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

Múlt és jövő: Új algoritmusok lineáris temporális tulajdonságok szaturáció-alapú modellellenőrzésére

TDK-dolgozat

Készítette:
Molnár Vince
Segesdi Dániel

Konzulensek:
dr. Bartha Tamás
Vörös András

2013.

Tartalomjegyzék

1. fejezet

Bevezető és motiváció

A technológia fejlődésével a számítógépes rendszerek alkalmazási köre ma már olyan biztonságkritikus rendszerekre is kiterjed, amelyek helyes működésétől sokszor teljes vállalatok sorsa, vagy akár emberéletek is függhetnek. Az ilyen rendszerek mérete és bonyolultsága ráadásul egyre nő, így szükség van megbízható, automatikus ellenőrző módszerek kifejlesztésére. A *formális módszerek* a tervezési folyamat helyességét garantálják azzal, hogy matematikai igényességgel igazolják a rendszer (biztonság szempontjából) kritikus tulajdonságainak teljesülését.

A biztonságkritikus rendszerek formális verifikációjára az elmúlt évtizedekben számos módszert dolgoztak ki, és a terület ma is dinamikusan fejlődik. Az egyik ilyen módszer a *modellellenőrzés*, amely a rendszerekről készített modellek lehetséges állapotait veszi számításba és veti össze a formalizált követelményekkel, hibás állapotszekvenciákat keresve.

A modellellenőrzés területén jellemzően a használt formalizmus és a modell leírásának módja alapján különíthetünk el különböző megoldás-csoportokat. Az egyes formalizmusok jellemzően különböző algoritmusokat igényelnek, míg az eltérő modellreprezentációk gyakran teremtenek lehetőséget specifikus tulajdonságok könnyebb ellenőrzésére. A követelmények felírásához használt formalizmus leggyakrabban valamilyen *temporális logika*, ezek segítségével a rendszerek lehetséges állapotsorozataira lehet elvárásokat megfogalmazni.

Kutatócsoportunk több tagja is foglalkozott már korábban a temporális logikákat használó modellellenőrzés témakörével, azon belül is főképp az *elágazó idejű temporális logikával* (CTL), illetve egy speciális algoritmussal, az ún. *szaturációval*. Ez az algoritmus rendkívül hatékonynak bizonyult nagyméretű, párhuzamos és aszinkron rendszerek állapotainak szimbolikus felderítésére.

Kutatásaink során elsőként javasoltunk szaturáció alapú algoritmust *lineáris idejű temporális logikával* (LTL) leírt tulajdonságok modellellenőrzésére. Akkori megoldásunk (3. fejezet) újszerűsége mellett azonban nem volt kellően általános megközelítés, ez pedig gátat szabott az ellenőrizhető rendszerek bonyolultságának.

Jelen dolgozatban a lineáris idejű modellellenőrzőnk számos továbbfejlesztését és kiterjesztését mutatjuk be. Új eredményeink közül a legfontosabbnak az ún. *múlt idejű lineáris temporális logikával* (PLTL) leírt specifikációk támogatását tartjuk. Jelentősen optimalizáltuk, javítottuk a modellellenőrzés köztes lépéseit is, és az egyes részproblémákra új, a korábbi módszereknél hatékonyabb és egyszersmind általánosabb algoritmust adtunk. A használt modellek tekintetében a korábban csak színezetlen Petri-hálókon működő modellellenőrzőt képessé tettük közvetlenül színezett Petri-hálókon való működésre is, így jelentősen bővítve az ellenőrizhető modellek körét.

Munkánk legjobb tudomásunk szerint egyedülálló, mivel rajtunk kívül korábban semmilyen olyan, általunk ismert megoldást nem publikáltak a lineáris idejű modellellenőrzés

problémájára, ami a rendkívül hatékony szaturációs algoritmust és a PLTL formalizmust használta volna. A megoldásban használt algoritmusok is egyediek, különösképp a szaturációs algoritmus kiterjesztése és az elfogadó lefutásokat kereső algoritmus használ egészen új megközelítést. A kutatásaink során kifejlesztett eszközt össze is vetettük korábbi algoritmusunk teljesítményével, valamint méréseket végeztünk a Paksi Atomerőműtől és a nemzetközi CERN kutatóintézettől kapott valós modelleken is. Ezek eredményei – mint az a 6. fejezetben látható – igazolták munkánk létjogosultságát.

Dolgozatunk a következőképpen épül fel. Először a munkánk elméleti hátterét adó ismereteket foglaljuk össze a 2. fejezetben, majd a 3. fejezetben röviden ismertetjük korábbi kutatásaink eredményeit is. A 4. fejezetben a modellellenőrzés során vizsgálandó követelmények kezelését mutatjuk be, úgy mint a logikai kifejezések automatává alakítását (4.1. szakasz), illetve a keletkezett automaták optimalizálását (4.2. szakasz). A következő, 5. fejezet a modellellenőrzés során használt algoritmusokat ismerteti, úgy mint a kifejezéssel komponált modell közös állapotterének szaturáció alapú felderítését (5.1. szakasz) és a kifejezéseket sértő futásokat kereső algoritmust (5.2. szakasz). Végül fejlesztéseink számszerű eredményeit a 6. fejezetben mutatjuk be, a 7. fejezetben összefoglalva munkánkat.

2. fejezet

Háttérismeretek

Ebben a fejezetben röviden bemutatjuk kutatásaink elméleti hátterét. Munkánk során a modellellenőrzési problémával foglalkoztunk (2.3. szakasz), amelynek keretében lineáris temporális logika (2.1. szakasz) segítségével megfogalmazott kritériumok vizsgálatát tettük lehetővé. Ez a fejezet mutatja be a modellellenőrzés köztes lépéseiben alkalmazott különböző automatatípusokat (2.2. szakasz), valamint az ellenőrzés alanyát képező modell kezelésének kérdéskörét is (2.4. szakasz), ide értve a modell állapotterének felderítését és reprezentációját.

2.1. Követelmények specifikálása temporális logikákkal

A modellellenőrzés során a vizsgált rendszer tulajdonságait vetjük össze az elvárt viselkedéssel. Ezt a specifikált viselkedést sokszor deklaratívan, különböző logikák megfogalmazásával célszerű megfogalmazni. A dinamikus viselkedésre vonatkozó tulajdonságok leírására használhatók a temporális logikák, amelyek közül a jelen dolgozatban bemutatott modellellenőrző algoritmus a *lineáris idejű temporális logikát* (LTL) használja. Az alábbiakban bemutatjuk az LTL felépítését, majd egy praktikus kiterjesztését, a *múltidejű temporális logikát* (PLTL) is.

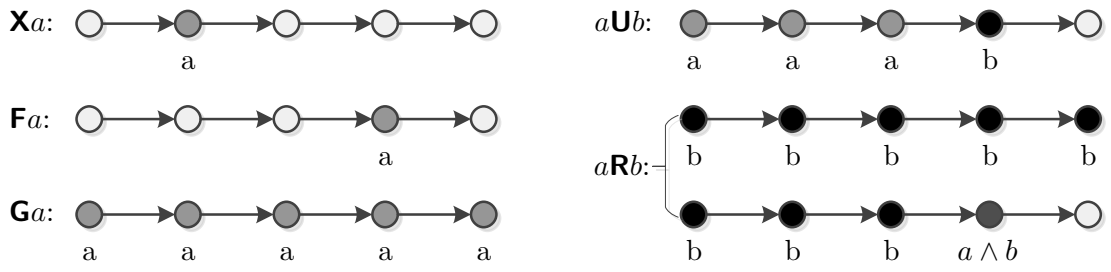
2.1.1. Lineáris idejű temporális logika

Mint minden logika, a lineáris idejű temporális logika (LTL) [4] is elemi kifejezésekből (ebben az esetben állapotkifejezésekből), illetve az ezek kapcsolatát kifejező logikai operátorokból épül fel. Ezeken kívül azonban *reaktív* rendszerekben 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 definiáljuk a munkánk alapját képező hat LTL temporális operátort:

- **X** („ne**X**t time") a következő állapotra ír elő feltételt.
- **F** („in the **F**uture") a feltétel azonnali vagy jövőbeli bekövetkezését követeli meg.
- **G** („**G**lobally") akkor teljesül, ha a kifejezés az összes jövőbeli és a jelenlegi állapotra is igaz.
- **U** („**U**ntil") 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** („**R**elease") 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.



2.1. ábra. Az LTL temporális operátorok szemléletes jelentése.

Az előbbi operátorokkal definiált LTL kifejezések csak végtelen hosszú állapotszekvenciákon érvényesek. Időnként célszerű, hogy véges szekvenciákra is definiálni lehessen LTL kifejezéseket, így bevezetjük az alábbi operátort is:

- $\tilde{\mathbf{X}}$ („weak ne**X**t time") az **X** logikai duálisa, amely ugyanúgy a következő állapotra ír elő feltételt, de akkor is teljesül, ha nem létezik következő állapot.

Végtelen szekvenciákon az **X**duálisa önmaga, vagyis ilyenkor $\mathbf{X} \equiv \tilde{\mathbf{X}}$. Véges lefutások vizsgálatakor azonban kérdés az is, hogy létezik-e következő állapot, így a duálisban ezt is másképp kell kezelni. Mivel minden temporális operátor visszavezethető **X**-re (rekurzívan) [9], így elegendő ezt az egy operátort „felkészíteni" a véges szekvenciákra és a rekurzív képletekben megfelelően használni a kétféle operátort.
Az LTL kifejezések szintaxisa a következő módon definiálható [4]:

- 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 $\mathbf{X}\ p$ és $p\ \mathbf{U}\ q$ is állapotkifejezések.

A többi operátor a következő szabályok segítségével fejezhető ki:

- $p \vee q \equiv \neg(\neg p \wedge \neg q)$
- $\tilde{\mathbf{X}}\ p \equiv \neg\mathbf{X}\ \neg p$
- $p\ \mathbf{R}\ q \equiv \neg(\neg p\ \mathbf{U}\ \neg q)$
- $\mathbf{F}\ p \equiv \mathit{True}\ \mathbf{U}\ p$
- $\mathbf{G}\ p \equiv \neg\mathbf{F}\ \neg p$

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

- $\mathbf{F}\ \psi \equiv \mathit{True}\ \mathbf{U}\ \psi$
- $\mathbf{G}\ \psi \equiv \mathit{False}\ \mathbf{R}\ \psi$

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:

- $\neg(\mu\ \mathbf{U}\ \eta) \equiv (\neg\mu)\ \mathbf{R}\ (\neg\eta)$
- $\neg(\mu\ \mathbf{R}\ \eta) \equiv (\neg\mu)\ \mathbf{U}\ (\neg\eta)$
- $\neg\mathbf{X}\ \mu \equiv \tilde{\mathbf{X}}\ (\neg\mu)$

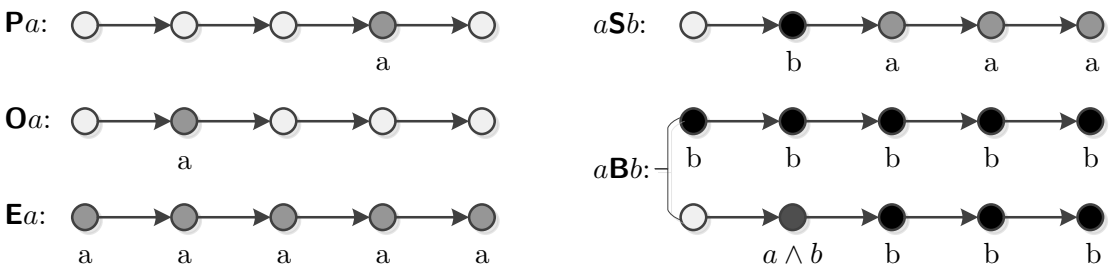
2.1.2. Múltidejű lineáris temporális logika

A lineáris idejű temporális logika egy lehetséges kiterjesztése a múltidejű temporális operátorok bevezetése, melyek ugyan nem növelik a logika kifejezőerejét, viszont bizonyos tulajdonságok sokkal intuitívabban, olykor egy ekvivalens LTL kifejezéshez képest exponenciálisan tömörebben megfogalmazhatók [8].

Az alábbiakban definiáljuk a már ismertetett LTL operátorok múltidejú párjait:

- $\mathbf{P}$ („**P**revious time") az $\mathbf{X}$ operátor múlt idejű megfelelője, az előző állapotra ír elő feltételt.
- $\tilde{\mathbf{P}}$ („weak **P**revious time") az $\tilde{\mathbf{X}}$ operátor múlt idejű megfelelője, a $\mathbf{P}$ logikai duálisa, amely ugyanúgy az előző állapotra ír elő feltételt, de akkor is teljesül, ha nem létezik előző állapot.
- $\mathbf{O}$ („**O**nce in the past") az $\mathbf{F}$ operátor múlt idejű megfelelője, a feltétel jelenlegi vagy múltbeli bekövetkezését követeli meg.
- $\mathbf{E}$ („**E**arlier always") a $\mathbf{G}$ operátor múlt idejű megfelelője, ami akkor teljesül, ha a kifejezés az összes múltbeli *és a jelenlegi állapotra is* igaz.
- $\mathbf{S}$ („**S**ince") az $\mathbf{U}$ operátor múlt idejű megfelelője, egy kétoperandusú operátor, ami akkor teljesül, ha valamikor a múltban létezett egy állapot, amire a második operandus igaz, és azóta minden állapot kielégíti az első operandust.
- $\mathbf{B}$ („**B**egins") az $\mathbf{R}$ operátor múlt idejű megfelelője, az $\mathbf{S}$ operátor logikai duálisa, teljesüléséhez a második operandusnak kell teljesülnie attól kezdve, hogy egy állapotra teljesül az első operandus *is*. Az első operandusnak azonban nem kell mindenképpen teljesülnie a múltban.

A jövőidejű esettel ellentétben itt mindenképpen szükség van a $\tilde{\mathbf{P}}$ operátorra, mivel a múltidejű operátorok által vizsgált szekvencia (a kezdőállapotból a jelenlegi állapotba) minden esetben véges. Azt is meg kell jegyezni, hogy a PLTL valóban csak egy kiterjesztés, mivel a leggyakrabban csak jövő idejű kifejezések részkifejezéseként alkalmazunk múltidejű formulákat, egyéb esetben ugyanis egyedül a kezdőállapotból álló egyelemű szekvenciára fogalmaznánk meg predikátumokat, így értelmetlen lenne a formalizmus használata.
A PLTL kifejezések szintaxisa a következővel egészül ki az LTL-hez képest:



2.2. ábra. A PLTL temporális operátorok szemléletes jelentése.

- Ha p és q állapotkifejezések, akkor $\mathbf{P}\ p$ és $p\ \mathbf{S}\ q$ is állapotkifejezések.

A többi múltidős operátor a következő szabályok segítségével kifejezhető:

- $\tilde{\mathbf{P}}\ p \equiv \neg\mathbf{P}\ \neg p$
- $p\ \mathbf{B}\ q \equiv \neg(\neg p\ \mathbf{S}\ \neg q)$
- $\mathbf{O}\ p \equiv \mathit{True}\ \mathbf{S}\ p$
- $\mathbf{E}\ p \equiv \neg\mathbf{O}\ \neg p$

Egy PLTL kifejezés *negált normál formáján* egy olyan ekvivalens kifejezést értünk, amiben az LTL kifejezések negált normál formájában megengedett operátorokon kívül kizárólag a $\mathbf{P}$, $\tilde{\mathbf{P}}$, $\mathbf{S}$ és $\mathbf{B}$ temporális operátorok szerepelhetnek, és negálás csak atomi kijelentések előtt állhat. Ahhoz, hogy egy kifejezést negált normál formába hozzunk, az LTL esetén megismert módszert kell alkalmaznunk az alábbi azonosságokkal kiegészítve:

- $\mathbf{O}\ \psi \equiv \mathit{True}\ \mathbf{S}\ \psi$
- $\mathbf{E}\ \psi \equiv \mathit{False}\ \mathbf{B}\ \psi$

Illetve:

- $\neg(\mu\ \mathbf{S}\ \eta) \equiv (\neg\mu)\ \mathbf{B}\ (\neg\eta)$
- $\neg(\mu\ \mathbf{B}\ \eta) \equiv (\neg\mu)\ \mathbf{S}\ (\neg\eta)$
- $\neg\mathbf{P}\ \mu \equiv \tilde{\mathbf{P}}\ (\neg\mu)$

A múlt idejű temporális logika bevezetésével sok tulajdonság intuitívabban, kompaktabb módon, és ezáltal érthetőbb alakban megfogalmazhatóvá válik. Erre egy példa az alábbi specifikációs követelmény múlt idejű temporális logika segítségével megfogalmazva:

$$\mathbf{G}\ (\mathit{hibajelzés} \Rightarrow \mathbf{O}\ \mathit{hiba})$$

Ez azt a tulajdonságot írja le, hogy ha *hibajelzés*t észlelünk, akkor előtte valamikor *hiba* is történt. Jól látszik, hogy a PLTL megfogalmazás ilyen formában könnyen értelmezhető, ezzel szemben a kifejezés LTL megfelelőjéről ugyanez már nem mondható el:

$$\neg((\neg\mathit{hiba})\ \mathbf{U}\ (\mathit{hibajelzés} \wedge \neg\mathit{hiba}))$$

Ez a kifejezés pontosan akkor teljesül, amikor az eredeti, PLTL-ben megadott tulajdonság, hiszen az, hogy hibajelzés előtt nem történt hiba megegyezik azzal, hogy nincs olyan állapot, ahol hibajelzés van, hiba nincs, és egész addig sem történt hiba. Ebből az egyszerű példából is látszik, hogy bár léteznek ekvivalens megfogalmazások a múltidejű kifejezésekre, azok az LTL kifejezések sokszor a tulajdonság nem teljesülésére fogalmaznak meg negált kifejezést, nehezen értelmezhetőek és nehezen is fogalmazhatók meg, ami megkönnyíti a hibás kifejezések felírását.

Érdemes megemlíteni, hogy bár a kifejezések ekvivalensek, a belőlük generált Büchi automaták nem feltétlenül azonosak, viszont az általuk elfogadott nyelvek igen.

2.2. Automataelmélet

Véges automata alatt egy olyan matematikai számítási modellt értünk, ami a bemenetére adott szimbólumsorozatokról, ún. *szavakról* a bemenet méretétől független, véges és állandó mennyiségű memória felhasználásával eldönti, hogy egy *nyelv* részét képezik-e. Egy véges automata működhet véges és végtelen bemeneteken (úgynevezett szavakon).

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

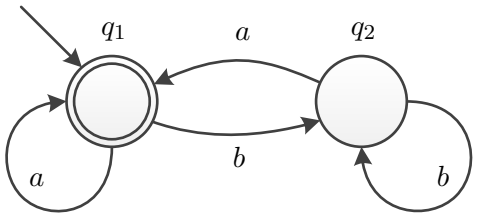
- Σ a véges *ábécé*.
- Q a lehetséges *állapotok* véges halmaza.
- $\Delta \subseteq Q \times \Sigma \times Q$ az *állapotátmeneti reláció*.
- $Q^0 \subseteq Q$ a *kiinduló állapotok* halmaza.
- $F \subseteq Q$ az *elfogadó állapotok* halmaza.

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

Legyen $v \in \Sigma^*$ egy szó, melynek hossza $|v|$. $\rho : \{0, 1, \ldots, |v|\} \mapsto Q$ 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 (*engedélyezett*), 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.



2.3. ábra. Egy egyszerű véges automata.

1. példa. *Tekintsük a 2.3. 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 a + választást jelöl, a * 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 jelezzük, nem-determinisztikus automatákkal foglalkozunk.

Mint korábban is említettük, a modellellenőrzés által vizsgált rendszerek többnyire reaktív rendszerek, ami azt jelenti, hogy rendeltetésszerű működésük során nem állnak le. A továbbiakban ezért bemutatjuk azokat a végtelen szavakon működő automata osztályokat is, amik ilyen rendszerek tulajdonságainak 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.

2.2.1. Büchi automaták

A legegyszerűbb végtelen szavakon működő véges automaták a Büchi [4] 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 [13].

2. példa. *A 2.3. 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.*

2.2.2. 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

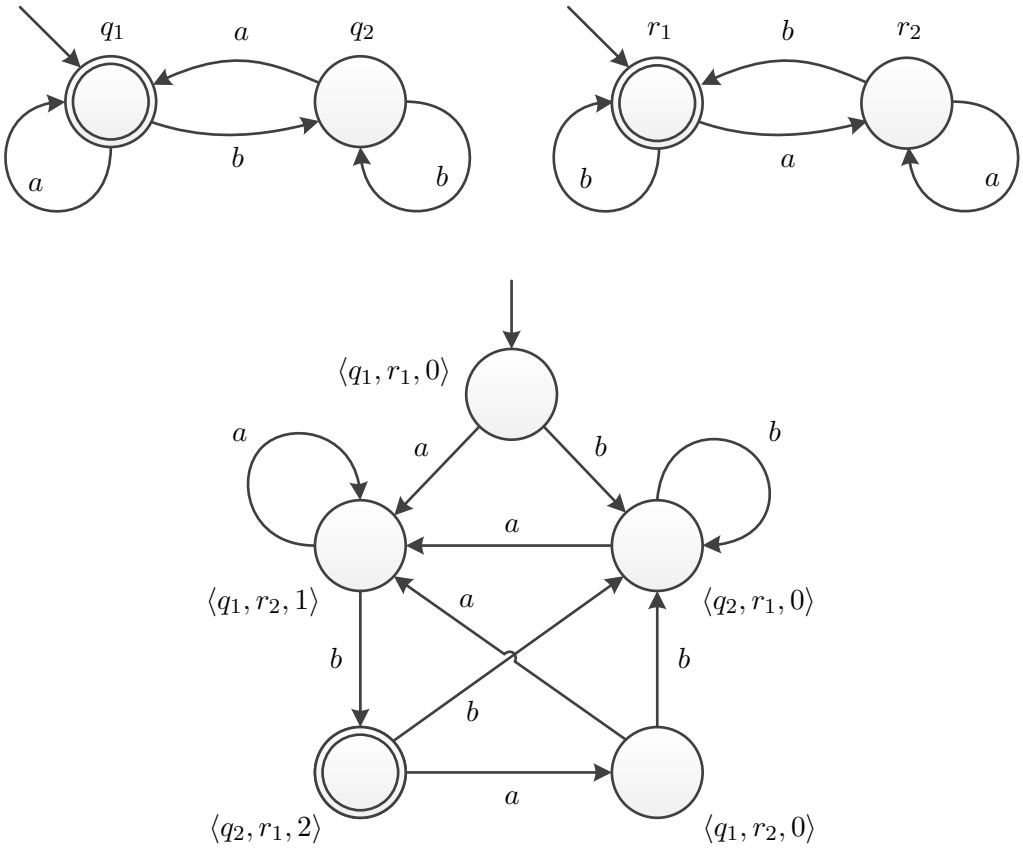
$$\mathcal{A}_1 \cap \mathcal{A}_2 = \langle \Sigma, Q_1 \times Q_2 \times \{0, 1, 2\}, \Delta, Q_1^0 \times Q_2^0 \times \{0\}, Q_1 \times Q_2 \times \{2\} \rangle$$

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 $y = x$.



2.4. ábra. Két Büchi automata és szinkron szorzatuk [4]. 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 jelenjenek 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.

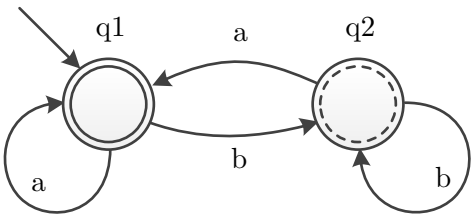
Szerencsére a szorzat automata előállítása ennél sokkal egyszerűbb, ha az egyik automata minden állapota elfogadó állapot. Ilyen automatát kapunk például, ha egy modell állapotterét automataként reprezentáljuk, ami – mint később bemutatjuk – éppen így is lesz az LTL modellellenőrzés során.

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 $(q_j, a, q_n) \in \Delta_2$.

2.2.3. 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. [4]

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.*



2.5. ábra. Egy Büchi automata.

3. példa. *Tekintsük most a 2.5. 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 jelenti. 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 $F = \{P_1, \ldots, P_n\}$ mellett $\mathcal{A}' = \langle \Sigma, Q \times \{0, \ldots, n\}, \Delta', Q^0 \times \{0\}, Q \times \{n\} \rangle$.

