióra optimalizált heurisztika

Az eseménylokalitásra és rekurzióra optimalizált heurisztika (összetett heurisztika) két részből tevődik össze:

- először P-invariánsokon alapuló metrikák szerint megtörténik a Petri-háló *részmodellekre bontása*,
- majd T-invariánsokon alapuló metrikák alapján kialakul a részmodellek egy *sorrendezése*.

Részmodellekre bontás

A 4.5.1. fejezetben bemutatott heurisztikához hasonlóan jelen heurisztika is a P-invariánsokból indul ki, azonban a részmodellekre bontás során a P-invariánsokból származtatott metrikákat is figyelembe veszek. A kiindulási pont az, hogy ha egy p hely szerepel egy P-invariánsban, amelyben a keringő tokenek összege C, akkor várhatóan p-nek legfeljebb $C + 1$ állapota lesz. Amennyiben p szerepel egy C_1 és egy C_2 tokenösszegű P-invariánsban is, akkor természetesen legfeljebb $\min(C_1, C_2) + 1$ állapota lehet.

Ez alapján az egyes P-invariánsokra megadható egy felső állapottérbecslés, mely az egyes helyek állapotszámainak szorzata. Természetesen ez nem egy pontos számítás, de könnyen elvégezhető és elfogadható eredménnyel szolgál. Amennyiben ez egy korlátnál magasabb, az adott helyhalmazt ketté kell bontani, majd ezeket újra megvizsgálni.

Az algoritmus a T-invariánsokat is kihasználja, ugyanis ha egy P-invariáns helyhalmazát fel kell osztani két részre, célszerű lenne ezt úgy tenni, hogy minél kevesebb tranzíció befolyásolja mindkét létrejövő szintet. Ennek érdekében a T-invariánsok alapján a heurisztika sorrendbe igyekszik rendezni a helyeket egy a hálón végzett mélységi bejárás segítségével. Sajnos a sorba rendezés nem feltétlen lehetséges, mivel egy invariánsban elágazás is lehet.

Fontos azt is figyelembe venni, hogy egy hely több invariánsban is szerepelhet, viszont egy hely pontosan egy szinthez tartozhat az MDD-ben. Így amikor egy szint létrejön, az abban szereplő helyeket ki kell törölni a maradék helyhalmazokból.

A fentiek eredményeképp előállnak a hálóbeli helyeknek olyan halmazai, melyek lokális állapottere várhatóan kezelhető méretű.

Sorrendezés

A sorrendezés alapját [6] adta. Eszerint a szaturáció akkor hatékony, ha a tranzíciók kevés szintet fognak át, azaz ha $U = \sum_{t \in T} Top(t) - Bot(t) + 1$ kicsi. (A T a háló tranzícióit jelöli, ahogyan a 2.1.1. fejezetben is olvasható.) Ez a cikk szerint egy gyorsan számítható mérték, mely jó eredményt hoz.

Ezt a megoldást alkalmaztam jelen esetben is az előző fejezetben leírtak szerint felbontott szintek sorrendezésénél. Az algoritmus egy kiválasztásos rendezés mintájára íródott függvény, mely az aktuális szintet azzal a szinttel cserélni fel, amellyel a szintcsere az U metrikát a legkisebbre csökkenti.

4.5.3. Szintcsere megvalósítása

A következő fejezetben bemutatandó heurisztikához szükséges egy új MDD-műveletet bevezetni, ez a szintcsere. Szintcsere során a k. és $k - 1$. szintet cseréljük fel úgy, hogy az MDD továbbra is változatlan függvényt kódoljon. Természetesen k csak olyan értéket vehet fel, hogy a szintcsere ne érintse a terminális csomópontokat tartalmazó 0. szintet, azaz $2 \leq k \leq K - 1$, ahol az eddigi jelöléseknek megfelelően K a szintek számát jelöli, tehát a legmagasabb szint sorszáma $K - 1$.

A szintcserét úgy hajtjuk végre, hogy ha a k. szinten lévő v csomópontra $v[i][j] = w$, akkor a csere után $v[j][i] = w$ igaz legyen. Azaz a k. és $k - 1$. szint cseréje a $k - 2$. szintet már nem befolyásolja. Amennyiben egy $k + 1$. szintű u csomópontra $u[l] = v$ igaz volt,

akkor ez a szintcsere után is igaz marad, azaz az iménti szintcsere a $k+1$. szintet sem befolyásolja [20].

A szintcsere a következőképp zajlik:

1. Minden k. szintű v csomópontra elkészítünk egy csomópontokat tartalmazó, $|D_k| \times |D_{k-1}|$ méretű $\mathbf{C}_v$ mátrixot. Ezeket a mátrixokat kitöltjük úgy, hogy

$$\mathbf{C}_v[i,j] = w \Leftrightarrow v[i][j] = w.$$

2. Ez után az összes k. szintű v csomópont gyerekeit törljük, majd újraépítjük úgy, hogy $v[a][b] = w \Leftrightarrow \mathbf{C}_v^T[a,b] = w$.

A pszeudókódokban a j. és $j+1$. szint felcserélését a *SwapLevels*(j, $j+1$) függvényhívás jelöli.

4.5.4. Lokalitás alapú szintcsere heurisztika

A lokalitás alapú szintcsere heurisztika célja az, hogy a konkrét kiértékelendő CTL-kifejezés ismeretében úgy rendezze át az egyes szinteket, hogy a konkrét kifejezés ellenőrzése minél gyorsabb legyen. Ez főként az EU kifejezések kiértékelésekor fontos, mivel a 3.2.3. fejezetben leírtak alapján ilyenkor csak az események egy részhalmazával tudunk szaturációt végrehajtani, tehát az MDD-t célszerű úgy átrendezni, hogy erre az állapothalmazra minél hatékonyabb szaturáció legyen végezhető.

Ebben a heurisztikában is a [6] szerinti metrikát használtam fel, azaz úgy rendezem át a szinteket, hogy az $U_{\mathcal{A}} = \sum_{t \in \mathcal{A}} Top(t) - Bot(t) + 1$ mérték minél kisebb legyen, azonban csak az $\mathcal{A}$ halmazbeli eseményekre.

A lehetséges sorrendezések száma n szint esetén $O(n!)$. Emiatt már kisebb MDD-k esetén sem lehet az összes lehetséges sorrendezést kipróbálni, így egyszerűsítést eszközöltem. Az algoritmus sorban, alulról felfelé minden i. szintet megpróbál feljebb helyezni. Ezek közül arra az átszintezésre tér át, mely a legjobb metrikaértékkel rendelkezik.

Mivel a szintek valós cseréje költséges művelet, ezért az itt leírt algoritmus először lefut tényleges átszintezés nélkül, majd ezután egy buborékos rendezés jellegű algoritmus segítségével történik meg a szintek igazi cseréje, így biztosítva, hogy csak a szükséges szintcserék történjenek meg.

A lokalitás alapú szintcsere heurisztikát mutatja be a 10. algoritmus. Ebben $\mathcal{A}$ jelöli azt az eseményhalmazt, melyre az $U_{\mathcal{A}}$ mérték minimalizálásra kerül. Egy l változóba egy aktuális szintsorrendezés elmentését a *SaveLevels*(l), egy l változóba elmentett szintsorrendezés visszatöltését a *RestoreLevels*(l) hívás jelöli.

4.5.5. MDD-minimalizáló szintcsere heurisztika

Számos esetben, például metszetképzésnél a futásidő szempontjából fontos, hogy az MDD-ben hány csomópont található. A tapasztalataim azt mutatják, hogy a szintek átrendezésével az MDD-k mérete jelentősen redukálható. Ennek kihasználására készült el egy MDD-minimalizáló heurisztika, mely a döntési diagram méretét igyekszik csökkenteni. Az MDD mérete főképp az EG kifejezések kiértékelésekor fontos, mivel ezek megvalósítása nem szaturáción alapszik. Fontos megjegyezni, hogy ez az első ilyen jellegű próbálkozás az MDD-k esetén.

