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

Komplex rendszerek modellezése és verifikációja

TDK-dolgozat

Készítette:
Darvas Dániel
Jámbor Attila

Konzulensek:
dr. Bartha Tamás
Vörös András

2011.

Tartalomjegyzék

1. fejezet

Bevezető és motiváció

A rendszertervezés során a szoftver- és hardverrendszerek helyességének ellenőrzése, azaz verifikációja egyre nagyobb szerepet kap. A verifikációval kapcsolatban alapvető elvárás, hogy teljes körű és matematikai precizitású legyen, ami formális módszerek alkalmazását igényli. Komplex rendszerek esetén a formális verifikáció igen számításigényes probléma. Bár a rendelkezésre álló számítási kapacitás növekszik, még mindig komoly kihívást jelent a nagyméretű, összetett architektúrák ellenőrzése.

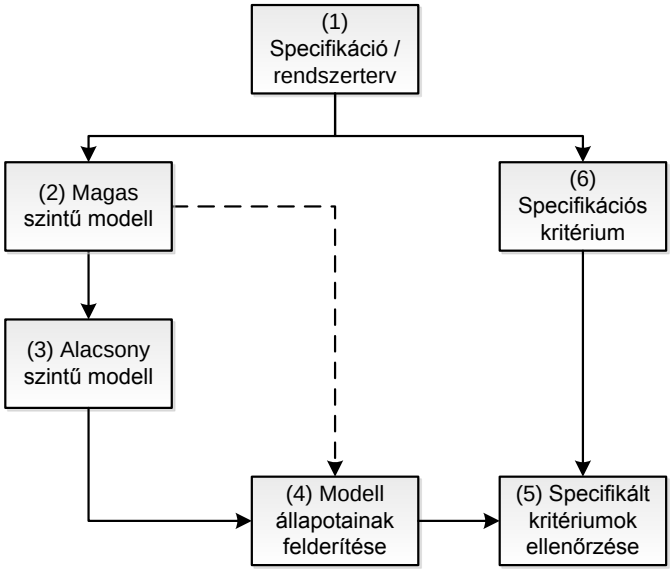
A verifikáció folyamatában gyakran alkalmazott módszer a modellellenőrzés. Ennek során megtervezzük a rendszer egy modelljét – esetünkben Petri-hálók segítségével –, majd felderítjük annak állapotterét. Utána az állapottérben a specifikációhoz kötődő kritériumok teljesülését ellenőrizzük – esetünkben temporális logikai kifejezések segítségével megfogalmazott követelmények formájában – matematikai módszerek alkalmazásával.

Korábbi munkáinkban [13, 16] bemutattuk, hogy aszinkron rendszerek esetén az ún. szaturációs algoritmus használatával akár nagy méretű állapotterek is hatékonyan felderíthetőek és kompakt formában tárolhatóak. Összetett rendszerek modellellenőrzése során azonban felmerülnek olyan problémák, amelyekre az eddigi algoritmusaink nem nyújtottak megfelelő megoldást. Ilyenek többek között a nagy komplexitású és/vagy végtelen állapotterű, vagy bonyolultabb adatszerkezetet tartalmazó modellek analízise. Emellett problémát jelent az is, hogy bizonyos rendszerméret felett az egyszerű Petri-hálók már nem alkalmasak a rendszer hatékony és átlátható modellezésére.

Dolgozatunkban olyan módszereket mutatunk be, amelyek a fent ismertetett problémákra nyújtanak megoldást.

A korábbi munkáink során alkalmazott, jelen dolgozatban kiegészített verifikációs folyamat áttekintése látható az 1.1. ábrán. A korábbi megoldásainkban a vizsgálandó rendszerterv (1) alapján egy alacsony szintű modellt (3) kellett készíteni (esetünkben egyszerű Petri-hálót). Természetesen köztes fázisban a vizsgálatot végző személy készíthetett magas szintű modellt (2), azonban az ellenőrzéshez ezt alacsony szintűvé kellett átalakítania. Ennek a modellnek már fel tudtuk deríteni a lehetséges állapotait (4), majd ezen információ birtokában a specifikációban szereplő követelmények (6) ellenőrizhetők voltak (5). Ezt a folyamatot jelen munkánk során az alábbiakkal egészítettük ki:

- Kifejlesztettünk egy új *korlátos állapottér-felderítő és specifikációs kritériumokat ellenőrző* algoritmust annak érdekében, hogy bizonyos kritériumokat nagyon nagy, vagy akár végtelen lehetséges állapottal rendelkező modelleken is vizsgálhassunk. Így nem szükséges a specifikált kritériumok ellenőrzése előtt a modell összes állapotának felderítése, elegendő a kritériumok vizsgálatához szükséges részhalmaz bejárása. Ennek a megközelítésnek az alkalmazását illusztrálja az alábbi példa:



1.1. ábra. Magas szintű áttekintés a verifikációs folyamatról

1. példa. *Tekintsük az 1.2. ábrán látható intuitív modellt, amelyben egy küldő egy várakozási soron keresztül üzeneteket küld egy fogadónak. Amennyiben azt kívánjuk vizsgálni, hogy lehetséges-e legalább egy üzenet célbajuttatása, akkor nem szükséges azokkal az esetekkel foglalkoznunk a vizsgálat során, amikor egynél több üzenet van a sorban, mert az átvitel lehetségességét ez nem befolyásolja. Azonban ha nem korlátos módszereket használunk, az összes lehetséges állapot feltérképezendő a vizsgálat előtt. A várakozási sor mérete viszont nagyon nagy is lehet, illetve bizonyos esetekben akár végtelennek is tekinthetjük, ilyenkor pedig a modell a korábbi módszereinkkel nem vizsgálható.*

1.2. ábra. Motivációs példa a korlátos modellellenőrzés használatára

- Megvalósítottunk ún. *vezérelt szaturációs* algoritmusokat, amelyek segítségével – a későbbi fejezetekben kifejtett okoknál fogva – jelentős gyorsítást értünk el bizonyos *specifikált kritériumok ellenőrzése* terén, illetve nagy mértékben tudtuk javítani az előző pontban említett korlátos modellellenőrzés hatékonyságát.

- Elsőként dolgoztunk ki módszertant a színezett Petri-hálók szaturációs ellenőrzésére, illetve kifejlesztettünk különböző megoldásokat a vizsgálat hatékonyabbá tételére. (Színezett Petri-hálók használata esetén adatstruktúrákat is felhasználhatunk a modellben, így kompaktabb leírást adhatunk.) Munkánkkal lehetőség nyílt színezett hálók közvetlen vizsgálatára, így közbenső átalakítási lépés nélkül egy magas szintű modellen végzünk állapotfelderítést és kritériumellenőrzést (az 1.1. ábrán szaggatott nyíllal jelölve).

  A kompaktabb modellek használatának motivációját mutatja a 2. példa.

2. példa. *Vizsgáljuk a híres étkező filozófusok probléma [14] modelljét! Egy lehetséges Petri-hálós modellben minden filozófushoz rendelünk egy „eszik”, egy „gondol-*

kozik" és egy „villa" helyet. (A Petri-hálók definícióját a 2.1. fejezetben olvashatja.) Így n étkező filozófus leírásához $3n$ *helyre van szükség. Színezett hálók alkalmazása esetén az egyes filozófusok az adatszerkezetekkel leírhatók, így a modell konstans* 3 *színezett helyből áll, mivel csak a működés strukturális leírása szükséges. Ráadásul a modell színezett hálók esetén paraméterezhető, azaz a filozófusok számának megváltozása strukturális változást nem okoz, csak az adatszerkezeteket kell módosítani.*

A dolgozat felépítése a következő: a 2. fejezetben bemutatjuk a szaturációs modellellenőrzéshez szükséges háttérismereteket. A 3. fejezetben ismertetjük a szaturációs algoritmusainkban használt hatékony állapotátmenet-leképezéseket, amely elengedhetetlen a színezett Petri-hálók kezeléséhez. Ezek után a 4. fejezetben ismertetjük a szaturáció alapú korlátos modellellenőrzést, amelynek segítségével lehetőség nyílik bizonyos követelmények ellenőrzésére a teljes állapottér felderítése nélkül. Az 5. fejezetben leírjuk a vezérelt szaturáció alapú megvalósításunkat, illetve azt is, hogy ennek segítségével hogyan hoztuk létre az első igazán hatékony, szaturációt használó korlátos modellellenőrző algoritmust. Ezt követően a 6. fejezetben bemutatjuk a szaturációs algoritmusok kiterjesztését a kompaktabb leírást biztosító színezett Petri-hálókra és bemutatunk egy új algoritmust, amely segítségével színezett Petri-hálós modellek is sok esetben hatékonyan vizsgálhatók. Végül a 7. fejezetben leírjuk a vizsgálataink eredményeit, majd a 8. fejezetben összefoglaljuk munkánkat.

2. fejezet

Háttérismeretek

E fejezetben bemutatjuk a vizsgálandó modellek leírására használt Petri-hálók főbb tulajdonságait (2.1) valamint ezek összetett adatszerkezeteket tartalmazó általánosítását, a színezett Petri-hálókat (2.2), valamint a megvalósításunkban használt többértékű és élcimkézett döntési diagramokat (2.4). Ismertetjük továbbá a modellellenőrzés alapjait (2.5) és az utóbbi időkben ehhez használt főbb megközelítéseket (2.6). Emellett röviden kitérünk a munkánk során felhasznált és kibővített *PetriDotNet* keretrendszer rövid ismertetésére is (2.7).

2.1. Egyszerű Petri-hálók

A Petri-háló a rendszermodellezés és rendszeranalízis egyik elterjedt modellezési eszköze. Egyidejűleg grafikus és matematikai reprezentációt is definiál, így bizonyos méretig könnyen kezelhető, jól áttekinthető és vizsgálható. A Petri-hálók fő alkalmazási területe a konkurens aszinkron elosztott rendszerek modellezése [21].

2.1.1. Alapstruktúra

Az egyszerű Petri-háló egy irányított, súlyozott, páros gráf, amelyben 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.

Ezek alapján a Petri-háló formálisan egy olyan $PN = (P, T, E, W, M_0)$ struktúra [18], ahol:

$P = \{p_1, p_2, \ldots, p_\pi\}$ a helyek véges halmaza,
$T = \{t_1, t_2, \ldots, t_\tau\}$ 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 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 $t\bullet = \{p|(t,p) \in E\}$.

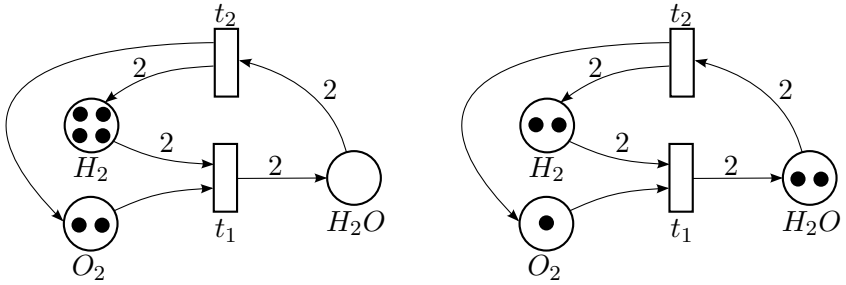
A Petri-háló dinamikus viselkedését az egyes állapotokban értelmezett *tüzelések* határozzák meg. A tüzelések szabályai a következők:

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.

3. példa. *A 2.1. ábrán egy Petri-háló látható, amely egy egyszerű kémiai folyamatot modellez [13]. 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). ábrán látható 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

2.1.2. Tiltó élek

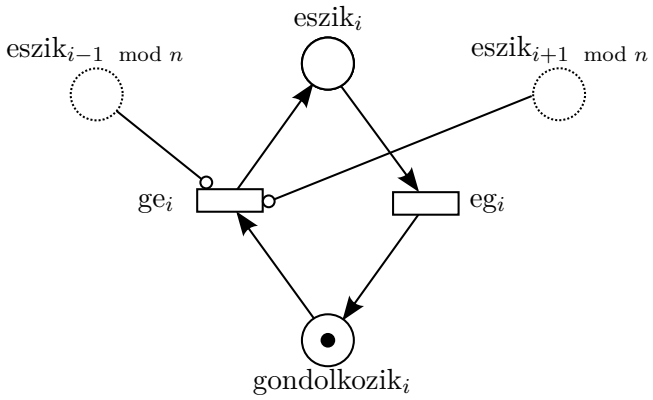
A Petri-hálók gyakran használt kiegészítései a *tiltó élek* (inhibitor arcs). A tiltó élekkel kifejezhető, hogy bizonyos feltételek teljesülésekor egy adott működés ne hajtódjon végre. A tiltó élek halmaza egy $I \subseteq (P \times T)$ halmaz. A szokványos élekhez hasonlóan a tiltó élekhez is rendelhetők 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.

A tiltó élek bevezetésével a Petri-háló kifejezőereje megnő, viszont számos analízis módszer ezekre a hálókra már nem (vagy csak korlátozott mértékben) használható [21].

4. példa. *A jól ismert étkező filozófusok probléma megfogalmazható egyszerűen 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.*

Ezt reprezentálja a 2.2. ábra, amely a filozófusok modelljének i. filozófusra vonatkozó részét mutatja be. Ahogyan az ábrán is látható, a tiltó élek végeire nem nyilat, hanem üres kört rajzolunk.



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

2.2. Színezett Petri-hálók

A színezett Petri-hálók az egyszerű Petri-hálók kiterjesztése gazdagabb adatstruktúrák segítségével. Így lehetőség nyílik a modellek kompaktabb, átláthatóbb formában történő leírására.

A színezett Petri-háló formálisan egy $CPN = (P, T, E, \Sigma, C, G, AE, M_0)$ struktúra [21], ahol:

$P = \{p_1, p_2, \ldots, p_\pi\}$ a helyek véges halmaza,
$T = \{t_1, t_2, \ldots, t_\tau\}$ a tranzíciók véges halmaza,
$E \subseteq (P \times T) \cup (T \times P)$ az élek halmaza,
$\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_\kappa\}$ a típusok (színosztályok) véges halmaza,
$C : P \to \Sigma$ színfüggvény, amely megadja az egyes helyeken érvényesnek tekintett tokenek típusát (színosztályát),
$G : T \to GR$ az egyes tranzíciókhoz tartozó g őrfeltételek halmaza,
$AE : E \to AEX$ az egyes élekhez tartozó ae élkifejezések halmaza,
$M_0 : P \to TK$ a kezdőállapotban az egyes helyeken lévő tk token-multihalmazok halmaza.

Az egyes *színosztályok* vagy típusok tokenek értékkészletét definiáló halmazok. Ezekkel adható meg, hogy egy p helyen milyen tokenek fordulhatnak elő.

Az egyes *élkifejezések* tokenkonstansok és tokenváltozók, valamint ezeken végzett műveletekből álló struktúrák. Ezek azt adják meg, hogy a hozzájuk kötődő tranzíció tüzelése esetén az élhez tartozó helyről – bemenő él esetén – milyen tokeneket kell elvenni vagy – kimenő él esetén – milyen tokeneket kell kirakni.

Az *őrfeltételek* változókon értelmezett logikai kifejezések, amelyek a hozzájuk tartozó tranzíció engedélyezettségét befolyásolják.

Ez a megadás igen általános, mivel nem állít semmit arról, hogy milyen σ, $c \in \Sigma$, $g \in GR$, $a \in AE$ és $k \in TK$ értékek engedhetők meg. Ezek különböző megkötéseivel különböző kompaktságú és kifejezőerejű színezett Petri-hálók definiálhatók. Például az egyes σ típusok általános esetben korlátlan számú, tetszőleges típusú lehetséges tokent tartalmazhatnak.

Egy t tranzíció engedélyezett, ha létezik a t-hez kapcsolódó e élekre írt $AE(e)$ élkifejezésekhez olyan változó-érték hozzárendelés, amely kielégíti a t tranzíció $G(t)$ őrfeltételét, valamint a t tranzíció bemenő éleire írt élkifejezések által meghatározott tokenek a megfelelő bemeneti helyekről elvehetők és hasonlóan a kimenő élekre írt élkifejezések szerinti tokenek a kimeneti helyekre kirakhatók.

Korábbi munkánk során kiegészítettük a PetriDotNet keretrendszert színezett Petri-hálók szerkesztésével. Ennek során a *jólformált színezett Petri-hálók* egy fajtájára esett a

választásunk. Jólformált színezett hálók esetén az egyes σ halmazokra megkötést teszünk: minden σ_i halmaznak véges elemszámúnak kell lennie.

Az általunk definiált és támogatott jólformált színezett hálókban *egyszerű* és *összetett színosztályokat* használunk. Egyszerű színosztály csak véges enumeráció vagy egész számok egy véges részhalmaza lehet. Például egyszerű színosztály lehet $\sigma_1 = \{\alpha, \beta, \gamma\}$, $\sigma_2 = \{2, 3\}$ vagy $\sigma_3 = \{alma, kivi, szilva\}$. Az összetett színosztály olyan színosztály, amely kettő vagy több egyszerű színosztály Descartes-szorzatából keletkezik, pl. $\sigma_4 = \sigma_1 \times \sigma_2 = \{(\alpha, 2), (\alpha, 3), (\beta, 2), (\beta, 3), (\gamma, 2), (\gamma, 3)\}$.

A PetriDotNet keretrendszer által támogatott és jelen dolgozatunk további részében vizsgált hálók élkifejezései és őrfeltételei az alábbi módon definiálhatók. A konstansok és változók egyszerű színosztályok egy-egy elemét jelölik.

```
<élkifejezés> ::= <érték_vektor> | <élkifejezés> '++' <érték_vektor>
<érték_vektor> ::= <érték> | '[' <érték>, <érték>, ... ']'

<őrfelt> ::= <kif> | (<őrfelt>) | <őrfelt> <boolop> <őrfelt>
<boolop> ::= '&&' | '||'
<relációs_jel> ::= '>' | '<' | '>=' | '<=' | '=' | '!=' | '<>'
<kif> ::= <érték> <relációs_jel> <érték>
<érték> ::= <konstans> | <változó> | succ <változó>
```

Tehát egy $G(t)$ őrfeltétel egy `<őrfelt>` kifejezéssel, míg egy $AE(e)$ élkifejezés egy `<élkifejezés>` kifejezéssel adható meg.

2.3. Állapottér

Gyakran felmerülő kérdés, hogy egy adott $\mathbf{m}$ állapotból egy tüzeléssel milyen állapotokba kerülhet a háló. Ezért bevezetjük az ún. *next-state függvényt* ($\mathcal{N}(\mathbf{m})$), amely megadja az $\mathbf{m}$ állapotból egyetlen tüzeléssel elérhető állapotok halmazát. Jelölje $\mathcal{N}_t(\mathbf{m})$ azt a $t \in T$ tranzícióra szorított next-state függvényt, amely azt adja meg, hogy $\mathbf{m}$ állapotból t tranzíció egyetlen tüzelésével milyen állapotok érhetők el. Ennek megfelelően $\mathcal{N}(\mathbf{m}) = \bigcup_{t \in T} \mathcal{N}_t(\mathbf{m})$. Értelmezzük továbbá 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 $\mathcal{N}^*(\mathbf{s_0})$ halmazát a háló *állapotterének* nevezzük.

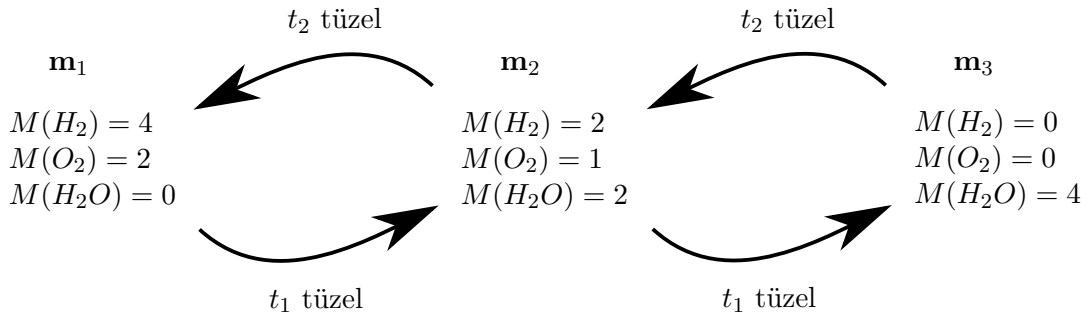
A hálók állapottere a tapasztalat szerint gyakran exponenciálisan nő a háló méretével, ezért már relatíve kis modellek esetén is óriásira nőhet az állapotok száma. Ezt a jelenséget *állapottér-robbanásnak* nevezik. Emiatt az állapottér felderítése nem egyszerű feladat.

Egy modell állapotterében szereplő állapotokhoz definiálhatjuk azok *távolságát*. Egy állapot távolsága alatt azt a legkisebb d számot értjük, ahány tüzeléssel a kezdőállapotból az adott állapot elérhető. Azaz $\mathbf{s}$ állapot távolsága d, ha

$$\forall i < d : \mathbf{s} \notin \mathcal{N}^i(\mathbf{s_0}) \wedge \mathbf{s} \in \mathcal{N}^d(\mathbf{s_0})$$

ahol $\mathbf{s_0}$ a kezdőállapot, $\mathcal{N}^i$ pedig a $\mathcal{N}$ függvény i-szer alkalmazva ($i > 0$). Az $\mathbf{s}$ állapot d távolságát szokás jelölni $\delta(\mathbf{s}) = d$ formában is.

5. példa. *A 2.1. ábrán látható Petri-háló állapottere látható a 2.3. 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*



2.3. ábra. A vízbontás modelljének állapottere

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.

A példában ha $\boldsymbol{m}_1$ a kezdőállapot, akkor $\delta(\boldsymbol{m}_1) = 0$, $\delta(\boldsymbol{m}_2) = 1$ és $\delta(\boldsymbol{m}_3) = 2$.

2.4. Döntési diagramok

A többváltozós függvények hatékony tárolása nem újkeletű probléma. A kiindulást a *döntési fák* szolgáltatták, amelyek 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.