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.

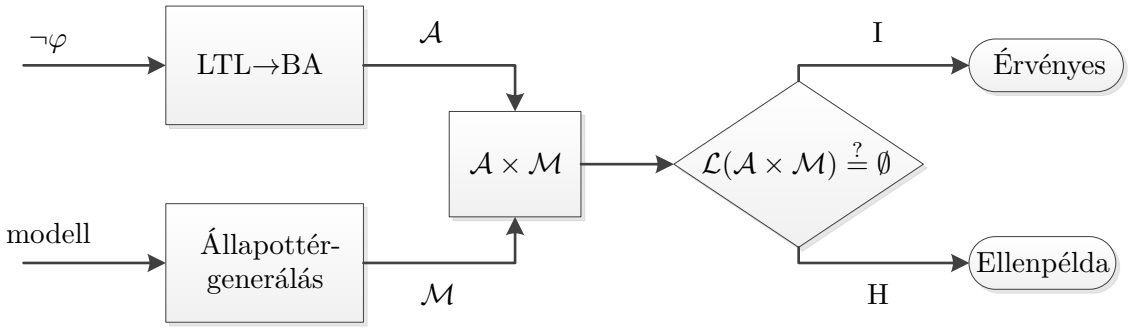
2.3. Modellellenőrzés

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 [4]. Az r követelmény többféle, különböző kifejezőerejű formalizmussal megadható, amikkel eltérő típusú kifejezések értékelhetőek ki.

Munkánk során temporális logikákat használtunk, 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 mutatjuk be.

2.3.1. Automataelméleti megközelítés

Az LTL temporális logika könnyű használhatósága és intuitív jellege miatt többféle modellellenőrző algoritmust is kifejlesztettek már hozzá. Ezek jellemzően automataelméleti megközelítést alkalmaznak [6].



2.6. ábra. LTL modellellenőrzés automatákkal.

Az LTL modellellenőrzés automataelméleti megközelítés esetén a következő módon vázolható:

1. Adott a vizsgálandó kifejezés. Ennek képezzük a negáltját, és építünk egy *specifikációs automatát*, amelynek 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 állapotváltozásaikor a célállapottal léptetjük a specifikációs automatát.[1]
3. Vizsgáljuk meg, hogy a *szorzat automata* által elfogadott nyelv üres-e. Ha üres, akkor a modell nem tartalmaz olyan lefutást, amely kielégítené a vizsgálandó kifejezés

[1] Technikailag ez úgy definiálható, hogy az állapottér automata-reprezentációjában is a modell állapotai a bemenetek, tehát minden átmenetet a célállapottal címkézünk. Ekkor a két automata szótára megegyezik, és a korábban ismertetett módon képezhető a szorzat automata.

negáltját, tehát a kifejezés érvényes a modellre. Ha nem üres, az adódó nyelv szavai megfelelnek azoknak a lefutásoknak, amelyek megsértik a specifikációt, tehát értékes ellenpéldákhoz jutottunk.

A második lépésben az állapottér automatává alakítása technikailag történhet úgy, hogy az automata-reprezentációban a modell állapotait definiáljuk bemenetekként, és minden átmenetet a célállapottal címkézünk. Ekkor a két automata szótára megegyezik, és a korábban ismertetett módon képezhető a szorzat automata. A gyakorlatban célszerűbb lehet a szinkron szorzatot úgy képezni, hogy először a modell állapotterében lepünk, majd az elért állapotot bemenetként használva a specifikációs automatában is. Sőt, hogy a specifikációs automata független legyen az állapottér nagyságától, célszerű csak az állapotokra érvényes predikátumokat (atomi kijelentéseket) bemenetnek választani, és a léptetés előtt elvégezni a célállapot-predikátumhalmaz átalakítást. Így a specifikációs automata a modelltől függetlenül, csak a predikátumokon dolgozva egyszerű maradhat, a modell állapottere pedig a Büchi-automatától különböző formalizmusokban (pl. Kripke-struktúra) is megadható.

A különböző megvalósításoknak tehát a specifikációs automata építését, a szorzat képzésének módját, valamint a nyelv ürességének ellenőrzését kell definiálnia. Utóbbi probléma véges modellek esetén egy körkeresési feladat, ugyanis ekkor a szorzat állapottér is véges, ebben pedig csak úgy lehet elfogadó állapotot végtelen sokszor érintő lefutás, ha a lefutás „vége” egy elfogadó állapotot is tartalmazó hurok.

2.4. Szimbolikus állapottér-generálás szaturációval

Az előző szakaszban feltételeztük, hogy a vizsgált modell állapottere adott. Ebben a szakaszban ennek az állapottérnek a felderítésével és tárolásával foglalkozunk, amelyre egy szimbolikus algoritmust fogunk használni. Az algoritmus kódolva tárolja és járja be az állapotokat, ahol minden részmodellt (komponenst) egy-egy változó segítségével fogunk kódolni. Az első alfejezet a megközelítéshez szükséges hátteret mutatja be.

2.4.1. Az állapottér-generálási probléma

Tekintsük az $M = \langle \mathcal{S}, \mathcal{S}_{init}, \mathcal{E}, \mathcal{N} \rangle$ diszkrét állapotterű modellt[15, 3], ahol

- az $\mathcal{S}$ állapottér az L darab részmodell állapottereinek $\mathcal{S}_L \times \cdots \times \mathcal{S}_1$ Descartes-szorzataként adódik, vagyis minden $\mathbf{i}$ globális állapot egy $\langle i_L, \ldots, i_1 \rangle$ L-es, ahol $i_k \in \mathcal{S}_k$ a k. *állapotváltozó* minden k-ra ($L \geq k \geq 1$);
- $\mathcal{S}_{init} \subseteq \mathcal{S}$ a kezdőállapotok halmaza;
- $\mathcal{E}$ az (aszinkron) események halmaza;
- az $\mathcal{N} : \mathcal{S} \to 2^{\mathcal{S}}$ állapotátmeneti függvény diszjunkt partíciók uniójaként adott, vagyis $\mathcal{N} = \bigcup_{\varepsilon \in \mathcal{E}} \mathcal{N}_\varepsilon$, ahol $\mathcal{N}_\varepsilon(\mathbf{i})$ az $\mathcal{E}$ esemény tüzelésével $\mathbf{i}$-ből egy lépéssel, nemdeterminisztikusan elérhető állapotok halmaza.

Azt mondjuk, hogy az ε esemény *engedélyezett* az $\mathbf{i}$ állapotban, ha $\mathcal{N}_\varepsilon(\mathbf{i}) \neq \emptyset$. Az $\mathcal{N}$ állapotátmeneti függvényhez hasonlóan definiáljuk az $\mathcal{N}^{-1}$ és $\mathcal{N}_\varepsilon^{-1}$ *inverz állapotátmeneti függvényeket* is, például $N_\varepsilon(\mathbf{i})$ jelölje azon állapotok halmazát, amelyekből az $\mathbf{i}$ globális állapot ε tüzelésével egy lépésben elérhető. A továbbiakban az általánosság megsértése nélkül feltételezzük, hogy $\mathcal{S}_k = \{0, \ldots, n_k - 1\}$.

A *lokalitás* az aszinkron rendszerek egyik alapvető tulajdonsága, ami azt jelenti, hogy a legtöbb esemény jellemzően csak néhány komponenst érint. Azt mondjuk, hogy egy ε

esemény *független* a k. részmodeltől, ha az engedélyezettsége nem függ annak i_k állapotától és a tüzelése nem is változtatja meg azt. Ha ε nem független a k. részmodelltől, akkor k az ε *hatáskörébe* (support) tartozik, vagyis $k \in supp(\varepsilon)$. Jelölje $Top(\varepsilon)$ a $supp(\varepsilon)$-ban lévő legmagasabb indexet és legyen $\mathcal{E}_k$ a $\{\varepsilon \in \mathcal{E} : Top(\varepsilon) = k\}$ halmaz.

Az *állapottér-generálás problémája* azon $\mathcal{S}_{rch}$ állapothalmaz kiszámítását jelenti, melynek elemei az S_{init} kezdőállapotokból elérhetők. *Explicit állapottér-generálás* alatt olyan megoldásokat értünk az állapottér-generálás problémájára, amik a kezdőállapot(ok)ból az állapotátmeneteken egyenként lépkedve valamilyen gráfbejáró algoritmus segítségével sorra veszik és eltárolják az elérhető állapotokat. Ez a módszer nagyon gyorsan korlátokba ütközik, mivel az állapotok egyenkénti tárolása és kezelése rendkívül sok erőforrást igényel. *Szimbolikus állapottér-generálás* alatt olyan technikákat értünk, amik az állapotokat valamilyen kódolással, kompaktan tárolják. Az következőkben egy ilyen, általunk is használt megoldás állapottér-reprezentációját és egy ezen működő hatékony algoritmust mutatunk be.

2.4.2. Diszkrét állapotterű rendszerek szimbolikus kódolása

Nagyméretű állapothalmazok kezelésére és tárolására döntően kétféle megoldás létezik. Az *explicit* megközelítés az állapotokat egyenként, felsorolásszerűen tárolja el. Bonyolultabb rendszerek állapottere azonban gyakran hatalmas lehet, így ezek a megoldások gyorsan korlátokba ütközhetnek. Ennek kiküszöbölésére vezették be a *szimbolikus* állapottér-reprezentációkat, amelyek az állapottér redundanciájának kihasználásával sok esetben kompakt tárolást tesznek lehetővé.

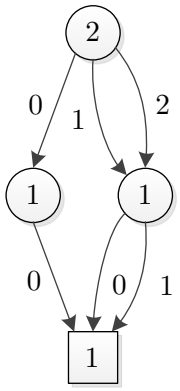
Diszkrét állapotterű modellek állapotainak kódolására ún. *többértékű döntési diagramokat* (multi-way decision diagram, MDD) [10, 2] fogunk használni. Az MDD-k a bináris döntési diagramok (binary decision diagram, BDD) természetes kiterjesztései, amennyiben a kódolt változók nem csak binárisak, hanem tetszőleges egészértékűek is lehetnek. Ez a tulajdonság alkalmassá teszi őket kezdetben ismeretlen (de remélhetőleg korlátos méretű) állapotterek hatékony reprezentálására.

A BDD-hez hasonlóan az MDD-ket is egy irányított, körmentes gráf reprezentálja, ahol a csomópontokat szintekbe rendezzük. Egy MDD-nek két *terminális* csomópontja van a legalsó szinten, a **0** és az **1**, illetve egyetlen *gyökér* csomópontja a legfelső szinten. Az éleket az adott szint által kódolt változó lehetséges értékeivel címkézzük. A gyökértől valamelyik terminális csomópontba vezető irányított út megadja a változók lehetséges lekötéseit: ha az út **0**-ban ér véget, a lekötés nincs az MDD által kódolt lehetséges lekötések halmazában, ellenkező esetben (ha **1**-be vezet) pedig igen.

Egy állapothalmazt egy L szintű *kváziredukált* MDD-vel írhatunk le. Egy a csomópont szintjét $a.lvl$ jelöli, ahol $L \geq a.lvl \geq 0$. Az előzőek alapján $a.lvl = 0$, ha a a **0** vagy **1** csomópont, illetve $a.lvl = L$, ha a a gyökér csomópont. Ha $a.lvl = k > 0$, akkor a-nak n_k kimenő éle van, mindegyik egy különböző $i_k \in \mathcal{S}_k$ címkével jelölve. Az i_k-val címkézett élen elérhető csomópontot jelölje $a[i_k]$. A kváziredukált tulajdonság miatt megköveteljük, hogy ez az $a[i_k]$ csomópont a $k-1$. szinten legyen, ha $a[i_k] \neq \mathbf{0}$, vagyis az MDD-ben nem lehet *szintugrás*.

4. példa. *A 2.7. ábrán látható MDD például a* $\{\langle 0,0\rangle, \langle 0,1\rangle, \langle 0,2\rangle, \langle 1,1\rangle, \langle 1,2\rangle\}$ *állapothalmazt kódolja. Az ábrán nem tüntettük fel a* ***0*** *csomópontba vezető éleket, az átláthatóság kedvéért a továbbiakban is így teszünk majd.*

Az állapotátmeneti függvény kódolását tekintve a lokalitás kihasználásával kompakt szimbolikus ábrázolást érhetünk el. Mivel az $\varepsilon \in \mathcal{E}_k$ állapotátmeneti függvény nem érinti az $i_L, \ldots, i_{k+1}$ változókat, $\mathcal{N}_\varepsilon(\langle i_L, \ldots, i_1\rangle)$ felírható $\{\langle i_L, \ldots, i_{k+1}\rangle\} \times \mathcal{N}_\varepsilon(\langle i_k, \ldots, i_1\rangle)$



2.7. ábra. Egy *többértékű döntési diagram.*

formában. Szavakkal, $\mathcal{N}_\varepsilon$-t csak az alsó k állapotváltozóra kell alkalmazni, így az MDD-vel kódolt állapothalmazon $\mathcal{N}_\varepsilon$ tüzelése csak az MDD k. szintű csomópontjaiból, mint gyökerekből induló rész-MDD-ket változtatja meg.

Az $\mathcal{N}$ teljes állapotátmeneti függvényt egy $2L$-szintű MDD-vel reprezentáljuk, a szinteket $L, L', \ldots 1, 1'$ módon sorrendezve. A páros sorszámú, k alakú szintek az átmenet kezdőállapotát, a páratlan sorszámú, k' alakú szintek pedig a végállapotát jelölik, $U(k) = U(k') = k$ pedig annak az állapotváltozónak a sorszáma, amelyre az adott szintpár hatással van. A 2.4.1. szakasznak megfelelően azonban nem tároljuk a teljes függvényt, hanem csak az egyes események állapotátmeneti függvényeit. Ehhez egy $\varepsilon \in \mathcal{E}_k$ eseményhez tartozó $\mathcal{N}_\varepsilon$ állapotátmeneti függvényt egy $2k$ szintű MDD-vel (röviden *2k-MDD*) reprezentálunk[2], és az átmeneti függvényt csak a $k = \mathit{Top}(\varepsilon)$ szinten kezdjük el feldolgozni.

A továbbiakban az MDD csomópontokat kisbetűkkel fogjuk jelölni, az MDD-kre pedig a gyökér csomópontjaikkal fogunk hivatkozni, vagyis az „a MDD" az „a gyökér csomópontú MDD"-t jelenti. Az a MDD a $\mathcal{B}(a) \subseteq \mathcal{S}_{a.lvl} \times \ldots \times \mathcal{S}_1$ (rész)állapothalmazt kódolja, az állapotváltozókat az a-ból az **1**-be vezető utaknak megfelelően lekötve. A jelölések egyszerűsítésére bizonyos „globális" mennyiségekre csak az őket leíró MDD-k segítségével fogunk hivatkozni, például $\mathcal{N}$ a kontextustól függően jelölheti majd magát az állapotátmeneti függvényt, illetve az azt leíró MDD-t is.

Adott $\mathcal{X} = \mathcal{B}(x) \subseteq \mathcal{S}$ állapothalmaz esetén az $\mathcal{N}_\varepsilon(\mathcal{X})$ állapothalmaz kiszámítása a *RelProd* szimbolikus operátor segítségével implementálható, ami egy L szintű és egy $2L$ szintű MDD-t fogad bemenetként, és egy L szintű MDD-t ad vissza:

$$\mathcal{N}_\varepsilon(\mathcal{X}) = \mathcal{B}(\mathit{RelProd}(x, \mathcal{N}_\varepsilon)).$$

Az MDD-k használata nem vezet nagyobb hatékonyságra a BDD-k használatánál, vagy akár az explicit megoldásoknál, ráadásul a hatékonyság és az MDD-k mérete a BDD-k használatához hasonlóan nagyban függ a változók sorrendezésétől, optimális sorrendezést találni pedig matematikailag nehéz feladat. Azért használunk mégis MDD-ket a szimbolikus tárolásra, mert olyan modellek esetén, amikben az egyes részkomponensek lokális állapotterének mérete nem ismert előre (ilyenek például a *Petri-hálók*), az állapottér felderítése közben talált új lokális állapotok reprezentálására szükség esetén könnyen növelhetjük az MDD szintjeinek élszámát.

[2]Minden egyes részmodellhez egy kezdő- és egy végállapot tartozik, ezeket kódolja a páros sok $2k$ szint.

2.4.3. A szaturációs algoritmus

Az állapottér-generálás problémája megfogalmazható a kezdőállapotokra nézve az állapotátmeneti függvény tranzitív lezártjaként. A szimbolikus állapottér-generáló algoritmusok is ezt a szemléletet követik, a legegyszerűbb esetben közvetlenül az előbbi definíció szerint: szélességi bejárást (BFS) követve újra és újra alkalmazzák az állapotátmeneti függvényt, amíg az $\mathcal{S}_{init} \cup \mathcal{N}(\mathcal{S}_{init}) \cup \mathcal{N}^2(\mathcal{S}_{init}) \cup \ldots$ számítási sorozat fixpontra nem jut. Az $\mathcal{X}$ halmazból előrefelé elérhető állapotok halmaza $\mathcal{N}^*(\mathcal{X}) = \mathcal{X} \cup \mathcal{N}(\mathcal{X}) \cup \mathcal{N}^2(\mathcal{X}) \cup \ldots$ módon számítható, míg a visszafelé elérhető állapotok $(\mathcal{N}^{-1})^*(\mathcal{X}) = \mathcal{X} \cup \mathcal{N}^{-1}(\mathcal{X}) \cup \mathcal{N}^{-2}(\mathcal{X}) \cup \ldots$ szerint adódnak.

A szaturációs algoritmus [1, 2, 3] egy új iterációs stratégia, amely másféle megközelítést használ, rekurzívan kiszámítva a *lokális fixpontokat* és kihasználva a dekomponált állapotátmeneti függvények lokalitását. A kulcsgondolat az, hogy az egyes eseményeket a *Top* értékük szerint sorrendezzük: az $\mathcal{E}_k$-beli eseményeket csak akkor kezdjük eltüzelni a k. szinten lévő a csomóponton, ha korábban kimerítően eltüzeltük az összes $\mathcal{E}_h$-beli eseményt minden, a $h < k$ szinteken lévő a alatti csomóponton, egészen addig, amíg már nem található új állapot. Ezután az a csomópont akkor lesz *szaturált*, ha fixpont a k. szint feletti szintektől független eseményekre nézve, vagyis $\forall h, k \geq h \geq 1, \forall \varepsilon \in \mathcal{E}_h, \mathcal{B}(a) \supseteq \mathcal{N}_\varepsilon(\mathcal{B}(a))$. Egy a csomópont szaturálásakor tehát elsőként szaturálnunk kell az összes $a[i_k]$ csomópontot, majd az $\mathcal{E}_k$ beli eseményeket újra-és újra el kell tüzelnünk az a csomóponton mindaddig, amíg új állapotok keletkeznek. A szaturáltság definíciójának következménye, hogy új alsóbb szintű csomópont keletkezésekor azt is azonnal szaturálni kell.

Az 1. algoritmus mutatja a szaturáció pszeudokódját. Feltételezzük, hogy $\mathcal{N}_L, \ldots, \mathcal{N}_1$ előre ismertek. A pszeudokód az előre irány algoritmusát írja le, a visszafelé történő felderítés egyszerűen kapható az $\mathcal{N}\,\mathcal{N}^{-1}$-re cserélésével. A bemenet a szaturálandó s csomópont, amit az S_{init} kezdőállapotokat kódoló MDD gyökerére kell állítani. A *Saturate* függvény sorban szaturálja az MDD csomópontjait, lentről felfelé haladva. Az algoritmus a lokalitás által okozott redundanciát az események alsóbb szinteken történő tüzelése mellett a pszeudokódban látható *cache* alkalmazásával is kihasználja. A hagyományos *RelProd* operátorhoz képest a *RelProdSat* függvény mindig szaturált csomópontot ad vissza.

2.4.4. Vezérelt szaturáció

A *vezérelt szaturációs algoritmus* [14] elkészítésének motivációja kezdetben a CTL (Elágazó-idejű Temporális Logika, Computation Tree Logic) alapú modellellenőrzésből eredt, ahol az egyik logikai operátor ellenőrzéséhez visszafelé elérhető állapotokat kellett keresni. Eközben nem lehetett kilépni egy korábban már felderített c állapothalmazból (*„constraint"*), aminek biztosítására egy hagyományos, BFS alapú algoritmus az újonnan elért állapotokat egyszerűen c-val metszhette minden lépés után. Ha azonban a szaturációs algoritmusban is hasonló elvet követnénk, minden esemény eltüzelése után el kéne végeznünk egy metszést, ami költséges művelet, így elrontotta volna a szaturációs algoritmus hatékonyságát[14].

A vezérelt szaturációs algoritmus ehelyett a „lép és vág" megközelítés helyett az „ellenőriz és lép" stratégiát alkalmazza, ami c-n belülre *„kényszeríti"* az állapottér-felderítést minden egyes rekurzív *RelProd* műveletnél. A vezérelt szaturáció a következő megfigyelésen alapszik:

$$\mathcal{B}(t) = \mathit{RelProd}(s, r) \cup \mathcal{B}(c) \Rightarrow \mathcal{B}(t[i']) = \bigcup_{\forall i \in \mathcal{S}_l} (\mathit{RelProd}(s[i], r[i][i']) \cap \mathcal{B}(c[i'])),$$

ahol s és t l szintű, állapothalmazokat kódoló MDD-k, r pedig $2l$ szintű, állapotátmeneti függvényt kódoló 2k-MDD. A megfigyelés segítségével az ellenőrzést mindig egy

Algoritmus 1 A szaturációs algoritmus

```
1: function SATURATE(mdd s)
2:     if InCache_Saturate(s, t) then return t;          ▷ ne dolgozzunk feleslegesen
3:     level k ← s.lvl;
4:     mdd t ← 0;
5:     for each i ∈ S_k : s[i] ≠ 0 do      ▷ először szaturáljuk az alsóbb csomópontokat
6:         t[i] ← Saturate(s[i]);
7:     end for
8:     repeat                                   ▷ tüzeljük N_k-t amíg van változás
9:         for each i, i' ∈ S_k : N_k[i][i'] ≠ 0 do
10:            t[i'] ← Union(t[i'], RelProdSat(t[i], N_k[i][i']));
11:        end for
12:    until t nem változik;
13:    t ← InsertUT(t);          ▷ hash-tábla (unique table) a duplikációk elkerülésére
14:    CacheAdd_Saturate(s, t);                  ▷ elkerülendő a felesleges munkát
15:    return t;
16: end function

17: function RELPRODSAT(mdd s, r)
18:    level k ← s.lvl;
19:    mdd t ← 0;
20:    if s = 1 és r = 1 then  return t;                          ▷ terminális eset
21:    if InCache_RelProdSat(s, r, t) then return t;
22:    for each i, i' ∈ S_k : r[i][i'] ≠ 0 do
23:        t[i'] ← Union(t[i'], RelProdSat(s[i], N_k[i][i']));
24:    end for
25:    t ← InsertUT(t);                          ▷ eddig hasonló a RelProd-hoz...
26:    t ← Saturate(t);                              ▷ ...de most szaturáljuk t-t
27:    CacheAdd_RelProdSat(s, r, t);
28:    return t;
29: end function
```

szinttel lejjebbre vezethetjük vissza, a terminális szinten pedig azt kell megvizsgálni, hogy a metszés művelet mindkét operandusa **1**-e. A 2. algoritmus mutatja a vezérelt szaturáció pszeudokódját. Jól látható, hogy a rekurzív hívások előtt a metszésnek megfelelően ellenőrzésre kerül a kényszer, a hívásban pedig a megfelelő, eggyel alacsonyabb szintszámú rész-MDD kerül átadásra. Ezáltal a kényszer ellenőrzése felfogható a jól ismert „ha-akkor-egyébként" vezérlési szerkezet – „switch-case" szerkezetként is ismert – egészértékű kiterjesztéseként, ugyanis minden szinten a célállapottól függően egy új kényszert rendelünk a rekurzív híváshoz. Belátható, hogy így az algoritmus által talált új állapotok garantáltan c-beliek lesznek.

