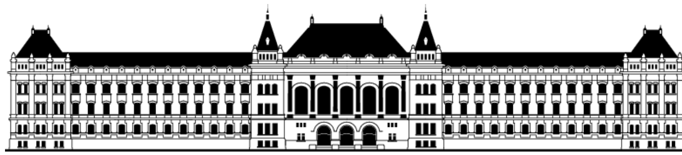


Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Aszinkron rendszerek heurisztikával támogatott modellellenőrzése

SZAKDOLGOZAT

Készítette:
Szabó Tamás
IV. évfolyam
mérnök-informatika

Konzulensek:
Vörös András
doktorandusz

Horváth Ákos
tudományos segédmunkatárs

2010. december 10.

SZAKDOLGOZAT-FELADAT

Szabó Tamás
mérnök informatika szakos hallgató részére

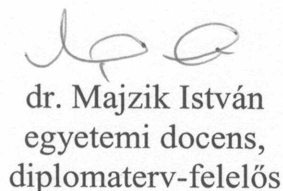
Aszinkron rendszerek heurisztikával támogatott modellellenőrzése
(A feladat szövege a mellékletben)

A szakdolgozat-feladatot összeállította és tanszéki konzulense:

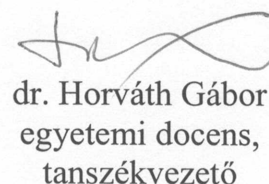
Vörös András
doktorandusz

A záróvizsga tárgya: Adatvezérelt alkalmazások fejlesztése:(bmeviaua369)

A szakdolgozat-feladat kiadásának napja: 2010. szeptember 6.
A szakdolgozat beadásának határideje: 2010. december 10.



dr. Majzik István
egyetemi docens,
diplomaterv-felelős



dr. Horváth Gábor
egyetemi docens,
tanszékvezető

A szakdolgozatot bevette:
A szakdolgozat beadásának dátuma:
A szakdolgozat bírálója:

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

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Aszinkron rendszerek heurisztikával támogatott modellellenőrzése

Szakdolgozat-feladat melléklete

Napjainkban a beágyazott és ezen belül is az elosztott, aszinkron, misszió kritikus rendszerek tervezésénél egyre nagyobb szerepet kapnak a különböző formális módszereken alapuló technikák. Legnagyobb előnyük, hogy már a tervezés kezdeti fázisától képesek a rendszer helyes működésének a vizsgálatára és verifikálására.

Ezen a téren az egyik legszélesebb körben elterjed formális módszer a modellellenőrzés. A modellellenőrzés során a rendszer egy véges állapotterű modelljén vizsgáljuk a követelmények teljesülését. A szimbolikus, azaz döntési diagram alapú szaturációs algoritmus is egy modellellenőrző algoritmus, amelyet elosztott rendszerekhez fejlesztettek ki.

A Méréstechnika és Információs Rendszerek tanszéken fejlesztett *PetriDotNet* keretrendszer Petri hálók szerkesztésére, szimulációjára és analízisére szolgáló program, amely rendelkezik a szaturációs algoritmust megvalósító modullal. Ez azonban az algoritmust szekvenciálisan hajtja végre, nem használja ki a ma már elterjedt többprocesszoros gépek és többmagos processzorok jelentette többleterőforrásokat.

A hallgató feladata a szaturációs állapottérgeneráló algoritmus vizsgálata, párhuzamosításának kidolgozása, megvalósítása, továbbá a modellellenőrzés más területein sikerrel alkalmazott heurisztikák vizsgálata, alkalmazása, a hatékonyabb párhuzamos állapottérgenerálás érdekében.

A hallgató feladatának a következőkre kell kiterjednie:

- Mutassa be szaturációs állapottérgeneráló algoritmust, és annak párhuzamos változatát
- Implementálja a megtervezett párhuzamos szaturációs állapottérgeneráló algoritmust
- Dolgozzon ki szaturáció alapú heurisztikákat, amelyek építenek a szimbolikusan kódolt részállapothalmazokban tárolt információra
- Implementálja a kidolgozott heurisztikákat a *PetriDotNet* keretrendszerbe
- Mérésekkel vizsgálja meg a heurisztikák hatékonyságát

Vörös András
doktorandusz

HALLGATÓI NYILATKOZAT

Alulírott Szabó Tamás, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, és a szakdolgozatban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik.

Budapest, 2010. december 10.

Szabó Tamás
hallgató

Tartalomjegyzék

1 BEVEZETÉS ... 1

2 HÁTTÉRISMERETEK ... 3

2.1 A formális módszerek szerepe az informatikai rendszerek tervezésében ... 3
2.1.1 Korábbi állapottér-felderítő algoritmusok ... 5
2.2 Aszinkron rendszerek ... 6
2.3 Petri-hálók ... 7
2.3.1 A Petri-hálók eredete ... 7
2.3.2 A Petri-hálókról általában ... 7
2.3.3 A Petri-hálók struktúrája, definíció ... 7
2.3.4 A Petri-hálók dinamikus viselkedése ... 8
2.3.5 Tüzelési szekvencia ... 10
2.3.6 Petri-hálók invariánsai ... 10
2.4 Döntési diagramok és alkalmazásaik a modellellenőrzésben ... 11
2.4.1 Bináris döntési diagramok ... 11
2.4.2 Többértékű döntési diagramok ... 14
2.5 Állapottér-bejárás ... 16
2.5.1 Állapottér-felderítés explicit módszerekkel ... 16
2.5.2 Állapottér-felderítés szimbolikus módszerekkel ... 17

3 A SZATURÁCIÓS ALGORITMUS ... 19

3.1 Az állapottér dekompozíciója ... 19
3.2 Az események lokalitása ... 20
3.3 A helyben frissítés módszere (in-place update) ... 21
3.4 Rekurzív mélységi állapottér-felderítés ... 21
3.5 Szaturáció ... 22

4 A PÁRHUZAMOS SZATURÁCIÓS ALGORITMUS ... 25

4.1 Az algoritmus bemutatása ... 25
4.2 Implementációs részletek ... 26
4.2.1 Várakozási sor bevezetése a feladatok végrehajtásához ... 26
4.2.2 Csomópont adatszerkezeteinek kiegészítése ... 26
4.2.3 A tüzelési gyorsítótár kiegészítése ... 26
4.2.4 Az MDD tároló kiegészítése ... 27
4.2.5 Az algoritmus főbb függvényeinek bemutatása ... 27
4.3 A részgráfzárolás továbbfejlesztése ... 29
4.3.1 A részgráfok zárolása ... 30
4.3.2 Az írási-olvasási zárak alkalmazása ... 30
4.3.3 Lokális szinkronizáció ... 31

5 HEURISZTIKÁN ALAPULÓ TRANZÍCIÓ-ELŐRETÜZELÉS ... 33

5.1 Korábbi heurisztikán alapuló megoldások ... 34
5.2 Az előretüzelés működésének bemutatása ... 34
5.3 Az eltüzelendő tranzíció kiválasztása, a heurisztikák bemutatása ... 36
5.4 Az előretüzelés implementációja ... 39
5.4.1 Saját threadpool implementáció bemutatása ... 41
5.4.2 A szálak menedzselése ... 41

6 MÉRÉSI EREDMÉNYEK ... 45

6.1 A mérési környezet ... 45
6.2 A vizsgált modellek ... 45
6.3 A mért értékek ... 45
6.4 Slotted Ring modell ... 46

6.5 FMS MODELL 48
6.6 KANBAN MODELL 50
6.7 A MÉRÉSI EREDMÉNYEK ÉRTÉKELÉSE 51

7 ÖSSZEFOGLALÁS 53

IRODALOMJEGYZÉK 54

ÁBRÁK JEGYZÉKE 56

FÜGGELÉK A 57

FÜGGELÉK B 65

Kivonat

Napjaink informatikai alkalmazásaival szemben támasztott fontos követelmény a szolgáltatásbiztonság, azaz annak az igénye, hogy mindig, igazoltan bízni lehet a rendszer által nyújtott szolgáltatásokban. A szoftver és hardver rendszerek azonban mára olyan komplexitást értek el, hogy támogatás nélkül azt a fejlesztőmérnökök már nem képesek átlátni. Manapság a probléma megoldására a különböző formális módszereken alapuló megoldások terjedtek el leginkább. A szakdolgozat ezek közül a modellellenőrzés témakörével foglalkozik részletesen, amely egy automatizált verifikációs módszer a rendszerrel szemben támasztott követelmények kimerítő ellenőrzéséhez. Ehhez a legtöbb esetben szükség van arra, hogy a rendszer elérhető állapotait előállítsuk, ez után nyílik lehetőség a kívánt tulajdonság ellenőrzésére [6][7].

A modellellenőrzés során problémát jelent az úgynevezett állapottér robbanás nevű jelenség, amely azt mondja, hogy a rendszer komplexitásának növekedésével a rendszer állapotterének felderítéséhez szükséges futási idő és az állapotok tárolásához szükséges tárigény exponenciálisan nő. Ezen nehézségek orvoslására számos irányban indultak el a kutatók. Az idő –és tárigény problémáját hatékonyan lehet kezelni az úgynevezett szaturációs algoritmus [4][5] alkalmazásával, amelyet aszinkron rendszerek állapotterének felderítéséhez fejlesztettek ki. Jogos igényként merül fel a párhuzamos modellellenőrző kialakítása is, hiszen a manapság használatos hardverek nagy része már több processzormaggal rendelkezik. A TDK dolgozatunkban a szaturációs algoritmus párhuzamosítási lehetőségeivel foglalkoztunk [2], amelynek fontosabb részei a szakdolgozatban is bemutatásra kerülnek.

A modellellenőrzés során a futási idő csökkentésében a párhuzamos megoldások mellett nagy szerepet kapnak a különböző heurisztikán alapuló megoldások is. A szakdolgozatban részletesen bemutatásra kerül az úgynevezett heurisztikán alapuló tranzíció-előretüzelés módszere, amely a TDK dolgozat egyik megjelölt lehetséges folytatási iránya volt. Az algoritmus motivációját az adta, hogy a párhuzamos szaturációs algoritmus egyes modellek esetén nem használta ki teljesen a rendelkezésre álló számítási erőforrásokat. Ezeket a tartalék erőforrásokat arra lehetne kihasználni, hogy az állapottér egyes részeit előre felderítésük, ezzel tovább gyorsítva a párhuzamos algoritmus futását. Ebben kapnak szerepet a heurisztikák, ugyanis nem triviális az, hogy milyen módon kell az állapottér részeket előre felderíteni úgy, hogy az ténylegesen hasznos legyen később.

A szakdolgozat keretében különféle heurisztikákat dolgoztam ki és ezeket különböző statisztikák alapján hasonlítottam össze. Az elkészült modult integráltam a Méréstechnika és Információs Rendszerek Tanszéken fejlesztett *PetriDotNet* modellellenőrző keretrendszerbe [1] és több modellre is összehasonlítottam a meglevő szekvenciális és párhuzamos állapottér-felderítő algoritmusok futási sebességét az előretüzeléses változatéval. A mérések alapján elmondható, hogy több esetben sikerült valós sebességnövekedést elérni a párhuzamos változathoz képest, de olyan eset is volt, amelyre a szekvenciális változat ugyan gyorsabb volt, mint a párhuzamos, de az előretüzeléssel is kiegészítve a párhuzamos algoritmust már nem volt tapasztalható jelentős különbség a két változat futási idejében.

Abstract

Nowadays, hardware and software systems require a high degree of assurance of design correctness. However, these systems have reached such a complexity that IT professionals are no longer able to handle the whole lifecycle of the product without support. The verification of these systems is often based on formal methods, and especially on model checking [6][7], which is a widely used approach to investigate the behaviour of discrete state models. State-space generation is an essential prerequisite to verify the requirements against the system. Nevertheless the exhaustive generation of complex systems' state-spaces is extremely hard because of the time and storage requirements. The state space explosion phenomenon describes this behavior: as the complexity of a system increases, the memory and time required to store the combinatorically expanding state space can easily become excessive.

According to the newest experimental results, the saturation algorithm [4][5] can be efficiently used to overcome the limitations of memory and time requirements. Nevertheless, most of today's hardware are built with multi-core processors so the need for parallel algorithms is growing to utilize the additional resources and to shorten the runtime length of the state-space generation. Formerly, we made a Scientific Students' Association Report which investigates the parallelization of the saturation algorithm [2]. The main aspects of the parallel algorithm are described in this paper too.