2.4.1. Többértékű döntési diagramok

A *többértékű döntési diagramok* (multi-valued decision diagram, MDD) a döntési fákból vagy a Bryant által [1]-ben bemutatott bináris döntési diagramokból származtathatók. Az MDD-k segítségével gráfalapú reprezentációval $f(x_n, \ldots, x_1) \to \{0, 1\}$ függvények kódolhatók, ahol 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 mint gráf V csúcshalmazában minden v csúcshoz (csomóponthoz) hozzárendelünk egy $level(v) \in \{0, 1, \ldots, n\}$ számot, ezt nevezzük a csomópont szintjének. (Az átláthatóság kedvéért pszeudókódokban más jelölést is alkalmazunk: $level(v) \equiv v.level$.) Az olyan w csomópontokat, amelyekre $level(w) = 0$, *terminális csomópontoknak* nevezzük. Ezekhez egy bináris érték is tartozik: $value(w) \in \{0, 1\}$. A többi csomópontot *nemterminális csomópontnak* nevezzük. Ezekhez értéket nem rendelünk, azonban kimenő irányított éleik vannak. Egy v nemterminális csomópontból $|D_l|$ darab, nullától *sorszámozott* él vezet ki, ha $level(v) = l$. Ha egy v csomópont i. éle egy w csomópontba vezet, akkor azt $v[i] = w$ formában írhatjuk. A nemterminális csomópontok közül kitüntetett szerepű a gyökércsomópont: ez az egyetlen r csomópont, amelyre $level(r) = n$.

Az általunk használt döntési diagramokban nem engedjük meg, hogy létezzen különböző v és v' csomópont úgy, hogy $level(v) = level(v')$ és $\forall i : v[i] = v'[i]$. Az ilyen tulajdonságú döntési diagramokat *kanonikus döntési diagramoknak* nevezzük.

Amennyiben minden nemterminális csomópont minden gyereke pontosan egy szinttel van lejjebb, akkor a döntési diagramot *kváziredukált döntési diagramnak* nevezzük. Tehát formálisan akkor kváziredukált egy döntési diagram, ha:

$$\forall v \in V : level(v) > 0, \forall i \in \{0, 1, \ldots, |D_{level(v)}| - 1\} : level(v[i]) = level(v) - 1$$

Ilyen esetben a kódolt függvényre:

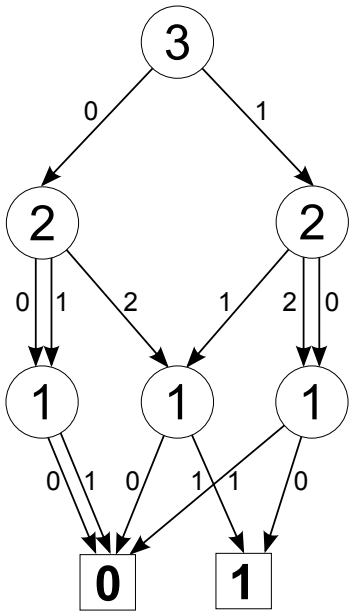
$$f(x_n, \ldots, x_1) = 1 \Leftrightarrow value(r[x_n][x_{n-1}] \ldots [x_1]) = 1$$

(Az r a függvényt kódoló diagram gyökérelemét jelöli.)

A Bryant által bemutatott bináris döntési diagramok az MDD-k speciális típusainak tekinthetők, ahol $\forall i : D_i = \{0, 1\}$.

Az MDD-ket grafikusan is ábrázolhatjuk. A terminális csomópontokat négyzettel (és a beleírt *value* értékkel) jelöljük, míg a nemterminális csúcsok jele a kör. A körök belsejét üresen hagyjuk vagy beleírhatjuk a reprezentált csomóponthoz tartozó *level* értéket. A csomópontokból kiinduló élekre ráírjuk az él sorszámát. Az átláthatóbb ábrázolás érdekében gyakran csak azokat az éleket és csomópontokat ábrázoljuk, amelyek a gyökér és a terminális egy csomópont közti utak valamelyikén található, a terminális nullába vezető utakat nem.

6. példa. *Egy többértékű döntési diagram látható a 2.4 ábrán. Az általa kódolt függvény az alábbi* (x_1, x_2, x_3) *értékekre ad igaz (1) eredményt:* $(0, 2, 1), (1, 0, 0), (1, 1, 1), (1, 2, 1)$*. Tehát például* $f(0, 2, 1) = 1$*, de* $f(0, 0, 0) = 0$*.*



2.4. ábra. Példa többértékű döntési diagramra

Lehetőség van MDD-k kompaktságának további növelésére. Amennyiben egy v csomópont minden kimenő éle azonos w alsóbb szintű csomópontba mutat, akkor a v csomópont szükségtelen, minden v-be mutató él közvetlen mutathat a w csomópontba. Ilyenkor a fentiekben adott interpretációk közvetlenül nem alkalmazhatók. Úgy kell azonban tekinteni, hogy a v csomópont szintje által reprezentált változó értéke nem befolyásolja a függvény által rendelt értéket.

2.4.2. Élcímkézett döntési diagramok

Az *élcímkézett döntési diagramok* (edge-valued decision diagrams, EDD) a többértékű döntési diagramok kiterjesztései. Az EDD-k segítségével $f(x_n, \ldots, x_1) = \mathbb{N} \cup \{\infty\}$ függvények fejezhetők ki. Ezt úgy érhetjük el, hogy a diagram minden éléhez a sorszáma mellett egy élsúlyt is rendelünk. Így egy v csomópont i. gyereke egy w csomópont egy s súlyú élen

keresztül ($s \in \mathbb{N} \cup \{\infty\}$), amelyet így jelölhetünk $v[i] = \langle w, s \rangle$. Egy EDD-t *kanonikusnak* nevezünk, ha minden csomópontjából vezet ki nulla súlyú él.

Az élsúlyok használata miatt szükségtelen több terminális csomópont definíciója, ezért az EDD-k esetén egyetlen terminális csomópont létezik, amelyet $\perp$ jelöl. További addíció az MDD-khez képest, hogy az EDD-k esetén a gyökérelemhez rendelünk egy $\rho \in \mathbb{N} \cup \{\infty\}$ értéket, amelyet *lebegő élsúlynak* (dangling edge weight) nevezünk.

Az így leírt döntési diagram által kódolt függvény:

$$f(i_n, \ldots, i_1) = s \Leftrightarrow r[i_n] = \langle v_{n-1}, s_{n-1} \rangle, v_{n-1}[i_{n-1}] = \langle v_{n-2}, s_{n-2} \rangle, \ldots, v_1[i_1] = \langle \perp, s_1 \rangle \\ \wedge\, s = s_1 + \ldots + s_{n-1} + \rho$$

Néhány egyszerűsítő jelölést is bevezetünk. Egy $e = \langle v, s \rangle$ él esetén $e.node = v$ és $e.value = s$.

Az élcímkézett döntési diagramok részletesebb definíciója és a fenti definíciók következményei bővebben a [28, 25]-ben olvashatók.

2.5. Modellellenőrzés

A modellellenőrzés egy olyan verifikációs technika, amely a rendszer egy véges modelljén vizsgálja meg, hogy bizonyos adott tulajdonságok teljesülnek-e [21]. Formálisan: azt vizsgáljuk, hogy egy M modellre egy adott r követelmény igaz-e [4]. Az r követelmény többféle, különböző kifejezőerejű formalizmussal is megadható, amelyekkel eltérő típusú kifejezések értékelhetőek ki. A következőkben az *elágazó idejű temporális logikával* foglalkozunk.

A CTL (Computational Tree Logic, elágazó idejű temporális logika) széles körben használt formalizmus egyszerűsége és kifejezőereje miatt. Segítségével kijelentések igazságának logikai időbeli változását vizsgálhatjuk [21]. 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:

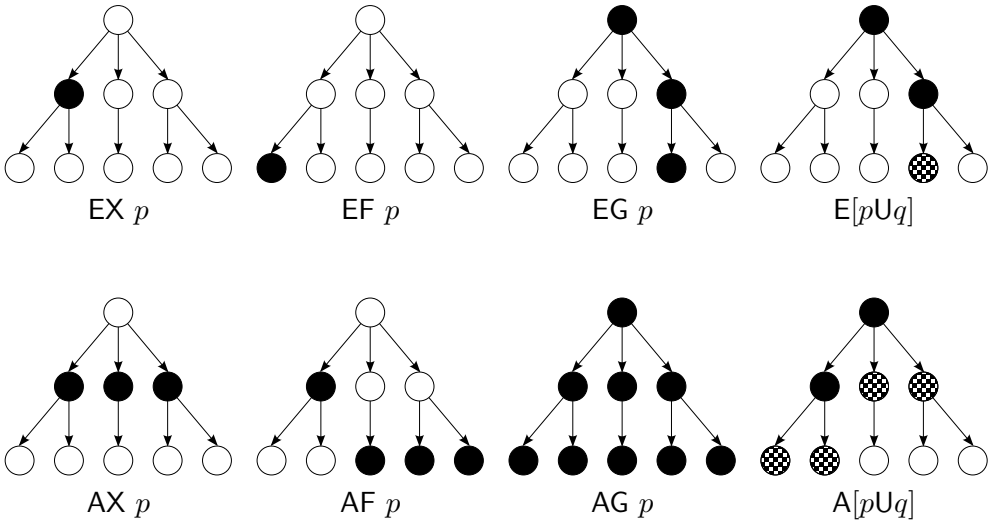
- Minden P atomi kijelentés egy állapotkifejezés.
- Ha p és q állapotkifejezések, akkor $\neg p$ és $p \wedge q$ is állapotkifejezés.
- Ha p és q állapotkifejezések, akkor $\mathsf{X}\ p$ és $p\ \mathsf{U}\ q$ útvonalkifejezések.
- Ha s útvonalkifejezés, akkor $\mathsf{E}\ s$ és $\mathsf{A}\ s$ állapotkifejezések.

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

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:



2.5. ábra. Minták a CTL operátorokra

- 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 2.5 ábrán látható [12]. 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.

A nyolc lehetséges főbb operátor (EX, EF, EU, EG, AX, AF, AU, AG) egymástól nem független. Választható úgy három operátor, hogy azokból az összes többi kifejezhető legyen. Például az $\{\mathsf{EX}, \mathsf{EU}, \mathsf{EG}\}$ operátorhalmazból az összes többi kifejezhető a következőképp [9]:

- $\mathsf{AX}\ p \equiv \neg\mathsf{EX}\ \neg p$,
- $\mathsf{AG}\ p \equiv \neg\mathsf{EF}\ \neg p$,
- $\mathsf{AF}\ p \equiv \neg\mathsf{EG}\ \neg p$,
- $\mathsf{A}\ [p\ \mathsf{U}\ q] \equiv \neg\mathsf{E}\ [\neg q\ \mathsf{U}\ (\neg p \wedge \neg q)] \wedge \neg\mathsf{EG}\ \neg q$,
- $\mathsf{EF}\ p \equiv \mathsf{E}\ [\mathit{true}\ \mathsf{U}\ p]$.

Bár a korábban leírt formális szintaxisból következik, külön kiemelendő, hogy a CTL-kifejezések egymásba is ágyazhatók (pl. $\mathsf{EF}(\mathsf{EG}\ p)$).

2.6. Szaturáció alapú modellellenőrzés

A bemutatott modellellenőrzéshez tipikusan szükség van a modell elérhető állapotainak, azaz állapotterének felderítésére.

Az állapottér felderítésére számos módszer létezik. Legegyszerűbb esete az *explicit állapottér-generálás*, amikor kiindulunk egy kezdeti tokeneloszlással, majd tüzelésekkel újabb állapotokat derítünk fel. Ezt mindaddig folytatjuk, amíg már egyik állapotból sem

lehetséges tüzeléssel új állapotba eljutni. Ezzel a megoldással azonban könnyen akadályba ütközhetünk, hiszen minden egyes állapot egyenként történő bejárása és eltárolása nagyobb modellek esetén lehetetlen a kielégíthetetlen erőforrásigény miatt.

Szimbolikus megközelítésekkel, megfelelő állapottér-leképezésekkel nagyobb méretű modellek is vizsgálhatók, például az [1]-ben bemutatott bináris döntési diagramok (BDD-k) felhasználásával [4].

A 2000-es évek elején egy újabb módszert javasoltak a hatékony állapottér-generálásra: az ún. *szaturációs algoritmust* [5, 6]. 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, amely jobban alkalmazkodik a döntési diagramok tulajdonságaihoz és az általuk kódolt részinformációkhoz, így jelentős gyorsulást értek el.

Ciardo és társai a szaturációs algoritmus segítségével bizonyos modelleknél sikeresen derítettek fel 10^{6000} méretű állapotteret néhány perc alatt egy közönséges személyi számítógéppel [7]. A szaturáció általában jelentősen kisebb idő- és memóriaigényű aszinkron rendszerek esetén, mint a többi algoritmus. Az algoritmus működéséről részletesen olvashat az alábbi irodalmakban: [5, 6, 7, 13, 16].

A következőkben röviden leírjuk a klasszikus szaturációs algoritmus főbb jellegzetességeit. Bemutatjuk, hogy az [5, 6, 7]-ben leírt technikák a modellek milyen típusú dekompozíciójára építenek (2.6.1), hogyan kezelik az eseményeket (2.6.2) és hogyan használják a next-state függvényeket (2.6.3). Bemutatjuk továbbá egy Petri-háló állapotterének leírását MDD-vel (2.6.4), majd ismertjük a szaturációs algoritmust és ennek tulajdonságait (2.6.5–2.6.6).

2.6.1. A modell dekomponálása

Az állapottér generálása előtt a Petri-hálót K darab részmodellre bontjuk, 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 szimbolikusan $(i_1, \ldots, i_K)$ 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:

- $\mathbf{i} = (i_1, \ldots, i_K)$, 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.

Az egyszerű Petri-hálók tetszőleges partícionálása Kronecker-konzisztens felbontáshoz vezet [7]. A következő fejezetekben leírtak mind Kronecker-konzisztens felbontás esetére vonatkoznak, ha külön nem jelezzük ennek ellenkezőjét.

2.6.2. Események kezelése

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, amelyek 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, amelynek lokális állapotát az esemény tüzelése befolyásolhatja. Hasonlóképp $Bot(\alpha) = l$ jelöli azt az l. legkisebb szintet, amelynek 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:

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

2.6.3. A next-state függvény

Az $\mathcal{N}$ next-state függvényt globális tárolás helyett dekomponáljuk események és részmodellek szerint [3]. Így $K \cdot |\mathcal{E}|$ db függvényre bontható az eredeti $\mathcal{N}$ függvény. A kapott $\mathcal{N}_{k,\alpha}$ lokális next-state függvények megadják, hogy adott k. szinten az egyes lokális állapotokból α esemény hatására milyen lokális állapotok érhetők el. Ez a fajta dekompozíció aszinkron rendszerek esetén különösen hatékony [6]. Mint az előző fejezetben bemutattuk, az események hatásai aszinkron rendszerek esetén jellemzően lokálisak, amit a next-state függvény dekompozíciójával hatékonyan ki is tudunk használni. Szinkron rendszerek esetén az algoritmus a többi szimbolikus módszerhez hasonlóan működik.

Mivel a szaturáció lényegében független az állapotátmenetek tárolásának módjától, ezért a tényleges megvalósítási lehetőségeket a 3. fejezetben tárgyaljuk.

2.6.4. Állapottér leírása többértékű döntési diagrammal

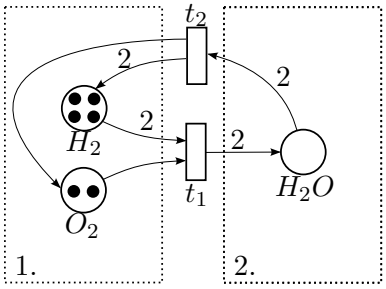
Az [5, 6, 7]-ben a K részre bontott modell globális állapotait lokális állapotokból képzett $(i_K, \ldots, i_1)$ K-sokkal írhatjuk le. Az ezekből álló állapotteret egy $K + 1$ szintű MDD-ben tárolhatjuk, amelyben minden (nemterminális) szinthez a Petri-háló helyeinek egy-egy diszjunkt halmaza tartozik. Ez a döntési diagram csak az állapotokat tárolja, a dolgozatban tárgyalt megoldásokban az állapotátmenetek kódolása külön struktúrában történik.

7. példa. *Példaképp a 2.1(a) ábrán látható modell állapotterét mutatjuk be MDD-vel leírva. Bár a modell mérete nem kívánja meg feltétlenül, a modellt partícionájuk: az első részmodellbe a H_2 és O_2 helyek, a második részmodellbe pedig a H_2O hely fog tartozni. Ezt mutatja a 2.6(a) ábra.*

Ez után kódoljuk az egyes szinteken lehetséges állapotokat a 2.6(b) és a 2.6(c) ábra szerint. 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 2.6(d) ábrán látható MDD által kódolt függvény is pontosan ezen értékekre fog 1 értéket adni.

Természetesen a 2.6(d) ábrán látható MDD ugyanazon állapotokat kódolja, mint amelyek a 2.3 ábrán láthatók, csak más kódolás alapján.



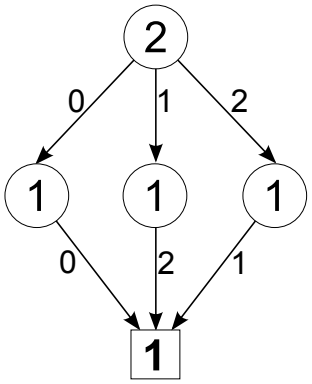
(a) Partícionált modell

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

2.6. ábra. Állapottér kódolása MDD-vel

2.6.5. A szaturációs algoritmus működése

A korábbi megközelítsekhez képest [5, 6]-ban a már felderített állapotteret leíró csomópontok bejárására egy speciális iterációs stratégiát használnak, amelyet *szaturációnak* neveznek. Ennek az iterációs stratégiának a lényege az, hogy mélységi bejárást hajtunk végre az MDD-n. A bejárás során az egyes eseményeket – azok lokalitását kihasználva–, megfelelő sorrendben és kimerítően tüzeljük el. Ezt a szaturációs stratégiát kiegészítve a hatékony állapottér-reprezentációval gyors és memóriahatékony algoritmust kapunk.

Szaturálás során egy i. szinten lévő csomópont esetén az összes olyan α eseményt eltüzeljük, amelyre $Top(\alpha) = k$. Egy csomópont *szaturált*, ha az általa reprezentált állapothalmazból 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, amelyekre $Top(\alpha) = 1$).

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

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

Formálisan: legyen $\mathcal{B}(k,p)$ a p csomópont részgráfja által kódolt állapotok halmaza ($level(p) = k$). Legyen továbbá $\mathcal{E} = \{e|Top(e) \leq k\}$. Egy p csomópont szaturált, ha $\mathcal{B}(k,p) = \bigcup_{\forall e \in \mathcal{E}} \mathcal{N}_e^*(\mathcal{B}(k,p))$. Ha az r gyökércsomópont szaturált lesz és így $\mathcal{B}(K,r) = \mathcal{N}^*(\mathbf{s_0})$, akkor az állapottér-generálás befejeződik [7].

A szaturációs algoritmus pontosabb működését és a használt algoritmusok pszeudókódját megtalálhatja a következő munkákban: [5, 6, 9, 13].

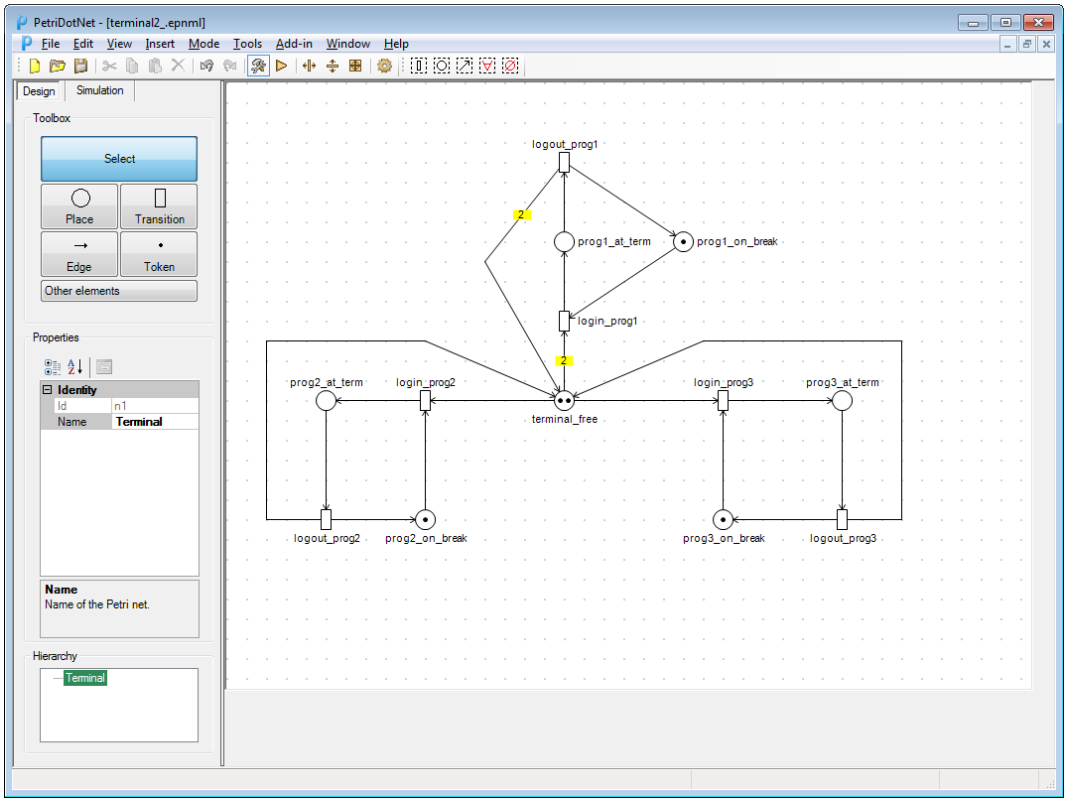
2.6.6. Klasszikus modellellenőrzés szaturációs alapokon

Szaturációs algoritmusok segítségével nem csak állapottér-felderítés, hanem modellellenőrzés is végezhető [9]. A klasszikus megközelítést vizsgáltuk és megvalósítottuk korábbi munkáink során [13, 16, 24], részletesen ezekben a munkákban olvashat a témáról. Jelen dolgozatban helyhiány miatt csak az ehhez képest végzett algoritmikus és implementációs fejlesztéseinket ismertetjük.