Látható, hogy a kényszer ellenőrzése gyakorlatilag tetszőleges p feltételre kicserélhető lenne, amennyiben p minden l szinten csak a felette lévő $L - l$ darab szinten bekövetkezett és az aktuális, vizsgált lépés céljától függene. Egy ilyen feltétel ugyanis $p[i]$ megfelelő definiálásával ábrázolható lenne MDD-ként, ha $p[i]$ **0**-t vagy **1**-t adna vissza a 0. szinten, felette pedig **0**-t vagy egy tetszőleges (akár p) nemterminális q csomópontot. A csomó-

pontok ekkor az eddigi döntések *emlékei* lennének[3], és ennek alapján, valamint i értéke alapján $p[i] = \mathbf{0}$-val jelezhetnék a tiltást, $p[i] = q$-val a kérdés alsóbb szintre tolását, és végül a legalsó szinten $p[i] = \mathbf{1}$-gyel a végleges elfogadást. Ezt a szemléletet használja ki a következő fejezetben bemutatásra kerülő korábbi algoritmusunk, illetve a kényszer *emlékezőképességét* kihasználó, az 5.1. szakaszban tárgyalt új módszerünk is.

[3]Érdemes megfigyelni, hogy bár a csomópontok több úton is elérhetőek lehetnek a gyökér csomópontból, szaturálásuk során mindig egy konkrét útvonal végén vizsgáljuk őket. Ezt az útvonalat jegyzi meg a kényszer.

Algoritmus 2 A vezérelt szaturációs algoritmus

```
1: function ConsSaturate(mdd s, c)
2:     if InCache_ConsSaturate(s, c, t) then return t;
3:     level k ← s.lvl;
4:     mdd t ← 0;
5:     for each i ∈ S_k : s[i] ≠ 0 do        ▷ először szaturáljuk az alsóbb csomópontokat
6:         if c[i] ≠ 0 then
7:             t[i] ← Saturate(s[i], c[i]);
8:         else                    ▷ ezen az ágon a kényszer miatt nem fedezhető fel új állapot
9:             t[i] ← s[i];
10:        end if
11:    end for
12:    repeat                                  ▷ kiszámítjuk a lokális fixpontot
13:        for each i, i' ∈ S_k : N_k[i][i'] ≠ 0 do
14:            if c[i'] ≠ 0 then          ▷ csak akkor tüzelünk, ha a kényszer is megengedi
15:                t[i'] ← Union(t[i'], ConsRelProd(t[i], c[i'], N_k[i][i']));
16:            end if
17:        end for
18:    until t nem változik;
19:    t ← InsertUT(t);
20:    CacheAdd_ConsSaturate(s, c, t);
21:    return t;
22: end function
23: function ConsRelProd(mdd s, c, n)
24:    level k ← s.lvl;
25:    mdd t ← 0;
26:    if s = 1 és n = 1 then return t;                          ▷ terminális eset
27:    if InCache_ConsRelProd(s, c, n, t) then return t;
28:    for each i, i' ∈ S_k : n[i][i'] ≠ 0 do
29:        if c[i'] ≠ 0 then          ▷ csak akkor tüzelünk, ha a kényszer is megengedi
30:            t[i'] ← Union(t[i'], ConsRelProd(s[i], c[i'], n[i][i']));
31:        end if
32:    end for
33:    t ← InsertUT(t);
34:    t ← ConsSaturate(t, c);                                   ▷ szaturáljuk t-t
35:    CacheAdd_ConsRelProd(s, c, n, t);
36:    return t;
37: end function
```

3. fejezet

Korábbi eredmények

Korábbi kutatásaink során elsőként adtunk szaturáció alapú algoritmust az automataelméleti modellellenőrzés problémájára [11], ebben a fejezetben ezt mutatjuk be röviden. Megoldásunk legfontosabb jellemzője az volt, hogy inkrementálisan és „on-the-fly" módon végezte az ellenőrzést, vagyis az állapottér felderítése közben folyamatosan vizsgálta a szorzat automata nyelvének ürességét, és azonnal megállt, ha a specifikációt sértő lefutást talált.

A 2.3.1. szakasznak megfelelően az algoritmus működése három fő részre osztható: az LTL-ben megfogalmazott specifikáció automatává alakítására (3.1. szakasz), a szorzat állapottér felderítésére (3.2. szakasz), és az elfogadó lefutások keresésére (3.3. szakasz).

3.1. LTL transzformáció

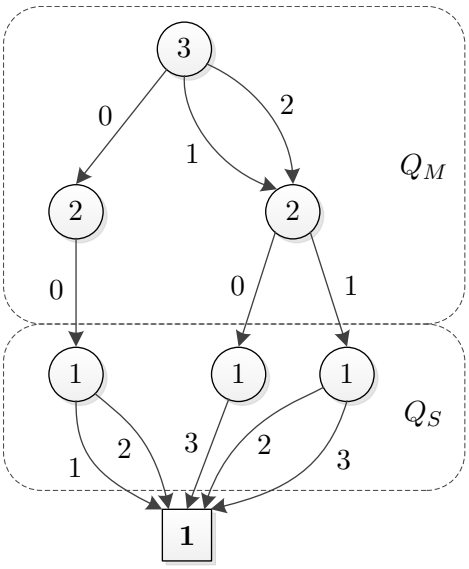
A specifikációk definiálását megoldásunk LTL kifejezések segítségével tette lehetővé. Ehhez adni kellett egy algoritmust, ami a logikai formulákat automatává alakítja. A felhasznált algoritmus Gerth, Peled, Vardi és Wolper algoritmusa [6] lett, amely az ún. *tabló módszer* felhasználásával negált normál formában lévő LTL kifejezésekből készített általánosított Büchi automatákat. A modellellenőrzés megkönnyítésére ezeket az automatákat hagyományos, nem általánosított Büchi automatákká alakítottuk és bizonyos optimalizálásoknak és egyszerűsítéseknek vetettük alá.

A választott transzformációs algoritmus az egyik legegyszerűbb megvalósítás, így a kapott automata nem volt minimális. Rendelkezett azonban egy olyan tulajdonsággal, amit az állapottér felderítése közben lehetőség adódott a szorzat állapottér szimbolikus kódolására és vezérelt szaturáció alapú felderítésére. Az algoritmus ugyanis a tabló módszer használata miatt úgy állította elő az egyes állapotokat, hogy azokhoz „belépési feltételeket" definiált, és az utolsó lépésben ezeket emelte ki az állapotátmenetek címkéibe. Emiatt minden átmenet, ami ugyanabba az állapotba ment, azonos címkével rendelkezett. Ez a tulajdonság ugyan lehetővé tette a szorzat állapottér hatékony, „on-the-fly" számítását, de egyúttal redundáns automatákat is eredményezett, amiket a tulajdonság megtartása mellett csak korlátozottan lehetett minimalizálni.

3.2. Szorzat állapottér generálása

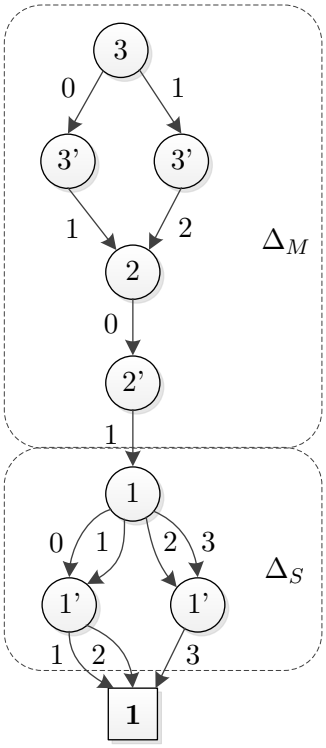
A modellellenőrző algoritmusunk [12] egyik újítása az volt, hogy közvetlenül a szorzat állapotteret derítette fel, szaturációval, a vizsgált rendszer magas szintű modelljét és a specifikációs automatát felhasználva. Ehhez elsőként találnunk kellett egy megfelelő szimbolikus reprezentációt a kompozit állapotok és a kompozit események állapotátmeneti függvényeinek reprezentálására. Amiatt, hogy a specifikációs automata átmenetei nem pusztán álla-

potpárokból álltak, hanem azokat engedélyező bemenetekből is, utóbbi feladat megoldása közel sem volt triviális, ami akkori megoldásunk egyik fontos kontribúciójának számított.



3.1. ábra. Egy szorzat állapotteret kódoló MDD.

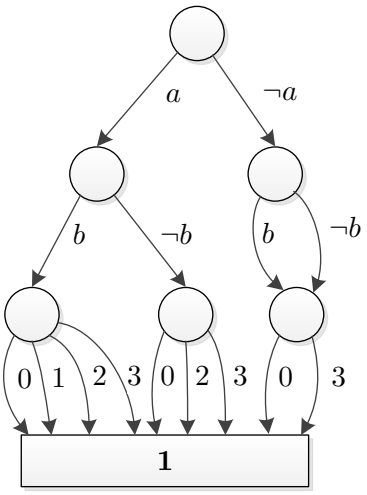
Algoritmusunk az állapotteret a klasszikus szaturációs algoritmusban szokásos L szint mellett további egy szinttel ábrázolta (3.1. ábra), ez a további szint kódolta az általában kisméretű specifikációs automata állapotait. Mivel az automatát minden kompozit esemény érintette, a szaturációs algoritmus hatékonyságát adó lokalitás megőrzése érdekében az automata állapotait a legalsó szintre helyeztük.



3.2. ábra. Egy szinkron eseményt kódoló MDD.

Az állapotátmeneti függvények közvetlen reprezentálásakor elhanyagoltuk az automata átmeneteiben szereplő bemenetet, és csak a – bármilyen bemenet hatására engedélyezett – átmeneteket ábrázoltuk a függvényeket kódoló 2k-MDD-k legalsó két szintjén (3.2. ábra). Magát a 2k-MDD-t a modellbeli eseményeket leíró 2k-MDD és az automata átmeneteit kódoló kétszintű 2k-MDD metszeteként állítottuk elő, ezáltal az így kódolt események azt fejezték ki, hogy a modell egy lépésekor az automata mindig bárhová léphet. Ezt a szabadságot a *vezérelt szaturációs algoritmussal* és egy megfelelően definiált, ún. *predikátum kényszerrel* ellensúlyoztuk. Ez a kényszer akadályozta meg, hogy a szorzat állapotterében olyan lépéseket is megtegyünk, amit a szinkron szorzat (szinkronitása) egyébként nem engedett volna meg.

A predikátum kényszer definiálása a 2.4.4. szakasz végén említett szemléletet használta fel, vagyis a kényszer immár nem azt követelte meg, hogy egy c-vel adott állapottéren belül maradjunk, hanem minden egyes állapotra kiértékelt egy bizonyos feltételt, és csak annak teljesülése esetén vette fel az új állapotot az állapottérbe. Ez a feltétel az volt, hogy a modellbeli új állapot (mint célállapot) által igazzá tett atomi kijelentések halmaza (mint a specifikációs automata bemenete) olyan legyen, amivel beléphetünk az automatabeli célállapotba. Itt használtuk ki a specifikációs automaták azon tulajdonságát, hogy az egyes átmenetek tüzelésére szabott feltételek helyett a célállapothoz rendelt belépési feltétellel dolgozhatunk.



3.3. ábra. Egy predikátum kényszert kódoló MDD.

A predikátum kényszer felépítése még az állapottér felderítése előtt, pusztán az automata ismeretében elvégezhető volt, mivel struktúráját függetlenítettük az állapottértől. Ezt úgy értük el, hogy a kényszert leíró MDD-ben az egyes atomi kijelentések igazságtartalmát reprezentáló szinteket vettünk fel a kijelentések tárgyát képező részmodellek indexelésének sorrendjében (mindegyik kijelentéshez egyet). Egy ilyen szint által kódolt részállapottér tehát egy $p \in AP$ predikátum igaz (*True*) vagy hamis (*False*) értékét tartalmazta. Itt is a legalsó szintre szúrtuk be az automata megengedett állapotai kódoló részt, ezáltal a kényszer MDD által kódolt globális állapotok automatabeli bemenetek és célállapotok lehetséges kombinációit jelentette.

A vezérelt szaturációs algoritmus futtatása közben a kényszer „léptetését” a $c[i]$ operátor megfelelő felüldefiniálásával végeztük, így ez az operátor végezte el az egyértelmű leképezést a modell célállapotának adott szinten vizsgált i részállapotáról az i-n értelmezett $L(i)$ atomi kijelentések lekötéseire.[1] A predikátum kényszer szintjeinek imént defini-

[1]Egy részállapotra 0, egy vagy több kifejezés is vonatkozhat.

ált sorrendje miatt így a az egyes lekötések szerint egyértelműen meghatározható volt a visszaadandó kényszer csomópont, ennek megvalósítását mutatja a 3. algoritmus.[2] A pszeudokódban szereplő $c[i]$ az MDD csomópontokon megszokott klasszikus függvényt jelenti, míg az imént felüldefiniált $c[i]$ függvényt számíthatjuk *StepConstraint*(c, i, l) függvénnyel, ahol l az a szint, amin az i részállapotot értelmezzük. AP_l az l. szintre vonatkozó predikátumok halmaza. Minden lekötéshez egy lépés tartozik, így egy szinten a kényszerben 0 vagy több lépést is tehetünk.

Algoritmus 3 A kényszer MDD-t léptető *StepConstraint* függvény

```
1:  function StepConstraint(c, i, l)
2:      if l = 1 then return c[i];      ▷ az alsó szinten a bemenet az automata célállapota
3:      for each p ∈ AP_l do            ▷ a többi szinten ki kell értékelni a megfelelő p-ket
4:          if p igaz i-re then
5:              c ← c[1];               ▷ az 1 lokális állapot a True értéket reprezentálják
6:          else
7:              c ← c[0];                          ▷ a 0 lokális állapot pedig a False-t
8:          end if
9:      end for
10:     return c;        ▷ az esetleges lekötéseknek megfelelő lépések után visszadjuk c-t
11: end function
```

A vezérelt szaturációs algoritmus ezzel a kényszerrel a szorzat állapottér elérhető állapotait derítette fel, együttesen megvalósítva a 2.3.1. szakaszban bemutatott sémából az állapottér-generálást és a szorzat állapottér számítását.

3.3. Elfogadó lefutások keresése

Korábbi munkánk legnagyobb kontribúciója egy újszerű, szimbolikus, szaturáció alapú és inkrementális körkereső algoritmus volt. Mint azt a 2.3.1. szakaszban is megemlítettük, végtelen szavak és véges modell esetén egy elfogadó lefutás mindenképpen egy elfogadó állapotot is tartalmazó hurokba torkollik, a modellellenőrző algoritmusnak tehát ilyen köröket kell hatékonyan keresnie.

Korábban is léteztek már szimbolikus körkereső (fixpontszámító) algoritmusok, azonban ezeket a teljes állapottéren kellett futtatni, ezzel pedig elveszett az explicit LTL modellellenőrző algoritmusok egyik legfontosabb előnye, az „on-the-fly” működés. Az „on-the-fly” megoldások kiemelkedő erénye, hogy ellenpélda létezése esetén nem derítik fel a teljes állapotteret, csak addig futnak, amíg meg nem találják azt. Ez rendkívül előnyös, mivel a modellellenőrzést alkalmazó munkafolyamatokban jellemzően hibás modelleket ellenőriznek többször, minden javítás után újra próbálkozva, amíg végül a javított modell teljesíti a követelményeket. Fontos tehát, hogy az esetleges ellenpéldákat minél hamarabb találják meg az algoritmusok.

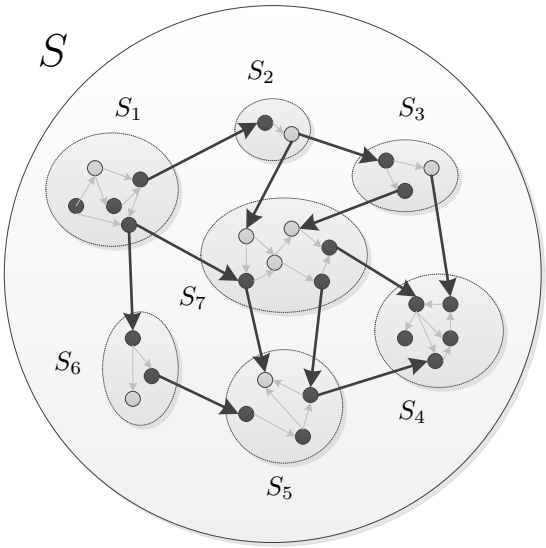
Ahhoz, hogy a korábbi szimbolikus körkereső technikákkal „on-the-fly” modellellenőrzést végezzünk, időről időre futtatni kellene őket, de ezek az algoritmusok nem képesek inkrementális működésre, tehát a menet közbeni ellenőrzéseknek óriási teljesítményveszteség lenne az ára. Az általunk fejlesztett algoritmus így legfőképp inkrementális működésre és a teljesítménynövekedés reményében a szaturációs algoritmus alkalmazására törekedett.

Az algoritmus azon a megfigyelésen alapult, hogy ha egy állapot része egy körnek, akkor legalább egy lépéssel (tehát nemtriviálisan) elérhető önmagából. Ezt általánosítottuk

[2]Technikai részlet, de az így meghatározott leképezésnek szüksége van az i részállapothoz tartozó szint számára is.

állapothalmazokra is, kimondva, hogy ha egy kör tartalmaz $\mathcal{X}$-beli elemet, akkor $\mathcal{X} \cap \mathcal{N}^+(\mathcal{X}) \neq \emptyset$, ahol $\mathcal{N}^+(\mathcal{X}) = \mathcal{N}(\mathcal{X}) \cup \mathcal{N}(\mathcal{N}(\mathcal{X})) \cup \ldots$ az állapotátmeneti függvény legalább egyszeri alkalmazásával elérhető állapotok halmazát jelöli. Egy-egy metszés azt jelenti, hogy $\mathcal{X}$-et azokra az állapotokra szűkítjük, amelyek potenciálisan elérhetők önmagukból. Ha tehát a szűkítések során fixpontra jutunk, az állapothalmazon valóban áthalad egy $\mathcal{N}$-beli átmeneteket felhasználó kör, ha viszont $\mathcal{X}$ „elfogy", akkor kör sem lehetett [12]. Szaturáció alkalmazásával $\mathcal{N}^*(\mathcal{X})$ számítható, amelyben a triviálisan elérhető állapotok is szerepelnek, de ez a következőkben bemutatott működés miatt nem jelentett problémát.

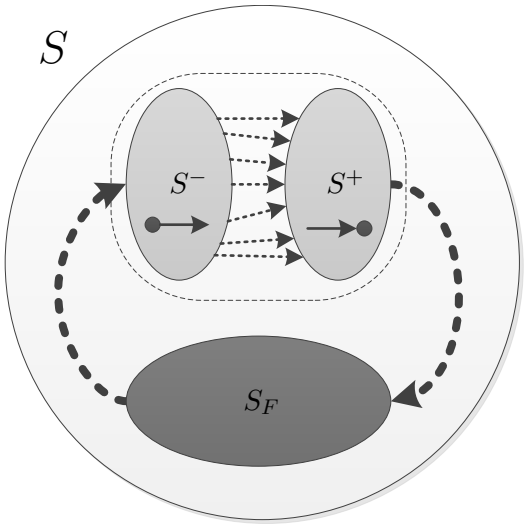
A valódi, inkrementális „on-the-fly" működéshez először is definiálnunk kellett a menet közbeni ellenőrzés finomságát. A szaturációs algoritmushoz alkalmazkodva azt mondtuk, hogy körkeresést minden l. szinten lévő n csomópont szaturálttá válásakor futtatunk, ami azért is célszerű választás, mert a csomópont által kódolt részállapottér ilyenkor zárt az $\mathcal{E}_l$-beli eseményekre. Az inkrementalitást pedig úgy értük el, hogy minden kereséskor csak az alacsonyabb szintű $n[i]$ csomópontok szaturálása óta újonnan megjelenő $\mathcal{E}_l$-beli eseményeket is használó köröket kerestük az egyes lépésekben. Ez azért volt elegendő, mert a csak alacsonyabb szintű eseményeket felhasználó körök az alacsonyabb szintű csomópontok szaturálása után előkerültek volna. Ennek szemléltetése látható a 3.4. ábrán.



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

Ahhoz, hogy ezt megvalósítsuk, már nem volt elég egyetlen halmazt ($\mathcal{S} = \mathcal{B}(n)$-t) szűkíteni. Olyan köröket kerestünk, amelyek érintik az $\mathcal{E}_l$ eseményekhez tartozó $\mathcal{N}_{\mathcal{E}_l}$ átmenetek $\mathcal{S}$-beli $\mathcal{S}^- = \mathcal{N}_{\mathcal{E}_l}^{-1}(\mathcal{S})$ kezdőállapotait és $\mathcal{S}^+ = \mathcal{N}_{\mathcal{E}_l}(\mathcal{S}^-)$ végállapotait, és köztük egy $\mathcal{N}_{\mathcal{E}_l}$-beli átmeneten haladnak. Mivel az algoritmus szempontjából csak az elfogadó állapotot is tartalmazó körök számítanak, felvettünk egy harmadik halmazt is az $\mathcal{S}$-beli $\mathcal{S}_F$ elfogadó állapotokkal.

Ezekre a halmazokra is kiterjeszthető a felhasznált megfigyelés. Ha létezik olyan kör, ami érint F-beli állapotot és tartalmaz $\mathcal{N}_{\mathcal{E}_l}$-beli átmenetet, akkor $\mathcal{S}^- \leftarrow [\mathcal{S}^- \cap \mathcal{N}^*(\mathcal{S}_F \cap \mathcal{N}^*(S^+ \cap \mathcal{N}_{\mathcal{E}_l}(S^-)))]$ műveletnek van nem $\emptyset$ fixpontja $\mathcal{S}^-$-ra nézve, vagyis $\mathcal{S}^-$-ban van olyan állapot, ami egy $\mathcal{N}_{\mathcal{E}_l}$-beli állapotátmenettel egyet lépve majd elfogadó állapotot érintve önmagából elérhető. Ugyanez igaz $\mathcal{S}^+$-ra és $\mathcal{S}_F$-re is a művelet megfelelő módosításával. A $\mathcal{N}_{\mathcal{E}_l}$ egyszeri alkalmazása garantálja, hogy a műveletek a nemtriviálisan elérhető állapotokat állítják elő, így a többi helyen $\mathcal{N}^*$ használható, amit a szaturációs algoritmussal hatékonyan számíthatunk.



3.5. ábra. Elfogadó erősen összefüggő komponensek inkrementális keresése.

Az ebből adódó, a halmazokat ciklikusan szűkítő megoldás pszeudokódja a 4. algoritmus mutatja be. Itt a fenti három fixpontot párhuzamosan és iteratívan számítva az algoritmus addig szűkíti $\mathcal{S}^+$-t, $\mathcal{S}^-$-t és $\mathcal{S}_F$-et, amíg valamelyik el nem fogy, vagy a szűkítés során nem változik. Ha egy halmaz elfogyott, az bizonyítja, hogy nincs a feltételeknek megfelelő kör $\mathcal{S}$-ben, míg a másik esetben a halmazokban maradt állapotok ilyen körökhöz tartoznak [12]. A szűkítő lépések halmazokkal szemléltetve a 3.5. ábrán láthatók.

Algoritmus 4 Elfogadó erősen összefüggő komponensek inkrementális keresése

```
1:  function DetectCircles(S, F, N_{E_l})
2:      S⁺ ← N(S)_{E_l};
3:      S⁻ ← N⁻(S)_{E_l};
4:      S_F ← F;
5:      if S_F = ∅ then return false;
6:      repeat
7:          S⁻ ← S⁻ ∩ N*(S_F);
8:          S⁺ ← S⁺ ∩ N_{E_l}(S⁻);
9:          S_F ← S_F ∩ N*(S⁺);
10:     until S⁺ és S⁻ és S_F nem változik tovább;
11:     if S_F = ∅ then
12:         return False;
13:     else
14:         return True;
15:     end if
16: end function
```

3.4. Értékelés

Korábbi kutatásaink eredményeképp elsőként javasoltunk szaturáció alapú algoritmust a lineáris idejű modellellenőrzés problémájára, amely – mint a fejezetben bemutattuk – szimbolikus ábrázolás mellett is inkrementálisan volt képes „on-the-fly” működésre. Megoldásunk újszerű volt mind a szorzat állapottér felderítését, mind a körkeresést tekintve,

de éppen ezért algoritmusunk sok szempontból kezdetleges volt, és inkább elméleti eredménynek volt tekinthető.