A 4.5.4. fejezetben leírtak alapján itt sem lehetséges az összes lehetőség kipróbálása. Ezért a következő egyszerűsített módszert alkalmaztam: az algoritmus felülről lefelé minden szintet igyekszik az alatta lévővel kicserélni mindaddig, amíg ez az MDD méretében

Algoritmus 10 LevelSwappingHeuristicByTopBot

Input: $\mathcal{A}$: eseményhalmaz
1. **for** $i = 1$ to $K - 2$ **do**
2. $min = \sum_{\alpha \in \mathcal{A}} Top(\alpha) - Bot(\alpha)$
3. $levels = \emptyset$
4. **for** $j = i$ to $K - 1$ **do**
5. SwapLevels(j, $j + 1$) // j. és $j + 1$. szint cseréje
6. **if** $min > \sum_{\alpha \in \mathcal{A}} Top(\alpha) - Bot(\alpha)$ **then**
7. $min = \sum_{\alpha \in \mathcal{A}} Top(\alpha) - Bot(\alpha)$
8. SaveLevels($levels$) // aktuális szintsorrend eltárolása
9. **end if**
10. **end for**
11. **if** $levels \neq \emptyset$ **then**
12. RestoreLevels($levels$) // eltárolt szintsorrend visszaállítása
13. **end if**
14. **end for**

csökkenést okoz. A tapasztalatok azt mutatják, hogy az algoritmus bizonyos modelleknél képes drasztikus csomópont-csökkentésre. Ennek hatását a program teljesítményére az 5.2.2. fejezet mutatja be.

A leírtak pszeudókódját mutatja a 11. algoritmus. Ebben m az az MDD, melynek méretét csökkenteni kívánjuk a heurisztika segítségével. A $nodeCount(m)$ megadja az m MDD-ben található csomópontok számát.

Algoritmus 11 LevelSwappingHeuristicByMDDSize

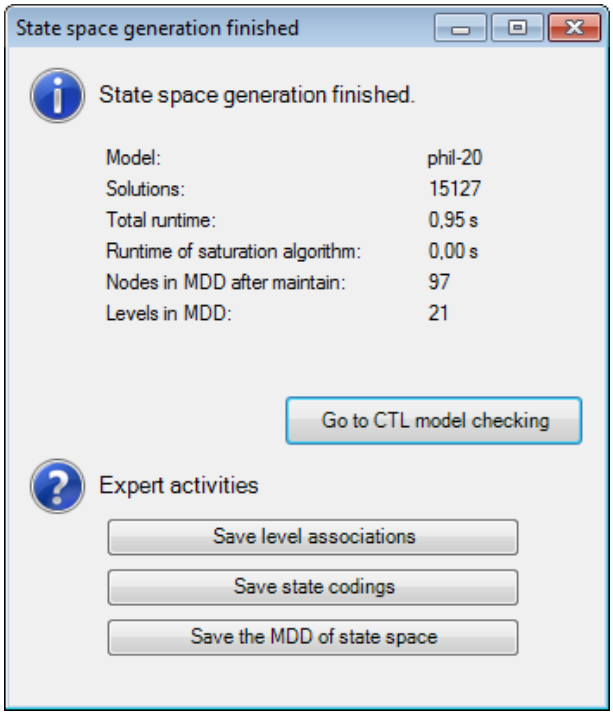
Input: m: MDD // m a metrika alapjául szolgáló MDD
1. $min = nodeCount(m)$ // min-be kerül m csomópontjainak száma
2. **for** $i = K$ downto 2 **do**
3. **for** $j = i$ downto 2 **do**
4. SwapLevels(j, $j + 1$) // j. és $j + 1$. szint cseréje
5. **if** $min < nodeCount(m)$ **then**
6. SwapLevels(j, $j + 1$) // rossz lépés visszavonása
7. break for // kilépés a ciklusból
8. **else**
9. $min = nodeCount(m)$
10. **end if**
11. **end for**
12. **end for**

4.6. Az elkészített program funkcionalitása felhasználói oldalról

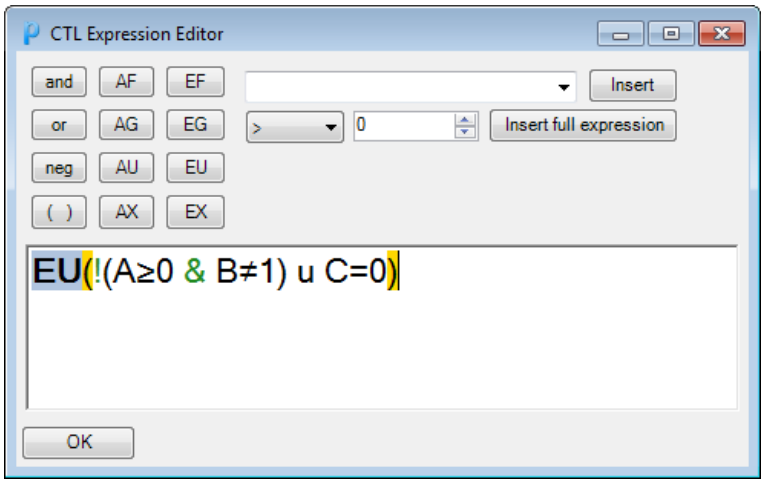
Az elkészített program a *PetriDotNet* keretrendszer egyik beépülő modulja. Elindítása után a keretrendszerben megnyitott Petri-háló analízisére van lehetőség. A program segítségével egyszerű Petri-hálók analízise végezhető el, melyek kiegészülhetnek kapacitáskorlátokkal, tiltó élekkel és lehetnek hierarchikus felépítésűek is.

A beépülő modul elindítása után lehetőség van szintezési algoritmus választására. A szintezés elvégezhető manuálisan vagy a fejezet elején leírt két szinthozzárendelő heurisztika egyikével. Ez után a program elvégzi az állapottér felderítését, erről statisztikai információkkal is szolgál (4.7. ábra). Ennek végeztével egy grafikus felületen megadhatók CTL-kifejezések (4.8. ábra), melyeket feldolgoz, kiértékel és eredményét kijelzi.

A CTL-kifejezések mind a 12, a 2.4. fejezetben definiált operátort tartalmazhatják (AG, AX, AF, AU, AW, AR, EG, EX, EF, EU, EW, ER) és a kifejezések akár egymásba is ágyazhatók. A helyes szintaxis leírása megtalálható a program felhasználói dokumentációjában, a *PetriDotNet* keretrendszer honlapján [23].



4.7. ábra. Az állapottér-generálás adatait jelző ablak



4.8. ábra. A CTL-kifejezés szerkesztőablaka

5. fejezet

Eredmények

A következőkben mérésekkel támasztom alá az implementált fejlesztéseket és a szaturáció hatékonyságát különböző szempontok szerint. Az 5.1. fejezetben az előző fejezetben leírt főbb implementációs fejlesztések hatásait mutatom be. Az 5.2. fejezetben leírom az egyes heurisztikák által elért eredményeket. Az 5.3. fejezetben más, elterjedt modellellenőrzőkkel vetem össze az elért eredményeket.

Az itt leírt méréseket a következő konfiguráción végeztem: Intel Q8400 2,6 GHz processzor, 4 GB memória, Windows 7 (x64) operációs rendszer, .NET 4.0 futtatókörnyezet, PetriDotNet 1.2 verzió. A méréseknél az időadatok mindenhol három mérés átlagából adódtak és másodpercben szerepelnek. Heurisztikáknál feltüntetésre kerül a teljes idő is, mely magában foglalja a heurisztika futásidejét.

A mérésekhez használt különböző modellek leírása a C. függelékben található. A modellek PNML formátumúak [22]. A nagyobb modellek (különböző típusú étkező filozófusok modellek, réselt gyűrű modell) elkészítése egy automatikus generáló programmal történt. Ezen modelleknél a fájlokban az egyes részmodellek egymás után következnek. A felhasznált modellek PNML formátumban letölthetők a *PetriDotNet* keretrendszer honlapjáról [23].

Az alábbi méréseket sajnos [5][8][9] eredményeivel nem lehet összevetni. Ennek több oka van: egyrészt nem tették közzé az elkészített programot, illetve nem specifikálták kellően részletesen a használt hardverkonfigurációt, másrészt a mérési eredményeikbe nem számították bele bizonyos kulcsfontosságú lépések időtartamát.

5.1. Implementációs fejlesztések hatásai

A 4. fejezetben számos implementációs fejlesztést írtam le. Ezek közül három hatását vizsgálom meg a következő mérések segítségével: a lokalitás kihasználását (4.3.7. fejezet), a csomópontok újrahasznosítását (4.4.1. fejezet) és a megfelelő Kronecker-mátrix implementációt (4.2.3. fejezet).