Beyond parallel algorithm heuristics based model checkers are used to shorten the time required to investigate the behavior of the system. As part of my work I developed a herustics based algorithm for transition pre-firing. The motivation of the algorithm was given by the results of the parallel saturation algorithm's profiling, which carried out that in certain scenarios not all of the computational resources were exploited during the state space generation. One may manage to utilize these additional resources to pre-generate parts of the system's state space, so that it will be useful for the saturation algorithm and reach further performance improvement. However, the applied heuristics have an important role as it is a nontrivial task to find the useful parts.

In this paper I experimented several heuristics, measured and compared their efficiency based on statistics. I integrated the pre-firing modul into the *PetriDotNet* [1] framework developed at the Department of Measurement and Information Systems. As part of my work I compared the runtime length of the formerly implemented sequential and parallel algorithms against the one that exploits pre-firing too, with the use of several real-world systems' models. The experimental results showed that I managed to achieve additional speedups in the generation of several models' state space. In some cases the sequential algorithm was faster than the parallel one, but with the addition of pre-firing in the parallel saturation there wasn't significant difference in the runtime length anymore.

1 Bevezetés

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

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

A rendszer verifikációja és validációja

Az imént említett problémák orvoslására sokféle szoftver- és rendszerfejlesztési metodika született már, amelyek többé-kevésbé garantálni tudják a kiinduló specifikáció és az implementációs modellek ekvivalenciáját. Napjainkban azonban egyre inkább nő a *szolgáltatásbiztonság* iránti igény, amely magával vonta a *formális módszerek* alkalmazását. Ilyen esetben a rendszer működését valamilyen precíz matematikai formulával írjuk le (Petri-háló [6][7][8][9], formálisan helyes UML szekvencia diagram [24], állapotgépek [26]). Ezen alapulva célunk a rendszer helyes működésének igazolása. Az ellenőrzés során két alapvető megközelítést illetve azok kombinációját lehet alkalmazni: *verifikációt* és *validációt*. A verifikáció elvégzéséhez többféle módszer terjedt el, többek között ilyen a tesztelés, szimuláció, modellellenőrzés és a tételbizonyítás.

Jelen dolgozat a *modellellenőrzés* témakörével foglalkozik részletesebben. A modellellenőrzés során a rendszer egy véges modelljén bizonyítjuk be, hogy a megkívánt tulajdonság teljesül-e, például párhuzamosan dolgozó komponensek esetén előfordulhat-e, hogy a rendszer holtpontra jut, közösen használt erőforrások esetén felléphet-e kiéheztetés. Ehhez szükség van először a rendszer elérhető állapotainak felderítésére illetve tárolására, csak ez után lehetséges a rendszerrel szemben támasztott követelmények ellenőrzése. A jelenlegi iparban használt alkalmazások állapottere azonban olyan nagy, hogy rendkívül kifinomult és komplex algoritmusokra van szükség a feladatok megoldásához [25].

A rendszer állapotterének felderítése

A formális módszereken alapuló állapottér-felderítés két alapvető nehézsége:

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

A futási időre megoldást jelenthetnek a párhuzamos modellellenőrző megoldások. Ez abból a szempontból is érthető igény, hogy a mai hardverek nagy része már több processzorral vagy többmagos processzorral rendelkezik. A korábban elkészült TDK dolgozat [2] az úgynevezett szaturációs algoritmus párhuzamosítási lehetőségeivel foglalkozik, amely egy szimbolikus technika a rendszer állapotterének felderítésére. A modellellenőrző algoritmusok két nagy csoportba sorolhatók:

- Az *explicit technikák* a rendszer állapotait és az átmeneti információkat egyenként, explicit módon tárolják el, amelyet szélességi vagy mélységi bejárás során derítenek fel.
- Ezzel szemben a *szimbolikus technikák* implicit módon, kódolva tárolják a rendszer állapotait. Ehhez valamilyen kompakt, tárterület szempontjából hatékonyabb adatstruktúrát használnak, például *döntési diagramokat* (decision diagram) [6], hash táblát.

A szakdolgozat egy olyan technikát mutat be, amellyel a párhuzamos algoritmusok hatékonysága tovább növelhető; *heurisztikával támogatott modellellenőrzés*. Ebben az esetben az állapottér felderítése során a párhuzamos algoritmus alkalmazásakor a maradék számítási erőforrásokat arra használjuk fel, hogy a rendszer állapotterét előre, heurisztikus módon felderítsük.

A dolgozat céljai, felépítése

- A 2. fejezet áttekinti mindazokat a formális módszerekhez kapcsolódó háttérismereteket, amelyek szükségesek a szaturációs algoritmus – mely a 0. fejezetben kapott helyet – megértéséhez; döntési diagramok, Petri-hálók és állapottér-felderítés.
- A TDK dolgozatban elkészült párhuzamos szaturációs algoritmus bemutatására a 4. fejezetben kerül sor. Ebben a részben bemutatásra kerülnek a főbb függvények, az algoritmikai megfontolások és a hozzá kapcsolódó fejlesztéseink.
- A TDK dolgozat lehetséges folytatási irányaként megjelölt heurisztikán alapuló tranzíció-előretüzelés algoritmusa az 5. fejezetben kerül bemutatásra. Ebben a fejezetben részletesen bemutatom a saját algoritmikai és implementációs eredményeimet az előretüzeléses algoritmushoz kapcsolódóan. Az algoritmushoz kapcsolódóan többféle heurisztikát is kidolgoztam, amelyek szintén részletesen bemutatásra kerülnek. A szakdolgozat keretében elkészült állapottér-felderítő modult integráltam a Budapesti Műszaki és Gazdaságtudományi Egyetem Méréstechnika és Információs Rendszerek Tanszékén fejlesztett *PetriDotNet* [1] modellellenőrző keretrendszerbe. Ez az eszköz Petri-hálókat használ a rendszer modelljének leírására.
- A 6. fejezetben található mérési eredmények kettős célt szolgálnak:
    - A szaturációs algoritmus különféle változatainak összehasonlítása egymással: előretüzelést is alkalmazó párhuzamos, előretüzelés nélküli párhuzamos és szekvenciális szaturációs algoritmusok.
    - Az előretüzeléshez kapcsolódó heurisztikák hatékonyságának vizsgálata.
- A 7. fejezetben az összefoglalás után megjelölöm a szakdolgozat lehetséges folytatási irányait.

2 Háttérismeretek

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

A 2.1. fejezet áttekintést nyújt a különböző verifikációs technikákról. A 2.2. fejezet az aszinkron rendszerek fontosabb tulajdonságait részletezi. A *PetriDotNet* modellellenőrző keretrendszer a Petri-hálókat alkalmazza a rendszer modellezésére, ezért a 2.3. fejezet a szükséges háttérismereteket tartalmazza a Petri-hálókról.

A 2.4 fejezet mutatja be a döntési diagramokat, amelynek a 3. fejezetben ismertetett szaturációs algoritmusban is fontos szerepe van. A 2.5. fejezet pedig bemutat néhány explicit és szimbolikus algoritmust a rendszer állapotterének felderítéséhez.

2.1 A formális módszerek szerepe az informatikai rendszerek tervezésében

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

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

A komplex rendszerek validációjának és verifikációjának eszközei többek között:

- tesztelés
- szimuláció
- tételbizonyítás
- modellellenőrzés.

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

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

A *tételbizonyítás*sal [6] axiómákból kiindulva, valamilyen matematikai logika alkalmazásával lehetséges a rendszer helyes működésének igazolása. Ez az eljárás egy rendkívül időigényes folyamat, amely nagy szaktudást is igényel, így csak olyan rendszerek esetén alkalmazzák, ahol ez igazán szükséges. Jellemzően ilyen alkalmazási területek a biztonsági protokollok helyes működésének igazolása. Egy ilyen folyamat hetekig, hónapokig is eltarthat. Fontos azonban megemlíteni azt, hogy a rendszerrel szemben támasztott, automatikusan verifikálható követelmények tere véges, például nem létezik olyan algoritmus, amely képes bizonyítani, hogy egy program futása során valamikor biztosan fog-e terminálódni. A tételbizonyítással végtelen állapotterű rendszerek esetén is lehet érvelni valamely tulajdonság igazságértékével kapcsolatban, illetve ezen a területen is léteznek már automatizált eszközök, például a FOL (First Order Logic, elsőrendű logika) megoldók [27].

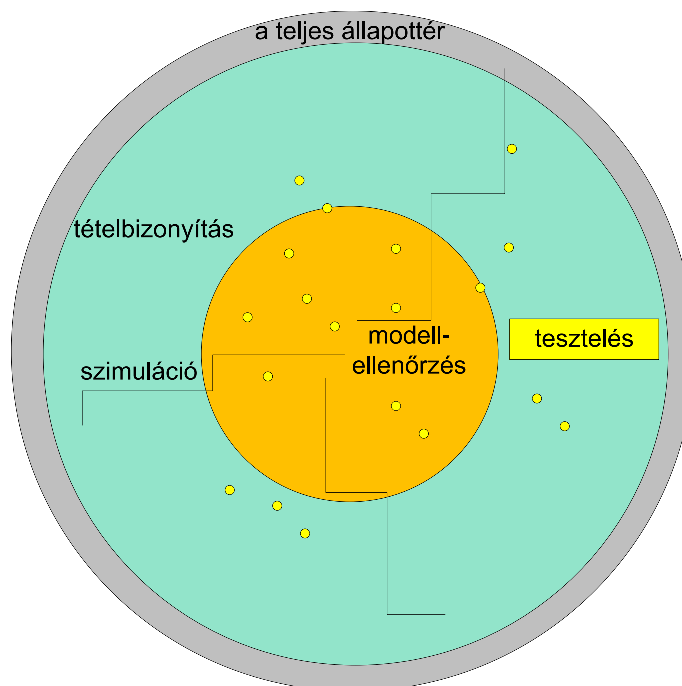
A *modellellenőrzés* egy olyan technika, amely a véges állapotterű rendszerek esetén tudja vizsgálni a követelmények teljesülését vagy nem teljesülését. A modell végessége miatt a folyamat automatizálható és kellő méretű erőforrások biztosítása mellett a modellellenőrző algoritmus biztosan terminálódni fog. A problémát az jelenti, hogy a mai ipari alkalmazások olyan méretű állapottérrel rendelkeznek, amelyek kimerítő felderítése nehéz feladat, mivel az erőforrások szükséges mérete nem, vagy csak nehezen biztosítható.

A négy ismertetett módszer egymáshoz való viszonyát szemlélteti az 1. ábra. A rendszer lehetséges állapotai közül az egyszerű tesztelés csak 1-1 szűk állapotcsoportot tud megvizsgálni (az ábrán a kis, sárga pontoknak felel meg). A rendszer helyes működésének bizonyíthatósága így jól látható módon meglehetősen időigényes és költséges feladat.

A szimuláció a rendszer egy állapotából kiindulva lefutási utakat tud megvizsgálni, ezáltal az állapotok egy véges halmazát vizsgálja meg (az ábrán törött vonal szemlélteti).

A modellellenőrzés ezekkel szemben a rendszer állapotterének egy korlátos részén vizsgálja meg a követelmények teljesülését vagy nem teljesülését, habár igaz, hogy csak egy korlátos részét ellenőrzi, de manapság egyre többen a small scope hypothesisre [22] alapozva azt mondják, hogy így is jól lehet következtetni a teljes rendszer hibamentességére (az ábrán narancssárga színű kör szemlélteti).

A tételbizonyítás módszere még ennél is nagyobb állapotteret fed le, de alkalmazása meglehetősen költséges és időigényes feladat, amely speciális szaktudást is igényel (az ábrán zöld színű kör szemlélteti) és általában csak nagyon indokolt esetben használják nagyméretű rendszereknél (lásd a biztonsági szabványnak számító *common criteria* legmagasabb *EAL6* és *EAL7* szintű megfelelőségnek a bizonyítására [23]).



1. ábra: A tesztelés, szimuláció, modellellenőrzés és tételbizonyítás szemléltetése

Mindezek alapján elmondható, hogy a legoptimálisabb megoldás a különféle technikák együttes alkalmazása, de ez a legtöbb esetben nem megoldható. A modellellenőrzés erőssége pont abban rejlik, hogy automatizált módon képes elfogadható bizonyossággal ellenőrizni a rendszer működését. A modellellenőrzés során az esetek többségében szükség van arra, hogy a rendszer elérhető állapotainak halmazát először legeneráljuk és eltároljuk, csak ez után lehetséges a rendszerrel szemben támasztott követelmények ellenőrzése. A következő fejezet a különféle állapottér-felderítést végző algoritmusokat foglalja össze.