A szorzat állapottér felderítésére használt algoritmusunk legnagyobb hátránya az volt, hogy előírta a specifikációs automaták állapotaira vonatkozóan a bemenő élek azonos címkézését. Ez – bár a megfelelő állapotok szétbontásával minden automata ilyen alakúra hozható – gátat szabott a speciális automaták egyszerűsítésének. Utóbbi viszont rendkívül kívánatos lenne, mivel a szorzat állapottér mérete lineáris a specifikációs automata méretében. Ráadásul a modell állapottere általában hatalmas, a pontos felső korlát pedig ennek az állapottérnek a mérete szorozva a specifikációs automata állapotainak számával, így minden specifikációs automata állapot potenciálisan a modell állapotterének egy újbóli bejárását jelentheti.

A körkeresés a tapasztalatok szerint sok esetben kiválóan teljesített, nem teljesülő kifejezések esetén az „on-the-fly” működés miatt nagyon gyorsan talált ellenpéldát, sokszor csak néhány állapot felderítésével. Nagyon gyengén teljesített viszont olyan esetekben, amikor az állapottér nem tartalmazott kört, de sok elfogadó állapotot tartalmazott, és „hosszú” irányított körmentes részekből állt, amelyek több szinthez tartozó eseményeket felváltva használtak. Az ilyen esetekben az algoritmus rengetegszer kezdett „feleslegesen” körkeresésbe, egyre növekvő halmazokat szűkítve lépésenként az üres halmazra.

Az 5.2.1. fejezetben azokat az algoritmikus fejlesztéseinket fogjuk bemutatni, amik a korábbi eredmények alapjain, de sokszor egészen új megközelítéssel ezeket a gyengeségeket hivatottak kiküszöbölni. Az általános automaták kezelését a következő, 4. fejezetben bemutatott automatagenerálási és egyszerűsítési eredményeink motiválják. A 6. fejezetben meg fogjuk mutatni az új algoritmusok által a korábbi megoldásokhoz képest elért hatékonyságbeli fejlődést.

4. fejezet

Követelmények formalizálása és optimalizálása

Ebben a fejezetben a munkánk követelményspecifikus részét mutatjuk be, vagyis azt, hogy hogyan tudjuk hatékonyan felhasználni a PLTL speicifkációs nyelvet a modellellenőrzés során. Mint korábban bemutattuk, a modellellenőrző algoruitmust nem közvetlenül a temporális kifejezéseken dolgozik, hanem az abból generált Büchi automatán [4]. Ebben a fejezetben bemutatunk a Büchi automaták generálására egy hatékony algoritmust, majd ezt későbbiekben egyszerűsítő algoritmusokkal egészítjük ki, hogy a szorzat állapottér méretének csökkentésével növeljük a modellellenőrzés hatékonyságát. Fontos kiemelni, hogy az itt befektetett idő minden későbbi modellellenőrzési ciklusban megtérül [5], mivel az egyszerűsített automatát tetszőlegesen sokszor felhasználhatjuk a folyamatosan javított modell újbóli ellenőrzésére.

4.1. Múlt- és jövőidejű temporális logikák transzformálása Büchi automatákká

Habár a PLTL kifejezések akár exponenciálisabban tömörebbek lehetnek az ekvivalens LTL megfogalmazásoknál, fontos kérdés, hogy hatékonyan tudunk-e előállítani belőle Büchi automatákat. Szerencsére a transzformáció algoritmikus komplexitása nem nő meg a múlt idős operátorok bevezetésével, a probléma mindkét logika esetén a PSPACE-nehéz halmazba esik. [8, 9]

Az alábbiakban egy a PLTL kifejezésekből úgynevezett *temporális tablót* létrehozó algoritmust [7], majd ennek a hatékonyabb, továbbfejlesztett változatát mutatjuk be.

Elemi és összetett kifejezések

Egy kifejezést *elemi kifejezésnek* nevezünk, ha az *propozíció*, *True*, *False*, $\boldsymbol{X}p$, $\tilde{\boldsymbol{X}}p$, $\boldsymbol{P}p$, vagy $\tilde{\boldsymbol{P}}p$ alakú, különben *összetett kifejezésnek* nevezzük.

Minden összetett kifejezés szétbontható az operátorok szemantikája alapján egy, vagy két *előfeltételre*, amik kifejezések halmazai. Az összetett kifejezés pontosan akkor teljesül, ha az előfeltételek közül legalább az egyikben szereplő minden kifejezés teljesül. Ezeket az előfeltételeket elegendő a negált normál formában szereplő operátorokra bevezetni, hiszen minden formula ilyen alakra hozható, és az algoritmus végig ilyen kifejezésekkel dolgozik.

Az egyes operátorokhoz tartozó felbontásokat a következő táblázat tartalmazza:

Kifejezés	pre_1	pre_2
$p \wedge q$	$\{p, q\}$	$-$
$p \vee q$	$\{p\}$	$\{q\}$
$p\ \mathbf{U}\ q$	$\{q\}$	$\{p, \mathbf{X}(p\ \mathbf{U}\ q)\}$
$p\ \mathbf{R}\ q$	$\{p, q\}$	$\{q, \tilde{\mathbf{X}}(p\ \mathbf{R}\ q)\}$
$p\ \mathbf{S}\ q$	$\{q\}$	$\{p, \mathbf{P}(p\ \mathbf{S}\ q)\}$
$p\ \mathbf{B}\ q$	$\{p, q\}$	$\{q, \tilde{\mathbf{P}}(p\ \mathbf{B}\ q)\}$

Atomok és azok fedése

Definíció. *Azt mondjuk, hogy kifejezések egy S halmaza* lokálisan konzisztens*, ha nem szerepelnek benne ellentmondó kifejezések, azaz kielégíti a következőket:*

- *False* $\notin S$.
- *Ha* $\neg p \in S$*, akkor* $p \notin S$.
- *Ha* $\tilde{\boldsymbol{P}}$ *False* $\in S$*, akkor* $\boldsymbol{P}\ p \notin S$ *bármely* p *kifejezésre.*

5. példa. *Lokálisan konzisztensek az alábbi halmazok:*

- $S_1 = \{a, \boldsymbol{X}\ a\}$
- $S_2 = \{a, \tilde{\boldsymbol{P}}\ \mathit{False}, \tilde{\boldsymbol{P}}\ a\}$
- $S_3 = \{a, a\ \boldsymbol{U}\ b, \boldsymbol{O}\ b\}$

Ezzel szemben a következő halmazok nem lokálisan konzisztensek:

- $S_4 = \{a, \boldsymbol{X}\ a, \mathit{False}\}$
- $S_5 = \{a, \tilde{\boldsymbol{P}}\ \mathit{False}, \boldsymbol{P}\ a\}$
- $S_6 = \{a, a\ \boldsymbol{U}\ b, \neg a\ \boldsymbol{R}\ \neg b\}$

Definíció. *Legyen* φ *egy kifejezés. Ekkor* φ lezártja $CL(\varphi)$*, kifejezések legkisebb olyan halmaza, melyre teljesülnek a következők:*

- $\tilde{\boldsymbol{P}}$ *False* $\in CL(\varphi)$.
- $p \in CL(\varphi)$ *pontosan akkor, ha* $\neg p \in CL(\varphi)$.
- *Ha* $\boldsymbol{X}\ p \in CL(\varphi)$ *vagy* $\boldsymbol{P}\ p \in CL(\varphi)$*, akkor* $p \in CL(\varphi)$.
- *Ha egy* p *összetett kifejezés, és* $p \in CL(\varphi)$*, akkor minden* q *kifejezésre, ami szerepel* p *előfeltételeiben,* $q \in CL(\varphi)$.

6. példa. *A* $\varphi = a\ \boldsymbol{U}\ b$ *kifejezés lezártja például az alábbi halmaz:*

$$CL(\varphi) = \{\tilde{\boldsymbol{P}}\ \mathit{False}, \boldsymbol{P}\ \mathit{True}, a\ \boldsymbol{U}\ b, \neg a\ \boldsymbol{R}\ \neg b, a, \neg a, b, \neg b, \boldsymbol{X}(a\ \boldsymbol{U}\ b), \tilde{\boldsymbol{X}}(\neg a\ \boldsymbol{R}\ \neg b)\}$$

Definíció. φ-atomnak *nevezzük* φ *lezártjának egy olyan* A *részhalmazát, ami nem tartalmaz ellentmondást, és egy összetett kifejezés pontosan akkor szerepel benne, ha valamely előfeltétele teljesül. Azaz* A*-ra teljesülnek a következők:*

- A *lokálisan konzisztens.*
- *Ha* $p \in CL(\varphi)$ *egy összetett kifejezés, akkor* $p \in A$ *pontosan akkor, ha* $pre_i \subseteq A$ p *legalább egy* pre_i *előfeltételére.*

7. példa. *Az előző példánál maradva a $\varphi = a \ \mathbf{U} \ b$ kifejezéshez tartozó egyik lehetséges φ-atom a következő:*

$$A_\varphi = \{a \ \mathbf{U} \ b, b\}$$

Ez a halmaz φ-atom, mert lokálisan konzisztens, tartalmazza egyrészt a φ kifejezést, másrészt a benne szereplő egyetlen összetett kifejezés ($a \ \mathbf{U} \ b$) egyik előfeltétel (pre_1) minden elemét.

Definíció. *Azt mondjuk, hogy egy A φ-atom egy B φ-atom* általánosítása *(jelölés: $A \sqsubseteq B$), ha $A \subseteq B$ és φ lezártjában szereplő összes $p \ \mathbf{U} \ q$ alakú kifejezésre teljesül, hogyha $p \ \mathbf{U} \ q \in A$ (ezáltal $p \ \mathbf{U} \ q \in B$) és $q \in B$, akkor $q \in A$ is teljesül. Szemléletesebben ha B-ben kielégíthető $p \ \mathbf{U} \ q$ a q teljesülésével, akkor A-ban is.*

Megjegyzendő, hogy mivel B több kifejezést tartalmaz, mint A, ezért specifikusabb olyan értelemben, hogy a több kifejezés teljesülésének megkövetelése miatt kevesebb lehetséges esetet fed le, mint A. A atom egy B atom *szigorú általánosítása* (jelölés: $A \sqsubset B$), ha $A \sqsubseteq B$ és $A \neq B$ (azaz $A \subset B$).

8. példa. *Az alábbi B_φ atomra teljesül, hogy az előző példában szereplő $A_\varphi \subseteq B_\varphi$ annak általánosítása, hiszen az egyetlen $p \ \mathbf{U} \ q \in A_\varphi$ alakú kifejezésre teljesül, hogy $q \in B_\varphi$ és $q \in A_\varphi$.*

$$B_\varphi = \{a \ \mathbf{U} \ b, b, a, \mathbf{X}(a \ \mathbf{U} \ b)\}$$

Definíció. *φ-atomok egy $\{A_1, A_2, \ldots, A_k\}$ halmazát az $S \subseteq CL(\varphi)$ kifejezés-halmaz* (teljes) fedésének *nevezzük, ha:*

- *minden A_i atom tartalmazza S-t, azaz $S \subseteq A_i$ teljesül minden i-re.*
- *minden S-et tartalmazó Y φ-atomot általánosít valamelyik A_i atom.*

Ha egy fedésre igaz, hogy $A_i \not\sqsubset A_j$ bármely i-re és j-re, akkor azt *minimális fedésnek* nevezzük.

9. példa. *A $\varphi = a \ \mathbf{U} \ b$ kifejezés esetén $S = \{a \ \mathbf{U} \ b\} \subseteq CL(\varphi)$ kifejezéshalmaz egy lehetséges fedése a $Cover(S) = \{A_1, A_2\}$ halmaz, ahol*

$$A_1 = \{a \ \mathbf{U} \ b, b\}$$

$$A_2 = \{a \ \mathbf{U} \ b, a, \mathbf{X}(a \ \mathbf{U} \ b)\}$$

Definíció. *φ-atomok egy $\{A_1, A_2, \ldots, A_k\}$ halmazát egy B φ-atom és egy $S \subseteq CL(\varphi)$ kifejezés-halmaz* inkrementális fedésének *nevezzük, ha:*

- *Minden A_i atom tartalmazza az $S \cup B$ halmazt, azaz $(S \cup B) \subseteq A_i$ minden i-re.*
- *Minden Y φ-atomhoz, melyre teljesül, hogy $S \subseteq Y$ és $B \sqsubseteq Y$ (azaz minden olyan atomhoz, ami S-et tartalmazza és aminek B általánosítása), létezik olyan A_i atom, melyre $B \sqsubseteq A_i \sqsubseteq Y$.*

Vegyük észre, hogy az inkrementális fedés annyiban különbözik a teljes fedéstől, hogy a fedett halmaz (azaz $S \cup B$) egy részhalmaza maga is egy atom (B), és csak azoknak az Y atomoknak kell teljesíteniük az $A_i \sqsubseteq Y$ (és a $B \sqsubseteq A_i$) relációt, melyekre $B \sqsubseteq Y$ is teljesül. Emellett egy üres O atom és egy S kifejezéshalmaz inkrementális fedése meg fog egyezni az S halmaz teljes fedésével, hiszen ekkor $O \sqsubseteq Y$ (és $O \sqsubseteq A_i$) minden Y-ra (és A_i atomra) teljesül.

A bemutatott algoritmus működése során (ahogy azt [7]-ban megmutatták) elegendő inkrementális fedést számolni.

A tabló algoritmus

Egy atomot *kezdeti atomnak* nevezünk, ha tartalmazza a $\tilde{\mathbf{P}}\mathit{False}$ kifejezést, ugyanis ez csak akkor teljesülhet, ha nem létezik megelőző állapot.

Legyen S kifejezések egy halmaza. Ekkor definiáljuk a következő halmazokat:

- $\mathit{Next}(S) = \{p \mid \mathbf{X}p \in S\} \cup \{p \mid \tilde{\mathbf{X}}p \in S\}$.
- $\mathit{Prev}(S) = \{p \mid \mathbf{P}p \in S\} \cup \{p \mid \tilde{\mathbf{P}}p \in S\}$.

Egy A és egy B atom esetén az (A, B) pár *szomszédosan konzisztens*, ha $\mathit{Next}(A) \subseteq B$ és $\mathit{Prev}(B) \subseteq A$.

Az algoritmus egy gráfot készít és tart karban, melynek csúcsai φ-atomok, amiket irányított élek kötnek össze. A csúcsokra úgy is gondolhatunk, mint olyan állapotok, ahol teljesülnek az adott φ-atomban szereplő kifejezések. Egy $\langle A, B\rangle$ irányított él létezése azt jelenti, hogy B az A-t (időben) követő állapot lehet. Egy ilyen irányított élt *jövő-kielégítőnek* nevezünk, ha $\mathit{Next}(A) \subseteq B$, azaz B tartalmazza az A állapotban megfogalmazott, a következő állapotban teljesítendő kifejezéseket, illetve ehhez hasonlóan *múlt-kielégítőnek* nevezzük az élt, ha $\mathit{Prev}(B) \subseteq A$, azaz A tartalmazza a B állapotban megfogalmazott, az előző állapotban teljesítendő kifejezéseket. Ha egy él jövő- és múlt-kielégítő egyszerre, akkor azt mondjuk, hogy az él *kielégítő*, ami azt is jelenti, hogy A és B szomszédosan konzisztensek. Az algoritmus célja, hogy egy kezdeti gráfból kiindulva a nem kielégítő éleket kielégítő élekre cserélje, miközben új csúcsokat hoz létre, melyekkel ez teljesíthető.

Az algoritmus két fő részből áll. Az első fázisban egy adott φ kifejezéshez elkészítünk egy $G(\mathcal{V}, \mathcal{E}, \bar{\mathcal{E}})$ gráfot, ahol $\mathcal{V}$ jelöli a csúcsok halmazát, $\mathcal{E}$ az élek halmazát, míg $\bar{\mathcal{E}}$ a törölt élek halmazát. Az összes eddig létrehozott él halmazát, azaz az $\mathcal{E} \cup \bar{\mathcal{E}}$ halmazt $\mathcal{E}^+$-nal jelöljük. A kezdeti gráfban $\mathcal{V}$ kezdeti φ-atomok egy halmazából (melyek tartalmazzák a $\tilde{\mathbf{P}}\mathit{false}$ és φ kifejezéseket), és egy egyetlen kifejezést sem tartalmazó általános jövő-atomból (F) áll. A gráf minden csúcsából irányított él mutat F-be, a törölt élek halmaza pedig üres.

A második fázis abból áll, hogy amíg létezik olyan $\langle A, B\rangle$ él, ami nem kielégítő, egy új atomot hozunk létre, ami kiegészíti A-t, vagy B-t további kifejezésekkel úgy, hogy az él az adott irányban kielégítővé váljon. Ha már létezett a szükséges atom, akkor nem hozunk létre új atomot. A nem kielégítő élt helyettesítjük egy éllel, ami erre az atomra illeszkedik, az eredeti élt pedig áthelyezzük a törölt élek közé $\bar{\mathcal{E}}$-ba. Ezt addig ismételjük, amíg minden $\mathcal{E}$-ban szereplő él kielégítő lesz.

Ha például találunk egy $\langle A, B\rangle$ nem jövő-kielégítő élt, akkor a B atomot bővítjük azokkal a kifejezésekkel, amik szükségesek az él kielégítővé tételéhez, azaz $\mathit{Next}(A)$ elemeivel, majd vesszük a kapott kifejezéshalmaz teljes fedését, az új atomokat tartalmazó α-t. Ez után az α-ban szereplő összes B' atomot felvesszük a G gráf csúcsai közé, amennyiben azok még nem szerepeltek $\mathcal{V}$-ben, majd behúzzuk az új $\langle A, B'\rangle$ éleket. Ezt követően már csak annyit kell megtenni, hogy minden $\langle B, Y\rangle \in \mathcal{E}^+$ élhez felveszünk egy annak megfelelő $\langle B', Y\rangle$ élt $\mathcal{E}$-ba, majd az eredeti $\langle A, B\rangle$ élt áthelyezzük a törölt élek közé $\bar{\mathcal{E}}$-ba. Ily módon bármely eddigi irányított út, amely áthaladt az A, B, Y csúcsokon, továbbra is megmarad az A, B', Y csúcsokon keresztül. Az algoritmus ezzel analóg módon kezeli a nem múlt-kielégítő éleket is.

Egy új $\langle X, Y\rangle$ él hozzáadása ebben a fázisban úgy történik, hogy először megvizsgáljuk, hogy az él szerepelt-e már valamikor a gráfban, azaz eleme-e $\mathcal{E}^+$-nak. Ha még nem szerepelt, akkor felvesszük $\mathcal{E}$-ba, majd a hozzá kapcsolódó $\langle W, X\rangle$, $\langle Y, Z\rangle \in \bar{\mathcal{E}}$ törölt éleket visszahelyezzük $\mathcal{E}$-ba. Az élek ismételt felvétele után ezek az élek nyilván újra törlésre kerülnek majd, hiszen még mindig nem szomszédosan konzisztens atomok között futnak, így ez a lépés azért szükséges, hogy az él következő feldolgozásakor a most létrehozott $\langle X, Y\rangle$ élt felvegyük X és Y esetleges kiterjesztéseire illeszkedően is.

Fejlesztett tabló algoritmus

Az előző fejezetben ismertetett algoritmus helyes és teljes [7], viszont néhány ponton javítani lehet a teljesítményén:

- Élek feltétel nélküli öröklése: Amikor egy $\langle X, Y \rangle$ nem jövő-kielégítő élt javítunk, Y minden Y' kiterjesztéséhez felvesszük az összes $\langle Y, Z \rangle$ él módosított másolatát, azaz a $\langle Y', Z \rangle$ élt. Ilyenkor azt mondjuk, hogy Y' *jövő-örökölte* az $\langle Y, Z \rangle$ élt. Amikor öröklés történik, az adott Y' atom minden $\langle Y, Z \rangle$ élt örökölni fog $\mathcal{E}$-ból és $\bar{\mathcal{E}}$-ból is. Ennek a feltétel nélküli öröklésnek a hátránya, hogy redundáns atomokat és éleket hoz létre. Hasonlóan, amikor egy $\langle X, Y \rangle$ nem múlt-kielégítő él esetén minden X' atom *múlt-örökli* az összes $\langle W, X \rangle$ élt, felmerül ugyanez a probléma.
- Törölt élek ismételt felvétele: Az algoritmus új élek felvételekor az öröklések biztosítása végett bizonyos éleket visszahelyez a törölt élek halmazából, azaz $\bar{\mathcal{E}}$-ból az aktuális élek halmazába, $\mathcal{E}$-ba. Az ilyen ismételten felvett élek nem különböztethetők meg a többi nem kielégítő éltől, melyek még nem voltak soha feldolgozva, így az ismételt feldolgozásukkor a legtöbb műveletet mégegyszer feleslegesen hajtjuk végre.

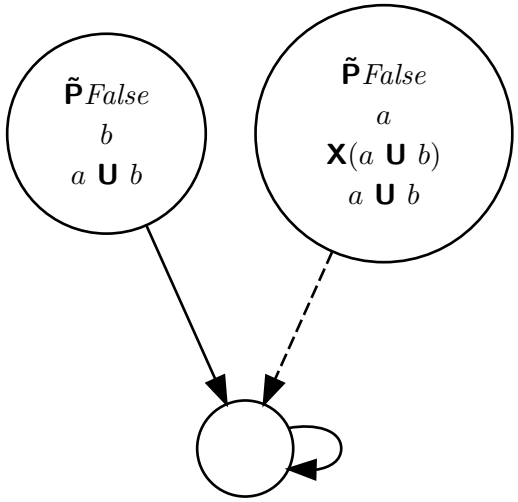
A fejlesztett algoritmus a fenti két problémát hivatott megoldani az itt bemutatott módosításokkal:

- Klón listák: Minden $A \in \mathcal{V}$ atomhoz karbantartunk két klón listát, a *jövő-klónokat* és a *múlt-klónokat*. Amikor egy új atomot hozunk létre, a hozzá tartozó klón listákat üresen inicializáljuk. Amikor egy $\langle A, B \rangle$ élt javítunk jövő irányban, minden $B' \in \mathit{Cover}(B \cup \mathit{Future}(A))$ atomot felveszünk B jövő-klónjai közé. A múlt irányú javításnál hasonlóan járunk el, minden $A' \in \mathit{Cover}(A \cup \mathit{Past}(B))$ atomot felveszünk A múlt-klónjai közé.
- Feltételes öröklés: Amikor egy $\langle A, B \rangle$ nem jövő-kielégítő élet javítunk, akkor B minden B' kiterjesztése csak akkor örököl egy $\langle B, Q \rangle \in \mathcal{E}$ élt, ha az adott él jövő-kielégítő. Hasonlóan egy nem múlt-kielégítő él javításánál csak múlt-kielégítő élek öröklődnek. Ha egy él egyik irányban sem kielégítő, akkor valamelyik irányban viszont öröklődnie kell. Ezen kívül amikor egy új, valamilyen irányban kielégítő $\langle A, B \rangle$ élt felveszünk, akkor az adott irányban öröklődik rekurzívan, az iránytól függően A, vagy B megfelelő klón listáján keresztül.

A javítások megoldják az eredeti algoritmus esetén fennálló problémákat, ugyanis:

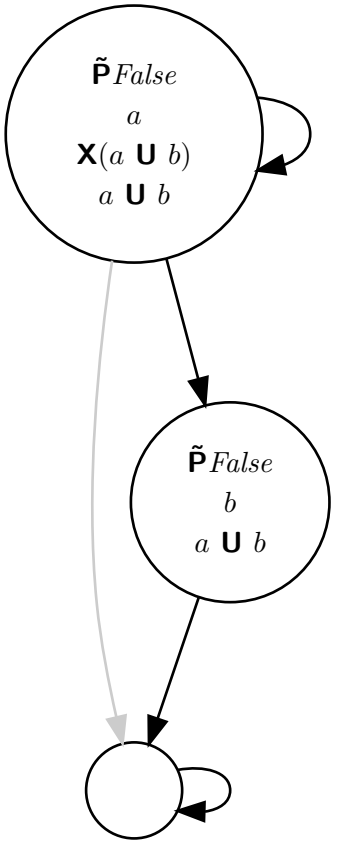
- Az élek feltétel nélküli öröklésből adódó probléma megoldódik azzal, hogy egy él javításánál csak az adott irányban kielégítő élek öröklődnek. Ezzel ugyanis azt érjük el, hogy nem a hibás (azaz nem kielégítő) élt másoljuk le, ami után minden példányt külön-külön fel kéne dolgozni, hanem előbb elvégezzük az él javítását, majd utólag a klónlistákon keresztül öröklődik a már kielégítő él. Az egyik irányban sem kielégítő élek valamilyen irányban történő öröklésére azért van szükség, mert ha nem öröklődne egyik irányban sem, akkor az él valamilyen irányban történő javításakor csak az adott irányú klónok örökölnék az élt, majd az eredeti él törlésre kerülne. Ez azért probléma, mert így a másik irányú klónok sosem örökölhetnék az élt, mert a már javított élekre illeszkedő csúcsok azokat már nem tartalmazzák a klón listáikban.
- A törölt élek ismételt felvételének problémája is megoldódik a klón listák bevezetésével és az ezeken keresztül rekurzívan történő örökléssel. A klón listákon keresztül a létrehozott élt minden szükséges klón atom (azaz az eredeti atom kiterjesztései) örökli, de mivel pont ezt szerettük volna elérni az élek ismételt felvételével, ezért az teljesen feleslegessé válik, így megszabadulhatunk a redundáns lépésektől.