Referenciának egy olyan a) programverziót tekintek, amelyben a műveletek (unió, metszet, részhalmaz-vizsgálat) nem lokálisan vannak megvalósítva, a csomópontok nem kerülnek újrahasznosításra, a Kronecker-mátrixok pedig listával vannak megvalósítva. A b) verzióban a műveletek megvalósítása lokális. A c) verzióban a műveletek lokálisak és a csomópontok is újrahasznosításra kerülnek, míg a d) verzióban a Kronecker-mátrixok is a 4.2.3. fejezetben leírtak szerinti megvalósításúak.

Minden mérés során manuálisan végeztem el a szinthozzárendeléseket. A filozófusoknál egy szintre egy filozófus és egy pálcika került, a réselt gyűrű modellnél egy alrendszer képezett egy szintet, a rugalmas gyártórendszer modellben pedig minden hely külön szintre került. Ez után állapottér-generálást és egy megadott EU kifejezés kiértékelését futtattam

le. Tekintve, hogy az állapottér-generálás sebességét ezek a fejlesztések csak kis mértékben befolyásolják, a táblázatban csak a modellellenőrzés időadatait tüntettem fel.

5.1. táblázat. Implementációs fejlesztések hatásai

N	$\lvert\mathcal{S}\rvert$	a) Referencia	b) Lokalitás	c) Csomópontok újrahasznosítása	d) Hatékony Kronecker-mátrix
Étkező filozófusok (Phil) CTL-kifejezés: $E(eszik_1 = 0\ U\ eszik_2 = 1)$					
20	15127	0,44 s	0,1 s	0,09 s	0,08 s
50	$2,81 \cdot 10^{10}$	$>$ 1800 s	0,25 s	0,21 s	0,21 s
100	$7,92 \cdot 10^{20}$	$>$ 1800 s	0,84 s	0,74 s	0,77 s
1000	$9,72 \cdot 10^{208}$	$>$ 1800 s	93,64 s	85,46 s	84,03 s
Étkező filozófusok (deadlockos, DPhil) CTL-kifejezés: $E[\neg(HasLeft_2 > 0 \wedge HasRight_2 > 0)\ U\ (HasLeft_1 > 0 \wedge HasRight_1 > 0)]$					
15	$1,46 \cdot 10^{15}$	$>$ 1800 s	0,20 s	0,11 s	0,12 s
100	$4,96 \cdot 10^{62}$	$>$ 1800 s	4,18 s	2,55 s	2,56 s
500	$3,03 \cdot 10^{313}$	$>$ 1800 s	81,81 s	73,30 s	73,76 s
Réselt gyűrű (SR) CTL-kifejezés: $E(B_1 \neq 1 \vee F_1 \neq 1\ U\ G_2 = 1 \wedge A_2 = 1)$					
30	10^{31}	$>$ 1800 s	28,00 s	27,49 s	25,46 s
Rugalmas gyártórendszer (FMS) CTL-kifejezés: $E(M1 > 0\ U\ (P1s = N \wedge P2s = N \wedge P3s = N))$					
15	$2,2 \cdot 10^{11}$	$>$ 1800	107,49 s	105,42 s	86,44 s

A fentiekből látható, hogy a műveletek lokális megvalósítása rendkívüli gyorsulást hoz, amely jelentősen kiterjeszti a modellellenőrzés alá vethető modellek körét. A másik két fejlesztés nem egyforma hatású minden modellre. Az étkező filozófus probléma esetén az látható, hogy a csomópontok újrahasznosítása jelentős gyorsulást hoz, viszont a Kronecker-mátrixok hatékonyabb reprezentációja gyakorlatilag nem befolyásolja a futási eredményeket. A rugalmas gyártórendszer (FMS) esetében pont fordított a helyzet: a csomópontok újrahasznosítása alig gyorsítja a modell ellenőrzését, viszont a hatékony Kronecker-reprezentáció jelentősen. Ennek az az oka, hogy az FMS modellben a csomópontok száma alacsony, viszont az állapotok száma az egyes szinteken relatíve nagy. Ezzel szemben az étkező filozófusoknál az egyes szintek állapotszáma kicsi, de nagy számú csomópont jön létre.

5.2. Heurisztikák eredményei

A 4.5. fejezetben négy heurisztikát ismertettem, ezek mérési eredményei következnek ebben a fejezetben. A heurisztikák két részre oszthatók: az első kettő az állapottér-generálás előtt a kezdeti szintezésnél használható, a másik kettő pedig a modellellenőrzés előtt optimalizálja az MDD-t. A mérési eredményeket is eszerint kettéosztva közlöm.

5.2.1. Heurisztikák eredményei az állapottér-generálásban

Az egyes modelleknél kétféle manuális szintezés eredményeit vetettem össze a kétféle heurisztika eredményeivel. Az egyik manuális szintezés esetén minden hely külön szintre került, a másiknál pedig egy-egy részmodell került egy-egy szintre. A mérési eredményeket a következőkben modellenkénti bontásban közlöm.

Az idő oszlopokban az állapottér-generáláshoz szükséges idő szerepel másodpercben. A csomópont oszlop a felderített állapottérben szereplő csomópontok számát tartalmazza, a szint oszlop pedig a létrehozott szintek számát. A heurisztikák esetén szerepel a teljes idő is (helyenként t. időnek rövidítve), ez a heurisztika futási ideje és az állapottér-generálás ideje együttesen. Ebbe beletartoznak az invariánsszámítások is, melyek nem kerültek op-

timalizálásra. A > 1800 celláknak megfelelő mérések során 30 percnél hosszabb volt a futásidő.

Étkező filozófusok modellek

A kétféle étkező filozófusok modell állapottér-generálásának mérési eredményei az 5.2. táblázatban találhatók.

Ezen modelleknél az „egy részmodell szintenként” szintezés optimális (tehát Phil modelleknél egy szinthez 3, DPhil modelleknél egy szinthez 6 hely tartozik). Az optimális szintezésnél az egymás mellett lévő filozófusok részmodelljei egymás melletti szinteken találhatók. A mérési eredményekből látható, hogy nem túlzottan nagy modellek esetén mindkét szintező heurisztika megközelítette az optimális megoldást. Nagyobb modellek esetén a P-invariáns heurisztika eredményei jobbnak bizonyultak.

Fontos megjegyezni, hogy a heurisztikák során jelentős időt vett igénybe az invariánsszámítás, azaz a Martinez–Silva-algoritmus lecserélése várhatóan további javulást eredményez a heurisztikáknál. Nagyon nagy modellek esetén a heurisztikák óriási időnövekedését is az okozta, hogy a Martinez–Silva-algoritmus túlzottan lassan adott eredményt. Jól látható ez a Phil-10k modellnél, ahol a P-invariánsok számítása már közel öt percet vett igénybe, maga az állapottér-generálás viszont csak 3,31 másodpercig tartott.

5.2. táblázat. Heurisztikák eredményei az állapottér-generálásban étkező filozófus modelleknél

Modell	**1 hely szintenként**			**1 részmodell szintenként**			**P-invariáns heurisztika**				**Összetett heurisztika**			
	idő	csomóp	szint	idő	csomóp	szint	idő	t. idő	csomóp	szint	idő	t. idő	csomóp	szint
Phil-100	0,01	1488	301	0,01	497	101	0,09	0,25	1186	201	0,08	0,26	3507	122
Phil-1000	0,24	14988	3001	0,14	4997	1001	0,22	2,4	11986	2001	5,9	9,89	157740	1171
Phil-10k	3,67	149988	30001	1,49	49997	10001	3,31	298,96	119986	20001	—	> 1800	—	—
DPhil-100	0,09	6728	601	0,03	499	101	0,05	0,14	2672	301	0,29	0,44	9733	243
DPhil-500	0,53	33928	3001	0,16	2499	501	0,39	2,37	13472	1501	6,07	9,59	134197	1171
DPhil-1000	1,35	67928	6001	0,47	4999	1001	0,76	9,26	26972	3001	32,76	47,98	603201	2342
DPhil-5000	9,16	339928	30001	1,73	24999	5001	4,29	282,94	134972	15001	—	> 1800	—	—

Rugalmas gyártórendszer modell

A rugalmas gyártórendszer modell nem osztható részmodellekre, így ilyen mérést nem végeztem. A 13. példa alapján a P-invariánsok szerinti szintezés itt nem ad jó eredményt, mivel az egyes szintek lokális állapotainak száma nagyon nagy lesz.