2.1.1 Korábbi állapottér-felderítő algoritmusok

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

- explicit
- szimbolikus

technikákat.

Az explicit technikák a rendszer egyes elérhető állapotait egyenként tárolják el. Ennek hátránya, hogy viszonylag kisméretű modell esetén is az állapotok száma hatalmas méretűre nőhet, ezért a rendelkezésre álló erőforrások hamar elfogyhatnak. Ezen probléma orvoslására számos kutatás történt, melyek azonban csak korlátozott sikereket értek el. Az egyik kutatási irányban [20], azt használták ki, hogy az állapottérben fellelhető bizonyos fokú szimmetria esetén az eredeti állapotok halmazának csak

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

A szimbolikus technikák az elérhető állapotokat kódolják és a tároláshoz valamilyen kompakt, tárigény szempontjából kedvezőbb adatstruktúrát használnak fel (hash-tábla, döntési diagramok). Ennek következménye, hogy a szimbolikus technikák [4][5][6] a hatékonyabb tárolás miatt már nagy állapottér mérettel is meg tudnak birkózni, de a memória és futási időbeli korlátok itt is jelentkeznek. Ezen a területen is számos módszerrel próbáltak javítani az algoritmusokon a kutatók. Említésre méltó például az az ötlet, hogy az iterációk számát próbáljuk meg csökkenteni, melyen belül elérhető a rendszer összes állapota. Ehhez egy statikus elemzés alapján a tranzíciók sorrendezhetőek, így növelve az esélyét annak, hogy új állapotot derítünk fel az egymást követő iterációk során.

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

A dolgozatban részletesen bemutatásra kerül a heurisztikán alapuló állapottér-felderítés módszere is, amellyel mind a szekvenciális, mind a párhuzamos algoritmusok hatékonysága tovább növelhető. Ebben az esetben arról van szó, hogy a ki nem használt erőforrásokat arra használjuk fel, hogy a döntési diagram egy-egy részét előre felépítsük ezáltal csökkentve az algoritmusok futási idejét. A módszer hatékonysága nagymértékben függ az alkalmazott heurisztikától, a saját implementációban bemutatásra kerül több aszinkron rendszerek esetén alkalmazható heurisztika is.

2.2 Aszinkron rendszerek

A napjainkban használatos informatikai rendszerek nagy része aszinkron rendszer. Az ilyen rendszerek a szolgáltatásaikat több, egymástól függetlenül működő komponens együttműködésével nyújtják. Aszinkron rendszer például egy repülőgép irányító berendezése, ahol a sok, különböző mérőműszer egymástól függetlenül működik, de adatcsere céljából meg kell oldani közöttük a kommunikációt. Ugyanakkor egy hálózati protokoll is egy aszinkron rendszernek feleltethető meg, ahol a független komponensek maguk a kommunikációban résztvevő számítógépek, és tetszőleges időpillanatban kezdeményezhet valaki kommunikációt. A dolgozatban például találhatók mérési eredmények a *Slotted Ring* [19] hálózati protokollra állapotterének felderítési idejére vonatkozóan.

Összefoglalva az aszinkron rendszerek tulajdonságait:

- Egymástól függetlenül működő komponensek.
- Nincs globális óra, az egyes komponensek csak saját lokális órával rendelkeznek.
- A komponensek egymással valamilyen csatornán keresztül, aszinkron módon kommunikálnak. Itt külön nehézséget jelent a komponensek közötti szinkronizáció megoldása a globális óra hiánya miatt.

A leírt tulajdonságok miatt az aszinkron rendszerekre a nemdeterminisztikus működés jellemző. Ezen rendszerek állapottere az elosztottság miatt hatalmas méretű lehet, ennek következményeként a hibakeresés és a szolgáltatásbiztonság garantálása is sokkal nehezebb.

Az aszinkron rendszerek modellezésére jól használhatók a Petri-hálók, amelyet a következő fejezet részletez.

2.3 Petri-hálók

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

2.3.1 A Petri-hálók eredete

A Petri-hálók alapjait Carl Adam Petri nevű, német matematikus dolgozta ki 1939-ben. Eredetileg kémiai folyamatok leírására szánta, a matematikai alapjait doktori disszertációjában dolgozta ki 1962-ben [10].

2.3.2 A Petri-hálókról általában

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

Petri-hálók használatával

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

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

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

2.3.3 A Petri-hálók struktúrája, definíció

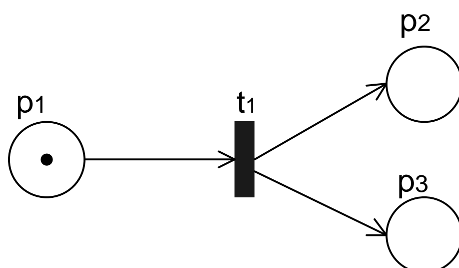
A Petri-háló struktúráját tekintve egy irányított, súlyozott élű páros gráf. A gráfban két fajta csomópont lehet: hely $(p \in P)$ és tranzíció $(t \in T)$. A helyeket a Petri-hálóban körrel ábrázoljuk, a tranzíciókat téglalappal. Az irányított élek mehetnek helyből

tranzícióba vagy tranzícióból helybe (páros gráf). Ezek alapján a Petri-hálók formális definíciója:

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

- $P = \{p_1, p_2, \dots, p_n\}$ az állapotok véges számú halmaza,
- $T = \{t_1, t_2, \dots, t_n\}$ a tranzíciók véges számú halmaza,
- $F \subseteq (P \times T) \cup (T \times P)$, itt F az élek véges számú halmaza, az $\times$ művelet a halmazok közötti Descartes szorzatot jelöli,
- $P \cap T = \emptyset$ és $P \cup T \neq \emptyset$.

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



2. ábra: Egy egyszerű Petri-háló

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

- egy $p \in P$ hely ősei a bemenő tranzíciói: $\bullet p = \{t|(t, p) \in E\}$
- egy $p \in P$ hely utódai a kimenő tranzíciói: $p\bullet = \{t|(p, t) \in E\}$
- egy $t \in T$ tranzíció ősei a bemenő helyei: $\bullet t = \{p|(p, t) \in E\}$
- egy $t \in T$ tranzíció utódai a kimenő helyei: $t\bullet = \{p|(t, p) \in E\}$

A 2. ábrán látható példára $\bullet p_1 = \{\}$, $\bullet p_2 = \{t_1\}$ és $\bullet p_3 = \{t_1\}$.

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

Az állapotváltozók szerepét az ún. token tölti be (grafikusan ezt a hely körébe rajzolt ponttal jelezzük). Egy hely állapota a benne levő tokenek számával egyezik meg. A háló állapota egy token eloszlással jellemezhető, $\overline{M_0}: P \rightarrow N$ leképezés, ahol N a természetes számok halmazát jelöli. $\overline{M_0}$ tehát egy $|P|$ elemű vektor és azt adja meg, hogy az egyes helyeken hány darab token van.

Az élekhez ugyanakkor súlyt is rendelhetünk, amely egy pozitív természetes szám lehet. A $w(e) = k$ élsúly azt jelenti, hogy k darab él fut párhuzamosan a két hely között. Az egyszeres súlyokat nem szokás feltüntetni, a többszörös súlyokat az élre írjuk rá. Formálisan a súlyfüggvény a $W: E \rightarrow N^+$ leképezéssel adható meg.

2.3.4 A Petri-hálók dinamikus viselkedése

A háló állapotváltozása tranzíciók tüzelésével történhet. Egy tranzíció akkor engedélyezett, ha minden bemeneti helyen elégséges számú token van. Minden

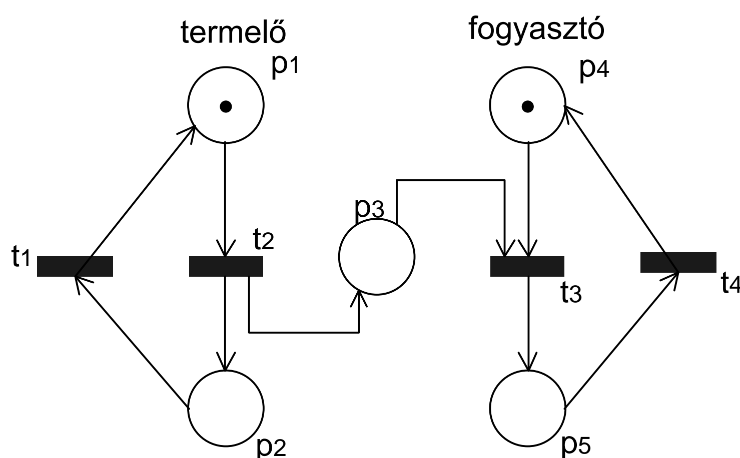
egyszeres él egy tokent tud szállítani. Formálisan a tranzíció engedélyezettségének feltétele:

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

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

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

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



3. ábra: Termelő-fogyasztó rendszer Petri-háló alapú modellje

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

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

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

2.3.5 Tüzelési szekvencia

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

2.3.6 Petri-hálók invariánsai

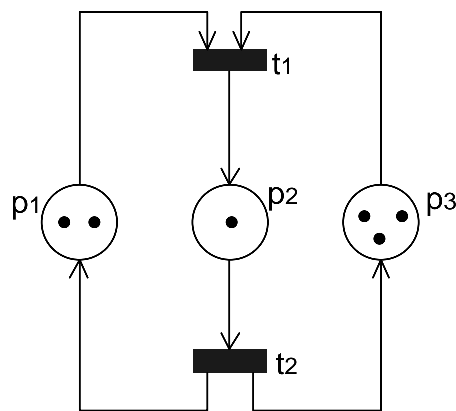
Az invariáns tulajdonságok azt fogalmazzák meg, hogy a rendszer állapottere helyes működés esetén nem alakul tetszőlegesen, ezek alapján megkülönböztetünk P- és T-invariánsokat.

2.3.6.1 T-invariánsok

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

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

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



4. ábra: Petri-háló T-invariánsainak szemléltetése

2.3.6.2 P-invariánsok

Jelen esetben azt követeljük meg, hogy a tokenek számának a helyek valamely részhalmaza felett egy súlyvektorral képzett súlyozott összege állandó maradjon, azaz a $\vec{\sigma}$ tüzelési szekvenciára σ_P^T M = állandó. A $\vec{\sigma}$ tüzelési szekvencia a $M_0[\vec{\sigma} > M$ állapotátmenetet hajtja végre, így kapjuk az alábbi egyenletet:

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

A súlyozott tokenösszeget kiszámítva:

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

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

$$(1)\ W\sigma_P = 0.$$

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

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

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

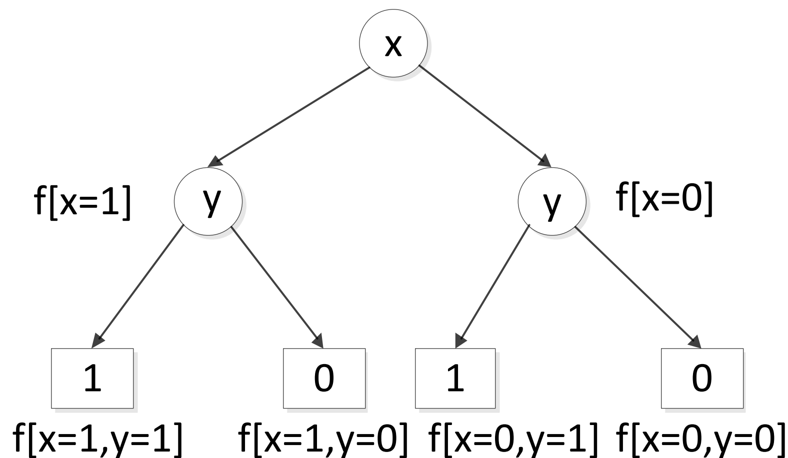
2.4 Döntési diagramok és alkalmazásaik a modellellenőrzésben

A nagyméretű állapothalmazok tárolási és kezelési problémájának megoldására nyújtanak alternatívát a szimbolikus technikák [4][5][6]. Itt az állapothalmazok explicit tárolása helyett azok karakterisztikus függvényét (bináris függvény) tároljuk [14][15] és a halmazműveleteket is a karakterisztikus műveletek segítségével értelmezzük. Egy állapothalmaz karakterisztikus függvénye az az állapotváltozókon értelmezett logikai függvény, amely akkor és csakis akkor igaz, ha az állapot az adott halmazba tartozik. Az állapotváltozók számát ezek alapján a kódolandó állapotok száma határozza meg. A karakterisztikus függvények használatának matematikai alapját a Stone-tétel (Boole-algebrák reprezentációs tétele) adja: minden véges Boole-algebra izomorf egy véges *S* halmaz részhalmazainak algebrájával.