10. példa. *Az alábbiakban az algoritmus futása látható egy egyszerű esetben, melyre a két algoritmus által végrehajtott lépések megegyeznek. A feldolgozott kifejezés* $\varphi = a \ \boldsymbol{U} \ b$. *A 4.1. ábrán látható az algoritmus által generált kezdeti gráf, melyen egyrészt szerepelnek a kifejezés fedéséhez szükséges atomok, másrészt az általános jövő-atom, mely egyetlen kifejezést sem tartalmaz. Az egyetlen nem kielégítő él szaggatott nyíllal van jelölve.*



4.1. ábra. Az algoritmus által előállított kezdeti gráf.

Az algoritmus ezt az élt fogja először javítani, ezzel létrehozva a 4.2. ábrán látható állapotot. Ebben a lépésben a megfelelő kifejezéshalmaz fedésében szereplő atomokba húzunk be új éleket. Mivel a fedésben szereplő összes atom megtalálható volt már a gráfban, ezért új csúcsot nem hoztunk létre. A javított él ezután törlésre kerül, az ábrán szürkén van feltüntetve. A lépés befejeztével a gráfban nincs több nem kielégítő él, így az algoritmus futása leáll.



4.2. ábra. A gráf az él javítása után.

Specifikációs automata készítése tablóból

Az algoritmus által készített $G(\mathcal{V}, \mathcal{E}, \bar{\mathcal{E}})$ gráf a következő módon konvertálható általánosított Büchi automatává:

- A Σ ábécé a φ specifikációs kifejezésben szereplő atomi propozíciók halmazából 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 Boole-algebrai kifejezésekkel több ilyen α együttesen is reprezentálható.
- Az automata Q állapothalmaza a G gráfban található atomokból ($\mathcal{V}$), valamint egy *init* csomópontból áll.
- $(A, \alpha, B) \in \Delta$ (ahol $A, B \in \mathcal{V}$) akkor és csak akkor, ha α igazzá teszi a B-ben lévő *ponált és negált atomi propozíciók konjunkcióját*, és ezen kívül $\langle A, B \rangle \in \mathcal{E}$, vagy $A = \mathit{init}$ és $\tilde{\mathbf{P}}$ *False* $\in B$.
- A Q^0 kezdőállapothalmaz egyetlen eleme az *init* csomópont.
- Az F elfogadó komponens minden $\mu\ \mathbf{U}\ \psi$ alakú részkifejezéshez tartalmaz egy $P_i \in F$ állapothalmazt – P_i minden olyan A állapotot tartalmaz, amire $\psi \in A$ vagy $\mu\ \mathbf{U}\ \psi \notin A$ 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 $\mu\ \mathbf{U}\ \psi$ alakú részkifejezés, akkor F üres lesz, ami – mint azt a 2.2.3 fejezetben bemutattuk – hagyományos automaták esetén az $F = Q$ elfogadó állapot halmaznak felel meg.

4.2. A specifikációs automata egyszerűsítése

A tabló algoritmussal létrehozott Büchi automaták különleges tulajdonsága, hogy azokban az azonos állapotokba mutató átmeneteken azonos kifejezések szerepelnek. Ez a tulajdonság abból ered, hogy a transzformáció során létrehozott gráfban a csúcsokhoz rendelünk kifejezéshalmazokat, majd később minden bejövő élt az ilyen halmazok kielégítéséhez szükséges atomi kijelentésekkel címkézünk. Ezt a tulajdonságot a korábbi munkáinkban szereplő algoritmusok még kihasználták, viszont az 5.1 fejezetben leírt módosítások lehetővé teszik általános Büchi automaták használatát is. Mivel a fent ismertetett tulajdonság jelentős redundanciát okozhat egy automatában, és az automaták mérete nagy mértékben befolyásolja a modellellenőrzés sebességét, felmerült az igény az automaták oly módon történő egyszerűsítésére, ami nem feltétlenül tartja meg ezt a tulajdonságot.

Az itt bemutatott algoritmus azon az észrevételen alapul, hogy az olyan állapotok, melyekből azonos bemenetekre azonos állapotokba vezetnek átmenetek valójában helyettesíthetőek egyetlen állapottal, melybe az eredeti állapotokba mutató minden átmenet be van húzva, hiszen az ezzel az átlakítással létrehozott új automata és az eredeti automata által elfogadott nyelvek megegyeznek.

A továbbiakban tekintsük az $\mathcal{A} = \langle \Sigma, Q, \Delta, Q^0, F \rangle$ Büchi automatát, ahol $p, q \in Q$.

Jelöljük a q állapotba mutató állapotátmenetek halmazát $\bullet q$-val, a kimenő állapotátmenetek halmazát pedig $q\bullet$-val.

Hasonlónak nevezünk egy $t = (q, a, q')$ és egy $u = (p, b, p')$ állapotátmenet, ha $a = b$ és $q' = p'$, azaz az állapotátmenetek csak a kiindulóállapotjaikban különböznek egymástól.

Azt mondjuk, hogy p és q *hasonló* állapotok, ha $|p \bullet| = |q \bullet|$ és minden $t \in p\bullet$ állapotátmenethez létezik egy $u \in q\bullet$ állapotátmenet, amire teljesül, hogy t és u hasonló állapotátmenetek.

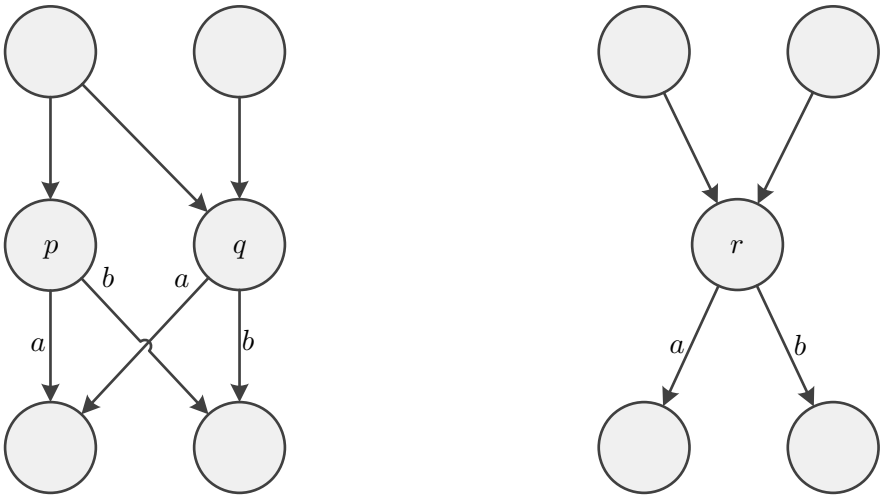
Egy p és egy q állapot *összevonható*, azaz $p \simeq q$, ha hasonló, és $p \in F \Leftrightarrow q \in F$.

Az automata egyszerűsítő algoritmus az i-edik iterációban az A_i automatában megkeresi az összevonható $p, q \in Q_i$ állapotokat, és létrehoz egy $\mathcal{A}_{i+1}$ Büchi automatát az alábbi módon:

- Törli az állapotok halmazából a p és q állapotokat, és felvesz helyettük egy új r állapotot.
- Minden $(w, a, x) \in \Delta_i$ állapotátmenet helyett felvesz egy (w, a, r) átmenetet, ahol $x \in \{p, q\}$, $w \in Q_i$ és $a \in \Sigma$.
- Minden (p, a, w) állapotátmenet helyett felvesz egy (r, a, w) átmenetet.

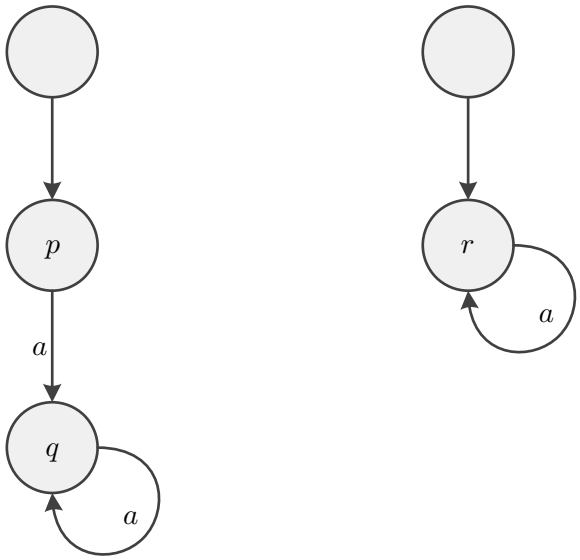
Ha nem talál újabb összevonható állapotokat, az algoritmus leáll.

Ahogy a 4.3. ábrán is látszik, ez szemléletesen azt jelenti, hogy a p és q állapotokat úgy vonja össze, hogy azok be- és kimenő állapotátmeneteit átirányítja az új r állapotba.



4.3. ábra. Állapotok összevonása egyszerű esetben.

Érdemes megfigyelni, hogy ha az összevonás következtében két állapot között több ugyanolyan átmenet jönne létre, akkor csak az egyiket tartjuk meg (hiszen Δ egy halmaz, amiben nem szereplhet többször egy (p, t, q) átmenet). Egy első ránézésre kevésbé egyértelmű esetet láthatunk a 4.4. ábrán, ahol a q állapotból egy önmagába mutató átmenet indul ki. A szabályt természetesen ilyenkor is alkalmazhatjuk, hiszen p-ből és q-ból így is ugyanoda mutató, azonos címkéjű átmenetek vezetnek ki.



4.4. ábra. Állapotok összevonása hurokél esetén.

Az algoritmus kevesebb, mint $|Q|$ lépés után leáll, hiszen minden lépésben eggyel csökken az állapotok száma, és ha egy automata csak egy állapottal rendelkezik, akkor abban biztos nincsenek összevonható állapotok.

Az algoritmus helyességének bizonyításához az alábbi két állítást kell belátni:

- Minden olyan $v \in \Sigma^\omega$ szót, amit az A_i automata elfogadott, az A_{i+1} automata is elfogad.
- Ha egy $v \in \Sigma^\omega$ szót az A_i automata nem fogad el, akkor azt az A_{i+1} automata sem fogadja el.

Ezekből ugyanis az következik, hogy az algoritmus végén megkapott automata pontosan azokat a szavakat fogadja el, melyeket az eredeti automata is, tehát a két automata által elfogadott nyelv ekvivalens, azaz az algoritmus helyes.

Egy szót az automata pontosan akkor fogad el, ha létezik hozzá elfogadó lefutás. Ha egy v szót elfogad az automata, két esetet különböztethetünk meg aszerint, hogy az (összes) elfogadó lefutás áthalad-e az összevont állapotokon. Ha az A_i automatában létezik v-hez olyan lefutás, ami nem tartalmazza sem a p sem a q állapotot, akkor ez a lefutás az A_i automatában is létezik, és ott is elfogadó. Amennyiben az A_i automatában a v szót elfogadó összes lefutás tartalmazza a p, vagy q állapotok valamelyikét, tekintsünk egy ilyen ρ lefutást. Ekkor az A_{i+1} automatában létezik ρ' elfogadó lefutás a v szóhoz. Tegyük fel, hogy a lefutás k-adik eleme valamely törölt állapot, azaz $\rho(k) \in \{p, q\}$. Ekkor az A_{i+1} automatában létezik olyan ρ' elfogadó lefutás, aminek a k-adik eleme az új r állapot, ugyanis ha A_i automatában szerepelt például egy $(\rho(k-1), a, p) \in \Delta_i$ és egy $(p, b, \rho(k+1)) \in \Delta_i$ átmenet, akkor az A_{i+1} automatában szerepelnek a $(\rho(k-1), a, r) \in \Delta_{i+1}$ és $(r, b, \rho(k+1)) \in \Delta_{i+1}$ (hasonlóan q-ra is igaz), azaz a ρ lefutás k-adik állapotát r-re cserélve egy helyes ρ' lefutást kapunk A_{i+1}-ben. Fontos továbbá, hogy a lefutás elfogadó marad, mert ha p és q elfogadó állapotok voltak, akkor r is az, egyébként pedig nem változtattunk az elfogadó állapotokon a lefutásban.

Ha egy v szót nem fogad el az A_i automata, akkor minden hozzá tartozó lefutásra igaz, hogy az nem elfogadó. Ahhoz, hogy belássuk, hogy az A_{i+1} automatában sincs v-hez elfogadó lefutás elég azt megmutatnunk, hogy minden A_{i+1}-beli ρ' elfogadó lefutáshoz létezik A_i-beli ρ elfogadó lefutás, hiszen ebből következik, hogy ha nem létezik A_i-ben ilyen lefutás, akkor A_{i+1}-ben sem létezhet. Ezt viszont könnyen beláthatjuk, ugyanis ha az A_{i+1} automatában létezik v-hez egy ρ' elfogadó lefutás, akkor ahhoz tudunk egy ρ elfogadó lefutást mutatni az A_i automatában. Ha ρ'-ben nem szerepel sehol az r állapot, akkor a $\rho = \rho'$ lefutás megtalálható A_i-ben. Amennyiben r szerepel ρ'-ben k-adik állapotként, helyettesíthetjük azt p vagy q állapottal, mert ha A_{i+1}-ben szerepel egy $(\rho'(k-1), a, r) \in \Delta_{i+1}$ és egy $(r, b, \rho'(k+1)) \in \Delta_{i+1}$ állapotátmenet, akkor az A_i automatában az algoritmus működéséből következően szerepel egy $(\rho'(k-1), a, x) \in \Delta_i$ és egy $(x, b, \rho'(k)) \in \Delta_i$ állapotátmenet, ahol $x \in \{p, q\}$. Az ilyen cseréket végrehajtva a teljes lefutáson kapunk egy ρ lefutást a v szóhoz az A_i automatában, ami pontosan akkor elfogadó, ha ρ' is az volt A_{i+1}-ben.

5. fejezet

Új algoritmusok ω-reguláris modellellenőrzésre

Ebben a fejezetben az új modellellenőrző algoritmusunk részleteit tárgyaljuk. Megoldásunk egy tetszőleges, a 2.4.1. szakaszban definiált alakú (vagy ilyenné alakítható) modellt és egy Büchi-automata segítségével ábrázolt ω-reguláris specifikációs követelményt vár bemenetként, és a 2.3.1. szakaszban bemutatott modellellenőrzési sémát valósítja meg szaturáció segítségével. Az 5.1. szakaszban bemutatjuk a szorzat állapottér szimbolikus felderítésére adott új algoritmusunkat, míg az 5.2. fejezetben az elfogadó lefutások kereséséhez használt *hibrid* körkereső algoritmusunk kerül ismertetésre.

5.1. Szorzat állapottér képzése általános Büchi automatával

A 4. fejezetben bemutatott automata-redukciók olyan automatákat eredményeznek, amikre többé már nem érvényes a 3. fejezetben és [11]-ben kihasznált speciális tulajdonság, vagyis az egy állapotba vezető átmeneteken különböző címkék is lehetnek. Ez – és az egyéb forrásból származó, akár kézzel megadott automaták támogatásának igénye[1] – motiválja a korábbi algoritmusunk lecserélését.

5.1.1. Az algoritmus bemutatása

Megoldásunk a korábbiakhoz hasonlóan ábrázolja az állapotteret (a felső L szinten a komponensek változóit kódolva, a legalsón pedig a specifikációs automata állapotait), és szintén a vezérelt szaturációs algoritmusból indul ki. Ezúttal azonban kilépünk a kényszer felüldefiniálásával elérhető megoldások köréből, és kis mértékben ugyan, de magát a vezérelt szaturációs algoritmust is megváltoztatjuk. A kényszer analógiáját most is megőrizzük, azonban most az *emlékezőképességét* fogjuk kihasználni.

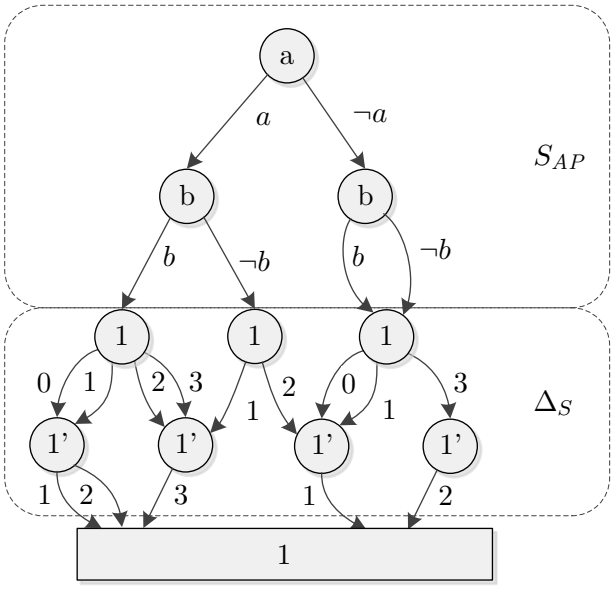
Tekintsük a 2.4.4. szakaszban említett „switch-case” analógiát. Az ott használt kényszernek, mint feltételnek azt a kérdést kellett megválaszolnia, hogy egy adott célállapot benne van-e a kényszer által kódolt állapothalmazban. A 3.2. szakaszban bemutatott korábbi megoldásunk esetén is egy eldöntendő kérdést reprezentáltunk a kényszerrel: be lehet-e lépni a korábban lekötött predikátumokkal, mint bemenettel az adott specifikációs automata állapotba. Jelen megoldásunkban a kérdés már nem eldöntendő lesz – itt lépünk ki a „klasszikus” vezérelt szaturációs algoritmusból.

A motiváció továbbra is egy automata léptetése, amelynek bemenetét a modellbeli lépés célállaptán érvényes predikátumok halmaza adja. Ezek lekötése megtörténhet a modell

[1] Ne feledjük, hogy az LTL-ben leírható követelmények mindegyike leírható Büchi-automatákkal is, de ugyanez fordítva már nem igaz!

léptetése közben (ahogy azt a 3.2. szakaszban bemutatott korábbi megoldásunk is tette), ezután már csak az a kérdés, hogy az automata hová léphet. Ebből látszik is az új kérdés, amit az általánosított kényszerünknek kell megválaszolnia: *Adott bemenet esetén melyek azok az állapotátmenetek, amelyek engedélyezettek a specifikációs automatában?*

Ahhoz, hogy ezt a kérdést megválaszoljuk az általánosított kényszer segítségével, azt a megfigyelést használjuk ki, hogy a bináris kérdésekre adott válaszokat a **0** és az **1** csomópontok adták, amik technikailag MDD-k, akárcsak a most keresett válaszok (az engedélyezett ámenetek halmazát 2k-MDD-vel ábrázoljuk). A következőkben használt *általánosított kényszer* MDD-je tehát legyen olyan alakú, hogy az alsó $2k$ szint egy 2k-MDD, amely az engedélyezett *állapotátmeneteket* írja le, a felső $|AP|$ szint pedig az egyes predikátumok lekötéseit reprezentáló MDD. Ha a korábbiakhoz hasonlóan kikötjük, hogy a specifikációs automatát kis mérete miatt egyetlen változó segítségével kódoljuk (tehát $k = 1$), akkor egy $|AP| + 2$ szintű MDD-t kapunk. Egy ilyen általánosított kényszer látható az 5.1. ábrán.



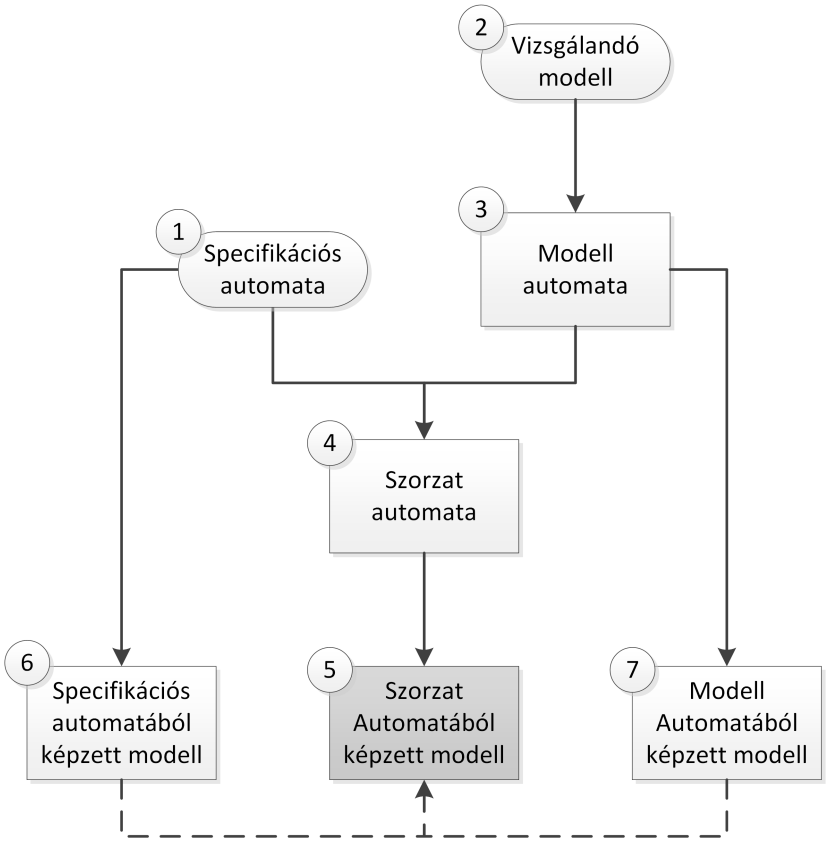
5.1. ábra. Egy általánosított kényszer. A felső szinteken predikátumok lekötéseit, az alsón automataátmeneteket kódolunk.

Ezután már csak a vezérelt szaturációs algoritmust kell módosítanunk, hogy a szorzat állapotteret kódoló MDD legalsó, a specifikációs automatát kódoló szintjén a kényszer válaszát ne annak eldöntésére használja, hogy bevegye-e a talált állapotot az elérhető állapotok halmazába, hanem arra, hogy eldöntse, milyen állapotátmeneti függvényt alkalmaz az alsó szinten. Az így módosított eljárás pszeudokódját a fejezet végén látható 6. algoritmus írja le.

A modellbeli állapotátmenetek reprezentációiba ezúttal nem vesszük be a specifikációs automata átmeneteit, vagyis $2L$ szintű 2k-MDD-kben tároljuk őket. Az algoritmus egyetlen változtatása a 31-33. sorokban látható. Itt „emeljük le” a válaszul kapott 2k-MDD-t a módosított kényszerről, és cseréljük ki rá a modellbeli esemény 2k-MDD-jének legalsó szintjén lévő **1** csomópontját. Az algoritmus így a lekötött bemenettel léptethető átmeneteket fogja végrehajtani az automata állapotain. A kényszer **0** értéke a felsőbb szinteken most azt jelenti, az eddigi lekötés már nem egészíthető ki úgy, hogy a specifikációs automatának legyenek engedélyezett átmenetei. Ekkor a felderítés az adott ágon leállhat, mivel a szinkron szorzat lépésekor mind a két komponensnek tudnia kell lépni. Ahogy az a 40. sorban is látható, a specifikációs automata szintjén (1. szint) már nem használjuk a kényszert.

