



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

Szaturáció alapú modellellenőrzés lineáris idejű tulajdonságokhoz

Szakdolgozat

Készítette:
Molnár Vince

Konzulensek:
dr. Bartha Tamás
Vörös András

2012.

Tartalomjegyzék

Kivonat	7
Abstract	9
Bevezető	11
1. Háttérismeretek	13
1.1. Modellellenőrzés	13
1.2. Lineáris idejű temporális logika	13
1.3. Automataelméleti háttér	15
1.3.1. Büchi automaták	16
1.3.2. Automaták szinkron szorzata	16
1.3.3. Általánosított Büchi automaták	18
1.3.4. Transzformációs algoritmus	19
1.4. Döntési diagramok	20
1.5. Szaturáció	22
1.5.1. A modell dekomponálása	22
1.5.2. Az állapottér leírása	22
1.5.3. Események	23
1.5.4. Next-state függvény	23
1.5.5. A szaturációs algoritmus működése	23
1.5.6. Vezérelt szaturáció	24
2. LTL modellellenőrzés	25
2.1. Áttekintés	25
2.1.1. Algoritmikus problémák	26
2.2. A szaturáció kiterjesztése a szinkron szorzat automatára	27
2.2.1. Szimbolikus állapottér-reprezentáció	28
2.2.2. Események szinkronizációja	29
2.2.3. Predikátum kényszerek	31
2.2.4. Formális megközelítés	32
2.2.5. Az új vezérelt szaturációs algoritmus	33
2.3. Erősen összefüggő komponensek keresése	36
2.3.1. A szaturációs algoritmus lépései halmazelméleti szempontból	37
2.3.2. Körök felderítése követelmények mellett	38
2.3.3. Inkrementális körkereső algoritmus	40
2.3.4. Az algoritmus helyességének és teljességének bizonyítása	42
2.4. Optimalizációk	44
2.4.1. Az LTL kifejezés egyszerűsítése	44
2.4.2. A Büchi automata egyszerűsítése	45

2.5. Az algoritmus összefoglalása	46
3. Implementáció	49
3.1. Áttekintés	49
3.2. LTL és automaták	49
3.2.1. LTL kifejezések	49
3.2.2. Automaták	52
3.3. Szaturáció	52
3.3.1. Meglévő szaturációs kódok és felépítésük	52
3.3.2. Konceptió	53
3.3.3. Burkoló osztályok	54
3.3.4. Új adatszerkezetek és felépítésük	56
3.3.5. A szaturációs algoritmus módosításai	56
3.4. Körkeresés	57
4. Mérési eredmények	61
4.1. Összehasonlító mérések	61
4.2. Fairness kritériumok vizsgálata	62
4.3. Skálázódás és optimalizációk	63
4.3.1. Étkező filozófusok	64
4.3.2. Réselt gyűrű	67
4.3.3. Kanban	70
4.3.4. Értékelés	73
5. Összefoglalás	75
5.1. Elért eredmények	75
5.1.1. Elméleti eredmények	75
5.1.2. Gyakorlati eredmények	75
5.2. Továbbfejlesztési lehetőségek	75
Köszönetnyilvánítás	77
Ábrák jegyzéke	80
Irodalomjegyzék	82
Függelék	84
F.1. Gerth, Peled, Vardi és Wolper algoritmus a LTL transzformációs algoritmus	84



SZAKDOLGOZAT-FELADAT

Molnár Vince (XD5Z68)

szigorló mérnök-informatikus hallgató részére

Szaturáció alapú modellellenőrzés lineáris idejű tulajdonságokhoz

Napjainkban a legtöbb területen, így a biztonságkritikus rendszerek területén is egyre elterjedtebbek a beágyazott vezérlő és felügyelő eszközök. Egy biztonságkritikus rendszer hibája akár katasztrofális következményekhez is vezethet, így e beágyazott eszközök helyes működésének bizonyítása kiemelten fontos feladat. Formális módszerek használatával már a fejlesztési folyamat elején, tervezési időben, matematikai precizitással vizsgálható a rendszerek helyes működése. A *modellellenőrzés* az egyik leggyakrabban használt formális módszer. A modellellenőrzés végrehajtása során a rendszer modelljén vizsgáljuk az ún. *temporális logika* segítségével megfogalmazott követelmények teljesülését. Többféle temporális logika létezik, ezek közül a lineáris idejű (LTL) és az elágazó idejű (CTL) temporális logika a legelterjedtebb. Kifejezőerejük nem összehasonlítható, azonban az LTL sokszor praktikusabb kompaktsága és átláthatósága miatt, ami a CTL-nél könnyebben használhatóvá teszi. Ennek azonban ára is van, az LTL modellellenőrzés komplexitása a PSPACE osztályba tartozik, tehát szükséges, hogy hatékony algoritmusokat alkalmazzunk.

A szaturáció egy szimbolikus iterációs algoritmus, amely hatékony állapottér-felderítésre és CTL modellellenőrzésre képes. Szaturáció-alapú megoldások eddig csak CTL modellellenőrzéshez léteztek, LTL-hez nem

A hallgató feladata megvizsgálni, hogy lehet-e szaturációs alapokon LTL modellellenőrzést végezni.

A hallgató feladatai a következőkre terjednek ki:

- Mutassa be röviden a szaturációs állapottér-generáló és modellellenőrző algoritmust és az LTL modellellenőrzés elméleti háttérét!
- Mutassa be az irodalomkutatás alapján a szaturációs algoritmus továbbfejlesztési lehetőségeit lineáris idejű tulajdonságok vizsgálatára!
- Implementálja a megtervezett algoritmust a PetriDotNet keretrendszerbe!
- Mérésekkel vizsgálja meg az elkészített algoritmus hatékonyságát!

Tanszéki konzulens: Dr. Bartha Tamás, docens

Vörös András, tudományos segédmunkatárs

Budapest, 2012. október 17.

.....
Dr. Jobbágy Ákos
tanszékvezető

HALLGATÓI NYILATKOZAT

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

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2012. december 16.

Molnár Vince
hallgató

Kivonat

Napjainkban a legtöbb területen, így a biztonságkritikus rendszerek területén is egyre elterjedtebbek a beágyazott vezérlő és felügyelő eszközök. Egy biztonságkritikus rendszer hibája akár katasztrofális következményekhez is vezethet, így e beágyazott eszközök helyes működésének bizonyítása kiemelten fontos feladat. Formális módszerek használatával már a fejlesztési folyamat elején, tervezési időben, matematikai precizitással vizsgálható a rendszerek helyes működése. A modellellenőrzés az egyik leggyakrabban használt formális módszer, mert komplex kérdéseket képes vizsgálni és sok információt szolgáltat. A modellellenőrzés végrehajtása során a rendszer modelljén vizsgáljuk az ún. temporális logika segítségével megfogalmazott követelmények teljesülését. Többféle temporális logika létezik, ezek közül a lineáris idejű (LTL) és az elágazó idejű (CTL) temporális logika a legelterjedtebb. Kifejezőerejük nem összehasonlítható, azonban az LTL sokszor praktikusabb kompaktsága és átláthatósága miatt, ami a CTL-nél könnyebben használhatóvá teszi. Ennek azonban ára is van, az LTL modellellenőrzés komplexitása a PSPACE osztályba tartozik, tehát szükséges, hogy hatékony algoritmusokat alkalmazzunk.

A szaturáció egy szimbolikus iterációs algoritmus, amely hatékony állapottérfelderítésre és CTL modellellenőrzésre képes. Hatékonyságát egy speciális iterációs stratégiának és a szimbolikus, azaz tömören kódolt állapottárolásnak köszönheti. Szaturáció-alapú megoldások léteznek CTL modellellenőrzésre, azonban ezek nem használhatóak LTL modellellenőrzésre, annak eltérő karakterisztikája miatt. Szimbolikus algoritmusokat próbáltak már újabban LTL modellellenőrzésre használni, azonban a szaturációs algoritmust még senki sem alkalmazta sikerrel ezen a területen.

Dolgozatomban azt vizsgálom meg, miként lehetne a szaturációs algoritmust kiterjeszteni LTL modellellenőrzésre. Munkám eredménye egy olyan automataelméleti megközelítés, amelyet szaturációs alapokon valósítottam meg, így létrehozva egy új, hatékony modellellenőrző algoritmust.

Dolgozatomban bemutatom a szaturáció alapú lineáris idejű temporális logikán alapuló modellellenőrzés elméleti hátterét és az általam kifejlesztett új modellellenőrző algoritmust.

Abstract

Nowadays, embedded controllers are becoming more and more widespread. Since the failure of a safety-critical system may have disastrous effects, the verification of these embedded systems is an important task. We can check the correct behaviour of a system with the help of formal methods in a mathematically sound way in the early phases of development. Model checking is a widely used technique to answer complex verification questions, providing a great deal of information. During the execution of model checking, we examine the validity of requirements expressed with temporal logics. There exist several temporal logics; the most widespread are linear temporal logic (LTL) and computation tree logic (CTL). Their expressiveness cannot be compared, nevertheless using LTL is often more convenient, due to its compactness and simplicity. This does not come for free though – the complexity of LTL model checking is in the PSPACE class, thus it is vital to use efficient algorithms.

Saturation is a symbolic iteration algorithm that is capable of effective state-space exploration and CTL model checking. Its efficiency is due to its iteration strategy and the symbolic, thus compact coding of states. There are saturation-based solutions for CTL model checking, but they cannot be used for LTL model checking because of their different characteristics. There were attempts to use symbolic algorithms for LTL model checking lately – however no one has ever successfully applied saturation to this problem domain yet.

In my work I examine how the saturation algorithm could be applied to LTL model checking. The result is an automata-theoretic approach that I developed based on saturation, thus creating a new, efficient model checking algorithm.

In this work, I will present the theoretical background of the saturation-based linear temporal logic model checking and the new model checking algorithm that I introduced.

Bevezető

Napjainkban a legtöbb területen, így a biztonságkritikus rendszerek területén is egyre elterjedtebbek a beágyazott vezérlő és felügyelő eszközök. Egy biztonságkritikus rendszer hibája akár katasztrofális következményekhez is vezethet, így e beágyazott eszközök helyes működésének bizonyítása kiemelten fontos feladat. Formális módszerek használatával már a fejlesztési folyamat elején, tervezési időben, matematikai precizitással vizsgálható a rendszerek helyes működése. A modellellenőrzés [7] az egyik leggyakrabban használt formális módszer, mert komplex kérdéseket képes vizsgálni. A modellellenőrzés során a rendszer modelljén vizsgáljuk az ún. temporális logika segítségével megfogalmazott követelmények teljesülését.

Többféle temporális logika létezik [7], ezek közül a lineáris idejű (LTL) és az elágazó idejű (CTL) temporális logika a legelterjedtebb. Kifejezőerejük nem összehasonlítható, azonban az LTL sokszor praktikusabb kompaktsága és átláthatósága miatt, ami a CTL-nél könnyebben használhatóvá teszi. Bizonyított, hogy ha egy temporális logikai állítás CTL és LTL segítségével is kifejezhető, akkor az LTL kifejezés mindig rövidebb, vagy legfeljebb ugyanolyan hosszú [14]. Ennek azonban ára is van, az LTL modellellenőrzés komplexitása a PSPACE osztályba tartozik, tehát szükséges, hogy hatékony algoritmusokat alkalmazzunk.

A szaturáció egy szimbolikus iterációs algoritmus, amely hatékony állapottérfelderítésre és CTL modellellenőrzésre képes. Hatékonyágát egy speciális iterációs stratégiának és a szimbolikus, azaz tömören kódolt állapottárolásnak köszönheti. Szaturáció-alapú megoldások léteznek CTL modellellenőrzésre [17], azonban ezek nem használhatóak LTL modellellenőrzésre, annak eltérő karakterisztikája miatt. Szimbolikus algoritmusokat próbáltak már újabban LTL modellellenőrzésre használni [12], azonban a szaturációs algoritmust még senki sem alkalmazta sikerrel ezen a területen.

Dolgozatomban azt vizsgálom meg, miként lehet a szaturációs algoritmust kiterjeszteni LTL modellellenőrzésre. Munkám eredménye egy olyan automataelméleti megközelítés, amelyet szaturációs alapokon valósítottam meg, így létrehozva egy új, hatékony modellellenőrző algoritmust. Bemutatom a szaturáció alapú lineáris idejű temporális logikán alapuló modellellenőrzés elméleti háttérét és az általam kifejlesztett új modellellenőrző algoritmust.

A dolgozat felépítése a következő. Először bemutatom a munkám alapját képező háttérismereteket az 1. fejezetben, ismertetve a modellellenőrzés problémakörét, a lineáris temporális logikát, a felhasznált különböző osztályú automatákat, illetve az algoritmusom alapját képező szaturációs algoritmust. A 2. fejezetben megadom az új modellellenőrző algoritmusom elméleti felépítését és bemutatom a működését. A 3. fejezetben írok az algoritmus implementációjának részleteiről és a használt keretrendszerről. A 4. fejezetben röviden bemutatok néhány, az algoritmus hatékonyságát alátámasztó mérési adatot. Végül az 5. fejezetben összefoglalom munkám eredményeit, és röviden megadok néhány olyan irányvonalat, amelyek mentén az új algoritmusom továbbfejleszthető.

1.

Háttérismeretek

Ebben a fejezetben röviden bemutatom munkám elméleti háttérét. Munkám során a modellellenőrzési problémával foglalkoztam (1.1. fejezet), amely során lineáris temporális logika (1.2. fejezet) segítségével megfogalmazott kritériumok vizsgálatát tettem lehetővé. Ebben a fejezetben bemutatom a modellellenőrzés során alkalmazott különböző automata-típusokat (1.3. fejezet), valamint a döntési diagramokat (1.4. fejezet). Legvégül ismertetek egy speciális iterációs stratégiát használó algoritmust: a szaturációt (1.5. fejezet), amely algoritmust sikeresen adaptáltam, hogy komplex modellellenőrzési problémákat oldjak meg a segítségével.

1.1. Modellellenőrzés

A modellellenőrzés egy elterjedt verifikációs technika, amely általában a rendszer egy véges modelljén vizsgálja meg, hogy a specifikációs tulajdonságok teljesülnek-e. Formálisan azt vizsgáljuk, hogy egy M modellre egy adott r specifikációs követelmény igaz-e [7]. Az r követelmény többféle, különböző kifejezőerejű formalizmussal megadható, amikkel eltérő típusú kifejezések értékelhetők ki.

Munkám során temporális logikákat használtam, amelyek segítségével kijelentések igazságának logikai időbeli változása írható le. Ezen belül jelen munka tárgyát a lineáris idejű temporális logika alapú modellellenőrzés képezi, a következőkben az ehhez kapcsolódó háttérismereteket mutatom be.

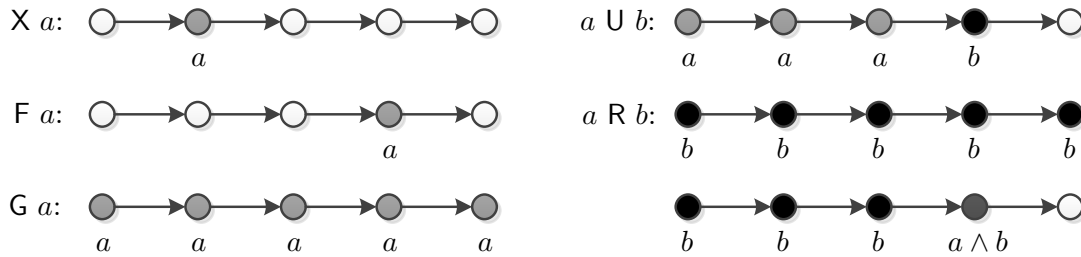
1.2. Lineáris idejű temporális logika

Mint minden logika, a lineáris idejű temporális logika (LTL) [7] is elemi kifejezésekből (ebben az esetben állapotkifejezésekből), illetve ezek kapcsolatát kifejező logikai operátorokból épül fel. Ezeken kívül azonban egy *reaktív* rendszerben az állapotok közötti átmenetek is érdekesek. Az ilyen rendszerek folyamatos kölcsönhatásban vannak a környezetükkel, és legtöbbször ciklikus viselkedésűek, azaz nem áll le a futásuk. Emiatt a vizsgálatuk során nem elég egy korlátos lefutást vizsgálni.

A temporális logikák ezeket az állapotátmeneteket és állapotszekvenciákat képesek formálisan leírni, megfogalmazni. A lineáris temporális logikában (akárcsak közeli rokonában, az elágazó idejű temporális logikában) az idő nem jelenik meg közvetlenül, csak logikai formában. Egy kifejezés például előírhatja, hogy egy állapotnak valamikor a jövőben elő kell állnia, vagy egyáltalán nem állhat elő. Ezeket a fogalmakat a temporális logikák speciális, ún. temporális operátorokkal írják le, amelyek egymásba ágyazhatók, logikai operátorokkal kombinálhatók.

Az alábbiakban definiálom a munkám alapját képező öt LTL temporális operátort:

- **X** („neXt time”) a következő állapotra ír elő feltételt.
- **F** („in the Future”) a feltétel jövőbeli bekövetkezését követeli meg.
- **G** („Globally”) azt fejezi ki, hogy a kifejezés az összes jövőbeli állapotra igaz.
- **U** („Until”) egy kétoperandusú operátor, ami akkor teljesül, ha valamikor a jövőben létezik egy állapot, amire a második operandus igaz, és addig minden állapot kielégíti az első operandust.
- **R** („Release”) az U operátor logikai duálisa, teljesüléséhez a második operandusnak kell teljesülnie mindaddig, míg egy állapotra nem teljesül az első operandus *is*. Az első operandusnak azonban nem kell mindenképpen teljesülnie a jövőben.



1.1.1. ábra. Az LTL temporális operátorok szemléletes jelentése.

Az LTL kifejezések szintaxisa a következő módon definiálható [7]:

- Minden P atomi kifejezés egy állapotkifejezés.
- Ha p és q állapotkifejezések, akkor $\neg p$ és $p \wedge q$ is állapotkifejezés.
- Ha p és q állapotkifejezések, akkor $X p$ és $p U q$ is állapotkifejezések.

A többi operátor a következő szabályok segítségével kifejezhető:

- $p \vee q \equiv \neg(\neg p \wedge \neg q)$
- $p R q \equiv \neg(\neg p U \neg q)$
- $F p \equiv True U p$
- $G p \equiv \neg F \neg p$

Egy LTL kifejezés *negált normál formáján* egy olyan ekvivalens kifejezést értünk, ami-
ben kizárólag a $\wedge$, $\vee$ és $\neg$ Boole-operátorok, valamint az X , U és R temporális operátorok
szerepelhetnek, és negálás csak atomi kijelentések előtt állhat. Ahhoz, hogy egy kifejezést
negált normál formába hozzunk, elsőként át kell írunk az $F \psi$ és $G \psi$ alakú kifejezéseket
az alábbi azonosságok szerint:

- $F \psi \equiv True U \psi$
- $G \psi \equiv False R \psi$

Ezután a megfelelő Boole-algebrai azonosságok segítségével minden logikai operátort
ki kell fejeznünk az *és* ($\wedge$), *vagy* ($\vee$), illetve *nem* ($\neg$) operátorok segítségével. Végül a
negálásokat „be kell vinnünk” az atomi kijelentések elé a De Morgan-azonosságok és az
alábbi három temporális azonosság segítségével:

- $\neg(\mu U \eta) \equiv (\neg\mu) R (\neg\eta)$
- $\neg(\mu R \eta) \equiv (\neg\mu) U (\neg\eta)$
- $\neg X \mu \equiv X (\neg\mu)$

1.3. Automataelméleti háttér

Véges automata alatt egy olyan matematikai számítási modellt értünk, ami a bemenet méretétől független, véges és állandó mennyiségű memóriával rendelkezik. Egy véges automata működhethet véges és végtelen bemeneteken (úgynevezett szavakon).

Formálisan, egy (véges szavakon működő) véges $\mathcal{A}$ automata egy $(\Sigma, Q, \Delta, Q^0, F)$ ötös, ahol

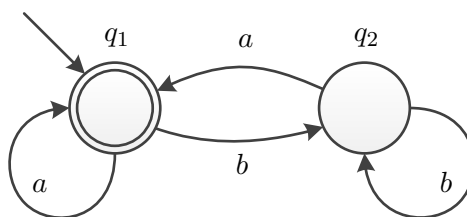
- Σ a véges ábécé.
- Q a lehetséges állapotok véges halmaza.
- $\Delta \subseteq Q \times \Sigma \times Q$ az állapotátmeneti reláció.
- $Q^0 \subseteq Q$ a kiinduló állapotok halmaza.
- $F \subseteq Q$ az elfogadó állapotok halmaza.

Egy automatát grafikusan egy olyan élcímkezett gráffal ábrázolhatunk, amiben a csomópontoknak Q , az éleknek pedig Δ elemei felelnek meg. Az élek címkei Σ elemeiből állnak elő, ezért a betűket sokszor élkifejezéseknek is hívjuk.

Legyen $v \in \Sigma^*$ egy szó, melynek hossza $|v|$. $\rho : \{0, 1, \dots, |v|\} \mapsto Q$ leképezést $\mathcal{A}$ egy lefutásának nevezzük v -n, ahol:

- $\rho(0) \in Q^0$, tehát az első állapot egy kiinduló állapot.
- Az i . állapotból az $i + 1$. állapotba a bemenet i . betűjének olvasásakor akkor léphetünk, ha az átmenet szerepel az állapotátmeneti relációban, vagyis minden $0 \leq i < |v|$ -re $(\rho(i), v(i), \rho(i + 1)) \in \Delta$.

Ekkor azt mondjuk, hogy $\mathcal{A}$ olvassa v -t, vagyis v bemenete $\mathcal{A}$ -nak. Egy ρ lefutást v -n elfogadónak nevezünk, ha egy elfogadó állapotban ér véget, vagyis $\rho(|v|) \in F$. Egy $\mathcal{A}$ automata akkor és csak akkor fogadja el a v szót, ha létezik $\mathcal{A}$ -nak v -n elfogadó lefutása. Az $\mathcal{A}$ által elfogadott $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ nyelv az összes $\mathcal{A}$ által elfogadott szó halmaza.



1.2. ábra. Egy egyszerű véges automata.

1. példa. Tekintsük az 1.2. ábrán látható automatát. A kezdőállapot q_1 , ezt a forrás nélküli nyíl jelöli. Ez egyben az egyetlen elfogadó állapot is, amit a dupla kör jelöl.

Ez az automata elfogadja például az „aabba” szót, mivel ez a bemenet a $q_1q_1q_1q_2q_2q_1$ lefutást eredményezi, aminek utolsó állapota elfogadó állapot.

Az összes elfogadott szó, vagyis az automata nyelve együtt $\varepsilon + (a + b)^*a$ alakban írható fel, ahol $+$ választást jelöl, $*$ pedig tetszőleges, de véges számú ismétlődést. A nyelv tehát az ε üres szóból, vagy olyan szavakból áll, amikben tetszőleges számú „a” vagy „b” után „a” zárja a sort.

Egy véges automata lehet determinisztikus vagy nem-determinisztikus. Előbbi esetben Δ -t egy egyértelmű leképezésként értelmezzük, vagyis $\Delta : Q \times \Sigma \mapsto Q$. Ezzel kikötjük,

hogy egy (q, a, q') és egy (q, a, q'') alakú tranzíció esetén $q' = q''$ mindig teljesül. Nem-determinisztikus esetben $q' \neq q''$ is megengedett. A továbbiakban, ha külön nem jelzem, nem-determinisztikus automatákkal foglalkozunk.

Mint az már korábban is felmerült, az általam vizsgált rendszerek többsége rendeltetésszerű működése során nem áll le. A továbbiakban ezért bemutatom azokat a végtelen szavakon működő automata osztályokat is, amik ilyen rendszerek modellezésére alkalmazhatók. Ezeknek az automatáknak a struktúrája megegyezik a véges szavakon működő automatákéval, azonban Σ^ω -beli szavakat ismernek fel, ahol az ω felső index a szóban szereplő végtelen számú betűre, vagyis állapotátmenetre utal.

1.3.1. Büchi automaták

A legegyszerűbb végtelen szavakon működő véges automaták a Büchi [7] automaták. Egy Büchi automatának ugyanolyan komponensei vannak, mint egy véges szavakon működő automatának. Egy $\mathcal{A}$ Büchi automata egy lefutása $v \in \Sigma^\omega$ -n is majdnem ugyanúgy kerül definiálásra, ezúttal azonban $|v| = \omega$. Így a ρ lefutás értelmezési tartománya a természetes számok halmaza, vagyis $\rho : \mathbb{N} \mapsto Q$.

Jelöljük $\inf(\rho)$ -val azoknak az állapotoknak a halmazát, amelyek végtelenül gyakran szerepelnek egy lefutásban. Egy ρ lefutás $\mathcal{A}$ -n akkor és csak akkor *elfogadó*, ha van olyan elfogadó állapot, ami végtelenül gyakran jelenik meg ρ -ban, vagyis $\inf(\rho) \cap F \neq \emptyset$.

A Büchi automaták ún. ω -reguláris nyelveket fogadnak el. Ugyanilyen nyelveket képesek leírni az LTL kifejezések is, azonban a Büchi automaták kifejezőereje nagyobb. Elmondható, hogy egy LTL kifejezéshez mindig létezik azonos szavakat elfogadó Büchi-automata, azonban nem minden Büchi-automatához alkotható ekvivalens LTL kifejezés [15].

2. példa. Az 1.2. ábrán látható automata Büchi automataként is értelmezhető. Ebben az esetben elfogadja például az $(ab)^\omega$ szót, ami „a”-k és „b”-k „a”-val kezdődő végtelen hosszú váltakozása. Az automata által elfogadott nyelv tartalmaz minden olyan szót, ami végtelenül sok a-t tartalmaz. Ezeket a szavakat a $(b^*a)^\omega$ ω -reguláris kifejezés írja le.

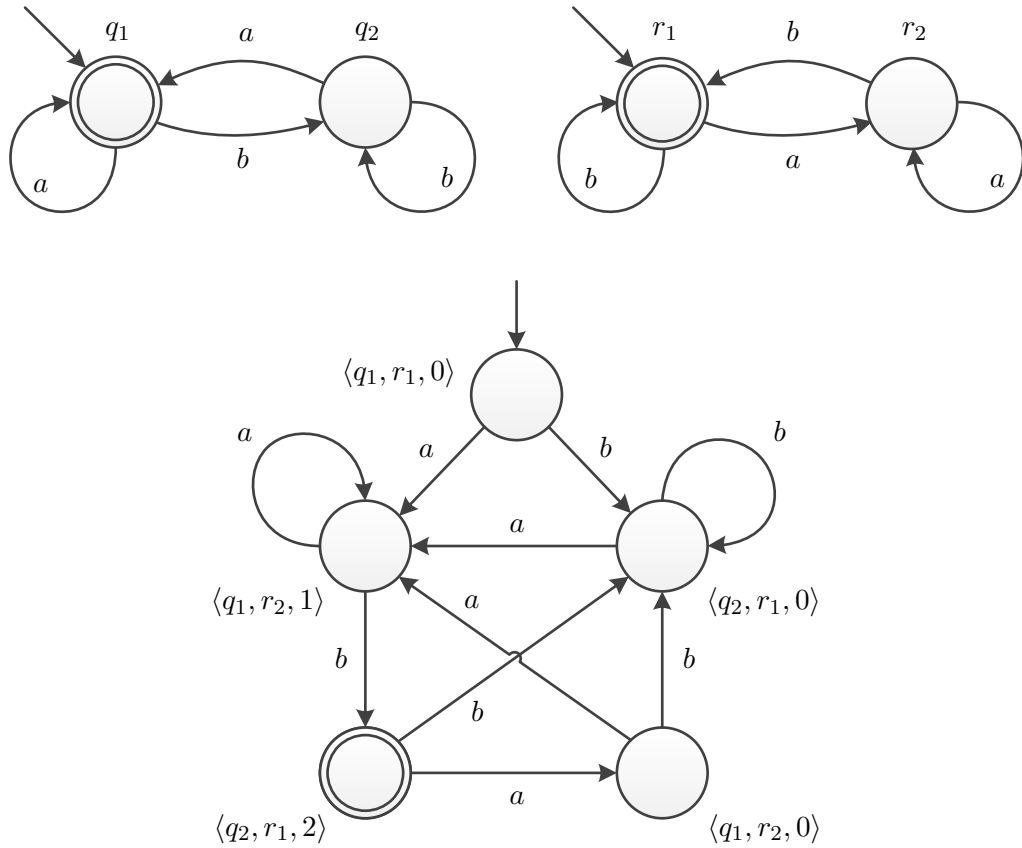
1.3.2. Automaták szinkron szorzata

Két egyszerű Büchi automata *szinkron szorzatán* egy olyan automatát értünk, ami pontosan azokat a szavakat fogadja el, amelyeket mindkét automata elfogad. Formálisan, ha adottak $\mathcal{A}_1 = \langle \Sigma, Q_1, \Delta_1, Q_1^0, F_1 \rangle$ és $\mathcal{A}_2 = \langle \Sigma, Q_2, \Delta_2, Q_2^0, F_2 \rangle$ Büchi automaták, a szorzat automata által elfogadott nyelv $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$, az automata pedig

$$\mathcal{A}_1 \cap \mathcal{A}_2 = \langle \Sigma, Q_1 \times Q_2 \times \{0, 1, 2\}, \Delta, Q_1^0 \times Q_2^0 \times \{0\}, Q_1 \times Q_2 \times \{2\} \rangle$$

alakban adódik. A tranzíciókat tekintve $(\langle r_i, q_j, x \rangle, a, \langle r_m, q_n, y \rangle) \in \Delta$ akkor és csak akkor áll fenn, ha

- $(r_i, a, r_m) \in \Delta_1$ és $(q_j, a, q_n) \in \Delta_2$, tehát a megfelelő állapotátmenetek a két szorzandó automatában is megléphetők
- a harmadik komponens állapota az $\mathcal{A}_1$ és $\mathcal{A}_2$ elfogadó állapotaitól függ:
 - ha $x = 0$ és $r_m \in F_1$, akkor $y = 1$
 - ha $x = 1$ és $q_n \in F_2$, akkor $y = 2$
 - ha $x = 2$, akkor $y = 0$
 - egyébként $y = x$.



1.3. ábra. Két Büchi automata és szinkron szorzatuk [7]. A szorzat automatában csak az elérhető állapotok szerepelnek.

A harmadik komponens biztosítja, hogy a két automata elfogadó állapotai közül mindkettőből végtelenül gyakran jelenjen meg állapot. Önmagában az $F = F_1 \times F_2$ megkötés nem lenne elégséges, mivel a szorzat automatát a két szorzandó automata olyan együtteseként kell elképzelnünk, ahol mindkettő egyszerre lép a bemenet hatására. A szorzat akkor fogad el egy szót, ha $\mathcal{A}_1$ és $\mathcal{A}_2$ is elfogadja. Ez viszont nem jelenti azt, hogy a két automatának bármikor is egyszerre kellene elfogadó állapotban lennie, elég, ha mindkettő végtelen gyakran halad át ilyen állapoton (pl. felváltva).