2.4.1 Bináris döntési diagramok

Szimbolikus modellellenőrzés során a karakterisztikus függvénnyel kódolt állapottér tárolására a redukált, sorrendezett bináris döntési diagramok (reduced ordered binary decision diagram, ROBDD) hatékonyan alkalmazhatók [6][7]. A döntési diagramok bevezetéséhez szükség van az *if-then-else* operátor definiálására; $x \to x_1, x_0 = (x \wedge x_1) \vee (\neg x \wedge x_0)$. A kifejezés az x_1 értékét veszi fel, ha x igaz, x_0 értékét veszi fel, ha x hamis. Amikor a karakterisztikus függvényt ki akarjuk értékelni az előbbi operátor segítségével akkor az iteratív alkalmazás során, a jobb oldalon egyre kevesebb változó marad, mindezt úgy képzelhetjük el, hogy egy döntési fa szintjein megyünk lefelé. A folyamatot szemlélteti az 5. ábra: a példa az $f = (x \wedge y) \vee (\neg x \wedge y)$

karakterisztikus függvényhez tartozó döntési fát mutatja. Először az *x* változó értékét kell behelyettesíteni, ezzel a fában egy szinttel lejjebb jutunk, ahol az *y* változó értékét kell behelyettesíteni. A levél csomópontokban a 0 vagy 1 logikai értékek szerepelnek, látható, hogy igaz értéket akkor kapunk, ha $x = 1$ és $y = 1$ vagy $x = 0$ és $y = 1$.



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

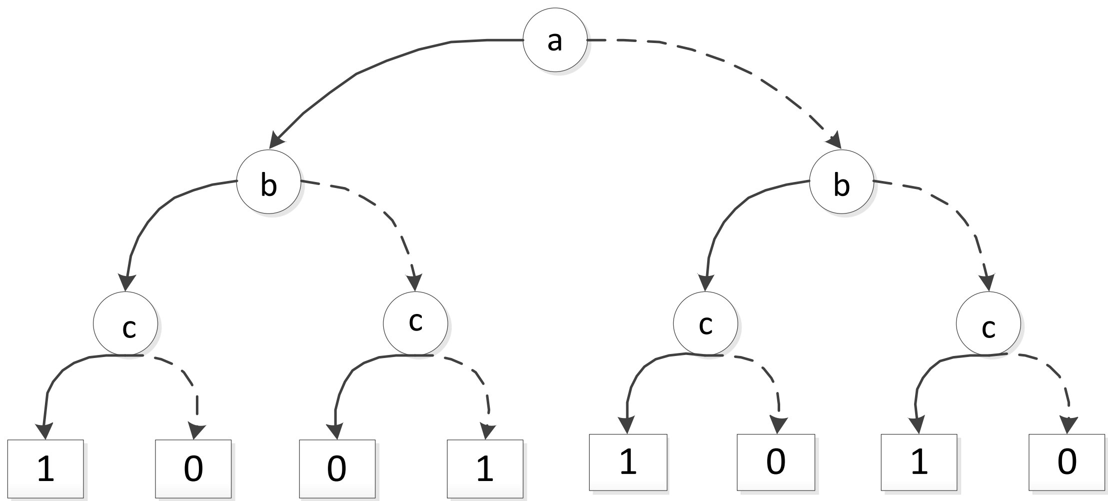
A fa levél csomópontjaiban már az igaz vagy hamis logikai értékek maradnak csak. A feladatunk tehát a döntési fán a gyökértől kiindulva a karakterisztikus függvénynek megfelelő változó értékét behelyettesíteni így egy szinttel lejjebb jutunk. A bináris döntési fát többféleképpen egyszerűsíthetjük:

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

Az ROBDD az alábbi tulajdonságokkal rendelkezik:

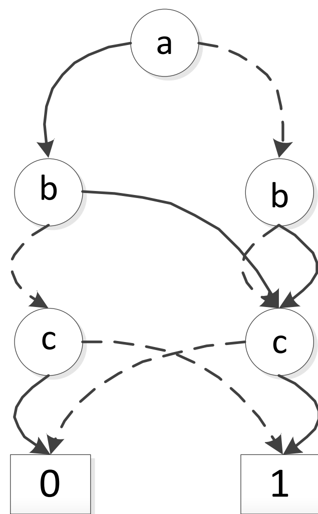
- Irányított, aciklikus gráf egy gyökér és két levél csomóponttal. A levelekben tehát csak az igaz és hamis logikai konstansok vannak, a csomópontokat egy teszt változóval címkézzük.
- Minden csomópontból két él indul ki, a teszt változóba való 0 és 1 logikai értékek behelyettesítését reprezentálják.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

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



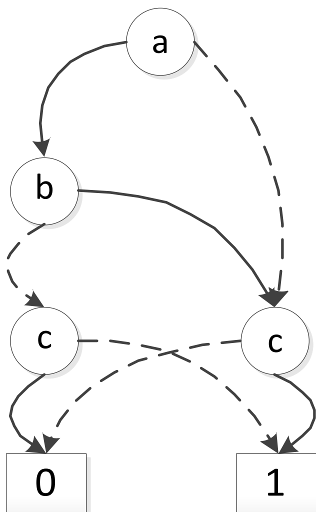
6. ábra: Karakterisztikus függvény döntési fa alapú reprezentációja

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



7. ábra: Karakterisztikus függvény BDD alapú reprezentációja

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



8. ábra: Karakterisztikus függvény ROBDD alapú reprezentációja

Fontos megjegyezni, hogy a döntési diagram mérete szempontjából lényeges, hogy mi a teszt változók sorrendje. Az optimális sorrendezés NP teljes probléma, de sok esetben heurisztikus módszerekkel jó eredményt lehet találni [16]. A *PetriDotNet* modellellenőrző program a döntési fa teszt változóinak és ezzel együtt a szintek kialakításához többféle módszert biztosít. Lehetőség van

- a Petri-háló P-invariánsai
- manuális szintezés
- összetett, a P- és T-invariánsokat is figyelembe vevő módszer [3]

alapján kialakítani a szintezést.

2.4.2 Többértékű döntési diagramok

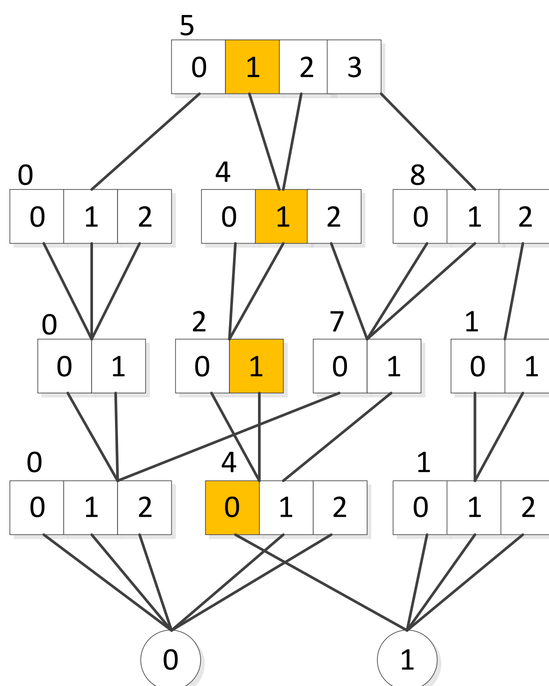
Aszinkron rendszerek esetén lehetőség nyílik arra, hogy a modellt kisebb, egymással kölcsönhatásban levő részmodellekre partícionáljuk. Tegyük fel, hogy a rendszert K darab részmodellre lehet felbontani. Ekkor egy globális i állapotot egy K elemű vektorral tudunk leírni $(i_1, i_2, \ldots, i_K)$ formában. Ezzel a jelöléssel i_n egy lokális állapot a *n-ik* részmodellben $(K \geq n \geq 1)$. Ezek alapján a rendszer lehetséges állapotainak halmazát $(\hat{S})$ a K darab részmodell lehetséges állapotai halmazának Descartes szorzata adja meg. Petri-hálók esetén ezt úgy lehet megvalósítani, hogy a rendszer egy globális állapotát jelentő tokeneloszlás az egyes részmodellek lokális állapotát reprezentáló token-eloszlásokból képzett vektor adja meg.

Ezen tulajdonság kihasználásával egy sokkal kompaktabb, tárigény szempontjából kedvezőbb adatstruktúrát lehet kialakítani; a többértékű döntési diagramokat. A bináris döntési diagramokkal ellentétben a *többértékű döntési diagram* [17][18] nem bináris típusú változókat tartalmaz, egy változó egy véges halmaz elemei közül tetszőleges értéket felvehet. A szakdolgozat alapját jelentő szaturációs algoritmus az úgynevezett *kvázi-redukált többértékű döntési diagramokat* [4] (quasi-reduced multi-way decision diagram, quasi-reduced MDD) használja. A kvázi-redukált jelző arra utal, hogy a duplikált csomópontok itt sem megengedettek a döntési diagramban, de az olyan csomópontok igen, melyek minden éle azonos csomópontba mutat, ugyanis az algoritmus során ez is lényegi információt fog hordozni, így ezeket nem redukáljuk.

A kvázi-redukált MDD főbb tulajdonságai:

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

A 9. ábra mutat példát a kvázi-redukált MDD-re. Az itt látható MDD 4+1 szintből áll. A korábbi jelöléseket alkalmazva írhatjuk például, hogy $\langle 4|5\rangle\,[0] = \langle 3|0\rangle$. Az egyes csomópontokon belül a lokális állapotok találhatók a négyzetekben. A rendszer elérhető állapotai a gyökér csomópontból kiindulva az 1-es terminális csomópontig vezető utakon olvashatók le a lokális állapotok mentén kódolva, így például előfordulhat az 1-1-1-0 állapot a rendszer működése során (sárga színnel).



9. ábra: Kvázi-redukált MDD

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

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

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

2.5 Állapottér-bejárás

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

- $\hat{S}$ a rendszer lehetséges állapotainak véges halmaza
- s ϵ $\hat{S}$ a rendszer kezdeti állapotát jelenti
- $\mathcal{N}$: $\hat{S} \rightarrow 2^{\hat{S}}$ a következő állapot (next state) függvény, amely azt hivatott megmutatni, hogy egy adott állapotból melyik állapot(ok) érhetők el egy lépésben.

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

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

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

2.5.1 Állapottér-felderítés explicit módszerekkel

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

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

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

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

2.5.2 Állapottér-felderítés szimbolikus módszerekkel

A szimbolikus technikák az imént bemutatásra került algoritmussal ellentétben állapothalmazokon manipulálnak és az elért állapotok tárolásához valamilyen kompakt, tárigény szempontjából hatékonyabb adatstruktúrát használnak. Ezekben az algoritmusokban is közös az, hogy a szélességi bejáráson alapulnak. Az elérhető állapotok előállítása a következő-állapot függvény (*Next State Function,* $\mathcal{N}(\mathcal{S})$) iteratív alkalmazásával lehetséges.

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

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

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

Az algoritmus egy továbbfejlesztett változata a láncolásos (chaining) algoritmus [4], melyben a következő állapot függvényt külön specifikusan egy eseményhez kapcsolódóan hajtjuk végre és a felderített állapotokat még a ciklusmagon belül hozzáadjuk a korábban felderítettekhez. Ez után folytatjuk más esemény eltüzelését, majd ha mind el lett tüzelve, akkor fog megtörténni a ciklusmag újbóli meghívása. Ennek előnye, hogy ha az események tüzelésének sorrendje olyan, hogy az egy egész útvonalat határoz meg az állapotok között (tegyük fel hogy *k* hosszúságú), akkor az algoritmus a ciklusmag egyszeri lefuttatásával megtalálja ezt, szemben a korábban közölt két algoritmussal, melyben *k* iterációra lenne ehhez szükség. Ez az algoritmus tehát úgy képzelhető el, hogy a szélességi bejárás közben ezeket az útvonalakat a mélységi bejáráshoz hasonlóan végig tudja követni.