2.7. A PetriDotNet keretrendszer

A *PetriDotNet* egy Petri-hálók szerkesztésére, szimulációjára és analízisére eszköz (2.7. ábra), amely fejlesztése a BME Méréstechnika és Információs Rendszerek Tanszéken folyik 2008 óta. A program képes kezelni az egyszerű Petri-hálókon kívül tiltó élekkel kiegészített vagy kapacitáskorlátokkal ellátott hálókat és időzített, sztochasztikus modelleket is, továbbá jólformált színezett Petri-hálókat. A keretrendszer .NET alapokon nyugszik, így Windows, Linux és Mac OS alatt is futtatható (.NET vagy Mono frameworkkel). Előnye a manapság elérhető többi hasonló alkalmazáshoz képest, hogy bárki könnyen fejlesztheti hozzá beépülő modulokat, illetve felhasználóbarát, könnyen használható. A korábbi években több TDK-dolgozat, szakdolgozat és önálló laboratóriumi munka keretében készültek különféle beépülők a funkcionalitás növelésére, illetve az oktatásban is felhasználásra került.

Részletesen a *PetriDotNet* keretrendszerről a keretrendszer honlapján [29] olvashat. Ugyanitt a keretrendszer le is tölthető, kipróbálható.



2.7. ábra. A *PetriDotNet* keretrendszer főablaka

3. fejezet

Állapotátmenet-leképezés

A 2.3. fejezetben bevezettük az $\mathcal{N}$ állapotátmenet-függvényt. Mivel az aszinkron rendszerek esetén a lehetséges állapotátmenetek száma igen nagy lehet, és azokat szimbolikus módszerek használatakor el kell tárolni, ezért ennek hatékony megvalósítása fontos szempont. Példaként megemlítjük, hogy egy egyszerű, 10 filozófust tartalmazó modellben az állapotátmeneti reláció nagyságrendileg ezer különböző állapotátmenetet tartalmaz, míg az állapotok száma alig több mint 100. Ebben a fejezetben azt mutatjuk be, hogy ezeket a next-state függvényeket hogyan tudjuk hatékonyan építeni és tárolni.

Az állapotátmenet-relációk hatékony számításának szempontjából fontos, hogy milyen a modell karakterisztikája. Egyszerű Petri-hálók esetén, amelyekre teljesül a Kronecker-konzisztencia feltétele, a Kronecker-mátrixos állapotátmenet-tárolás hatékony megoldást nyújt. Ez fejti ki részleteiben a 3.1. fejezet. Azonban sok modell nem teljesíti ezt a feltételt, ilyenkor a bonyolult állapotátmeneteket szimbolikusan, MDD-ket használva tudjuk eltárolni. Ez a megoldás sokkal rugalmasabb, mint a Kronecker-mátrix alapú reprezentáció, és akár komplex állapotátmenet-relációk kezelésére is alkalmas. Ennek bemutatása történik a 3.2. fejezetben.

3.1. Kronecker-mátrixok

Korábban a 2.6.1. fejezetben leírtuk, hogy a Kronecker-konzisztens modellek esetén a next-state függvény dekomponálható események és részmodellek szerint. A dekompozíció eredménye, hogy az egyes lokális állapotátmenet-függvények mérete kisebb lesz, mint dekompozíció nélkül. A k. részmodell α eseményéhez (ahol $1 \leq k \leq K$, $\alpha \in \mathcal{E}$) tartozó lokális állapotátmenet-függvényt jelölje $\mathcal{N}_{k,\alpha}$. Ezek könnyen kódolhatóak az $\mathbf{N}_{k,\alpha} \in \{0,1\}^{|\mathcal{S}_k|\times|\mathcal{S}_k|}$ mátrixokkal (ahol $\mathcal{S}_k$ a k. szint lokális állapotainak halmaza) a következő módon:

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

A fenti mátrixokat az egyszerűség kedvéért a következőkben *Kronecker-mátrixoknak* nevezzük [2]. Egyszerű Petri-hálók esetén ezek a mátrixok kevés 1-es értéket tartalmaznak, ugyanis az ilyen típusú hálók szemantikájából következően a tranzíciók egyértelműen meghatározzák, hogy mely lokális állapotból mely lokális állapotba vezetnek át. Így a Kronecker-mátrixokban soronként legfeljebb egyetlen 1-es érték szerepelhet, a többi érték 0 lesz.

8. példa. *Tekintsük a 2.6(a) ábrán látható modellt illetve dekompozíciót. A részmodellek állapotainak kódolása legyen a 2.6(b) és a 2.6(c) ábrák szerinti.*

Ekkor a t_1 esemény tüzelésével az 1. részmodell a 0 állapotból az 1 állapotba, újabb tüzelésével pedig a 2 állapotba kerül. A t_1 esemény a 2. részmodellben a $0 \to 2$ és $2 \to 1$

átmeneteket eredményezi. A t_2 esemény hatására a $2 \to 1$ és az $1 \to 0$ állapotátmenetek mehetnek végbe az 1. részmodellben. A 2. részmodellt a t_2 tüzelése az 1 állapotból 2-be, a 2 állapotból pedig a 0 állapotba viszi.

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

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$
(a) $\mathbf{N}_{1,t_1}$ Kronecker-mátrix

$$\begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$
(b) $\mathbf{N}_{1,t_2}$ Kronecker-mátrix

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$
(c) $\mathbf{N}_{2,t_1}$ Kronecker-mátrix

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$
(d) $\mathbf{N}_{2,t_2}$ Kronecker-mátrix

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

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 rájuk.

Mindazonáltal a Kronecker-mátrixok és a részmodellekre bontás nem old meg minden problémát. A gyakorlatban azok a modellek, amelyek adatokat is tartalmaznak, csak nehezen vagy egyáltalán nem írhatóak le egyszerű Petri-hálók segítségével [15]. Ilyenkor a modellezést egy magasabb szintű, nagyobb kifejezőerejű modellezési nyelv segítségével kell elvégezni. Ez lehet akár prioritásokat kezelő Petri-háló vagy színezett Petri-háló is. A Petri-hálóknak ezen utóbbi típusai viszont nem teljesítik a Kronecker-konzisztencia feltételeit, így általános állapotátmeneteik nem kódolhatók Kronecker-mátrixok segítségével. Ilyen esetekben nyújthat megoldást az állapotátmeneti relációk döntési diagram alapú leképzése, amelyek segítségével tetszőleges relációt tudunk ábrázolni [11].

3.2. Többértékű döntési diagramok használata az állapotátmenetek tárolására

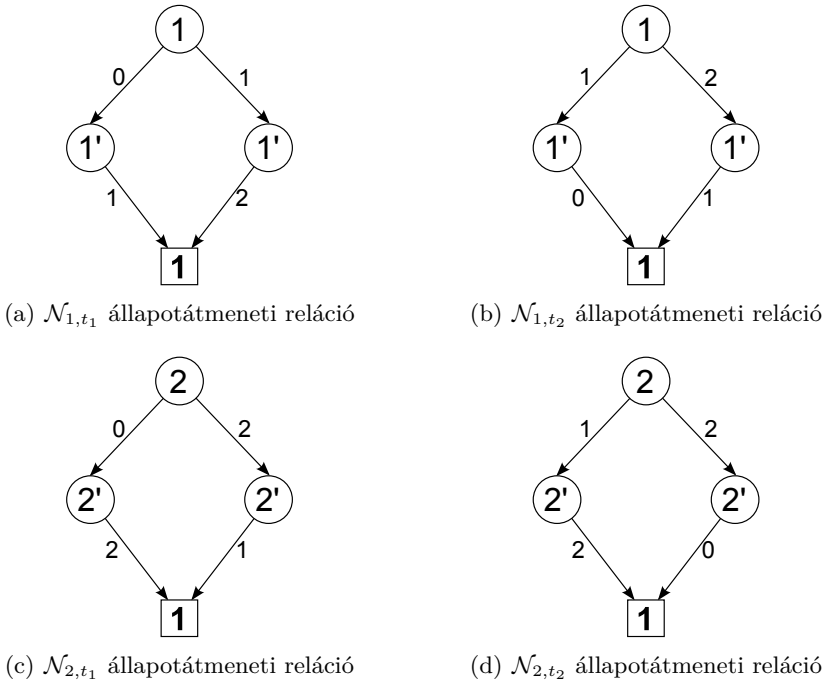
A következőkben azt fogjuk bemutatni, hogy a 2.4.1. fejezetben bemutatott döntési diagramok miként alkalmazhatók az állapotátmenetek szimbolikus kódolására.

Ha a vizsgált modellt a korábbiaknak megfelelően K részre bontjuk, akkor az $\mathcal{N}$ next-state függvény leírható egy $2 \cdot K$ szintet tartalmazó MDD-vel (ahol K a részmodellek száma az állapottér kódolásában). Egy útvonalat, amely az MDD gyökerétől valamelyik terminális csomópontba vezet, tekintsük az $(i_K, i'_K \ldots, i_1, i'_1)$ sorozatnak. Itt az $i_j : 1 \leq j \leq K$ jelölje a j. szinten azt az állapotot, amelyben az MDD által kódolt esemény tüzelése előtt vagyunk, míg az $i'_j : 1 \leq j \leq K$ jelölje a j. szinten azt az állapotot, amelybe az esemény tüzelése után jutunk. Ennek megfelelően $i_j, i'_j \in \mathcal{S}_j : 1 \leq j \leq K$. Egy $(i_K, i'_K \ldots, i_1, i'_1)$ sorozat tehát azt a globális állapotátmenetet kódolja, ahol a kiinduló állapot $(i_K \ldots, i_1)$, a tüzelés utáni állapot pedig $(i'_K \ldots, i'_1)$, illetve az $i_j, i'_j \in \mathcal{S}_j$ átmenet akkor lehetséges, ha a neki megfelelő útvonal az MDD gyökerétől a terminális 1 csomópontba vezet.

Kronecker-konzisztens esetben az állapotátmeneti relációkat a következő módon tudjuk döntési diagramokra leképezni: $j \in \mathcal{N}_{k,\alpha}(i) \Leftrightarrow n[i][j] = \mathbf{1}$, ahol n a k. szint α eseményéhez

tartozó állapotátmeneti relációt kódoló MDD gyökércsomópontja. Az előzőekből adódik, hogy a Kronecker-mátrixszal kódolható relációk döntési diagramokkal is könnyedén reprezentálhatók.

9. példa. *Tekinstsük a 3.1. ábra Kronecker-mátrixait. A mátrixok által kódolt állapotátmenetek egyszerűen kódolhatóak többértékű döntési diagramokkal is. Az MDD-vel történő kódolást mutatja a 3.2. ábra, amelyen az átláthatóság kedvéért nem tüntettük fel a terminális 0 csomópontot.*



(a) $\mathcal{N}_{1,t_1}$ állapotátmeneti reláció (b) $\mathcal{N}_{1,t_2}$ állapotátmeneti reláció

(c) $\mathcal{N}_{2,t_1}$ állapotátmeneti reláció (d) $\mathcal{N}_{2,t_2}$ állapotátmeneti reláció

3.2. ábra. Next-state függvény reprezentálása MDD-vel

A Kronecker-mátrixokkal szemben a döntési diagramokkal nem csak Kronecker-konzisztens modellek állapotátmenti reláció írhatók le. Köszönhetően annak, hogy a döntési diagramok nagyobb kifejezőerejűek, tetszőleges modellek véges állapotátmeneti relációi leírhatóak velük, így alapul szolgálnak további fejlesztéseinkhez.

4. fejezet

Korlátos modellellenőrzés szaturációs algoritmussal

Ugyan a klasszikus szaturációs modellellenőrzési technikák bizonyos jól meghatározott problémaosztályokra (bonyolult logikát és adatokat nem tartalmazó aszinkron rendszerekre) igen hatékonyan működnek, más problémákra nem nyújtanak megoldást. Ilyen például a nagyon bonyolult vagy végtelen állapotterű modellek kezelése.

Számos esetben azonban erre nincs is szükség, mivel a vizsgálandó kifejezésekre válasz adható a teljes állapottér egy részének felderítése és vizsgálata alapján is. Ez a *korlátos modellellenőrzés* módszerének alapja.

Ebben a fejezetben röviden bemutatjuk a korlátos modellellenőrzést, a szaturáció alapú korlátos állapottér-felderítő algoritmusok működését, majd megmutatjuk, hogyan lehet ezeket az algoritmusokat modellellenőrzésre használni, illetve leírjuk az algoritmusokon végrehajtott fejlesztéseinket is.

4.1. Korlátos modellellenőrzés

A korlátos modellellenőrzés során a specifikált kritériumokat nem a modell teljes állapotterén, csak annak egy k-korlátos részén vizsgáljuk. (A k értéket a vizsgálat előtt rögzítjük.) Egy $\mathcal{S}$ állapottér k-korlátos része alatt a következő $\mathcal{S}_{b,k}$ halmazt értjük:

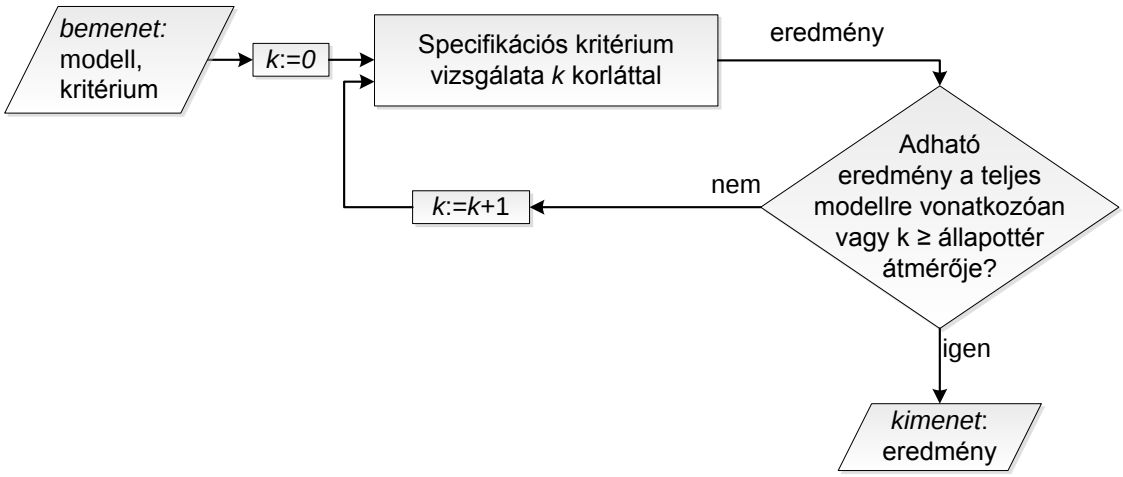
$$\mathcal{S}_{b,k} = \{\mathbf{s} | \mathbf{s} \in \mathcal{S}, \delta(\mathbf{s}) \leq k\},$$

ahol $\delta(\mathbf{s})$ alatt az $\mathbf{s}$ állapot kezdőállapottól mért távolságát értjük, ahogyan ezt a 2.3. fejezetben definiáltuk.

Ez a folyamat ismételhető egyre növekvő k korlátokkal mindaddig, amíg a specifikációs kritérium kiértékelése meg nem tehető a teljes állapottérre vonatkozóan. Például ha egy $\mathcal{S}$ állapot *elérhetőségét* vizsgáljuk a modellben és egy adott k korláttal $\mathcal{S}$ nem elérhető, akkor ez csak a k-korlátos állapottérrészre vonatkozóan ad eredményt, ebből még nem vonhatunk le következtetést a teljes állapottérre nézve, csak ha a k korlát már elérte az állapottér átmérőjét, tehát a teljes állapottér felderítésre került. Ha viszont a korlátos részben $\mathcal{S}$ elérhető, akkor ez a teljes modellre vonatkozóan is azt jelenti, hogy $\mathcal{S}$ elérhető. Ezt a megközelítést illusztrálja a 4.1. ábra.

4.2. Korábbi módszerek korlátos szaturációra

A szaturáció alapú korlátos állapottér-felderítés alapjait a [27]-ben fektették le. A cél egy olyan algoritmus fejlesztése volt, amely segítségével bejárható egy állapottér azon



4.1. ábra. Iteratív korlátos modellellenőrzés

része, amelyben a kezdőállapottól legfejlebb k távolságra elhelyezkedő állapotok találhatók. Azonban ebben a publikációban a szaturációs alapokon történő korlátos modellellenőrzést nem tárgyalták.

A klasszikus szaturációs algoritmus kiterjesztése korlátos esetre nem triviális. A speciális bejárási sorrend és a lokális tüzelések miatt nehéz korlátozni az állapottér bejárását. Ennek érdekében az egyik lehetőség az iterációs sorrend megváltoztatása, ám így elveszítenénk a szaturációs algoritmus hatékonyságát. Emiatt az algoritmus kiegészítését választottuk a [27]-ben bevezetett újításokkal.

Ezek az újítások az algoritmus több részét is érintik. Egyrészt a klasszikus szaturációs algoritmusokban használt többértékű döntési diagramok (MDD-k) helyett a korlátos algoritmusok élcímkézett döntési diagramokat (EDD-ket, lásd 2.4.2. fejezetben) használnak. Ezek segítségével minden egyes felderített állapothoz el lehet tárolni az állapot távolságát a kezdőállapottól. Azonban a lokális tüzelések miatt ez nem elegendő, mivel elképzelhető, hogy az állapottérbe bekerülnek olyan állapotok is, amelyek a kívánt k távolságkorláton kívül esnek. Ennek érdekében Yu és társai ún. *vágófüggvényeket* definiáltak. Ezek segítségével a korláton túli állapotok az állapottérből eltávolíthatók.

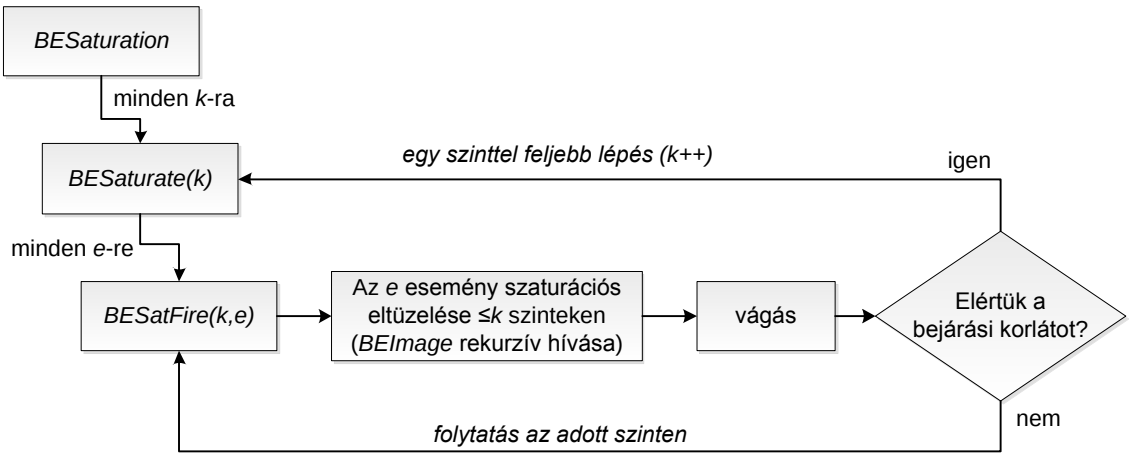
4.2.1. A korlátos szaturációs állapottér-felderítő algoritmusok működése

Ebben a fejezetben röviden ismertetjük a Yu, Ciardo és Lüttgen által [27]-ben bemutatott EDD-alapú állapottér-felderítő algoritmuson alapuló, [25]-ben bemutatott korlátos modellellenőrző algoritmusunkat. A Yu és társai által elvégzett munka lényege az volt, hogy a korábbiakban ismeretett klasszikus szaturációs algoritmust kiegészítették a bejárást a tárolt állapotok távolságán alapulva korlátozó elemekkel.

A szaturációs korlátos állapottér-felderítés működésének bemutatását az elemi műveletektől kezdjük és haladunk a teljes felderítést elvégző függvény felé. A következő leírás megértését segíti a magas absztrakciós szintű, szemléltető 4.2. ábra.

A *BoundedEDDImage* segédfüggvény (4. algoritmus) végrehajt egy tüzelési lépést az állapottér egy szintjén, majd a szaturációs iterációs sorrendet követve érvényesíti a változásokat az alsóbb változó szinteken is, a korlátos bejárási szabályokat figyelembe véve.

Ez után egy vágófüggvény segítségével a korlátos állapottér-reprezentáción kívül eső állapotokat eltávolítja. Yu és társai kétféle vágófüggvényt definiáltak, a *Truncate* hívás helyére behelyettesíthető a szigorú vágófüggvény (*TruncateExact*, 6. algoritmus, amely pszeudókódja tartalmazza már a később tárgyalásra kerülő fejlesztéseinket) vagy a közelí-



4.2. ábra. Szemléltető ábra a korlátos szaturációs algoritmusokhoz

tő vágófüggvény (*TruncateApprox*, 5. algoritmus). Az ezek közti különbség a 4.3. fejezetben olvasható. Végül a függvény a létrehozott új csomópontokat szaturálja a *BoundedEDDSaturate* függvény hívásával, azaz létrehozza az új csomópontokból elérhető lokális korlátos részállapottér reprezentációját.

A *BoundedSatFire* függvény (3. algoritmus) egy megadott p csomópont által reprezentált részállapottéren egy adott e eseményt kimerítően eltüzel. Működése hasonló a *BoundedEDDImage* függvényhez, de új csomópontokat nem hoz létre, csak a p csomópontot módosítja. Ezt a függvényt hívjuk meg az aktuálisan szaturálásra kerülő szintre, majd ez a függvény fogja rekurzívan az alsóbb szintekre eltüzelni az adott eseményt a *BoundedEDDImage* függvény segítségével. Azaz létrehozza a korlátos állapottér reprezentációt (az adott eseményre nézve).

A *BoundedSaturate* függvény (2. algoritmus) egy adott p csomópontra egy l szinten az összes $e \in \mathcal{E}_l$ eseményt kimerítően eltüzeli a *BoundedSatFire* függvény hívásaival.

Az egész algoritmus belépési pontja a *BoundedSaturation* függvény (1. algoritmus), amely létrehozza a kezdőállapotnak megfelelő csomópontokat az állapottérben, majd alulról felfelé haladva minden szinten a kezdőállapot csomópontjait szaturálja a *BoundedSaturate* függvénnyel. Így előállítja a kezdőállapotból elérhető állapotok tranzitív lezártját a bejárási korlát figyelembevételével, ami maga a korlátos állapottér.