A harmadik komponens kezdetben 0. Akkor változik 1-re, ha megjelenik egy állapot az első automata elfogadó állapotai közül. Ha ekkor a második automata elfogadó állapotai közül is érkezik egy állapot, akkor a harmadik komponens 2-re növekszik, és megkapjuk a szorzat automata egy elfogadó állapotát. A következő állapotban a számláló visszatér 0-ba, és újra az első automata elfogadó állapotait várjuk. A szorzat automata akkor fogad el egy lefutást, ha az végtelen sok olyan állapotot tartalmaz, amiben a harmadik komponens értéke 2.

Ezzel biztosítottuk, hogy *mindkét* automata elfogadó állapotai végtelen gyakran jelennek meg. Ha valamely ponton az egyik automatától nem találunk több elfogadó állapotot, akkor az az automata nem fogadja el a szót. A számláló nem növekszik tovább, így a szorzat automata lefutása sem lesz elfogadó.

Megjegyzendő, hogy az így kapott szorzat automata a két állapottér és a $\{0, 1, 2\}$ halmaz Descartes szorzata miatt olyan állapotokat is tartalmaz, amik nem elérhetők a kezdőállapotokból. A gyakorlati alkalmazás során ezeket érdemes felderíteni és eltávolítani.

Szerencsére a szorzat automata előállítása ennél sokkal egyszerűbb, ha az egyik automata minden állapota elfogadó állapot. Ilyen automatát kapunk például, ha egy modell

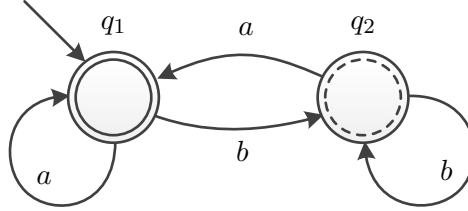
állapotterét automataként reprezentáljuk, ami – mint később bemutatom – éppen így is lesz az LTL modellellenőrzés során.

A szorzat automata ilyenkor $\mathcal{A}_1 \cap \mathcal{A}_2 = \langle \Sigma, Q_1 \times Q_2, \Delta', Q_1^0 \times Q_2^0, Q_1 \times F_2 \rangle$ alakú. Az elfogadó állapotok $Q_1 \times F_2$ -beli párok, amikben a második tag a második automata egy elfogadó állapota. Az állapotátmenetek esetében $(\langle r_i, q_j \rangle, a, \langle r_m, q_n \rangle) \in \Delta'$ akkor és csak akkor áll fenn, ha $(r_i, a, r_m) \in \Delta_1$ és $(q_j, a, q_n) \in \Delta_2$.

1.3.3. Általánosított Büchi automaták

Időnként kényelmesebb lehet a Büchi automaták egy olyan osztályát használni, amelyben több elfogadó állapot halmaz van. Ez nem befolyásolja az automata kifejezőerejét, de kompaktabb reprezentációt tesz lehetővé. [7]

Egy *általánosított Büchi automata* elfogadó állapotokat kódoló komponense $F \subseteq 2^Q$ alakú, tehát Q valamely részhalmazainak halmaza. Egy ilyen automata valamely ρ lefutása akkor és csak akkor elfogadó, ha minden $P_i \in F$ -re $\inf(\rho) \cap P_i \neq \emptyset$. Ez azt jelenti, hogy minden elfogadó állapot halmaz legalább egy elemének végtelenül gyakran kell előfordulnia ρ -ban. Vegyük észre, hogy ez egyben azzal is jár, hogy *üres F esetén az automata minden lefutása elfogadó.*



1.4. ábra. Egy Büchi automata.

3. példa. Tekintsük most az 1.4. ábrát, amin az előző automatához képest a q_2 állapot most egy második elfogadó állapot halmazhoz tartozik.

Az automata most azokat a szavakat fogadja el, amiben végtelenül gyakran szerepel q_1 és q_2 is. Ezek a szavak az $(a^+b^+)^{\omega}$ alakúak, ahol a^+ tetszőlegesen véges sok, de legalább egy megjelenést jelent. Ez a kifejezés lényegében azt írja le, hogy egyik betűből sem lehet végtelenül sok közvetlenül egymás után, de az egész szóban mindkettőnek végtelenül sokszor kell szerepelnie.

Egy általánosított Büchi automata átalakítható egy egyszerű Büchi automatává. Az $\mathcal{A} = \langle \Sigma, Q, \Delta, Q^0, F \rangle$ általánosított Büchi automata „hagyományos” megfelelője $F = \{P_1, \dots, P_n\}$ mellett $\mathcal{A}' = \langle \Sigma, Q \times \{0, \dots, n\}, \Delta', Q^0 \times \{0\}, Q \times \{n\} \rangle$.

A Δ' állapotátmeneti reláció úgy épül fel, hogy $(\langle q, x \rangle, a, \langle q', y \rangle) \in \Delta'$ akkor és csak akkor áll fenn, ha $(q, a, q') \in \Delta$, x -re és y -ra pedig a következő szabályok érvényesek:

- Ha $q' \in P_i$ és $x = i - 1$, akkor $y = i$.
- Ha $x = n$, akkor $y = 0$.
- Minden egyéb esetben $y = x$.

Az elv nagyon hasonló a szorzat automaták elkészítésénél látottakhoz: a második komponens felelős azért, hogy minden elfogadó állapothalmazból végtelenül gyakran szerepeljenek állapotok. Akárcsak akkor, most is érdemes a kiinduló állapotokból nem elérhető állapotokat felderíteni és eltávolítani. Az átalakítás az automata méretét legfeljebb $n + 1$ -szeresére növeli.

1.3.4. Transzformációs algoritmus

Gerth, Peled, Vardi és Wolper algoritmus [9] lehetővé teszi egy negált normál formában lévő φ LTL kifejezés általánosított Büchi automatává transzformálását. Terjedelmi okokból az algoritmus részletes bemutatására nincs lehetőség, azonban a függelékben megtalálható a működését leíró pszeudokód.

Az algoritmus alapvető adata eleme a *csomópont*, ez reprezentálja a keletkező automata állapotait. Egy q csomópont a következő mezőkből áll:

- *ID* : NodeID - A csomópont egyedi azonosítója.
- *Incoming* : NodeID lista - A forrás-csomópontok listája. Minden eleme egy olyan csomópontot jelöl, amiből él mutat q -ba.
- *Old* : Formula lista - A már feldolgozott kifejezéseket tartalmazza.
- *New* : Formula lista - A még fel nem dolgozott kifejezések listája.
- *Next* : Formula lista - A következő csomópontban feldolgozandó formulák gyűjteménye.

A csomópontok a temporális kifejezés kiértékelésének fázisait jelentik. Az algoritmus futása során összegyűjti a már elkészült csomópontokat egy *Nodes* nevű listában, belőlük keletkeznek az automata állapotai. Létezik még egy speciális *init* csomópont is, ami az automata kezdőállapota lesz. A *Nodes* lista kezdetben üres.

Az algoritmus az EXPAND függvény segítségével rekurzívan feldolgozza a keletkező csomópontokat és létrehozza az ún. *tablót* [7]. Ez egy, a kiértékelés menetét ábrázoló fa struktúra, amelyben megjelennek az ismétlődések is az ismétlődő kifejezés elejére visszamutató élekként.

Egy csomópont feldolgozásakor az algoritmusnak alapvetően három célja van.

- A diszjunkciók mentén szétbontja az állapotot több lehetséges esetre.
- Meghatározza, hogy az atomi kijelentések milyen hozzárendelése esetén érthető el a reprezentált állapot.
- Meghatározza, hogy milyen kifejezéseknek kell teljesülnie a következő állapot(ok)ban.

Kezdetben a *New* mező tartalmazza a feldolgozásra váró kifejezéseket. Ezek a kifejezések (az első csomópont kivételével) az előzőleg befejezett csomópont *Next* mezőjéből származnak. Az algoritmus a kifejezéseket a fő operátoruk szerint dolgozza fel:

- Ponált és negált atomi kijelentések esetén¹ ellenőrzi, hogy azok nincsenek-e ellentmondásban már feldolgozott formulákkal. Minden olyan csomópont eldobásra kerül, amiben p -nek és $\neg p$ -nek is teljesülnie kellene.
- Konjunkciók esetén a két operandust egyenként visszateszi a *New* mezőbe feldolgozásra, hiszen ezeknek egyszerre kell teljesülniük a csomópont által reprezentált későbbi automata állapot(ok)ban.
- Diszjunkciók esetén az algoritmus a csomópontot két részre bontja, mivel bármelyik operandus teljesülése elegendő, ez pedig két lehetséges utat jelent.
- X operátor esetén az operandus a *Next* mezőbe kerül, és a következő csomópontban kerül feldolgozásra.

¹Emlékezzünk, hogy a negált normál forma miatt negálás csak atomi kijelentés előtt állhat.

- Az U és R operátorokat az algoritmus rekurzív képletük segítségével visszavezeti az előbbi operátorokra.

Egy csomópont akkor tekinthető véglegesnek, ha a *New* mezőben szereplő összes kifejezés feldolgozásra került. Ekkor az algoritmus a csomópontot a *Nodes* listába illeszti, szükség esetén összevonva egy már létező, azonos jelentésű csomóponttal - így jöhetnek létre visszacsatolások a „fában”.

Az így elkészített *Nodes*-beli csomópontok általánosított Büchi automatává konvertálhatók:

- A Σ ábécé a φ -ben szereplő propozíciókból képzett halmazokból áll. Egy $\alpha \in \Sigma$ a propozíciók egy olyan értékadásának felel meg, ahol az α -ban szereplő propozíciók igazságtartalma *True*, az α -ban nem szereplőké pedig *False*. A gyakorlatban ezek a halmazok Boole-algebrai kifejezésekkel is reprezentálhatók.
- Az automata Q állapothalmaza a *Nodes* listában található csomópontokból, valamint az *init* csomópontból áll.
- $(r, \alpha, r') \in \Delta$ akkor és csak akkor, ha $r \in Incoming(r')$ és α igazgá teszi az *Old*(r')-ben lévő *ponált és negált propozíciók konjunkcióját*.
- A Q_0 kezdőállapothalmaz egyetlen eleme az *init* csomópont.
- Az F elfogadó komponens minden $\mu \cup \psi$ alakú részkifejezéshez tartalmaz egy $P_i \in F$ állapothalmazt - P_i minden olyan r állapotot tartalmaz, amire $\psi \in Old(r)$ vagy $\mu \cup \psi \notin Old(r)$ teljesül. Ezek azok az állapotok, amikben a részkifejezés már teljesült, így biztosítható, hogy egy végtelen hosszú μ sorozatot nem fogad el az automata, viszont ha a részkifejezés egyszer már teljesült egy elfogadó lefutásban, akkor abban a lefutásban minden további állapotra is teljesülni fog. Ha nincs $\mu \cup \psi$ alakú részkifejezés, akkor F üres lesz, ami - mint az előző fejezetben bemutattam - hagyományos automaták esetén az $F = Q$ elfogadó állapot halmaznak felel meg.

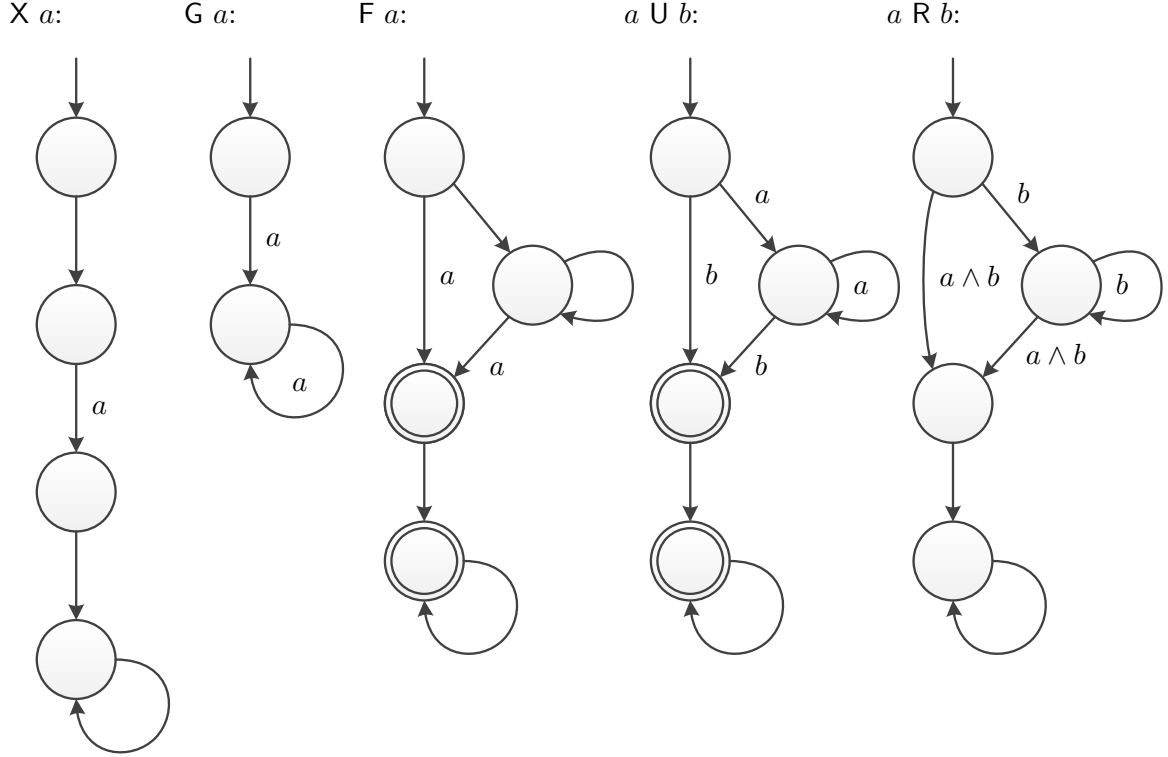
Az így készített automaták mérete és elkészítésük ideje legrosszabb esetben φ méretében exponenciális, azonban a tapasztalatok [7] azt mutatják, hogy a keletkező automata mérete általában kicsi.

A bemutatott algoritmusnak van egy olyan tulajdonsága, ami kiemelten fontos lesz a szaturáció kiterjesztésekor. A konstrukcióból adódóan a keletkező automata egy állapótába *kizárólag ugyanolyan címkéjű élek* futnak be. Ez akkor sem romlik el, ha a kapott általánosított Büchi automatát az előbb bemutatott módon egyszerű Büchi automatává alakítjuk. A 2.2. fejezetben bemutatott megoldások ezt a redundanciát fogják kihasználni.

1.4. Döntési diagramok

A *döntési diagramok* [1] olyan gráfok, amik többváltozós függvények kompakt reprezentálását teszik lehetővé. A döntési fák redukciója során kapjuk meg őket, ahol a fában lévő azonos jelentésű csomópontokat összevonjuk. Ezzel jelentős mennyiségű csomópontot spórolhatunk meg, információvesztés nélkül.

Formálisan egy *többértékű döntési diagram* (Multiple-valued Decision Diagram, MDD) egy olyan körmentes irányított gráf, ami egy K változóból álló $f : \{0, 1, \dots\}^K \rightarrow \{0, 1\}$ függvényt reprezentál [2]. Egy MDD csomópontjait $K + 1$ szintre osztjuk, a 0. szinten lévő két csomópontot terminális csomópontoknak (*terminális 0* és *terminális 1*), a többi nemterminális csomópontoknak nevezzük. Mindegyik $k > 0$ szinthez (illetve a hozzá tartozó változóhoz) rendelünk egy véges D_k értelmezési tartományt, aminek mérete n_k . Az egyszerűség kedvéért D_k elemeit az általánosság megszorítása nélkül vegyük

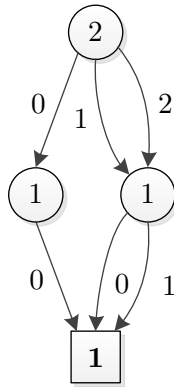


1.5. ábra. Az LTL temporális operátoraiból generált általánosított Büchi automaták. Az éleken proposíciók konjunkciói láthatók.

$D_k = \{0, 1, \dots, n_k - 1\}$ -nek. A k . szinten lévő csomópontok mindegyikéből pontosan n_k él fut $k - 1$ szintű csomópontokba.

Egy MDD *kanonikus*, ha nem létezik két azonos k . szintű csomópont, amiknek $k - 1$. szintű szomszédai megegyeznek. Egy MDD *kváziredukált*, ha a szintek között is megengedjük a redundáns csomópontokat, vagyis egy csomópont minden éle vezethet ugyanabba az alacsonyabb szintű csomópontba, *redukált*, ha ilyen csomópontok nem lehetnek, és az élek közvetlenül vezethetnek $k - 1$ -nél alacsonyabb szintű csomópontba is.

Az MDD által kódolt függvény kiértékeléséhez az K . szinten lévő egyetlen r *gyökér csomópont*ból elindulva kell bejárnunk a gráfot, minden lépésnél a k . változó értékének megfelelő élen tovább haladva. Az eredmény annak a terminális csomópontnak az értéke (0 vagy 1), amibe a 0. szinten érkezünk. Formálisan, ha egy v csomópont i . gyerekét $v[i]$ -vel jelöljük, akkor $f(x_n, x_{n-1}, \dots, x_1) = 1 \Leftrightarrow r[x_n][x_{n-1}] \dots [x_1] = \mathbf{1}$.



1.6. ábra. Egy többértékű döntési diagram.

4. példa. Az MDD-k által kódolt függvények vektorhalmazok kifejezésére is alkalmazhatók. Ekkor a kiértékelésük eredményét úgy interpretáljuk, hogy az adott paraméterezés, mint vektor, része-e a halmaznak. Példaként tekintsük az 1.6. ábrán látható döntési diagramot, aminek két változója x_1 és x_2 , értelmezési tartományuk pedig $D_1 = \{0, 1\}$ és $D_2 = \{0, 1, 2\}$.

Az ábrán csak a terminális egyesbe mutató éleket ábrázoltam. A kódolt halmaz a $\{(0, 0), (0, 1), (0, 2), (1, 1), (1, 2)\}$ elemekből áll, ezek az értékek teszik a kódolt függvény értékét eggyé. Az összes lehetséges kombináció közül egyedül az $(1, 0)$ nem szerepel a halmazban. Ne feledjük, hogy a párosok a diagramon most lentől felfelé olvashatók ki, mivel az első változó van a legalsó szinten.

1.5. Szaturáció

Egy modell állapotterének felderítésére több módszer létezik. A legegyszerűbb eset az *explicit állapotter-generálás*, amikor a kiinduló állapotból az állapotátmeneteken egyenként lépkedve valamilyen gráfbejáró algoritmus segítségével sorra vesszük és eltároljuk az elérhető állapotokat. Ez a módszer nagyon gyorsan korlátokba ütközik, mivel az állapotok egyenkénti tárolása és kezelése rendkívül sok erőforrást igényel.

Szimbolikus állapotter-generálás alatt olyan technikákat értünk, amik az állapotokat valamilyen kódolással, kompaktan tárolják. A hagyományos megoldások állapotváltozók segítségével kódolják az állapotokat, egy állapot $(v_1, v_2, \dots, v_n)$ alakú vektor. Az állapotteret leíró struktúrák a változók értelmezési tartományától függően lehetnek például bináris vagy többértékű döntési diagramok, amiknek a segítségével már nagyobb állapotterek is tárolhatók és kezelhetők.

A *szaturációs algoritmus* [2] egy olyan új iterációs stratégia, amelynek segítségével akár hatalmas állapottereket is be lehet járni és az állapotteret kódolva, döntési diagram formájában lehet reprezentálni. A módszer erejét az adja, hogy a bejárás alkalmazkodik az MDD-k struktúrájához, és kihasználja a modellben előforduló *lokalitásokat*. Emiatt a szaturációs algoritmus rendkívül jól teljesít *aszinkron rendszerek* állapottereinek a felderítésében, mivel egy aszinkron rendszerben az események jellemzően lokálisak, a modellnek csak egy kis részét érintik.

1.5.1. A modell dekomponálása

A lokalitás minél hatékonyabb kihasználása érdekében a szaturációs algoritmus első lépése a modell dekomponálása. Ez azt jelenti, hogy a modellt n darab diszjunkt részre osztjuk, ami Petri-hálók [11] esetében például hely halmazok képzését jelenti. Az egyes komponensek lokális állapotainak halmaza lesz a globális állapot egyes változóinak értelmezési tartománya. Egy globális állapot az n komponens lokális állapotaiból képzett vektorként áll elő.

1.5.2. Az állapotter leírása

Az állapotteret így könnyedén ábrázolhatjuk egy $K + 1$ szintű MDD-ben, ahol $K = n$, tehát minden változóra egy szint jut. Az MDD egy olyan függvényt fog kódolni, aminek bemeneti paraméterei a lokális állapotváltozók, értéke pedig 1, ha az adott globális állapot szerepel az állapotterben, illetve 0, ha nem.

1.5.3. Események

Egy-egy modellbeli esemény tipikusan csak kevés komponens állapotát érinti. Ezt kihasználva az állapottér felderítés hatékonyabbá tehető, ha egy eseménnyel csak azokban a komponensekben foglalkozunk, amikre ténylegesen hatással vannak.

Az események hatása tehát korlátok közé szorul, ennek kifejezésére Top_e -vel jelöljük a legmagasabb sorszámú, Bot_e -vel a legalacsonyabb sorszámú komponenseket, amiket az esemény érint. Ezen az intervallumon kívül egyáltalán nem foglalkozunk az eseménnyel.

1.5.4. Next-state függvény

A *next-state függvény* ($\mathcal{N}$) határozza meg a modell állapotátmeneteit. $\mathcal{N}(s)$ megadja az s állapotból egy lépéssel elérhető állapotok halmazát. $\mathcal{N}(S)$ megadja az S állapothalmazból egy lépéssel elérhető állapotok halmazát. Az egyes eseményekhez is társítható egy-egy $\mathcal{N}_e$ next-state függvény, ekkor a teljes állapotátmenet reláció: $\mathcal{N} = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e$, ahol $\mathcal{E}$ a modellbeli események halmaza.

Az események next-state függvényei is dekomponálhatók az egyes szinteket érintő $\mathcal{N}_{(i,e)}$ részekre. Ezek azt adják meg, hogy egy esemény eltűzelése milyen hatással van egy adott komponens lokális állapotára. A next-state függvényt az állapottérhez hasonlóan szintén egy MDD fogja kódolni, ennek azonban $2 \cdot K + 1$ szintje van. A $2 \cdot k + 1$ alakú szintek élei a kiinduló, a $2 \cdot k$ alakúak élei pedig a végállapotot adják meg. A teljes next-state függvény egy MDD-ben történő előállítás nem szükséges, elegendő minden eseményhez külön diagramban számon tartani a hozzájuk tartozó átmeneteket. Az események eltűzeléséhez azonban a szintekhez tartozó komponensekből elő kell állítani az esemény next state MDD-jét, ami az $\mathcal{N}_{(i,e)}$ -ket kódoló MDD-k metszeteként adódik [5].

1.5.5. A szaturációs algoritmus működése

Az állapottér-generálás problémája nem más, mint az $\mathcal{N}$ next-state függvény $\mathcal{N}^*$ *transzitiv lezártjának* meghatározása a kezdőállapotokból kiindulva. A szaturációs algoritmus a transzitiv lezártat csomópontról csomópontra képzi az MDD-ben felfelé haladva, így felderítés közben az állapottér kódoló MDD maximális mérete várhatóan nem lesz sokkal nagyobb a felderítés során, mint a végén. Ez nagy előny a tradicionális szélességi bejárás alapú szimbolikus algoritmusokhoz képest, amelyek jellemzően sokkal nagyobb köztes állapottér reprezentációt építettek, mint a végső állapottér reprezentáció [7].

Jelölje $\mathcal{B}(k, p)$ az állapotok azon halmazát, amelyeket a k . szinten lévő p csomópont által meghatározott rész-MDD kódol². Legyen $\mathcal{E}_k$ azoknak az e eseményeknek a halmaza, amikre $Top_e \leq k$. Azt mondjuk, hogy egy k . szinten lévő p csomópont *szaturált*, ha $\mathcal{B}(k, p) = \bigcup_{e \in \mathcal{E}_k} \mathcal{N}_e^*(\mathcal{B}(k, p))$, tehát eddig a szintig nincs több olyan esemény, aminek eltűzelésével új állapotot tudnánk felfedezni. A $\mathcal{B}(k, p)$ állapothalmaz *zárt* $\mathcal{E}_k$ -ra nézve. Az állapottér-generálás véget ér, amikor a legfelső szinten lévő gyökér csomópont szaturálttá válik, ekkor az állapottér $S = \mathcal{N}^*(s_0)$.

Az algoritmus kezdetekor megépítjük a kezdeti állapotot leíró MDD-t. Elsőként az 1. szinten lévő csomópontokat szaturáljuk, azaz megpróbálunk eltűzelni rájuk minden olyan e eseményt, amire $Top_e = 1$. Ezután a következő, 2. szinten lévő csomópontok következnek azokkal az eseményekkel, amikre $Top_e = 2$. Ha az első szinten új csomópont jött létre, azt azonnal, rekurzívan szaturáljuk. Az algoritmus végigjár minden k szintet a $Top_e = k$ tulajdonságú eseményekkel, azonnal szaturálva az alacsonyabb szinteken keletkező csomópontokat is. Ha a legfelsőbb szinten lévő csomópont is szaturálttá válik, az állapottér-generálás befejeződött [2].

²A továbbiakban az *adott csomópont által kódolt állapotok* alatt ugyanezt fogom érteni.

1.5.6. Vezérelt szaturáció

Sokszor az állapottér bejárás során csak bizonyos feltételeknek megfelelő, azaz kényszereket kielégítő állapotokon szeretnénk áthaladni. Ennek megoldására több lehetőség is létezik, amelyek közül egy hatékony módszer az úgynevezett *vezérelt szaturációs* [17] algoritmus. Ez egy olyan komplex iterációs stratégia, ami adott $\mathcal{C}$ kényszerre, adott S_0 állapothalmazra és adott $\mathcal{N}$ állapotátmenet relációval hatékonyan képezi az alábbi állapothalmazt: $S = (\mathcal{N}(S_0) \cap \mathcal{C})^*$. Mint jól látható, általános esetben halmazok metszetét kell kiszámítanunk, ráadásul vagy minden lépés után egy költséges halmazműveletet kell végrehajtani, vagy pedig a korlátozott lépéseket el kell kódolni az $\mathcal{N}$ relációba, ami szintén költséges [17]. A vezérelt szaturáció célja, hogy megakadályozzon bizonyos lépéseket az állapottér-generálás közben. Ezt korábbi megoldások CTL modellellenőrzés során használták ki. Az U operátor számítását sikerült felgyorsítani vele, ahol is az előre felderített állapottéren kellett az állapottér alapján képezett kényszer segítségével iterálni. Én munkám során ettől eltérően, a szinkron szorzat automata vezérlésére fogom alkalmazni a módszert.

Az algoritmus alapgondolata, hogy a szaturációs algoritmusnak megadunk egy *kényszert*, ami egy ugyanolyan struktúrájú MDD, mint az állapottér kódoló diagram, és azokat az állapotokat adja meg, amikbe *szabad* lépni³. A szaturációs algoritmust ilyenkor nem az 1., hanem a $K + 1$. szinten kezdjük, és az adott csomópont szaturálása előtt először szaturáljuk az alacsonyabb szinten lévő csomópontokat. Az elsőként szaturált csomópont így is a legalsó szint egyik csomópontja lesz, de ahogy rekurzívan bejárjuk az MDD-t, a kényszer MDD-n is lehetőség van ugyanazon élek mentén lépni. A kívánt eredményt azzal érjük el, hogy egy esemény eltűzésével keletkező új állapotot csak akkor vesszük fel ténylegesen (vagyis húzzuk be az azt alkotó út utolsó élét a terminális 1-es csomópontba), ha a kényszerben az állapotot reprezentáló út is a terminális 1-esbe vezet. Az MDD-k által kódolt halmazokat tekintve ez éppen a metszés műveletet jelenti.

A kényszer lényegében „megjegyzi” az utat, amin át egy bizonyos csomópontba jutottunk (egy csomópontot egyszerre mindig csak egy úton vizsgálunk), ezáltal lehetővé teszi, hogy a lokális feladása nélkül globális állapotokról is dönthessünk. Ezt a tulajdonságát fogom kihasználni a 2.2.3. fejezetben bemutatásra kerülő új modellellenőrző algoritmusban.

³Elméletileg lehetőség van más szemantika alkalmazására is, például megadhatunk olyan állapotokat is, amibe *nem szabad* lépni. Jelen munka keretében a vezérelt szaturációs algoritmust én is egy eltérő szemantikájú kényszerrel fogom használni.

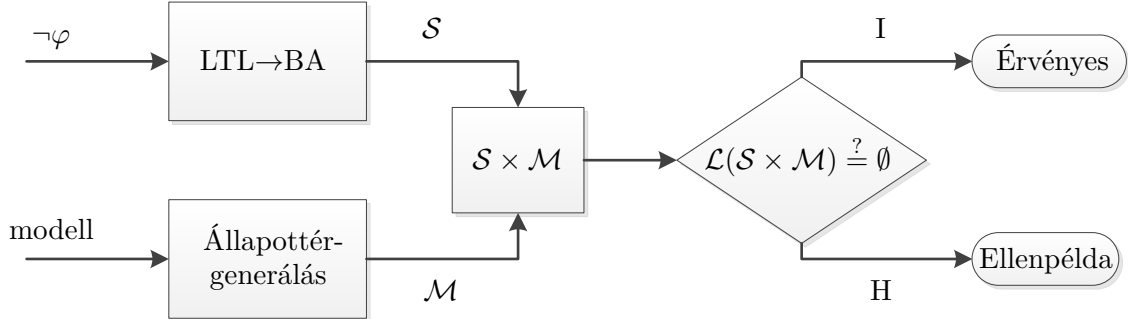
2.

LTL modellellenőrzés

Ebben a fejezetben mutatom be a munkám elméleti eredményét adó szaturáció alapú LTL modellellenőrző algoritmust. Elsőként adok egy áttekintést az algoritmus vázlatos működéséről (2.1. fejezet), majd a következő szakaszokban részletezem az egyes fázisokat (2.2-2.3. fejezetek). Végül adok néhány optimalizációs lehetőséget (2.4. fejezet) és összefoglalom a bemutatott részleteket (2.5. fejezet).

2.1. Áttekintés

Az LTL temporális logika könnyű használhatósága és intuitív jellege miatt többféle modellellenőrző algoritmust is kifejlesztettek már hozzá. Ezek jellemzően automataelméleti megközelítést alkalmaznak [9].



2.1. ábra. LTL modellellenőrzés automatákkal.

Az LTL modellellenőrzés automataelméleti megközelítés esetén a következő módon vázolható nagyvonalakban:

1. Adott a vizsgálandó kifejezés. Ennek képezzük a negáltját, és építünk egy *specifikációs automatát*, aminek ábécéje a vizsgálandó modell állapotaiból (pontosabban az azokra érvényes állapotkifejezésekből) áll, és az elfogadott nyelve megegyezik az LTL kifejezést nem kielégítő állapotsorozatokkal (szavakkal).
2. A vizsgálandó modell állapotterét szintén automataként reprezentálva számítsuk ki a két automata szinkron szorzatát olyan módon, hogy a modell állapotváltozásai a célállapottal léptetjük a specifikációs automatát.
3. Vizsgáljuk meg, hogy a *szorzat automata* által elfogadott nyelv üres-e. Ha üres, akkor a modell nem tartalmaz olyan lefutást, mely kielégítené a vizsgálandó kifejezés negáltját, tehát a kifejezés érvényes a modellre. Ha nem üres, az adódó nyelv szavai

megfelelnek azoknak a lefutásoknak, melyek megsértik a specifikációt, tehát értékes ellenpéldákhoz jutottunk.