3 A szaturációs algoritmus

A szaturációs algoritmus [4][11][12] egy szimbolikus technika a rendszer állapotterének felderítéséhez. Az algoritmus a rendszer Petri-háló alapú reprezentációjából kiindulva állítja elő az elérhető állapotokat reprezentáló kvázi redukált döntési diagramot. A szaturáció folyamatát szemlélteti a 10. ábra.

10. ábra: A szaturáció folyamata

Az algoritmus a következő lépésekből áll:

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

A modell dekompozíciója során a bemeneti Petri-hálót a 3.1. fejezetben leírtaknak megfelelően részekre bontjuk, amely alapján majd a szimbolikus kódolást menet közben elvégezzük. Ennél a lépésnél felhasználjuk az esemény lokalitás nevű tulajdonságot is, amely a 3.2. fejezetben kerül bemutatásra.

A dekompozíciót követően, a szaturáció során felhasznált helyben frissítés előnye, hogy jelentősen csökkenti az állapotok felderítése során létrehozott csomópontok számát, ezért lehet hatékony az algoritmus memória felhasználás szempontjából. A helyben frissítés módszere a 3.3. fejezetben kerül bemutatásra.

A szaturációs algoritmushoz (3.5. fejezet) a következő-állapot függvény Kronecker mátrixos reprezentációjára van szükség, amely egy kompakt ábrázolását jelenti a függvénynek (3.1. fejezet).

A 3.4 fejezetben megismerhető a szaturáció alapját képező rekurzív mélységi bejárás algoritmusa. Ennek jelentős szerepe van mind az állapottér felderítése, mind a döntési diagram építése során. Alkalmazásával jelentős sebességnövekedés és tárhatékonyság érhető el a korábban említett explicit és szimbolikus technikákhoz képest.

3.1 Az állapottér dekompozíciója

A szaturációs algoritmus akkor tud hatékonyan működni, ha a bemeneti Petri-háló megfelelően részmodellekre van felbontva. Aszinkron rendszerek esetén erre lehetőség is nyílik, azaz a független komponensek megfeleltethetők egy-egy részmodellnek. Ezen részmodellek lokális állapotai lesznek elkódolva ez után a többértékű döntési diagram egy-egy szintjén. A rendszernek egy globális állapota a döntési diagram gyökér csomópontjából a terminális 1 csomópontba vezető utak mentén olvasható le ez után.

A modell dekompozíciója miatt arra is lehetőség nyílik, hogy $\mathcal{N}$-et (következő állapot függvény) felbontsuk diszjunkt következő állapot függvények uniójára. Formálisan ez azt jelenti, hogy $\mathcal{N}(i) = \bigcup_{s \in \varepsilon} \mathcal{N}_\alpha(i)$. Itt $\mathcal{N}_\alpha$ a következő állapot függvény, amely az α eseményen alapul, ε az események véges számú halmaza. $\mathcal{N}_\alpha(i)$ azon állapotok véges halmazát jelöli, melyek az $\boldsymbol{i}$ állapotból kiindulva, az α esemény tüzelésének hatására elérhetőek. Azt mondjuk, hogy egy α esemény nincs hatással az $\boldsymbol{i}$ állapotra, ha $\mathcal{N}_\alpha(i) = \emptyset$.

A következő állapot függvény Kronecker mátrix alapú kódolása

A következő-állapot függvény hatékonyan és szemléletesen kódolható a Kronecker mátrixok [28] alkalmazásával, így a következő állapot függvény egy kétdimenziós mátrixként való reprezentációját kapjuk. A mátrixos forma leggyakoribb felírásában a sorok a részmodelleknek felelnek meg, az oszlopok pedig az eseményeknek. Erre akkor van lehetőség, ha a modell dekompozíciója *Kronecker konzisztens* [11], ez alatt azt értjük, hogy létezik $K \cdot |\varepsilon|$ darab függvény – ahol az egyes függvények $\mathcal{N}_{k,\alpha} : \mathcal{S}_k \to 2^{\mathcal{S}_k}$ alakúak – melyek egy adott esemény hatását írják le egy adott részmodellen (itt K a részmodellek száma, $|\varepsilon|$ pedig az események halmazának számossága). Más szavakkal annak kell teljesülnie, hogy egy globális állapoton való esemény tüzelés hatása meg kell, hogy egyezzen az egyes részmodelleken való tüzelések eredményének Descartes szorzatával. A lokális következő állapot függvény egy 0 és 1 elemeket tartalmazó mátrixként írható fel, $\mathcal{N}_{k,\alpha} \in \{0,1\}^{n_k \times n_k}$ (itt n_k a k-ik részmodell állapotainak száma). $\mathcal{N}_{k,\alpha}[i_k, j_k] = 1 \Leftrightarrow j_k \in \mathcal{N}_{k,\alpha}(i_k)$, azaz a mátrixelem csak akkor nem nulla, ha az i_k állapotból a j_k állapot elérhető valamilyen tüzelés végrehajtásával.

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

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

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

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

Az előálló mátrix rendkívül ritka, Petri-hálók esetén egy sor legfeljebb egy nemnulla elemet tartalmaz. Fontos megjegyezni, hogy színezetlen Petri-hálókra a Kronecker konzisztencia feltétel mindig teljesül.

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

3.2 Az események lokalitása

Fontos tulajdonsága a Petri-hálóknak az, hogy nem minden tranzíció befolyásol minden szintet, azaz a tüzelésével nem változik meg az adott szinten az elérhető állapotok

halmaza. E miatt az állapottér felderítése során felesleges minden szinten minden eseményt eltüzelni, ilyen módon jelentősen növelhető az algoritmus teljesítménye. Jelölje $Top(\alpha)$ azt a legmagasabb szintet, $Bot(\alpha)$ pedig azt a legalacsonyabbat, amelyet az α még befolyásol. Formálisan ez a $Top(\alpha) = max\{k : \mathcal{N}_{k,\alpha} \neq \boldsymbol{I}\}$ illetve $Bot(\alpha) = min\{k : \mathcal{N}_{k,\alpha} \neq \boldsymbol{I}\}$ formában adható meg. Az események ε halmazát felbonthatjuk K osztályra, $\varepsilon_k = \{\alpha \in \varepsilon, Top(\alpha) = k\}$, minden $K \geq k \geq 1$ -re. Lényeges tehát észrevenni, hogy egy esemény tüzelésekor nem szükséges a gyökér csomóponttól elindulni és onnan az alsóbb szintek felé végig eltüzelni azt. Egy α esemény, melyre $Top(\alpha) =$ k, független a K és k+1 közötti szintektől, így ilyenkor a tüzelést csak a k-ik és az alatti szintekre kell végrehajtani, ez hasonlóan igaz a $Bot(\alpha)$ alatti szintekre is.

A másik fontos észrevétel, hogy az előbb leírt jelenséget ki nem használó algoritmusok egy α esemény tüzelését a gyökér csomópontnál elkezdik és egy $\langle k|p\rangle$ csomópontra annyiszor fogják végrehajtani a tüzelést, ahány él vezet ebbe a csomópontba. Ezzel szemben, ha kihasználjuk azt, hogy egyből a k-ik szintre ugorhatunk a tüzelés végrehajtásával, akkor a $\langle k|p\rangle$ csomópontra csak egyszer fog ez megtörténni. Annyiban jelent ez nehézséget, hogy ha egy $\langle k|p\rangle$ csomópontot helyettesíteni akarunk egy $\langle k|q\rangle$ csomóponttal, akkor ahhoz az kell, hogy az összes $\langle k|p\rangle$-be menő élet át kell irányítani a $\langle k|q\rangle$ csomópontba. Ezen problémára ad megoldást a következő fejezetben ismertetett helyben frissítés módszere.

3.3 A helyben frissítés módszere (in-place update)

A helyben frissítés alapgondolata az, hogy egy $\langle k|p\rangle$ csomóponton a lehetséges eseményeket kimerítően eltüzeljük és a tüzelések eredményének megfelelően az elérhető állapotok halmazát inkrementálisan bővítjük. A módszer lényege az, hogy amikor egy olyan α eseményt tüzelünk el a $\langle k|p\rangle$ csomóponton, amire $Top(\alpha) \leq k$, akkor az új, elérhető állapotokat a $\langle k|p\rangle$ éllistájának folyamatos bővítésével kódoljuk el. Ennek az a jelentősége, hogy így hatékonyabb a döntési diagram építése, szemben azzal a módszerrel, hogy minden tüzeléssel létrehoznánk egy új csomópontot és a tüzelés eredményét az eredeti és az újonnan létrehozott csomópont uniójaként állítanánk elő. Ennek következménye, hogy így jelentősen csökken a memória felhasználása az algoritmusnak

3.4 Rekurzív mélységi állapottér-felderítés

Jellemzően a modellellenőrző algoritmusok szélességi, vagy kombinált szélességi-mélységi (pl. chaining [4]) állapottér bejárást alkalmaznak. Ilyenkor az állapottér egy részéből megvizsgálják az elérhető következő állapotokat, és azokkal bővítik az állapotok aktuális halmazát. A szaturáció újdonsága ezzel szemben abban rejlik, hogy a rekurzív bejárást nem egyenként az állapotokra, hanem csomópontokra alkalmazza, majd az esetleges változásokat rekurzívan lefelé érvényesítve számítja ki az elérhető állapothalmazt. Eközben a tüzelést mohó módon, kimerítően alkalmazza, amíg egy lokális fixpontot el nem érünk. Lokális fix pontról akkor beszélünk, ha az állapotok részhalmaza lokális tüzelésekkel (azaz olyan tüzelés, amely az adott állapothalmazban tüzelhető) tovább már nem bővíthető.

A rekurzív mélységi állapottér-felderítést két függvény végzi, a *SatFire* [A1] és a *SatRecFire* [A2]:

- A *SatFire* egy $\langle k|p\rangle$ csomóponton tüzel el egy α eseményt, ahol k = Top(α). Ehhez a helyben frissítés módszerét használja fel. A $\langle k|p\rangle$ csomópontból elérhető összes állapoton (ezek szintje k-1) végrehajtja az α esemény tüzelését, ehhez a *SatRecFire* függvényt használja fel.
- A *SatRecFire* függvény a paraméterül kapott α esemény tüzelését végzi el mindazokon az $\langle l|q\rangle$ csomópontokon, amelyekre teljesül, hogy $Top(\alpha) \geq l \geq Bot(\alpha)$, de nem módosítja ezeket a csomópontokat. E helyett létrehoz egy új csomópontot és azon tüzeli el az eseményt, amely így a tüzelés hatását reprezentálja $\langle l|q\rangle$-n és gyerekein. A metódusban használt *Cached* és *PutInCache* metódusok teljesen hasonlóak az unió műveletben használtakhoz, azzal a különbséggel, hogy ezek a tüzelési gyorsítótáron (*Fire Cache*, továbbiakban FC) – amely egy hash-tábla – végeznek műveleteket, nem pedig az uniók eredményét eltároló gyorsítótáron (*Union Cache*). A hash-tábla kulcsa ebben az esetben az unióban használt két csomópont helyett (melyeken az unió műveletet végeztük el) egy csomópont és egy esemény, az érték pedig a tüzelés eredményét reprezentáló csomópont.

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

3.5 Szaturáció

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

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

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

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

A szaturáció tehát egy rekurzív algoritmus, amely az MDD gyökér csomópontjából kiindulva építi fel a döntési diagramot. Az események tüzelését rekurzívan hajtja végre a szinteken lefelé haladva. Ha menet közben a *SatFire* által meghívott *SatRecFire* metódus létrehoz egy új csomópontot, akkor az azonnal szaturálva lesz. Ennek

következménye, hogy amikor egy felsőbb szinten levő csomóponton hajtjuk végre a szaturációt, akkor biztos az, hogy az alsóbb szinten levő gyerek-csomópontok - melyek elérhetőek a csomópontból - már szaturáltak. A végső MDD-be csak szaturált csomópontok fognak bekerülni. A szaturációs algoritmus erőssége pont abban rejlik, hogy a többi szimbolikus technikához képest a futás közben létrehozott csomópontok száma nagyságrendekkel kevesebb, azonban ez függ az alkalmazott szintezéstől is.