Ennél a modellnél a közel optimális kézi szintezés az, amikor egy-egy szinthez egy-egy helyet rendelünk olyan sorrendben, ahogy a helyek csatlakoznak egymáshoz tranzíciókon keresztül. Az 5.3. táblázatban látható eredmények alapján erre a modellre az összetett heurisztika jól közelíti ezt a közel optimális megoldást. Megjegyzendő azonban, hogy egy nem szakértő felhasználó szinte biztosan olyan szintezést állítana elő, melyre már az állapottér-generálás is elfogadhatatlanul hosszú időbe telne. Így az összetett heurisztikával elérhető eredmény különösen felértékelődik.

Réselt gyűrű modell

Réselt gyűrű esetén a T-invariánsokat kereső Martinez–Silva algoritmus már kis n-ekre is jelentősen visszaveti a teljesítményt, így ilyen méréseket nem végeztem. Az 5.4. táblázat eredményei alapján az látható, hogy még az egy szinthez egy részmodellt rendelő, igen jó szinthozzárendelésnél is jobb eredményekkel szolgál a P-invariáns alapú szintező algoritmus.

5.3. táblázat. Heurisztikák eredményei az állapottér-generálásban rugalmas gyártórendszer modellnél

Modell	**1 hely szintenként**			**Összetett heurisztika**			
	idő	csomópont	szint	idő	teljes idő	csomópont	szint
FMS-15	0,14	1108	23	0,23	0,33	3917	22
FMS-25	0,21	2568	23	0,96	0,98	10372	22
FMS-100	12,76	32643	23	74,21	74,86	157597	22
FMS-150	56,64	71443	23	309,63	312,09	352622	22

5.4. táblázat. Heurisztikák eredményei az állapottér-generálásban réselt gyűrű modellnél

Modell	**1 hely szintenként**			**1 részmodell szintenként**			**P-invariáns heurisztika**			
	idő	csomópont	szint	idő	csomópont	szint	idő	teljes idő	csomópont	szint
SR-30	1,56	9167	241	0,6	512	31	0,35	0,46	1027	62
SR-50	6,83	24777	401	2,56	1352	51	1,55	1,74	2707	102
SR-100	84,37	97052	801	22,59	5202	101	14,16	14,77	10407	202
SR-150	> 1800	—	—	97,99	11552	151	72,11	73,79	23107	302

Egyéb modellek

Az állapottér-generálás algoritmusát néhány olyan modellre is kipróbáltam, melyek valós rendszerek modelljeit jól közelítik. Ezek eredményei az 5.5. táblázatban találhatók. Mivel ezek nem bonthatók egyforma részmodellekre, így ilyen méréseket nem végeztem. A mért értékekből az látható, hogy a jellegében az FMS-re hasonlító DFMS modell esetén a szintenként egy hely rendeléséhez képest mindkét heurisztika kiváló eredményeket ért el. A laBPEL modell esetén is a P-invariáns heurisztika megközelítette a szintenként egy hely sorrendezést. Erre a modellre az összetett heurisztika nem fut le a többivel összemérhető idő alatt, de ez is a Martinez–Silva-algoritmus lassú T-invariánsszámítására vezethető vissza.

5.5. táblázat. Heurisztikák eredményei az állapottér-generálásban egyéb modelleknél

Modell	**1 hely szintenként**			**P-invariáns heurisztika**				**Összetett heurisztika**			
	idő	csomópont	szint	idő	teljes idő	csomópont	szint	idő	teljes idő	csomópont	szint
laBPEL	1,37	8764	88	1,95	2	978	23	—	> 120	—	—
DFMS-10	0,15	599	16	0,01	0,08	144	8	0,04	0,09	317	11
DFMS-100	0,61	599	16	0,02	0,03	144	8	0,04	0,05	317	11
DFMS-250	1,79	599	16	0,03	0,03	144	8	0,06	0,05	317	11
DFMS-1000	16,9	599	16	0,13	0,12	144	8	0,15	0,15	317	11
DFMS-10000	>1800	599	16	10,18	10,17	144	8	10,38	10,37	317	11

5.2.2. Modellellenőrzést javító heurisztikák eredményei

Lokalitás alapú szintcsere heurisztika eredményei

Az 5.6. táblázatban leírt mérésekkel a 4.5.4. fejezetben leírt lokalitás alapú szintcsere heurisztika eredményességét vizsgálom. A mérést futási időkből látható, hogy – legalábbis a vizsgált modellek esetében – a [6]-ban javasolt metrika a gyakorlatban nem hoz jelentős gyorsulást. Ennek az oka az, hogy a metrika nem hatékony szintsorrendezés esetén is nagyon közel kerülhet a minimális értékéhez.

A mérések során használt CTL-kifejezések:

- Egyszerűsített étkező filozófus (Phil) modellnél: $E[eszik_1 = 0\, U\, eszik_2 = 1]$

- Étkező filozófus (DPhil) modellnél:
  $E[\neg(HasLeft_2 > 0 \wedge HasRight_2 > 0)\, U\, (HasLeft_1 > 0 \wedge HasRight_1 > 0)]$
- Réselt gyűrű (SR) modelleknél: $E[B_1 \neq 1 \vee F_1 \neq 1\, U\, G_2 = 1 \wedge A_2 = 1]$

5.6. táblázat. Lokalitás alapú szintcsere heurisztika mérési eredményei

Modell	Állapottér mérete	P-invariáns heurisztika, átszintezés nélkül	P-invariáns heurisztika, átszintező heurisztikával	összetett heurisztika, átszintezés nélkül	összetett heurisztika, átszintező heurisztikával
Phil-100	$7,92 \cdot 10^{20}$	0,69 s	0,69 s	3,26 s	3,23 s
DPhil-100	$4,96 \cdot 10^{62}$	6,27 s	8,27 s	29,06 s	27,47 s
SR-30	10^{31}	12,88 s	12,56 s	—	—
SR-50	$1,7 \cdot 10^{52}$	48,19 s	47,08 s	—	—

MDD-minimalizáló szintcsere heurisztika eredményei

Az általam javasolt MDD-minimalizáló szintcsere heurisztikát EG kifejezésekkel vizsgáltam, ennek eredményei az 5.7. táblázatban olvashatók. Látható, hogy a leírt esetekben a heurisztika szignifikáns csökkenést okozott a csomópontszámban, amely a kifejezés kiértékeléséhez szükséges időt is lecsökkentette.

A mérések során használt szintezések és kifejezések:

- DFMS-100: $EG(true)$, P-invariáns szintezés
- laBPEL: $EG(true)$, 3 hely/szint
- auctionBPEL: $EG(true)$, 3 hely/szint
- 8 bástya: $EG(true)$, összetett heurisztika

5.7. táblázat. MDD-minimalizáló szintcsere heurisztika mérési eredményei

Modell	Állapottér mérete	átszintezés nélkül		átszintező heurisztikával	
		idő	csomópont	idő	csomópont
DFMS-100	33028	2,68 s	144	1,86 s	62
laBPEL	26062	21,93 s	700	19,31 s	536
auctionBPEL	21775	16,49 s	2881	15,88 s	2476
8 bástya	$1,4 \cdot 10^6$	115,95 s	131216	59,58 s	65837

5.3. Összehasonlítás más eszközökkel

A munkám során elkészült szaturációs modellellenőrzőt összevetettem néhány elterjedt modellellenőrző programmal. Az összehasonlítást az alábbi eszközökkel terveztem végezni: Integrated Net Analyzer (INA) 2.2 [24], UPPAAL 4.0.13 [27], SAL 3.0 [26], NuSMV 2.5.1 [25].