Emellett az algoritmus fontos része a vágófüggvények működéséhez szükséges állapottávolság-számítás. Amikor egy $\mathbf{s}$ állapotból egy e eseményt eltüzelünk, a létrejövő új állapot távolsága $\delta(\mathbf{s}) + 1$. Ezt az inkrementálást a *BoundedEDDSatFire* függvény végzi a 11. sorban.

A döntési diagramokban szereplő távolságokat emellett a *BoundedEDDImage* függvény is befolyásolja úgy, hogy ha egy szaturációs lépést végrehajtunk egy állapothalmazon, akkor a döntési diagram által kódolt részállapottér távolságát a régi v távolság és a szaturáció során kapott normalizálással előálló γ távolság összegeként kapjuk.

Az algoritmusokban szereplő egyéb függvények jelentései:

- *NewNode(l)*: létrehoz egy új EDD csomópontot az l szinten.

- *Normalize(s)*: normalizál egy s csomópontot, azaz kiszámítja az s-ből kimenő élek minimális w élsúlyát, ezt kivonja mindegyik kimenő él súlyából és visszaadja w-t, így biztosítva a kanonikus reprezentációt.

- *CheckIn(l,s)*: elhelyezi s csomópontot az l szint csomópontjai közt az egyedi csomóponttárolóban, kivéve, ha már szerepelt ebben s-sel azonos jelentésű s' csomópont. Ez utóbbi esetben s' a visszatérési értéke, különben s.
- *UnionMin(a,b)*: elkészíti a és b csomópontok unióját reprezentáló részgráfot (részEDD-t) úgy, hogy az elemekhez rendelt távolság a korábbi értékek minimuma legyen.
- A *Confirm* függvény az eredeti algoritmusoknak nem része, jelentését később ismertetjük.

Megjegyzendő, hogy az eredeti algoritmushoz képest az itt szereplő pszeudókódokban a könnyebb érthetőség kedvéért nem tértünk ki az állapotátmeneti $\mathcal{N}$ függvény reprezentációjára. Implementációnkban ezt a 3. fejezetben leírt mindkét módszerrel megvalósítottuk, így a *BoundedEDDSatFire* és *BoundedEDDImage* függvények paraméterei sem feltétlen események, hanem kódolt eseménycsoportok is lehetnek, azonban ez az algoritmus helyességét nem változtatja meg [27].

Algoritmus 1: BoundedEDDSaturation

kimenet : gyökér : csomópont

```
1  l ← ⊥;                                          // terminális csomópont
2  for k = 1 to K do
3  |  Confirm(k, 0);
4  |  n ← NewNode(k);
5  |  n[0] ← ⟨0, l⟩;
6  |  BoundedEDDSaturate(n);
7  |  n ← CheckIn(k, n);
8  |  l ← n;
9  end
10 return l;
```

Algoritmus 2: BoundedEDDSaturate

bemenet: p : csomópont
kimenet : csomópont

```
1  l ← p.level;
2  repeat
3  |  foreach e ∈ E_l do                          // ∀e : Top(e) = l
4  |  |  BoundedEDDSatFire(p, e);
5  |  end
6  until p nem változik;
7  return p;
```

4.2.2. Megoldandó problémák a korábbi algoritmusokban

Bár Yu és társai a [27]-ben megalapozták a szaturáció alapú korlátos modellellenőrzést, korántsem adtak maradéktalanul megoldást a fejezet bevezetőjében leírt kihívásokra [25].

A priori ismeretek A [27]-ben leírt algoritmusok elméleti jellegűek, közvetlenül nem implementálhatók, mivel az állapottér-felderítő algoritmus épít a lokális állapotok

Algoritmus 3: BoundedEDDSatFire

bemenet: p : csomópont, e : esemény
kimenet : changed : logikai

```
1  l ← p.level;
2  i ← 0;
3  if l < Bot(e) then
4  |  return false;
5  end
6  repeat
7  |  i ← i + 1;
8  |  foreach (i, j) ∈ N_{l,e} do
9  |  |  if p[i].value ≥ bound then continue;
10 |  |  ⟨v, q⟩ ← BoundedEDDImage(p[i], e);
11 |  |  ⟨w, s⟩ ← Truncate(⟨v + 1, q⟩);
12 |  |  ⟨u, r⟩ ← UnionMin(p[j], ⟨w, s⟩);
13 |  |  if ⟨w, s⟩ ≠ ⟨u, r⟩ then
14 |  |  |  if j állapot nem megerősített then Confirm(l, j);
15 |  |  |  p[j] = ⟨u, r⟩;
16 |  |  end
17 |  end
18 until p nem változik;
19 return i > 1;
```

Algoritmus 4: BoundedEDDImage

bemenet: $\langle v, q\rangle$: él, e : esemény
kimenet : él

```
1  l ← q.level;
2  if l < Bot(e) then
3  |  return ⟨v, q⟩;
4  end
5  s ← NewNode(l);
6  foreach (i, j) ∈ N_{l,e} do
7  |  if p[i].value > bound then continue;
8  |  ⟨v, u⟩ ← BoundedEDDImage(q[i], e);
9  |  ⟨w, o⟩ ← Truncate(⟨v, u⟩);
10 |  ⟨u, r⟩ ← UnionMin(s[j], ⟨w, o⟩);
11 |  if ⟨w, o⟩ ≠ ⟨u, r⟩ then
12 |  |  if j állapot nem megerősített then Confirm(l, j);
13 |  |  s[j] = ⟨u, r⟩;
14 |  end
15 end
16 s ← BoundedEDDSaturate(s);
17 γ ← Normalize(s);
18 s ← CheckIn(l, s);
19 return ⟨γ + v, s⟩;
```

halmazának ismeretére, tehát a priori ismeretnek tekintik a lokális állapottereket. Ezek viszont tipikusan csak az állapottér feltérképezése után állnak rendelkezésre. (Elvileg ezt az információt a felhasználótól is várhatnánk, azonban ez nem várható el a modellellenőrzést végző személytől, továbbá aláásná a matematikailag megalapozott válasz adásának lehetőségét.)

Végtelen állapotterű modelleknél probléma továbbá az is, hogy a szaturációs algoritmusokban használt állapotátmenet-reprezentációkkal végtelen számú állapotátmenet nem kezelhető, azaz ilyen esetek kezelése az algoritmusok kiegészítése nélkül elvileg sem lehetséges.

Ennek kiküszöbölése érdekében ki kellett az algoritmusokat egészítenünk úgy, hogy az elvárt ismereteket futás közben állítsák elő.

Specifikációs kritériumok vizsgálata A cikkben csak az állapottér felderítésének folyamatáról esik szó, arról nem, hogyan lehet ez alapján formális kritériumok teljesülését vagy meghiúsulását vizsgálni. Ezért munkánk során integrálnunk kellett a korlátos állapottér-felderítő algoritmusokat a követelmények ellenőrzését végző algoritmusokkal. Munkánk [25] az első megvalósítása a korlátos szaturációs modellellenőrző algoritmusnak.

Megfelelő bejárási korlát Megoldatlan volt továbbá az is, hogy hogyan határozzuk meg a megfelelő bejárási korlátot. Az eredeti algoritmus bemenetként várta a bejárandó részállapottér mélységét. Ezt esetünkben úgy célszerű megválasztani, hogy az állapottérnek feleslegesen nagy részét ne kelljen bejárni, azonban a vizsgált követelményre nézve már lehessen következtetést levonni belőle. Az optimális korlát megállapítása viszont nem várható el sem a felhasználótól, sem egy algoritmustól.

Emiatt szisztematikus módszert kellett adnunk a különböző nagyságú korlátok kipróbálására.

Szigorú korlátok használata Yu és társai publikációjukban kétféle bejárási korlátot használtak: szigorú és közelítő (megengedő) korlátokat. (Ezek pontos definiálása a 4.3. fejezetben olvasható.) Arra a következtetésre jutottak, hogy a szaturációs algoritmus hatékonyan nem használható szigorú bejárási korláttal, mivel ennek nagy a számítási igénye, azaz hatékonyan csak közelítő korlátok használhatóak.

Bizonyos felhasználási területeken és bizonyos modellek esetén azonban szükséges lehet szigorú bejárási korlátok használatára [27]. Ezért a korábbi, nem hatékony szigorú korlátokat használó megoldások hatékonyságát meg kellett növelnünk.

A következő fejezetekben bemutatjuk, hogy az itt ismeretett problémákra milyen megoldásokat adtunk munkánk során.

4.3. Korlátos állapottér-felderítés implementációja és integrációja a klasszikus szaturáció alapú modellellenőrzővel

4.3.1. Az algoritmusok továbbfejlesztése

Az előző fejezetben ismertetett négy probléma közül kettő megoldására a [27]-ben leírt algoritmusokat kellett kiegészítenünk. Az állapottér-felderítés a priori ismeretigényének kiküszöbölésére ún. on-the-fly frissítéseket vezettünk be. Másrészt a szigorú vágófüggvényt is módosítanunk kellett, mivel az eredeti változata nem volt hatékony, itt gyorsítótárakkal egészítettük ki a korábbi megoldást.

On-the-fly frissítések (Folyamatos frissítések) Annak érdekében, hogy az állapottérről ne kelljen a priori ismeretekkel rendelkeznünk, az algoritmusokat kiegészítettük folyamatos frissítésekkel. Ez a módszer korábban már alkalmazásra került szaturációs algoritmusokban [9], viszont a publikált korlátos állapottér-felderítő algoritmusok nem voltak felkészítve folyamatos frissítésekre.

A folyamatos frissítések lényege az, hogy nem tekintjük adottnak egy lokális állapottér elemeit, hanem minden alkalommal, amikor egy lokális állapotba eljut az algoritmus, felfedezzük az ebből egy lépéssel esetlegesen elérhető további állapotokat, tehát menet közben építjük a lokálisan elérhető állapotok halmazát és a lokális állapotátmeneti relációt.

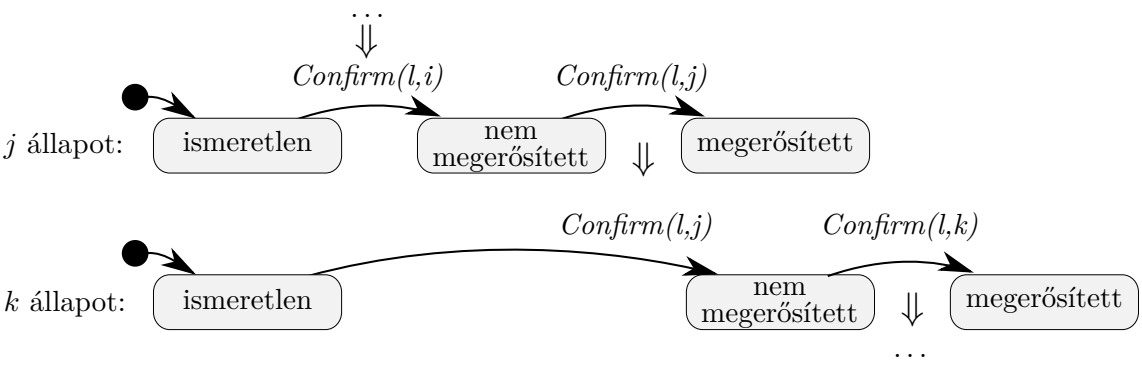
Az on-the-fly frissítések érdekében a lokális állapotokat három csoportra osztjuk:

- *megerősített* állapot: olyan lokális állapot, amely a felderített állapottér valamely globális állapotának része,
- *nem megerősített* állapot: olyan lokális állapot, amelyről van információnk, illetve lokális állapotátmenet is létezik hozzá megerősített állapotból, azonban nem ismert, hogy ez az állapotátmenet globálisan is végrehajtható-e,
- *ismeretlen* állapot: olyan lokális állapot, amelyről az algoritmusnak aktuálisan nincs tudomása.

Minden alkalommal, amikor egy tüzeléssel eljutunk egy új állapotba, az új állapot lokális részállapotait megerősítettnek jelöljük, hiszen biztosan globálisan is elérhető. Minden alkalommal, amikor ilyen megerősítés történik, a megerősített állapotból lokálisan rákövetkező állapotokat keresünk, amelyeket nem megerősített állapotként eltárolunk. A megerősítést és az új állapotok keresését végzi a *Confirm(l,j)* függvény, amely az l szinten lévő j lokális állapotot megerősítettnek jelöli és felderíti a j-ből l szinten egy lépéssel elérhető lokális állapotokat.

Ezt a folyamatot illusztrálja a 4.3. ábra egy l. szinten lévő j és k állapotra. A példában i az első olyan lokális állapot a szinten, amelyből elérhető j állapot (azaz i az első felderített s állapot, amelyre $j \in \mathcal{N}_l(s)$).

Az i lokális állapot megerősítése során felfedezésre kerül a j állapot, azonban még nem biztos, hogy globálisan is elérhetővé válik, így a j „nem megerősített” állapotba kerül. Amikor egy sikeres tüzelés során felfedezzük, hogy j állapotba is el lehet jutni, lefut a *Confirm(l,j)* függvény és a j állapot megerősítetté válik. Ezzel egyidőben minden olyan k lokális állapot is „nem megerősített” állapotba kerül, amelyek eddig ismeretlenek voltak, de amelyekre $k \in \mathcal{N}_l(j)$.



4.3. ábra. Lokális állapotok felderítése

Confirm hívások a kódban három helyen találhatóak. Egyrészt a *BoundedEDD-Saturation* függvény 3. sorában, amely a kezdőállapotot jelöli megerősítettnek (hiszen ez

biztosan elérhető), másrészt a *BoundedEDDSatFire* algoritmus 14. sorában és a *BoundedEDDImage* algoritmus 12. sorában, amelyek a felderített végrehajtható állapotátmenetek esetén jelölik a korábban nem megerősített állapotokat megerősítettnek.

Az állapotok ilyen, inkrementális módon történő felderítéséről bővebben [6, 13]-ban olvashat.

Hatékony szigorú vágófüggvény A korábbi munkák alapján hatékonyan vágófüggvényként csak közelítő jellegű vágófüggvény használható a szigorú vágófüggvény nagy lépésszáma miatt. A közelítő vágás (5. algoritmus) mindössze azt képes garantálni, hogy egy K részre dekomponált modell felderített $\mathcal{S}_b$ korlátos állapotterére d korlát használata esetén $\{\mathbf{s}|\mathbf{s} \in \mathcal{N}^{\leq d}(\mathbf{s_0})\} \subseteq \mathcal{S}_b$ és $\{\mathbf{s}|\mathbf{s} \in \mathcal{N}^{>K \cdot d}(\mathbf{s_0})\} \cap \mathcal{S}_b = \emptyset$, azaz a d korláton belül minden állapot szerepel az állapottérben és egyetlen állapot sem szerepel a $d \cdot K$ korláton kívülről.

Bizonyos esetekben, például tesztgenerálás során azonban szükség lehet szigorú vágófüggvények alkalmazására. Ekkor a felderített korlátos állapottér $\mathcal{S}_b = \{\mathbf{s}|\mathbf{s} \in \mathcal{N}^{\leq d}(\mathbf{s_0})\}$, tehát az állapottérben pontosan azok az állapotok szerepelnek, amelyek legfeljebb d távolságra vannak a kiinduló állapottól. Továbbá egyes modelleknél a közelítő vágófüggvény túlzottan sok állapotot derít fel feleslegesen. csomópont részgráfja által reprezentált állapothalmaz vágását visszavezeti p csomópont leszármazottainak vágására, azaz a függvény rekurzív.

Az algoritmusok vizsgálata során rájöttünk arra, hogy a szigorú vágófüggvényt kizárólag olyan csomópontokkal hívjuk meg, amelyek már szerepelnek a csomópontok egyedi tárolójában. Más okoknál fogva azonban ezeket a csomópontokat már nem módosítjuk, a belőlük kimenő élek változatlanok, és szintén ilyen „megváltoztathatatlan" csomópontokba mutatnak.

Ezért bevezettünk egy gyorsítótárat annak céljából, hogy megakadályozzuk a redundáns vágásokat ugyanazon a csomóponton. Ez nem változtatja meg az algoritmus működését, mivel ha a csomópontok és részgráfja változatlan, akkor a vágófüggvény eredményének sem szabad változnia. Így az egyes függvényhívásokhoz szükséges rekurzív hívások számát tudjuk drasztikusan csökkenteni az ismételt meghívások során.

A szigorú vágófüggvény [27]-ben leírt eredeti algoritmusának általunk kiegészített változata a 6. algoritmus. Az algoritmus 9. sorában látható a kiszámított értékek cache-be helyezése, a 2. sora pedig a cache-ből történő értékvisszaadást valósítja meg, amennyiben lehetséges.

Megjegyzendő, hogy mivel a függvény a bemenő paraméterként kapott $\langle v, p \rangle$ él v értékét csak akkor módosítja, ha v nagyobb a használt korlátnál, ezt az esetet viszont még a cache vizsgálata előtt lekezeljük. Ezért a cache-ben nem szükséges az élsúlyokat eltárolni, elegendő az élek végpontját.

4.3.2. Az állapottér-generálás és klasszikus szaturációs modellellenőrzés integrációja

A 4.3.1. fejezetben leírtak által lehetővé vált az állapotterek hatékony felderítésének implementációja. Azonban a célunk egy korlátos állapottéren alapuló szaturációs modellellenőrző létrehozása volt. Mivel ilyen megoldást eddig még nem vizsgáltak, ezért az implementáció kihívásai, az algoritmus helyessége és az elkészült algoritmus teljesítménye egyaránt vizsgálatunk tárgyát képezik. A teljesítményre vonatkozó megfigyeléseink a 7. fejezetben olvashatók.

Mivel korábban már készítettünk szaturációs modellellenőrző algoritmusokat [13], ezért a kihívást az jelentette, hogy ezek a módszerek korlátos állapottéren is működőképesek legyenek. A klasszikus szaturációs modellellenőrzők két fő információt várnak el a modellről:

Algoritmus 5: Közelítő vágófüggvény (TruncateApprox)

bemenet: $\langle v, p\rangle$: él
kimenet : él

```
1 if v > bound then return ⟨∞, ⊥⟩;
2 else return ⟨v, p⟩;
```

Algoritmus 6: Szigorú vágófüggvény (TruncateExact)

bemenet: $\langle v, p\rangle$: él
kimenet : él

```
1  if v > bound then return ⟨∞, ⊥⟩;
2  if vágási cache tartalmaz t értéket ⟨v, p⟩-hez then return ⟨v, t⟩;
3  n ← NewNode(p.level);
4  foreach i ∈ S_{p.level} do
5      r ← TruncateExact(⟨v + p[i].value, p[i].node⟩);
6      n[i] ← ⟨r.value − v, r.node⟩;
7  end
8  n ← CheckIn(p.level, n);    // elhelyezés a csomópontok egyedi tárolójában
9  ⟨v, p⟩ kulcshoz n érték beszúrása a vágási cache-be;
10 return ⟨v, n⟩;
```

Algoritmus 7: EDD–MDD konverzió

bemenet: E : EDD
kimenet : M : MDD

```
1  z_0 ← terminális 0 M-ben;
2  map(⊥) ← terminális 1 M-ben;
3  for i = 1 to K do
4      z_i ← NewNode(i);                          // K a gyökércsomópont szintje
5      ∀j : z_i[j] ← z_{i−1};                     // nullcsomópont létrehozása M-ben
6      foreach p_E ∈ {i. szint csomópontjai E-ben} do
7          p_M ← NewNode(i);                      // új csomópont létrehozása M-ben
8          for k = 0 to |D_i| − 1 do
9              if p_E[k].value = ∞ then
10                 p_M[k] ← z_{i−1};  // ∞ súlyú élek a terminális 0 felé vezetnek
11             else
12                 p_M[k] ← map(p_E[k].node);
13             end
14         end
15         if ∃p'_M ∈ M : p'_M = p_M then map(p_E) ← p'_M;
16         else
17             p_M csomópont elhelyezése M döntési diagramban;
18             map(p_E) ← p_M;
19         end
20     end
21 end
22 return M;
```

az állapotátmenet-függvényeket és a felderített állapotteret. Az állapotátmenetek a korlátos és nemkorlátos esetben nem különböznek, így ezen nem kellett változtatnunk. Az állapotterek reprezentációjában azonban van különbség: a korlátos állapottér-felderítés az állapotokat egy EDD-ben tárolja, míg a modellellenőrzés az állapottér MDD-jét használja, mivel a kifejezések kiértékelésekor az állapotok távolságinformációira már nincsen szükség.

Két megoldási lehetőség van erre a problémára: vagy a korábbi algoritmusokat adaptáljuk EDD-kre, vagy definiálunk egy megfelelő konverziót EDD-ből MDD-be. Mivel az EDD-ben tárolt távolságinformációk nem szükségesek és a távolságinformáció nélküli állapottér MDD-vel kompaktabb módon reprezentálható, ezért az EDD MDD-vé konvertálását választottuk. (Pontosabban az EDD-ket kváziredukált MDD-vé alakítottuk annak érdekében, hogy a korábbi algoritmusok változtatás nélkül használtatók legyenek. Így nem megengedettek olyan $v \to w$ élek, amelyeknél $level(v) \neq level(w) + 1$.)

Ha az állapottér EDD-je által reprezentált függvényt $f_E(x_n, \ldots, x_1)$-nek tekintjük, akkor a távolságinformációk eldobásával ebből készített MDD-nek a következő f_M függvényt kell reprezentálnia:

$$f_M(x_n, \ldots, x_1) = \begin{cases} 0, \text{ ha } f_E(x_n, \ldots, x_1) = \infty \\ 1, \text{ ha } f_E(x_n, \ldots, x_1) < \infty \end{cases}$$

Az ennek megfelelően megtervezett és implementált konverzió a 7. algoritmus. Ez alulról felfelé haladva elkészíti az EDD-beli csomópontok megfelelőjét az MDD-ben, az élsúlyok eldobása után esetlegesen előálló azonos jelentésű csomópontokat pedig összevonja. Emellett a végtelen élsúlyt tartalmazó utakat a terminális nullába vezeti, míg a többit a terminális egybe az EDD-beli $\perp$ csomópont helyett.