LTL modellellenőrzés során az állapotteret többféle módon reprezentálhatjuk, járhatjuk be. Léteznek explicit állapotbejárást alkalmazó algoritmusok, amelyek egyenként bejárják az állapottérben fellelhető állapotokat. Ha az állapottér túl nagy, akkor hamar erőforrás korlátokba ütköznek ezek az algoritmusok, még abban az esetben is, ha redukációs technikákat használnak (pl. [10]).

Az utóbbi időben megjelentek a szimbolikus LTL modellellenőrző algoritmusok is, amelyek bizonyították hatékonyságukat [12]. Ezen algoritmusok szélességi állapottér-bejárást használnak, és kódolva tárolják el az egyes állapotokat, lehetővé téve nagy állapothalmazok kezelését. A szaturációs algoritmus is egy szimbolikus technika, ami egy speciális iterációs stratégiával járja be az állapotokat. Ezen tulajdonságai miatt különösen hatékony elsősorban aszinkron rendszerek esetén; azonban ezen tulajdonságok miatt nehéz az algoritmust LTL modellellenőrzésre használni. Szaturációs alapokon korábban nem készítették LTL modellellenőrző algoritmust.

2.1.1. Algoritmikus problémák

Az LTL modellellenőrzési probléma a PSPACE-nehéz komplexitási osztályba tartozik, tehát praktikusán nem várható el egy algoritmustól sem, hogy minden esetre hatékonyan fusson. Egy S állapotterű rendszer esetén a φ LTL formula ellenőrzésének idő komplexitása $2^{O(|\varphi|)}O(|S|)$ nagyságrendű [14].

Két alapvető probléma merül fel. Az algoritmus, amelyet be fogok mutatni a specifikációs automata felépítéséhez, a kiinduló LTL kifejezés hosszában exponenciálisan sok állapotot hoz létre, és időbeli komplexitása is hasonlóan alakul. Az, hogy a negált kifejezésre alkalmazzuk a transzformációt, további méretnövekedéssel jár, egy általánosított Büchi automata átalakítása Büchi automatává pedig még tovább növeli az adódó automata méretét.

A másik alapvető probléma az állapottér mérete. Az ún. *állapottér-robbanás* azt a jelenséget takarja, hogy egy viszonylag egyszerű modellnek is rendkívül sok állapota lehet. Aszinkron rendszerek esetén jellemzően a komponensek számának növelésével exponenciálisan nő az elérhető állapotok száma. Vegyük például az ún. étkező filozófusok [6] modelljét, melyben a bonyolultságot könnyen lehet szabályozni a filozófusok számával. Tíz filozófus esetén a modellnek még csak $1,8 \cdot 10^6$ állapota van, száz filozófus esetén már 10^{62} állapot adódik, míg ezer filozófussal a modell állapottere 10^{629} méretű. Az LTL modellellenőrzés során képezni kell a kifejezésből előállított automata és az állapottér szinkron szorzatát, amely azt eredményezi, hogy önmagukban azok az algoritmusok, amelyek az egyik komponens állapotterét hatékonyan bejárnák, jellemzően innentől nem hatékonyak a két automata eltérő karakterisztikája miatt. A szorzat automata állapottérének mérete akár a két automata méretének a szorzata is lehet, amely sokkal bonyolultabb állapottér reprezentációt eredményez, mintha külön-külön kéne megvizsgálni őket.

A korábbi explicit LTL modellellenőrző algoritmusok legfontosabb jellemzői:

- Az állapotokat explicit módon tárolják, az állapottér-robbanás hamar problémát okoz.
- Hatékonyan vizsgálják menet közben a szinkron szorzat automata állapotterét, „on-the-fly”, azaz menet közbeni modellellenőrzést tudnak végezni.
- Számos redukció és egyszerűsítés alkalmazható, ilyenek például a részleges rendezéses algoritmusok, amelyek valamelyest csökkentik az állapottér-robbanás problémáját azon az áron, hogy csak csökkentett kifejezőerejű LTL kifejezéseket tudunk vizsgálni

A korábbi szimbolikus LTL modellellenőrző algoritmusok legfontosabb jellemzői:

- Az állapotokat szimbolikusan tárolják, ezáltal képesek kezelni az állapottér-robbanás problémáját.
- Nem tudják menet közben vizsgálni a szinkron szorzat automata állapotterét, „on-the-fly”, azaz menet közbeni modellellenőrzést nem tudtak végezni.
- Számos redukció és egyszerűsítés nem alkalmazható (például a részleges rendezéses algoritmusok).
- A használt szimbolikus algoritmusok nem hatékonyak aszinkron rendszerek esetén.
- Az állapotátmenet reláció a szinkron szorzat automatában nagyon nagyra nő, ami jelentősen csökkentette az algoritmusok hatékonyságát.
- Szélességi bejárást alkalmaznak, amely nem hatékony LTL modellellenőrzés esetén.

Munkám során a két algoritmus-család előnyeit ötvöztem egy algoritmusban. Ez az első szaturáció alapú LTL modellellenőrző algoritmus, amely a következő előnyökkel bír:

- Az állapotokat szimbolikusan tárolom, így tudom kezelni az állapottér-robbanás problémáját.
- Menet közben vizsgálom a szinkron szorzat automata állapotterét, „on-the-fly”, azaz menet közbeni modellellenőrzést végez az algoritmusom, ezáltal nem feltétlenül kell bejárni a teljes állapotteret.
- Kihasználok a szaturációs algoritmus hatékonyságát aszinkron rendszerek esetén.
- Az új, vezérelt szaturáció alapú algoritmussal elkerülöm a szimbolikus állapotátmenet-reprezentáció bővítését az állapottérből származó megkötésekkel.
- Az új, vezérelt szaturáció alapú algoritmussal egy kombinált szélességi-mélységi bejárást alkalmazok az iteráció során.

A háttérismeretek között az 1.3.4. fejezetben már bemutattam egy automata konstrukciós algoritmust. Kell tehát egy-egy módszer a szinkron szorzat hatékony előállítására szaturáció segítségével (2.2. fejezet), illetve az eredmény nyelv ürességének – lehetőleg menet közbeni, „on-the-fly” – ellenőrzésére (2.3. fejezet). A következőkben ezeket a fő fázisokat mutatom be lépésről lépésre, hogy aztán összefoglalhassam az adódó LTL modellellenőrző algoritmust (2.5. fejezet).

2.2. A szaturáció kiterjesztése a szinkron szorzat automatára

Ebben a részben megmutatom, hogyan lehet felhasználni a rendkívül hatékony szaturációs algoritmust úgy, hogy a modellünk állapottere helyett annak a specifikációs automatával ($\mathcal{S}$) vett szinkron szorzatát derítse fel.

Az automaták szinkron szorzatának képzésénél (1.3.2. fejezet) láthattuk, hogy abban az esetben, ha az egyik automata összes állapota elfogadó állapot, akkor a szorzat automata $\mathcal{M} \cap \mathcal{S} = \langle \Sigma, Q_M \times Q_S, \Delta', Q_M^0 \times Q_S^0, Q_M \times F_S \rangle$, a Δ' állapotátmeneti reláció ismerttetett felépítése mellett. A továbbiakban, mivel az egyik automata ($\mathcal{M}$) a modell állapottere, tehát minden állapota elfogadó állapot, ezt az esetet vizsgáljuk.

Beszélnünk kell még arról, hogy mi lépteti az egyes automatákat, és hogyan néz ki egy szinkron lépés. Az esetünkben alkalmazott szinkron szorzat ugyanis nem egy hagyományos szinkron szorzat, mivel a két automatának nem egyezik meg az ábécéje. Erre azonban megoldást nyújt egy megfelelő leképezés a két ábécé között.

Tekintsünk a modell állapotterére úgy, mint egy olyan automatára, mely a modell lehetséges állapotait fogadja el betűkként. Ez azért is szerencsés, mert ebben az esetben a szavak, amiket az automata olvas, a modell lehetséges állapotainak sorozatai, tehát lehetséges lefutások¹ lesznek.

A leképezést ezek után úgy adjuk meg, hogy a modell egy állapotátmenetének megfelelő specifikációs automata lépés a modellbeli célállapotra *igaz* állapotkifejezések halmaza legyen. A szorzat automata egy lépése tehát úgy néz ki, hogy lépünk egyet a modellben, és megnézzük, hogy a célállapotra igaz állapotkifejezések segítségével hová léphetünk az automatában.

Erre a konstrukcióra kell tehát alkalmazni a szaturációs algoritmust. Meg kell oldani, hogy a felderített állapotok a $Q_M \times Q_S$ halmazból kerüljenek ki, illetve hogy új állapotok felderítésénél a modell új állapotaihoz rendelhető specifikációs automatabeli állapotokat is hozzárendeljük, valamint ne derítsünk fel olyan állapotokat, amelyekhez nem tartoznak ilyen legális állapotok, tehát a szorzat állapottérben tett lépések konzisztensek legyenek a Δ' állapotátmeneti relációval. Mindeközben törekednünk kell a helyesség és hatékonyság megtartására.

2.2.1. Szimbolikus állapottér-reprezentáció

A lehetséges állapotok tehát a modell állapotaiból és a specifikációs automata állapotaiból képzett párosok. Ahhoz, hogy a szaturációs algoritmus megfelelően, egy állapotként kezelje ezeket a párosokat, a specifikációs automata állapotait és a modell állapotait egyetlen MDD-ben célszerű ábrázolni. Kézenfekvő megoldás, hogy az MDD szintjeinek egy részét a modell állapotainak kódolására, egy másik részét pedig az automata állapotainak tárolására használjuk fel.

Annak a kérdésnek a megválaszolásához, hogy a specifikációs automata állapotai az MDD alsó vagy felső szintjein helyezkedjenek-e el, a szaturációs algoritmus eseménykezelése ad alapot². Láthattuk, hogy a szaturációs algoritmus az eseményeket az általuk érintett legmagasabb szinten veszi figyelembe, legkorábban ennek a szintnek a szaturálásakor tüzei el őket. Általános esetben bármely esemény eltűzése megváltoztathatja a szorzat állapothoz tartozó automataállapotot is, ez pedig azt jelenti, hogy ha az automata állapotait a felső szinten tárolnánk, minden esemény érintené a legfelsőbb szinteket. Ez azzal járna, hogy a szaturáció során elveszítenénk a dekompozíció egyik legnagyobb előnyét, a lokalitás kihasználásának képességét. Ekkor ugyanis minden eseményt a legfelső szinteken kellene eltűelni, azaz globális lépéseket kapnánk.

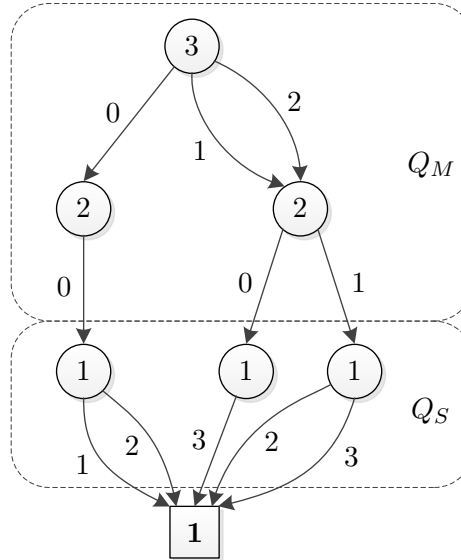
Ezzel gyakorlatilag egy szélességi keresést hajtánánk végre, amit el szeretnénk kerülni. Ennek érdekében az automata állapotait az MDD alsó szintjein fogjuk kódolni. Ezzel ugyan az események hatásának határa ismét kitolódik, azonban lefelé, ami nem befolyásolja komoly mértékben az algoritmus hatékonyságát, hiszen az algoritmusnak egyébként is mindenképp el kell jutnia a terminális szintig az állapot felvételéhez.

Magukat a specifikációs automata állapotokat is többféleképp ábrázolhatjuk. Egyrészt az egyszerűséget szem előtt tartva nyugodtan kódolhatjuk az automata összes állapotát egyetlen szinten. Ekkor minden elérhető állapotot sorszámozunk 0-tól n -ig, ahol n a specifikációs automata állapotainak száma, és ezekkel a sorszámokkal címkézzük az első szinten lévő csomópont éleit. A másik vélet az lehetne, ha az állapotokat binárisan kódolva egy BDD-vel ábrázolnánk, és ezt illesztenénk a modell állapotait leíró MDD alá.

¹A kifejezés itt kissé összemosódik az automaták lefutásával, de ez nem véletlen, a modell egy lefutása pontosan az automata egy lefutását fogja adni.

²Pusztán a döntési diagram szempontjából teljesen közömbös, hogy alul vagy felül kódoljuk a különböző állapotrészleteket.

Mivel elméletileg nem látszik számottevő különbség a két véglet között, és egyelőre mérési eredmények sem állnak rendelkezésre a kérdés megalapozott eldöntéséhez, az egyszerűbb és közvetlenebb megoldást választottam, és az automatát egyetlen szinten kódolom. A két megoldás között sok automataállapot esetén érdemes lehet köztes megoldásként több szintet is használni.



2.2. ábra. Egy szorzat állapotteret kódoló MDD.

2.2.2. Események szinkronizációja

Az algoritmusban gondoskodni kell arról, hogy az állapotátmenetek konzisztensek legyenek a szorzat automata állapotátmenet relációjával. Ehhez két dolgot kell biztosítani:

- Minden potenciálisan érvényes (q_M, q_S) párt meg kell találnunk.
- Csak azokat az átmeneteket szabad meglépni, amelyeket mindkét automatában megléphetünk egyszerre.

Egy atomi kijelentés (predikátum) definíció szerint *egy komponensben értelmezett*, ha az alanya a modellben (például egy Petri háló [11] egy helye) az adott komponensben található. Hasonlóan, egy atomi kijelentés *egy MDD valamely szintjén értelmezett*, ha a hozzá tartozó komponensben értelmezett. Ez azzal jár, hogy az ilyen kijelentések igazságtartalmát csak az adott komponens lokális állapota befolyásolja. Emiatt azt is mondjuk, hogy egy atomi kifejezés igaz vagy érvényes egy lokális állapotra, ha a lokális állapot annak a komponensnek az állapota, amiben a kifejezés értelmezett, és ebben az állapotban az alany kielégíti a rá felírt predikátumot.

Az állapotátmenet relációk helyességének biztosítása érdekében a szaturációs algoritmus során használt eseményeket kell szinkronizálnunk a specifikációs automata eseményeivel. Mivel a specifikációs automata állapotait az MDD legalsó szintje reprezentálja, ezért az eseményeket is úgy kell kibővíteni, hogy tartalmazzanak egy, a legalsó szintre ható komponenst is. Ez azzal jár, hogy minden esemény érinti az MDD legalsó szintjét, technikailag ez az összes esemény *Bot* értékét 1-re csökkentjük. Azonban végig kell gondolni a következő szempontokat:

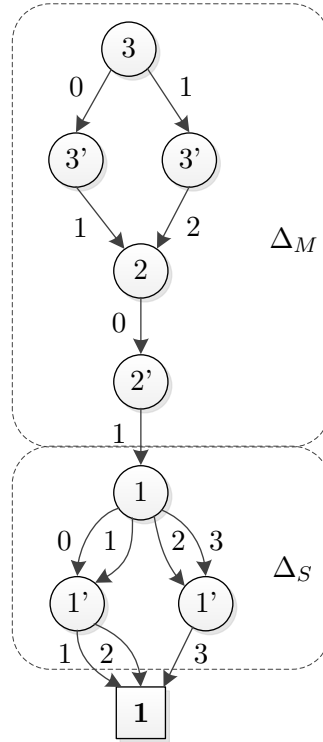
1. Míg a modell állapotátmeneteit leíró eseményeket folyamatosan építjük az állapottér felderítése során, a specifikációs automata állapottere a kezdetektől adott, akárcsak az állapotátmenetek – ezt érdemes lenne kihasználni.

2. Az automata állapotátmenetét nem csak a kiinduló állapot engedélyezi, hanem a modell célállapota is.
3. Egy eseménynél, amelynek *Top* értéke kisebb, mint valamely olyan szint sorszáma, amiben atomi kijelentés értelmezett, nem ismerhetjük a szinthez tartozó komponens lokális állapotát, és ezáltal a kijelentés igazságtartalmát sem.

A három probléma súlyosság szerint növekvő sorrendben áll. A 2. nem oldható meg a 3. miatt, mivel hiába vesszük figyelembe a célállapotokat, ha azok egy része még nem ismert. Az 1. sem oldható meg a 2. miatt, mivel az esemény jelenlegi felépítettsége (pontosabban a benne szereplő célállapotok) határozzák meg, hogy a specifikációs automatában mit léphetünk és mit nem.

A megoldáshoz kicsit vissza kell lépünk a problémák láncában, egészen addig a pontig, ahol meghatároztuk, hogy fel kell fedeznünk az új állapothoz tartozó minden *lehetséges* specifikációs automata állapotot, illetve csak azokat az átmeneteket *szabad* meglépni, amiket egyszerre mindkét automatában megléphetünk. A két kiemelt szó mentén a probléma szétbontható.

Bővítsük az eseményt úgy, hogy az élkifejezésektől függetlenül megengedjen minden lépést, ami a kiinduló specifikációs automata állapotból megtehető valamilyen élen keresztül. Nevezzük ezt az állapotátmenet relációt a specifikációs automata *egyszerűsített állapotátmenet függvényének*. Ekkor az állapotátmenet reprezentáló MDD előre elkészíthető, hiszen függetlenítettük az esemény többi részétől, és egyszerűen hozzákapcsolható a meglévő események MDD reprezentációihoz, azok legalsó két szintjeként. Ezzel megoldottuk a „lehetséges” részproblémát.



2.3. ábra. Egy szinkron eseményt kódoló MDD.

A „szabad” részprobléma megoldásához hívjuk segítségül a vezérelt szaturációt.

2.2.3. Predikátum kényszerek

Mint láthattuk, a vezérelt szaturáció lényege az, hogy valamilyen kényszert alkalmazunk az állapotfelderítés során, ami behatárolja a vizsgált és felfedezett állapotok körét. Módkban áll tehát meghatározni, hogy a lehetséges lépések közül mit *szabad* meglépni.

Azt szeretnénk megszabni, hogy egy új állapot felderítése csak akkor történhessen meg, ha a modell állapotára érvényesek azok az atomi kijelentések, *predikátumok*, melyek segítségével a kiinduló állapotból az adott specifikációs automata állapotba beléphetünk. Ez önmagában még nem megvalósítható, mivel a kényszer tipikusan csak az új állapottól függ, a régitől nem. Ugyanakkor ha figyelembe vesszük azt is, hogy (mint a 1.3.4. fejezet végén kiemelttem) a konstrukciós algoritmus által készített automata állapotaiba csak egyféle címkéjű élek mutatnak, máris megszűnik a függőség a kiinduló állapottól. Az ugyanis, hogy egy adott specifikációs automata állapotba milyen predikátumok teljesülése esetén léphetünk, mostantól csak és kizárólag az adott állapottól függ, nem számít, hogy mely állapottól készülünk lépni.

A kényszernek tehát azt kell kódolnia, hogy milyen specifikációs automata állapotba milyen predikátumok hatására léphetünk. A klasszikus vezérelt szaturációban a kényszer egy olyan MDD, amely struktúráját tekintve nagyon hasonló az állapotteret kódoló MDD-vel – szintjeik száma azonos, és az azon belüli élszámok is megegyeznek. A kényszer gyakorlatilag nem más, mint azon állapotok halmaza, amibe léphetünk. Ez a struktúra esetünkben annak felelne meg, hogy minden eddig ismert lehetséges globális állapothoz (az összes ismert lokális állapot minden kombinációjához) tárolni kellene a legelső szinten a lehetséges specifikációs automata állapotokat. Ez két dolog miatt nem célszerű:

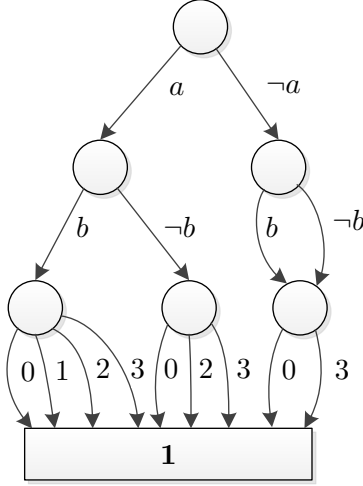
- Az így keletkező MDD feleslegesen bonyolult lehet (bonyolultabb, mint az állapotteret leíró MDD), mivel minden lehetséges kombinációt tárolunk, függetlenül attól, hogy azok valóban elérhetőek-e.
- Minél több a kényszert leíró MDD csomópontjainak száma, annál kevesebb redundanciát tud kihasználni az algoritmus a gyorsítótár³ hatékonyságának romlása miatt.
- A szaturációs algoritmus során kezdetben nem ismerjük az összes lehetséges lokális állapotot, azokat menet közben derítjük fel, így nem megvalósítható a kényszert leíró MDD statikus felépítése (holott a specifikációs automatában elvileg minden információ rendelkezésre áll az állapotátmeneti relációból). A dinamikus, felderítés közbeni építés viszont komoly erőforrásokat igényelne.

A fenti okok miatt a kényszer esetében is bevezetünk egy leképezést. Ez a leképezés – hasonlóan az automaták bemenetein alkalmazotthoz – a modell egy globális állapotát a rajta érvényes atomi kijelentések, predikátumok halmazára fordítja le. A specifikációs automata állapotátmeneteit ezek függvényében már statikusan is meghatározhatjuk – ezt írja le az állapotátmeneti reláció is.

A kényszer ekkor a $2^{AP} \mapsto Q_S$ függvény leírása MDD segítségével, ahol AP az automata forrásában szereplő atomi kijelentések halmaza. A függvényt páronként kódoljuk az MDD-ben, és ezt a struktúrát (az állapotokat a rájuk érvényes atomi kifejezések halmazává fordító leképezéssel együtt) *predikátum kényszernek* nevezzük.

A szaturációs algoritmus az állapottér MDD-n történő rekurzív lépései során a predikátum kényszert ugyanúgy a lokális célállapottal léptetjük, mint a hagyományos vezérelt szaturációban. A leképező logika ezt a lokális célállapotot alakítja át a rá igaz atomi kijelentések halmazára, majd az MDD-ben aszerint lép, hogy az adott szinten értelmezett

³A gyorsítótárban *(kiinduló csomópont, esemény csomópont, kényszer csomópont)* hármasok adják az eredmény csomópontok kulcsát.



2.4. ábra. Egy predikátum kényszert kódoló MDD.

predikátumok közül melyik igaz és melyik nem. Ha egy szinten több predikátum is értelmezett, mindegyikhez társít egy lépést, ha pedig egy sem, akkor lépés helyett ismét visszaadja az aktuális csomópontot. A legalsó szinten azok a specifikációs automata állapotok szerepelnek (a *terminális egybe* kötve), amiknek a belépési feltételét kielégíti az addig lekötött predikátumok halmaza.

Ezzel megoldottuk a „szabad” részproblémát is. Elértük, hogy az eseményeknél tett lazítás mellett a szaturációs algoritmus ténylegesen csak elérhető állapotokat fedezzen fel.

2.2.4. Formális megközelítés

Az alábbiakban formálisan is bemutatom a gondolatmenet lényegét. Legyenek adottak a szorzat automata állapotátmenet relációjának elemei a következő formában:

$$(\langle q_M, q_S \rangle, \alpha, \langle q'_M, q'_S \rangle), \quad (2.1)$$

ahol

- $\langle q_M, q_S \rangle, \langle q'_M, q'_S \rangle \in Q$,
- $q_M, q'_M \in Q_M$,
- $q_S, q'_S \in Q_S$,
- $(q_M, q'_M) \in \mathcal{N}$, valamint
- $(q_S, \alpha, q'_S) \in \Delta_S$.

α nem más, mint a q'_M -ben igaz atomi kijelentések halmaza, vagyis

$$\alpha = \mathcal{P}(q'_M) \quad (2.2)$$

Azt is tudjuk, hogy a transzformációs algoritmus miatt csak egyféle bemenettel juthatunk el egy adott specifikációs automata állapotba. Emiatt egy $\langle q'_M, q'_S \rangle$ páros egyértelműen meghatározza α -t, vagyis

$$\alpha = \mathcal{F}(q'_M, q'_S).$$

E mentén a 2.1. reláció veszteségmentesen és függőségörző módon kétfelé bontható:

$$(\langle q_M, q_S \rangle, \langle q'_M, q'_S \rangle), \quad (2.3)$$

$$(\alpha, \langle q'_M, q'_S \rangle). \quad (2.4)$$

Az utóbbi, 2.4. reláció a 2.2. összefüggést felhasználva $(\mathcal{P}(q'_M), \langle q'_M, q'_S \rangle)$ alakú lesz, ami viszont a függvényyszerű függőség miatt a következő relációval azonos jelentést hordoz:

$$(q'_M, q'_S). \quad (2.5)$$

2.5 esetén az összes lehetséges modellbeli állapothoz tárolni kellene a lehetséges specifikációs automata állapotokat, ami a korábban ismertetett okok miatt rendkívül költséges lenne. Ehelyett felhasználhatjuk a $\mathcal{P}$ függvényt, aminek segítségével már az állapotter-generálás előtt is felépíthetjük a következő relációt:

$$(\mathcal{P}(q'_M), q'_S). \quad (2.6)$$

$\mathcal{P}$, valamint a 2.3. és 2.6 relációk segítségével a 2.1. reláció már bármikor visszaállítható.

2.2.5. Az új vezérelt szaturációs algoritmus

Az így létrehozott struktúrák segítségével a szaturációs algoritmus már felhasználható a *szinkron szorzat állapotter felderítésére*. A szükséges módosítások kimerülnek abban, hogy másképp kell léptetnünk a kényszert, és az automata figyelembe vételével kell inicializálni az adatszerkezeteket.

Az adatszerkezetek inicializálásában még egy feladat felmerül. A modell kezdőállapota(i) nem feleltethető meg a specifikációs automata kezdőállapotának, mivel az az állapot egy mesterséges kezdőállapot. Az állapotfelderítés indításakor a kezdőállapotnak megfelelően először lépnünk kell egyet a specifikációs automatában. Ez szemléletesen megfelel annak, hogy kezdetben mind a modell, mind az automata egy nem definiált, inicializálatlan állapotban van, az első lépéssel kerülnek a kezdeti állapotaikba. Gyakorlatilag célszerűbb megvizsgálni a modell kezdőállapotaira fennálló predikátumokat, és ezek alapján kiválasztani a lehetséges kezdő specifikációs automata állapotokat a mesterséges kiindulási állapotból elérhető állapotok közül. Én is ezt teszem, mivel így megspórolható egy mesterséges inicializáló állapotátmenet.

Az alábbiakban bemutatom az adódó szaturációs algoritmus pszeudokódját, azon belül is a három fő (Saturate, SatFire, SatRecFire) és a predikátum kényszer MDD-jét léptető függvényt (StepConstraint). Az algoritmusok három további segédfüggvényt [5] használnak:

- Az egyik a **NewNode**, ami egy l egész értéket vár paraméterként, visszatérési értéke pedig egy, az MDD l . szintjére beszúrt új csomópont. A teljes szinkron szorzat állapotteret a **GenerateProductStateSpace** függvény deríti fel.
- A másik a **CheckIn**, ami egy csomópontot és egy szintszámot vár, és vagy egy már létező, vele ekvivalens csomópontot ad vissza, vagy az MDD-be történő beszúrás után a paraméterként kapott csomópontot.
- A harmadik a **Confirm**, ami egy lokális állapotot és egy l szintszámot vár, és a lokális állapotot elérhetőnek jelöli meg, S_l -be helyezve azt.

Az algoritmus jelen megfogalmazása feltételezi, hogy az $\mathcal{N}$ next-state függvény adott⁴. $\mathcal{N}_e$ jelöli az e esemény next-state függvényét, S_l pedig az l szinten elérhető (vagyis már felderített) állapotok halmazát.

A **StepConstraint** függvény (1. algoritmus) egy kényszer MDD-beli c csomópontot, egy i lokális állapotot és egy l szintszámot vár. A szintszám a lokális állapot szintszáma kell,

⁴A gyakorlatban $\mathcal{N}$ -t célszerű az állapotter felderítése közben lépésről lépésre építeni.

hogy legyen, és ha egy, akkor c szintszámának is egynek kell lennie (az MDD felépítéséből adódóan). A visszatérési érték egy kényszer MDD-beli csomópont lesz, ami egy hagyományos kényszer c csomópontjának i . gyerekét emulálja.

Ha a szintszám egy, akkor a kényszert már az automatával léptetjük, így az algoritmus az eddig bejárt út mentén igaz predikátumok alapján terminális nullát vagy egyet ad vissza aszerint, hogy az állapot megengedett-e. Természetesen ez is az MDD-ben van elködölve, így a visszatérési érték $c[i]$. Minden egyéb esetben veszi a szinten értelmezett atomi kijelentések AP_l halmazát, és egy előre meghatározott sorrendben ellenőrzi, hogy az i lokális állapot igazzá teszi-e a proposíciót. Ha igen, akkor $c[1]$ -be lép tovább, ha nem, akkor $c[0]$ -ba, és az új csomóponttal folytatja a predikátumok kiértékelését. Ha az összes atomi kijelentés kiértékelése megtörtént, azt a csomópontot adja vissza, amibe a kiértékelések során eljutott.

Algoritmus 1 A kényszer MDD-t léptető StepConstraint függvény

```

1: function STEPCONSTRAINT( $c, i, l$ )
2:   if  $l = 1$  then return  $c[i]$ ;
3:   for all  $p \in AP_l$  do
4:     if  $p$  igaz  $i$ -re then
5:        $c \leftarrow c[1]$ ;
6:     else
7:        $c \leftarrow c[0]$ ;
8:     end if
9:   end for
10:  return  $c$ ;
11: end function

```

A Saturate függvény (2. algoritmus) egy részállapotteret kódoló s MDD csomópontot és egy hozzá tartozó kényszer MDD részletet reprezentáló c csomópontot vár. Először megvizsgálja, hogy a kért hívás kiszámításra került-e már ugyanezekkel a paraméterekkel - ha igen, az eredmény csomópontot visszaadja a cache-ből. Ha nincs cache találat, akkor létrehoz egy új t csomópontot a paraméterként kapott s szintjén. Szaturálja s összes gyereket, ha a kényszer engedi, az eredményeket pedig beköti t megfelelő éleire. Ezután minden, s szintjéhez tartozó eseményt eltűzel t -n mindaddig, amíg ez változást okoz. A tüzelést a SatFire függvény végzi. Végül meghívja a CheckIn függvényt, ezzel t bekerül az állapotter MDD-be. A hívás eredményét lementi a cache-be.