Ezek közül egyik eszköz sem támogatja a .pnml formátumú bemenetet, így a modellek átalakítására volt szükség. Az INA számára szükséges .pnt formátumhoz a *PetriDotNet* keretrendszer már tartalmaz egy export modult, ezzel a szükséges modellek elkészíthetők. A NuSMV és SAL modellek szintén elkészíthetők a Petri-hálók megfelelő, speciális formátumúra hozásával, ehhez az általam készített konvertáló programot használtam. Az

UPPAAL automaták hálózatán képes analízist végezni, ehhez a modelleket egyesével újra kellett tervezni. Ezen kívül az UPPAAL és SAL programok nem támogatják teljeskörűen a CTL-kifejezések kiértékelését: az UPPAAL rendszerben EU operátor nem értékelhető ki, a SAL programmal pedig csak az LTL-re konvertálható CTL-kifejezések vizsgálata lehetséges.

A táblázatokban az egyes eszközök számítással töltött idejét tüntettem fel. Egyes programok külön végeznek állapottér-generálást és modellellenőrzést, ilyenkor a táblázatban a két időadat összege szerepel. Az én megoldásom eredményei a PetriDotNet oszlopban szerepelnek.

5.3.1. EF operátort tartalmazó kifejezések kiértékelése

Az EF operátorokat tartalmazó kifejezések vizsgálatánál a megvalósításomat az INA, NuSMV és UPPAAL programokkal vetettem össze, ennek eredménye az 5.8. táblázatban látható. Az INA program eredményeinél jól látható, hogy mivel gyakorlatilag explicit állapottér-generálást végez, így már 10^6 nagyságrendű állapottereket sem képes hatékonyan felderíteni. Bár ez a közölt mérésekből nem látszik megjegyzendő, hogy az explicit állapottér-generálás miatt a CTL-kifejezések kiértékelése nagyon gyors.

A NuSMV program mérési eredményeinél is látható, hogy bár jelentősen gyorsabb az INA-nál, mégis a futásidő az állapottér-méretével arányos. Így olyan nagy állapotterű, erősen aszinkron modellekre nem hatékony, mint az a DPhil modell méréseiből látható.

Az UPPAAL program mért időadatai az általam írt programmal összemérhetőek. Ennek két oka is lehet. Egyrészt az UPPAAL is alkalmaz bizonyos redukciós technológiákat, azonban ezek nem Petri-hálóhoz adaptáltak, hanem az UPPAAL által kezelt automatákhoz. Másrészt az UPPAAL modellellenőrzésnél egy jelentősen egyszerűbb feladatot old meg, mint a PetriDotNet beépülő modulom. Az én megoldásomban az összes olyan kezdőállapotot feltérképezem, melyre az adott CTL-kifejezés igaz lesz, ezzel szemben az UPPAAL a feltérképezést befejezi, amikor a tényleges kezdőállapotot megtalálta. Emiatt gyakorlatilag a modell méretétől független tud lenni a kifejezés kiértékelése réselt gyűrű modell esetén. Azonban ezzel elveszíti a kifejezések egymásba ágyazhatóságának és ezáltal a bonyolultabb specifikációs követelmények ellenőrzésének lehetőségét, ezzel szemben a beépülő modulom az egymásba ágyazott CTL-kifejezések vizsgálatára is lehetőséget ad.

5.8. táblázat. EF operátort tartalmazó kifejezések mérési eredményei

Modell	Állapottér mérete	INA	NuSMV	UPPAAL	PetriDotNet
DPhil-5	13640	1,3 s	$<$ 0,5 s	0,5 s	0,01 s
DPhil-10	$1,8 \cdot 10^6$	$>$ 1800 s	$<$ 0,5 s	0,5 s	0,01 s
DPhil-100	10^{62}	$>$ 1800 s	4,8 s	1,3 s	0,12 s
DPhil-500	10^{313}	$>$ 1800 s	98,7 s	2,8 s	0,92 s
DPhil-1000	10^{629}	$>$ 1800 s	$>$ 1800 s	4,5 s	2,65 s
SR-5	$5,4 \cdot 10^5$	45,6 s	$<$ 0,5 s	1,6 s	0,24 s
SR-30	10^{31}	$>$ 1800 s	1,8 s	1,6 s	0,93 s
SR-50	10^{52}	$>$ 1800 s	4,3 s	1,6 s	3,39 s

A mérésekhez használt CTL-kifejezések a következők voltak:

- Étkező filozófus (DPhil) modelleknél:
  $EF((HasLeft_2 > 0 \wedge Fork_1 > 0) \vee (HasRight_2 > 0 \wedge Fork_2 > 0))$
- Réselt gyűrű (SR) modelleknél: $EF(A_1 > 0 \vee B_1 > 0)$

5.3.2. EU operátort tartalmazó kifejezések kiértékelése

Az EU operátorok mérésekor az összehasonlításokat az INA programmal végeztem, mivel az UPPAAL nem támogatja ezen kifejezések kiértékelését. Az 5.9. táblázatban látható mérési eredményekből jól látható, hogy az INA már relatíve kis modellek esetén sem képes a kifejezések kiértékeléséhez szükséges állapotteret legenerálni. Számos modell esetén az általam írt program egy másodpercen belüli futással képes kiértékelni olyan kifejezéseket, melyeket az INA 1800 másodperc alatt sem volt képes.

A mérésekhez használt CTL-kifejezések a következők voltak:

- Egyszerűsített étkező filozófus (Phil) modelleknél: $E[eszik_1 = 0\, U\, eszik_2 = 1]$
- Étkező filozófus (DPhil) modelleknél:
  $E[\neg(HasLeft_2 > 0 \wedge HasRight_2 > 0)\, U\, (HasLeft_1 > 0 \wedge HasRight_1 > 0)]$
- Rugalmas gyártórendszer (FMS-N) modelleknél:
  $E[M1 > 0\, U\, (P1s = P2s = P3s = N)]$
- Réselt gyűrű (SR) modelleknél: $E[B_1 \neq 1 \vee F_1 \neq 1\, U\, G_2 = 1 \wedge A_2 = 1]$

5.9. táblázat. EU operátort tartalmazó kifejezések mérési eredményei

Modell	Állapottér mérete	INA	PetriDotNet
Phil-20	15127	33 s	0,05 s
Phil-30	$1,9 \cdot 10^6$	$>$ 1800 s	0,06 s
Phil-100	$7,9 \cdot 10^{20}$	$>$ 1800 s	0,64 s
Dphil-5	13640	1,3 s	0,37 s
Dphil-10	$1,8 \cdot 10^6$	$>$ 1800 s	0,06 s
Dphil-15	$2,5 \cdot 10^9$	$>$ 1800 s	0,11 s
FMS-2	3444	4,3 s	0,11 s
FMS-3	48590	115 s	0,12 s
FMS-5	$2,9 \cdot 10^6$	$>$ 1800 s	0,93 s
SR-5	53856	45,6 s	0,26 s

5.4. Extrém nagy modellek állapottér-generálása

Az MDD-alapú tárolás hatékonysága figyelhető meg az extrém nagy modelleknél. Óriási méretű állapotterek is gyorsan generálhatók, illetve tárolhatók a memóriában, ez szintén alátámasztja az MDD és szaturáció alapú megoldások létjogosultságát. Az alábbi mérések ezt mutatják be. A programommal sikerült legenerálni a 100.000 filozófus deadlockos modelljének állapotterét, mely körülbelül 10^{62690} állapotot tartalmaz. Ez jelentősen nagyobb állapottér felderítését jelenti, mint a [8]-ban elért maximális, 10^{6269} méretű állapottér. A mért eredmények az 5.10. táblázatban találhatók. Ahhoz azonban, hogy ekkora modellek ellenőrzése is lehetséges legyen különösen fontos, hogy a szinthozzárendelés és a szintek sorrendezése is közel optimális legyen. Ezért az itt bemutatott méréseknél a szintezést manuálisan végeztem.