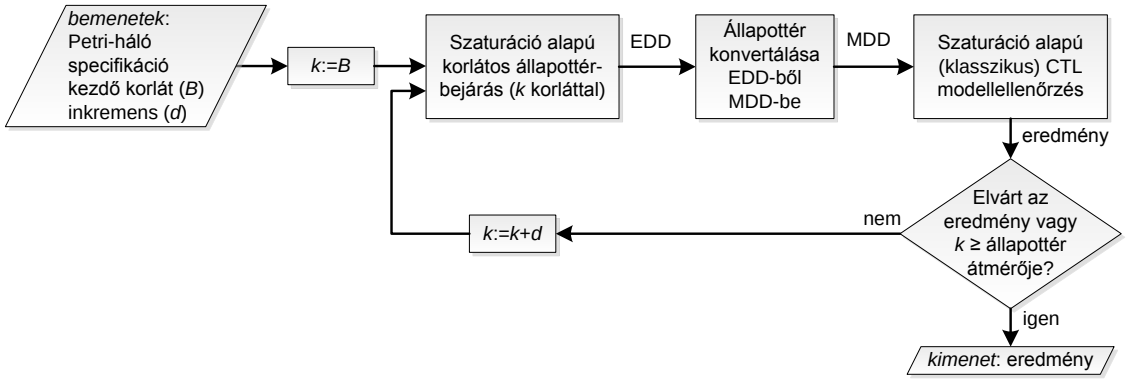
Az így megvalósított konverzió segítségével korlátosan felderített állapottereken is lehetőség nyílt szaturációs modellellenőrzés futtatására. A megoldás hatékonyságát a 7. fejezetben tárgyaljuk.

4.3.3. Iteratív megközelítés

Az előző fejezetekben ismertetett kiegészítések után még egy megoldandó probléma maradt: hogyan lehet meghatározni a korlátos állapottér-felderítés megfelelő korlátját? Tegyük fel, hogy egy $\mathbf{s}'$ állapotot keresünk. Legyen ennek távolsága a kezdőállapottól d, azonban ezt a priori nem tudjuk. Amennyiben a korlátos állapottér-generálás során d-nél kisebb korlátot alkalmazunk, $\mathbf{s}'$ nem lesz benne az állapottérben, így a modellellenőrzési fázisban sem találhatjuk meg. Ha viszont d-nél jelentősen nagyobb korlátot használunk, akkor feleslegesen sok állapotot kell felderíteni, ami csökkenti a megoldás hatékonyságát. A megfelelő korlát azonban jelentősen függ a modelltől és a vizsgálandó kifejezéstől egyaránt, így nehéz becslést adni rá.

Ezért munkánk során egy *iteratív eljárást* fejleszetettünk, amellyel a korlátot automatikusan és szisztematikusan növelve vizsgálhatóak korlátos állapottereken specifikációs kritériumok. Ez a megközelítés látható a 4.4. ábrán. Ennek során a kiértékelendő követelményt egy kezdő B távolságkorláttól indulva, d növekménnyel vizsgáljuk mindaddig, amíg a követelmény teljesülése vagy meghiúsulása megválaszolható a korlátos állapottér alapján vagy pedig a modell teljes állapottere felderítésre került (ebben az esetben a követelmény biztosan kiértékelhető).

Ezzel létrehoztunk a munkánk során egy olyan szaturációs megoldást a korábban leírt korlátos állapottér-generálás és a teljes állapottér felderítésén alapuló modellellenőrzés alapján, amely képes korlátos modellellenőrzést végezni Petri-hálókon, megteremtve a nagy komplexitású vagy végtelen állapotterű modellek szaturációs modellellenőrzésének lehetőségét.



4.4. ábra. Iteratív korlátos modellellenőrzés szaturáció alapú algoritmusokkal

4.4. A korlátos modellellenőrzés hatása a komplex rendszerek verifikációjára

Az itt leírt módszerek hozzájárulnak a komplex rendszerek nagyobb körének vizsgálhatóságához. A korlátos modellellenőrzés segítségével vizsgálhatóvá váltak a *végtelen állapotterű rendszerek*, amelyek analízisére a klasszikus szaturációs algoritmus elméleti szinten sem alkalmas kimerítő tulajdonsága miatt. Emellett hatékonyabban lehet véges, de nagy állapotterű összetett modellekben olyan kifejezéseket vizsgálni, amelyek eldöntéséhez elegendő az állapottér kis részének bejárása. Például ha egy vezérlőben hibát keresünk, nem feltétlenül szükséges a teljes állapottér bejárása, mivel a tesztelés korai fázisaiban tipikusan kis mélységben találhatunk hibát. Az ilyen ún. *sekély hibák* (shallow bugs) keresésére a korlátos módszerek különösen alkalmasak.

Bár a 4.2.2. fejezetben felvetett problémák megoldásával megteremtettük a szaturációs korlátos modellellenőrzés lehetőségét, méréseink alapján az algoritmusok további fejlesztésre szorultak. Ennek érdekében a korlátos modellellenőrzést továbbfejlesztettük az ún. vezérelt szaturáció használatával. Ez a fejlesztés kerül bemutatásra az 5. fejezetben.

5. fejezet

Vezérelt szaturációs algoritmus

A modellellenőrzés egyik sarkalatos pontja az, hogy milyen algoritmusokat használunk az egyes operátorok kiértékelésére. A korábban publikált szaturációs algoritmusok [9] egy lehetséges továbbfejlesztését közölte Zhao és Ciardo 2009-ben [28]. A módszerüknek a *vezérelt szaturáció* (constrained saturation) nevet adták.

Módszerük segítségével hatékonyabbá tehető az EF és EU operátorok (és így az ezekre épülő AG és AU operátorok) kiértékelése mind korlátos, mind teljes állapotteret felderítő modellellenőrzés esetén. Jelen fejezetben ezt az algoritmust mutatjuk be, illetve adaptáljuk a korlátos modellellenőrzésre.

5.1. A szaturációs modellellenőrzés továbbfejlesztési lehetőségei

Újabb szaturációs algoritmusok vizsgálatához az vezetett minket, hogy az EU kifejezések hatékonyságával nem voltunk elégedettek. Emellett azt is tapasztaltuk, hogy EF operátorok esetén is lecsökken a korlátos modellellenőrzés hatékonysága.

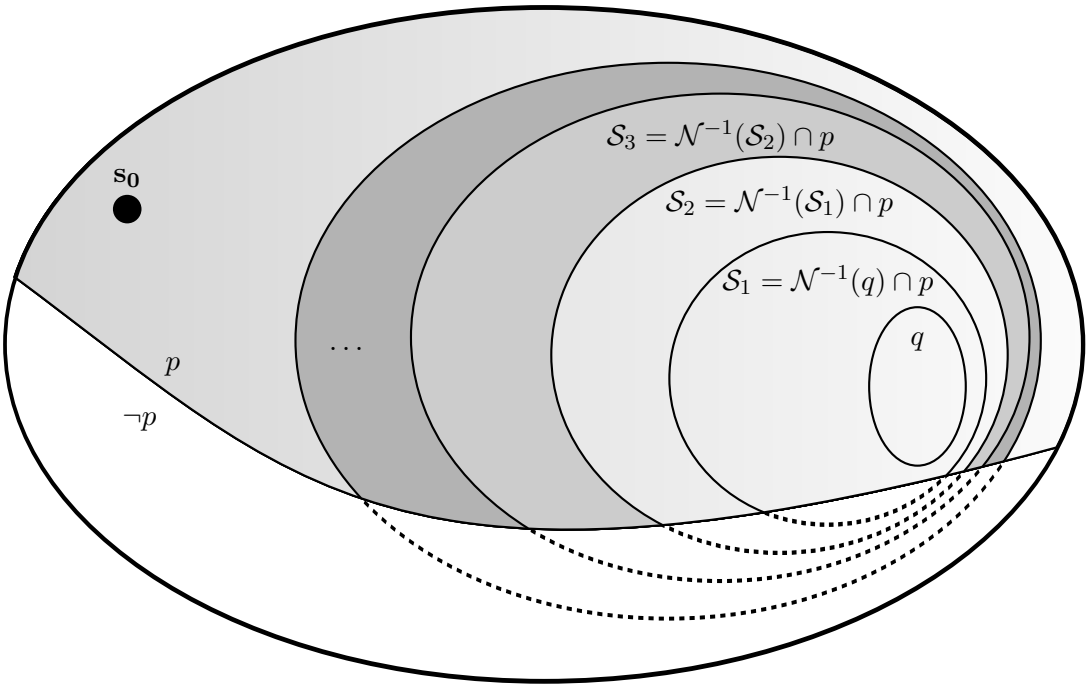
Ennek megértéséhez vizsgáljuk meg egy $\mathsf{E}\ p\ \mathsf{U}\ q$ kifejezés kiértékelését! Ahogyan az a 2.5. fejezetben olvasható, e kifejezés szemléletesen akkor teljesül, ha a kezdőállapotból tüzelésekkel eljuthatunk egy q állapotba (pontosabban a q állapothalmaz egyik állapotába) úgy, hogy közben kizárólag p halmazbeli állapotokba kerül a modell. Ennek vizsgálatát szemlélteti az 5.1. ábra, amely az állapotteret és az egyes állapothalmazok viszonyát mutatja, segítve a következőkben leírtak megértését.

A teljesülés vizsgálatának módja a következő: kiindulunk a q állapothalmazból, majd megvizsgáljuk, milyen állapotokból juthattunk el q-ba. Ez a $\mathcal{N}^{-1}(q)$ halmaz. Ebből azonban csak azok az állapotok érdekesek számunkra, amelyek szerepelnek p-ben, azaz a $\mathcal{S}_1 = \mathcal{N}^{-1}(q) \cap p$ halmaz. Ez után azt vizsgáljuk, hogy ebbe az $\mathcal{S}_1$ halmazba milyen állapotokból juthatunk. A $\mathcal{S}_1$ halmazba az előzőek alapján a $\mathcal{S}_2 = \mathcal{N}^{-1}(\mathcal{S}_1) \cap p$ állapotokból juthatunk. Ezt tovább folytatva előállítjuk az $\mathcal{S}_U = \mathcal{S}_1 \cup \mathcal{S}_2 \cup \cdots$ halmazt, amely megadja az összes olyan állapotot, amelyből q elérhető p-beli állapotokkal. Az $\mathsf{E}\ p\ \mathsf{U}\ q$ kritérium pontosan akkor teljesül, ha $\mathbf{s_0} \in \mathcal{S}_U$ [9].

Az EF kifejezések vizsgálata ehhez hasonló, mivel a 2.5. fejezetben leírtak alapján $\mathsf{EF}\ q \equiv \mathsf{E}\ [true\ \mathsf{U}\ q]$, tehát a különbség csak annyi, hogy nem szükséges a p-vel metszés az $\mathcal{S}_i$-k előállítása során.

Ezzel a megközelítéssel szaturációs módszerek esetén két probléma adódik:

- EU operátor kiértékelése esetén nagyon sokszor kell a p-vel metszés műveletet elvégeznünk. Valójában az állapothalmazok döntési diagramoknak felelnek meg, korábbi



5.1. ábra. Szemléltetés az $\mathsf{E}\ p\ \mathsf{U}\ q$ kifejezés kiértékeléséhez

tapasztalataink [13] alapján viszont a döntési diagramok metszés művelete igen költséges.

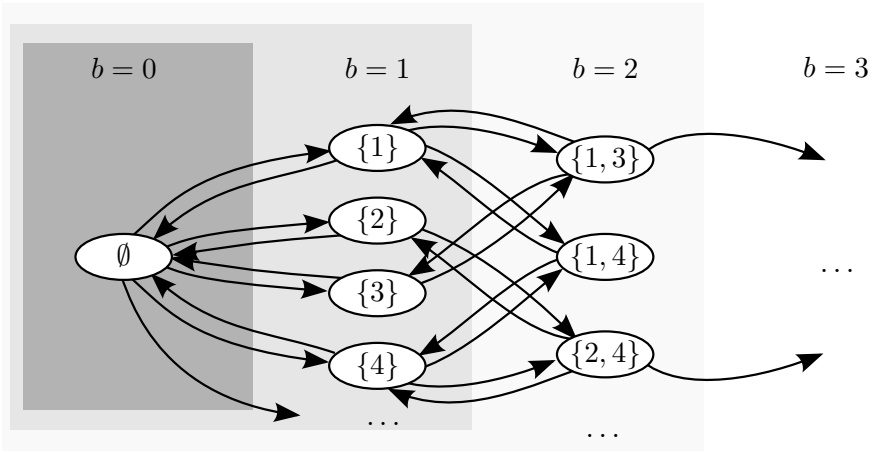
- Korlátos modellellenőrzésnél egy komplexebb probléma is felmerül az EF operátor kapcsán. Mivel az állapotátmenet-függvény komponensenként (részmodellenként) partícionált (Kronecker-mátrixos tárolás esetén), ezért a klasszikus szaturációs modellellenőrzés során bejárásra kerülnek olyan állapotok is, amelyek a modell és az állapotátmenet-függvény alapján elérhetők, azonban a megadott bejárási korláton kívül esnek, így a korlátos állapottérnek nem részei. Az ilyen állapotok bejárása felesleges, mivel eltávolításra kerülnek az algoritmus végén egy metszés művelettel, viszont jelentős overheadet okozhat. Ezt a problémát illusztrálja a 10. példa.

10. példa. *Tekintsük a 4. példában bevezetett (2.2. ábrán látható) egyszerűsített étkező filozófusok modellt! A modell globális állapotát az határozza meg, hogy mely filozófusok esznek és melyek gondolkoznak. Ennek megfelelően az egyes állapotokat az étkező filozófusok halmazával jelöltük. Kezdőállapotban mindegyik filozófus gondolkozik, ezért az evő filozófusok halmaza üres.*

Ha $b = 1$ korláttal felderítjük a modell $\mathcal{S}_1$ állapotterét, az alábbi halmazt kapjuk: $\mathcal{S}_1 = \{\emptyset, \{1\}, \{2\}, \{3\}, \ldots\}$. Mivel az atomi esemény a modellben egy filozófus állapotváltoztatása, ezért nyilvánvaló, hogy $b = 1$ korlát esetén pl. az $\{1, 3\}$ állapot nem lesz az állapottér része. Viszont a modell összes lokális állapotátmenete felderítésre kerül, külön-külön mindegyik filozófus mindegyik állapota ismertté válik.

Példaként vizsgáljuk ezen az $\mathcal{S}_1$ állapottéren az $\mathsf{EF}\ \{1\}$ kifejezést! Ennek során a klasszikus szaturációs algoritmus $\{1\}$-ből indulva felderíti azokat a globális állapotokat, amelyek az állapotátmenet-függvények alapján felderíthetők – jelen esetben ez a teljes $\mathcal{S}$ állapottér.

Ha n filozófus modelljét vizsgáljuk, a kezdőállapottól egy távolságra levő állapotok száma $n + 1$, viszont a teljes állapottér mérete $|\mathcal{S}| \approx 1,62^n$, így a példában jelentős felesleges számításigényt okoz a klasszikus szaturációs modellellenőrző algoritmus.



5.2. ábra. Az étkező filozófusok egyszerűsített modelljének állapottere

A vezérelt szaturáció lényege az, hogy a klasszikus szaturáció „lép és vág” jellegű működése helyett egy „vizsgál és lép” jellegű működést használnak. Azaz a hibás vagy szükségtelen lépéseket nem utólag, egy költséges vágás művelet segítségével távolítják el, hanem a lépés elvégzése előtt ellenőrzik, hogy a végrehajtandó lépés szabályos-e. Ezzel elkerülhető a nagy mennyiségű metszés művelet elvégzése, illetve a korlátos állapottér vizsgálatánál a bejárás megszorítható a korlátos állapottérben szereplő állapotokra.

5.2. Alkalmazás a nemkorlátos modellellenőrzésben

A munkánk során megvalósítottuk a [28]-ban javasolt vezérelt modellellenőrző algoritmust úgy, hogy az korlátos és nemkorlátos modellellenőrzésre egyaránt használható legyen. Az előbbiekben és [28]-ban leírtak alapján a munkánk során implementált vezérelt szaturációs algoritmusokkal az EU és EF operátorok (valamint az ezekre épülő AG és AU operátorok) kiértékelése nemkorlátos esetben is jelentősen gyorsítható.

Megjegyzendő, hogy az 5.1. fejezetben leírtakhoz képest a korábbi szaturációs megoldások tartalmaztak a metszés műveletek elkerülésére nézve optimalizációt: definiálták az események *biztonságos* részhalmazát, amelyek garantáltan nem vezetnek ki a p halmazból, így utánuk a metszet művelet alkalmazása nem szükséges. Azonban a többi esemény okozta metszés is túl nagy erőforrásigényű, valamint az algoritmus kezdetén az egyes események biztonságosságának megállapítása is jelentős számítási kapacitást foglalt le. (A pontos megvalósításról részletesen olvashat korábbi implementációnk ismertetésében [13].) A vezérelt szaturáció eliminálja a metszet műveletet, mivel az állapottérben nem történhet olyan lépés, amely nem p-beli állapothoz vezetne.

A megvalósítás alapját a következőkben is olvasható, [28]-ban közölt pszeudókódok adták. Maga a vezérelt szaturációs algoritmus belépési pontja a *ConsSaturate* függvény (8. algoritmus), amelynek meg kell adni a szaturálandó csomópontot (másképp az állapothalmazt, amelyből a bejárás indul) és a megkötést (azaz azt az állapothalmazt, amelyből nem léphet ki a bejárás során). Egy $\mathsf{E}\ p\ \mathsf{U}\ q$ kifejezés kiértékelésére a *EUConsSat* függvény szolgál (10. algoritmus).

A módszer lényegi eleme, a „vizsgál és lép” megközelítés két helyen figyelhető meg a pszeudókódokban. Egyrészt a *ConsSaturate* függvény 8. sorában: ez alapján ha $a[i] = \mathbf{0}$, azaz az a csomópont i. gyereke a terminális nullába mutat, tehát a kényszert jelentő döntési diagramban a vizsgált állapot nem szerepel, akkor az $s[i]$ változatlan marad, a kényszerben nem szereplő állapotok nem kerülnek bejárásra. A másik vizsgálat a *ConsSaturate* függvény 16. sorában és a *ConsRelProd* függvény 8. sorában található. Itt amennyiben

Algoritmus 8: ConsSaturate

```
bemenet: a, s : csomópont        // a: megkötés, s: szaturálandó csomópont
kimenet : csomópont
1  if vezérelt szaturációs cache tartalmaz t értéket (a, s)-hez then
2  |  return t;
3  end
4  l ← s.level;
5  t ← NewNode(l);
6  r ← N_l^{-1};
7  foreach i ∈ S_l : s[i] ≠ 0 do
8  |  if a[i] ≠ 0 then
9  |  |  t[i] ← ConsSaturate(a[i], s[i]);
10 |  else
11 |  |  t[i] ← s[i];
12 |  end
13 end
14 repeat
15 |  foreach i, i' ∈ S_l : r[i][i'] ≠ 0 do
16 |  |  if a[i'] ≠ 0 then
17 |  |  |  u ← ConsRelProd(a[i'], t[i], r[i][i']);
18 |  |  |  t[i'] ← Union(t[i'], u);
19 |  |  end
20 |  end
21 until t nem változik;
22 t ← CheckIn(l, t);
23 (a, s) kulcshoz t érték beszúrása a vezérelt szaturációs cache-be;
24 return t;
```

a vizsgált állapot nem eleme a megkötést hordozó döntési diagramnak, akkor a rekurzív végrehajtás megszakad és a függvények a terminális nullával térnek vissza, azaz a bejárás nem folytatódik olyan állapotokra, amik nem teljesítik a kényszert.

5.3. Vezérelt szaturáció alkalmazása korlátos modellellenőrzésben

A korlátos szaturációs modellellenőrzés során előkerülő problémák miatt korábbi megoldásunk nem volt univerzálisnak tekinthető. Jelen munkánk során azonban továbbfejlesztettük az algoritmust, és a vezérelt szaturáció alkalmazásával a modellellenőrző algoritmus hatékonyságát szignifikánsan megnöveltük. A vezérelt szaturáció használata korlátos modellellenőrzésre egy új algoritmust eredményez, amelyet a dolgozatunk előtt nem publikáltak.

Ahogyan azt a 10. példa is szemléltette, a konkrét problémát az jelenti, hogy a megelőző állapotok számítása során olyan állapotok is vizsgálatra (majd kimetszésre) kerülnek, amelyek valójában nem részei az $\mathcal{S}$ felderített állapottérnek (például ez merül fel az $\mathsf{EF}\ p$ kifejezés kiértékelése során).

Ennek elkerülése érdekében az $\mathsf{EF}\ p$ kifejezés kiértékelésekor megkötést tehetünk az állapotokra: csak olyan $\mathbf{s}$ állapotokat vizsgálunk meg, amelyekre $\mathbf{s} \in \mathcal{S}_b$ ($\mathcal{S}_b$ a korlátos

Algoritmus 9: ConsRelProd

bemenet: a, s, r : csomópont
kimenet : csomópont

```
1  if s = 1 ∧ r = 1 then return 1;
2  if RelProd cache tartalmaz t értéket (a, s, r)-hez then
3  |  return t;
4  end
5  l ← s.level;
6  t ← 0;
7  foreach i, i' ∈ S_l : r[i][i'] ≠ 0 do
8  |  if a[i'] ≠ 0 then
9  |  |  u ← ConsRelProd(a[i'], s[i], r[i][i']);
10 |  |  if u ≠ 0 then
11 |  |  |  if t = 0 then t ← NewNode(l);
12 |  |  |  t[i'] ← Union(t[i'], u);
13 |  |  end
14 |  end
15 end
16 t ← ConsSaturate(a, CheckIn(t));
17 (a, s, r) kulcshoz t érték beszúrása a RelProd cache-be;
18 return t;
```

Algoritmus 10: EUConsSat

bemenet: p, q : csomópont // E p U q kiszámítása
kimenet : csomópont

```
1  p ← Intersect(p, S);        // S: elérhető állapotok halmaza
2  q ← Intersect(q, S);
3  s ← ConsSaturate(p, q);
4  return s;
```

Algoritmus 11: EFBoundedConsSat

bemenet: p : csomópont // EF p kiszámítása korlátos állapottér esetén
kimenet : csomópont

```
1  p ← Intersect(p, S);
2  s ← ConsSaturate(S_b, p);        // S_b: elérhető állapotok halmaza
3  return s;
```

felderített állapottér). Azaz a vizsgálat ekvivalens lesz az $\mathsf{E}\ \mathcal{S}_b\ \mathsf{U}\ p$ kifejezés vizsgálatával. Ezt valósítja meg az *EFBoundedConsSat* függvény (11. algoritmus).