A SatFire eljárás (3. algoritmus) szintén egy részállapotteret kódoló s MDD csomópontot és egy hozzá tartozó kényszer MDD részletet reprezentáló c csomópontot vár, továbbá meg kell adni neki egy esemény next-state függvényét leíró MDD r gyökér csomópontját is. Visszatérési értéke nincs, a megadott eseményt eltűzeli a részállapottéren. Az eseményben kódolt, a jelenlegi szinten kódolt (i, i') állapotátmenetek mindegyikére megvizsgálja, hogy a kényszerben is léphet-e i' -be. Ha igen, akkor meghívja a tüzelést alacsonyabb szinteken elvégző SatRecFire-t, az eredményt pedig visszauniózza az i' ágon eddig kódolt állapotokhoz. Ha a felfedezett állapot új, akkor meghívja a CONFIRM függvényt, ami regisztrálja az új elérhető állapotokat⁵. Mindezt addig ismétli, amíg s változik a tüzelés hatására.

A SatRecFire függvény (4. algoritmus) ugyanazokat a paramétereket fogadja, mint a SatFire, azonban r már nem egy gyökér-csomópont, hanem annak valamilyen szintű utódja. A visszaadott csomópont a tüzelés és a keletkező csomópontok szaturálása során létrejött új állapotokat reprezentálja.

⁵A next-state függvény is ilyenkor építhető tovább, ha nem feltételezzük adotttnak.

Algoritmus 2 A Saturate függvény

```
1: function PRODUCTSATURATE( $s, c$ )
2:   if  $\exists t : (s, c, t) \in \text{SaturateCache}$  then
3:     return  $t$ ;
4:   end if
5:    $l \leftarrow \text{Level}(s)$ ;
6:    $t \leftarrow \text{NEWNODE}(l)$ ;
7:   for all  $i \in S_l : s[i] \neq 0$  do
8:      $c_i \leftarrow \text{STEPCONSTRAINT}(c, i, l)$ ;
9:     if  $c_i \neq 0$  then
10:       $t[i] \leftarrow \text{PRODUCTSATURATE}(s[i], c_i)$ ;
11:    else
12:       $t[i] \leftarrow s[i]$ ;
13:    end if
14:  end for
15:  repeat
16:    for all  $\mathcal{N}_e : \text{Top}_e = l$  do
17:       $r \leftarrow \mathcal{N}_e$ ;
18:       $t \leftarrow \text{PRODUCTSATFIRE}(t, s, r)$ ;
19:    end for
20:  until  $t$  nem változik tovább;
21:   $t \leftarrow \text{CHECKIN}(l, t)$ ;
22:   $\text{SaturateCache} \leftarrow \text{SaturateCache} \cup \{(s, c, t)\}$ ;
23:  return  $t$ ;
24: end function
```

Ha a nulladik szinten vagyunk, és az állapot és az esemény is a terminális egyes csomópont⁶, akkor az esemény eltűzelhető, a függvény a terminális egyes csomóponttal tér vissza. Ellenkező esetben először ellenőrzi, hogy a kért hívás kiszámításra került-e már ugyanezekkel a paraméterekkel - ha igen, az eredmény csomópontot visszaadja a cache-ből. Ha nem, akkor a jelenlegi szinten kódolt (i, i') állapotátmenetek mindegyikére megvizsgálja, hogy a kényszerben is léphet-e i' -be. Engedélyezett lépés esetén meghívja önmagát a kiinduló állapothoz tartó eggyel alacsonyabb szintű csomóponttra. Ha az u visszatérési érték nem terminális nulla, tehát az esemény alacsonyabb szinteken sikeresen tüzelt, akkor – szükség esetén létrehozva egy új csomópontot az eredmény tárolására – képezi az eddig, illetve újonnan felfedezett állapotok unióját leíró csomópontot. Akárcsak a SatFire esetében, új állapot felfedezése esetén most is meghívódik a CONFIRM függvény. A keletkező t csomóponttra meghívja a Saturate függvényt, majd az eredményt lementi a cache-be, és visszaadja t -t.

A bemutatott algoritmus futtatását egy Saturate hívással kell kezdeni, aminek s paramétere a kezdeti állapot(halmaz) gyökér-csomópontja lesz, c paramétere pedig a kényszer MDD gyökere. Amikor ez a hívás visszatér, az állapottér tartalmazni fogja a szinkron szorzat automata állapotterét.

⁶Azt, hogy a kényszer nem terminális nulla, az algoritmus előző szinten már megvizsgálta a függvényhívás előtt.

Algoritmus 3 A SatFire függvény

```
1: procedure PRODUCTSATFIRE( $s, c, r$ )
2:    $l \leftarrow \text{Level}(s)$ ;
3:   repeat
4:     for all  $i, i' \in S_l : r[i][i'] \neq 0$  do
5:        $c_{i'} \leftarrow \text{STEPCONSTRAINT}(c, i', l)$ ;
6:       if  $c_{i'} \neq 0$  then
7:          $u \leftarrow \text{PRODUCTSATRECFIRE}(s[i], c_{i'}, r[i][i'])$ ;
8:          $s[i'] \leftarrow s[i'] \cup u$ ;
9:         if  $i' \notin S_l$  then
10:            $\text{CONFIRM}(i', l)$ ;
11:         end if
12:       end if
13:     end for
14:   until  $s$  nem változik tovább;
15: end procedure
```

2.3. Erősen összefüggő komponensek keresése

A szorzat állapottér bejárását követően (még szerencsésebb esetben a bejárás közben) meg kell vizsgálnunk, hogy az adódó automata nyelve üres-e. Az erre a kérdésre adott válasz direkt módon megválaszolja azt a kérdést is, hogy a modell teljesíti-e a megkövetelt tulajdonságokat.

Mivel egy Büchi automata nyelvről beszélünk, amely végtelen hosszú szavakat fogad el, olyan utakat kell találnunk az automata (véges) állapotterében, amik végtelen hosszúak, és végtelenül gyakran haladnak át elfogadó állapotokon. Általánosított Büchi automata esetében mindegyik elfogadó állapot halmazból át kellene haladni legalább egy állapoton, így a vizsgálat leegyszerűsítése végett a transzformációs algoritmus kimeneteként adódó általánosított Büchi automatát először egyszerű Büchi automatává redukáljuk.

Egy véges állapottérben végtelen utak kizárólag akkor fordulhatnak elő, ha az őket reprezentáló irányított gráfban létezik legalább egy *irányított kör*. Ezt gráfelméleti nyelven úgy is megfogalmazhatjuk, hogy a gráfnak van nemtriviális *erősen összefüggő komponense*⁷, vagy hurokkel rendelkező csomópontja. Ekkor egy lefutás két szakaszra bontható: egy véges hosszú útra, amely egy nemtriviális erősen összefüggő komponensbe vezet, és erre a komponensre, ahol a lefutás körbe kerül, és végtelen sokszor áthalad a kör állapotain. Ahhoz, hogy ez a lefutás elfogadó legyen, tehát végtelenül gyakran haladjon át elfogadó állapotokon, a második szakasznak, vagyis az erősen összefüggő komponensnek legalább egy elfogadó állapotot kell tartalmaznia.

Olyan köröket keresünk⁸ tehát, amik tartalmaznak legalább egy elfogadó állapotot. Az alábbiakban bemutatom, hogyan tehető meg mindez a szaturációs algoritmus keretein belül, a bejárasi stratégiát kihasználva, és a körkeresés folyamatát *inkrementálisan*, „on-the-fly” módon beépítve az állapottér felderítésébe.

A most következő algoritmus esetében nagyon nehéz lenne bemutatni a kialakulásához vezető utat. Szaturációs alapon, inkrementálisan és „on-the-fly” módon még senki nem tudott eredményesen kört keresni egy állapottér felderítése közben, így az alábbiakban

⁷Egy nemtriviális erősen összefüggő gráfban vagy algráfban bármely két csomópont elérhető egymásból, és a gráf legalább két csomópontot tartalmaz.

⁸Körkeresés alatt ebben a fejezetben egy olyan műveletet értek, amely képes megállapítani, hogy egy gráfban van-e kör, és ha igen, akkor visszaad *néhány* olyan állapotot, amely valamilyen körben található. Utóbbi információ segítségével a kör később könnyen rekonstruálható.

Algoritmus 4 A SatRecFire függvény

```
function PRODUCTSATRECFIRE( $s, c, r$ )
  if  $s = 1 \wedge r = 1$  then return 1;
  if  $\exists t : (s, c, r, t) \in \text{SatRecFireCache}$  then
    return  $t$ ;
  end if
   $l \leftarrow \text{Level}(s)$ ;
   $t \leftarrow 0$ ;
  for all  $i, i' \in S_l : r[i][i'] \neq 0$  do
     $c_{i'} \leftarrow \text{STEPCONSTRAINT}(c, i', l)$ ;
    if  $c_{i'} \neq 0$  then
       $u \leftarrow \text{PRODUCTSATRECFIRE}(s[i], c_{i'}, r[i][i'])$ ;
      if  $u \neq 0$  then
        if  $t = 0$  then  $t \leftarrow \text{NEWNODE}(l)$ ;
         $t[i'] \leftarrow t[i'] \cup u$ ;
        if  $i' \notin S_l$  then
           $\text{CONFIRM}(i', l)$ ;
        end if
      end if
    end if
  end for
   $t \leftarrow \text{PRODUCTSATURATE}(\text{CHECKIN}(l, t), c)$ ;
   $\text{SaturateCache} \leftarrow \text{SatRecFireCache} \cup \{(s, c, r, t)\}$ ;
  return  $t$ ;
end function
```

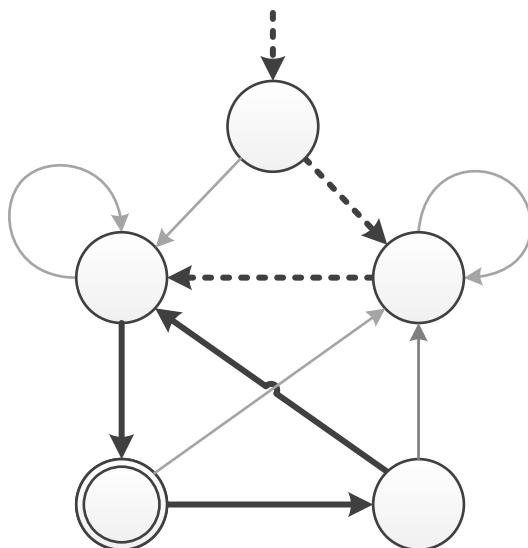
bemutatott megoldások a követelmények direkt kielégítésén túl nem következnek triviálisan semmilyen létező általános megoldásból. Építenek a szaturáció bonyolult gráfbejárási stratégiájára, és halmazelméleti megfontolásokra, amelyek bemutatásával már érthetővé válik a végleges algoritmus.

Először bemutatom a szaturációs algoritmus lépéseinek halmazelméleti értelmezését, illetve a segítségükkel adódó műveleteket, amelyeket felhasználhatok a továbbiakban (2.3.1. fejezet). Ezután a modellellenőrzőben használt megoldásnak sikeresen alapot adó halmazelméleti alapötletet mutatom be (2.3.2. fejezet). Végül vázolom, hogyan használható ez a megközelítés inkrementális körkeresésre, illetve hogyan lehet elfogadó állapotokat tartalmazó köröket keresni (2.3.3. fejezet).

2.3.1. A szaturációs algoritmus lépései halmazelméleti szempontból

A szaturációs algoritmus három fő függvénye a Saturate, SatFire és a SatRecFire. A Saturate egy csomópont által kódolt állapothalmaz tranzitív lezártját képezi az $\mathcal{N}$ next-state függvény által kódolt Δ állapotátmenet reláció azon eseményeire nézve, amiknek Top értéke kisebb, vagy egyenlő a csomópont szintjénél. A SatFire ezen belül az adott csomóponton egy adott eseményt tüzel addig, amíg ez új állapotok megjelenését okozza, egyben szaturálva a megjelenő új csomópontokat. Az új állapotok mindig visszakerülnek a kiindulási állapothalmazba, így a következő tüzelés hatására már rajtuk keresztül is új állapotok érhetők el. A tüzelés végigvitelére és az alacsonyabb szinteken megjelenő új állapotok szaturálására szolgál a SatRecFire.

A továbbiakban két műveletet szeretnék használni: a tranzitív lezártat, illetve egy adott állapotátmenet-halmaz állapotátmeneteinek egyszeri meglépését. A tranzitív lezárt



2.5. ábra. Egy elfogadó lefutás két szakasza.

képzésére használható a *Saturate* függvény, azonban az egyszeri lépést egyik függvény sem valósítja meg. A műveletet *SatFire* és a *SatRecFire* függvények kis módosításával végzem el – a *SatFire*-ben megszüntetem a ciklust, amely az eseményeket újra és újra eltűzeli az addig felderített állapothalmazon, a *SatRecFire* függvényben pedig elhagyom a keletkező csomópontok szaturálását. A módosított függvényeket egy *SingleStep* nevű függvényben egyesítem, amelynek pszeudokódját a kevés eltérés miatt külön nem mutatom be. A *SingleStep* függvény szintén saját gyorsítótárral dolgozik a redundáns lépések megtakarítása érdekében.

A fenti két művelet az állapotátmeneteken visszafelé lépve is megvalósítható. Az állapotátmeneteket leíró struktúrákból az átmenet visszafelé is kiolvasható, ezzel nincs probléma. A kényszer viszont másképp kell használni, mivel a 2.2.3. fejezetben úgy definiáltam, hogy az adott állapotba lépés feltételeit adja meg, azt kihasználva, hogy egy specifikációs automata *állapotba* csak egyféle feltétellel léphetünk. Visszafelé viszont ez nem igaz. Egy egyszerű megfontolással azonban belátható, hogy visszafelé szaturálva egyáltalán nincs szükség a kényszerre.

Mivel egy adott specifikációs automata állapotba csak egyféle feltétellel rendelkező élen lehet eljutni, minden olyan elérhető szorzat állapotban, amiben ez a specifikációs állapot szerepel, a modellállapot rész teljesíti ezt a feltételt. Bármilyen állapotból is érkezünk tehát ebbe az állapotba, a predikátum kényszer engedélyezni fogja az átmenetet. Így aztán visszafelé megkötés nélkül léphetünk. Más szavakkal: a predikátum kényszer hivatott megakadályozni az érvénytelen állapotokba történő lépéseket, de érvényes állapotba bárhonnan juthattunk.

Mindezek mellett viszont visszafelé szaturálás esetén mindig számolni kell azzal a lehetőséggel, hogy a kezdeti állapothalmazból valójában elérhetetlen állapotokat is felfedezhetünk. Ennek kiküszöbölésére hozták létre a klasszikus vezérelt szaturációt, ami a felfedezett állapotokat adja meg kényszerként, és elveti a lépéseket, ha azok eredménye nem található meg ebben a kényszerben. Visszafelé lépéskor ezt fogom használni.

2.3.2. Körök felderítése követelmények mellett

A *Saturate* függvényt felhasználva a tranzitív lezárt képzéséhez adtam egy általánosan használható megoldást olyan körök keresésére, amelyek valamilyen követelményt kielégítenek. Ehhez kétféle követelményt definiáltam:

1. A kör tartalmazzon legalább egy *állapotot* egy S halmazból.
2. A kör tartalmazzon legalább egy *állapotátmenetet* e halmazból.

Az algoritmus helyes működéséhez vagy

- kell legalább egy 2-es típusú követelmény,
- vagy az 1-es típusú követelmények közül legalább két halmaznak diszjunktak kell lennie.

Ez természetesen azt vonja maga után, hogy bizonyos esetekre ez az algoritmus nem használható, azonban az általam alkalmazott keretek között ez nem okoz problémát. A korlátozások okait az algoritmus leírása során mutatom be.

Egy állapot akkor és csak akkor része egy körnek, ha elmondható róla, hogy valamilyen átmeneteken keresztül önmagából elérhető. A kör létezésének *szükséges és elégséges* feltétele, hogy az állapothalmaznak legyen ilyen tulajdonságú állapota.

Elméletileg tehát, ha megvizsgálunk minden s állapotot abból a szempontból, hogy a belőle elérhető állapotok tartalmazzák-e s -t, meg tudjuk mondani, hogy van-e kör az állapothalmazban. Ez azonban egyrészt nem hatékony, mert rendkívül sok műveletet igényel az állapotok egyenként történő vizsgálata, másrészt a *Saturate* függvény által kiszámolt tranzitív lezárt (amit használni szeretnénk) nem különböztetné meg a kiinduló és az elért állapotokat. Emiatt nem megállapítható, hogy a kezdőállapotból kiindulva ténylegesen visszajuthatunk-e oda.

Ha már két állapotunk lenne, amin a körnek át kell haladnia, akkor meg lehetne vizsgálni, hogy az egyik elérhető-e a másiktól, és fordítva⁹, hiszen nem kellene attól tartanunk, hogy a másik állapot valójában nem szerepel az elérhető állapotok között, csak a kezdőállapotok halmazában.

Ezek a vizsgálatok kiterjeszthetők állapothalmazokra is. Ez célszerű, mert a szaturációs algoritmus egész állapothalmazokkal is képes egyszerre dolgozni, így a hatékonyság általában nő, ha több állapotot kezelünk egyszerre. Ha egy állapothalmaz és a belőle legalább egy átmeneten keresztül elérhető állapotok halmazának metszete nem üres, akkor vannak benne olyan állapotok, amik *lehetséges*, hogy elérhetők önmagukból. Ez azonban nem biztos, mert lehet, hogy egy olyan állapotból voltak elérhetőek, ami a metszetben már nem szerepel. Ezért a lépést meg kell ismételniünk, egészen addig, amíg el nem fogyunk a metszetben szereplő állapotok, vagy nem változik tovább a metszet. Előbbi esetben nem volt olyan állapot, ami elérhető lenne önmagából, utóbbi esetben viszont a maradék állapotokból vezet út saját magukba, tehát kört találtunk.

Az egy állapottal végzett vizsgálatokhoz analóg módon a *Saturate* függvény használata esetén sajnos itt sem elég egyetlen állapothalmaz, mivel az eredményben szerepelni fog az összes kiinduló állapot, így a metszet soha nem csökkenti az „esélyes” állapotok számát. Ha viszont két diszjunkt halmazt alkalmazunk, és azt mondjuk, hogy olyan köröket keresünk, amik mindkét halmaz legalább egy elemét tartalmazzák, akkor a problémát ugyanúgy sikerült megoldani, mint az előző esetben.

Jelenleg az algoritmus még nem kezeli a hurokéleken át létrejövő köröket. Erre megoldást nyújthatnak a 2-es típusú követelmények, melyeket visszavezetnek az 1-es típusúakra, viszont kihasználom az adódó többletinformációt is.

Egy állapotátmenet vagy él két – nem feltétlenül különböző – állapottá alakítható, a kezdeti és a végállapottá. Egy kör, amiben az adott él szerepel, át kell, hogy haladjon mindkét állapoton, a két állapot között pedig az élen át kell haladnia. Az előző esetben

⁹Ez az erősen összefüggő komponens klasszikus definíciója is, viszont bármely két csomópontra nézve. Jelen esetben megköveteljük, hogy a két csomópont különböző legyen - végső soron ebből adódik a 2-es megkötés.

ez annyi megkötéssel jár, hogy a kiinduló állapotból a végállapotba a meghatározott élen kell tudni eljutni, a végállapotból a kiindulóba pedig akárhogy. Vegyük észre, hogy itt nem kell, hogy a két állapot különböző legyen, hiszen egy él két végpontjáról beszélünk, ha ez egy hurokél, akkor azonnal találtunk egy kört.

Ezt az esetet is érdemes élhalmazokra (állapotátmeneti relációkra) is megfogalmazni. Élhalmaz kétféleképp kerülhet szóba: egyetlen él helyett a teljes eseményt vesszük figyelembe, minden állapotátmenetével, illetve ilyen eseményekből egyszerre többet is vizsgálhatunk. Most is több iterációban kell szűkíteni a halmazokat. A végállapotokból a kezdőállapotokba a *Saturate* hívással kell tudnunk eljutni, a kezdőállapotokból a végállapotokba pedig valamelyik élen át az adott élhalmazból.

A két halmazt úgy a legcélszerűbb megkonstruálni, hogy elsőként az egész állapottérben lépünk *egy* az összes állapotátmenet mentén, ekkor előáll a végállapotokat tartalmazó halmaz, majd ebből a halmazból visszafelé léphetünk egyet ugyanígy, de az inverz átmeneteket használva¹⁰. Az egyszeri lépésre a *SingleStep* függvényt használom.

A kétféle követelmény tetszős szerint kombinálható akárhány halmazra, a fentebb leírt megkötéseket szem előtt tartva. Meg kell jegyezni azonban, hogy a megadott követelmények *sorrendje számít*¹¹. Az éleknek megfelelő két állapothalmaz között mindig az adott élen át kell tudni egy lépésben eljutni a végállapotok halmazába. Amikor a következő halmazt szeretnénk szűkíteni, akkor a végállapotokból indítunk *Saturate* hívást, amikor az élek megfelelő halmazokat kell szűkíteni, akkor pedig a forrásállapotok halmazát szűkítjük.

Az algoritmus során a különböző követelményeket sorrendbe tesszük, és körbe-körbe szűkítjük őket az előzőből elérhető állapotokra. Abban az esetben, ha a halmazok már nem változnak tovább, az algoritmus elérte a fixpontját. Ha ekkor a halmazok üresek, akkor nincs olyan kör, ami megfelelne a követelményeknek, ha pedig a halmazok tartalmaznak elemeket, akkor kört találtunk, tehát megállhatunk. Belátható, hogy ha egy halmaz elfogy, akkor az az összes többi halmaz elfogyását is okozza, míg ha az egyik változatlan marad egy szűkítő lépés során, és *már mindegyik halmaz szűkítésre került egyszer*, akkor a többi halmaz sem változik többet.

A halmazokban maradt állapotok egy része valamilyen kör részét képezi, illetve minden halmazban lehetnek olyan állapotok, amelyek az előző halmazból elérhetőek, de nem tartoznak semmilyen körbe. Ezeket szükség esetén könnyen eltávolíthatjuk az inverz állapotátmeneteket alkalmazva, és fordított sorrendben szűkítve a halmazokat, ez azonban nyugodtan megtehető a kör létének bebizonyosodása után.

A hatékonyságot illetően megmutatható, hogy minél kevesebb élet/állapotot tartalmaznak a követelmények, annál kevesebb iterációra lesz szükség (2. lemma).

2.3.3. Inkrementális körkereső algoritmus

Az új kereső algoritmus a fenti általános algoritmust felhasználva a következő nagyon erős tulajdonságokkal rendelkezik:

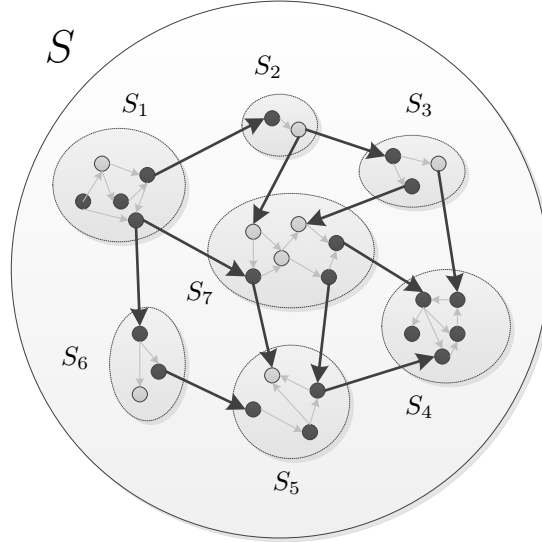
- „*on-the-fly*”, tehát az állapottér-generálás közben kiértékelhetők a részeredmények,
- *inkrementális*, tehát a keresés folytatódólagos, ami csökkenti az iteráció redundanciáját.

Ehhez a szaturációs algoritmus iterációs stratégiáját kell kihasználni. Amikor egy csomópont szaturálttá válik, az azt jelenti, hogy az általa kódolt részállapottér minden, csak erre a részmodellre vonatkozó eseményre zárt, tehát mindent felfedeztünk, amit lehetett.

¹⁰Természetesen ez fordított sorrendben is megtehető.

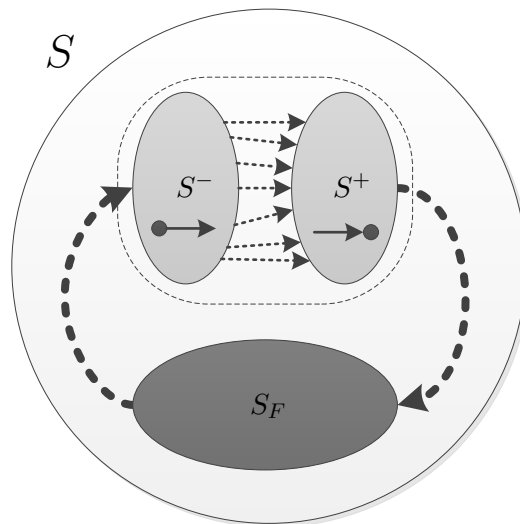
¹¹Mivel ez egy ciklikus permutáció, két követelmény esetén – mint a következő fejezetben felhasznált verzióban – egyetlen sorrend létezik, így ott ezzel nem kell majd foglalkoznom.

Egy csomópont ezen tulajdonsága nem romlik el¹², ráadásul az is elmondható, hogy ha egy csomópont szaturált, akkor az alatta lévő csomópontok is azok. Az adott szinten lévő csomópont az alatta lévő, szaturált csomópontok által kódolt állapothalmazok unióját kódolja. Az általa reprezentált (állapotokból és állapotátmenetekből álló) részgráf tehát a *gyerekei által kódolt részgráfok és az új események által reprezentált állapotátmenetek uniója*.



2.6. ábra. Egy szaturált csomópont állapottere a legfelső szintű eseményekkel és az elfogadó állapotokkal.

Utóbbi gondolat a kulcs: mindig, amikor egy csomópont szaturálttá válik, elegendő olyan köröket keresni, amikben legalább egyszer megjelenik valamelyik új állapotátmenet. Ha ugyanis egy kör nem halad át ilyen átmeneten, akkor már a korábban szaturált csomópontok esetében is felfedeztük volna. Ezzel egyfajta inkrementalitást vezetünk be, amitől javul az algoritmus skálázódása.



2.7. ábra. Elfogadó erősen összefüggő komponensek inkrementális keresése.

Ahhoz, hogy elfogadó köröket keressünk, azt is biztosítanunk kell, hogy a kör áthaladjon legalább egy elfogadó állapoton. Ez biztosítható egy 1-es típusú követelmény meg-

¹²Természetesen előfordulhat, hogy az egész csomópont lecserélődik, de az egy újabb szaturálandó csomópontot eredményez, amire szaturálása után ugyanez lesz érvényes.

adásával, ami az elfogadó állapotok halmazát fogja felhasználni a körök szűrésére. Ezzel nem csak azt érjük el, hogy csak a nekünk megfelelő köröket találja meg az algoritmus, de sokkal gyorsabban is teheti mindezt, mivel sok kört eleve nem kell figyelembe vennie¹³. Az elfogadó állapotok halmaza halmazműveletek segítségével hatékonyan számolható.

Algoritmus 5 Elfogadó erősen összefüggő komponensek inkrementális keresése

```

1: function DETECTCIRCLES( $S, F, \mathcal{N}_{\mathcal{E}_l}$ )
2:    $S^+ \leftarrow \mathcal{N}(S)_{\mathcal{E}_l}$ ;
3:    $S^- \leftarrow \mathcal{N}^-(S)_{\mathcal{E}_l}$ ;
4:    $S_F \leftarrow S \cap F$ ;
5:   if  $S_F = \emptyset$  then return false;
6:   repeat
7:      $S^- \leftarrow S^- \cap \mathcal{N}^*(S_F)$ ;
8:      $S^+ \leftarrow S^+ \cap \mathcal{N}_{\mathcal{E}_l}(S^-)$ ;
9:      $S_F \leftarrow S_F \cap \mathcal{N}^*(S^+)$ ;
10:  until  $S^+$  és  $S^-$  és  $S_F$  nem változik tovább;
11:  if  $S_F = \emptyset$  then
12:    return false;
13:  else
14:    return true;
15:  end if
16: end function

```

2.3.4. Az algoritmus helyességének és teljességének bizonyítása

Az algoritmus helyességét és teljességét a fentebb bemutatott, az algoritmus során ténylegesen használt esetre bizonyítom be.

A bizonyításához alapvetően négy állítást kell belátnunk:

1. Ha van a követelményeket kielégítő kör, akkor egyik halmaz sem lesz üres a fixponton.
2. Ha nincs megfelelő kör, akkor minden halmaz üres lesz a fixponton.
3. Ha egyik halmaz sem üres a fixponton, akkor van kör is.
4. Ha minden halmaz üres a fixponton, akkor nincs kör.

Jelölje a továbbiakban $R^- \subseteq S^-$, $R^+ \subseteq S^+$ és $R_F \subseteq S_F$ az algoritmus fixpontjaként kapott három halmazt.

1. lemma. Ha létezik irányított kör, amelynek elemei egy C halmazt alkotnak, és $\exists s^-, s^+, s_F \in C$ és $s^- \in S^-$, $s^+ \in S^+$, $s_F \in F$, valamint $(s^-, s^+) \in e$, akkor $R^-, R^+, R_F \neq \emptyset$.

Bizonyítás Az algoritmus során s_F nem tűnhet el, mert – amiatt, hogy egy irányított körben minden csomópont elérhető minden másiktól – s^+ miatt elérhető S^+ -ból. Ugyanez igaz s^+ -ra is, mivel s^- miatt elérhető S^- -ből e -n keresztül. Mindez még egyszer elmondható s^- -ről is, amely s_F miatt elérhető F -ből. Ezáltal a fixpont elérésekor mindenképp fennáll, hogy $s^- \in R^-$, $s^+ \in R^+$ és $s_F \in R_F$, tehát egyik halmaz sem lehet üres.

¹³Abban az esetben persze, amikor minden állapot elfogadó, nem érdemes felvenni a követelményt. Ha viszont nincs elfogadó állapot az adott állapottér-részletben, akkor el sem kell indítani a körkeresést

2. lemma. Az algoritmus egy, a három kiinduló halmaz és a szűkítő lépések által meghatározott maximális $G(V, E)$ körmentes irányított gráfot (DAG) legfeljebb $\min(|S^-|, |S^+|, |F|)$ iteráció alatt eltávolít a halmazokból.

Bizonyítás Tekintsük a DAG egy topologikus rendezését. Az első csomópont a konstrukcióból adódóan valamely S halmaz eleme, és a DAG maximális tulajdonsága miatt nem érthető el az őt megelőző S halmazból, ezért az algoritmus eltávolíthatja. Az olyan állapotok, amelyek nem részei egyetlen S halmaznak sem, gond nélkül eltávolíthatók amikor a topologikus rendezés szerint eléri őket az algoritmus. A következő valamilyen S -beli csomópontot már úgy éri el az algoritmus, hogy az előtte lévő csomópontokat mind eltávolíthatta, így ez a csomópont is kivágható. Amikor az algoritmus a rendezés utolsó csomópontjához ér, az is eltávolításra kerül, így a DAG összes csomópontjáról bebizonyosodik, hogy nem lesz része egyik R halmaznak sem.