5.10. táblázat. Extrém nagy modellek állapottér-generálási eredményei

Modell	állapottér	idő	csomópontok száma	szintek száma
DPhil-10k	10^{6269}	4,76 s	49999	10001
DPhil-20k	10^{12538}	12,26 s	99999	20001
Phil-100k	10^{20898}	47,60 s	499997	100001
DPhil-50k	10^{31345}	39,45 s	249999	50001
DPhil-100k	10^{62690}	149,71 s	499999	100001

6. fejezet

Összefoglalás

6.1. Elért eredmények összegzése

A célom egy jól használható, hatékony modellellenőrző fejlesztése volt a legújabb kutatási eredmények felhasználásával. Munkám során elkészítettem a PetriDotNet keretrendszerhez egy olyan modellellenőrző modult, amely tartalmazza és gyakorlatban alkalmazza mindezeket az új algoritmusokat. Ezeket kiegészítettem saját implementációs fejlesztésekkel, melyek további gyorsításokat eredményeztek és heurisztikákkal, melyek gyorsító hatásuk mellett lehetővé teszik a szélesebb körű felhasználást. Implementáltam olyan hatékonyságnövelő algoritmusokat, melyekeket ebben a kontextusban korábban nem publikáltak. A munkám eredménye egy gyors modellellenőrző program lett, mely a jelenleg elterjedt eszközökkel összevetve is kiváló eredményeket ért el, így a kitűzött célt sikerült teljesítenem. Többek közt a grafikus felhasználói felületnek, a kifejezések CTL szintaxisú megadásának és a heurisztikáknak köszönhetően a nagy teljesítmény mellett a felhasználhatóbarát működést is sikerült megvalósítani. Az elkészült modul használatra kész és elérhető a keretrendszer honlapján [23].

Fejlesztéseimmel a *PetriDotNet* keretrendszer használhatósága megnőtt, teljeskörűen felhasználhatóvá vált mind az oktatásban, mind az iparban modellezésre és modellellenőrzésre. A programot jelenleg is használják a BME mérnök-informatikus mesterképzésén a Formális módszerek tárgyban és további oktatási felhasználása is várható.

6.2. Továbbfejlesztési lehetőségek

A *PetriDotNet* keretrendszer és az MDD-alapú analízis modul fejlesztése várhatóan tovább folytatódik a közeljövőben. A legfontosabb további irányok a következők:

- Terveink szerint a modult kiegészítjük korlátos modellellenőrzővel annak érdekében, hogy a kifejezések kiértékelésére ne csak egy igaz vagy hamis választ adjon az ellenőrző, hanem egy példát vagy ellenpéldát is.
- A heurisztikákat gátló Martinez–Silva invariánsszámító algoritmust összetettebb, gyorsabb működésű algoritmusra szeretnénk cserélni.
- További dekomponáló és sorrendező heurisztikákat szeretnénk keresni, melyekkel még hatékonyabb lehet az automatikus, szakértő beavatkozást nem igénylő modellellenőrzés.
- Mivel a szaturációs algoritmus nem kizárólag egyszerű Petri-hálókra alkalmazható, tervezzük a vizsgálható problémaosztály kiterjesztését. Lehetséges fejlesztési irányok: színezett Petri-hálók, sztochasztikus, időzített Petri-hálók, Petri-hálóknál magasabb szintű modellek vizsgálata.

A. függelék

Jelölések jegyzéke

A dolgozatomban számos jelölést használok. A könnyebb érthetőség kedvéért itt összefoglalom a fontosabbak jelentéseit.

Jelölés	Jelentés
$M(p)$	p hely tokenszáma
$\mathbf{m}$ vagy $\mathbf{i}$	A Petri-háló globális állapota.
$(i_K, \ldots, i_1)$	A Petri-háló globális állapota $i_K, \ldots, i_1$ lokális állapotokkal kifejezve.
$\mathcal{E}$	A modell összes eseményének halmaza.
α	A modell egyik eseményének szokásos jelölése.
$Top(\alpha)$	Az α esemény által befolyásolt legfelső szint száma.
$Bot(\alpha)$	Az α esemény által befolyásolt legalsó szint száma.
$\mathcal{E}_k$	A modell olyan α eseményének halmaza, melyekre $Top(\alpha) = k$.
$\mathcal{N}(\mathbf{m})$	Next-state függvény, mely megadja az $\mathbf{m}$ állapotból egyetlen tüzeléssel elérhető állapotok halmazát.
$\mathcal{N}_t(\mathbf{m})$	Next-state függvény, mely megadja az $\mathbf{m}$ állapotból t tranzíció (esemény) egyetlen tüzelésével elérhető állapotok halmazát.
$\mathcal{N}_{\mathcal{A}}(\mathbf{m})$	Next-state függvény, mely megadja az $\mathbf{m}$ állapotból egyetlen $\mathcal{A}$ halmazbeli tüzeléssel elérhető állapotok halmazát.
$\mathcal{N}_{\mathcal{A}}(\mathcal{X})$	Next-state függvény, mely megadja az $\mathcal{X}$ halmazba tartozó állapotokból egyetlen $\mathcal{A}$ halmazbeli tüzeléssel elérhető állapotok halmazát.
$\mathcal{N}^*(\mathbf{m})$	Az $\mathbf{m}$ állapotból elérhető állapotok halmaza.
$v[i]$	A v MDD-csomópont i. indexű gyereke.
$\mathcal{S}$	A Petri-háló elérhető állapotainak halmaza.
$\mathcal{S}_k$	A Petri-háló k. szinten lévő elérhető lokális állapotainak halmaza.
K	Nemterminális csomópontokat tartalmazó szintek száma egy MDD-ben.
$\mathbf{N}_{k,\alpha}$	A k. szinten α eseményhez tartozó Kroenecker-mátrix.
$\mathbf{I}$	Identitásmátrix.

B. függelék

A szaturációs állapottér-generáló algoritmusok pszeudókódjai

Algoritmus 12 GenerateStateSpace

Input: **s** : kezdőállapot, $\mathcal{N}$: állapot $\rightarrow$ állapothalmaz (next-state függvény)
Output: elérhető állapotok halmaza

```
1.  Np, Nr : csomópont
2.  k : egész
3.  Np = terminális egy
4.  for k = 1 to K do
5.     Confirm(k, 0)
6.     Nr = NewNode(k)
7.     Nr[0] = Np
8.     Saturate(Nr)
9.     CheckIn(Nr)
10.    Np = Nr
11. end for
12. return  Nr
```

Algoritmus 13 Saturate

Input: *Np* : csomópont
Output: logikai

```
1.  e : esemény
2.  chng : logikai
3.  chng = true
4.  while chng = true do
5.     chng = false
6.     for each α ∈ E_k do
7.        chng = SatFire(α, Np) ∨ chng
8.     end for
9.  end while
10. return  chng
```

Algoritmus 14 SatFire

Input: α : esemény, Np : csomópont
Output: logikai

```
1.  Nf, Nu : csomópont
2.  i, j : lokális állapot
3.  L : lokális állapotok halmaza
4.  chng : logikai
5.  chng = false
6.  L := {i ∈ S_k : Np[i] ≠ 0, N_{k,α}[i, ·] ≠ 0}
7.  while L ≠ ∅ do
8.     i = tetszőleges elem L-ből
9.     i eltávolítása L-ből
10.    Nf = SatRecFire(α, Np[i])
11.    if Nf nem nullcsomópont then
12.       for each j-re ahol N_{k,α}[i, j] = 1 do
13.          Nu = Union(Nf, Np[j])
14.          if Nu ≠ Np[j] then
15.             if j ∉ S_k then
16.                Confirm(k,j)
17.             end if
18.             Np[j] = Nu
19.             chng = true
20.             if N_{k,α}[j, ·] ≠ 0 then
21.                L = L ∪ {j}
22.             end if
23.          end if
24.       end for
25.    end if
26. end while
27. return chng
```

Algoritmus 15 SatRecFire

Input: α : esemény, Np : csomópont
Output: csomópont

```
1. Nf, Nu, Ns : csomópont
2. i, j : lokális állapot
3. k : egész
4. L : lokális állapotok halmaza
5. chng : logikai
6. k = level(Np)
7. if k < Bot(α) then
8.    return Np
9. end if
10. if CachedFire(α, Np, out Ns) then
11.    return Ns
12. end if
13. Ns = NewNode(lk)
14. chng = false
15. L := {i ∈ S_k : Np[i] ≠ 0, N_{k,α}[i, ·] ≠ 0}
16. while L ≠ ∅ do
17.    i = tetszőleges elem L-ből
18.    i eltávolítása L-ből
19.    Nf = SatRecFire(α, Np[i])
20.    if Nf nem nullcsomópont then
21.       for each j-re ahol N_{k,α}[i, j] = 1 do
22.          Nu = Union(Nf, Np[j])
23.          if Nu ≠ Np[j] then
24.             if j ∉ S_k then
25.                Confirm(k,j)
26.             end if
27.             Np[j] = Nu
28.             chng = true
29.          end if
30.       end for
31.    end if
32. end while
33. if chng = true then
34.    Sautrate(Ns)
35. end if
36. CheckIn(Ns)
37. PutInCacheFire(α, Np, Ns)
38. return Ns
```

C. függelék

A mérésekhez használt modellek

A következőkben bemutatom azokat a modelleket, melyeket a mérések során használtam.