Ebből következően a probléma kiküszöbölésére a korábbiakban is használhattuk volna a klasszikus EU operátor implementációnkat. Bár így kevesebb állapotot vizsgált volna meg a függvény, mint a klasszikus EF operátor esetén, azonban az EU operátorban szereplő műveletek, eseménybesorolások és egyéb, az EF operátor szempontjából nem szükséges lépések jelentős lassulást okoztak volna.

Ahogyan az a 7.2. fejezetben olvasható mérési eredmények megmutatták, a korlátos modellellenőrzés elérhetőségi vizsgálata jelentősen hatékonyabbá vált a vezérelt szaturáció alkalmazásának köszönhetően. Tudomásunk szerint a mi munkánk eredményezte az első vezérelt szaturációs korlátos modellellenőrző algoritmust.

6. fejezet

Színezett Petri-hálók analízise szaturációs alapokon

A színezett Petri-hálók az egyszerű Petri-hálók kiterjesztései abból a célból, hogy a meglévő modelljeinket kompaktabb módon tudjuk ábrázolni, továbbá képesek legyünk olyan rendszerek modellezésére is, amelyekhez az egyszerű Petri-hálók nem nyújtanak megfelelő támogatást (2.2. fejezet). Színezett hálók segítségével a modellezés során különféle adatszerkezeteket használhatunk, amelyekkel paraméterezhető modellek készítésére is lehetőségünk nyílik. Amellett, hogy a színezett Petri-hálók (az egyszerű Petri-hálókhoz hasonlóan) hatékonyan használhatók aszinkron rendszerek modellezésére, új, bonyolultabb logikájú modellek építését is támogatják. Erre a megváltozott modellező nyelvre viszont nem alkalmazható a szaturáció eddig megismert változata az újonnan bevezetett elemek miatt.

E területet érintő munkánk során célunk kettős volt. Először kidolgoztuk a színezett Petri-hálók szaturációs ellenőrzését, majd megvizsgáltuk a gyakorlati alkalmazhatóságát. A most következő fejezet felépítése a következő: a 6.1. fejezetben áttekintjük, hogy a színezett Petri-hálók állapotátmeneti relációinak tárolásához milyen új módszereket kell bevezetnünk. A következő, 6.2. fejezetben bemutatjuk az általunk kidolgozott szaturációs algoritmust a színezett hálókra. Végül a 6.3. fejezetben megvizsgáljuk, hogy a színezett hálók analízisét hogyan tudjuk hatékonyabbá tenni az állapotátmeneti relációk leképzésének módosításával.

6.1. Állapotátmenet leképezés

Azt már láttuk a 9. példában, hogy MDD-kkel bármilyen állapotátmenet-függvényt tudunk kódolni, ami Kronecker-mátrixok segítégével leírható. Most azt mutatjuk meg, hogy az MDD-kkel kódolható állapotátmeneti relációk köre ennél jóval szélesebb, ugyanis tetszőleges véges függvény leírható döntési diagramok segítségével. A Kronecker-mátrix alapú leképezésnél az egyes részmodellekhez tartozó mátrixok függetlenek, így pedig egymástól elkülönülve tárolhatók. Ezzel szemben egy nem Kronecker-konzisztens modell esetében az állapotátmenet-tárolás helyességének megőrzése céljából az átmenetfüggvényeket nem partícionálhatjuk vertikálisan. Ez lehetővé teszi olyan állapotátmeneti függvények reprezentálását is, ahol egy reláción belül egy átmenet végrehajthatósága függ egy felsőbb szinten végrehajtott állapotátmenettől is. Ennek szemléltetését egy példa segítségével fogjuk megtenni (11. példa).

11. példa. *Tekintsük a 6.1(a) ábrán látható színezett Petri-hálót. A P_1 és P_3 helyek színosztálya legyen boolean (B), ami true vagy false értékeket tud felvenni. A P_2 hely C szín-*

osztályának értékészlete legyen $\{A, B\}$*. A modellt dekomponáljuk úgy, hogy az 1. állapotváltozó (részmodell) a* P_1 *hely állapotát, a 2. állapotváltozó a* P_2*, a 3. állapotváltozó pedig a* P_3 *hely állapotát kódolja. A kódolásnak megfelelően mindhárom részmodellnek 4–4 lokális állapota van, amelyeket a 6.1(b), a 6.1(c) és a 6.1(d) táblázatok írnak le. Kezdetben a* P_1 *helyen 1 db true és 1 db false token, a* P_2 *helyen 1 db A és 1 db B értékű token található, a* P_3 *hely pedig üres. Az áttekinthetőség kedvéért az elérhetőségi gráfot feltüntettük a 6.1(e) ábrán.*

A T_1 *tranzíció tüzelése után a* P_3 *hely állapota a* b_1 *változó lekötésétől függ. A lekötés a* $(P_1, P_2, P_3) = (0, 0, 0)$ *állapotban lehet* $b_1 = true$ *vagy* $b_1 = false$*. Ha például a* $b_1 = true$ *lekötés valósul meg, akkor az 1. részmodell* 0*-ból* 1 *állapotba, miközben a 3. részmodell is* 0 *állapotból* 1 *állapotba megy át. Az állapotátmeneteket a 6.1(f) ábrán látható MDD kódolja. (Itt az MDD-k korábbi definíciójával szemben a csomópontokban nem a szint sorszámát, hanem a részmodell azonosítóját, vagyis egy állapotváltozót tüntettük fel. A vesszős szintek a tüzelés utáni állapotválozókat jelölik.) A döntési diagram alapján megállapítható, hogy a modell nem Kronecker-konzisztens, ugyanis a teljes állapotátmeneti függvény nem egyezik meg a lokális állapotátmeneti függvények Descartes-szorzatával.*

Végül figyeljük meg, hogy mivel szimbolikus algoritmust használunk, amely előretekintően deríti fel az állapotátmeneteket, a végső állapotátmeneti relációba olyan átmenetek is belekerülnek, amelyeknek a kiinduló állapotai például nem is fordulnak elő az állapottérben.

6.1.1. Konjunktív módon partícionált állapotátmeneti relációk

A nem Kronecker-konzisztens modellek állapotátmenet-relációinak felépítése költséges művelet. Minden egyes új állapot felderítésekor elvégezzük az összes lehetséges állapotátmenet előállítását ebből az állapotból, minden lehetséges lokális állapottal vett kombinációra. Ha azonban a modell tulajdonságaira is figyelünk és az eseményekhez tartozó állapotváltozók szerepét is számításba vesszük a tüzelésekben, hatékonyabban elő tudjuk állítani ezeket a relációkat. Ezt a megoldást [11]-ben mutatták be, általános állapotátmenet-relációk hatékony előállítására.

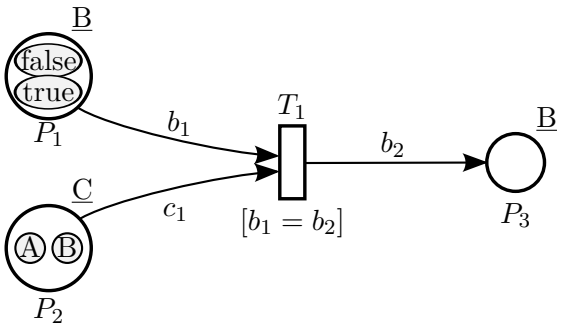
Az állapotátmenetek diszjunktív partícionálását, azaz a tranzíciók mentén való felbontását már láttuk a 2.6.1. fejezetben. A diszjunktív partícionálás során minden $\alpha \in \mathcal{E}$ eseményhez hozzárendelünk egy $\mathcal{N}_\alpha$ állapotátmeneti függvényt, amely az $\mathcal{N}$ állapotátmeneti függvény α eseményre vonatkozó részét jelöli. Ennek megfelelően a teljes állapotátmenet-reláció előáll: $\mathcal{N} = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha$. A továbbiakban minden diszjunktív részt felbontunk konjunktív módon is. Kronecker-konzisztens modell esetén ezt minden részmodell mentén megtehetjük, azaz

$$\mathcal{N}_\alpha = \bigwedge_{k=1}^{K} \mathcal{N}_{k,\alpha},$$

ahol K a részmodellek száma. Ez azért tehető meg, mert ez pontosan a Kronecker-konzisztencia feltételének felel meg.

Nem Kronecker-konzisztens esetben viszont az állapotátmeneti reláció nem feltétlenül bontható K részre, mivel a komponensek bizonyos részhalmazai α kontextusában együtt kezelendők a színezett Petri-hálók szemantikája által definiált kényszerek miatt. Ilyen kényszerek az élkifejezések és őrfeltételek által jelentett megszorítások. Az előbbieknek megfelelően az $\mathcal{N}_\alpha$ függvényt $C_{\alpha,1}, C_{\alpha,2}, \ldots, C_{\alpha,L}$ részfüggvényekre bontjuk, amiket *konjunktoknak* nevezünk. A felbontást úgy kell elkészíteni, hogy az alábbi feltételek együttesen teljesüljenek:

- az összefüggő részhalmazok egy konjunktban szerepelnek



(a) A vizsgált szenezett Petri-háló modell

i_1	P_1
0	{1'false,1'true}
1	{1'false}
2	{1'true}
3	{}

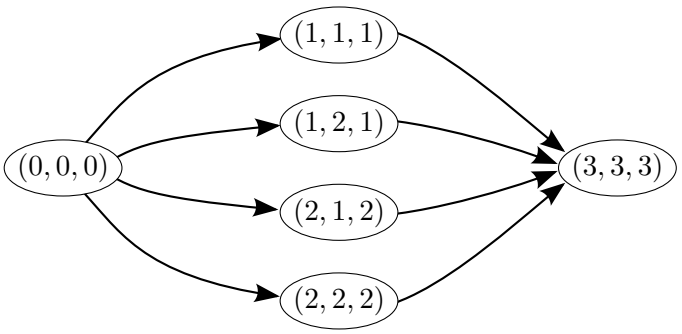
(b) Állapotok kódolása az 1. részmodellben

i_2	P_2
0	{1'A,1'B}
1	{1'A}
2	{1'B}
3	{}

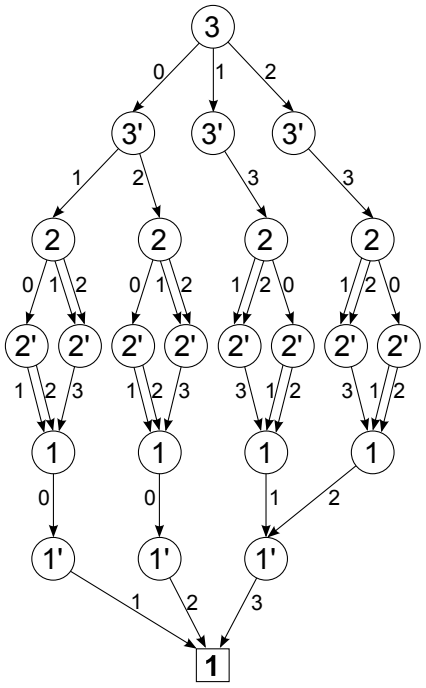
(c) Állapotok kódolása a 2. részmodellben

i_3	P_3
1	{}
1	{1'true}
2	{1'false}
3	{1'false,1'true}

(d) Állapotok kódolása a 3. részmodellben



(e) Állapottér (szimbolikus elérhetőségi gráf)

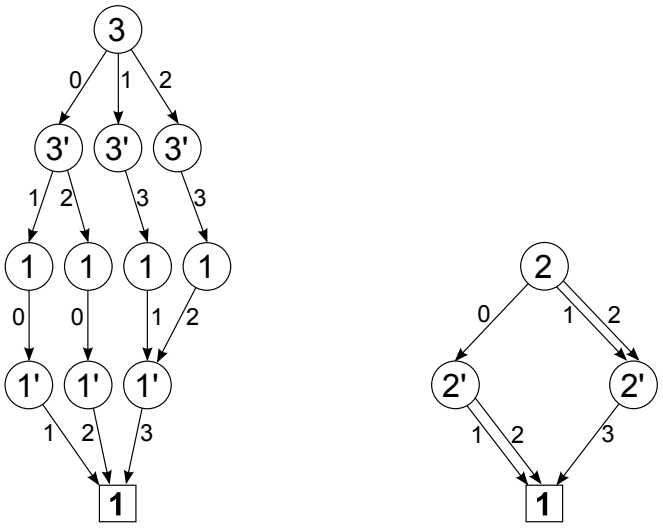


(f) Next-state függvény

6.1. ábra. Példa MDD-vel történő állapotátmenet-kódolásra

- minden $\mathcal{N}_{k,\alpha}$ részfüggvény pontosan egy konjunkt felépítésében játszik szerepet
- $\mathcal{N}_\alpha = \bigwedge_{l=1}^{L} C_{\alpha,l}$

12. példa. *Tekintsük a 11. példa modelljét, az ott definiált részmodellekre bontást és állapotkódolást. Az őrfeltétel $[b_1 = b_2]$ megkötése miatt a P_1 és P_3 helykhez tartozó részmodelleket együtt kell kezelnünk az átmeneti relációk tárolásakor. A 6.1(f) ábrán az MDD struktúrájából látszik, hogy a 2 és 2′ szintek redundanciát hordoznak. Az előbbiek miatt a T_1 tranzícióhoz tartozó állapotátmeneti reláció 2 konjunktív részre bontható. Az egyik konjunkció az 1. és 3. részmodellhez (6.2(a) ábra), a másik konjunkció a 2. részmodellhez fog tartozni (6.2(b) ábra). Azt kell még megfigyelnünk, hogy a két MDD metszete pontosan a 6.1(f) ábrán látható MDD-t adja.*



(a) Az 1. és 3. részmodellhez tartozó konjunkt rész

(b) A 2. részmodellhez tartozó konjunkt rész

6.2. ábra. Konjunktív módon partícionált állapotátmeneti relációk

6.2. Szaturáció alapú vizsgálat megvalósítása

A 2.6. fejezetben bemutattuk a szaturációs algoritmus egyszerű Petri-hálókra alkalmazható változatát. Bizonyos átalakításokkal az algoritmus alkalmazható színezett Petri-hálókra. A következőkben ezeket fogjuk áttekinteni.

A szaturációs algoritmus bemenetként megkapja a szimbolikusan leképezett dekomponált modellt, valamint a magas szintű események halmazát. Ezután a szaturáció egy speciális iterációs stratégia során felderíti az állapotteret és előállítja a szimbolikus reprezentációját. A színezett Petri-hálók dekompozíciója és változókra való leképzése az egyszerű Petri-hálókéhoz hasonlóan elvégezhető azon az áron, hogy az adatstruktúrák nagyobb lokális állapotteret fognak eredményezni. Mivel a színezett Petri-hálóknak bonyolult, komplex az állapotátmenet-függvényük, amely jellemzően nem Kronecker-konzisztens, a szaturációs algoritmus által használt állapotátmenet-relációkat kell adaptálni az új formalizmushoz. Megfelelő állapotátmenet-reláció esetén a szaturációs algoritmus iterációs sorrendje helyes marad, és eredményül a színezett Petri-háló szimbolikus állapottér-reprezentációját fogja adni.

Korábban a 3.2 fejezetben ismertettük, hogy a döntési diagramok hogyan alkalmazhatók a megváltozott állapotátmeneti-függvények tárolására és kezelésére. A 6.1. fejezetben kitértünk azok konjunktív partícionálására is, amelynek kidolgoztuk a színezett Petri-hálókra való adaptálását. Ezt a 6.2.1. részben mutatjuk be. Az új állapotokat felderítő algoritmusban szükséges változtatások ismertetése a 6.2.2 részben található.

6.2.1. Konjunktív partíciók kialakítása

Az állapotátmeneti függvény konjunktív-diszjunktív partícionálásával az a célunk, hogy az állapotátmeneteket részekre bontsuk és ezeket a kis részeket külön tudjuk kezelni. A diszjunktív partícionálást egyszerűen megtehetjük, ha az állapotátmeneti relációt események szerint felbontjuk. Ezt követően a kapott részeket felosztjuk további, úgynevezett konjunktív relációkra.

A konjunktív részekre bontó algoritmus működése röviden a következő. A függvény bemenő paramétere egy e esemény, amely esemény dekomponált állapotátmeneti relációit szeretnénk előállítani. Kezdetben minden részmodell, amelyet az e esemény befolyásol, egy-egy külön halmazba kerül. Ezután megvizsgáljuk, hogy az élkifejezések és őrfeltételek milyen kényszereket állítanak. Hogy az állapotátmeneti relációk helyességét biztosítsuk, a felmerülő kényszereket együtt kell kezelni, azaz egy halmazba kell tenni a hozzájuk kapcsolódó állapotváltozókat. Ezek alapján az alábbi két esetben össze kell vonni a megfelelő halmazokat:

- ha találunk két olyan élet, amelyek a vizsgált tranzícióhoz kapcsolódnak, élkifejezéseik azonos változókat tartalmaznak, valamint az élek végpontjaiban levő helyeknek megfelelő részmodellek különböző halmazokhoz tartoznak,
- ha az őrfeltételben két változó valamilyen relációban áll egymással.

A most bemutatott új partícionálást felhasználva az egyes konjunktív részek egymástól függetlenül építhetők a szaturáció során, így kihasználva a modellből nyert információkat.

6.2.2. Új állapotok felderítése

Az egyszerű Petri-hálók szaturációs algoritmusához képest színezett hálók esetén az új állapotok, illetve állapotátmenetek felderítése jelent még változást. Az egyes $e \in \mathcal{E}$ eseményekhez az $\mathcal{N}_e$ állapotátmeneti függvényeket úgy kapjuk meg, hogy a hozzájuk tartozó konjunkt részeket egymástól függetlenül építjük, majd képezzük azok metszetét. Ennek következménye, hogy valamely konjunktív rész megváltozása esetén mindig frissíteni kell a teljes $\mathcal{N}_e$ átmeneti relációt annak olvasása előtt.

Az algoritmus bemenő paramétere k és i, vagyis a k. szinten szeretnénk az i. állapotból felderíteni az állapotátmeneteket. Ekkor a következő lépéseket hajtjuk végre:

1. Megkeressük azokat az e eseményeket, amelyek befolyásolják a k. szintet.
2. Minden e eseményhez megkeressük azt a C konjunktív részt, amely a k. szintet befolyásolja.
3. Felderítjük az új állapotátmeneteket az i. állapotból és eltároljuk a C konjunkció által reprezentált állapotátmeneti relációban.
   (a) A nem lekötött élváltozókhoz megkeressük az összes lehetséges behelyettesítést a változók színosztályának értékkészlete alapján;
   (b) megnézzük, hogy a tranzíció tüzelhet-e a megtalált változóbehelyettesítéssel;

(c) a felderített átmenetet elkódoljuk a C-hez tartozó döntési diagramban.

4. Ha felderítettünk új átmenetet, akkor az e eseményhez tartozó döntési diagramot frissítjük a konjunkt részek metszetének képzésével.

Az így kapott algoritmus részenként állítja elő az állapotátmeneti relációt, kisebb részekre bontva a frissítési műveleteket.

6.3. Állapotátmenetek hatékony kezelése

6.3.1. Felmerülő problémák

A színezett Petri-hálók tulajdonságai jelentősen eltérnek az egyszerű Petri-hálókétól. Ennek az az oka, hogy tömör formában tudnak akár hatalmas Petri-hálókat reprezentálni azáltal, hogy a strukturális elemeket adatszerkezetekkel helyettesítik. Ilyenkor azonban az állapottér-felderítés során nem tudjuk kihasználni a komponensek aszinkron viselkedését, sőt a lokális aszinkronitás miatt sok redundáns műveletet végzünk, amely az algoritmus szimbolikus voltából ered. Mivel egy színezett Petri-hálóbeli hely a vele ekvivalens működésű színezetlen modellben sok színezetlen helynek felel meg, ezért a színezett hálók esetén a lokális állapotterek jellemzően sokkal nagyobbak és bonyolultabbak. Ezek kezelése sok erőforrást igényel. Hasonlóan, a színezett tranzíciók is logikailag több színezetlen tranzíciót reprezentálnak.

Ez a nagyfokú tömörség azt eredményezi, hogy kevesebb döntési diagram változóval kódoljuk ugyanazt az állapothalmazt, amelynek következménye, hogy az állapottér-reprezentáció kevesebb redundanciát tartalmaz, így pedig az állapotteret kódoló döntési diagram mérete megnő. Ez kevésbé hatékony tárolást jelent, amely bizonyos esetekben konvergál az explicit állapottároláshoz. Mivel az állapottér reprezentációval kapcsolatos problémák az imént említett tömörségből fakadnak, ezért erre a kihajtogatás (színosztályok felbontása az értékkészlet elemeit reprezentáló helyekre) és egyéb, a dolgozatunkban nem tárgyalt dekompozíciós módszerek jelenthetnek megoldást. Mivel a szaturáció a lokálisan szinkron, globálisan aszinkron rendszerek esetén hatékony, ezért a nagy lokális aszinkronitással rendelkező rendszerek hatékony analízisét dolgozatunkban nem vizsgáljuk. Munkánk során egy másik irányba indultunk el; az adatfüggő állapotátmenet-relációk hatékony kezelésére kerestünk megoldást.

6.3.2. Hatékonyságnövelő megoldások

Az előző fejezetben láttuk, hogy a bemutatott problémák miatt a megvalósított színezett Petri-hálós szaturáció nem volt kellően hatékony, mert a konjunktív felbontás ellenére is hatalmas lokális állapotátmeneteket kellett felderíteni.

Tekintsük példaként az étkező filozófusok paraméterezhető, színezett Petri-hálós modelljét, amely a honlapunkon elérhető [29]. Itt bonyolult logika határozza meg a kényszereket a változók között, így a változók között levő összefüggések miatt nem dekomponálható a reláció az eddigi módszerrel. Mi ezt a problémát kívántuk orvosolni. A kidolgozott módszer a színezett Petri-hálók szemantikáját kihasználva dekomponálja az állapotátmeneti relációkat, így hatékonyabbá téve azok építését.