5.1.2. Formális leírás és a helyesség bizonyítása

Ebben a fejezetben megmutatjuk, hogy a fentebb bemutatott algoritmus a 2.2.2. szakaszban ismertetett automata szorzatot állítja elő. Ehhez elsőként definiáljuk a modell állapotterét reprezentáló Büchi-automatát, ez alapján pedig a specifikációs automatával képzett szorzatot. Bemutatunk egy közvetlenül ennek elérhető állapotait felderítő, a bizonyításhoz használt, szaturáció alapú köztes algoritmust, végül pedig megmutatjuk, hogy ebből származtatható az általunk adott, helyesség szempontjából ekvivalens, de a vezérelt szaturáció használata miatt hatékonyabb [14] megoldás. A köztes algoritmus egy „klasszikus" (módosítatlan) szaturációs algoritmust fog futtatni a szorzat állapottérből képzett modellen. Ezen a modell állapotait és állapotátmeneteit a szorzat automatából állítjuk elő, de az eseményeit a specifikációs és a modell automatákból képzett modellek eseményeivel definiáljuk, hogy megőrizzük a szaturáció hatékonyságához szükséges lokalitást. Ezt a gondolatmenetet mutatja be az 5.2. ábra. A továbbiakban az érthetőség kedvéért zárójelben az ábrán szereplő számokkal is fogunk hivatkozni az éppen említett struktúrákra.



5.2. ábra. A bizonyítás menetében használt különböző struktúrák és származtatásuk.

Az alábbiakban használt jelölések a következők. AP-vel jelöljük az ellenőrizendő követelményekben használt atomi kijelentések (állapotkifejezések) halmazát, az α és β görög kisbetűkkel pedig ennek részhalmazait. Az egyes struktúrákban a p, q, r, s kisbetűkkel állapotokat fogunk jelölni. $\mathcal{A} = \langle \Sigma, Q, \Delta, Q^0, F \rangle$ az ismertnek tekintett specifikációs automata (az 5.2. ábrán az 1. elem), ahol

- $\Sigma = 2^{AP}$ a specifikációs automatata ábécéje, elemei az AP részhalmazai, a következőképpen értelmezve: ha $\alpha \in \Sigma$ tartalmazza az a kijelentést (predikátumot), akkor a-nak *True* értéket kell felvennie, ellenkező esetben *False*-t;

- Q a specifikációs automata állapotainak halmaza, ezekből $Q^0 \subseteq Q$ tartalmazza a kezdőállapotokat, F pedig az elfogadó állapotokat;
- $\Delta \subseteq Q \times \Sigma \times Q$ a specifikációs automata állapotátmeneteinek halmaza.

$M = \langle \mathcal{S}, \mathcal{S}_{init}, \mathcal{E}, \mathcal{N} \rangle$ a vizsgálandó modell (az 5.2. ábrán a 2. elem), ahol:

- $\mathcal{S}$ a modell állapottere, amely $\mathcal{S}_1 \times \ldots \times \mathcal{S}_L$ alakban a részmodellek állapottereinek Descartes-szorzata, ezekből $\mathcal{S}_{init} \subseteq \mathcal{S}$ tartalmazza a modell kezdőállapotait;
- $\mathcal{E}$ az események halmaza;
- $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{S}$ a modell állapotátmeneteinek halmaza (az állapotátmeneti függvény).

$\mathcal{S}_{rch} \subseteq \mathcal{S}$ az $\mathcal{S}_{init}$-ből $\mathcal{N}$ átmeneteken keresztül elérhető állapotok halmaza (lásd 2.4.1. szakasz). Az $L : \mathcal{S} \to 2^{AP}$ függvény megadja az $s \in \mathcal{S}$ állapoton *True* értéket felvevő $P \in AP$ pedikátumok halmazát, másképp írva $L(s) = \alpha$, $\alpha = \{P : P(s) = True\}$.

A probléma formalizálása és a modell automata

Ezek után definiáljuk az M modellt reprezentáló $\mathcal{M} = \langle \Sigma, S, N, S^0, S \rangle$ *modell automatát* (az 5.2. ábrán a 3. elem):

- $\Sigma = 2^{AP}$ a specifikációs automatához is tartozó ábécé.
- $S = \mathcal{S} \cup \{s_0\}$ a modell állapotainak megfelelő állapotok és egy további kitüntetett s_0 kezdőállapot.
- $N \subseteq S \times \Sigma \times S$ a modell automata állapotátmeneteinek halmaza.
- $S^0 = \{s_0\}$ a kitüntetett kezdőállapot.
- Az elfogadó állapotok halmaza a teljes S, ezt az alábbiakban hamarosan megmagyarázzuk.

Az N állapotátmeneti reláció akkor tartalmaz egy $\langle s, \alpha, r \rangle$ átmenetet, ha

- $\langle s, r \rangle \in \mathcal{N}$, vagyis az állapotok között létezik átmenet a modellben, illetve
- $\alpha = L(r)$, vagyis a bemenet a célállapoton igaz predikátumok halmaza.

Felveszünk még továbbá egy $\langle s_0, L(r), r \rangle$ átmenetet minden $r \in \mathcal{S}_{init}$ állapotra is, ezek lesznek az *inicializáló átmenetek*, amelyek a modellt alapállapotba viszik. Erre azért van szükség, mert a specifikációs automata kezdőállapotai is a kiértékelés kezdetét reprezentálják, amikor még nem ismert a modell kezdőállapota sem.

Magyarázat. *Most tehát a specifikációs (1.) és a modell automata (3.) is $\Sigma = 2^{AP}$ feletti szavakon, vagyis a modell állapotain értelmezhető állapotkifejezések szekvenciáin működik. Ez praktikus, mert a specifikációs automata független a konkrét modelltől, azonban a szemléletesség kedvéért érdemes meggondolni, hogy az állapotátmeneteket leszámítva ezzel ekvivalens megoldás adódna, ha Σ elemeinek $\mathcal{S}$-beli állapotokat választanánk. Ekkor a specifikációs automata $\langle p, \alpha, q \rangle$ átmenetei a $\{\langle p, s, q \rangle : L(s) = \alpha\}$ átmenetekre esnének szét, a modell automata átmenetei pedig $\langle s, r, r \rangle$ alakúak lennének. Az automaták ekkor $\mathcal{S}$ feletti lefutásokat olvasnának, $\mathcal{M}$ az M-ben előforduló lefutásokat fogadná el (innen adódik, hogy minden S-beli állapot elfogadó), $\mathcal{A}$ pedig a specifikációt megsértő lefutásokat. A továbbiakban bonyolultsága miatt nem ezt a megközelítést használjuk, de érdemes mindig szem előtt tartani, hogy valójában ez a motiváció a szorzat automata elkészítése mögött.*

A szorzat automata legyen $\mathcal{P} = \langle \Sigma, P, \delta, P^0, \mathcal{F} \rangle$ alakú (az 5.2. ábrán a 4. elem), ahol a 2.2.2. szakasznak megfelelően

- $\Sigma = 2^{AP}$ a közös ábécé,
- $P = S \times Q$ a kompozit állapotok halmaza,
- a $\delta \subseteq P \times \Sigma \times P$ állapotátmeneti reláció akkor tartalmaz egy $\langle\langle s, p\rangle, \alpha, \langle r, q\rangle\rangle$ átmenetet, ha $\langle s, \alpha, r\rangle \in N$, illetve $\langle p, \alpha, q\rangle \in \Delta$,
- $P^0 = S^0 \times Q^0$ a kompozit kezdőállapotok halmaza, valamint
- $\mathcal{F} = S \times F$ a kompozit elfogadó állapotok halmaza.

Jelölje $P_{rch} \subseteq P$ a P^0-ból elérhető kompozit állapotok halmazát. Jól látható, hogy ekkor $P_{rch} \subseteq (\mathcal{S}_{rch} \cup \{s_0\}) \times Q$ is fennáll. *A modellellenőrzés problémájának megoldásához ezt kell kiszámítanunk.*

A szorzat állapottér közvetlen számítása

Most konstruálunk egy algoritmust, amely a fenti problémát közvetlen módon reprezentálva a szaturációs algoritmushasználatával, annak módosítása nélkül oldja meg. Erre azért van szükség, hogy az új algoritmusunkat egy bizonyíthatóan jól működő [2] algoritmusra vezethessük vissza a helyesség bizonyításához.

Ehhez az imént bemutatott szinkron szorzat automatát (4.) reprezentálni fogjuk a 2.4.1. szakaszban bemutatott módon, hogy azon közvetlenül futtatva a szaturációs algoritmust megkapjuk a kívánt P_{rch} halmazt. Definiáljuk tehát a $D = \langle \mathcal{D}, \mathcal{D}_{init}, \mathcal{E}_+, \mathcal{N}_+ \rangle$ *kompozit modellt* (az 5.2. ábrán a 5. elem):

- $\mathcal{D} = \{P \setminus P^0\} \times \Sigma$ a kompozit automata[2] és egy képzeletbeli, a legutolsó bemenetet tároló „regiszter” együttes állapota.
- $\mathcal{D}_{init} \subseteq \mathcal{D}$ a kompozit automatában P^0-ból egy lépéssel elérhető állapotok és a lépést kiváltó bemenetek párosainak halmaza, vagyis $\{\langle r, \alpha\rangle : \exists s \in S^0, \langle s, \alpha, r\rangle \in \delta\}$.
- $\mathcal{E}_+$ az eredeti M modell minden eseményéhez tartalmaz egy *kiterjesztett eseményt*, amelynek hatásköre a kompozit modell többi komponensére is kiterjed.
- $\mathcal{N}_+$ a kompozit modell állapotátmeneti függvénye, amely $\mathcal{N}_+ = \bigcup_{\varepsilon \in \mathcal{E}_+} \mathcal{N}_\varepsilon$ alakban állítható elő.

A kompozit modell állapotterét a 2.4.1. szakaszban látott módon az egyes komponensek állapottereivel felírva

$$\mathcal{D} = \underbrace{\mathcal{S}_1 \times \ldots \mathcal{S}_L}_{\mathcal{S}} \times \underbrace{\mathcal{S}_{P_1} \times \ldots \times \mathcal{S}_{P_{|AP|}}}_{\mathcal{S}_\Sigma} \times \underbrace{\mathcal{S}_\mathcal{A}}_{\mathrm{Q}}$$

felbontást kapunk, ahol $\mathcal{S}_1, \ldots, \mathcal{S}_L$ az M eredeti modell részmodelleinek állapotterei, $\mathcal{S}_{P_1}, \ldots, \mathcal{S}_{P_{|AP|}} = \{\mathit{True}, \mathit{False}\}$ a Σ elemeinek megfelelően az egyes predikátumok értékét reprezentáló komponensek állapotterei, $\mathcal{S}_\mathcal{A} = Q$ pedig a specifikációs automata (1.) állapottere. Ehhez a dekompozícióhoz fogjuk most megalkotni az egyes $\mathcal{E}$-beli események állapotátmeneti függvényeit úgy, hogy a D modell (5.) és a $\mathcal{P}$ automata (4.) kölcsönösen megfeleltethető legyen egymásnak. Ebből már adódni fog $\mathcal{N}_+$, és mivel a 2.4. szakaszban megmutattuk, hogyan alkalmazható a szaturációs algoritmus az így megadott modell $\mathcal{D}_{rch}$

[2]Itt már eltávolítjuk az automatában technikai okokból bevezetett kitüntetett kezdőállapotot.

elérhető állapotainak felderítésére, elkészültünk az algoritmussal. A szaturációs algoritmus ekkor az elérhető állapotok felderítésekor a $P_{rch} = \{\langle s, q\rangle : \exists \alpha \in \Sigma, \langle\langle s, q\rangle, \alpha\rangle \in \mathcal{D}_{rch}\}$ elérhető kompozit állapotok halmazát is előállítja. A következő fejezetben megmutatjuk, hogyan származtatható ebből az 5.1.1. szakaszban bemutatott hatékonyabb, a vezérelt szaturációs algoritmus alapjain megvalósított algoritmusunk.

Az eseményekhez tartozó állapotátmeneti függvények megadásához elsőként vizsgáljuk meg, hogyan kapcsolódik össze a két automata (1. és 3.) a szinkron szorzatban. Ha $\mathcal{A}$-t (1.) és $\mathcal{M}$-et (3.) is külön-külön ábrázolnánk modellként (6. és 7.) az előbb bemutatott módon (tehát a legutolsó bemenetet is az állapotokban tárolva), akkor mind a két modell egyformán tartalmazná az AP-beli predikátumokat reprezentáló *predikátum komponenseket*. Az események engedélyezettsége mindkét esetben csak a többi komponens állapotaitól függene, hiszen a predikátum komponensek az előző bemenetet tükrözik, ami a lépés szempontjából irreleváns. Tüzelésük viszont úgy változtatná mindkét komponenst, hogy a kapott új kombinációk az eredeti automatabeli állapotátmeneti relációban megengedettek legyenek. Ezzel azt modelleznénk, hogy tetszőleges bemenet hatására hogyan viselkedhet az adott automata.

A kompozit modellben (4.) is ez a cél, azonban itt a predikátum komponensek közösek. A szinkron szorzat miatt mindkét automatában (vagyis mindkét hozzájuk tartozó modellben) lépnünk kell, de mivel mindkettő megváltoztatja a predikátum komponensek állapotát, így csak azok az átmenetek hajthatók végre, amik ugyanolyan állapotba viszik őket. Mivel pedig a predikátum komponensek célállapotait úgy definiáltuk, hogy a legutolsó bemenetet ábrázolják, ezzel éppen azt mondtuk, hogy a teljes rendszer akkor léphet, ha a két automata (1. és 3.) ugyanannak a bemenetnek a hatására lépni tud, ami pedig a szinkron szorzat (4.) állapotátmeneteinek definíciója.

Ezek után az $\mathcal{A}$ automatához tartozó modellben (6.) definiáljunk egyetlen $\varepsilon_{\mathcal{A}}$ eseményt, melynek állapotátmeneti relációja $\mathcal{N}_{\varepsilon_{\mathcal{A}}} = \{\langle\langle \alpha, p\rangle, \langle \beta, q\rangle\rangle : \langle p, \beta, q\rangle \in \Delta, \alpha \in \Sigma\}$, az $\mathcal{M}$-hez tartozó modellben (7.) pedig minden M-ben definiált $\varepsilon \in \mathcal{E}$ eseményhez egy ε_+ eseményt, melynek állapotátmeneti relációja $\mathcal{N}_{\varepsilon_+} = \{\langle\langle s, \alpha\rangle, \langle r, \beta\rangle\rangle : \langle s, r\rangle \in \mathcal{N}_{\varepsilon}, \alpha \in \Sigma, \beta = L(r)\}$. A D modell (5.) $\varepsilon \in \mathcal{E}_+$ eseményhez tartozó állapotátmeneti függvénye ekkor $(\mathcal{N}_{\varepsilon_+} \times \mathcal{I}(Q)) \cap (\mathcal{I}(\mathcal{S}) \times \mathcal{N}_{\mathcal{A}})$ alakban áll elő, ahol $\mathcal{I}(\mathcal{X}) = \{\langle x, x\rangle : x \in \mathcal{X}\}$ a csak a másik modellben szereplő komponensek állapotainak helybenhagyását jelenti. Itt a metszés fejezi ki azt, hogy mindkét lépést egyszerre kell végrehajtani, és a predikátum komponenseket ugyanabba az állapotba kell vinni. Ezzel a predikátum komponenseken keresztül szinkronizáltuk a modell és a specifikációs automata lépéseit. Az állapotátmeneti függvény előállítását az 5.3. ábra szemlélteti.

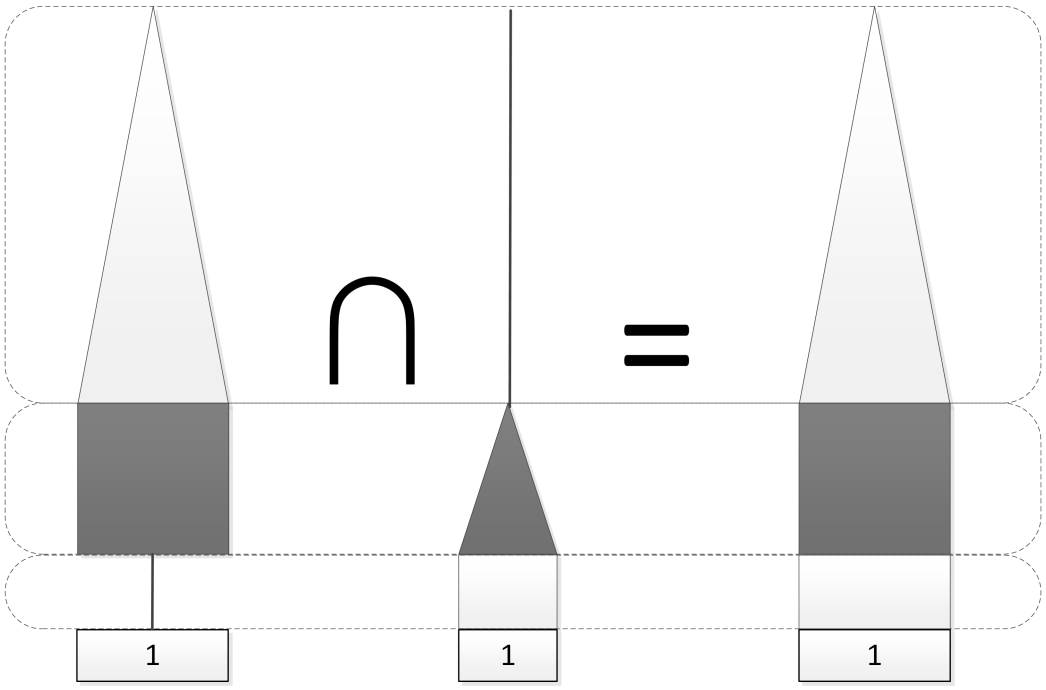
Magyarázat. *A célunk itt az eredeti modell lokális eseményeinek megőrzése. Emiatt a szorzat automatához tartozó modell állapotátmeneteit az eredeti modell eseményeinek megfelelően dekomponáljuk eseményekké.*

Ezzel elkészült a szinkron szorzást szaturációs alapokon elvégző *referencia algoritmus*, melyet a következő szakaszban felhasználunk a hatékonyabb, vezérelt szaturációs alapokon működő új algoritmusunk helyességének bizonyításához.

Az új algoritmus levezetése

Az 5.1.1. szakaszban bemutatott algoritmus az előzőekhez nagyon hasonlóan működik. A különbség az, hogy az állapottér reprezentációból kihagyjuk a predikátum komponensek állapotait, az előző metszést pedig az általánosított kényszer segítségével fogjuk elvégezni.

Tudjuk, hogy a kényszer „léptetése” a lokális célállapotokkal történik. A 3.2. szakasz alapján az is világos, hogy léptetés hatására a kényszerben a lokális célállapotra igaz



5.3. ábra. A szinkron állapotátmeneti függvény előállítása. Felül a modell, középen a predikátum komponensek, alul a specifikációs automata átmenetei láthatók.

predikátumok értéke lekötésre kerül, vagyis az az út, amin a kényszerben haladunk, megfelelne a predikátum komponensek állapotainak a lépés után. A kényszer léptetése tehát nem más, mint a predikátum komponensek állapotának beállítása, de ezúttal nem az állapottérben, hanem egy külön saját MDD-ben. Maga az állapotátmeneti függvény, ami a predikátum komponenseket beállítja ebben az esetben az L címkéző függvény lesz, amit könnyen megvalósíthatunk MDD-k nélkül is. Ráadásul most a külön MDD miatt a predikátum komponensek együttes állapottere előre ismert és felépíthető, mivel mindegyik komponens a többitől függetlenül *True* vagy *False* értéket vehet fel.

Vizsgáljuk most meg, hogy hogyan viszonyul a kényszerhez a specifikációs automata. A kényszer felépítése olyan, hogy a predikátumok lekötéseinek „alján" az ezzel a lekötéssel engedélyezett állapotátmenetek találhatók, vagyis egy $\mathcal{N}_{\mathcal{A}}$-hoz nagyon hasonló strukturával van dolgunk. Mindösszesen annyi a különbség, hogy az $\mathcal{N}_{\mathcal{A}}$-ban a predikátum komponens tetszőleges kiinduló állapotát kódoló komponenseket elhagyjuk, mivel nem hordoznak információt, vagyis képezzük a $\{\langle p, \langle \beta, q\rangle\rangle : \langle p, \beta, q\rangle \in \Delta\}$ állapotátmeneti relációt és az elemeit $\langle \beta, p, q\rangle$ alakban kódoljuk. Jól látható, hogy ez megfelel az eredeti automata állapotátmeneti relációjának.

Az állapotátmeneti reláció előállításakor használt metszet pontosan azokat az állapotátmeneteket állította elő, amelyekben egy adott $\langle s, r\rangle$ modellbeli átmenetet előidézni képes α bemenethez létezett olyan $\langle p, q\rangle$ specifikációs automatabeli átmenet, amit szintén α idéz elő. Ugyanez történik akkor is, amikor tekintünk egy $s \in \mathcal{S}_{rch}$ állapotot, a kényszer végigléptetésével kiválasztjuk az átmenetet lehetővé tevő bemenetet (ez $\alpha = L(s)$ lesz), és meglépünk minden olyan specifikációs automatabeli átmenetet, ami ezzel a bemenettel végrehajtható.

Ezzel megmutattuk, hogy az algoritmusunk pontosan ugyanazokat a műveleteket végzi el, mint az előző referencia megoldás, ezáltal bizonyítottuk, hogy az algoritmus valóban a P_{rch} halmaz kiszámítását végzi el, mégpedig a predikátum komponensek kihagyása miatt ezúttal közvetlenül.

5.2. Elfogadó lefutások hatékony keresése

Korábbi munkánkban sikerrel bevezettünk és alkalmaztunk egy új, szaturáció alapú, szimbolikus algoritmust, melynek segítségével az elfogadó lefutásokhoz tartozó köröket az állapottér felderítése *közben*, „on-the-fly" módon tudtuk detektálni. Az algoritmus inkrementális volt abban az értelemben, hogy minden csomópont szaturálása után lefutott, és csak a csomópont szintjén megjelenő éleket is tartalmazó köröket vizsgálta, azonban sok esetben még így is túlságosan lassúnak bizonyult. Ennek oka a köröket reprezentáló fixpont kiszámításának módjában fedezhető fel, ugyanis ehhez a szaturált csomópont által kódolt állapothalmazt kellett ciklikusan addig szűkíteni, amíg meg nem találtuk a kört. Abban az esetben, ha egyáltalán nem volt kör (mindig ez a helyzet akkor, ha az ellenőrizendő kifejezés érvényes a modellre), az algoritmusnak minden lépésben a teljes halmazt „el kellett fogyasztania", ezek mérete pedig ráadásul egyre nő a felderítés során. Emiatt, bár újszerűsége folytán a jelen munka alapjait is adó komoly elméleti eredmény, a körkeresés sok esetben a régi modellellenőrző algoritmus egyértelmű gyenge pontjának bizonyult. Ebben a szakaszban bemutatjuk az ezt kiküszöbölő algoritmikus fejlesztéseket.

Vizsgálataink alapján két új módszert dolgoztunk ki, amelyek jól kiegészítik egymást. Az egyik azon esetek előfordulását hivatott csökkenteni, amikor az algoritmus feleslegesen fut le és végzi el az iteratív szűkítéseket, mert a vizsgált részállapottér nem tartalmaz kört. Ennek megelőzésére bevezetjük a *lokális körkeresés* módszerét, ami az állapottér egy *absztrakcióját* felhasználva szükséges feltételt fogalmaz meg a kör létezésére. Ez az absztrakció kellően kisméretű lesz explicit körkereső algoritmusok alkalmazásához (ezek jóval gyorsabbak szimbolikus társaiknál, de csak kis állapottereken alkalmazhatók), így *hibrid* megoldásunk ötvözi a szimbolikus és explicit megoldások előnyeit.

A másik irányvonal a szűkítendő halmaz méretét hivatott csökkenteni, ami a szűkítő lépések számának csökkenéséhez, ezáltal pedig teljesítménynövekedéshez vezet. Ehhez az algoritmust nem a teljes szaturált állapottérről fogjuk indítani, hanem csak olyan állapotok halmazából, amik közül legalább az egyiken az állapottérbeli körök biztosan áthaladnak, ha vannak. Ilyen állapotokat úgy állítunk elő, hogy az *állapottér bejárása közben* a lépések célállapotai közül kigyűjtjük azokat, amelyeket korábban már megtaláltunk. Ezek a *visszatérő állapotok* két okból jelenhetnek meg: vagy egy kör bejárása közben értünk vissza az állapotszekvencia elejére, vagy két úton értünk egy állapotot. Ennek eldöntése már a meglévő algoritmusunk használható, azonban sokkal kisebb állapothalmazon, ezáltal sokkal hatékonyabban.