Érdemes külön megvizsgálni azt a nemtriviális esetet, hogy lehetséges-e, hogy egy $s \in S^- \cap S^+ \cap F$ állapot kör jelenléte nélkül is megmaradhat-e. Abban az esetben, ha szűkítő lépések csak a tranzitív lezárttal való metszésből állnának, mindez lehetséges volna, mert a tranzitív lezárt a kiinduló csomópontokat is tartalmazza. Így egyik metszés sem távolítaná el a csomópontot, annak ellenére, hogy egyetlen csomópont jelen esetben nem tekinthető körnek.

Az egyik szűkítő lépés azonban nem tranzitív lezárttal, hanem az e -n keresztül egyetlen lépéssel elérhető csomópontokkal metsz, így az említett esetben két lehetőség adódik:

- $(s, s) \in e$, ebben az esetben viszont a hurokélen át mégis van egy kör, ami ellentmondás, mivel a lemma DAG-okról szól.
- $\exists s_1 \in S^-, s_2 \in S^+$, hogy $(s_1, s) \in e$ és $(s, s_2) \in e$. Ekkor s nem képes önmagában biztosítani, hogy ne kerüljön eltávolításra. s_2 -t az algoritmus a topologikus rendezésben előrébb állóként előbb-utóbb a metszés során elhagyja, ezután pedig s , ennek következtében pedig s_1 sem szerepel majd semelyik R -ben.

Az algoritmus minden szűkítő lépés során eltávolít legalább egy elemet az adott halmazból. Ha nem tenné, akkor a változatlan halmazzal szűkített következő halmaz sem változna, és fixpontra jutnánk. Mivel minden halmaz minden körben egyszer kerül szűkítésre, az algoritmus legfeljebb annyi teljes iterációt tesz meg, amennyi alatt a legkisebb halmaz elemei elfogynak, vagyis a lépésszám $\min(|S^-|, |S^+|, |F|)$.

3. lemma. Ha nem létezik irányított kör, amelynek elemei egy C halmazt alkotnak, és $\exists s^-, s^+, s_F \in C$ és $s^- \in S^-, s^+ \in S^+, s_F \in F$, valamint $(s^-, s^+) \in e$, akkor $R^-, R^+, R_F = \emptyset$.

Bizonyítás Ekkor a három halmaz (R^+, R^-, R_F) és a szűkítő lépések (amik meghatározzák, milyen módon juthat az algoritmus az egyik halmazból a következőbe) egy vagy több maximális irányított körmentes gráfot definiálnak. A 2. lemma értelmében a talált körmentes részgráfok mind lebontásra kerülnek, tehát ha nincs kör, az R halmazok üresek lesznek a fixponton.

Tétel Az algoritmus helyes és teljes.

Bizonyítás A bizonyítandó négy állítás közül az első kettőt az 1. és a 3. lemma látja be, a másik kettő pedig ezek segítségével indirekt módon bizonyítható.

2.4. Optimalizációk

Az algoritmus érzékeny pontja a szorzat automata mérete, hiszen a szaturációnak adott esetben a teljes szorzatot be kell járnia. A modell állapotterének csökkentésére használhatók a 2.1.1. fejezetben említett redukciók és egyszerűsítések, mint például a részleges rendezés, amely a tetszőleges sorrendben eltűzelhető események lehetséges permutációit reprezentálja kompaktan. Hasonló redukciókat érdemes alkalmaznunk a specifikációs automatára is, hiszen a szorzat állapotter mérete szempontjából meghatározó a kisebb automata állapotterének méretére.

A redukciókat és egyszerűsítéseket már az LTL kifejezés szintjén érdemes elkezdni, így eleve kisebb automatához juthatunk. A transzformációs algoritmus elvégzése után pedig a keletkező automatát számos technika segítségével még tovább csökkenthetjük.

2.4.1. Az LTL kifejezés egyszerűsítése

Negált normál formába hozás után az LTL kifejezés számos logikai azonosság mentén egyszerűsíthető. Az alábbiakban bemutatom a [8]-ban javasolt redukciókat.

Ehhez elsőként két fogalmat kell bevezetni, amelyek nyelvek egy-egy speciális osztályát definiálják. Az ezekre vonatkozó állítások és tételek felhasználásával lehetséges a kifejezés rövidítése.

Definíció Egy ω -reguláris $\mathcal{L}$ nyelv zárt a véges prefixálásra¹⁴, ha minden $w \in \Sigma^\omega$ és $v \in \Sigma^*$ szavakra igaz, hogy amennyiben $w \in \mathcal{L}$, akkor $vw \in \mathcal{L}$.

Állítás Ha adott egy ψ LTL kifejezés, amelynek nyelve zárt a véges prefixálásra, illetve egy tetszőleges γ formula, akkor igazak a következő azonosságok: (1) $\gamma \cup \psi \equiv \psi$, (2) $F \psi \equiv \psi$.

Általánosságban PSPACE-nehez kérdés az, hogy egy kifejezés nyelve zárt-e a véges prefixálásra, azonban a következő fogalom és a rá épülő tétel segítségével egy elégséges kritérium adható, ami hatékonyan felhasználható a kifejezés egyszerűsítésére.

Definíció A tisztán élőségi kifejezések családjába azok az LTL kifejezések tartoznak, amelyek negált normál formáira igazak a következők:

1. Minden $F \varphi$ alakú formula tisztán élőségi kifejezés.
2. Ha adottak ψ_1 és ψ_2 tisztán élőségi kifejezések, valamint egy tetszőleges γ LTL kifejezés, akkor $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, $\psi_1 \cup \gamma$, $\psi_1 R \psi_2$, $G \psi_1$ és $X \psi_1$ is tisztán élőségi kifejezések.

4. lemma. Minden φ tisztán élőségi kifejezés definiál egy véges prefixálásra zárt $\mathcal{L}(\varphi)$ nyelvet.

A bizonyítás mellőzése mellett a 4. lemma segítségével a véges prefixálásra zárt nyelveket leíró kifejezések nagy többsége felismerhető.

Hasonló módon bevezethetők az utótagokra zárt nyelvek és a tisztán invariáns kifejezések fogalmai is.

Definíció Egy ω -reguláris $\mathcal{L}$ nyelv zárt az utótag képzésre, ha minden $w \in \mathcal{L}$ szó w' utótagjára $w' \in \mathcal{L}$.

Állítás Ha adott egy ψ LTL kifejezés, amelynek nyelve zárt az utótag képzésre, illetve egy tetszőleges γ formula, akkor igazak a következő azonosságok: (1) $\gamma R \psi \equiv \psi$, (2) $G \psi \equiv \psi$.

¹⁴Prefixálás: előtag balról történő illesztése.

Definíció A tisztán invariáns kifejezések családjába azok az LTL kifejezések tartoznak, amelyek negált normál formáira igazak a következők:

1. Minden $G \varphi$ alakú formula tisztán invariáns kifejezés.
2. Ha adottak ψ_1 és ψ_2 tisztán invariáns kifejezések, akkor $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, $\psi_1 U \psi_2$, $\psi_1 R \psi_2$, $G \psi_1$ és $X \psi_1$ is tisztán invariáns kifejezések.

5. lemma. Minden φ tisztán invariáns kifejezés definiál egy utótag-képzésre zárt $\mathcal{L}(\varphi)$ nyelvet.

A 4. és 5. lemmákból, az állításokból, valamint a temporális logika szemantikájából következnek az alábbi átírási szabályok:

1. $(\varphi U \psi) \wedge (\gamma U \psi) \equiv (\varphi \wedge \gamma) U \psi$
2. $(\varphi R \psi) \vee (\gamma R \psi) \equiv (\varphi \vee \gamma) R \psi$
3. $(\varphi U \psi) \vee (\varphi U \gamma) \equiv \varphi U (\psi \vee \gamma)$
4. $(\varphi R \psi) \wedge (\varphi R \gamma) \equiv \varphi R (\psi \wedge \gamma)$
5. $F(\varphi U \psi) \equiv F \psi$
6. $G(\varphi R \psi) \equiv G \psi$
7. Ha ψ egy tisztán élősségi kifejezés, akkor $(\varphi U \psi) \equiv \psi$ és $F \psi \equiv \psi$.
8. Ha ψ egy tisztán invariáns kifejezés, akkor $(\varphi R \psi) \equiv \psi$ és $G \psi \equiv \psi$.

Az azonosságok mindegyikére igaz az, hogy a jobb oldal legalább eggyel kevesebb operátort tartalmaz, mint a bal. Mivel a transzformációs algoritmus érzékeny a bemeneti LTL kifejezésben szereplő operátorok számára, az átírásokkal szerencsés esetben sok felesleges automata állapotot spórolhatunk meg. Az egyszerűsítések természetesen „kézzel” is alkalmazhatók, egy jól megválasztott specifikációs kritérium nem mindig lesz tovább egyszerűsíthető.

2.4.2. A Büchi automata egyszerűsítése

Miután az LTL kifejezésből nyert általánosított Büchi automatát egyszerű Büchi automatává konvertáltuk, transzformációs algoritmus és a konverzió által létrehozott redundanciát többféleképpen is csökkenthetjük. Ezeket a megoldásokat mutatom be az alábbiakban. A második és harmadik redukció szintén [8]-ból származik.

Elfogadó állapotok redukálása

Már a transzformációs algoritmus is sok felesleges elfogadó állapotot visz az automatába, az egyszerű Büchi-automatává alakítás ezeket meg is többszörözheti. Ha az általánosított Büchi automatában nincs elfogadó állapot halmaz, a megfelelő egyszerű Büchi automatában minden állapot elfogadó lesz. Semmi haszna azonban az olyan elfogadó állapotoknak, amik nem szerepelnek egy körben, hiszen egy elfogadó lefutásnak végtelen sokszor kell elfogadó állapotot érintenie, ez pedig csak a körökben lévő elfogadó állapotok segítségével lehetséges.

Mint korábban, körök helyett itt is beszélhetünk hurokélekkel rendelkező állapotokról és nemtriviális erősen összefüggő komponensekről. Erősen összefüggő komponensek keresésére hatékony explicit módszerek léteznek, amelyek az automata kis mérete miatt itt is használhatók. A modellellenőrzőben Tarjan mélységi bejárás alapú algoritmusát [13] fogom felhasználni az erősen összefüggő komponensek keresésére.

Felesleges tehát minden olyan elfogadó állapot, amely nem rendelkezik hurokállal, vagy nem tartozik egy nemtriviális erősen összefüggő komponenshez – ezek eltávolítása a kifejezés által leírt nyelvet nem befolyásolja. Rendkívül sokat javít azonban a körkereső algoritmus hatékonyságán, mivel a 2. lemmában bemutatott lépésszámot a legtöbb esetben direkt módon csökkenti. Ezt a hatást megvizsgálom a 4. fejezetben is.

„Zsákutca” állapotok eltávolítása

A felesleges elfogadó állapotok eltávolítása után az olyan állapotokat is érdemes törölni, amelyekből nem érhető el elfogadó kör. Ezek meghatározásához először a Tarjan algoritmus segítségével meg kell határozni az elfogadó erősen összefüggő komponenseket, illetve a hurokállal rendelkező elfogadó állapotokat, majd ezekből képezni kell a visszafelé elérhető állapotok halmazát. Minden olyan állapot, amely nem része az így kapott halmaznak, törölhető.

Azonos élcímkéjű elfogadó zárt körök redukálása

Ez a redukció *nem ekvivalens* véges szavakon működő véges automaták esetén, azonban Büchi automaták esetén igen. Az alapgondolat az, hogy ha egy olyan körbe jutottunk, ahonnan nem lehet kijutni, van benne elfogadó állapot, és a kör állapotai között futó minden él címkéje megegyezik, akkor az egész komponens kezelhetjük egyetlen, a közös címkével ellátott hurokállal rendelkező elfogadó állapotként.

Definíció Azonos élcímkéjű elfogadó zárt körnek nevezzük egy $Q' \subseteq Q$ állapothalmazt az A Büchi automatában, ha

1. α az egyetlen kifejezés, ami a Q' -n belül futó tranzíciók örfeltétele, vagyis $\forall (q_1, \xi, q_2) \in \Delta, q_1, q_2 \in Q'$ esetén $\xi = \alpha$.
2. Q' elemei egy irányított kört alkotnak.
3. Q' -ből nem vezet ki tranzíció, tehát $\neg \exists (q', \xi, q) \in \Delta$, ahol $q' \in Q'$ és $q \notin Q'$.
4. Q' tartalmaz elfogadó állapotot, vagyis $Q \cap F \neq \emptyset$.

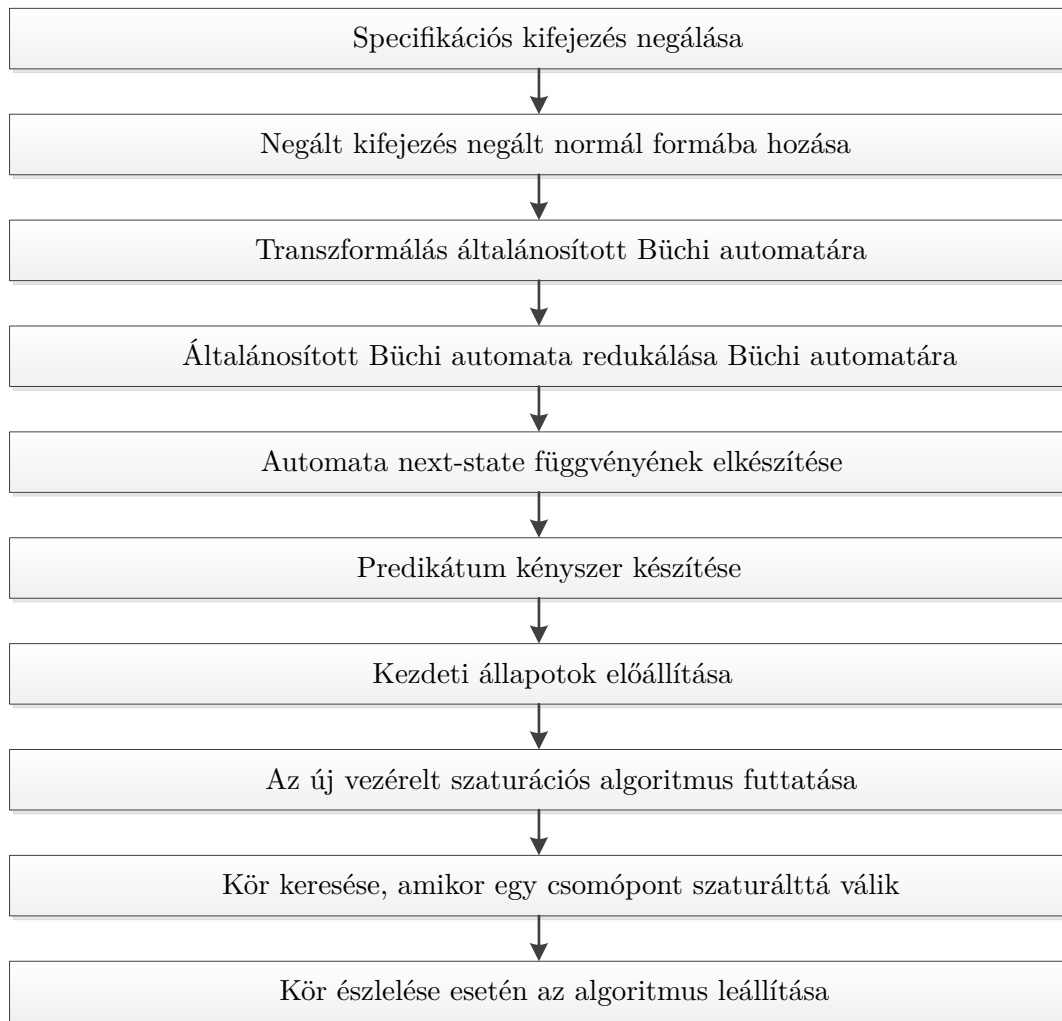
Az átalakítás nem érinti a Büchi automata nyelvét [8]. Azonos élcímkéjű elfogadó zárt körök – bár nagyon speciálisnak tűnnek – igen gyakran előfordulnak az olyan Büchi automatákban, amik általánosított Büchi automaták átalakításával keletkeztek. A több elfogadó állapot halmaz ugyanis meg többszörözhet egy állapotot, amelynek másolatai többnyire ilyen komponenseket alkotnak. Az 1.5. ábrán látható, egyetlen operátorból álló kifejezésekből alkotott általánosított Büchi automatákból készített egyszerű Büchi automatákban is van ilyen kör.

2.5. Az algoritmus összefoglalása

Ezzel minden, a bevezetőben vázolt problémára bemutattam a megoldásomat. Összefoglalva, az LTL modellellenőrző algoritmusom a 2.8. ábrán látható folyamattal szemléltethető.

Az automata next-state függvényét reprezentáló MDD-t minden esemény alsó két szintjére illesztjük. Az elfogadó állapotokat tartalmazó köröket kereső algoritmust a 2. algoritmus 22. sora után kell elindítani.

Ha a modell teljesíti a specifikációt, az algoritmus végigfut, és a felderített automata nyelve üres lesz. Ha létezik olyan lefutás, ami megsérti a specifikációt, akkor az algoritmus az első ilyen lefutás észlelésekor leáll, a lefutás rekonstruálásához pedig értékes adatokat szolgáltat. Megjegyzendő, hogy az így kapott ellenpélda nem feltétlenül a legrövidebb.



2.8. ábra. A szaturáció alapú LTL modellellenőrző lépései.

Ebben a fejezetben bemutatottam minden elméleti eredményt, ami a hatékony LTL szaturáció megvalósításához szükséges. Az általam bevezetett új megközelítések, megoldások és algoritmusok már lehetővé teszik egy, a gyakorlatban is használható, nagy teljesítményű, szaturáció alapú LTL modellellenőrző implementálását.

3.

Implementáció

Az előző fejezetben ismertetett elméleti eredményeket a tanszéken fejlesztett PetriDotNet [18] keretrendszerben implementáltam. A PetriDotNet egy .NET alapú Petri-háló szerkesztő program, amely rugalmas plugin-rendszerének köszönhetően számtalan, többnyire szintén a tanszéken fejlesztett bővítménnyel rendelkezik. Ezek között megtalálható az összes eddigi, Petri-hálókra épülő tanszéki modellellenőrző megoldás [4, 5, 16].

Azzal, hogy ezt az eszközt választottam, egyúttal az is eldőlt, hogy az algoritmusomat Petri-hálókra kívánom futtatni. Ez önmagában nem lenne követelmény, hiszen maga az algoritmus csak kódolt állapotokkal dolgozik, azonban így könnyen fel tudtam használni a meglévő kódokat. A céloom elsősorban színezetlen hálók kezelése, azonban a kódot úgy készítettem el, hogy később könnyedén lehessen bővíteni színezett modellekre is.

Ebben a fejezetben részletesen bemutatom a fejlesztés folyamatát és a közben felmerült tervezői döntéseket és megoldásokat. Először vázolom a program architektúráját, majd az előző fejezet fő pontjai szerint bemutatom a megoldott problémákat és az érdekesebb megoldásokat részletesebben is kifejtem.

3.1. Áttekintés

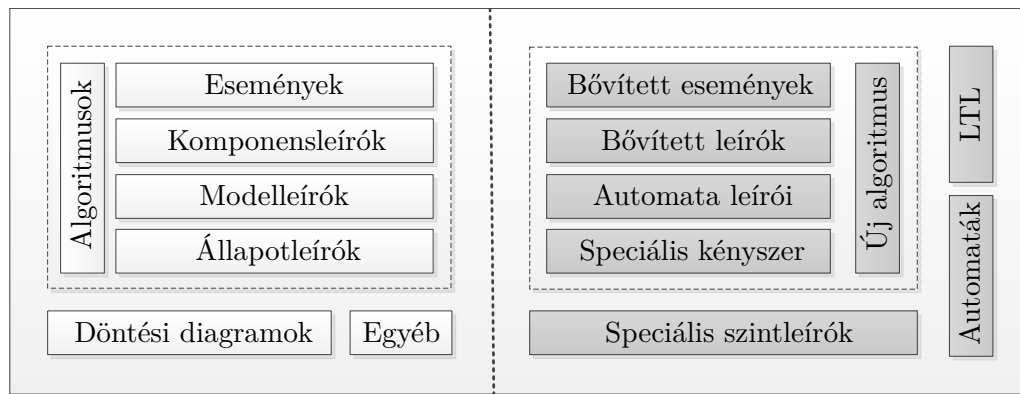
Az implementáció során a korábbi szaturációs megoldásokat is tartalmazó MDDAnalysis-Component nevű plugint bővítettem ki. Ez a komponens tartalmazza az MDD-k implementációját, illetve számos működő szaturációs algoritmust a hozzájuk tartozó adatszerkezetekkel, melyek megteremtik a kapcsolatot a modell és az állapottere között. Ezeket az adatszerkezeteket bővítettem illetve módosítottam kis mértékben, létrehoztam egy új osztályt a szorzat automata állapotterének felderítésére, és természetesen bevezettem egy-egy csomagot az automaták és az LTL kifejezések kezelésére. A 3.1. ábra mutatja a bővített komponens jelen munka tekintetében fontos összetevőit, szürke háttérrel jelölve az általam megírt részeket.

3.2. LTL és automaták

A modellellenőrzés kezdetekor a felhasználótól be kell kérni az ellenőrizendő LTL kifejezést. Ez egy karakterlánc, amit fel kell dolgoznom egy automatává, majd az automata segítségével le kell generálnom a meglévő szaturációs adatszerkezetek kiegészítéseit.

3.2.1. LTL kifejezések

A kifejezés tartalmazhatja az összes temporális operátort a hagyományos jelölésük szerint (tehát például az **U** operátort nagy U betűvel kell beírni), illetve ötféle logikai operátort:



3.1. ábra. Az MDDAnalysisComponent.dll felépítése, szürkével jelölve az új részeket.

vagy (+), és (*), kizáró vagy ($\neg$), implikáció ($\Rightarrow$, csak egy irányba) és ekvivalencia ($\Leftrightarrow$). Az operátorok precedenciája a szokásos, zárójelek alkalmazhatók. Azok a szövegrészek, melyek nem tartalmaznak operátorokat, atomi kijelentésekként kerülnek feldolgozásra, azonban csak a kifejezés *lefordításakor*, ami a kifejezés élelciklusában bármikor¹ megtörténhet a megfelelő modell megadásával.

Egy atomi kijelentés színezetlen Petri-hálók esetén tartalmazhatja egy hely azonosítóját, egy összehasonlító operátort ($>$, $>=$, $<$, $<=$, $=$, $\neq$), valamint egy nemnegatív egész számot. Egy ilyen állítás az adott helyen lévő tokenek számára ad predikátumot. Érdeemes megjegyezni, hogy több ilyen állítás konjunkciója vagy diszjunkciója nem számít atomi kijelentésnek, még akkor sem, ha ugyanarra a helyre vonatkoznak. Érvényes atomi kijelentés még a **true** és a **false** is, melyek a mindig teljesülő *True* és a soha nem teljesülő *False* értékekké fordulnak le.

Az algoritmusomban a kifejezések reprezentálására kifejezésfákat használok, az alábbi sorok ezt a struktúrát mutatják be.

Egy kifejezésfa egy olyan fa struktúra, amelynek belső csomópontjai operátorok, levelei pedig atomi kijelentések. LTL kifejezésfa esetében a levelek állapotkifejezések, a csomópontok pedig a Boole-algebrából ismert, valamint a korábbiakban definiált temporális operátorok. Egy kifejezésfa kiértékelésekor rekurzívan járunk el, a kifejezésfa csúcsán álló, legkisebb precedenciájú csomópont úgy állítja elő az eredményt, hogy a gyereke(i) által visszaadott eredményt használja fel. Ezáltal a fában alul elhelyezkedő, magas precedenciájú operátorok állítják elő először az eredményeiket az atomi kijelentések segítségével, amelyeknek a kiértékeléskor mindig van valamilyen értékük, és ezek az eredmények terjednek felfelé a fában.

A kifejezésfa felépítése ezek után természetesen precedencia alapján történik. A legkisebb precedenciájú operátor lesz a gyökér (ha több is van, a bal oldalt választjuk), majd a két oldalon ismét így választunk gyökeret a két részfának. A zárójelek értelemszerűen olyan al-fákat képviselnek, amiben a legkisebb precedenciájú csomópont képviseli az egész alkifejezést, de a felépítéskor mindennél nagyobb precedenciájúnak kezeljük a zárójel miatt.

Szemléletesen úgy képzelhetjük el a folyamatot, hogy az inorder bejárással kiterített kifejezésfát „megfogjuk” és megemeljük a fő operátora mentén, ami így magával húzza a saját operandusait, majd azok a sajátjaikat, és végül megkapjuk a kifejezésfát.

Ez a konstrukció azért nagyon előnyös, mert a részfák képében hatékonyan kezelhetők alkifejezések, a műveletek pedig lokálisan és rekurzívan könnyedén végrehajthatók az egyes csomópontokon.

¹Ez azt jelenti, hogy akár az automata megkonstruálása után is elegendő lefordítani a kijelentéseket.

Az LTL minden operátorához készítettem egy osztályt, illetve további négyet az atomi kijelentéseknek, a `true`-nak és a `false`-nak, valamint a zárójeleknek. Utóbbi osztály egy feldolgozott kifejezésben önállóan sosem jelenik meg, a kifejezésfa felépítése közben azonban jelentőséggel bír. Ezeket az osztályokat hierarchiába szerveztem, ahol az alaptípus a `Node` lett. Ebből származik az `Operator` és az `AtomicProposition`. Az `Operator` leszármazottai a `UnaryOperator` és a `BinaryOperator`, ezekből származik az összes operátor az operandusai számának megfelelően. A `Parenthesis` osztály közvetlenül az `Operator` osztálytól származik, hogy megkülönböztethető legyen a klasszikus operátoroktól.

A beérkező karakterláncot a hatékonyság érdekében először az elejétől a végéig feldolgozom, és egy, a fenti osztályok példányaiból álló listát hozok létre belőlük, amelyben minden objektum azon a helyen szerepel, ahol a szövegben szerepelt a forrása. Ezután a kifejezésfáá alakítás egy rekurzív függvénnyel történik, amely először megkeresi a legkisebb precedenciájú operátort (több azonos esetén a bal oldalt), majd ha az nem atomi kifejezés, akkor operandusainak előállítására meghívja önmagát a jobb (és bináris operátor esetén a bal) oldalán lévő listarészletekre is. A zárójelek kezelése némileg áthágja ezt a folyamatot, mivel zárójelek észlelésekor az egész zárójeles részt külön dolgozzuk fel már az objektumok létrehozásakor, és egy `Parenthesis` objektumba eltesszük a kialakult kifejezésfát a listába. Ha a kifejezésfáá alakítás során egy zárójeles kifejezést értékül adnánk egy operátor valamely argumentumának, vagy a teljes fa gyökere lenne zárójel, akkor a zárójelet kicseréljük a benne található gyökércsomópontra.

A kapott kifejezésfa negált normál formára hozása a `Node` osztály absztrakt metódusával kezdeményezhető, és rekurzívan hívódik végig a kifejezésfán:

- Az **F** és **G** operátorokat ki kell cserélnünk a fentiek szerint **U** és **R** operátorokra, majd venni ezek negált normál formáját.
- Az implikáció ($\Rightarrow$), az ekvivalencia ($\Leftrightarrow$) illetve a kizáró vagy ($\oplus$) operátorokat kifejtjük és átalakítjuk az alapoperátorokkal kifejezett formájukra, majd vesszük a negált normál formájukat.
- Az **U**, **R**, **X**, $\wedge$ és $\vee$ operátorok esetén az operandusokat átalakítjuk a negált normál formájukra.
- A $\neg$ operátor esetén az operandust átalakítjuk a negált normál formájára, majd az új operandus típusa szerint a következőket tehetjük:
 - **U**, **R**, **X**, $\wedge$ és $\vee$ típusú operandusok esetén a negálást átalakítjuk a fenti azonosságok (De Morgan, LTL) szerint és vesszük az eredmény negált normál formáját.
 - $\neg$ esetén megszüntetjük a dupla negálást és vesszük a belső negálás operandusának negált normál formáját.
 - *True* vagy *False* értékek esetén a negált konstans helyére behelyettesítjük az ellentettjét.
 - Atomi kijelentések esetén nem teszünk semmit.
- Ha a csomópont egy atomi kifejezés, akkor nem kell tennünk semmit.

A keletkező `Node`-fát egy `Expression` nevű típus foglalja magába, ami felelős a kifejezés kezeléséért és felépítéséért, valamint tárolja a kifejezésben szereplő atomi kijelentéseket és megakadályozza, hogy ugyanaz a kijelentés több objektumpéldányban is megjelenjen².

²Így egy atomi kijelentéshez pontosan egy objektum tartozik, a továbbiakban tehát elegendő a referenciájukat használni a megkülönböztetésükre.

3.2.2. Automaták

Az automaták csomagját néhány nagyon egyszerű elemből építettem fel. Létrehoztam egy **State** és egy **Transition** típust, és a belőlük építkező **Automaton** osztályt. A **State** osztály példányai egyedi azonosítóval rendelkeznek, ismerik a beléjük futó és belőlük induló tranzíciókat, valamint azon elfogadó állapot halmazok azonosítóit, amikben szerepelnek. A **Transition** osztály példányai ismerik a forrás és a célállapotot, illetve tartalmaznak egy LTL kifejezések konjunkcióját reprezentáló **Node** listát (ebben csak atomi kifejezések és negáltjaik szerepelhetnek) is, ami az élkifejezést adja meg. Praktikus okokból a **State** osztályban helyt kap a konstrukciós algoritmus során keletkező **Expressions** tulajdonság is, ami megegyezik az befutó tranzíciók élkifejezéseivel. Egy **Automaton** objektum az automata 1.3. fejezti definíciójának megfelelően tartalmaz egy állapothalmazt a lehetséges állapotok és a kezdőállapotok, valamint egyet-egyét a különböző elfogadó állapot halmazok reprezentálására. Tartalmaz ezen kívül egy **Transition** halmazt is a lehetséges átmenetek tárolására.

Egy **Automaton** objektum egy **Expression** objektumból konstruálható. A kifejezésnek normál formában kell lennie, ha ez nem teljesül, először meghívódik a **ToNegationNormalForm()** metódus. A konstrukciós algoritmus az 1.3.4. fejezetben leírtaknak megfelelően működik, azonban rekurzív hívások helyett egy **Stack** használatával iteratíván. Az algoritmusban említett *csomópont* struktúrát a **ConstructionNode** osztály valósítja meg, amely a felsorolt öt mezőt **Node** listákként tartja számon. Mivel az itt szereplő kifejezések csak a kiinduló kifejezés alkifejezései lehetnek, új **Node**-ok létrehozására nincs szükség, elegendő referenciákat tárolni a kifejezésfa csomópontjaira. Az elfogadó állapotok a forrás **Expression** objektum által kigyűjtött **Until** objektumok alapján kerül megállapításra. A csomópontok átalakításakor a keletkező **State** objektumok azonosítói nullától, folytonosan lesznek számozva.

Egy **Automaton** objektum reprezentálhat hagyományos és általánosított Büchi automata is, utóbbi esetben képes önmagát hagyományossá alakítani. Ehhez az új állapotteret egy sorosított szélességi kereséssel deríti fel, ami az automaták viszonylag kis méretére való tekintettel elegendően hatékony megoldás³. Az átalakítás végére csak egyetlen elfogadó állapot halmaz lesz, és a csomópontok azonosítói ismét nullától, folytonosan lesznek számozva.