Az új algoritmus fő újítása, hogy az állapotátmenetet tároló döntési diagramokban a lehetséges átmenetek mellett elkódolja az átmenethez szükséges változókötéseket is az alábbi módon: minden tokenváltozó számára egy-egy új szintet hozunk létre a döntési diagramban az átmeneteket kódoló szintek fölött. A változók értékkészleteit szimbolikusan elkódoljuk, és ezek fogják alkotni az egyes változószintek lokális állapotterét. Ha egy új állapotátmenetet derítettünk fel, akkor az eddigi kódolás mellett a tüzelésnél alkalmazott

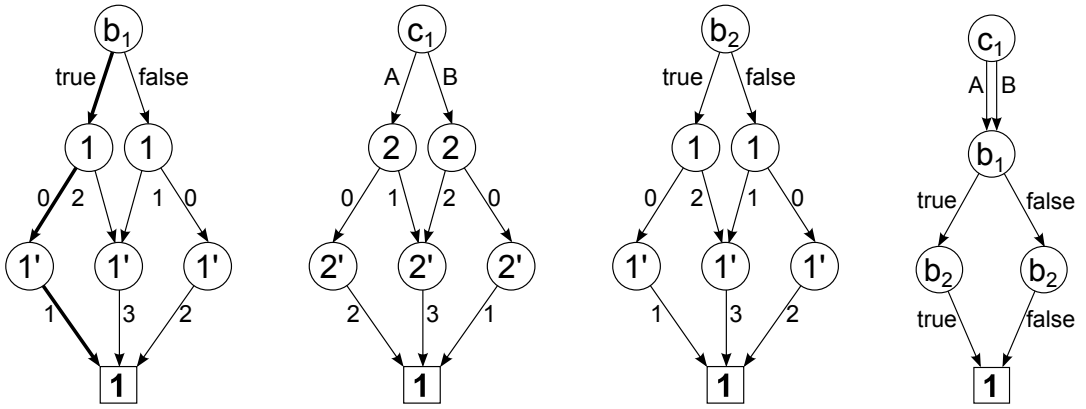
változólekötésnek megfelelő szinteken is tárolni kell a megfelelő értékeket. Az így kapott döntési diagramok a megfelelő finomságú partícionáláson túl már az élkifejezések jelentette kényszereket is teljesítik. Azonban, ha őrfeltételünk is van, akkor az ezek jelentette megszorításokat is szem előtt kell tartani. Ennek megadására egy új konjunkt relációt hoztunk létre. Ebben az előbb ismertetett módon azokat a lehetséges változólekötéseket tároljuk, amelyek teljesítik az őrfeltétel kényszereit. Ennek a megoldásnak az előnye, hogy ismerve az egyes változók típusának értékkészleteit a kényszereket offline tudjuk számolni. Végül a $\mathcal{N}_e$ reláció frissítésekor a metszetképzést elvégezzük az őrfeltételt kódoló döntési diagrammal is, hogy csak a megengedett változólekötéseket használó tüzelések hatásait tartsuk meg.

A továbbiakban még egy problémát kell megoldani az új frissítési mechanizmust illetően: az $\mathcal{N}_e$ relációk frissítésekor a metszetképzés eredménye tartalmazza a tokenváltozólekötésekhez tartozó döntési diagram szinteket és lekötéseket is, tehát ezeket utólag el kell távolítani. A változólekötéseket is tartalmazó relációt jelöljük $\mathcal{N}'_e$-vel, a tokenváltozókat kódoló szinteket pedig v_i-vel. Ekkor $\mathcal{N}_e = \forall i \exists v_i \mathcal{N}'_e$, azaz az állapotátmeneti reláció az összes kényszer által reprezentált állapotváltozókra vett reláció lesz, az élfeltételek nélkül.

13. példa. *Tekintsük a már többször vizsgált modellt (6.1(a) ábra). A bemutatott módszernek megfelelően az állapotátmeneti relációt most bontsuk 3 részre, és mindegyikben kódoljuk el az átmenetekhez szükséges változólekötéseket is. Az így kapott részállapotátmeneti relációkat a 6.3(a), a 6.3(b) és a 6.3(c) ábrák mutatják.*

Vizsgáljuk meg azt az állapotátmenetet, amikor a P_1 hely kezdő állapotában elveszünk egy true értékű tokent, azaz úgy tüzelünk a T_1 tranzícióval, hogy a b_1 változó értékét lekötjük true-ra. Ezt az átmenetet a szemléltetés kedvéért kiemeltük a 6.3(a) ábrán.

Az algoritmusnak megfelelően megadjuk továbbá az őrfeltétel kényszereit reprezentáló döntési diagramot is (6.3(d) ábra). Az ábrán azt látjuk, hogy a c_1 változó értékére nincs megkötés, ellenben a kényszereknek megfelelően, b_1 változó értéke mindig megegyezik a b_2 változó értékével.



(a) Az 1. részmodellhez tartozó konjunkt rész

(b) A 2. részmodellhez tartozó konjunkt rész

(c) A 3. részmodellhez tartozó konjunkt rész

(d) Őrfeltételt kódoló konjunkt rész

6.3. ábra. Változólekötéseket is tároló konjunktív állapotátmeneti relációk

A kidolgozott módszer előnye, hogy sikerült a tranzíció őrfeltételének kényszereit offline kiszámolni, amelyet így nem kell frissíteni menet közben, ráadásul ezzel sikerült gyorsítani az állapotátmenet-felderítést is. Ezen felül lokálisan vizsgálhatóvá tettük a változólekötések helyekre vett hatásait és így egy, a korábbinál hatékonyabb felderítést tudunk megvalósítani.

6.3.3. Összegzés

A fejezetben bemutattuk, hogyan fejlesztettük tovább a szaturációs algoritmust, hogy színezett Petri-hálók analízisére is alkalmazható legyen. Először az állapotátmenetek tárolóját Kronecker-mátrixról döntési diagramra cserélve kialakítottunk egy kezdeti algoritmust. Ezt implementálva a PetriDotNet keretrendszerbe, majd méréseket végezve arra a megállapításra jutottunk, hogy a színezett Petri-hálók megváltozott jellemzőkkel bíró állapotátmenet-relációinak kódolása még nem kellően hatékony. Később egy alapjaiban új módszert dolgoztunk ki, amely szerint az állapotátmenetekkel együtt elkódoljuk a tüzelésben szerepet játszó tokenváltozók értékeit is. Ezzel a rugalmas megközelítéssel elértük, hogy a modelljeinket kellően finoman partícionálva alkalmazhassunk a szaturációs algoritmust. Ez a továbbfejlesztés tette lehetővé azt, hogy való, ipari környezetből származó modelleket tudjunk vizsgálni (7.4. fejezet).

7. fejezet

Eredmények

Ebben a fejezetben a korábbiakban bemutatott fejlesztések hatékonyságát kívánjuk mérési eredményekkel alátámasztani. A 7.1. fejezetben az 5. fejezetben bemutatott vezérelt szaturációs algoritmus eredményei olvashatók teljes állapottér-bejáráson alapuló modellellenőrzés esetén. A 7.2. fejezetben a 4. fejezetben leírt iteratív korlátos modellellenőrzés gyorsító hatásai, illetve a korlátos állapottér-felderítés vezérelt szaturációs modellellenőrzéssel kiegészített változatának eredményei olvashatók. A színezett Petri-hálók analízisének eredményei a 7.3. fejezetben találhatók. Végül a 7.4. fejezetben egy valós, a Paksi Atomerőműben üzemelő biztonsági funkció verifikációja során szerzett tapasztalatainkat írjuk le.

A méréseket a következő konfiguráción végeztük: Intel L5420 2,5 GHz processzor, 8 GB memória, Windows Server 2008 R2 (x64) operációs rendszer, .NET 4.0 futtatókörnyezet. A mért időadatokba az állapottér-generálás és a modellellenőrzés idejét is beleszámítottuk, azonban a mérések nem foglalják magukban a hálók heurisztikus dekompozíciójának futásidejét, ahol ezt használtuk. Azokban az esetekben, ahol a táblázatban az időadat helyett „—” szerepel, a mérés futásideje meghaladta az 1200 másodpercet vagy a memóriafoglalás a 6 GB-ot.

A mérések során használt modellek forrásai: [14, 10, 8, 17, 22]. A modellek részletes ismertetése terjedelmi okokból a dolgozatban nem lehetséges, azonban a felhasznált modellek PNML formátumban [26] letölthetők a *PetriDotNet* keretrendszer honlapjáról [29]. A felhasznált modelleket automatikus generáló programmal állítottuk elő, így a fájlokban az egyes részmodellek egymás után következnek.

A mérések során a fejlesztések eredményeit a korábbi megvalósításainkkal vetettük össze. Ezek hatékonyságáról már írtunk korábbi munkáinkban [13, 16]. A méréseket az elméleti háttérként használt irodalom [9, 27, 28] méréseivel sajnos nem tudjuk összehasonlítani, mivel a használandó programok nem érhetőek el publikusan, valamint a tanulmányok nem specifikálták kellő részletességgel a használt hardverkonfigurációt sem.

Egy-egy modellnél minden esetben azonos dekompozíciós módszert alkalmaztunk. A DPhil, Phil és Hanoi modellek esetén egy változószint három helyet reprezentál, az FMS modell esetén egy változószinthez egy helyet rendeltünk. Az SR modellben a [13]-ban bemutatott P-invariánsokon alapuló heurisztikát használtuk a modell dekomponálására.

7.1. A vezérelt szaturációs algoritmus vizsgálata

A fejezetben megvizsgáljuk, hogy a [13]-hoz képest milyen eredmények érhetők el, ha a klasszikus szaturációs modellellenőrző modul helyett a vezérelt szaturáción alapuló algoritmusokat használjuk. Az a) programváltozat a modellellenőrzés során klasszikus szaturációt használ és az állapotátmeneti függvényeket Kronecker-mátrixokkal kódolja. A b) verzió ve-

zérelt szaturációt használ és állapotátmeneti függvényei MDD-vel kódoltak a 3. fejezetben leírtak szerint.

A 7.1. táblázat eredményeiből és a 7.1. ábra alapján látható, hogy EF és EU operátorok használata esetén a modellellenőrzés folyamatát jelentősen felgyorsítja a vezérelt szaturációs algoritmus. Különösen jól látható ez a DPhil-1000, Phil-10000 és SR-100 modellek vizsgálatánál, ahol az a) klasszikus programverzió a megadott erőforráskorlátokon belül nem volt képes lefutni, a vezérelt szaturáció viszont jó eredményeket ért el.

A táblázatban szereplő eseteknél több mérést végeztünk és egyetlen esetben sem tapasztaltuk, hogy valamely modellre vagy kiértékelendő kifejezésre a vezérelt szaturációs algoritmus kevésbé lenne hatékony, mint a klasszikus megközelítés.

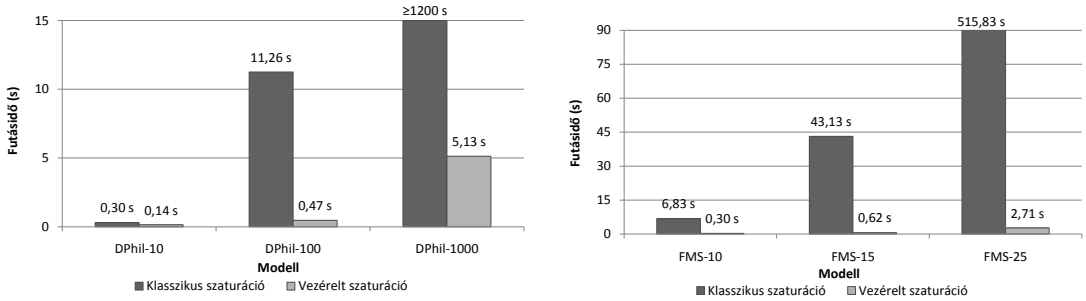
N	$\lvert\mathcal{S}\rvert$	a) Klasszikus szaturáció	b) Vezérelt szaturáció
N étkező filozófus, holtpontot tartalmazó modell (DPhil-N) CTL-kifejezés: $\mathsf{E}\ [\neg(HL_2 > 0 \wedge HR_2 > 0)\ \mathsf{U}\ (HL_1 > 0 \wedge HR_1 > 0)]$			
10	$1{,}8 \cdot 10^6$	0,30 s	0,14 s
100	10^{62}	11,26 s	0,47 s
1000	10^{629}	—	5,13 s
Rugalmas gyártórendszer (FMS-N) CTL-kifejezés: $\mathsf{E}\ (M_1 > 0\ \mathsf{U}\ (P1s = N \wedge P2s = N \wedge P3s = N))$			
10	$2{,}5 \cdot 10^9$	6,83 s	0,30 s
15	$2{,}2 \cdot 10^{11}$	43,13 s	0,62 s
25	$8{,}5 \cdot 10^{13}$	515,83 s	2,71 s
Hanoi tornyai, N gyűrűvel (Hanoi-N) CTL-kifejezés: $\mathsf{EG}\ (\mathsf{EF}\ (B_4 > 0))$			
12	$5{,}3 \cdot 10^5$	52,58 s	48,81 s
N étkező filozófus (Phil-N) CTL-kifejezés: $\mathsf{E}\ (eszik_1 = 0\ \mathsf{U}\ eszik_2 = 1)$			
100	$7{,}9 \cdot 10^{20}$	1,47 s	0,06 s
1000	$9{,}7 \cdot 10^{208}$	246,05 s	0,68 s
10000	10^{2089}	—	9,29 s
Réselt gyűrű protokoll, N csomóponttal (SR-N) CTL-kifejezés: $\mathsf{E}\ (B_1 \neq 1 \vee F_1 \neq 1\ \mathsf{U}\ G_2 = 1 \wedge A_2 = 1)$			
10	$8{,}3 \cdot 10^9$	1,06 s	0,17 s
20	$2{,}7 \cdot 10^{20}$	4,83 s	0,45 s
50	$1{,}7 \cdot 10^{52}$	49,55 s	3,31 s
100	$2{,}6 \cdot 10^{105}$	—	18,93 s

7.1. táblázat. Vezérelt szaturációs algoritmus használatának eredményei

7.2. A korlátos modellellenőrzés eredményei

Ebben a fejezetben a korlátos modellellenőrzés hatékonyságát mutatjuk be. Méréseink során három programverziót hasonlítottunk össze: a teljes állapottér-felderítést alkalmazó klasszikus a) verziót, a 4. fejezetben leírt iteratív korlátos modellellenőrzést végző b) verziót, illetve az előző kiegészítését az 5.3. fejezetben leírt vezérelt szaturációval. Az a) változatban állapotátmeneti függvények reprezentációjára Kronecker-mátrixokat, a b) és c) változatban MDD-ket használtunk.

A mérések során a kezdeti távolságkorlát 20, a korlát inkremense pedig 10 volt (a 4.3.3. fejezet jelöléseivel: $B = 20, d = 10$). Kivétel ez alól a $*$-gal jelölt eset, ahol $B = 10, d = 5$



(a) Étkező filozófusok modellje (b) Rugalmas gyártórendszer modellje

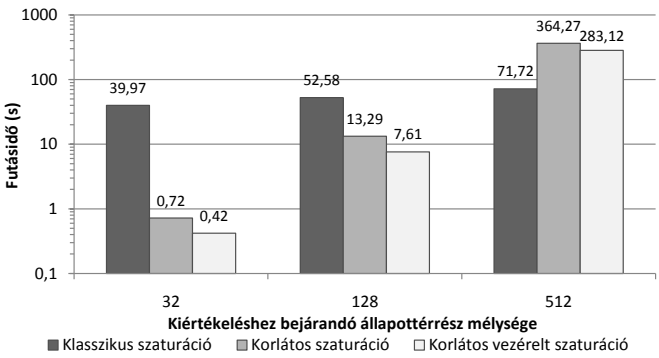
7.1. ábra. Vezérelt szaturációs algoritmus használatának eredményei

értékeket használtunk. Vágófüggvényként minden esetben a közelítő vágófüggvényt alkalmaztuk. A szigorú vágófüggvény hatékonyságával kapcsolatban [25] munkánkban olvashat.

A mérési eredményeket tartalmazó 7.2. táblázatban az időadatok mellett feltüntettük az egyes modellekben szereplő maximális mélységű állapot mélységét (azaz a modell állapotterének átmérőjét, $\delta_{max}(\mathcal{S})$-sel jelölve), illetve a kifejezés kiértékeléséhez szükséges állapottérrész átmérőjét is ($\delta_{max}(\mathcal{S}_b)$-vel jelölve). Ahol $\delta_{max}(\mathcal{S})$ értékként „≤ 20” szerepel, ott 20-nál kevesebb a felderítendő rész átmérője, azonban a kezdeti korlát megadott értéke miatt az állapottér 20 átmérőjű része mindenképp felderítésre került.

A mérési értékekből két fontos következtetés vonható le. Egyrészt a vezérelt szaturáció felhasználása az iteratív korlátos modellellenőrzésben csökkentette a verifikáció időszükségletét. Másrészt a c) vezérelt iteratív korlátos modellellenőrző eredményei jelentősen jobbak az a) változat eredményeinél azokban az esetekben, amikor $\delta_{max}(\mathcal{S}) \gg \delta_{max}(\mathcal{S}_b)$, azaz a specifikált kifejezés ellenőrzéséhez az állapottér relatíve kis részének bejárása elegendő. Bizonyos esetekben (mint az a DPhil-1000 modell vizsgálatánál látható) a modell ellenőrzése csak korlátos megközelítéssel volt lehetséges, bár állapottere nem végtelen.

Ugyanakkor azt is látni kell, hogy ha az állapottér nagy részének felderítése szükséges a követelmény kiértékeléséhez, akkor az iteratív megközelítés miatt az állapottér bizonyos részei sokszor kerülnek felderítésre. Emellett a távolságinformációk okozta többlet erőforrásigény is jelentőssé válhat. Így akár a teljes állapottér bejárásán alapuló változatnál jelentősen több időt igényelhet a korlátos ellenőrzés, ahogyan az a Hanoi-12 modell $\mathsf{EG}\ (\mathsf{EF}\ (B_2 > 0))$ kifejezésének vizsgálatánál láthatjuk. Az algoritmus skálázódását illusztrálja a 7.2. ábra is.



7.2. ábra. Korlátos modellellenőrző algoritmusok eredményei

N	$\delta_{max}(\mathcal{S})$	$\delta_{max}(\mathcal{S}_b)$	a) Klasszikus szaturáció	b) Korlátos szaturáció	c) Korlátos vezérelt szaturáció
Hanoi tornyai, N gyűrűvel (Hanoi-N) CTL-kifejezés: $\mathsf{EG}\ (\mathsf{EF}\ (B_2 > 0))$					
12	4095	512	71,72 s	364,27 s	283,12 s
CTL-kifejezés: $\mathsf{EG}\ (\mathsf{EF}\ (B_4 > 0))$					
12	4095	128	52,58 s	13,29 s	7,61 s
CTL-kifejezés: $\mathsf{EG}\ (\mathsf{EF}\ (B_6 > 0))$					
12	4095	32	39,97 s	0,92 s	0,63 s
$*$ 12	4095	32	39,97 s	0,72 s	0,42 s
N étkező filozófus, holtpontot tartalmazó modell (DPhil-N) CTL-kifejezés: $\mathsf{E}\ [\neg(HL_2 > 0 \wedge HR_2 > 0)\ \mathsf{U}\ (HL_1 > 0 \wedge HR_1 > 0)]$					
10	20	≤ 20	0,30 s	0,37 s	0,12 s
100	200	≤ 20	11,26 s	9,68 s	0,48 s
1000	2000	≤ 20	—	—	6,19 s
Réselt gyűrű protokoll, N csomóponttal (SR-N) CTL-kifejezés: $\mathsf{E}\ (B_1 \neq 1 \vee F_1 \neq 1\ \mathsf{U}\ G_2 = 1 \wedge A_2 = 1)$					
10	114	≤ 20	1,06 s	14,27 s	0,23 s
20	379	≤ 20	4,83 s	—	1,04 s
50	2074	≤ 20	49,55 s	—	3,29 s

7.2. táblázat. Korlátos modellellenőrző algoritmusok hatásai

Összefoglalva a tapasztalatainkat: a bemutatott vezérelt, iteratív korlátos megközelítésünk olyan esetekben hatékony, amikor az így bejárandó állapottér a teljes állapottérnek csak kis része. Ilyen esetekben viszont több nagyságrenddel is gyorsabb lehet a korlátos modellellenőrzés (lásd pl. a Hanoi-12 modellen vizsgált $\mathsf{EG}\ (\mathsf{EF}\ (B_6 > 0))$ kifejezés eredményeit).

7.3. Színezett Petri-hálók eredményei

Ebben a fejezetben a színezett Petri-hálókra elkészített szaturációs algoritmus alkalmazhatóságát vizsgáljuk.

A vizsgálatok szempontjából alapvetően kétfajta Petri-hálót különböztetünk meg, ugyanis ha színezett Petri-hálókat használunk, akkor azt jellemzően két ok miatt tesszük:

- paraméterezhető, kompakt modellt szeretnénk alkotni,
- szeretnénk használni azokat a bonyolult adatszerkezeteket, amelyeket a színezett Petri-hálók biztosítanak.

Ez a két problémakör különböző megoldásokat igényel a modellek teljesen eltérő jellegéből fakadóan.

Első körben a paraméterezhető modellekre vizsgáljuk a 6. fejezetben bemutatott új szaturációs algoritmust. Választásunk az étkező filozófusok paraméterezhető modelljére esett. Az étkező filozófusok színezett modelljének állapottér-felderítési statisztikáit a 7.3. táblázat mutatja. A táblázatban a következő adatokat tüntettük fel: a modellben szereplő filozófusok száma (N), színezetlen esetben az állapotfelderítés ideje, színezett esetben az állapotfelderítés ideje a 6.2. és a 6.3. fejezetekben bemutatott algoritmusokkal. Ezen felül

mind színezetlen, mind színezett esetben feltüntettünk egy kiválasztott eseményhez tartozó állapotátmeneti reláció méretét.

A táblázatban szereplő értékeket megvizsgálva azt tapasztaljuk, hogy erre a színezetlen hálóra az állapottér-felderítés a szaturációs algoritmussal hatékonyan elvégezhető, ahogyan ez látható volt korábbi munkáinkban is [13]. A színezett Petri-hálós modellre vonatkozó értékek alapján a modell egyértelműen exponenciálisan skálázódik, amely rendkívül kedvezőtlenül hat a futási időre. Ennek oka az állapotátmenetek számának exponenciális növekedése, amely kezelése hatalmas többletköltséget jelent.