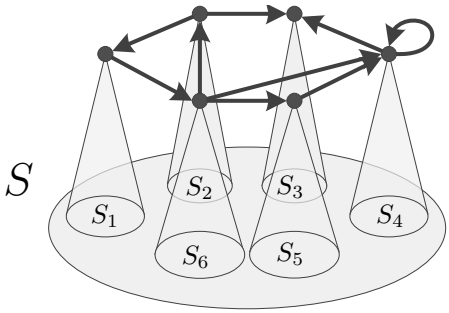
5.2.1. Lokális körkeresés

Elsőként definiáljuk az absztrakciót, amin a lokális körkeresést futtatni fogjuk.

Definíció. *Az n csomóponthoz tartozó lokális állapottérnek nevezzük és $\mathcal{G}(n) = (V, E)$-vel jelöljük azt az irányított gráfot, amelyben V elemei azok az $i \in \mathcal{S}_{n.lvl}$ lokális állapotok, amelyekre $\mathcal{B}(n[i]) \neq \emptyset$, E pedig a modellbeli $\mathcal{E}_l$ események állapotátmeneteinek vetületei, mint élek. Formálisan egy $\langle i_l, j_l \rangle \in E$, ha a többi részmodellben létezik olyan i_k és j_k $(k \neq l)$ lokális állapot, amiket i_l-hez illetve j_l-hez véve az $i = \langle i_1, \ldots, i_L \rangle$ és $j = \langle j_1, \ldots, j_L \rangle$ globális állapotokra $\langle i, j \rangle \in \bigcup_{\varepsilon \in \mathcal{E}_l} \mathcal{N}_\varepsilon$.*

Az n csomóponthoz tartozó lokális állapottér tehát az l. részmodell állapottere mindazon átmenetekkel, amik az alacsonyabb szintű részmodellek valamely állapota mellett megléphetők. Érdemes megfigyelni, hogy tényleg csak az alacsonyabb szintű komponensek állapota érdekes, mivel az $\mathcal{E}_l$ események definíció szerint nem függnek a magasabb szintű részmodellek állapotaitól. Másrészről a lokális állapottér éleit úgy is tekinthetjük, mint

az alacsonyabb szintű $n[i]$ csomópontok által kódolt diszjunkt $\mathcal{S} = \mathcal{B}(n[i])$ állapothalmazok között futó átmenetek események szerinti csoportjait. Ezt a megközelítést szemlélteti az 5.4. ábra.



5.4. ábra. Egy lokális állapottér és a hozzá tartozó globális állapothalmazok.

Definíció. *Az n csomóponthoz tartozó lokális körnek nevezünk minden olyan $\mathcal{C}_n$ lokális állapothalmazt, melynek elemeit egy $\mathcal{G}(n)$-beli irányított kör csúcsainak megfelelő lokális állapotok adják. Az n csomóponthoz tartozó elfogadó lokális kör egy olyan n-hez tartozó $\mathcal{C}_n^F$ lokális kör, amiben van olyan $i \in \mathcal{C}_n^F$ elfogadó csúcs, amire az $n[i]$ csomópont által kódolt állapothalmaz tartalmaz elfogadó állapotot (vagyis a legalsó szinten a specifikációs automatához tartozó lokális állapotnak megfelelő automatabeli állapot elfogadó).*

Lokális kör tehát akkor létezik, ha az l. részmodellben a többi részmodell állapotát szabadon állítgatva körbe tudunk lépkedni az állapottérben. Elfogadó (a globális állapottér szempontjából potenciálisan elfogadó) pedig akkor lesz, ha valamelyik i állapotában a $\mathcal{B}(n[i])$ részállapottérben szerepel elfogadó állapot, mivel ilyenkor van rá esély, hogy a lokális kör egy esetleges tényleges bejárása során (ilyen nem mindig létezik!) elfogadó állapotot is érintünk.

Hasonlóan definiálhatjuk az n csomóponthoz tartozó lokális erősen összefüggő komponenseket (local strongly connected componens, LSCC) és az n-hez tartozó lokális elfogadó erősen összefüggő koponenseket (local accepting strongly connected component, LASCC) is. Könnyen látható, hogy egy n csomóponthoz tartozó lokális állapottérben pontosan akkor létezik LASCC, ha elfogadó kör is található benne. A körkeresés elindítása az alábbi tételben megfogalmazott szükséges feltételtől fog függeni.

Tétel. *Ha egy l. szinten lévő n csomóponthoz tartozó lokális állapottér nem tartalmaz LASCC-t, valamint az $n[i]$ ($i \in \mathcal{S}_l$) alacsonyabb szintű csomópontok által kódolt $\mathcal{B}(n[i])$ részállapotterekben sincs $\mathcal{E}_1, \ldots, \mathcal{E}_{l-1}$-beli eseményeket használó elfogadó kör, akkor az n által kódolt $\mathcal{B}(n)$ (rész)állapottérben sem lesz $\mathcal{E}_1, \ldots, \mathcal{E}_l$-t használó.*

Bizonyítás. Indirekt módon bizonyítunk. Tegyük fel, hogy a feltételek fennállása mellett $\mathcal{B}(n)$ tartalmaz egy $\mathcal{C}_n^F$ elfogadó kört, melynek egyik $s \in \mathcal{C}_n^F$ állapota $\mathcal{B}(n[i])$-ben van. Vizsgáljuk meg a kört ebből az s állapotból. A kör mentén elindulva vissza kell érnünk s-be. Ez háromféleképpen történhet:

1. $\mathcal{B}(n[i])$ halmazon belül maradva, csak $\mathcal{E}_1, \ldots, \mathcal{E}_{l-1}$-beli eseményeket felhasználva,
2. $\mathcal{B}(n[i])$ halmazon belül maradva, de $\mathcal{E}_l$-beli eseményeket ($\mathcal{N}_l$-beli hurokéleket) is felhasználva, illetve
3. $\mathcal{B}(n[i])$ halmazból kilépve legalább egy másik $\mathcal{B}(n[j])$ ($j \neq i$) halmazt is érintve.[3]

[3]Vegyük észre, hogy $\mathcal{B}(n[i]) \cap \mathcal{B}(n[j]) = \emptyset$, ha $i \neq j$, mivel az $n.lvl$ szinthez tartozó részmodellben mindenképpen különböznek a két halmaz globális állapotai.

Az első eset ellentmond a feltételeknek, ugyanis ez éppen azt jelentené, hogy $\mathcal{B}(n[i])$-ben van csak $\mathcal{E}_1, \ldots, \mathcal{E}_{l-1}$-beli eseményeket használó elfogadó kör. A második esetben az $\mathcal{E}_l$-ből felhasznált esemény érintett átmenete[4] egy hurokélre kéződik le a $\mathcal{G}(n)$ lokális állapottérben, ami egy l. szinten lévő erősen elfogadó kört jelent, ezáltal $\mathcal{G}_l$ a feltételezéssel szemben tartalmaz LASCC-t.

A harmadik esetben azt használjuk ki, hogy a $\mathcal{B}(n[k])$ részállapotterek páronként diszjunktak, és köztük a tételben vizsgált események közül csak $\mathcal{E}_l$-beliek segítségével lehet lépni. Emiatt minden olyan átmenet, ami kilép egy $\mathcal{B}(n[k])$ halmazból, megjelenik az n-hez tartozó lokális állapottérben, mint két csúcs között futó él. Ahhoz, hogy a kör bejárása után visszatérhessünk s-be, a séta során minden alkalommal, amikor belépünk egy $\mathcal{B}(n[j])$-be, ki is kell lépnünk belőle, végül pedig újra be kell lépnünk $\mathcal{B}(n[i])$-be. Így az i és a meglátogatott j lokális állapotokhoz tartozó $\mathcal{G}(n)$-beli csúcsok „ki"-foka és „be"-foka csúcsonként egyenlő lesz, vagyis a bejárásunk $\mathcal{G}(n)$-ben egy kör. Ha pedig az eredeti $\mathcal{C}_n^F$ kör elfogadó volt, akkor rendelkezett legalább egy elfogadó állapottal, ami az azt tartalmazó, a séta során érintett $\mathcal{B}(n[j])$ halmazhoz tartozó $\mathcal{G}(n)$-beli csúcsot elfogadóvá teszi, emiatt pedig a $\mathcal{G}(n)$-ben talált kör is elfogadó lesz, ami ellentmond az indirekt feltevésnek. □

Az algoritmus korábban akkor indított körkeresést, amikor befejezte egy l szinten lévő n csomópont szaturálását. Ebben az állapotban teljesül, hogy az n csomópont szaturált, és mivel a teljes algoritmus véget ér egy elfogadó kör megtalálása után, azt is tudjuk, hogy az $n[i]$ alsóbb szintű szaturált csomópontok által kódolt részállapotterek sem tartalmaznak $\mathcal{E}_1, \ldots, \mathcal{E}_{l-1}$-beli eseményeket használó elfogadó kört.[5] Emiatt – alkalmazva a tételben kimondottakat – az új algoritmus csak akkor indít körkeresést, ha az n csomóponthoz tartozó lokális állapottér tartalmaz LASCC-t. Ezek keresése pedig hatékonyan elvégezhető az explicit módon ábrázolt lokális állapottéren, például a Tarjan-algoritmus segítségével.

5.2.2. Visszatérő állapotok keresése

Ezt a szakaszt is a továbbikaban használt egyik fontos fogalom definiálásával kezdjük.

Definíció. *Visszatérő állapotoknak nevezzük azon globális állapotokat, amelyek az állapottér felderítése során egy állapotátmeneti függvény alkalmazásakor egyaránt szerepelnek az addig felderített és a tüzeléssel elért állapotok között. Formálisan $\mathcal{R} = \mathcal{X} \cap \mathcal{N}_{\mathcal{X}}(\mathcal{X})$, ahol $\mathcal{X}$ egy állapothalmaz, $\mathcal{N}_{\mathcal{X}}$ pedig egy állapotátmeneti függvény.*

A továbbfejlesztett algoritmus alapját a következő tétel adja.

Tétel. *Amennyiben egy $\mathcal{X}$ állapothalmazon $\mathcal{N}_{\mathcal{X}}$ tetszőleges állapotátmeneti függvényt alkalmazva $\mathcal{X}'$ halmazt kapunk, és $\mathcal{X}'$ tartalmaz kört, de $\mathcal{X}$ nem, akkor az $\mathcal{N}_{\mathcal{X}}$ átmenetek tüzelése közben keletkeztek visszatérő állapotok.*

Bizonyítás. Ismét indirekt módon bizonyítunk. Tegyük fel, hogy a feltételek teljesülése mellett $\mathcal{N}_{\mathcal{X}}$ tüzelése közben nem keletkezik visszatérő állapot. Mivel $\mathcal{X}$-ben még nem volt kör, de $\mathcal{X}'$-ben már igen, a megjelenő kör mindenképpen használ $\mathcal{N}_{\mathcal{X}}$-beli átmeneteket. Egy ilyen átmenet visszavezethet egy $x \in \mathcal{X}$ állapotba, ekkor viszont x definíció szerint visszatérő állapot, ez pedig ellentmond az indirekt feltevésnek. A másik lehetőség az, hogy az átmenet egy $x' \in \mathcal{X}' \setminus \mathcal{X}$ új állapotba vezet. Ekkor ebből az állapotból nem vezethet tovább átmenet, mivel az $\mathcal{N}_{\mathcal{X}}$ állapotátmeneti függvényből ténylegesen alkalmazott $\langle x, x' \rangle$

[4] A $\mathcal{E}_l$-beli események definíció szerint hatással vannak az l. részmodell állapotára, de ez csak azt jelenti, hogy az állapotátmeneti függvényükben *létezik* olyan átmenet, ami i_l-t megváltoztatja. Ezek mellett lehetne olyan átmenetek is, amelyek i_l-t érintetlenül hagyják, de mégis ehhez az eseményhez tartoznak.

[5] Ezek szaturálása után az ilyen köröket már megtaláltuk volna.

átmenetekben minden x kiindulási állapot $\mathcal{X}$-beli (hiszen erre a halmazra alkalmaztuk $\mathcal{N}_{\mathcal{X}}$-et). Így az $\mathcal{X}' \setminus \mathcal{X}$-beli új állapotokba csak befelé vezetnek átmenetek (ebben a lépésben kerültünk ide először, továbblépni még nem volt lehetőségünk), ekkor viszont kör sem keletkezhetett, ami ismét ellentmondás. □

A tételt a következőképpen alkalmazzuk. Egy l. szinten lévő n csomópont szaturálásakor csak a körkeresés szempontjából ezen a szinten releváns $\mathcal{E}_l$ eseményekhez tartozó állapotátmeneti függvényeket vizsgáljuk, az ezek tüzelése során észlelt visszatérő állapotokat fogjuk gyűjteni. Minden egyes tüzelésnél az addig felderített, n által aktuálisan kódolt állapotteret fogjuk $\mathcal{X}$-nek tekinteni, $\mathcal{N}_\varepsilon$ ($\varepsilon \in \mathcal{E}_l$) pedig $\mathcal{N}_{\mathcal{X}}$ lesz. Minden egyes tüzelésnél megjegyezzük a visszatérő állapotokat, és ezek uniójára fogjuk alkalmazni a következő lemmát.

Lemma. *Ha $\mathcal{X}$ állapothalmaz nem tartalmaz kört, de tetszőleges $\mathcal{N}_i$ állapotátmeneti függvényekre az $\mathcal{X}' = \mathcal{X} \cup \mathcal{N}_1(\mathcal{X}) \cup \ldots \cup \mathcal{N}_n(\mathcal{N}_{n-1}(\ldots\mathcal{N}_1(\mathcal{X})))$ állapothalmaz igen, akkor $\mathcal{R} = \bigcup_{1<i<n} \mathcal{R}_i$ nem üres, vagyis valamelyik lépésben keletkezett visszatérő állapot.*

Bizonyítás. A lépések száma szerint induktívan bizonyítunk. Az iménti tétel szerint az állítás igaz $n = 1$ lépés esetén, tegyük fel, hogy az n. lépésben is igaz. Ez vagy azt jelenti, hogy $\mathcal{X}' = \mathcal{X} \cup \mathcal{N}_1(\mathcal{X}) \cup \ldots \cup \mathcal{N}_n(\mathcal{N}_{n-1}(\ldots\mathcal{N}_1(\mathcal{X})))$ állapothalmazban nincs kör, és így $\mathcal{R} = \bigcup_{1<i<n} \mathcal{R}_i$ lehet üres, vagy azt, hogy már találtunk kört, így az állítás szerint $\mathcal{R} \neq \emptyset$. Utóbbi esetben az állítás az $n + 1$. lépésben a szükségesség miatt triviálisan igaz, mivel $\mathcal{R}$ már csak bővülhet. Az előbbi esetben pedig $\mathcal{X}'$-re alkalmazhatjuk a fenti tételt, mivel $\mathcal{X}'$-ben nincs kör, így ha $\mathcal{X}'' = \mathcal{X}' \cup \mathcal{N}_{n+1}(\mathcal{X}')$ tartalmaz kört, $\mathcal{R}_{n+1}$ nem lesz üres, és így $\mathcal{R}' = \mathcal{R} \cup \mathcal{R}_{n+1}$ sem lehet az. □

A fentiek közvetlen következménye, hogy ha a lemma feltétele teljesül, akkor az említett kör(ök) mindenképpen érintenek legalább egy állapotot $\mathcal{R}$-ből. Ezáltal a lemmával nem csak egy újabb szükséges feltételt kaptunk egy kör létezésére, hanem egy olyan halmazt is konstruáltunk, ami általában szűkebb a teljes vizsgált állapothalmaznál, és biztosan tartalmaz körbeli állapotot. Ebből már elindítható a körkereső algoritmus is, amely a halmaz elemeit végül a tényleges egy kör részét képező állapotokra fogja leszűkíteni, ha vannak ilyenek. Ennek módját a következő, 5.2.3. szakasz mutatja be.

A visszatérő állapotok felderítését a hatékonyság kedvéért inkrementálisan, a szaturációs algoritmus keretein belül valósítottuk meg, megspórolva ezzel a definícióbeli halmazmetszéseket, ami szimbolikusan kódolt halmazok esetén költséges művelet. Az így módosított szaturációs függvények pszeudokódját a fejezet végén található 6. algoritmus mutatja be. A *ProductRelProd* függvény ezúttal paraméterül várja az u-val jelölt kiindulási állapotteret, amit a tételben $\mathcal{X}$ jelölt, illetve visszaadja az r visszatérő állapotokat kódoló csomópontot is. Az algoritmus a hagyományos kényszerhez hasonló módon ellenőrzi, hogy az új állapot u-ban van-e, de nem áll le, ha kilép belőle – viszont ha benne marad, a terminális szinten **1**-et visszaadva és a felsőbb szinteken r-be uniózva felveszi az állapotot a visszatérő állapotok halmazába. u és r értéke a cache-be is bekerül, u bemenő, r pedig kimenő paraméterként. A 23. sorban látható *DetectCircles* függvényt a következő szakaszban foglaljuk össze.

5.2.3. Összegzés

Az új körkereső algoritmusunk az imént bemutatott két fejlesztést többféleképpen is alkalmazza. Egyrészről az előző két szakaszban megfogalmaztunk egy-egy szükséges feltételt kör létezésére, melyeket az algoritmus a körök keresése előtt kis költséggel, a visszatérő

állapotok esetén inkrementálisan kiszámítva ellenőriz, megakadályozva a feleslegesen elindított lassú szűkítő lépéseket. Másrészt a szűkítő lépések hatékonysága is növelhető azáltal, hogy

1. a három kezdeti halmaz meghatározásakor nem a teljes szaturált részállapottérből lépünk vissza egyet az $\mathcal{E}_l$ eseményekkel, hanem csak a visszatérő állapotokból, illetve
2. a $\mathcal{E}_l$ eseményekkel csak olyan állapotokba lépünk, amiknek vetületei a lokális állapottérben egy LASCC-be tartoznak, sőt,
3. az egyes LASCC-knek megfelelően mindegyikhez külön kezdeti halmazokat rendelünk, és külön-külön szűkítve őket mindegyik csoportban csak az adott LASCC-hez tartozó állapotátmeneteket használunk fel.[6] Ezzel a problémát sok esetben *könnyen kezelhető részekre partícionálhatjuk.*

Az így módosított *IncrementalDetectCircles* függvény pszeudokódját az 5. algoritmus mutatja be. A 6. algoritmusban még MDD csomópontokkal paraméterezve hívjuk a függvényt, így elsőként vesszük a megfelelő halmazokat.[7] A *FindLASCCs* függvény képzi az 5.2.1. szakaszban bemutatott lokális állapotteret, majd Tarjan algoritmusa segítségével felderíti benne az egyes LASCC-ket, ezek listáját adva vissza. A 7-9. sorokban három szükséges feltétel kerül ellenőrzésre: nem indítunk körkeresést, ha nincs elfogadó állapot, s csomóponthoz tartozó LASCC (5.2.1. szakasz), vagy visszatérő állapot (5.2.2. szakasz). Ha minden szükséges feltétel teljesül, az egyes LASCC-kre külön-külön elvégezzük a szűkítő lépéseket megvalósító *DetectCircles* függvényt, csak az adott LASCC-n belül futó átmenetekkel dolgozva. Ez a függvény így már *csak a visszatérő állapotokból, csak a megfelelő átmenetekkel* képzett halmazokkal dolgozik, ami a korábbi lépésszámot jelentősen csökkenti.[8] Ha bármelyik ilyen keresés kört jelez, akkor a függvény igaz értékkel tér vissza, hiszen az elfogadó körhöz egy keresett lefutás is tartozik. Ha egyik LASCC-hez sem található tényleges (globális) elfogadó kör, akkor hamis érték jelzi, hogy az algoritmusnak tovább kell keresnie.

A szakasz zárásaképp megemlítjük, hogy a visszatérő állapotok és a lokális körök kihasználása elméletileg is jól kiegészíti egymást. A lokális körök egy absztrakt, a visszatérő állapotok pedig a valós állapotteret veszik alapul. A visszatérő állapotok által összegyűjtött állapotok kétfélék lehetnek: körben lévők, vagy több úton is elérhetők, a lokális körökből viszont kitűnhet, hogy az adott állapotok biztosan nem lehetnek körben. Ez fordítva is igaz: ahol a lokális körök szerint lehet kör, ott lehet, hogy nem keletkezik visszatérő állapot. Ennek eredményeképp – ahogy az a következő, a 6. fejezetben is látható – az esetek többségében az algoritmus nem fog feleslegesen szimbolikus körkeresést indítani.

[6] Az 5.2.1. szakaszban bemutatott bizonyításokban látható, hogy egy globális kör mindig egyetlen LASCC-n belül halad.

[7] A felhasznált halmazműveletek közvetlenül az MDD-n is megvalósíthatók, azonban ebben az algoritmusban a halmazokon van a hangsúly.

[8] Ne feledjük, hogy a legrosszabb esetben a halmazok méretének megfelelő számú szűkítő lépést kell végrehajtani!

Algoritmus 5 Az inkrementális körkeresés új algoritmusa

```
1: function IncrementalDetectCircle(mdd s, r)
2:     level l ← s.lvl;
3:     R = B(r);
4:     S ← B(s);
5:     F ← AcceptingStates(S);
6:     SCCs ← FindLASCCs(s, N_l);
7:     if F = ∅ then return False;      ▷ elfogadó állapot nélkül nem lehet elfogadó kör
8:     if R = ∅ then return False;                ▷ visszatérő állapotok szükséges feltétele
9:     if SCCs = ∅ then return False;                    ▷ lokális körök szükséges feltétele
10:    for each C_i^F ∈ SCCs do
11:        N_i ← TransitionsInside(C_i^F);
12:        if DetectCircles(R, F, N_i) then   ▷ ha C_i^F LASCC-hez található globális kör is
13:            return True;            ▷ elfogadó lefutást találtunk, az algoritmus leállhat
14:        end if
15:    end for
16:    return False;                        ▷ egyik LASCC-hez sem tartozott globális kör
17: end function
```

Algoritmus 6 A körkereséssel ellátott módosított vezérelt szaturációs algoritmus

```
1: function ProductSaturate(mdd s, c)
2:     if InCache_ProductSaturate(s, c, t) then return t;
3:     level k ← s.lvl; mdd t ← 0;
4:     for each i ∈ S_k : s[i] ≠ 0 do        ▷ először szaturáljuk az alsóbb csomópontokat
5:         if c[i] ≠ 0 then
6:             t[i] ← Saturate(s[i], c[i]);
7:         else                              ▷ ezen az ágon nem fogjuk tudni léptetni az automatát
8:             t[i] ← s[i];
9:         end if
10:    end for
11:    mdd r ← 0;                            ▷ a visszatérő állapotok tárolására
12:    repeat                                ▷ kiszámítjuk a lokális fixpontot
13:        for each i, i' ∈ S_k : N_k[i][i'] ≠ 0 do
14:            if c[i'] ≠ 0 then             ▷ ha még van esély léptetni az automatát
15:                mdd rc ← 0;
16:                t[i'] ← Union(t[i'], ProductRelProd(t[i], c[i'], N_k[i][i'], t[i'], rc));
17:                r[i'] ← Union(r[i'], rc);   ▷ r-be gyűjtjük a visszatérő állapotokat
18:            end if
19:        end for
20:    until t nem változik;
21:    t ← InsertUT(t); r ← InsertUT(r);
22:    CacheAdd_ProductSaturate(s, c, t);
23:    if IncrementalDetectCircles(t, r) kört jelez then
24:        Az algoritmus azonnal véget ér: a kifejezés nem érvényes a modellre.
25:    end if
26:    return t;
27: end function
28: function ProductRelProd(mdd s, c, n, u, r)
29:    level k ← s.lvl;
30:    mdd t ← 0;
31:    if k = 1 then                         ▷ ha az automatát kódoló szinten vagyunk...
32:        n ← c;                            ▷ ...az állapotátmeneti függvényt a kényszerből vesszük
33:    end if
34:    if s = 1 és n = 1 then                ▷ terminális eset
35:        if u = 1 then r ← t;              ▷ u-ban maradtunk, ez egy visszatérő állapot
36:        return t;
37:    end if
38:    if InCache_ProductRelProd(s, c, n, u, r, t) then return t;   ▷ r-t is kiolvassuk
39:    for each i, i' ∈ S_k : n[i][i'] ≠ 0 do
40:        if k = 1 vagy c[i'] ≠ 0 then      ▷ az automata szintjén már nem kell kényszer
41:            mdd rc ← 0;
42:            t[i'] ← Union(t[i'], ProductRelProd(s[i], c[i'], n[i][i'], u[i'], rc));
43:            r[i'] ← Union(r[i'], rc);
44:        end if
45:    end for
46:    r ← InsertUT(r); t ← ProductSaturate(InsertUT(t), c);
47:    CacheAdd_ProductRelProd(s, c, n, u, r, t);   ▷ r-t és u-t is lementjük
48:    return t;
49: end function
```

6. fejezet

Mérési eredmények

6.1. A generált specifikációs automata mérete

Ebben a fejezetben a specifikációs automata generálásával kapcsolatos újítások hatását mutatjuk be a korábbi eredményeinkhez képest. Az összehasonlítás alapjául az elkészített automaták mérete, azaz állapot- és tranzíciószáma szolgál, ugyanis a modellellenőrzés során ez kritikus a futásidő csökkentésének szempontjából [5]. A mérésekben egyrészt szerepel a korábbi munkánk során használt algoritmus, másrészt a jelen dolgozatban bemutatott fejlesztett tabló algoritmus, és végül a fejlesztett tabló algoritmus a specifikációs automata egyszerűsítésével kiegészítve.