4 A párhuzamos szaturációs algoritmus

A modellellenőrzés során jelentkező időigény problémájára megoldást jelenthetnek a párhuzamos algoritmusok. A párhuzamosítás iránti igény jogos, hiszen egyre több helyen jelennek meg a több processzorral rendelkező hardver architektúrák, és ha ezeket a többlet erőforrásokat hatékonyan ki tudjuk használni, akkor futási időbeli csökkenés érhető el az állapottér-felderítés során. Ez azt is jelenti, hogy a korábbiakhoz képest sokkal nagyobb állapottérrel rendelkező rendszerek vizsgálata válik lehetővé.

A TDK dolgozatban a korábban bemutatott szaturációs algoritmus párhuzamosítási lehetőségeit [5] vizsgáltuk meg. Jelen fejezet bemutatja az algoritmikai és implementációs megfontolásokat, illetve részletezi a hozzá kapcsolódó fejlesztéseinket, javításainkat.

4.1 Az algoritmus bemutatása

A szaturáció egy szimbolikus technika a rendszer állapotterének felderítéséhez, melynek részletes leírása megtalálható a 0. fejezetben. A [5]-ben található meg a szaturációs algoritmus osztott memóriás környezetben való párhuzamosításának részletei. Ebben részletesen leírják, hogy a korábbi mélységi -vagy szélességi bejáráson alapuló technikák párhuzamosítása esetén az jelenti a fő nehézséget, hogy az állapottér-felderítés során az egyes szálaknak sűrűn kell szinkronizálniuk egymással, az egyik szál által elérhető állapotok halmaza függ a többi szál által elvégzett műveletektől.

A szaturációs algoritmus is a szekvenciális jellegű feladatok közé tartozik, a párhuzamosításra azonban megoldást jelenthet, ha kihasználjuk az aszinkron rendszerek esetén megjelenő esemény lokalitás jelenségét. A lokális eseményeket kihasználva lehetővé válik, hogy olyan feladatokat hozzunk létre, amelyeket az egyes szálak párhuzamosan tudnak végrehajtani. Az így kialakuló feladatok azonban még mindig nem teljesen függetlenek, ezért a [5]-ben többféle kiegészítést adnak a szekvenciális szaturációs algoritmushoz:

- Felfelé mutató élek bevezetése: a párhuzamosan dolgozó szálak szinkronizációjának fő eszköze
- *Ops* (operations – műveletek) érték: minden egyes csomópont nyilván tartja egy változóban, hogy hány szál dolgozik éppen rajta párhuzamosan, ezáltal lehet megállapítani, hogy mikor válik szaturálttá.
- Az adatszerkezetek oly módon történő kiegészítése, hogy azok többszálú környezetben is garantálni tudják a konzisztenciát.

A felfelé mutató élek szerepe

A döntési diagram párhuzamos építése során előfordulhat az, hogy olyan lefelé mutató élet szeretne behúzni egy szál az általa épített részfából egy másik szál által építettbe, melynek felderítése még nem fejeződött be. Egy lefelé mutató élet csak akkor lehet behúzni, ha a cél csomópont már szaturált állapotú. A vázolt esetben ez nyilvánvalóan nem teljesül, így ilyenkor annak a szálnak várakoznia kellene, amelyik az élet szeretné behúzni. Ebben az esetben azonban az egyes szálak az algoritmus futása során sokat várakozhatnak, amely a teljesítmény leromlását eredményezné. Ezt a problémát orvosolják a felfelé mutató élek. Amikor egy szál olyan csomóponthoz akar lefelé mutató élet behúzni, amely még nem szaturált, akkor e helyett egy felfelé mutató élet

húz be a forrás csomópontba, majd folytatja a feladat végrehajtását. Innentől kezdve, ha a felfelé mutató él kezdőpontja szaturálttá válik, akkor az aktuálisan rajta dolgozó szál feladata lesz a felfelé mutató él lecserélése lefelé mutatóra és a további műveletek elvégzése.

4.2 Implementációs részletek

A szekvenciális algoritmushoz képest több változtatást kellett végrehajtani:

- Várakozási sor bevezetése a feladatok végrehajtására
- A döntési diagram egy csomópontját reprezentáló *MDDNode* osztályt több attribútummal is ki kellett egészíteni.
- Tüzelési gyorsítótár többszörös hozzáférésének biztosítása
- Az MDD tárolóhoz való párhuzamos hozzáférés megvalósítása

4.2.1 Várakozási sor bevezetése a feladatok végrehajtásához

A párhuzamos szaturáció során az egyes független műveletek végrehajtását külön szál végzi, azonban rendkívül rossz hatékonyságot eredményezne, ha minden egyes esetben új szálat indítanánk. A [5]-ben közölt algoritmus egy várakozási sort vezet be a végrehajtandó feladatok tárolásához és egy szálkészletet rendel ehhez a sorhoz. Egy szabaddá vált szál ebből a sorból vesz ki feladatot végrehajtásra. A saját implementációban a .NET keretrendszer által nyújtott *ThreadPool* osztályt használtuk fel a szálak menedzselésére.

4.2.2 Csomópont adatszerkezeteinek kiegészítése

A csomópontot reprezentáló adatszerkezetet több attribútummal kellett kiegészíteni:

- *Upward edges*: A felfelé mutató élek nyilvántartására szolgáló lista. A felfelé mutató él kezdőpontja az adott csomópont.
- *Ops*: A csomópontot párhuzamosan manipuláló szálak száma. Az *ops* változó értékének növelésére akkor van szükség, ha a csomópontot valamely szál elkezdi szaturálni vagy eseményt tüzel el rajta. A változó értékét akkor csökkentjük, ha az adott műveletek befejeződnek.
- *Saturating*: Annak jelzésére szolgál, hogy az adott csomópontot egy szál elkezdte-e már szaturálni. A változó alapértéke hamis, a szaturáció megkezdésekor átállítjuk igaz értékűre.
- *Key*: A tüzelési gyorsítótár – mely egy hash-tábla - címzésére szolgáló változó.

4.2.3 A tüzelési gyorsítótár kiegészítése

A tüzelési gyorsítótár alkalmazásával elkerülhető, hogy egy adott csomóponton egy adott eseményt többször eltüzeljünk. A hash-tábla kulcsa a csomópont-tranzíció pár, értéke pedig a tüzelés eredményét reprezentáló csomópont és egy bool érték, mely azt jelenti, hogy a csomópont szaturált-e már. A szekvenciális algoritmushoz képest bevezetett változtatások:

- A gyorsítótárhoz több szál fordulhat párhuzamosan, ezért meg kell oldani a konzisztencia kezelését. A tüzelési gyorsítótár frissítése, egy elem beszúrása vagy törlése nem atomi műveletek, ezért zárakat kell alkalmazni.

- Nem csak a tüzelés eredményét kell eltárolni, hanem azt a tényt is, hogy elkezdődött egy tüzelés. A szekvenciális algoritmusnál erre nem kellett figyelni, mivel ha egy tüzelés eredménye nem volt még benne a tárban, akkor a főszál végrehajtotta azt. Párhuzamos esetben előfordulhatna, hogy több szál is elkezdi végrehajtani ugyanazt a tüzelést, ezzel a módszerrel ez elkerülhető.

4.2.4 Az MDD tároló kiegészítése

A szaturációs algoritmus a döntési diagram csomópontjait egy MDD-tárolóba rakja. A párhuzamos algoritmusban meg kell oldani azt, hogy a tárolóhoz való hozzáférés atomi művelet legyen. A [5]-ben ehhez részgráfzárolást közölnek implementációs részletek megadása nélkül, ez azonban a saját implementációban rendkívül lassúnak bizonyult, ezért az algoritmust további vizsgálatoknak vetettük alá. Ennek eredményeként vezettük be a lokális szinkronizáció módszerét, amely a 4.3.3 fejezetben kerül bemutatásra.

4.2.5 Az algoritmus főbb függvényeinek bemutatása

A párhuzamos szaturációs algoritmushoz szükséges összes metódus pszeudokódja megtalálható az [A] függelékben, itt csak a lényegesebb függvények bemutatására szorítkozok. Mielőtt ezt megtenném, be kell vezetni a *ZeroNode* fogalmát.

Definíció: A *ZeroNode* az üres állapothalmazt reprezentáló csomópont, amelynek a szintjét jelöljük k-val:

- Ha $k = 1$, akkor a csomópont minden éle a terminális 0 csomópontba mutat
- Ha $k > 1$, akkor a csomópont minden éle a $k - 1$-ik szint *ZeroNode*-jába mutat.

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

A metódus egy $\langle k|p \rangle$ csomópont szaturációját végzi. Az elvégzett műveletek az alábbiak szerint foglalhatók össze:

- A csomópont *saturating* attribútumának értékét igazra állítja, ezzel jelezve, hogy a csomópont szaturációja elkezdődött.
- Az *ops* értéket megnöveli 1-gyel, ez mutatja azt, hogy egy újabb szál kezdte el manipulálni a csomópontot.
- Kimerítően eltüzeli az eseményeket az egyes nemnulla lokális állapotokon. Ehhez a *SatFire* függvényt hívja meg.
- Ez után az *ops* értéket csökkenti eggyel, mert a metódust végrehajtó szál befejezte a csomóponton a műveleteket.
- Ha így az *ops* érték 0-ra csökkent, akkor az azt jelenti, hogy a csomópont szaturálva van, így ilyenkor meg kell hívni a *NodeSaturated* függvényt a csomópontra.

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

A metódus egy $\langle k|p \rangle$ csomópont i lokális állapotában eltüzeli az összes engedélyezett eseményt.

- Az események tüzelését a rekurzív *SatRecFire* metódus meghívásával végzi el.
- Ha a *SatRecFire* visszatérési értéke nem *ZeroNode*, akkor zárolni kell a $\langle k|p\rangle$ csomópont alatti részgráfot.
- Az *i* állapotból az egyes események eltüzelésének hatására elérhető állapotokat jelölje *j*. A metódus az unióját képezi a $\langle k|p\rangle[j]$ csomópontnak és a *SatRecFire* visszatérési értékének, és ha ez különbözik a $\langle k|p\rangle[j]$ korábbi értékétől, akkor átállítja azt a *SatRecFire* visszatérési értékére.
- Feloldja a zárat a részgráfról.
- Ha új állapotokat értünk el a műveletet végrehajtva, akkor a *j*. állapotra meghívja a *Confirm* függvényt, illetve a *SatFire* függvényt is, hogy a *j*. állapoton is rekurzívan el legyenek tüzelve az események.

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

Az $\langle l|q\rangle$ csomóponton az e esemény kimerítő tüzelését végzi. A művelet több lépésből áll:

- Ha az *e* eseményre teljesül az, hogy $l < Bot(e)$, akkor a metódus visszatér az $\langle l|q\rangle$ csomóponttal
- Különben a tüzelési gyorsítótárban megnézi, hogy erre a csomópontra az *e* esemény el lett-e már tüzelve. A gyorsítótárat zárolni is kell, mivel több szál is hozzáférhet egyidejűleg.
- Ha benne van a gyorsítótárban a keresett elem mégpedig úgy, hogy az még nem szaturált, akkor felfelé mutató éleket kell beállítani az $\langle l+1|p\rangle$ csomópontra, ahonnan a *SatRecFire* hívás történt. Ebben az esetben *ZeroNode*-al kell visszatérni és a felfelé mutató élek garantálják azt, hogy a szál továbbmehessen, később majd egy másik szál feladata lesz a szaturáció befejezése.
- Ha gyorsítótárban talált érték már szaturált állapotú, akkor azzal kell visszatérni.
- Ha nincs találat, akkor új csomópontot kell létrehozni, ez lesz az $\langle l|s\rangle$, amely a tüzelés eredményét reprezentáló részgráf gyökere lesz. Az $\langle l+1|p\rangle$ csomópontra felfelé mutató éleket kell beállítani.
- A tüzelési gyorsítótáron a zárat fel kell oldani.
- $\langle l|s\rangle$ *ops* értékének megnövelése szükséges, ezzel jelezve, hogy a szál műveletet végez rajta.
- A tüzelési gyorsítótár zárolása után egy új elemet szúr be abba, a kulcs az $(l+1, p)$ páros, az érték pedig az $(\langle l|s\rangle, hamis)$ lesz. A hamis érték jelzi, hogy a csomópont még nem szaturált állapotú. Ez után feloldja a zárat a táron.
- Az *L* listába összegyűjtjük $\langle l|q\rangle$ mindazon lokális állapotait, amelyekben az *e* esemény engedélyezve van.
- Az alábbi műveletet addig hajtjuk végre, amíg az *L* lista üres nem lesz:
- Egy *g* lokális állapoton eltüzeli az *e* eseményt, mégpedig oly módon, hogy rekurzívan önmagát hívja meg a metódus. Az eredményt a *SatFire*-ben leírtaknak megfelelően dolgozza fel, azaz itt is unió műveletet kell végrehajtani az eredeti és az újonnan létrehozott csomópontokon.