Az új algoritmus annak ellenére, hogy hatékonyabban építi az állapotátmeneti relációkat, nem változtat az állapotfelderítés komplexitásán, hiszen az állapotátmenetek exponenciális növekedését nem befolyásolja. Ez a jelleg abból ered, hogy a modellben egy tranzíció sok színezetlen tranzíciót reprezentál, tehát az állapotátmenet mérete a színezetlen tranzíciós relációk méretének a Descartes-szorzata lesz. Az étkező filozófusok vizsgálata alapján megállapítottuk, hogy a paraméterezhető és sok lokális aszinkronalitást tartalmazó modellek vizsgálatára ilyen formában nem alkalmasak a szimbolikus módszerek. Ugyan korábban arról számoltunk be, hogy a szaturáció hatékony aszinkron rendszerek vizsgálatára [13, 16], de ha nem tudjuk megfelelően dekomponálni a modellünket, akkor a szimbolikus technikáknál még az explicit módszerek is jobban fognak teljesíteni, ugyanis utóbbiaknál nem soroljuk fel az összes lehetséges állapotátmenet kombinációját.

A fejezet elején már említést tettünk a színezett Petri-hálók másik csoportjáról, amelyeknél a bonyolult adatszerkezetek lehetőségeit használjuk ki. Az ilyen típusú modelleket a következő fejezetben vizsgáljuk egy valós esettanulmányon keresztül.

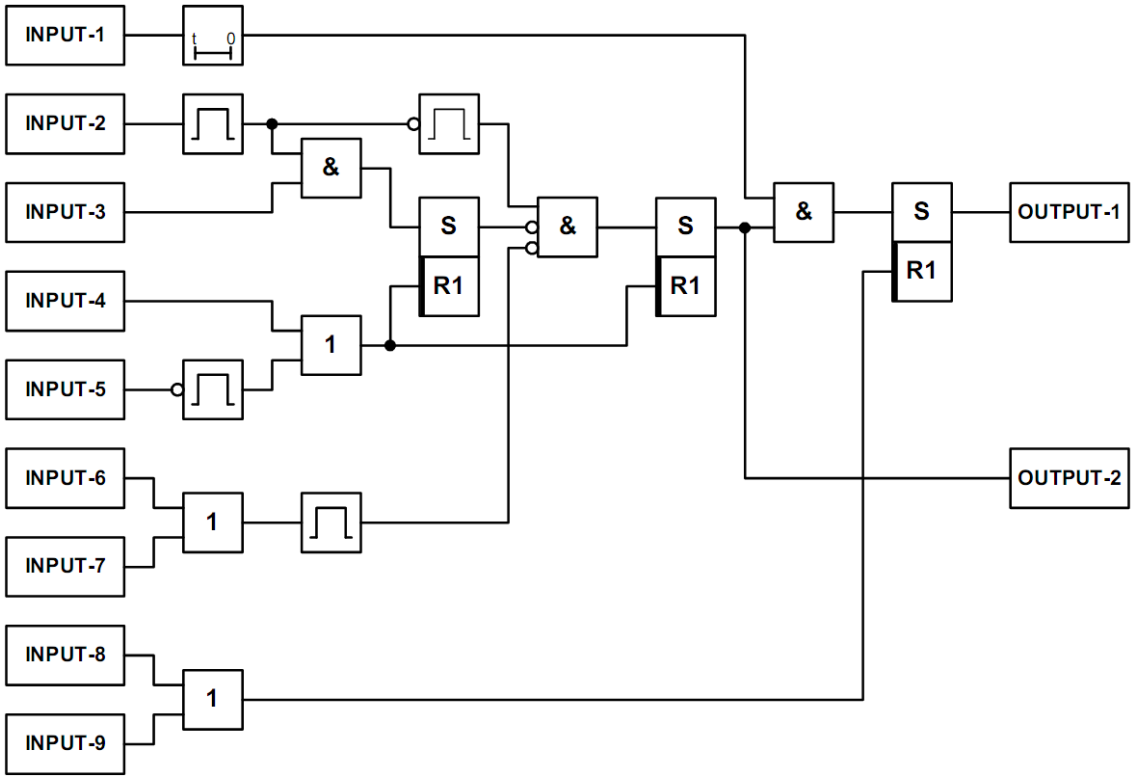
N	Színezetlen Petri-háló		Színezett Petri-háló		
	Futásidő	Állapot-átmenetek száma	Futásidő		Állapot-átmenetek száma
			Konjunktív partícionáló algoritmus	Változó-kódoló algoritmus	
5	0,01 s	10	0,56 s	0,20 s	1320
10	0,01 s	20	—	61,19 s	3 721 980

7.3. táblázat. Étkező filozófusok színezett Petri-hálós modelljének vizsgálata

7.4. Ipari esettanulmány

Színezetlen Petri-hálók helyett színezettek használatával sokkal komplexebb rendszerek modellezésére nyílik lehetőségünk. Segítségükkel már bonyolult ipari rendszerek verifikációja is megvalósítható.

A korábbiakhoz hasonlóan elsősorban itt is olyan ipari rendszerekre gondolunk, amelyek a globális aszinkronitás mellett lokálisan szinkron tulajdonságokkal bírnak. Az előző fejezetben láttuk, hogy a lokálisan aszinkron rendszerek esetén, mint amilyen az étkező filozófusok színezett Petri-hálós modellje volt, a szimbolikus technikák nem bizonyulnak hatékonynak. Ebben a fejezetben egy olyan ipari rendszert fogunk megvizsgálni, amiben megjelenik a korábban emlegetett adatfogalom, mindemellett aszinkron működésű. Ez a rendszer a Paksi Atomerőmű egyik védelmi funkciójának indító logikája, amit PRISE-ként rövidítünk [20, 19]. Ennek ellenőrzésére a korábbi módszerek nem voltak alkalmasak [23, 19].



7.3. ábra. A PRISE-modell funkcionális blokkdiagramja [20]

7.4.1. A PRISE-modell bemutatása

A vizsgált PRISE-modell a paksi atomerőmű egy biztonsági logikája. Célja a PRISE-eseménynek nevezett hiba detektálása és annak jelzése a kimeneten. A hiba tulajdonképpen az erőműben a primer (és emiatt radioaktív szennyeződéseket tartalmazó) hűtővíznek a szekunder körbe való szivárgást jelenti. A modell áttekintő blokk diagramját a 7.3. ábra mutatja. A modell a következő elemekből épül fel: pulzusgenerátorok, késleltetők, logikai és kapuk, logikai vagy kapuk, inverterek és SR flip-flopok. Ezek közül a késleltető feladata, hogy a bemeneten szereplő felfutó él a kimeneten csak D idő után jelenjen meg. Ezzel szemben a pulzusgenerátor a bemenet felfutó élét a kimeneten P ideig megtartja. A 7.3. ábrán az *Input-1* bemeneten egy késleltető, az *Input-2* bemeneten pedig egy pulzusgenerátor látható.

A logika a bemenő jeleket vizsgálja, és azok időbeli változása alapján jelzi a kimeneten, ha veszélyes PRISE-esemény következett be. Erre azért van szükség, mert az *Input-1* szenzor nem megbízható, illetve veszély esetén a szenzoroknak meghatározott sorrendben történő aktiválódása jelenti az esemény bekövetkeztét. Az esemény detektálásáról bővebben a [19]-ben olvashat.

A szakirodalom [19] alapján megalkottuk a PRISE-modell színezett Petri-hálós modelljét. A modellt alkotó komponensek viselkedését egyenként ellenőriztük, hogy megfelelnek-e a specifikációnak, majd ezekből a komponensekből összeraktuk a teljes modellt.

7.4.2. A PRISE-modell vizsgálata

A PRISE-modell verifikációjára több kísérlet is történt korábban. Ezek különböző modellezőeszközöket használtak, úgymint SAL [31], UPPAL [32], CPNTools [30], de egyikkel sem sikerült a teljes modellt egyben, a teljes működési tartományt vizsgálva verifikálni

[23]. A legtöbb kísérlet csak úgy tudott sikereket elérni, hogy megkötötték a működési tartományt vagy pedig a modellt dekomponálva, részenként vizsgálták.

Munkánk során többféle megközelítést is kipróbáltunk a PRISE-modell verifikációjára. A különböző szaturációs megközelítések jellege eltérő: míg az egyszerű Petri-hálók Kronecker-konzisztens dekompozícióján alapuló megoldás hatékony és kompakt állapotátmenet reprezentációkat használ és kis állapotátmenetekkel dolgozik, az általunk kifejlesztett színezett Petri-hálókra adaptált, rugalmas állapotátmenet-leképezés nem igényli a modell ennyire részletes felbontását. Ezáltal lehetőséget ad a modellben található logikai lépések egyben történő kezelésére. Ezt azon az áron teszi, hogy a bonyolultabb állapotátmeneti relációk kevésbé tárhatékonyak, mint a Kronecker-mátrixokon alapuló leképezés.

Elvégeztük a PRISE-modell színezett Petri-hálójának egyszerű Petri-hálóvá történő átalakítását, és különböző modelldekompozíciós beállítások mellett lefutattuk rajta az állapottér felderítő algoritmust. Azonban a modell paraméterezésétől függetlenül minden kísérlet sikertelen volt a nagy memóriaigény miatt. Lefuttattuk a PRISE-modellre a konjunktív partícionálást használó szaturációs algoritmust is, amely szintén memória korlátokba ütközött. Jól láthatóan egyik algoritmus sem tudta hatékonyan kezelni az állapotátmenetek reprezentációja által jelentett kihívást.

A korábbi próbálkozások sikertelenségének legfőbb oka a modell hatalmas állapottere volt. Mivel azok a módszerek nem szimbolikus állapottér-reprezentációkat alkalmaznak, a 10^{14} állapot tárolására a mi mérési környezetünkben nem volt esély (ha minden állapotot 1 biten tárolnánk, akkor is 11 terabájt memóriára lenne szükség). A mi szaturációs algoritmusaink ezzel ellentétben szimbolikus állapottér-reprezentációt alkalmaznak, amely sokkal hatékonyabb és kompaktabb állapottér-tárolást tesz lehetővé. Azonban megmutatkozott a szimbolikus módszerek nagy hátránya, hogy az állapotátmenet-relációt is tárolni kell: a viszonylag bonyolultabb átmenetek miatt az állapotátmenet-relációk építése megtöltötte a memóriát.

Az általunk fejlesztett új állapotátmenet-reláció építő algoritmust implementáltuk a keretrendszerben és – kihasználva az állapotátmenetek hatékony építését – a korábbi megközelítéseinknél jobb eredményt vártunk. Az erre vonatkozó eredményeket a következő alfejezetben mutatjuk be.

Verifikáció

Az általunk kifejlesztett algoritmust először az állapottér generálására használtuk. Ha az állapottér-reprezentáció rendelkezésünkre áll, akkor ezen később modellellenőrzési vizsgálatokat tudunk végezni. Az újfajta állapotátmenet-reláció építésnek köszönhetően sikerült az állapottér-generálást elvégezni a mérési környezetünkben. Ez az első sikeres kísérlet, hogy a PRISE-logikát egyben vizsgáljuk és a teljes működési tartományt bejárjuk.

A mérési eredményeinket a 7.4. táblázat tartalmazza. Az első 3 mérési eredmény az alábbi szakirodalmakból származik: [23, 19]. A SAT az egyszerű, színezetlen Petri-hálókkal dolgozó szaturációs algoritmust (2.6.5. fejezet), a K-SAT a konjunktív partícionálást használó algoritmust (6.2. fejezet), a VK-SAT pedig a tokenváltozók lekötéseit is tároló szaturációt jelenti (6.3. fejezet). A méréseket saját, kézzel összeállított [13] változósorrendezés mellett végeztük el. A mérés során felépített adatstruktúrák méretei a 7.5. táblázatban láthatóak.

A verifikációhoz azonban nem elegendő az állapottér felépítése, komplexitását tekintve ennél bonyolultabb feladat a modellellenőrzés. A PRISE-modell komplex logikát valósít meg, amely ellenőrzéséhez összetett temporális logikai kifejezésekre van szükség. A verifikáció során mind elérhetőségi, mind biztonságossági vizsgálatokat végeztünk. A következőkben bemutatunk 2 vizsgált követelményt és a hozzájuk kapcsolódó mérési eredményeinket:

	SAL	UPPAL	CPN Tools	SAT	K-SAT	VK-SAT
Futási idő	—	—	—	—	—	950 s
Memória foglalás	—	—	—	—	—	2,5 GB

7.4. táblázat. A PRISE-modell állapottér-felderítése

Vizsgált jellemző	Érték
Futási idő	950 s
Globális állapotok száma	$4{,}836 \cdot 10^{12}$
Állapottér MDD csomópontszáma	1 497
Lokális állapotátmenetek száma	10 082 881
Állapotátmeneti MDD csomópontszáma	782 159

7.5. táblázat. Állapottér-felderítés jellemzői VK-SAT algoritmussal

- holtpontmentesség ellenőrzése,
- nincs téves riasztás a rendszerben.

A holtpontmentesség egy általános modellellenőrzési feladat. A PRISE-modell esetében a holtpontmentesség azt jelenti, hogy a rendszer mindig képes reagálni a külső változásokra, amely egy védelmi logika esetén alapvető követelmény. A holtpontmentesség vizsgálata során az összes állapotra megvizsgáljuk, hogy továbbléphetünk-e belőle. Ez CTL temporális logikai nyelvvel megfogalmazva a következő kifejezés:

$$\mathsf{AG}\ (\mathsf{EX}\ \mathit{true})$$

A holtpontmentesség verifikációjának időigénye: 6,35 s.

A második vizsgálat arra vonatkozott, hogy történhet-e téves riasztás, azaz csak akkor történik riasztás, ha bekövetkezett a PRISE-esemény. Ezt a következőképpen fogalmaztuk meg CTL nyelven:

$$\neg(\mathsf{E}\ (\neg\langle \mathit{PRISE\text{-}esemény}\rangle\ \mathsf{U}\ \langle \mathit{Riasztás}\rangle))$$

A kifejezés verifikációjának időigénye: 2,17 s.

Összegzés

Az általunk kifejlesztett színezett Petri-hálós szaturáció algoritmust implementáltuk a PetriDotNet keretrendszerben és megvizsgáltuk a hatékonyságát egy valós, ipari modellen keresztül. A választott ipari esettanulmány a PRISE-modell volt. Választásunk azért erre a problémára esett, mert a korábbi, teljes működési tartomány verifikációjára tett kísérletek sikertelennek bizonyultak [23, 19]. Jelentős eredmény, hogy a mi megoldásunk nemcsak az állapotteret tudta felderíteni, hanem a modellellenőrzési problémára is megoldást nyújtott.

8. fejezet

Összefoglalás

8.1. Eredmények összegzése

Munkánk célja olyan módszerek vizsgálata, kidolgozása és implementálása volt, amelyek segítségével nagy komplexitású modellek verifikációja is lehetővé válik.

A dolgozatban ismertetett, általunk kidolgozott iteratív korlátos modellellenőrzési technikával, illetve az ezt kiegészítő vezérelt szaturációs ellenőrző algoritmus használatával létrehoztuk az első hatékony, szaturációra épülő modellellenőrző megoldást. Emellett a vezérelt szaturáció használatával a teljes állapotteret bejáró modellellenőrzés hatékonysága javult, elért eredményei jelentősen felülmúlják a korábbiakban bemutatott [13] algoritmusok eredményeit. A munkánk során kidolgozott algoritmusokat implementáltuk a *PetriDotNet* keretrendszerben, így azok elérhetők és használatra készek [29].

Emellett munkánk során elsőként vizsgáltuk meg a szaturációs modellellenőrzés használhatóságát színezett Petri-hálók esetén. Dolgozatunkban bemutattuk, hogy a szaturációs megoldás adaptálható erre a modelltípusra, azonban a kiegészítések nélküli adaptáció nem optimális, mivel ebben az esetben a hálók nem darabolhatók kellően kis komponensekre, amely viszont a hatékony szaturációs állapottér-felderítés szükséges feltétele.

E tapasztalatok által motiválva kidolgoztunk egy új megközelítést a színezett Petri-hálók állapotátmeneti függvényeinek tárolására. Ezzel a módszerrel már lehetővé vált komplex színezett Petri-hálók analízise is. Ahogyan azt a 7.4. fejezetben bemutattuk, sikeresen tudtunk állapottér-felderítést és bizonyos analíziseket végezni egy valódi biztonságkritikus ipari rendszeren, amely eddig a szaturációs módszerek nélkül nem volt lehetséges.

8.2. Továbbfejlesztési lehetőségek

Természetesen még sok munkánk van a szaturációs modellellenőrző fejlesztése terén. Reményeink szerint a *PetriDotNet* keretrendszer és a szaturációs modellellenőrző modul fejlesztése folytatódik. További munkánk során az alábbi irányokat szeretnénk megvizsgálni:

- Terveink közt szerepel az iteratív korlátos modellellenőrzés továbbfejlesztése úgy, hogy az egyes iterációk során a korábbi részállapotterekből minél több információt fel tudjunk használni, azaz a bejárási korlátok növelésekor a felderítés újrakezdése kevesebb információvesztéssel járjon.

- A vezérelt szaturáció jelentősen javította az EU és EF operátorok kiértékelésének hatékonyságát. Szeretnénk azonban olyan módszereket vizsgálni és kidolgozni, amelyek segítségével az EG operátor kiértékelése is jelentősen felgyorsítható.

- Szeretnénk a színezett Petri-hálók állapotátmenet-tárolását hatékonyabbá tenni. Az algoritmus – jellegéből adódóan – minden lehetséges állapotátmenet-relációt felépít egy új lokális állapot sikeres felderítésekor. Ha azonban nem a állapotátmenet-relációt építenénk, hanem engedélyezettségi relációt, akkor sok fölösleges műveletet nem kellene elvégeznünk. Így az állapotátmenet-relációt „mohó stílusban" csak akkor építenénk meg, amikor ténylegesen végre is hajtja az algoritmus.
- A további munkánk során hatékony modelldekompozíciót lehetővé tevő módszereket szeretnénk kidolgozni a színezett Petri-hálók automatikus modellellenőrzéséhez.

Ábrák jegyzéke

Irodalomjegyzék

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

[2] P. Buchholz, Gianfranco Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. on Computing*, 12:203–222, July 2000.

[3] Jerry R. Burch, Edmund Clarke, and David E. Long. Symbolic model checking with partitioned transition relations. In *International Conference on Very Large Scale Integration*, pages 49–58. North-Holland, 1991.

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

[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, 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.

[7] 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.

[8] Gianfranco Ciardo and Andrew S. Miner. Storage Alternatives for Large Structured State Spaces. In *Proceedings of the 9th ICCPE: Modelling Techniques and Tools*, pages 44–57, London, UK, 1997. Springer-Verlag.

[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] Gianfranco Ciardo and Andy Jinqing Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *CHARME'05*, pages 146–161, 2005.

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

[13] Darvas Dániel. Szaturáció alapú automatikus modellellenőrző fejlesztése aszinkron rendszerekhez. *Tudományos Diákköri Konferencia, Budapesti Műszaki és Gazdaságtudományi Egyetem, Villamosmérnöki és Informatikai Kar*, 2010.

[14] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.

[15] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems.* Springer, 2009.

[16] Jámbor Attila, Szabó Tamás. Aszinkron rendszerek modellellenőrzése párhuzamos technikákkal. *Tudományos Diákköri Konferencia, Budapesti Műszaki és Gazdaságtudományi Egyetem, Villamosmérnöki és Informatikai Kar*, 2010.

[17] 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.

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

[19] Erzsébet Németh and Tamás Bartha. Formal verification of safety functions by reinterpretation of functional block based specifications. In Darren Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems*, volume 5596 of *Lecture Notes in Computer Science*, pages 199–214. Springer Berlin / Heidelberg, 2009.

[20] Erzsébet Németh, Tamás Bartha, Cs. Fazekas, and K.M. Hangos. Verification of a primary-to-secondary leaking safety procedure in a nuclear power plant using coloured petri nets. *Reliability Engineering and System Safety*, 94 (5):942–953, 2009.

[21] Pataricza András, editor. *Formális módszerek az informatikában.* Typotex, 2. kiadás, 2005.

[22] R. Saad, S. Dal Zilio, and B. Berthomieu. Mixed Shared-Distributed Hash Tables Approaches for Parallel State Space Construction. In *ISPDC*, Cluj Napoca, Romania, 07/2011 2011. IEEE.

[23] Tóth Heinemann Zsófia. Diszkrét ipari irányítórendszerek modellezése és ellenőrzése formális módszerekkel (diplomaterv), 2009.

[24] Vörös András, Bartha Tamás, Darvas Dániel, Szabó Tamás, Jámbor Attila, Horváth Ákos. Parallel Saturation Based Model Checking. In *ISPDC*, Cluj Napoca, 2011. IEEE Computer Society, IEEE Computer Society.

[25] Vörös András, Darvas Dániel, Bartha Tamás. Bounded Saturation Based CTL Model Checking. In Jaan Penjam, editor, *Proceedings of the 12th Symposium on Programming Languages and Software Tools, SPLST'11*, pages 149–160, Tallinn, Estonia, 2011. Tallinn University of Technology, Institute of Cybernetics.

[26] Michael Weber and Ekkart Kindler. The Petri Net Markup Language. *Petri Net Technology for Communication-Based Systems*, pages 124–144, 2003.

[27] Andy Yu, Gianfranco Ciardo, and Gerald Lüttgen. Decision-diagram-based techniques for bounded reachability checking of asynchronous systems. *Int. J. Softw. Tools Technol. Transf.*, 11:117–131, February 2009.

[28] Yang Zhao and Gianfranco Ciardo. Symbolic CTL model checking of asynchronous systems using constrained saturation. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis*, ATVA '09, pages 368–381, Berlin, Heidelberg, 2009. Springer-Verlag.

[29] A *PetriDotNet* keretrendszer honlapja.
`http://petridotnet.inf.mit.bme.hu/`.

[30] A CPN Tools honlapja.
`http://cpntools.org/`.

[31] A SAL honlapja.
`http://sal.csl.sri.com/`.

[32] Az UPPAAL honlapja.
`http://www.uppaal.com/`.