C.1. Étkező filozófusok

Az étkező filozófusok problémát már tárgyaltam a 2. példában. Mint azt ott is említettem, az ott részletezett modell egy egyszerűsített modell. A bővebb modellben miután az egyes filozófusok arra az elhatározásra jutnak, hogy enni kezdenek, külön-külön veszik fel a bal és jobb oldali pálcikákat és egészen addig nem teszik le, míg az evést be nem fejezték. Könnyen látható, hogy ebben a modellben előállhat olyan állapot, amelyben egyik tranzíció sem tüzelhet, azaz holtpont (deadlock) alakul ki. Ha például minden filozófus enni akar és mindegyik már felvette a bal oldali pálcikáját, akkor az asztalon nincsen már felvehető pálcika, egyik filozófusnál sincs két pálcika, azaz senki sem tud enni, így senki nem fog pálcikát az asztalra visszahelyezni [12].

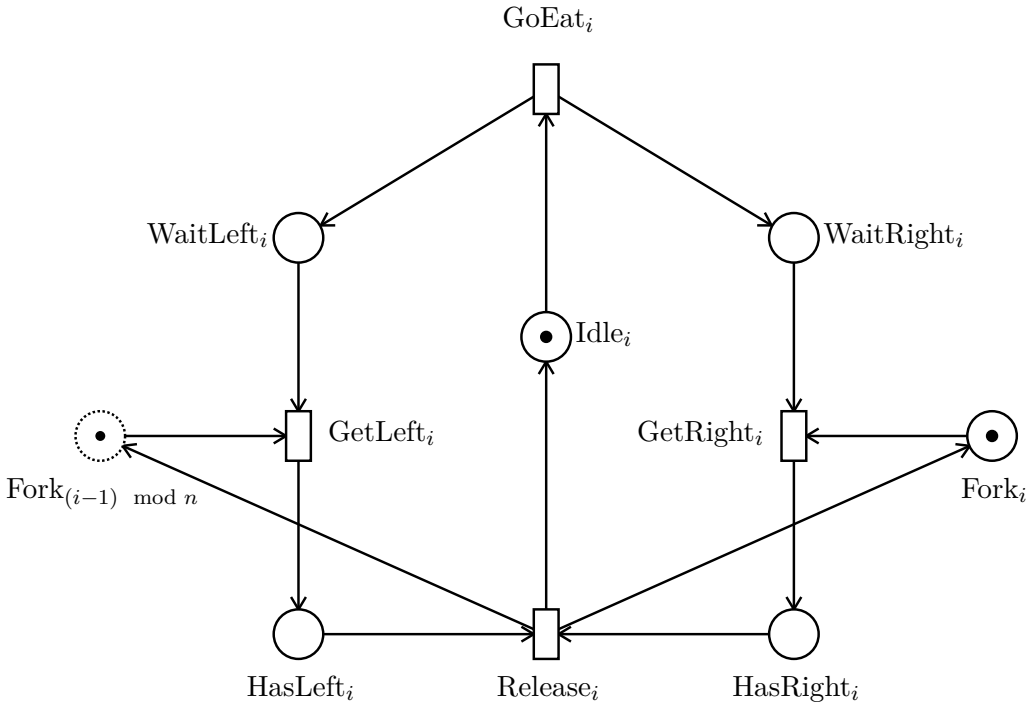
A deadlockos étkező filozófusok probléma i. filozófusra vonatkozó Petri-hálóját mutatja a C.1. ábra. Ilyen részmodellekből tetszőleges számú összakapcsolható, így leírható az n étkező filozófus problémája. Az n étkező filozófus deadlockos modelljére röviden DPhil-n néven hivatkozom.

C.2. Réselt gyűrű

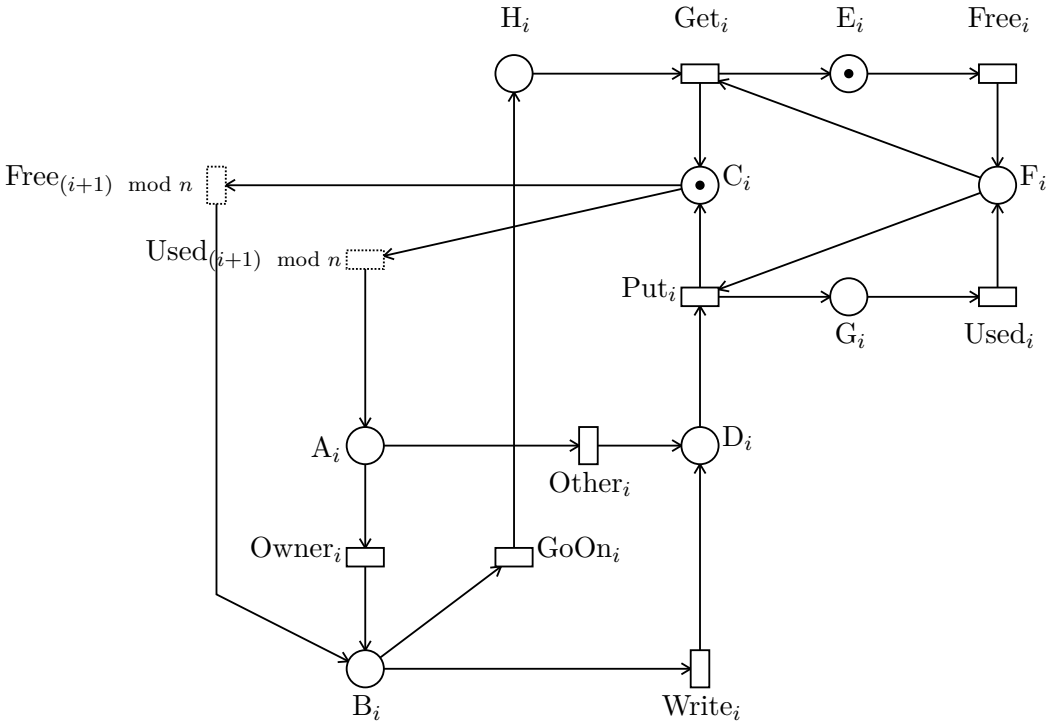
A réselt gyűrű (slotted ring) protokoll modell egy n csomópontból álló hálózatot modellez, melyben a csomópontok gyűrűre vannak fűzve és fix méretű keretekben kommunikálnak. Ebben jelzik, hogy a keret üres-e vagy sem. Ha egy csomópont kommunikálni próbál, vár egy üres keretre, majd ebben helyezi el az információt. Az i. csomópont modellje látható a C.2. ábrán [16]. Ha az $i+1$. csomópont felől üres keret érkezik, a $\text{Free}_{(i+1) \bmod n}$ tranzíció tüzel, teli keret érkezésekor pedig az $\text{Used}_{(i+1) \bmod n}$. Az egyes helyek intuitív jelentése a következő:

A_i	Az i. csomópontba foglalt keret érkezett.
B_i	Az i. csomópontba üres keret érkezett.
C_i	Az i. csomópont keret fogadására készen áll.
D_i	Az i. csomópont által küldendő üres keret továbbítási feldolgozásra készen áll.
E_i	Az i. csomópont üres keret továbbítására készen áll.
F_i	Az i. csomópont keret-továbbítására készen áll.
G_i	Az i. csomópont foglalt keret továbbítására készen áll.
H_i	Az i. csomópont által küldendő foglalt keret továbbítási feldolgozásra készen áll.