- Az *ops* értéket csökkentjük 1-gyel, ezzel jelezve, hogy az adott szál befejezte a műveleteket a csomóponton.
- Ha így 0 lett az *ops* változó értéke, akkor megvizsgáljuk, hogy vannak-e lefelé mutató élei a csomópontnak. Ha nincsenek lefelé mutató élek, akkor az azt jelenti, hogy nem lett módosítva a döntési diagram, ezért ilyenkor *ZeroNode*-al térünk vissza. Ha vannak, akkor azon csomópontokat is szaturálni kell, ezért meghívjuk a *QSaturate* metódust ezekre a csomópontokra.
- A függvény visszatér a *ZeroNode*-al.

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

A $\langle k|p\rangle$ csomópont szaturálttá válásakor végrehajtandó műveleteket végzi el a metódus:

- Az MDD tárolóban eltároljuk a $\langle k|p\rangle$ csomópontot.
- Ha a legfelső szintű csomópont vált szaturálttá, akkor befejeződött a döntési diagram építése és így az állapottér-bejárás. Ebben az esetben terminálni kell a program futását.
- A tüzelési gyorsítótárban el kell helyezni a csomópontot, immár igaz érték lesz a hash-táblában a csomópont mellett, ami azt jelzi, hogy szaturálva van. Ehhez természetesen zárolni kell a tárat, mivel annak módosítása nem atomi művelet.
- Sorra veszi a $\langle k|p\rangle$ csomópontból a $\langle k+1|r\rangle[i]$ csomópontba felfelé mutató éleket és az alábbi műveleteket hajtja végre:
  - A $\langle k+1|r\rangle$ gyökerű részgráfot zárolja.
  - Unió műveletet hajt végre a $\langle k+1|r\rangle[i]$ és a $\langle k|p\rangle$ csomópontokon. Ha az unó eredménye nem egyezik a $\langle k+1|r\rangle[i]$-vel, akkor átállítja az élet az unió eredményére.
  - Ha a $\langle k+1|r\rangle$ csomópont szaturációja már korábban elkezdődött, akkor a $\langle k+1|r\rangle[i]$ csomópontra *SatFire*-t hívunk.
- A $\langle k+1|r\rangle$ csomópont ops értékét csökkentjük 1-gyel. Ha az így 0-ra csökken, akkor megvizsgáljuk, hogy elkezdték-e már szaturálni a csomópontot. Ha igen, akkor a csomópont szaturálttá vált és meghívjuk rá a *NodeSaturated* metódust. Ha nem, akkor a csomópontot szaturáljuk *QSaturate* híváson keresztül.

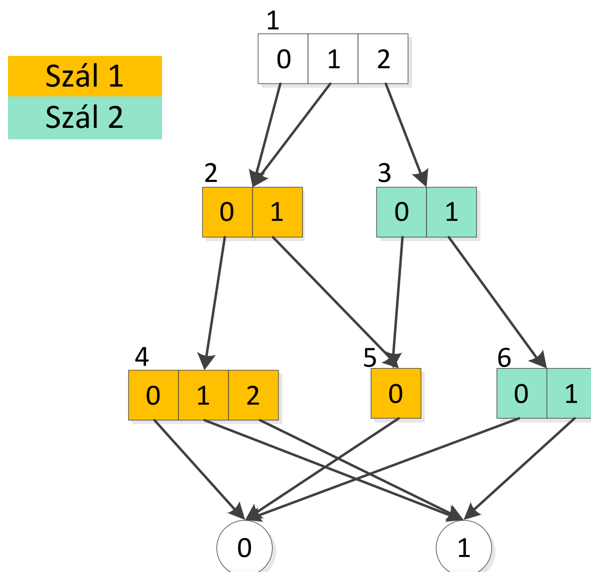
4.3 A részgráfzárolás továbbfejlesztése

A [5]-ben közölt algoritmus több helyen kiegészítésre szorult, illetve egyes részeknél nem közöltek implementációs részleteket. Ilyen volt például az MDD tároló szinkronizációja, amelyhez részgráfzárolást vezettek be a cikkben. Ez a saját implementációban rendkívül lassúnak bizonyult a zárkezelés magas adminisztrációs költségei miatt. Az algoritmust ezért további vizsgálódásnak vetettük alá és megvizsgáltuk az írási-olvasási zárak alkalmazását, majd saját fejlesztésként létrehoztuk a lokális szinkronizáció módszerét. A TDK dolgozatban ezek a fejlesztések részletesen bemutatásra kerültek, itt csak egy átfogó képet szeretnék nyújtani a zárolási módszerekről.

4.3.1 A részgráfok zárolása

Az implementáció alapjául szolgáló cikk a döntési diagram párhuzamos manipulációjához a részgráfzárolást alkalmazza, mint szinkronizációs eszköz. Itt arról van szó, hogy ha egy szál egy csomóponton műveletet akar végezni, akkor a csomópont (mint gyökér) egész részgráfját zárolni kell ehhez. Implementációra vonatkozó részleteket nem közöltek a cikkben, ezért saját megoldást dolgoztunk ki.

A megoldás bonyolultsága miatt tovább részleteket nem közlünk, csak egy példán keresztül mutatjuk az általunk kidolgozott megoldás működését (11. ábra). A példában az 1-es szál zárolja a 2-es csomópont és az az alatti részgráfot, mert ott éppen állapottér-felderítést hajt végre. Ha beérkezik egy 2-ik szál, amely a 3-as csomópontot szeretné zárolni, illetve az alatta lévő részgráfot, akkor ezt nem tudja megtenni, mert az 1-es szál által zárolt csomópontok között van olyan, amelyet ő is zárolni szeretne. A 2-es szálnak ezért várakoznia kell.



11. ábra: Az MDD zárolási módszerek szemléltetése

A problémát a részgráf zárolás adminisztrációja okozza. Ugyanis az egyes csomópontokat egyenként kell zárolni és a fő problémát az jelenti, hogy ha egy szál eljut egy darabig a zárolásban, de utána elakad, mert olyan csomópontot szeretne zárolni, amelyen már másik szál zárat tart fenn, akkor az addig általa zárolt csomópontokat sorra fel kell szabadítani. Ez hatalmas adminisztrációs terhet jelent, ezért a saját implementációnk rendkívül lassúnak bizonyult.

4.3.2 Az írási-olvasási zárak alkalmazása

A szaturációs algoritmusban a kritikus szakasz az unió művelete utáni élállítás. Ebben az esetben mindenképp garantálni kell azt, hogy csak egy szál férhessen hozzá a csomóponthoz. Látható tehát, hogy nagyobb hatékonyságot érhetünk el, ha ténylegesen csak az élállítás műveletét megelőzően garantáljuk azt, hogy csak egy szál férjen hozzá a csomóponthoz, majd a művelet elvégzése után el is vesszük ezt a jogot a száltól.

A módszer jól megvalósítható az írási-olvasási zárak alkalmazásával. Ebben az esetben egy zárat kétféle módban lehet elkérni, az egyes módok egymással való kompatibilitását szemlélteti a 12. ábra.

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

12. ábra: Az írási-olvasási zárak zár-kompatibilitási mátrixa

A hatékonyság növekedést az eredményezi, hogy egyidejűleg két vagy több szál is tarthat fenn olvasási zárat.

A zárak alkalmazásának szemantikája a következő volt:

- Ahol korábban részgráfzárolás volt, alkalmazzunk olvasási zárakat.
- Az élállítás művelete előtt a zárat fokozzuk fel írási zárrá, majd a művelet után minősítsük le olvasási zárrá.

Az implementációban a .NET keretrendszer által nyújtott *ReaderWriterLockSlim* [29] megoldást alkalmaztuk. Fontos azonban megjegyezni, hogy egyetlen globális zárat hoztunk létre, nem pedig ténylegesen a csomópontokat zároltuk. Ennek az előnye az, hogy így a holtpont elkerülés egyszerűbb.

4.3.3 Lokális szinkronizáció

Az írási-olvasási zárak alkalmazásának hátránya az volt, hogy hatóköre az egész döntési diagramra kiterjedt. Gyorsabb volt ugyan, mint a részgráfzárolás, de további vizsgálódásnak vetettük alá az algoritmust és arra jöttünk rá, hogy a zárkezelést még tovább lehet finomítani.

Az általunk javasolt megoldásban már nem a részgráfok alkotják a zárolandó objektumokat, hanem az egyes csomópontok. A vizsgálataink arra mutattak rá, hogy elegendő csupán azt a csomópontot zárolnia a szálaknak, amelyeken éppen élállítást fognak végrehajtani. A 11. ábrán bemutatott esetben a lokális szinkronizáció alkalmazásakor az 1. szálnak elegendő csak a 2. csomópontot zárolni, a 2. szálnak (amely korábban a 3-as csomópontot és az az alatti részgráfot akarta zárolni) pedig csak a 3-as csomópontot. Jól látható, hogy a két szál párhuzamosan tud futni egymás mellett, nem szükséges egyiknek sem várakozni.

Ezen megoldás helyességének ellenőrzéséhez sorra vesszük, hogy milyen követelményeket teljesít:

- Nem fordulhat elő konkurens csomópont-manipuláció: a csomópontot zárolva annak éllistája sem módosítható, ezért ezt a követelményt teljesíti a megoldás.
- Konzisztencia garantálása: a konzisztens állapotot garantálja az, hogy a műveletvégzés idejére a csomópont zárolva van. Fontos még azonban megvizsgálni az unió műveletét, mert az is módosíthatja a csomópontot. Az általunk javasolt szinkronizációs mechanizmus biztosítja, hogy az unió műveletét csak a csomópont-tárolóban elhelyezett csomóponton végezzük, ezáltal garantálva a konzisztenciát.

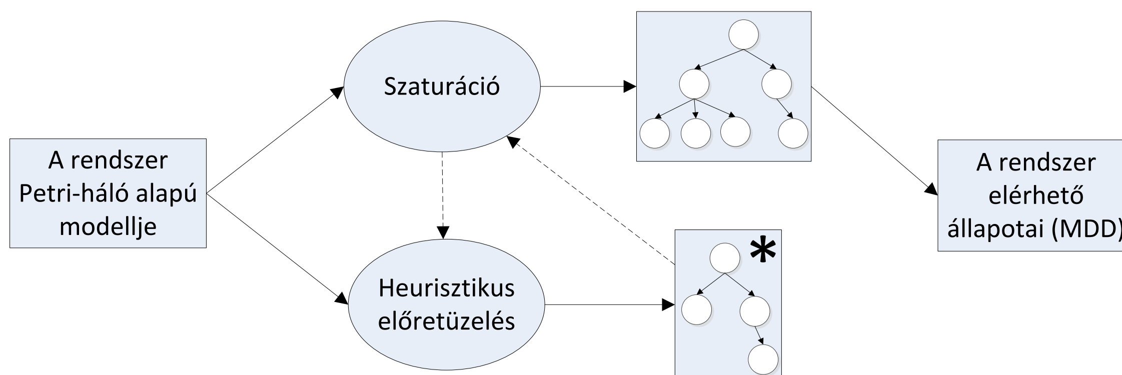
Ezek alapján elmondható, hogy a lokális szinkronizáció módszere helyes működést eredményez és hatékonyabb, mint a korábban ismertetett részgráfzárolás vagy az írási olvasási zárak alkalmazása.

5 Heurisztikán alapuló tranzíció-előretüzelés

A modellellenőrzés során a futási idő csökkentésében a párhuzamos megoldások mellett nagy szerepet kapnak a különböző heurisztikán alapuló megoldások is. A heurisztikák hatékonyan alkalmazhatók a követelmények ellenőrzésében, különösen, ha nem kimerítő ellenőrzést akarunk végezni a rendszeren, hanem csak egy példát akarunk keresni valamilyen követelmény nem teljesülésére.