3.3. Szaturáció

Ebből az **Automaton** objektumból már könnyen előállíthatóak a szaturációs algoritmus kibővítéséhez szükséges adatszerkezetek. Az alábbiakban bemutatom, hogyan használtam fel a meglévő implementációt a saját munkámhoz, hogy egy, a továbbiakban is rugalmasan bővíthető rendszert kapjak.

3.3.1. Meglévő szaturációs kódok és felépítésük

Mint mondtam, a tanszék korábbi munkássága során sokféle, viszonylag kiforrott és optimalizált szaturációs algoritmus változat rendelkezésemre állt a fejlesztés során. Az algoritmus variációk ugyanazokon az alapvető adatstruktúrákon dolgoznak, esetenként kibővítve őket egy leszármazott osztállyal. Az alapvető építőkövek a következők:

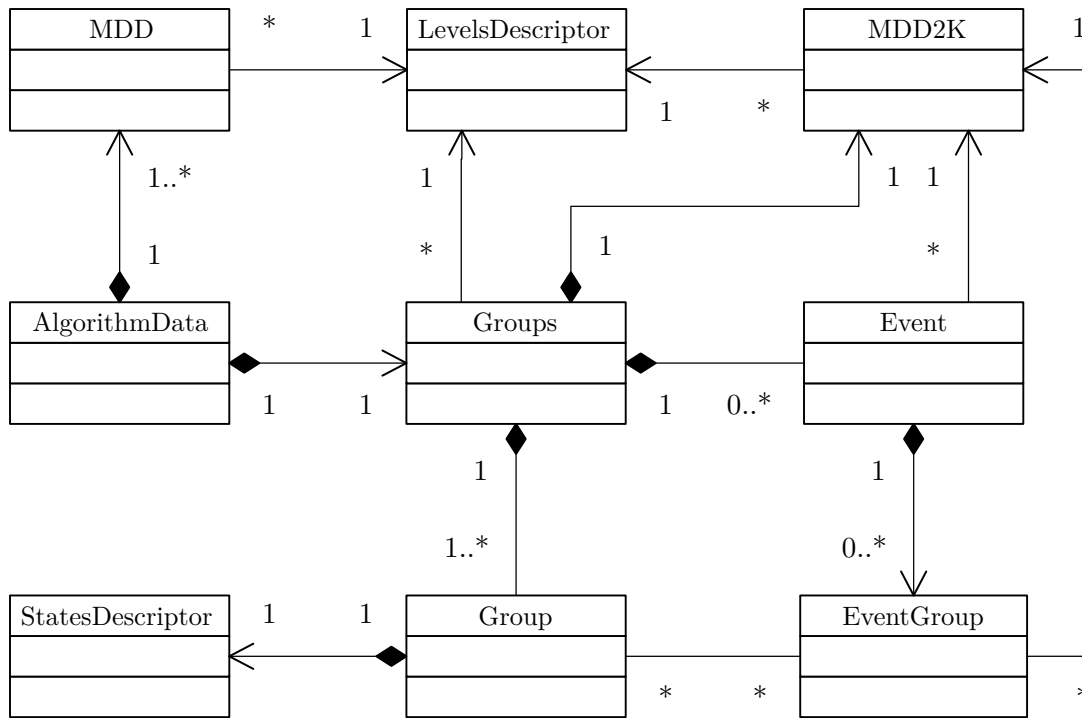
³Az 5.2. fejezetben vázolok még néhány lehetőséget, amik kihasználják azt, hogy az automata kis mérete miatt az előfeldolgozás fázisban explicit megoldások is használhatók.

- **LevelsDescriptor:** MDD-k struktúráját leíró objektumok, melyek tartalmazzák a szintszámot, a terminális csomópontokat, illetve az egyes szintek élszámát (értelmezési tartományuk méretét).
- **MDD:** Egy kváziredukált MDD-eket leíró osztály, aminek példányai egy LevelsDescriptor alapján épülnek fel.
- **MDD2K:** Egy speciális MDD, ami állapotátmeneteket kódol. Emiatt minden második szintje redukáltnak tekinthető, a többi továbbra is kváziredukált (az állapotátmenetet leíró két szint között nem kell csomópontnak lennie). Szintén egy LevelsDescriptorból épül fel, ami többnyire ugyanaz, mint amiből az állapottér MDD-je készül, ugyanis a konstruáláskor a LevelsDescriptor minden szintjéhez két szint készül.
- **Group:** A modell egy komponensét reprezentálja, van egy StatesDescriptor, egy szintszáma, illetve ismeri azokat az eseményeket, melyek *Top* értéke megegyezik a szintszámmal. A Groupok szintszám alapján vannak sorrendezve, és egy lefelé egyirányú láncba szerveződnek.
- **StatesDescriptor:** Egy Group állapotait írja le, ismeri a hozzá tartozó helyeket, megteremti a kapcsolatot a modellel. Számon tartja az összes eddig felfedezett állapotot és az ezek közül ténylegesen elérhetőket.
- **Event:** Többnyire egy modellbeli tranzíció által megléphető állapotátmeneteket képviseli. Az átmenetek egy MDD2K-ban vannak elkódolva, azonban kezelésüket ezek az objektumok végzik. Az átmeneteket lehetőség van több részletben, külön MDD2K-kban tárolni, ekkor a teljes állapotátmenet halmazt leíró metszet itt kerül kiszámításra. A részleteket tároló egy vagy több EventGroup itt kerül eltárolásra.
- **EventGroup:** Az Event egy vagy több szintet érintő részlete. Ismeri az érintett szinteket, tartalmazza és kezeli az adott szintekhez tartozó MDD2K részleteket és tüzeléskor a modell alapján további elérhetőnek tűnő állapotokat és állapotátmeneteket derít fel.
- **Groups:** A teljes modellt reprezentáló struktúra. Ismeri az összes Eventet, az összes Groupot, az állapotátmeneteket tároló MDD2K-t, illetve az állapottér LevelsDescriptorát.

A legtöbb osztálynak (azoknak, amelyek a modellel szorosabb kapcsolatban állnak) létezik színezett és színezetlen megfelelője is. A szaturációs algoritmusok különböző változatai ezeket az osztályokat bővítik a megfelelő működés elérése érdekében, az algoritmus maga pedig többnyire nagyon hasonlít valamelyik előzőhöz, többnyire csak annyi kiegészítéssel, hogy a saját Groups típusából példányosít egyet a futásához.

3.3.2. Konceptió

Ahhoz, hogy az algoritmusomat minimális kódduplikálással implementálhassam, én is hasonló útra léptem, azonban más megközelítést alkalmaztam. A modell típusa teljesen indifferens az algoritmus szempontjából. A meglévő adatszerkezetek mindegyike arra való, hogy a szaturációs algoritmus felé elfedje a modell sajátosságait, így jó lenne ezeket felhasználni. Éppen ezért az adatszerkezeteimet a legáltalánosabb osztályoktól származtattam, és úgy terveztem meg őket, hogy belül bármilyen meglévő osztályt tartalmazhassanak, mint a szorzat állapottér állapotainak valamilyen modellhez kapcsolódó részét, mellette az automata állapotterét reprezentáló saját adatstruktúrával. Az alapelv az lenne, hogy a meglévő osztályok csak egymással állnak kapcsolatban, és nem is tudnak arról, hogy valójában egy *burkoló osztály* részét képezik.



3.2. ábra. Meglévő kódok osztálydiagramja.

Sajnos ez nem minden esetben tehető meg, például az MDD-knek mindenképpen közösnek kell lennie. Ráadásul a 2.2. fejezetben elmondottaknak megfelelően az állapotok és események automatához kapcsolódó része az MDD-k alján kap helyet. Ez – bár az elmondott okokból szükséges – rendkívüli nehézségeket okozott az implementálás során, mivel így a kiegészítéseknek mindenképpen az 1-es szintszámot kell megkapniuk, tehát minden meglévő struktúra szintszámát eggyel meg kellett emelni. A bemutatott adatszerkezetben viszont, mint írtam, nagy szerepe van a szintszámoknak, az Eventek, a Groupok, az állapotter MDD és az események MDD2K-ja mind szintszám alapján állnak kapcsolatban, és a legtöbb esetben nagyon kényes helyeken kellett volna belenyúlni a kódjukba, hogy az átszámozást elérjem. Az alábbiakban bemutatom a választott megoldást⁴.

3.3.3. Burkoló osztályok

Az alapötlet tehát az volt, hogy a saját adatszerkezeteimet meglévő osztályok köré építem fel, *burkoló osztályokat* létrehozva. Új komponenst csak a Groupsba és az Eventbe kell beépíteni, ezek a részek tartalmazzák az automata állapotait és állapotátmeneteit.

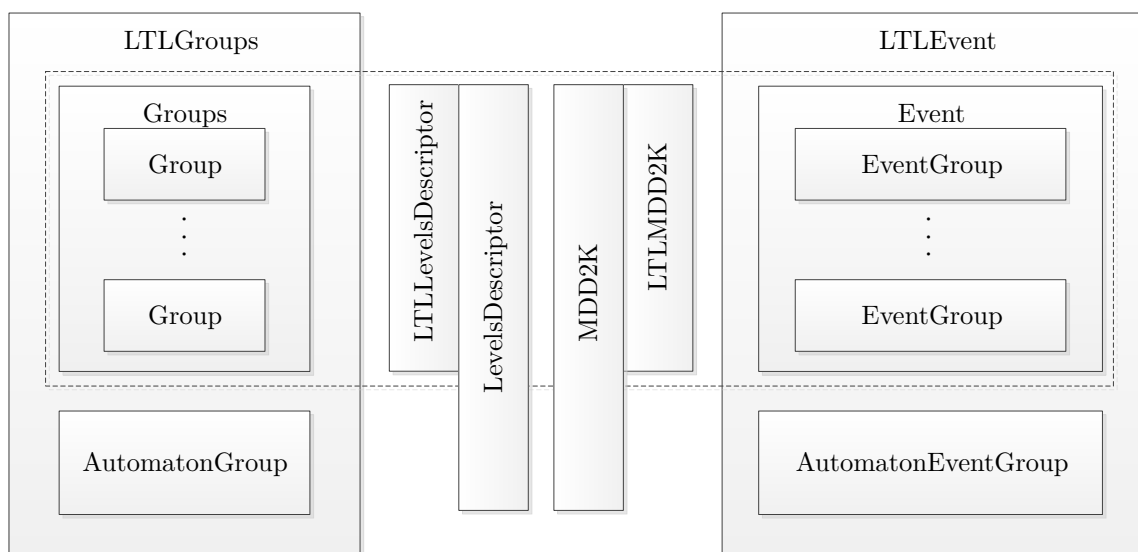
További burkolókat kellett létrehoznom a LevelsDescriptorhoz, és az MDD2K-hoz is a fentebb említett, szintszámmal összefüggő problémák miatt. A megoldásom lényege ugyanis az, hogy a meglévő adatszerkezeteket lényegi módosítás nélkül alkalmazom, és nem is tudnak róla, hogy burkoló osztályok részei. A modell állapotterét leíró objektumok pontosan úgy fognak viselkedni, mintha semmilyen automata nem lenne az algoritmusban. A belül tárolt szintszámuk megmarad, egymással kapcsolatban maradnak, a burkoló osztályok pedig elhítetik velük, hogy nincs külön legalsó szint, a szükséges adatokat és metódushívásokat pedig a szintszám transzformálásával közvetítik befelé.

Az ehhez szükséges burkoló osztályok és funkciójuk:

⁴ Az 5.2. fejezetben foglalkozni fogok az eszköz és az analízis modul jövőjével is. Az itt leírt, nem minden esetben kifinomult megoldásokat a PetriDotNet2 elkészültekor – amelyen jómagam is keményen dolgozok – egy alaposan megtervezett és újragondolt struktúrával fogjuk kiváltani.

- **LTLLevelsDescriptor:** Egy másik, belécsomagolt LevelsDescriptor tulajdonságait utánozza, azonban elrejt a legalsó szintet. Minden hívás, ami szintszámot tartalmaz, eggyel nagyobb szintre fog hivatkozni a belső LevelsDescriptorban (a 0. szintet leszámítva).
- **LTMDD2K:** A szerepe nagyon hasonló az előző osztályéhoz - elrejt a belé csomagolt MDD2K legalsó két szintjét. Saját LevelsDescriptorát egy LTLLevelsDescriptorba csomagolva tárolja.
- **LTLEvent:** Magába foglal egy meglévő eseményt, valamint az automata állapotátmeneteit leíró osztály egy példányát. Az esemény MDD2k-jának megépítéséhez először meghívja a belső esemény Build függvényét egy LTMDD2K objektummal, majd az így kihagyott legalsó két szintre beírja az automata állapotátmeneteit.
- **LTLGroups:** Magába foglal egy Groups objektumot és az automata állapotát számon tartó osztály egy példányát, valamint LTLEventeket tárol. Ezek létrehozásakor a belső Groups által létrehozott eseményeket csomagolja be és cseréli ki rájuk a belső Groups és az abban lévő Group-ok eseményeit. Állapotok megerősítésekor (Confirm hívás) újraépíti az összes esemény MDD2K-ját, hogy az új állapotot és átmenetek felderítésekor keletkező új MDD2K-k is tartalmazzák az automata szintet. A belső MDD2K-nak egy LTMDD2K-t ad értékül, illetve a LevelsDescriptor is becsomagolva adja tovább a belső Groupnak.

A Groups esetében még egy dolgot meg kellett oldani. Mint írtam, a Groupok össze vannak láncolva, így nekem is be kellett kötnöm az utolsó Group alá a saját, automatát leíró Group-omat. Ez a láncolás viszont teljesen a Group belsejében történik a Groupok létrehozásakor, így ezt a részt át kellett írnom. Az átírást mindenképpen szükségessé tette, hogy a StateDescriptorokat is kibővítssem, amiről a következőkben még tárgyalni fogok. Végül a kódban félig (a színezett hálók esetén) eddig is jelenlévő „Factory mintát” alkalmaztam, a Groupok létrehozását így delegálni tudtam a külső LTLGroupsnak, miközben annak híján a meglévő osztályok működése nem változott.



3.3. ábra. Az új és a meglévő osztályok viszonya. Bekeretezve láthatók a korábbi működést biztosító részek.

3.3.4. Új adatszerkezetek és felépítésük

Az automata reprezentálására további osztályokat is létrehoztam, melyek közül az `AutomatonGroup` és az `AutomatonEventGroup` a burkoló osztályok másik tagját képezi a modellt képviselő meglévő osztályok mellett, míg az `AutomatonStatesDescriptor` az automata állapotait tartja számon, az `UnColouredPredicateStatesDescriptor` pedig a modell állapotait bővíti ki a predikátumok igazságtartalmával az egyes állapotokban. Utóbbi `StateDescriptor` veszi át a színezetlen `Groupokban`⁵ az `UnColouredStatesDescriptor` osztály helyét, a cserét a `Groupok` készítésének említett delegálása teszi lehetővé. Az új osztályok feladata a következő:

- **`AutomatonStatesDescriptor`:** Csak az automata állapotainak számát tárolja, mivel számozásuk nullától, folytonosan történik. Ebből az információból előállítja a `StateDescriptor`októl elvárt adatokat.
- **`AutomatonGroup`:** Az osztály egyetlen feladata, hogy „elnyelje” a bele érkező, többnyire értelmetlen hívásokat. A legelső szinten jelen konstrukcióban soha nem lesz esemény, új állapotot pedig nem fedezhető fel, hiszen mindet előre ismerjük a konstrukciós algoritmus folytán. Az állapotok tárolására egy `AutomatonStatesDescriptor`tal rendelkezik.
- **`AutomatonEventGroup`:** Minden `LTLEvent` részét képezi, az állapotátmeneteit tartalmazó `MDD2K`-t az automata `Transition` gyűjteményéből építem fel. Élfeltételtől függetlenül mindegyik átmenetet tartalmazza.
- **`UnColouredPredicateStatesDescriptor`:** A 2.2.3. fejezetben bemutatottaknak megfelelően szükség van az specifikációs kifejezésben szereplő atomi kijelentések szintekhez rendelésére, és az egyes lokális állapotokról meg kell tudni mondani, hogy mely, azonos szinten értelmezett predikátumot teljesítenek, hogy ezzel az információval léphessünk a predikátum kényszerben. Ezt az információt költséges lenne minden alkalommal kiszámítani, és az állapot felfedezése után nem is változik, így célszerű a felderítéskor meghatározni és letárolni az igazságtartalmakat. Ezzel az információval és a szinten értelmezett predikátumok listájával egészíti ki ez az osztály az `UnColouredStatesDescriptor`-t.

A `Groupok` létrehozásakor a `StateDescriptor`okat külön kigyűjtöm a predikátum információk miatt, ebből építem fel ugyanis a kényszer `MDD`-t, amit a most következő fejezetben mutatok be.

3.3.5. A szaturációs algoritmus módosításai

Maga a szaturációs algoritmus és a hozzá szükséges lokális adatok a beszédes nevű `MDDProduct2KConsSaturationWithCircleDetectionData` osztályban kaptak helyet. Magát a szaturációs kódot részben át lehetett venni a meglévő implementációkból, hiszen az adatszerkezetek manipulálása pont azt a célt szolgálta, hogy az algoritmust ne kelljen túlságosan átírni.

Apró változtatások ennek ellenére szükségesek voltak. Egyrészt a klasszikus szaturáció implementációja a kezdőállapotot nem építette fel előre, és nem az állapotter-MDD gyökerét kezdte el szaturálni, hanem ezzel ekvivalens módon alulról indult el, és az egyes csomópontok szaturálásakor lépett csak feljebb. Nekem azonban szükséges volt a teljes kezdőállapoton végighaladni, mivel a kényszer léptetése csak így lehetséges. Másodszor ugyan létezett már vezérelt szaturációs kód, azonban azt az eddigi algoritmusok csak modellellenőrzés során, operátorok kiértékelésére használták, mégpedig *visszafelé* szaturálva.

⁵Ugyanilyen elven lehet kibővíteni a működést színezett esetre is.

További módosítást igényelt még a körkeresés indításának szükségessége. Lehetővé kellett tennem, hogy igény szerint egy csomópont szaturálttá válása után körkeresés indulhasson, viszont a keresés közben szaturált csomópontok esetében már ne. Ezen kívül, mint a 2.3.2. fejezetben említettem, visszafelé is kell tudni szaturálni. Ehhez nem kell mást tenni, mint az állapotátmenetek kiolvasásakor megcserélni a visszkapott két értéket, illetve a fejezetben leírtak szerint „hagyományosan” értelmezni a kényszert. Mindez egy kapcsolóval könnyedén, állítható módon megtehető, anélkül, hogy a kódot még egyszer meg kellene írni.

A kényszerhez kapcsolódik az utolsó változás, előrefelé szaturáláskor ugyanis nem az aktuális állapot, hanem a rá érvényes predikátumok szerint kell lépni. Ezt a logikát egy `PredicateMDD` nevű osztályba zártam, ami képes egy adott szintszám, a jelenlegi kényszer csomópont, és a leendő állapot alapján visszaadni a kényszer következő csomópontját (ami lehet terminális nulla, egy másik csomópont, vagy akár ismét a jelenlegi).

Ezt a struktúrát is előre fel lehet építeni az automatából. A konstrukció után is letárolható a minden adott állapotba mutató élen azonos élkifejezés, mégpedig negált vagy ponált atomi kijelentések konjunkciójaként (gyakorlatilag egy listában). Ebből lehetőség van minden állapothoz felvenni az összes olyan utat egy MDD-ben, ami mentén teljesülnek a belelépéshez szükséges predikátumok.

Maga az MDD annyi szintű lesz, ahány féle atomi kijelentés szerepel a vizsgálandó LTL kifejezésben, illetve további egy a lehetséges automata végállapotoknak. Az alsó szintet leszámítva a diagram bináris lesz, hiszen minden kifejezés vagy teljesül, vagy nem. A kényszer MDD megépítése előtt sorrendezni kell az atomi kijelentéseket aszerint, hogy melyik szinten értelmezettek. Több értelmezett predikátum esetén egy állapottér MDD szinthez több kényszer MDD szint fog tartozni, viszont itt is ki kell alakítani egy tetszőleges, de fix sorrendet. Ezután az MDD-ben minden állapotra felveszünk egy rész-MDD-t, amik uniójaként előáll a predikátum kényszer MDD-je. Az egyes rész-MDD-k a következő módon kerülnek kialakításra:

- A legelső él az automata állapotának azonosítója lesz.
- Ha egy szinthez tartozó atomi kijelentés az állapot belépési feltételei között
 - ponáltan szerepel, behúzzuk az 1. élet, ha
 - negáltan szerepel, behúzzuk a 0. élet, ha
 - sem ponáltan, sem negáltan, akkor behúzzuk az 0. és az 1. élet is.

A nem behúzott élek automatikusan a terminális nullába mutatnak.

A kényszer léptetése a következőképpen zajlik. Ha a szinthez nem tartozik predikátum (ez az információ a kibővített `StatesDescriptor`-ban megtalálható), akkor visszaadjuk a beadott kényszer csomópontot. Ha van ezen a szinten értelmezett predikátum, akkor a sorrendnek megfelelően ellenőrizzük, hogy az adott állapotra igazak-e, és ha igaza, az 1. élen lépünk tovább, ha nem, akkor a 0-on. Azt a csomópontot adjuk vissza, amibe az összes releváns predikátum vizsgálata után jutunk.

Ezzel mindent elkészítettem, ami a szaturációhoz szükséges. Állíthatóan előre vagy hátra képes lépegetni az algoritmus, a módosított adatszerkezetekkel a szorzat állapotteret deríti fel, az előre szaturáláskor pedig a predikátum kényszer biztosítja az automatában is helyes lépéseket.

3.4. Körkeresés

Hátra van még a körkeresés. A 2.5. fejezetben leírtaknak megfelelően csak a csomópontok szaturálása után kell köröket keresni, mégpedig csak az új, elfogadó állapottal rendel-

kezőket. A `Saturate` hívások eredményeinek eltárolása miatt pedig elég csak akkor, ha ténylegesen újból le kellett futtatni a függvényt.

Az `LTLConsSaturate()` metódus, ami a kódban a `Saturate` függvény implementációja, paraméterként várja, hogy kell-e kört keresnie. Ez a paraméter alapértelmezésben igaz, és a rekurzív hívások során továbbterjed. A körkeresés közbeni szaturálásokkor ugyanis nem kell újból körkereséseket kezdeményezni, hiszen új állapotok olyankor nem kerülnek felderítésre. Ha kell kört keresni, és az `LTLConsSaturate` metódus végigfutott (tehát nem volt cache találat), akkor meghívódik a `DetectCircles()` metódus, ami egy `bool` visszatérési értékkel jelzi, hogy talált-e kört, ha igen, akkor pedig egy kiementi paraméterként visszaadja a körben lévő elfogadó állapotokat. Ekkor egy olyan flag kerül beállításra, amittől az algoritmus futása minden szinten (`Saturate`, `SatFire`, `SatRecFire`) megszakad, és az algoritmus a pillanatnyilag használt MDD csomópontokra történő `CheckIn` hívások után megáll⁶.

A `DetectCircles()` metódus nem valósítja meg általánosan a 2.3.2. fejezetben leírt körkereső algoritmust. Teljesítmény okokból és a fejlesztési idő csökkentése miatt csak az itt szükséges speciális esetre implementáltam⁷. Meghívásához szükséges a szaturált csomópont és az aktuális kényszer csomópont megadása. Az algoritmus követelményhalmazait már a függvény állítja elő, azonban csak akkor, ha van esemény a szinten, illetve találtunk már elfogadó állapotokat.

Ehhez először regenerálja azokat az állapotokat, amik a szaturált csomópont szintjén regisztrált eseményeken át elérhetők. Ehhez az `LTLSatFire` metódus kódját használja (eltávolítva belőle a ciklust), valamint a `SatRecFire` egy olyan verzióját, amit a `LTLSingleStep` függvény valósít meg, és nem szaturálja a keletkező csomópontokat. Így mindegyik esemény pontosan egyszer tüzel. Ezután a kapott halmazból ugyanígy visszalépve egyet elkészíti az események kezdőállapotait tartalmazó halmazt is. Ehhez már a szaturált csomópontot, mint állapotteret adja meg kényszerként, mivel visszafelé szaturáláskor ebből a halmazból nem léphetünk ki, de a predikátum kényszerre már nincs szükség.

Az elfogadó állapotokat egy rekurzív függvény deríti fel, ami egy új MDD-be lemásolja azokat az utakat, amiknek a végén a specifikációs automata elfogadó állapotban van. Ez az algoritmus egyik szűk keresztmetszete, lépésszáma az MDD-ben található utakkal arányos, így nagyon fontos hatékonyan megvalósítani. Ennek érdekében a függvény nagyon erősen hagyatkozik a cache használatára, aminek segítségével kielégítő eredményeket képes felmutatni⁸. A későbbiekben célszerű lehet majd az elfogadó állapotokat „on-the-fly” módon gyűjteni az állapottér felderítése közben.

A létrehozott három állapothalmazt, amelyeket egy-egy MDD csomópont reprezentál, ezután elkezdem körbe-körbe szűkíteni az előtte következővel. Elsőként az élek kiinduló állapotait szűkítem az elfogadó állapotokból elérhetőkre, majd egyetlen tüzeléssel az élek végállapotait szűkítem a kezdőállapotokkal, végül a végállapotokból elérhető állapotokat tartom csak meg az elfogadó állapotokból. Ezt addig ismétlem, amíg egyszer valamelyik halmaz elfogy, vagy nem változik meg a szűkítés során. Az első körben viszont még nem állok le, ha valamelyik halmaz nem változik, mivel ekkor még nem garantált, hogy a változás hiánya ellenére a következő szűkítés nem fog további állapotokat kizárni. Később már elmondható, hogy ha egy állapothalmaz nem változott, akkor a rákövetkező sem fog, és így tovább.

⁶Emiatt az állapottér-MDD tartalmazni fogja az eddig felfedezett állapotokat.

⁷Általános esetben elméletileg semmi akadályja alacsonyabb szintű élet is megkötni a körben, azonban az ahhoz tartozó eseményt azon a szinten is kell elsűtni. A munkámhoz szükséges esetben elegendő lekérni a csomópont szintjéhez tartozó összes eseményt, és azokkal lépni egyet.

⁸Kicsit belegondolva a cache használata azt eredményezi, hogy semmit nem számolunk ki kétszer - ez a cache-ben való keresést leszámítva megegyezik azzal az esettel, amikor felderítés közben gyűjtjük az elfogadó állapotokat.

Ha a ciklus azért ér véget, mert valamelyik halmaz elfogyott, akkor a függvény hamis értéket ad vissza, nem volt a feltételeknek megfelelő kör az állapothalmazban. Ha azért állt le, mert egy halmaz nem változott, akkor pedig igaz értékkel tér vissza, és az elfogadó állapotok halmazának jelenlegi állapotát adja vissza - ezek az állapotok vannak benne vagy érhetőek el⁹ egy körből. Ha a függvény igaz értékkel tér vissza, az algoritmus leáll, az adatokat tartalmazó objektumból pedig kiolvasható az eredmény és az ellenpéldához vezető állapotok. Az ellenpélda előállítására hatékony módszerek adhatók, azonban ez túlmutat a munkám keretein.

Az implementáció és az algoritmus hatékonyságát a következő fejezetekben tárgyalt mérésekkel fogom alátámasztani.

⁹Korábban is utaltam rá, hogy a körben nem szereplő, csak abból elérhető állapotok szükség esetén könnyen eltávolíthatók.

4.

Mérési eredmények

A munkám során elkészített algoritmus újszerűsége miatt nehezen összehasonlítható létező megoldásokkal. Emiatt ez a fejezet az algoritmus teljesítményét és tulajdonságait főképp az alapul vett klasszikus szaturációs algoritmushoz, esetenként a CTL modellellenőrzéshez képest fogja vizsgálni. Megvizsgálom még ún. *fairness kritériumokat* is [7] – ezek azért nagyon érdekesek, mert CTL nyelven nem fogalmazhatóak meg.

A cél mindenképpen az, hogy a klasszikus szaturációs algoritmus teljesítményét minél jobban megközelítsem, még ha ez általános esetben lehetetlen is. Az alábbiakban különböző modellek és specifikációs kifejezések segítségével vizsgálom meg az algoritmus futásának paramétereit, elsősorban a futási időket és a felderített állapotok számát.

A 4.3. fejezetben az algoritmus skálázódását és a 2.4. fejezetben bemutatott optimalizációk hatását is megvizsgálom három paraméterezhető modell segítségével.

A méréseket a következő konfiguráción végeztem: Intel Q8400 2,6 GHz processzor, 4 GB memória, Windows 7 (x64) operációs rendszer, .NET 4.0 futtatókörnyezet. A mért idők az adatszerkezetek inicializálását nem tartalmazzák, ezek ideje minden futás során elhanyagolható mértékű volt.

4.1. Összehasonlító mérések

Elsőként az *F False* kifejezés kiértékelésének paramétereit mértem meg. Az algoritmus ilyenkor a *G True* kifejezést teljesítő lefutásokat keres ellenpéldaként, ami egy tetszőleges kör felderítését jelenti az állapottérben. Ha a kifejezés teljesül, akkor nincsen kör az állapottérben, ellenkező esetben az első találatkor leáll az algoritmus. Hasonló tulajdonság vizsgálható a CTL nyelvű *EG True* kifejezéssel is, ezért a 4.1. táblázatban össze is vetem a két megoldást.

Modell	S	Igaz	CTL	LTL	Felderítés	Körkeresés	Bejárt
DFMS-100	33 028	nem	1,86s	0,001s	57,9%	42,1%	14
laBPEL	26 062	igen	19,31s	3,279s	18,5%	81,5%	26 062
auctionBPEL	21 775	igen	15,88s	0,142s	36,4%	63,6%	21 775
DPhil-100	$4,96 \cdot 10^{62}$	nem	6,15s	0,006s	98,4%	1,6%	13
DPhil-1000	10^{625}	nem	–	0,369s	99,3%	0,7%	13
Hanoi-12	531 441	nem	0,45s	9,897s	97,6%	2,4%	531 441
Kanban-30	$1,35 \cdot 10^{14}$	nem	1,17s	0,001s	83,5%	16,5%	3

4.1. táblázat. A körkeresés vizsgálatának eredményei.

A korábbi, hasonló témájú munkák méréseit megismételve a 4.2. táblázatban további két kifejezésre összehasonlítom a tanszéki CTL modellellenőrzőt és az új LTL modellellenőrző algoritmusomat. A mérésekhez használt LTL kifejezéseket a CTL formulákat negálva

és az A útvonalkvantort leahyva állítom elő¹. Emiatt az algoritmusok által adott válasz pont ellenkező lesz, hiszen a CTL modellellenőrző kielégíthetőséget vizsgál, míg az LTL modellellenőrző a negált kifejezés érvényességét megdöntő ellenpéldát keres.