A 6.1. táblázatban látható eredményekből látszik, hogy a vizsgált kifejezéshez viszonylag kis méretű automaták generálhatóak. Az új algoritmus a korábbihoz képest nem hozott javulást, viszont az automata egyszerűsítésével csökkenteni tudtuk az állapotok számát.

$\mathbf{F}(p \wedge \mathbf{X}\ p \wedge \ldots \wedge \mathbf{X}^{n-1}p) \wedge \mathbf{F}(q \wedge \mathbf{X}\ q \wedge \ldots \wedge \mathbf{X}^{n-1}q)$						
n	Korábbi		Új		Új+egyszerűsítés	
	állapotok	tranzíciók	állapotok	tranzíciók	állapotok	tranzíciók
1	14	29	14	29	4	9
2	21	38	21	38	9	16
3	30	49	30	49	16	25
4	41	62	41	62	25	36
5	54	77	54	77	36	49
6	69	94	69	94	49	64
7	86	113	86	113	64	81
8	105	134	105	134	81	100
9	126	157	126	157	100	121
10	149	182	149	182	121	144
11	174	209	174	209	144	169
12	201	238	201	238	169	196
13	230	269	230	269	196	225
14	261	302	261	302	225	256
15	294	337	294	337	256	289

6.1. táblázat. Egymásba ágyazott $\mathbf{X}$ operátorok vizsgálata

A 6.2. táblázatban vizsgált kifejezéshez az előző méréstől eltérően rendkívül nagy automatákat generál mind a korábbi, mind a jelenleg használt algoritmus. Bár némi javulást tapasztalunk a korábbi implementációhoz képest, a jelentős javulást az automata egyszerűsítése hozza, aminek alkalmazása egy nagyságrenddel kisebb automatát eredményez.

$\mathbf{F}(p_1 \wedge \mathbf{F}(p_2 \wedge \mathbf{F}(\ldots \wedge \mathbf{F}(p_n)\ldots))) \wedge \mathbf{F}(q_1 \wedge \mathbf{F}(q_2 \wedge \mathbf{F}(\ldots \wedge \mathbf{F}(q_n)\ldots)))$						
n	Korábbi		Új		Új+egyszerűsítés	
	állapotok	tranzíciók	állapotok	tranzíciók	állapotok	tranzíciók
1	14	29	14	29	4	9
2	55	161	51	152	9	36
3	176	661	149	577	16	100
4	484	2255	380	1839	25	225
5	1168	6681	856	5110	36	441

6.2. táblázat. Egymásba ágyazott **F**operátorok vizsgálata

Az eddigi eredmények alapján akár kétségbe is vonhatnánk az új automata generáló algoritmus létjogosultságát, hiszen komoly javulást csak az automaták egyszerűsítése hozott, de a 6.3. táblázatban látható eredmények rögtön eloszlathatják kételyeinket. Az bemutatott tabló algoritmus a korábbi munkáinkban felhasznált transzformációs algoritmushoz képest hatalmas eltérést mutat.

$\mathbf{GF}\ p_1 \wedge \mathbf{GF}\ p_2 \wedge \ldots \wedge \mathbf{GF}\ p_n$						
n	Korábbi		Új		Új+egyszerűsítés	
	állapotok	tranzíciók	állapotok	tranzíciók	állapotok	tranzíciók
1	4	8	4	8	2	3
2	21	84	14	56	6	18
3	127	1016	33	264	10	52
4	1529	24464	77	1232	21	207

6.3. táblázat. **GF** operátorok konjunkcióinak vizsgálata

A 6.4. táblázatban látható adatok egy múltidejű operátorokat is tartalmazó kifejezéshez tartozó mérés eredményei. Ebben az esetben az összehasonlítások alapjául szolgáló korábbi algoritmust nem tudtuk vizsgálni, hiszen az nem alkalmas PLTL kifejezések transzformációjára. Bár az előző mérésben látott automata méretbeli javulás nem elhanyagolható, az új algoritmus legnagyobb előnye mégis az, hogy képes PLTL kifejezések feldolgozására is.

$\mathbf{G}(\ldots(((p_1 \Rightarrow \mathbf{O}\ p_2) \Rightarrow \mathbf{O}\ p_3)\ldots) \Rightarrow \mathbf{O}\ p_n)$						
n	Korábbi		Új		Új+egyszerűsítés	
	állapotok	tranzíciók	állapotok	tranzíciók	állapotok	tranzíciók
1	-	-	2	2	1	1
2	-	-	5	14	4	11
3	-	-	7	21	5	14
4	-	-	17	94	12	65
5	-	-	27	156	17	94
6	-	-	60	558	34	326

6.4. táblázat. Egymásba ágyazott **O** operátorok vizsgálata

A mérések során láthattuk, hogy bizonyos esetekben az új automata generáló algoritmus, más esetekben az automatákat egyszerűsítő algoritmus használata eredményezte az automaták jelentős méretbeli csökkenését.

6.2. Összehasonlítás a korábbi eredményekkel

Ebben a szakaszban korábbi algoritmusunkhoz hasonlítjuk az új eredményeket. Sokféle újításunk miatt kétféle verzióval is végzünk méréseket. A táblázatokban látott rövidítések a következő változatokat jelölik:

- MC_{old}: régi automaták, régi állapottér-generálás, régi körkeresés;
- MC_{new}: új automaták, új bejárás, régi körkeresés;
- MC^{+}_{new}: új automaták, új bejárás, új körkeresés.

Az algoritmusokat három *benchmark modellen* hasonlítottuk össze, különböző kifejezésekre mérve a futási időt és a körkeresés során elvégzett szűkítő lépések összesített számát.

Étkező filozófusok						
φ_1	$(idhl2 = 0) \vee (idhr2 = 0)\ \mathbf{U}\ (idhl1 = 1) \wedge (idhr1 = 1)$					
n	MC_{old}		MC_{new}		MC^{+}_{new}	
	futási idő	szűkítések	futási idő	szűkítések	futási idő	szűkítések
10	0.011874	29	0.012611	6	0.002171	3
100	0.007042	29	0.007159	6	0.008990	3
1000	0.188666	29	0.233658	6	0.270331	3
φ_2	$\mathbf{G}((idhl1 = 0) \vee (idhr1 = 0))$					
n	MC_{old}		MC_{new}		MC^{+}_{new}	
	futási idő	szűkítések	futási idő	szűkítések	futási idő	szűkítések
10	0.002402	29	0.004780	29	0.002329	3
100	0.011531	29	0.012659	29	0.013352	3
1000	0.337093	29	0.396293	29	0.469839	3
φ_3	$\mathbf{G}(true)$					
n	MC_{old}		MC_{new}		MC^{+}_{new}	
	futási idő	szűkítések	futási idő	szűkítések	futási idő	szűkítések
10	0.001256	14	0.001449	14	0.001393	3
100	0.006701	14	0.007517	14	0.007715	3
1000	0.191258	14	0.250687	14	0.268880	3
φ_4	$(\mathbf{GF}(idhr1 = 1) \Rightarrow \mathbf{GF}((idhr1 = 1) \wedge (idhl1 = 1)))$					
n	MC_{old}		MC_{new}		MC^{+}_{new}	
	futási idő	szűkítések	futási idő	szűkítések	futási idő	szűkítések
10	0.009083	0	0.011010	0	0.012742	0
100	0.583198	0	0.691970	0	0.720421	0
1000	152.997752	0	157.917258	0	162.200126	0
φ_5	$(\mathbf{GF}(idhr2 = 0) \Rightarrow \mathbf{GF}((idhr1 = 1) \wedge (idhl1 = 1)))$					
n	MC_{old}		MC_{new}		MC^{+}_{new}	
	futási idő	szűkítések	futási idő	szűkítések	futási idő	szűkítések
10	0.007728	40	0.002259	10	0.003203	5
100	0.018233	40	0.008470	10	0.009405	5
1000	0.376540	40	0.243783	10	0.283584	5
φ_6	$(\mathbf{GF}(idhl2 = 0) \Rightarrow \mathbf{GF}((idhr1 = 1) \wedge (idhl1 = 1)))$					
n	MC_{old}		MC_{new}		MC^{+}_{new}	
	futási idő	szűkítések	futási idő	szűkítések	futási idő	szűkítések
10	0.009061	40	0.001795	9	0.003048	5
100	0.019580	40	0.009044	9	0.009031	5
1000	0.367247	40	0.236932	9	0.270461	5

6.5. táblázat. Az étkező filozófusok modellen végzett mérések

A klasszikus étkező filozófusok modell az egyik legegyszerűbb paraméterezhető modell, amelyet tradícionálisan modellellenőrző algoritmusok összehasonlítására alkalmaznak. Algoritmusainkat ezen a modellen futtatva az látható, hogy az újítások nem javítottak érdemben a futási időn. Sőt, a legtöbb esetben mindkét variáns egy kicsivel (konstansszor) rosszabbul is teljesít, annak ellenére, hogy a körkereső lépések száma az MC^{+}_{new} variáns esetén szinte mindig 3, egy megkezdett körkeresés legkevesebb számú iterációja (vagyis ennyi szűkítő lépés mindenképpen kell, ha egyszer elindult a keresés). Az utolsó három kifejezés ún. *fairness kritárium*, amit CTL modellellenőrzőkkel közvetlenül nem lehet ellenőrizni. Ezekre a nehezebb feldatokra az új algoritmusunk tudott a réginél jobban működni, és itt

az automatákat is lehetett úgy egyszerűsíteni, hogy ez pozitívan befolyásolja az eredményeket.

Ezen a példán jól látható, hogy az egyszerűség időnként célravezetőbb. Algoritmusaink egy ilyen egyszerű modellen a kifinomultabb megoldásokkal járó „overhead" miatt maradtak alul, az MC^{+}_{new} variáns például azért, mert a már az állapottér felderítése közben is a körkeresés gyorsításán dolgozik – erre viszont ebben a modellben, ezekkel a kifejezésekkel nem volt szükség.

Round robin						
φ_1	$(pload1 = 0)$ **U** $(psend0 = 1)$					
n	MC_{old}		MC_{new}		MC^{+}_{new}	
	futási idő	szűkítések	futási idő	szűkítések	futási idő	szűkítések
10	0.059245	965	0.055871	710	0.037303	3
25	0.257230	2600	0.250587	1865	0.125289	3
50	0.886377	5325	0.889508	3790	0.352550	3
100	2.059751	6985	2.206035	4951	1.291659	3
φ_2	**G**$(true)$					
n	MC_{old}		MC_{new}		MC^{+}_{new}	
	futási idő	szűkítések	futási idő	szűkítések	futási idő	szűkítések
10	0.034400	485	0.041978	485	0.026696	3
25	0.163795	1310	0.186382	1310	0.098304	3
50	0.563595	2685	0.670309	2685	0.313001	3
100	1.551102	3556	1.818607	3556	1.156434	3

6.6. táblázat. A round robin modellen végzett mérések

A round robin modell már jól mutatta fejlesztéseink megtérülését. Az egyszerű kifejezések miatt itt nem jelent meg az automata egyszerűsítések hatása, viszont a fejlettebb körkeresés egyértelműen meghozta eredményét. A szűkítő lépések száma itt sem haladta meg a minimális hármat.

Réselt gyűrű						
φ_1	$(idph1 = 0) \vee (idpc1 = 0)$ **U** $(idpd2 = 1) \wedge (idpg2 = 1)$					
n	MC_{old}		MC_{new}		MC^{+}_{new}	
	futási idő	szűkítések	futási idő	szűkítések	futási idő	szűkítések
10	0.477320	7409	0.493576	6750	0.145260	3
30	-	-	-	-	1.450843	3
50	-	-	-	-	4.815128	3
100	-	-	-	-	27.169605	3
φ_2	**G**$((idpa1 = 0) \vee (idpd1 = 0))$					
n	MC_{old}		MC_{new}		MC^{+}_{new}	
	futási idő	szűkítések	futási idő	szűkítések	futási idő	szűkítések
10	0.383388	5984	0.419428	5984	0.094269	3
30	-	-	-	-	1.172604	3
50	-	-	-	-	3.876432	3
100	-	-	-	-	24.181976	3
φ_3	**G**$(true)$					
n	MC_{old}		MC_{new}		MC^{+}_{new}	
	futási idő	szűkítések	futási idő	szűkítések	futási idő	szűkítések
10	0.375199	5974	0.415352	5974	0.087862	3
30	-	-	-	-	1.026006	3
50	-	-	-	-	3.645011	3
100	-	-	-	-	22.163440	3

6.7. táblázat. A réselt gyűrű modellen végzett mérések

A réselt gyűrű modell korábbi munkáink egyik legkomolyabb bukását jelentette. Jelen munkánk során így különös várakozásokkal tekintettünk erre a mérésre. Nem csalódtunk, fejlesztéseink itt elsöprő eredménnyel jártak. Az MC_{old} és MC_{new} variánsokra 10 résztvevő felett az ellenőrzések kifutottak a mérésre szánt egy órából, míg az MC^{+}_{new} variáns a 100 résztvevős modellt is könnyedén, kevesebb, mint fél perc alatt ellenőrizte. Ennek oka látványos: 10 résztvevő esetén az első két variáns által végrehajtott több ezer szűkítő lépést az új körkereső algoritmus itt is a minimális háromra redukálta.

6.3. Valós esettanulmányok

Eredményeink ellenőrzésére két valós esettanulmányt is kaptunk. Ezeket *színezett Petri-hálókkal* lehetett hatékonyan modellezni, így algoritmusainkat képessé tettük ennek a magasszintű formalizmusnak az ellenőrzésére is.

6.3.1. A PRISE modell

A PRISE modell a Paksi Atomerőmű egyik védelmi rendszere, melynek modelljén három kifejezést ellenőriztünk.

PRISE modell			
	kifejezés	futási idő	eredmény
φ_1	**GF**$(id365 : 1 = 1 \vee id365 : 0 = 1)$	7.047969	érvényes
φ_2	**GF**$(id365 : 1 = 1) \Rightarrow$ **GF**$(id342 : 1 = 1))$	7.5107504	kielégíthető
φ_3	**GF**$(id365 : 1 = 1) \Rightarrow$ **GF**$(id342 : 1 = 1))$	13,8227741	nem érvényes

6.8. táblázat. Fairness kritériumok a PRISE modellen

Az ellenőrzött kifejezések a rendszer működésének alapvető tulajdonságait ellenőrzik, közülük kettő fairness-kritárium, amit a tanszéken korábban kifejlesztett CTL modellellenőrző nem tudott közvetlenül ellenőrizni. Látható, hogy a modell nagy mérete ellenére a kifejezések ellenőrzése mindhárom esetben csak 10 másodperc körüli időt vett igénybe.

6.3.2. Egy CERN-től kapott esettanulmány

Kezdődő együttműködésünk jegyében a CERN nemzetközi kutatóintézettől is kaptunk Petri-háló alapú modelleket.[1] Ezeket PLC vezérlők kódjából automatikusan generálták, és a generáló algoritmus jelenleg is fejlesztés alatt áll. Emiatt mi egy kis modellt tudtunk ellenőrizni, amire új algoritmusunk gond nélkül lefutott.

PRISE modell			
	kifejezés	futási idő	eredmény
φ_1	$\mathbf{GF}((id92 : 1 = 1) \vee (id92 : 0 = 1))$	0,0575599	érvénytelen
φ_2	$\mathbf{GF}(id92 : 1 = 1) \Rightarrow \mathbf{GF}(id95 : 1 = 1)$	0,080578	kielégíthető
φ_3	$\mathbf{GF}(id92 : 1 = 1) \Rightarrow \mathbf{GF}(id95 : 1 = 1)$	0,0891802	nem érvényes

6.9. táblázat. Fairness kritériumok a PRISE modellen

[1] A modellekért köszönet illeti Darvas Dánielt.

7. fejezet

Összefoglalás

Dolgozatunk zárásaképp összefoglaljuk az elért eredményeket, illetve röviden ismertetjük a további kutatási irányokat.

7.1. Eredményeink

Dolgozatunk tárgya a modellellenőrzés automataelméleti megközelítésének továbbfejlesztése. Az egykor rendkívül népszerű LTL alapú modellellenőrző algoritmusok (explicit megközelítésük folytán) nagyméretű modelleken sok esetben nem tudnak lépést tartani a szimbolikus technikákat alkalmazó modellellenőrző megközelítésekkel. Szimbolikus megvalósítások ugyan léteznek ezen a területen, de az explicit megközelítések adta előnyöket ezek nem, vagy csak kis mértékben képesek nyújtani. Kutatásaink ezt az állapotot kívánták megváltoztatni. Az automataelméleti modellellenőrzés „felélesztésére" jelen dolgozat a klasszikus séma mindhárom részterületén jelentős fejlesztéseket vezetett be.

A specifikációt reprezentáló automatával kapcsolatos eredményeink

Az első ilyen részterület a specifikációt reprezentáló automata felépítése és kezelése. Dolgozatunkban a *PLTL formalizmust is támogató*, lineáris temporális tulajdonságokból Büchi-automatákat előállító algoritmusokat vizsgáltunk meg és hasonlítottunk össze. Ezeket *implementáltuk*, és a leghatékonyabbnak bizonyult továbbfejlesztett tabló algoritmust beépítettük a modellellenőrző algoritmusunkba. Az így előállított (vagy bármely más forrásból származó, például kézzel rajzolt) automatákat többször is felhasználhatjuk, így célszerű időt fektetni az egyszerűsítésükbe – ezt teszi meg a bemutatott *Büchi-automata egyszerűsítő algoritmusunk*, melynek helyességét *formálisan bizonyítottuk* is.

A szorzat állapottér kiszámításában elért eredményeink

Egy másik fontos terület a szorzat állapottér kiszámítása. Dolgozatunk egyik fontos kontribúciója a szorzat állapottér közvetlen, szaturáció alapú, szimbolikus számítása a magasszintű modellből és a specifikációs automatából. Új algoritmusunk *tetszőleges Büchi-automatával* és tranzíciós rendszerre leképezhető magasszintű modellel képes dolgozni, *egyszerre számítva* a modell és a szorzat állapottér elérhető állapotait. A szorzat állapotteret közvetlenül, a szinkron átmeneteket pedig *dekomponálva* kódolva kifejlesztettünk egy, a *vezérelt szaturációt kiterjesztő* algoritmust. Ennek segítségével a szorzat állapotteret szimbolikusan is hatékonyan tudjuk bejárni, megőrizve a szaturációs algoritmus legfontosabb előnyeit. Az 5.1.2. szakaszban módszerünk helyességét *formálisan is igazoltuk*.

Az elfogadó lefutások keresése terén elért eredményeink

A harmadik terület az elfogadó lefutások keresése. Dolgozatunk másik fontos kontribúciója egy ezt megvalósító *hibrid és inkrementális* körkereső algoritmus, amely a szimbolikusan végrehajtott műveletekben kihasználja a szaturáció hatékonyságát, a modell megfelelő *absztrakcióin* dolgozva pedig más előnyökkel rendelkező *explicit módszereket* is alkalmaz. A *visszatérő állapotok* észlelésével az algoritmus már lényegében az állapottér bejárása közben megkezdi a körök keresését, majd az állapottér lokális absztrakcióján dolgozva *lokális körök* keresésével hatékony explicit módszereket alkalmaz, végül az így csökkentett méretű problémát tovább *partícionálva* szimbolikus módszerekkel ellenőrzi a kör létezését. A hatékonyság abban rejlik, hogy körmentes állapotterek esetén a keresés sok esetben még a költséges szimbolikus műveletek megkezdése előtt bebizonyítja a kör hiányát. Az 5.2. szakaszban ezeknek a módszereknek a helyességét is *matematikai bizonyításokkal igazoltuk*.

Algoritmusaink ellenőrzése során elért eredményeink

Új algoritmusaink hatékonyságát a mérési eredményeinket ismertető 6. fejezetben illusztráltuk, ahol benchmark modelleken végzett összehasonlító mérések mellett magasszintű *színezett Petri-hálókat* alkalmazva ellenőriztük a Paksi Atomerőmű egyik védelmi rendszerének modelljét, illetve egy, a CERN nemzetközi kutatóintézettől kapott esettanulmányt is.

7.2. Továbbfejlesztési lehetőségek

A jelen dolgozatban bemutatott algoritmusok nagy előrelépést jelentettek a korábbi kutatásainkban elért eredményekhez képest, azonban számos továbblépési lehetőséget is feltártunk munkánk során. A további irányokat az alábbiakban foglaljuk össze:

- A jobb használhatóság érdekében a megtalált körbe vezető legrövidebb utat is ki lehetne számolni, hogy a hibás modell javítása az ellenpélda alapján könnyebb legyen.
- Modellellenőrző algoritmusunk jelenleg tetszőleges ω-reguláris kifejezéssel leírható viselkedésbeli követelményt képes ellenőrizni. Sok olyan specifikációs formalizmus létezik, amivel ilyen követelményeket kényelmesen meg lehet fogalmazni, így mindenképpen hasznos lenne ezekhez a formalizmusokhoz Büchi-automata konstruáló algoritmust adni.
- Az elméleti eredményeinket és a korábbi CTL-t alkalmazó kutatások tanulságait felhasználva meg kellene vizsgálni a CTL* logikai formulában felírt kifejezések modellellenőrzését, amely egy nehezebb problémaosztályba tartozik másféle algoritmusokkal.
- Az elméleti eredményeket felhasználva a kompozícionális verifikációhoz is közelebb juthatunk, a komponensek állapottereinek elfogadó körök szerinti absztrahálásával.

Az első két pont megvalósításába várhatóan a közeljövőben belekezdünk, míg az utolsó két pont teljesen új, önálló kutatási témaként is megállja a helyét.

Ábrák jegyzéke

Irodalomjegyzék

[1] 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.

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

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

[4] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2001.

[5] Rüdiger Ehlers and Bernd Finkbeiner. On the virtue of patience: Minimizing büchi automata. In Jaco van de Pol and Michael Weber 0002, editors, *SPIN*, volume 6349 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2010.

[6] 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.

[7] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In Costas Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 97–109. Springer Berlin Heidelberg, 1993.

[8] François Laroussinie, Nicolas Markey, and Ph. Schnoebelen. Temporal logic with forgettable past. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 383–392, Washington, DC, USA, 2002. IEEE Computer Society.

[9] Orna Lichtenstein and Amir Pnueli. Propositional temporal logics: Decidability and completeness. *Logic Journal of the IGPL*, 8(1):55–85, 2000.

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

[11] Molnár Vince. Szaturáció alapú modellellenőrzés lineáris idejű tulajdonságokhoz. *Tudományos Diákköri Konferencia, Budapesti Műszaki és Gazdaságtudományi Egyetem, Villamosmérnöki és Informatikai Kar*, 2012.

[12] Molnár Vince. Szaturáció alapú modellellenőrzés lineáris idejű tulajdonságokhoz. *Szakdolgozat, Budapesti Műszaki és Gazdaságtudományi Egyetem, Villamosmérnöki és Informatikai Kar*, 2013.

[13] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.

[14] 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.

[15] Yang Zhao and Gianfranco Ciardo. Symbolic computation of strongly connected components and fair cycles using saturation. *ISSE*, 7(2):141–150, 2011.