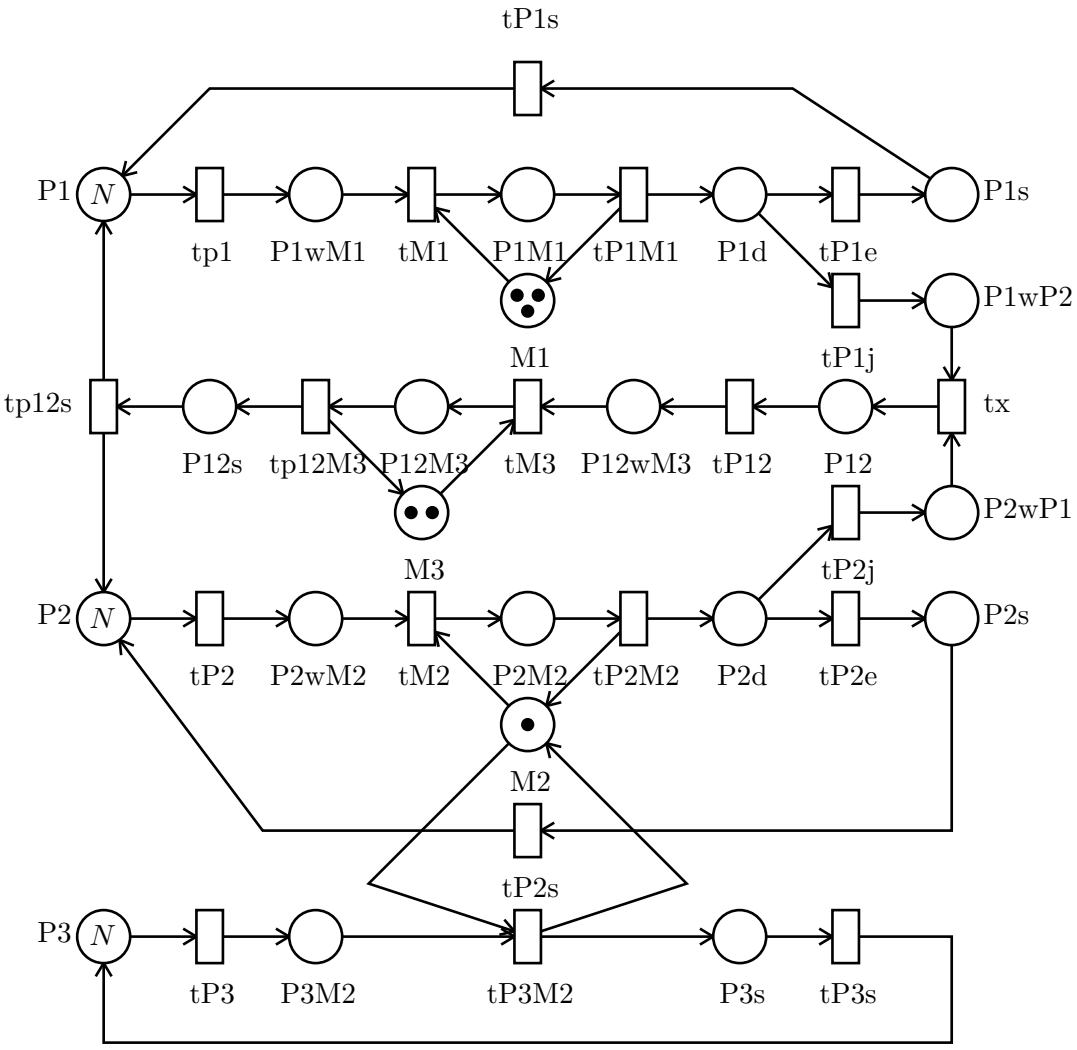
Az n csomópontból álló hálozat réselt gyűrű protokoll modelljére röviden SR-n néven hivatkozom.



C.1. ábra. Deadlockos étkező filozófusok modell



C.2. ábra. Egy réselt gyűrű (slotted ring) hálózat i. csomópontjának modellje



C.3. ábra. Rugalmas gyártórendszer (FMS) modellje

C.3. Rugalmas gyártórendszer

Egy három futószalagból álló gyártórendszert reprezentál az ún. rugalmas gyártórendszer (flexible manufacturing system, FMS) modellje, mely a C.3. ábrán látható. Hely hiányában a modell részletes jelentésével itt nem foglalkozom, a részletes leírás (beleértve az egyes tranzíciók és helyek jelentésének leírását) megtalálható a [10]-ben.

A modell paramétere N, amely a P1, P2, P3 helyek tokenszámát jelöli. Az $M(P1) = M(P2) = M(P3) = N$ tokenszámú modellre röviden FMS-N néven hivatkozom.

C.4. További modellek

A mérések során további modelleket is használtunk, azonban ezek részletes ismertetésére helyhiány miatt nincs lehetőség. A modellek letölthetők a PetriDotNet keretrendszer honlapjáról [23].

Az *auctionBPEL* és *laBPEL* modellek BPEL (Business Process Execution Language) nyelven készült modellek Petri-hálókká konvertált változatai. Mindkettő egy-egy üzleti folyamat működését írja le, így valós életbeli problémák modelljének tekinthető.

A *DFMS* modell egy speciális, a rugalmas gyártórendszeren alapuló modell, mely a kezdeti tokenszámokkal jól skálázható.

A *8 bástya* modellben egy szabályos, 8×8 mezőből álló sakktáblára kell elhelyezni bástyákat úgy, hogy azok ne kerüljenek egymással ütésbe.

Ábrák jegyzéke

Irodalomjegyzék

[1] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[2] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. pages 49–58. North-Holland, 1991.

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

[4] Gianfranco Ciardo. Data Representation and Efficient Solution: A Decision Diagram Approach, 2007.

[5] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: an efficient iteration strategy for symbolic state space generation. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 2031*, pages 328–342. Springer-Verlag, 2001.

[6] Gianfranco Ciardo, Gerald Lüttgen, and Andy Yu. Improving static variable orders via invariants. In Jetty Kleijn and Alex Yakovlev, editors, *Petri Nets and Other Models of Concurrency – ICATPN 2007*, volume 4546 of *Lecture Notes in Computer Science*, pages 83–103. Springer Berlin / Heidelberg, 2007.

[7] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. Saturation unbound. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 379–393. Springer, 2003.

[8] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. The saturation algorithm for symbolic state-space exploration. *Int. J. Softw. Tools Technol. Transf.*, 8(1):4–25, 2006.

[9] Gianfranco Ciardo and Radu Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. In *Computer Aided Verification (CAV'03), LNCS 2725*, pages 40–53. Springer-Verlag, 2003.

[10] Gianfranco Ciardo and Kishor S. Trivedi. A decomposition approach for stochastic reward net models. *Perform. Eval.*, 18(1):37–59, 1993.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[12] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. pages 198–227, 2002.

[13] Kondorosi Károly and Várkonyiné Kóczy Annamária, editors. *Operációs rendszerek mérnöki megközelítésben*. Panem Kiadó, 2000.

[14] D. Michael Miller and Rolf Drechsler. On the construction of multiple-valued decision diagrams. *Multiple-Valued Logic, IEEE International Symposium on*, 0:245, 2002.

[15] D.M. Miller and R. Drechsler. Implementing a multiple-valued decision diagram package. pages 52 –57, may. 1998.

[16] Andrew S. Miner and Gianfranco Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Proceedings of the 20th International Conference on Application and Theory of Petri Nets*, pages 6–25, London, UK, 1999. Springer-Verlag.

[17] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[18] Pataricza András, editor. *Formális módszerek az informatikában.* Typotex, 2 edition, 2005.

[19] Oriol Roig, Jordi Cortadella, and Enric Pastor. Verification of asynchronous circuits by bdd-based model checking of petri nets. In Giorgio De Michelis and Michel Diaz, editors, *Application and Theory of Petri Nets 1995*, volume 935 of *Lecture Notes in Computer Science*, pages 374–391. Springer Berlin / Heidelberg, 1995.

[20] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[21] Vörös András. Döntési diagramok alkalmazása on-line diagnosztikában (diplomaterv), 2009.

[22] Michael Weber and Ekkart Kindler. The Petri Net Markup Language. pages 124–144. 2003.

[23] A *PetriDotNet* keretrendszer honlapja.
http://www.inf.mit.bme.hu/research/tools/petridotnet/.

[24] Integrated Net Analyzer:
http://www2.informatik.hu-berlin.de/lehrstuehle/automaten/ina/.

[25] NuSMV: http://nusmv.fbk.eu/.

[26] SAL: http://sal.csl.sri.com/.

[27] UPPAAL: http://www.uppaal.com/.