Modell	S	Kifejezés	Igaz	CTL	LTL	Feld.	Körk.	Bejárt
Hanoi-12	531 441	$G \neg(B_6 > 0)$	nem	31,33s	10,2s	97,3%	2,7%	1771740
Kanban-30	$5,6 \cdot 10^{17}$	$G \neg(P_{out4} = 1)$	nem	19,89s	0,28s	99,2%	0,8%	$2,7 \cdot 10^{10}$

4.2. táblázat. Egyéb, CTL kifejezésként is felírható kifejezések vizsgálatának eredménye

4.2. Fairness kritériumok vizsgálata

Fairness kritériumokat nem lehet CTL nyelven megfogalmazni. Egy fairness kifejezés többnyire $GF a \Rightarrow GF b$ alakú, és azt fejezi ki, hogy ha egy állítás végtelenül gyakran teljesül, akkor egy másik is végtelenül gyakran fog. Az ilyen jellegű specifikációk ellenőrzése algoritmikusan rendkívül költséges feladat. Mint a 4.3. táblázat mérési eredményeiből is látszik, az algoritmusom jelentősen lassabban értékeli ki ezeket a kifejezéseket, mint a korábban vizsgáltakat, ugyanakkor a futási idő a probléma komplexitásához képest még így is megfelelő.

Modell	S	Kifejezés	Igaz	Idő	Feld.	Körk.	Bejárt
DPhil-100	$4,96 \cdot 10^{62}$	$GF (RH_1 = 1) \Rightarrow GF (RH_1 = 1 \wedge LH_1 = 1)$	igen	11,49s	5%	95%	$\sim 10^{63}$
DPhil-100	$4,96 \cdot 10^{62}$	$GF (RH_2 = 0) \Rightarrow GF (RH_1 = 1 \wedge LH_1 = 1)$	nem	0,02s	61,7%	38,3%	853
DPhil-100	$4,96 \cdot 10^{62}$	$GF (LH_2 = 0) \Rightarrow GF (RH_1 = 1 \wedge LH_1 = 1)$	nem	0,02s	59,4%	40,6%	873
FMS-25	$8,5 \cdot 10^{13}$	$GF (P12s = 25) \Rightarrow GF (P12WM3 > 0)$	nem	30,11s	4,4%	95,6%	$2,7 \cdot 10^{14}$
Kanban-30	$5,6 \cdot 10^{17}$	$GF (P1 > 0) \Rightarrow GF (PM1 > 0)$	nem	0,15s	40,4%	59,6%	60 881
Kanban-30	$5,6 \cdot 10^{17}$	$GF (P1 > 0 \wedge P2 > 0 \wedge P3 > 0 \wedge P4 > 0) \Rightarrow GF (PM1 > 0 \vee PM2 > 0 \vee PM3 > 0 \vee PM4 > 0)$	igen	3,09s	29,4%	70,6%	$1,3 \cdot 10^{14}$

4.3. táblázat. Fairness kritériumok vizsgálatának eredményei.

¹Bizonyított, hogy ha egy CTL kifejezésből az útvonalkvantor leahyzásával jelentésében nem ekvivalens LTL kifejezés keletkezik, akkor nem is létezik ilyen [3].

4.3. Skálázódás és optimalizációk

Az alábbiakban paraméterezhető modelleken vizsgálom az algoritmus skálázódását. A vizsgált modellek közül az étkező filozófusok és a réselt gyűrű modellek strukturálisan paraméterezhetők, ami azt jelenti, hogy ismétlődő részek számát határozhatjuk meg. A Kanban modellben ezzel szemben a kezdeti tokeneloszlásban megjelenő tokenek száma adja a paramétert.

A paraméter értéke szerint megvizsgálom a modellellenőrzés futási idejét, külön vizsgálva ezen belül a körkeresésre fordított időt. Minden modell esetében egy-egy invariáns jellegű, érvényes és érvénytelen kifejezéssel végeztem a méréseket. A nem teljesülő kifejezés esetében úgy építettem fel az atomi kijelentést, hogy annak vizsgálatához legalább a modell állapotterének felét fel kelljen deríteni.

Ezekben a mérésekben vizsgálom meg a 2.4. fejezetben bemutatott optimalizációk hatását is, köztük is elsősorban a 2.4.2. szakaszban tárgyalt saját javításom hatását. Az érvényes kifejezéseket úgy alkottam meg, hogy a körkereső algoritmus keveset vagy egyáltalán ne fusson, mivel így látható, hogy a vezérelt szaturációval történő állapotter-felderítés hogyan teljesít a szorzat állapotterben². Arra számítottam, hogy az optimalizációk elsősorban a körkeresést fogják befolyásolni³, így hatásukat csak a nem teljesülő kifejezés kiértékelésére vizsgáltam meg.

Ennek keretében megmértem a körkereső algoritmus által elvégzett szűkítő lépések számát összesítve, valamint átlagolva a verifikáció alatt megkezdett összes körkeresésre. Megvizsgálom még az egyes esetekben a körkeresés teljes idejében bekövetkezett javulást is, hiszen az optimalizációk elsődleges célja ennek az időnek a csökkentése. Látható lesz, hogy ez nincs egyértelmű kapcsolatban a másik két mért értékkel, ugyanis egy körkeresés időtartama nagyban függ olyan nehezen mérhető tulajdonságoktól is, mint a szűkítendő halmazok mérete, illetve az őket leíró MDD-k struktúrája.

Az elvégzett mérések során a kifejezések minden esetben egy 5 állapotból álló, 2 elfogadó állapotot tartalmazó automatára fordultak le. Ezt az optimalizációk 4 állapotra és 1 elfogadó állapotra csökkentették, ennek hatása lesz megfigyelhető az alábbiakban. További méréseket igényelne az optimalizáció mértékének mérése különböző automatákkal, azonban ezt jelen dolgozatban nem tekintettem célomnak. Az elfogadó állapotok számát a redukció a legtöbb esetben megfelezi, ennek hatása pedig nagy vonalakban látható lesz az elvégzett mérésekből is.

²Vezérelt szaturációt jellemzően előre felderített állapotterén alkalmaznak.

³A mérések során ez be is bizonyosodott, ám terjedelmi okokból és kevés információtartalmára való tekintettel az erre utaló adatokat nem mutatom be.

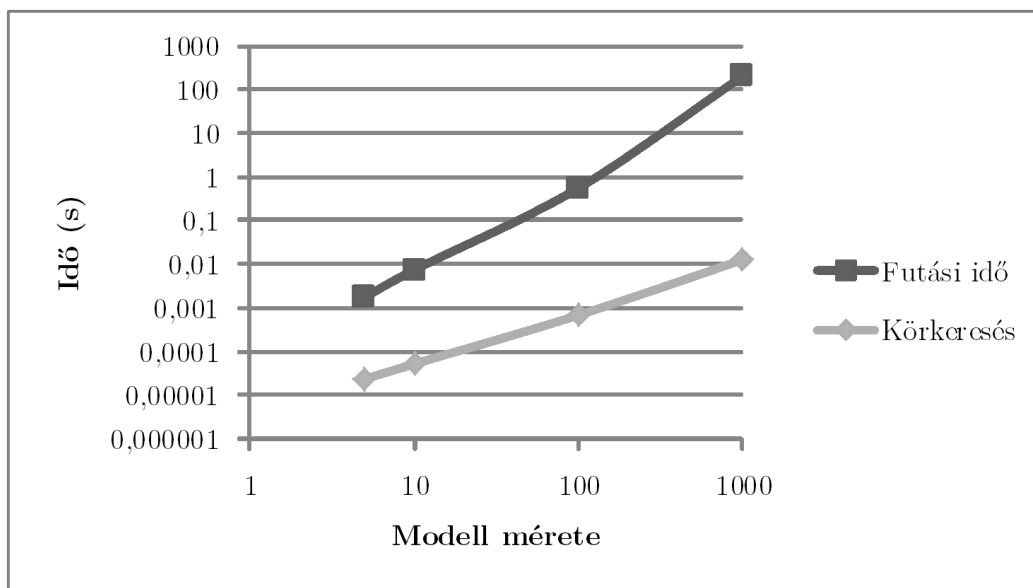
4.3.1. Étkező filozófusok

DPhil	Futási idő	Körkeresés
5	0,0017808 s	0,0000237 s
10	0,0075077 s	0,0000522 s
100	0,5578516 s	0,0006842 s
1000	207,2472739 s	0,0128278 s

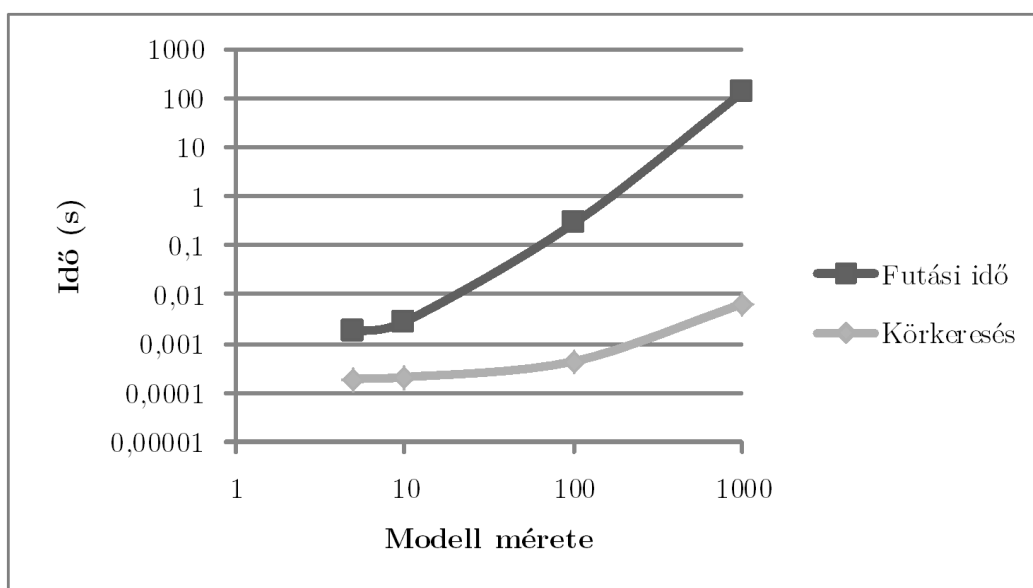
4.1. ábra. Teljes és körkereséssel töltött futási idő IGAZ kifejezés esetén.

DPhil	Futási idő	Körkeresés
5	0,0017657 s	0,0001925 s
10	0,0028032 s	0,0002077 s
100	0,2803778 s	0,0004371 s
1000	132,7993734 s	0,0065369 s

4.2. ábra. Teljes és körkereséssel töltött futási idő HAMIS kifejezés esetén.



4.3. ábra. Teljes és körkereséssel töltött futási idő IGAZ kifejezés esetén.



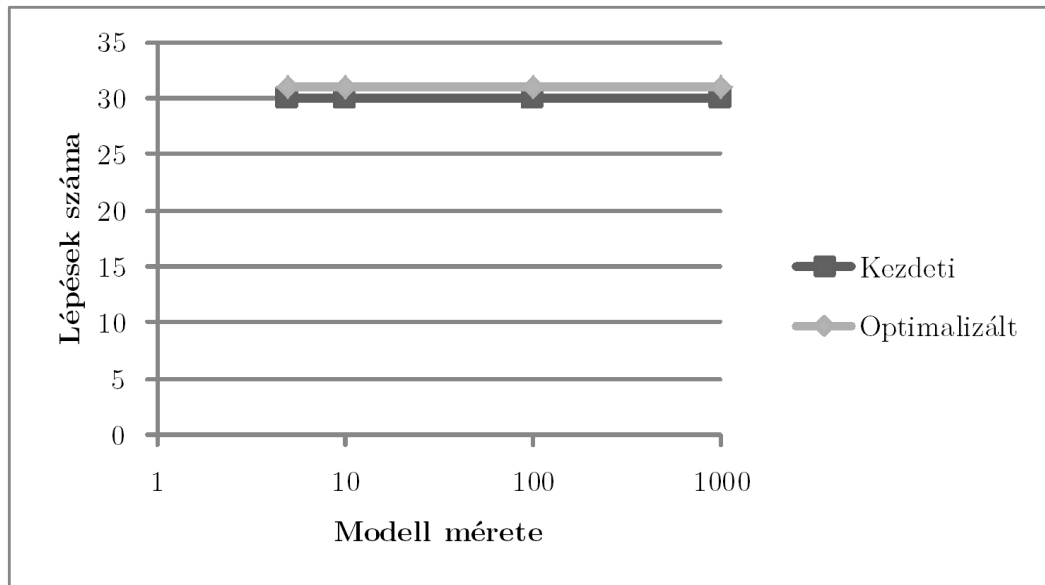
4.4. ábra. Teljes és körkereséssel töltött futási idő HAMIS kifejezés esetén.

DPhil	Kezdeti	Optimalizált
5	30	31
10	30	31
100	30	31
1000	30	31

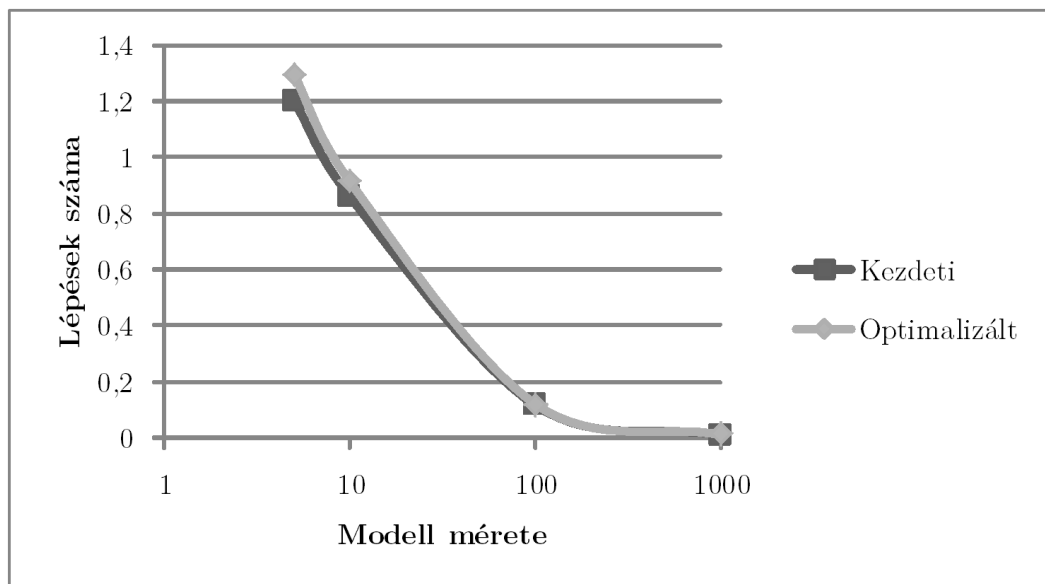
4.5. ábra. Az összes megtett szűkítő lépés az automata optimalizálásával és anélkül.

DPhil	Kezdeti	Optimalizált
5	1,2	1,291666667
10	0,857142857	0,911764706
100	0,115384615	0,11969112
1000	0,011952191	0,01235552

4.6. ábra. Körkeresésként átlagosan megtett szűkítő lépések az automata optimalizálásával és anélkül.



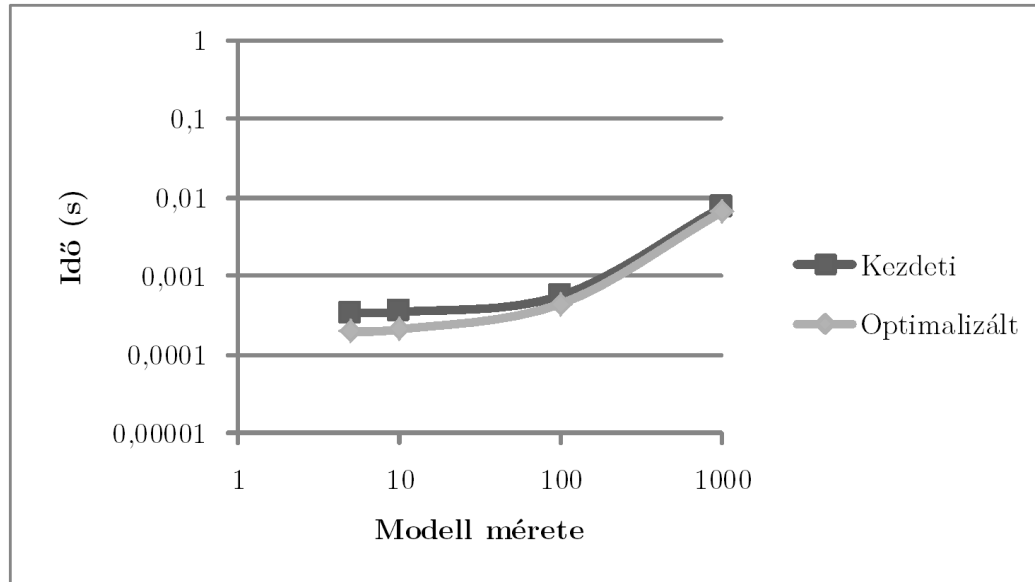
4.7. ábra. Az összes megtett szűkítő lépés az automata optimalizálásával és anélkül.



4.8. ábra. Körkeresésként átlagosan megtett szűkítő lépések az automata optimalizálásával és anélkül.

DPhil	Kezdeti	Optimalizált	Javulás
5	0,0003363 s	0,0001925 s	42,76%
10	0,0003462 s	0,0002077 s	40,01%
100	0,0005544 s	0,0004371 s	21,16%
1000	0,0074558 s	0,0065369 s	12,32%

4.9. ábra. A körkeresésre fordított idő az automata optimalizálásával és anélkül.



4.10. ábra. A körkeresésre fordított idő az automata optimalizálásával és anélkül.

Az étkező filozófusok modellre az algoritmus láthatóan jól skálázódik. A körkeresés ennél a modellnél egyik kifejezés esetén sem jelent szűk keresztmetszetet, az idő többségében a szorzat állapottér felderítése zajlik. A körkeresés szűkítő lépéseinek száma a vizsgált esetben nem függ a modell méretétől.

A 4.5. és 4.6. ábrákon érdekesnek tűnhet, hogy a szűkítő lépések száma az optimalizáció hatására kismértékben nőtt, míg a 4.9. táblázat szerint a körkeresésre fordított idő csökkent. Erre az adhat magyarázatot, hogy az automata szerkezete megváltozott az optimalizáció során, és eddig külön MDD részfákban tárolt állapottér-részletek az összevont két automataállapot által immár együtt kezelhetők. Az is látható, hogy az optimalizációval megtakarított idő nagyjából konstans, ami annak fényében nem meglepő, hogy a szűkítő lépések száma is az. A körkeresés időtartamának skálázódó részét a függvényhívások többletideje és az elfogadó állapotok halmazának kiszámítása adja.

Összességében a filozófus modelltől kijelenthető, hogy a körkeresésre invariáns kifejezések esetén jól skálázódik, azonban a vezérelt szaturációs állapottér-felderítés valamivel rosszabbul teljesít, mint a korábbi szaturációs megoldások teljesítménye alapján várható volt. Az optimalizációk viszonylag nagy mértékben javítják a körkeresés futási idejét.

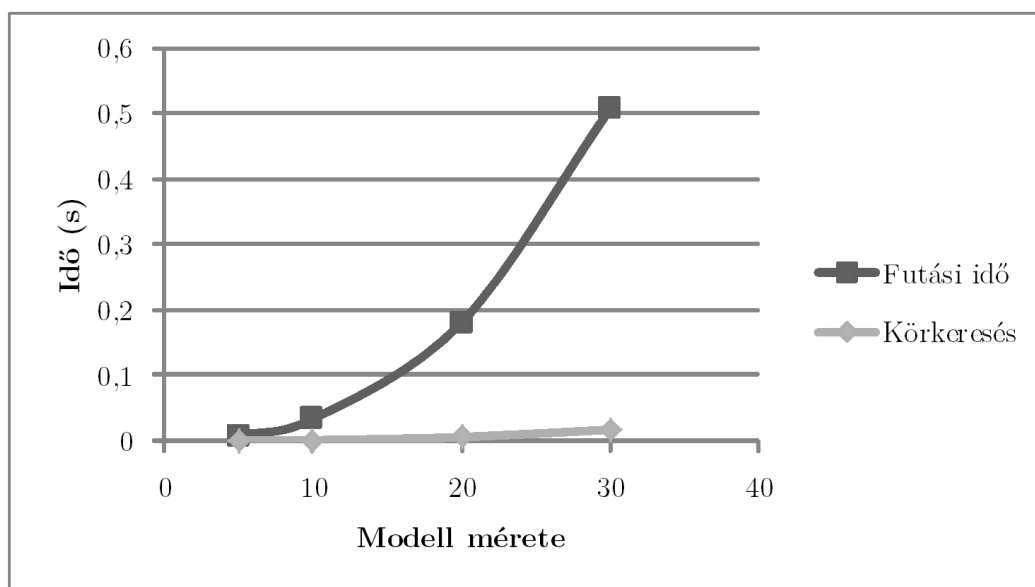
4.3.2. Réselt gyűrű

SR	Futási idő	Körkeresés
5	0,0082532 s	0,0001864 s
10	0,0337258 s	0,000969 s
20	0,1807162 s	0,0057285 s
30	0,5071189 s	0,0171788 s

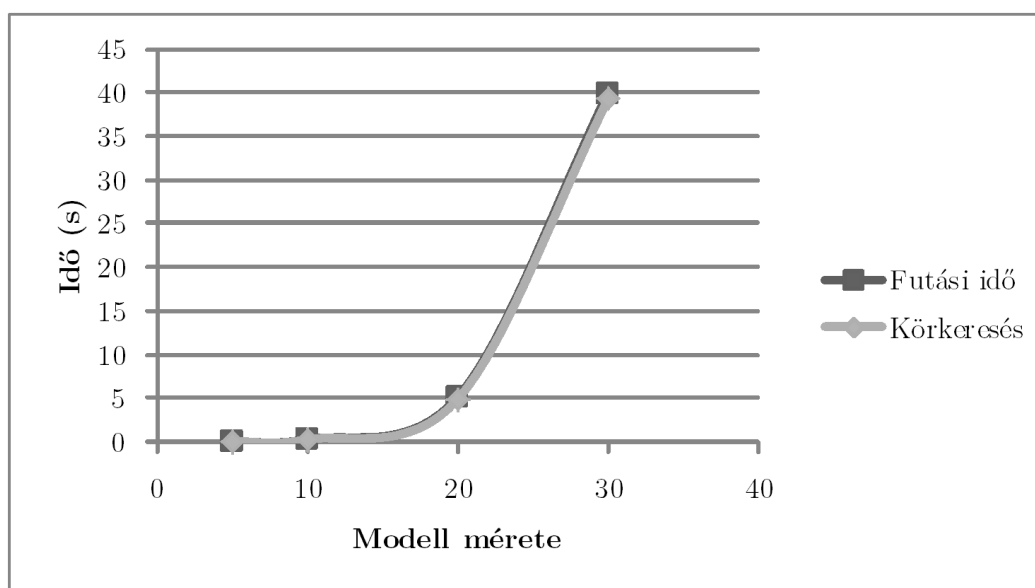
4.11. ábra. Teljes és körkereséssel töltött futási idő IGAZ kifejezés esetén.

SR	Futási idő	Körkeresés
5	0,0368103 s	0,0262097 s
10	0,3257244 s	0,2836643 s
20	5,0993482 s	4,8608076 s
30	39,8827556 s	39,200519 s

4.12. ábra. Teljes és körkereséssel töltött futási idő HAMIS kifejezés esetén.



4.13. ábra. Teljes és körkereséssel töltött futási idő IGAZ kifejezés esetén.



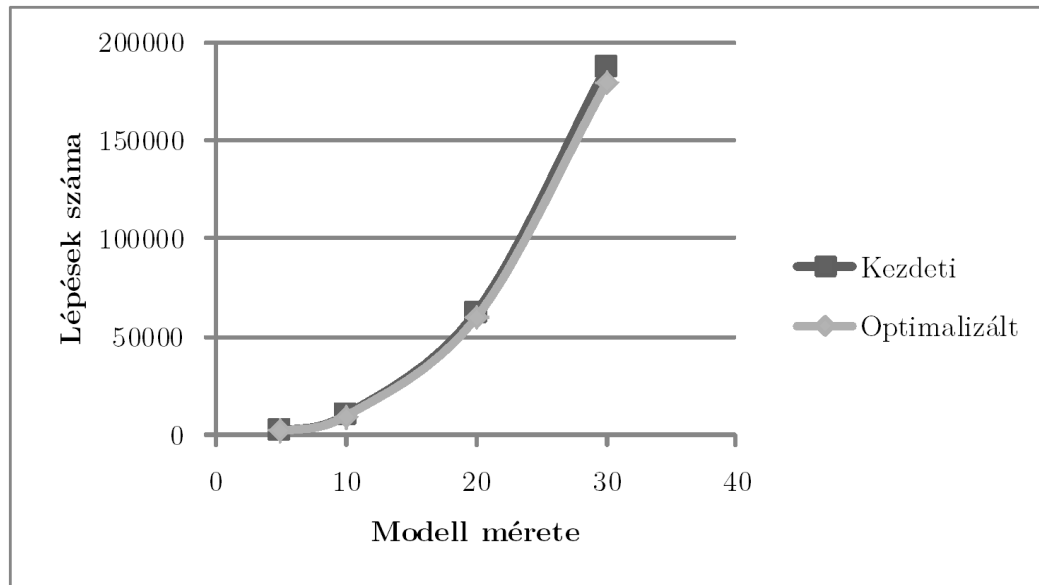
4.14. ábra. Teljes és körkereséssel töltött futási idő HAMIS kifejezés esetén.

SR	Kezdeti	Optimalizált
3	1885	1790
10	10232	9593
15	62465	59478
30	187071	178818

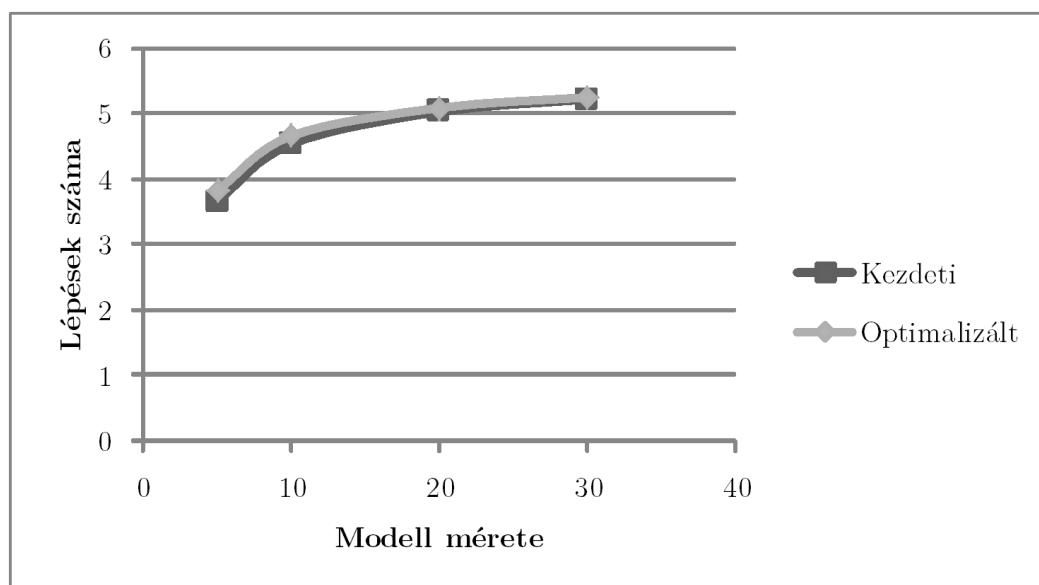
4.15. ábra. Az összes megtett szűkítő lépés az automata optimalizálásával és anélkül.

SR	Kezdeti	Optimalizált
3	3,667315175	3,824786325
10	4,533451484	4,659057795
15	5,025746239	5,084893562
30	5,214231959	5,247161008

4.16. ábra. Körkeresésként átlagosan megtett szűkítő lépések az automata optimalizálásával és anélkül.



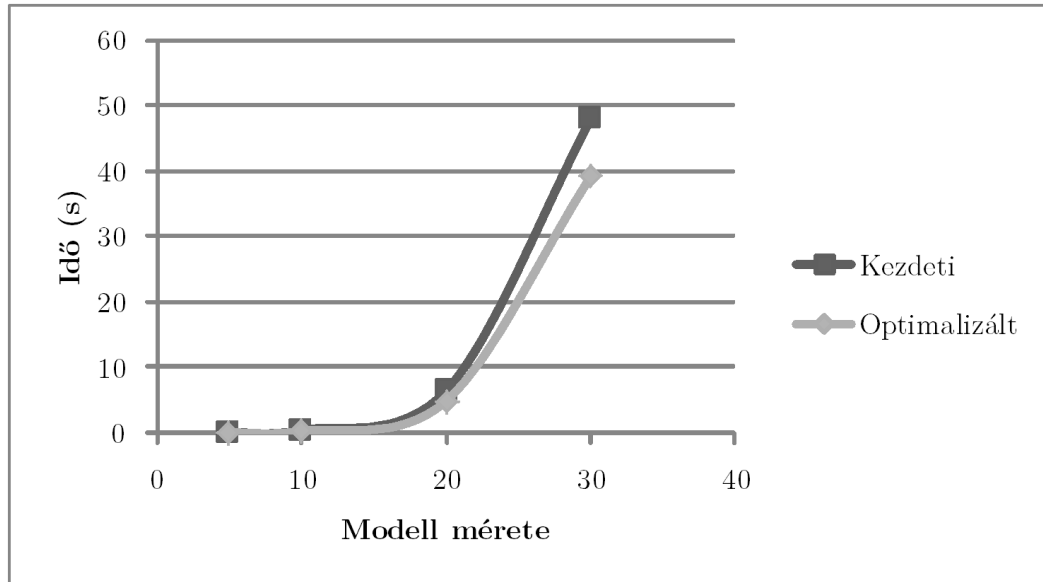
4.17. ábra. Az összes megtett szűkítő lépés az automata optimalizálásával és anélkül.



4.18. ábra. Körkeresésként átlagosan megtett szűkítő lépések az automata optimalizálásával és anélkül.

SR	Kezdeti	Optimalizált	Javulás
5	0,0468261 s	0,0262097 s	44,03%
10	0,4550905 s	0,2836643 s	37,67%
20	6,441455 s	4,8608076 s	24,54%
30	47,8976282 s	39,200519 s	18,16%

4.19. ábra. A körkeresésre fordított idő az automata optimalizálásával és anélkül.



4.20. ábra. A körkeresésre fordított idő az automata optimalizálásával és anélkül.

A réselt gyűrű modellre az algoritmus látványosan rosszabbul skálázódik. Az érvényes kifejezés esetében nem indul körkeresés, azonban az érvénytelen formula ellenpéldájának keresése közben egyértelműen ez a művelet dominálja a futási időt. Ez azért lehet, mert a réselt gyűrűben megtalált nagyméretű kör sűrűn tartalmaz elfogadó állapotokat, amelyek nagy körmentes irányított gráfokban is szerepelnek, így nagyon sok szűkítő lépést eredményeznek a körkereső fázisokban.

A 4.15. és 4.16. ábrákon is az látszik, hogy a modell méretének növelésével egyre több szűkítő lépést kell elvégeznie az algoritmusnak. Érdekesség, hogy az átlagos lépésszám növekedésének üteme lassul. A réselt gyűrű modell az előzetes mérések során is többször mutatott különleges viselkedést és rossz skálázódást, így a későbbiekben érdemes lesz további vizsgálatokat folytatni a modell és az algoritmus működésének jobb megértése és az ebből adódó esetleges továbbfejlesztése érdekében.

A filozófus modellhez hasonlóan a 4.19. ábrán itt is megfigyelhető, hogy az automata optimalizációk okozta strukturális változása a szűkítő lépések átlagos számát kis mértékben növelte, azonban ennél a modellnél az összesített lépésszám csökkent. A körkeresésre fordított idő is jelentősen csökkent az automata redukciója folytán, és a javulás mértéke is jobbnak látszik, mint a filozófus modellnél.

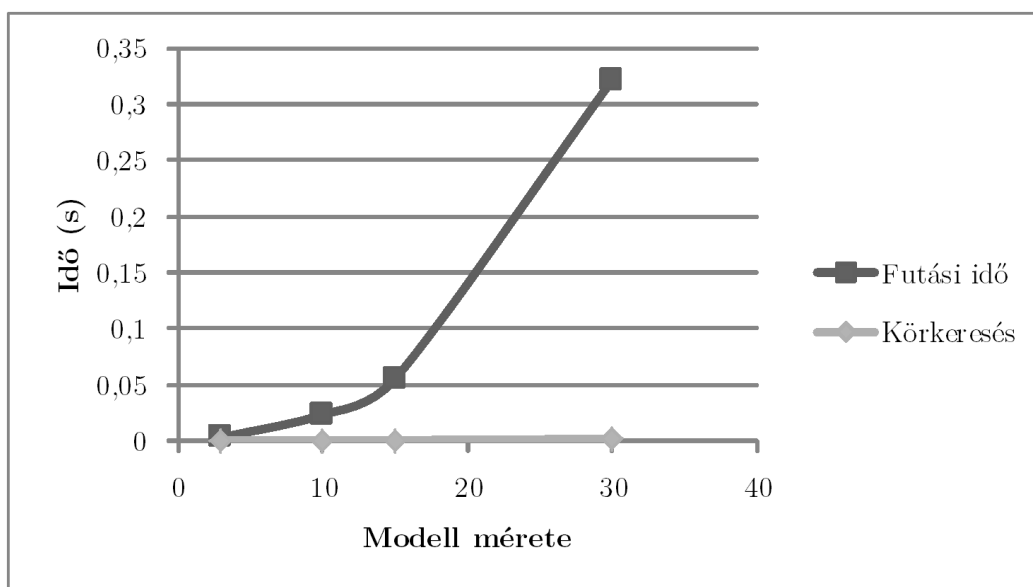
4.3.3. Kanban

Kanban	Futási idő	Körkeresés
3	0,0030495 s	0,0000651 s
10	0,02343 s	0,0002669 s
15	0,0556491 s	0,000479 s
30	0,3218458 s	0,0016359 s

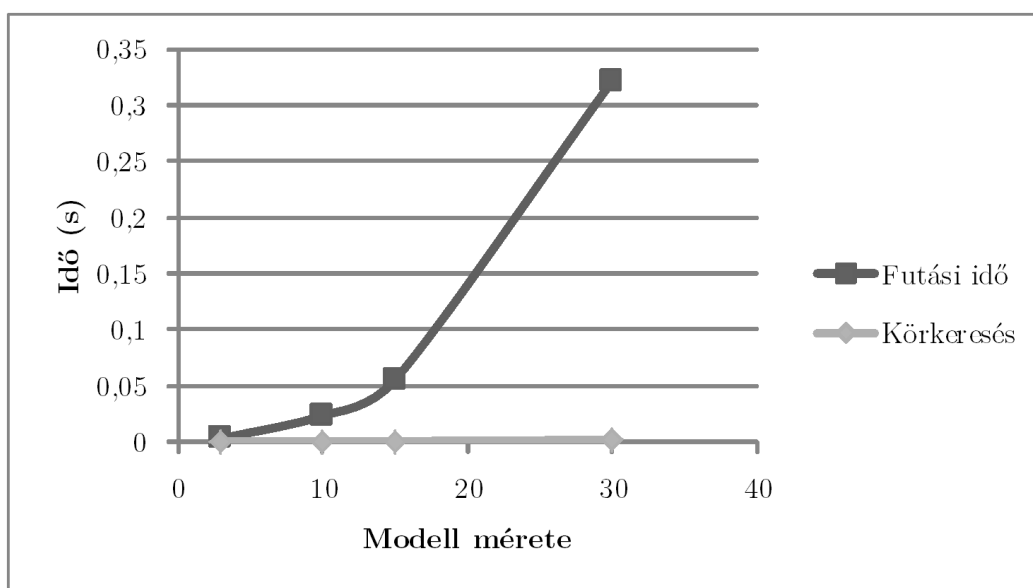
4.21. ábra. Teljes és körkereséssel töltött futási idő IGAZ kifejezés esetén.

Kanban	Futási idő	Körkeresés
3	0,0032279 s	0,0001125 s
10	0,0284097 s	0,0003823 s
15	0,0620972 s	0,0006341 s
30	0,3215266 s	0,0039584 s

4.22. ábra. Teljes és körkereséssel töltött futási idő HAMIS kifejezés esetén.



4.23. ábra. Teljes és körkereséssel töltött futási idő IGAZ kifejezés esetén.



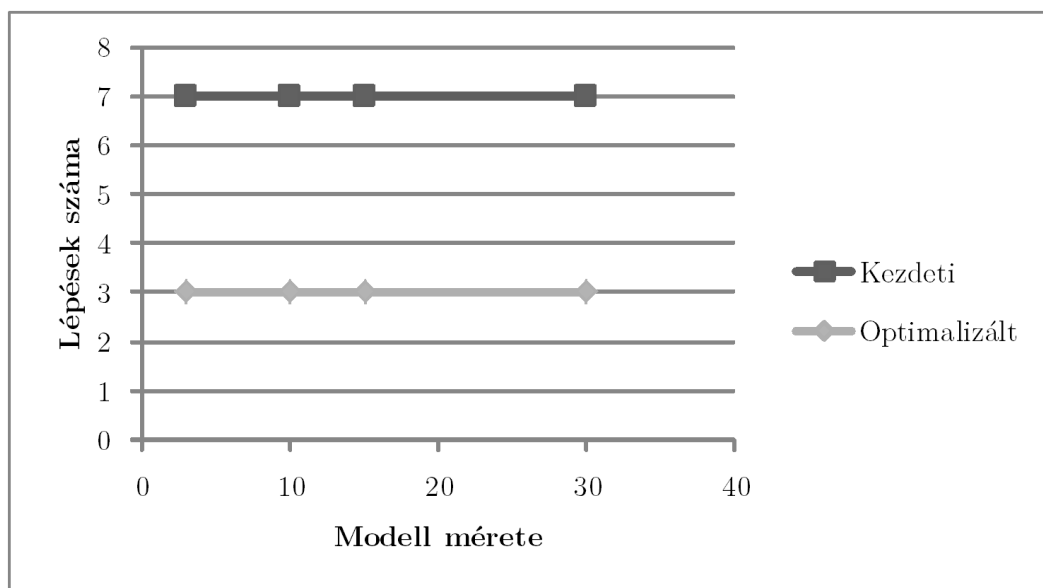
4.24. ábra. Teljes és körkereséssel töltött futási idő HAMIS kifejezés esetén.

Kanban	Kezdeti	Optimalizált
3	7	3
10	7	3
15	7	3
30	7	3

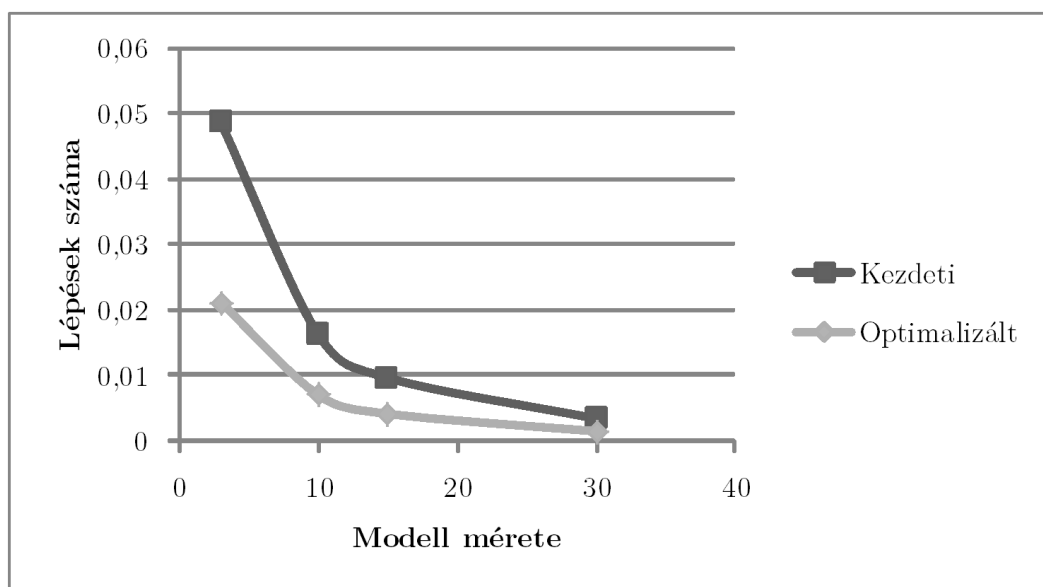
4.25. ábra. Az összes megtett szűkítő lépés az automata optimalizálásával és anélkül.

Kanban	Kezdeti	Optimalizált
3	0,04861111	0,021126761
10	0,01624129	0,006993007
15	0,00964187	0,004143646
30	0,00339641	0,001457018

4.26. ábra. Körkeresésként átlagosan megtett szűkítő lépések az automata optimalizálásával és anélkül.



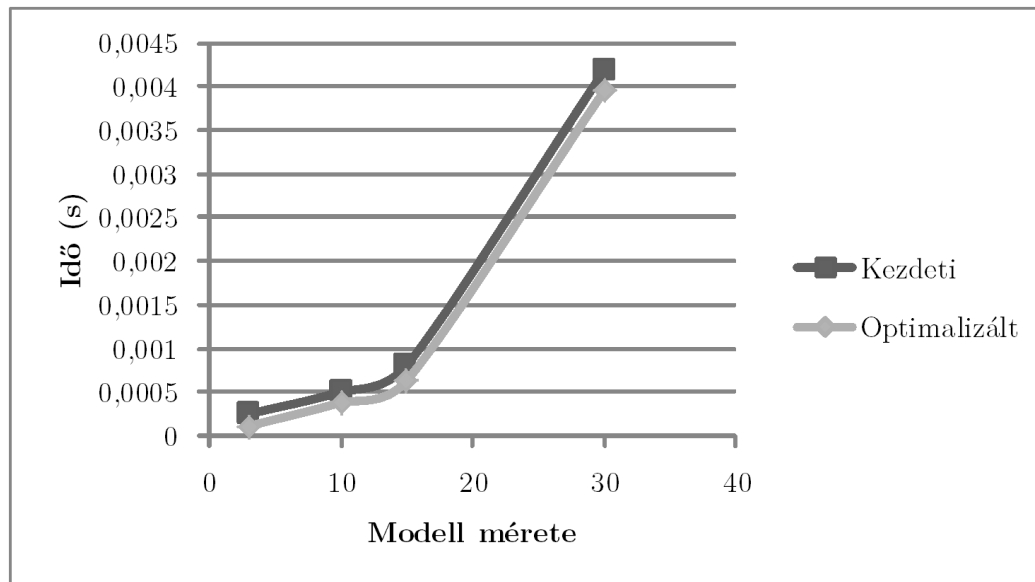
4.27. ábra. Az összes megtett szűkítő lépés az automata optimalizálásával és anélkül.



4.28. ábra. Körkeresésként átlagosan megtett szűkítő lépések az automata optimalizálásával és anélkül.

Kanban	Kezdeti	Optimalizált	Javulás
3	0,00025125 s	0,0001125 s	55,22%
10	0,0005047 s	0,0003823 s	24,25%
15	0,0008089 s	0,0006341 s	21,61%
30	0,0041912 s	0,0039584 s	5,55%

4.29. ábra. A körkeresésre fordított idő az automata optimalizálásával és anélkül.



4.30. ábra. A körkeresésre fordított idő az automata optimalizálásával és anélkül.

A Kanban modell ezen kifejezésekre a filozófus modellhez hasonlóan nem érzékeny a körkeresésre. Látható, hogy a futási időt az állapotter-generálás tölti ki, felesleges körkeresések valószínűleg nem, vagy csak kis számban indulnak.

Ezt láthatjuk a 4.25. és a 4.26. ábrákon is. Az összes szűkítő lépések száma ismét konstans, azonban az automata optimalizálása sokkal többet javít rajta, mint a filozófus modell esetén. Ezúttal az átlagos lépések száma is csökken, ami valószínűleg annak köszönhető, hogy a modell struktúrája helyett a kezdeti tokeneloszlás skálázása másképp változtatja az állapotteret kódoló MDD szerkezetét. A körkeresés futásának ideje ismét egy konstans értékkel csökkent az automata redukálásának hatására.

4.3.4. Értékelés

A vizsgált modellek példáján látható, hogy az algoritmus egészen másképp tud viselkedni különböző modellek esetén. Ez várható volt, mivel – mint a 2.1.1. fejezetben is kiemeltem – egy PSPACE-teljes probléma esetén semmilyen megoldástól nem várható el, hogy minden esetben jól teljesítsen.

A mérések alapján a továbbiakban célszerű lesz a réselt gyűrű példáján további méréseket és optimalizálásokat végezni, mivel a vizsgált modellek közül egyértelműen ezen teljesített legrosszabbul az algoritmusom. A többi modellel kapcsolatban is érdemes lehet további kísérleteket végezni, elsősorban más jellegű kifejezésekkel.

A további méréseket és elemzéseket az is indokolja, hogy a most összegyűjtött adatoknak is csak egy kis hányadát volt lehetőségem itt bemutatni, és számtalan egyéb tulajdonság vizsgálható. Mindezek alapos kiértékelése fontos fejlesztési lehetőségekre világíthat rá, ami egy eddig ismeretlen terület további megértését és felderítését segítené elő.

Összességében az algoritmusom a vizsgált esetekre a várakozásaimhoz képest jól teljesített. A vizsgált modellek felépítése és állapottere változatos méretű volt, az 1000 étkező filozófus modellje például 10^{625} állapottal rendelkezik, a réselt gyűrű pedig bonyolult struktúrával rendelkezik, ehhez képest a teljesítmény a legtöbb esetben elfogadható kereteken belül maradt. A szaturációs algoritmus általam használt verziója hasonlóan jó tulajdonságokat mutat, mint a klasszikus algoritmus, a szorzat állapottér felderítése és az „on-the-fly” körkeresés nem befolyásolja nagyon a működését.

5.

Összefoglalás

5.1. Elért eredmények

Munkám során az volt a célom, hogy egy hatékony, az eddigi megközelítések előnyeit a lehető legjobban ötvöző LTL modellellenőrzőt hozzak létre. Az elért elméleti eredmények mellett az is komoly motivációt jelentett, hogy a tanszéki modellező eszközt, a *PetriDotNet* keretrendszert [18] felruházzam LTL modellellenőrző képességekkel is.

5.1.1. Elméleti eredmények

Mint a mérések is mutatják, az algoritmusom hatékonysága a legtöbb esetben várakozáson felül teljesít. Mindez annak köszönhető, hogy az állapottér szimbolikus kódolása mellett ki tudom használni a szaturációs algoritmus kiváló teljesítményét aszinkron rendszerek esetén, illetve az „on-the-fly” modellellenőrzés miatt a nem érvényes állítások nagyon hamar leállítják az állapottér-generálást.

Legfontosabb elméleti eredményem tehát az LTL modellellenőrzés szaturációs alapokon történő megvalósítása, kombinálva egy szintén szaturációs alapú, „on-the-fly” kördetektáló algoritmussal. Tudomásom szerint ezek külön-külön sem léteztek eddig, kombinálásukkal azonban egy teljesen új algoritmust hoztam létre, amely remélhetőleg elősegíti majd az LTL modellellenőrzők ipari felhasználhatóságának növekedését.

5.1.2. Gyakorlati eredmények

Az elméleti eredményeket a gyakorlatban is megvalósítottam, új modellellenőrző funkcióval bővítve a *PetriDotNet* eddig is kiváló lehetőségeit. Ezzel a keretrendszer immár LTL kifejezések teljesülését is képes ellenőrizni, ami a két formalizmus nem összehasonlítható volta miatt kiterjeszti a vizsgálható problémák körét.

Az implementációt emellett úgy készítettem el, hogy többféle típusú modellel lehessen minimális módosításokkal alkalmazni. A kód így például könnyen kiterjeszthető színezett Petri-hálókra is.

5.2. Továbbfejlesztési lehetőségek

A megoldás újdonsága miatt rengeteg fejlesztési lehetőség áll előttem. Az alábbiakban bemutatok néhányat, bonyolultságuk szerinti sorrendben:

- A *PetriDotNet2* keretrendszer elkészültekor migrálni fogjuk a szaturációs és modellellenőrző algoritmusokat tartalmazó csomagot, ami lehetőséget ad majd az algoritmusok és adatszerkezeteik általánosítására. Ezt jelen algoritmus tükrében megkí-

sérlem majd úgy megoldani, hogy az LTL modellellenőrzés tökéletesen függetlenül működhessen bármilyen típusú modellen, akár időzített hálókon is. Ezen kívül az algoritmus működését konfigurálhatóvá, kezelését felhasználóbarátabbá teszem.

- A szaturációs algoritmus használata miatt számos redukciós technika alkalmazható lehet az állapottér-generálás hatékonyságának növelésére.
- A kördetektáló algoritmusom által visszaadott információk alapján elő fogok állítani egy-egy ellenpéldát azokban az esetekben, amikor a modell nem teljesíti a specifikációt.
- A meglévő CTL modellellenőrök, vagy egy esetlegesen kifejlesztésre kerülő „on-the-fly” CTL modellellenőrő és jelen munkám kombinálásával akár egy szaturációs alapú CTL* modellellenőrő is megvalósulhat.

Ezekon kívül az algoritmus működésének részleteit és további fejlesztési lehetőségeket kutatva további, átfogó méréseket fogok végezni a lehető legtöbb modell és kifejezés felhasználásával.

Köszönetnyilvánítás

Köszönöm a családom türelmét és támogatását, köztük is főleg édesanyámét. Köszönöm dr. Bartha Tamásnak, hogy szakértelmével hozzájárult munkám elkészítéséhez, illetve Jámbor Attilának és különösen Darvas Dánielnek, hogy tapasztalatukkal segítettek és korábbi munkáikkal megalapozták jelen dolgozat létrejöttét.

Köszönet illeti meg jó barátaimat, köztük is főképp Klenik Attilát, aki néha munkámat is átvállalva segített a nehéz időszakokban, illetve Kotán Zsófiát, aki kedves szavaival és töretlen támogatásával mindvégig tartotta bennem a lelket.

Végül, de nem utolsósorban köszönöm a Hibatűrő Rendszerek Kutatócsoport tagjainak, hogy köztük dolgozhattam, és megosztották velem munkahelyüket, barátságos mosollyal biztatva minden találkozáskor.

Ábrák jegyzéke

1.1.	Az LTL temporális operátorok szemléletes jelentése.	14
1.2.	Egy egyszerű véges automata.	15
1.3.	Két Büchi automata és szinkron szorzatuk [7]. A szorzat automatában csak az elérhető állapotok szerepelnek.	17
1.4.	Egy Büchi automata.	18
1.5.	Az LTL temporális operátoraiból generált általánosított Büchi automaták. Az éleken proposíciók konjunkciói láthatók.	21
1.6.	Egy többértékű döntési diagram.	21
2.1.	LTL modellellenőrzés automatákkal.	25
2.2.	Egy szorzat állapotteret kódoló MDD.	29
2.3.	Egy szinkron eseményt kódoló MDD.	30
2.4.	Egy predikátum kényszer kódoló MDD.	32
2.5.	Egy elfogadó lefutás két szakasza.	38
2.6.	Egy szaturált csomópont állapottere a legfelső szintű eseményekkel és az elfogadó állapotokkal.	41
2.7.	Elfogadó erősen összefüggő komponensek inkrementális keresése.	41
2.8.	A szaturáció alapú LTL modellellenőrző lépései.	47
3.1.	Az MDDAnalysisComponent.dll felépítése, szürkével jelölve az új részeket.	50
3.2.	Meglévő kódok osztálydiagramja.	54
3.3.	Az új és a meglévő osztályok viszonya. Bekeretezve láthatók a korábbi működést biztosító részek.	55
4.1.	Teljes és körkereséssel töltött futási idő IGAZ kifejezés esetén.	64
4.2.	Teljes és körkereséssel töltött futási idő HAMIS kifejezés esetén.	64
4.3.	Teljes és körkereséssel töltött futási idő IGAZ kifejezés esetén.	64
4.4.	Teljes és körkereséssel töltött futási idő HAMIS kifejezés esetén.	64
4.5.	Az összes megtett szűkítő lépés az automata optimalizálásával és anélkül.	65
4.6.	Körkeresésenként átlagosan megtett szűkítő lépések az automata optimalizálásával és anélkül.	65
4.7.	Az összes megtett szűkítő lépés az automata optimalizálásával és anélkül.	65
4.8.	Körkeresésenként átlagosan megtett szűkítő lépések az automata optimalizálásával és anélkül.	65
4.9.	A körkeresésre fordított idő az automata optimalizálásával és anélkül.	66
4.10.	A körkeresésre fordított idő az automata optimalizálásával és anélkül.	66
4.11.	Teljes és körkereséssel töltött futási idő IGAZ kifejezés esetén.	67
4.12.	Teljes és körkereséssel töltött futási idő HAMIS kifejezés esetén.	67
4.13.	Teljes és körkereséssel töltött futási idő IGAZ kifejezés esetén.	67
4.14.	Teljes és körkereséssel töltött futási idő HAMIS kifejezés esetén.	67
4.15.	Az összes megtett szűkítő lépés az automata optimalizálásával és anélkül.	68

4.16. Körkeresésenként átlagosan megtett szűkítő lépések az automata optimalizálásával és anélkül.	68
4.17. Az összes megtett szűkítő lépés az automata optimalizálásával és anélkül. . .	68
4.18. Körkeresésenként átlagosan megtett szűkítő lépések az automata optimalizálásával és anélkül.	68
4.19. A körkeresésre fordított idő az automata optimalizálásával és anélkül. . . .	69
4.20. A körkeresésre fordított idő az automata optimalizálásával és anélkül. . . .	69
4.21. Teljes és körkereséssel töltött futási idő IGAZ kifejezés esetén.	70
4.22. Teljes és körkereséssel töltött futási idő HAMIS kifejezés esetén.	70
4.23. Teljes és körkereséssel töltött futási idő IGAZ kifejezés esetén.	70
4.24. Teljes és körkereséssel töltött futási idő HAMIS kifejezés esetén.	70
4.25. Az összes megtett szűkítő lépés az automata optimalizálásával és anélkül. .	71
4.26. Körkeresésenként átlagosan megtett szűkítő lépések az automata optimalizálásával és anélkül.	71
4.27. Az összes megtett szűkítő lépés az automata optimalizálásával és anélkül. .	71
4.28. Körkeresésenként átlagosan megtett szűkítő lépések az automata optimalizálásával és anélkül.	71
4.29. A körkeresésre fordított idő az automata optimalizálásával és anélkül. . . .	72
4.30. A körkeresésre fordított idő az automata optimalizálásával és anélkül. . . .	72

Irodalomjegyzék

- [1] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.
- [2] 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.
- [3] Edmund M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 428–437, London, UK, UK, 1989. Springer-Verlag.
- [4] Darvas Dániel. Szaturáció alapú automatikus modellellenőrző fejlesztése aszinkron rendszerekhez. *Tudományos Diákköri Konferencia, Budapesti Műszaki és Gazdaságtudományi Egyetem, Villamosmérnöki és Informatikai Kar*, 2010.
- [5] Darvas Dániel, Jámbor Attila. Komplex rendszerek modellezése és verifikációja. *Tudományos Diákköri Konferencia, Budapesti Műszaki és Gazdaságtudományi Egyetem, Villamosmérnöki és Informatikai Kar*, 2011.
- [6] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.
- [7] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2001.
- [8] Kousha Etessami and Gerard J. Holzmann. Optimizing büchi automata. pages 153–167. Springer, 2000.
- [9] Rob Gerth, Doron Peled, Moshe Y. Vardi, R. Gerth, Den Dolech Eindhoven, D. Peled, M. Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *In Protocol Specification Testing and Verification*, pages 3–18. Chapman Hall, 1995.
- [10] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.
- [11] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
- [12] Roberto Sebastiani, Stefano Tonetta, and Moshe Y. Vardi. Symbolic systems, explicit properties: on hybrid approaches for ltl symbolic model checking. In *PROC. COMPUTER AIDED VERIFICATION*, pages 350–363. Springer, 2005.

- [13] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
- [14] Moshe Vardi. Branching vs. linear time: Final showdown. In *Proceedings of the 2001 Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001 (LNCS Volume 2031)*, pages 1–22. Springer-Verlag, 2001.
- [15] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.
- [16] 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.
- [17] 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.
- [18] A *PetriDotNet* keretrendszer honlapja. (elérve: 2012. október 26.)
<http://petridotnet.inf.mit.bme.hu/>.

Függelék

F.1. Gerth, Peled, Vardi és Wolper algoritmus LTL transzformációs algoritmus

Algoritmus 6 LTL transzformációs algoritmus

```
1: function CREATEGRAPH( $\varphi$ )
2:   return EXPAND( $[ID \leftarrow \text{NEWID}(),$ 
3:      $Incoming \leftarrow \{init\},$ 
4:      $Old \leftarrow \emptyset,$ 
5:      $New \leftarrow \{\varphi\},$ 
6:      $Next \leftarrow \emptyset], \emptyset);$ 
7: end function
8: function EXPAND( $q, Nodes$ )
9:   if  $New(q) = \emptyset$  then
10:    if  $\exists r \in Nodes : Old(r) = Old(q) \wedge Next(r) = Next(q)$  then
11:       $Incoming(r) \leftarrow Incoming(r) \cup Incoming(q);$ 
12:      return  $Nodes;$ 
13:    else
14:      return EXPAND( $[ID \leftarrow \text{NEWID}(),$ 
15:         $Incoming \leftarrow \{ID(q)\},$ 
16:         $Old \leftarrow \emptyset,$ 
17:         $New \leftarrow Next(q),$ 
18:         $Next \leftarrow \emptyset], Nodes \cup \{q\});$ 
19:    end if
20:  else
21:     $\eta \leftarrow \eta \in New(q);$ 
22:     $New(q) \leftarrow New(q) \setminus \{\eta\};$ 
23:    if  $\eta \in Old(q)$  then
24:      return EXPAND( $q, Nodes);$ 
25:    else
26:      if  $\eta$  propozíció  $\vee \eta$  negált propozíció  $\vee \eta = True \vee \eta = False$  then
27:        if  $\eta = False \vee \neg\eta \in Old(q)$  then
28:          return  $Nodes;$ 
29:        else
30:           $q' \leftarrow [ID \leftarrow \text{NEWID}(),$ 
31:             $Incoming \leftarrow Incoming(q),$ 
32:             $Old \leftarrow Old(q) \cup \{\eta\},$ 
33:             $New \leftarrow New(q),$ 
34:             $Next \leftarrow Next(q)];$ 
35:          return EXPAND( $q', Nodes);$ 
36:        end if
```

```

37:      else if  $\eta = \mu \cup \psi$  then
38:           $q_1 \leftarrow [ID \leftarrow \text{NEWID}(),$ 
39:               $Incoming \leftarrow Incoming(q),$ 
40:               $Old \leftarrow Old(q) \cup \{\eta\},$ 
41:               $New \leftarrow New(q) \cup \{\mu\},$ 
42:               $Next \leftarrow Next(q) \cup \{\mu \cup \psi\}];$ 
43:           $q_2 \leftarrow [ID \leftarrow \text{NEWID}(),$ 
44:               $Incoming \leftarrow Incoming(q),$ 
45:               $Old \leftarrow Old(q) \cup \{\eta\},$ 
46:               $New \leftarrow New(q) \cup \{\psi\},$ 
47:               $Next \leftarrow Next(q)];$ 
48:          return  $\text{EXPAND}(q_2, \text{EXPAND}(q_1, Nodes));$ 
49:      else if  $\eta = \mu \text{ R } \psi$  then
50:           $q_1 \leftarrow [ID \leftarrow \text{NEWID}(),$ 
51:               $Incoming \leftarrow Incoming(q),$ 
52:               $Old \leftarrow Old(q) \cup \{\eta\},$ 
53:               $New \leftarrow New(q) \cup \{\psi, \mu\},$ 
54:               $Next \leftarrow Next(q)];$ 
55:           $q_2 \leftarrow [ID \leftarrow \text{NEWID}(),$ 
56:               $Incoming \leftarrow Incoming(q),$ 
57:               $Old \leftarrow Old(q) \cup \{\eta\},$ 
58:               $New \leftarrow New(q) \cup \{\psi\},$ 
59:               $Next \leftarrow Next(q) \cup \{\mu \text{ R } \psi\}];$ 
60:          return  $\text{EXPAND}(q_2, \text{EXPAND}(q_1, Nodes));$ 
61:      else if  $\eta = \mu \vee \psi$  then
62:           $q_1 \leftarrow [ID \leftarrow \text{NEWID}(),$ 
63:               $Incoming \leftarrow Incoming(q),$ 
64:               $Old \leftarrow Old(q) \cup \{\eta\},$ 
65:               $New \leftarrow New(q) \cup \{\mu\},$ 
66:               $Next \leftarrow Next(q)];$ 
67:           $q_2 \leftarrow [ID \leftarrow \text{NEWID}(),$ 
68:               $Incoming \leftarrow Incoming(q),$ 
69:               $Old \leftarrow Old(q) \cup \{\eta\},$ 
70:               $New \leftarrow New(q) \cup \{\psi\},$ 
71:               $Next \leftarrow Next(q)];$ 
72:          return  $\text{EXPAND}(q_2, \text{EXPAND}(q_1, Nodes));$ 
73:      else if  $\eta = \mu \wedge \psi$  then
74:           $q' \leftarrow [ID \leftarrow \text{NEWID}(),$ 
75:               $Incoming \leftarrow Incoming(q),$ 
76:               $Old \leftarrow Old(q) \cup \{\eta\},$ 
77:               $New \leftarrow New(q) \cup \{\mu, \psi\},$ 
78:               $Next \leftarrow Next(q)];$ 
79:          return  $\text{EXPAND}(q', Nodes);$ 

```

```

80:         else if  $\eta = \mathbf{X} \mu$  then
81:              $q' \leftarrow [ID \leftarrow \text{NEWID}(),$ 
82:                  $Incoming \leftarrow Incoming(q),$ 
83:                  $Old \leftarrow Old(q) \cup \{\eta\},$ 
84:                  $New \leftarrow New(q),$ 
85:                  $Next \leftarrow Next(q) \cup \{\mu\}];$ 
86:             return EXPAND( $q', Nodes$ );
87:         end if
88:     end if
89: end if
90: end function

```