A TDK dolgozat továbbgondolásával, saját fejlesztésként hoztam létre a heurisztikán alapuló tranzíció-előretüzelés módszerét, amely a heurisztikáknak egy másfajta felhasználási területét jelenti. Az ötletet az adta, hogy a párhuzamos algoritmus az esetek többségében nem használta ki teljesen a rendelkezésre álló számítási erőforrásokat. Ha lehetőség lenne ezen maradék erőforrások hatékony kihasználására is, akkor tovább csökkenthetnénk a futási időt. A tranzíció-előretüzelés módszere a maradék erőforrásokat arra használja fel, hogy a döntési diagramot heurisztikus módon előreépítse, mégpedig oly módon, hogy az előre kiszámított eredményeket a szaturációs algoritmus a futása során felhasználja később. Jól látható, hogy egy alkalmas heurisztikával lehetőség nyílik arra, hogy előre kiszámítsunk bizonyos részgráfokat, amelyeket ez után már egyszerűen fel tudunk használni. Hangsúlyos azonban az alkalmas szó, mert később látni fogjuk, hogy a létrehozott heurisztikák hatékonysága nagymértékben függ a vizsgált modell struktúrájától is.

Az általam kidolgozott megközelítést szemlélteti a 13. ábra, a korábban a szaturáció működését bemutató ábra kiegészítéseként. Valóban, az előretüzelés úgymond kiegészítője a szaturációs algoritmusnak, amely szintén a bemeneti Petri-háló alapján deríti fel az elérhető állapotok halmazát. Rendkívül lényeges azonban a csillaggal megjelölt kimenet milyensége, attól függ ugyanis, hogy az eredményt felhasználja-e a szaturációs algoritmus a futása során. A „hasznos” előretüzelések eredménye és a szaturációs algoritmus által felderített állapottérből alakul ki a rendszer elérhető állapotainak halmaza.



13. ábra: Architektúrális ábra a heurisztikus-előretüzeléshez

Az ebben a fejezetben bemutatott módszer a párhuzamos algoritmushoz lett hozzáillesztve, de könnyedén kiegészíthető a szekvenciális változat is. A párhuzamos szaturációs algoritmus a *PetriDotNet* modellellenőrző keretrendszerbe lett integrálva, az előretüzeléses algoritmus ehhez lett hozzáfejlesztve. Az implementációhoz mindkét esetben a C# nyelvet választottuk.

5.1 Korábbi heurisztikán alapuló megoldások

A dolgozatban bemutatott algoritmus alapját a [21]-ban leírtak adják, azonban ez egy munkaállomások hálózatára optimalizált változata a szaturációs algoritmusnak. Az előretüzelés itt a csomópontokon eltüzelt tranzíciókból számított statisztikákon alapul. A statisztikák alapján futási időben különféle mintákat alakít ki az algoritmus, amelyet felhasználva a csomópontokat klaszterezi. Ezen csoportok alapján történik a tranzíciók előretüzelése. Az algoritmus memóriaigényét úgy csökkentették, hogy a csomópontokon eltüzelt tranzíciókat gráf alakban reprezentálták, ezáltal a mintáknak egy kompakt és hatékony kezelésére van lehetőség.

A HSF-SPIN eszköz [32] egy kiegészítés a széles körben használt SPIN modellellenőrző keretrendszerhez [31]. A kiegészítés a SPIN keretrendszer Promela modellező nyelvét használja a rendszer modelljének megadására, amely a SPIN-el ellentétben nem a kimerítő verifikációt célozza meg, hanem a rendszer működésében egy-egy specifikus, „érdekes" rész meglétét ellenőrzi (holtpont, adott tulajdonságú állapotok). Az előny itt abban mutatkozik meg, hogy egy specifikus eset irányított keresése hatékonyabb a heurisztikákat alkalmazva, mint ha a kimerítő verifikációt alkalmazzuk. A HFS-SPIN-t alkalmazva arra is lehetőség nyílik, hogy sokkal nagyobb méretű állapottér esetén is ellenőrizzük a követelményeket, olyanoknál, amelyekre esetleg a kimerítő verifikáció már nem működik. Az implementáció többféle heurisztikát is alkalmaz, ilyen az A*, Best-First, IDA*, amelyekről a hivatkozott irodalomban részletesen leírás található.

5.2 Az előretüzelés működésének bemutatása

Az előretüzeléses algoritmus motivációját – hasonlóan a [21]-ben közöltekhez – az adta, hogy a párhuzamos algoritmust egy 8 processzormaggal rendelkező számítógépen vizsgálva azt tapasztaltam, hogy a processzormagok összes kihasználtsága nagymértékben függött a vizsgált modelltől. Egyes esetekben a számítási kapacitások kihasználtsága nem haladta meg a 60%-ot. Ez magyarázható *Amdahl* törvényével [30], amely felső korlátot ad az elérhető sebességnövekedésre párhuzamos rendszerek esetén. Esetemben a letörés a modellek nagy részében 4 mag alkalmazásakor megtörtént, utána jelentős sebességnövekedés nem volt tapasztalható további processzormagok alkalmazásával. Ez azt jelenti, hogy a számítási erőforrások körülbelül 40%-a kihasználatlan maradt ezekre a modellekre, ezért volt fontos az, hogy valamilyen módon tovább lehessen növelni a kihasználtságot. Az előretüzelés megalkotásával célom ezen maradék erőforrások egy részének kihasználásával tovább csökkentetni a futási időt.

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

Az előretüzeléshez tehát szükség van egy csomópontra és a rajta eltüzelendő tranzícióra. Ezek kiválasztása azonban nem triviális feladat, ugyanis előfordulhat, hogy a

kiszámított csomópont teljesen fölösleges olyan szempontból, hogy arra soha nem lesz szüksége az algoritmusnak a futása során. Ez annak az esetnek felel meg, amikor az előretüzelés eredményét reprezentáló állapothalmaz nem része a rendszer ténylegesen elérhető állapotai halmazának. A szaturációs algoritmus sem hajtja végre minden csomóponton minden tranzíció tüzelését, csak azokat, amelyek ténylegesen hatással vannak az adott csomópontra. Az előretüzeléshez tehát nem megfelelő az, hogy tetszőlegesen kiválasztunk egy csomópont-tranzíció párt, mert így nagy valószínűséggel sok lesz a fölösleges műveletvégzés, ezek számát pedig minimalizálni kell, mivel az ez idő alatt felhasznált erőforrásokat esetleg a szaturációs algoritmus futására lehetett volna felhasználni (ez akár az algoritmus lassulásához is vezethet). A probléma kezeléséhez több dologra is szükség van:

- Korlátozni kell az egyidőben végezhető előretüzelési műveletek számát
- Az előretüzelést kisebb prioritású szálnak kell végrehajtania, hogy semmiképp se vegye el a számítási erőforrásokat a párhuzamos algoritmustól. Erre a problémára nyújt megoldást a később bemutatásra kerülő saját *ThreadPool* (szálkészlet, pool) implementáció.

A leírtak alapján az előretüzelés hasznosságát két csoportba sorolhatjuk be:

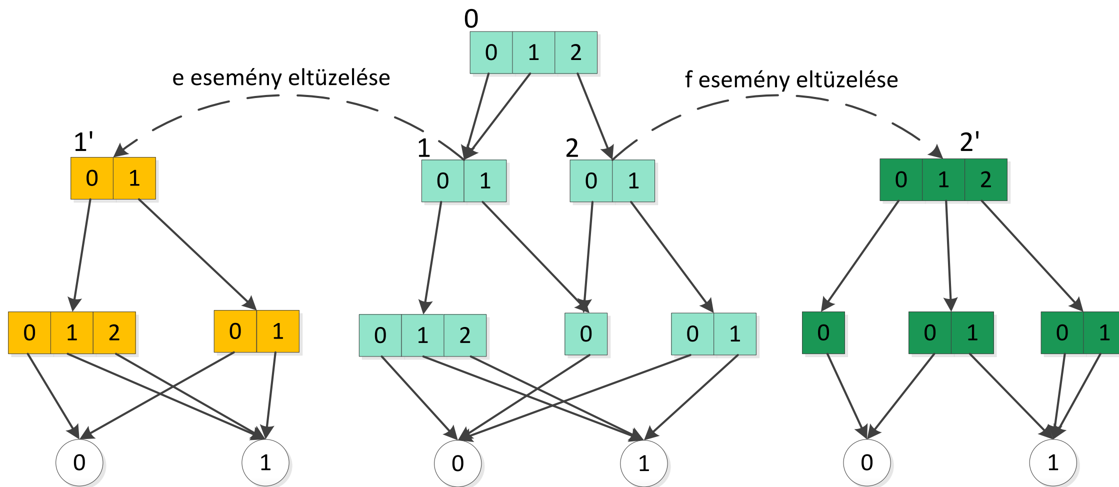
1. az előretüzelés teljesen felesleges, mert a tüzelési gyorsítótárba bekerült értéket az algoritmus nem fogja felhasználni
2. az előretüzelés olyan csomópontot eredményezett, melynek részfája által reprezentált állapotok halmaza részhalmaza a rendszer elérhető állapotai halmazának. Ez az az eset, amelynek eredményét a szaturációs algoritmus fel tudja használni.

A megfelelő tranzíció kiválasztására az implementáció során különféle heurisztikákat alkalmaztam, amelyek a csomópont tulajdonságait használják fel, ezekről bővebb információ az 5.3. fejezetben található.

A 14. ábra szemlélteti az előretüzelés működését. Az ábrán egy többértékű döntési diagram egy részlete látható. A szaturációs algoritmus során az 1-es és 2-es csomópontok szaturálttá váltak, így előretüzelést tudunk elindítani rajtuk. Tegyük fel, hogy e közben a párhuzamos algoritmus a 0. csomópont szaturációjával fogja folytatni a futását. Az 1. csomóponton valamely *e* eseményt eltüzelve kapjuk az 1’ csomópont által reprezentált állapothalmazt. Ez az eredmény azonban nem lesz hasznos számunkra a későbbiekben, mert ez a tüzelés ezen a csomóponton a bemeneti rendszer struktúrája miatt nem is fordulhatott volna elő, azaz ezt a szaturációs algoritmus nem fogja felhasználni. Az ilyen esetek elkerüléséhez azonban előzetes információval kellene rendelkezni az állapottérrel kapcsolatban, ez pedig nem kivitelezhető hatékony módon.

Ezzel szemben a 2. csomóponton az *f* eseményt eltüzelve a 2’ csomópont, mint gyökér alatti részfát kapjuk. Ez az előretüzelés hasznos olyan szempontból, hogy ha a szaturációs algoritmus ezt a tüzelést szeretné végrehajtani a 2-es csomóponton, akkor erre már nem lesz szükség, mert a tüzelési gyorsítótárban egyből megtalálja az eredményt. Az előretüzelésnek tehát az ilyen esetekben van hatékonyságnövelő szerepe. A cél tehát a „rossz” előretüzelések számának minimalizálása, amelynek egyik módja heurisztikák alkalmazása.

Az ábrán csak a jobb átláthatóság miatt van 3-3 terminális csomópont feltüntetve, valójában csak 1-1 terminális csomópont van.

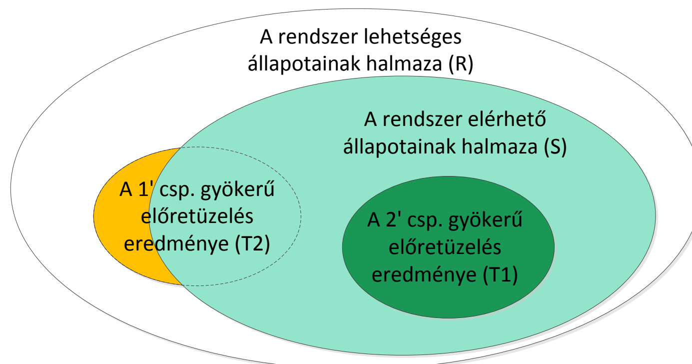


14. ábra: A tranzíció-előretüzelés szemléltetése

A korábban leírtaknak megfelelően az egyes előretüzelések által létrehozott állapothalmazok és a rendszer lehetséges, illetve elérhető állapotai halmazának egymáshoz való viszonyát szemlélteti a 15. ábra. A rendszer elérhető és lehetséges állapothalmazai között az $S \subseteq R$ reláció áll fenn.

A példát reprezentáló halmazok esetén:

- A hasznos előretüzelésre teljesül a $T1 \setminus S = \emptyset$ ($\setminus$ itt a halmazok közötti különbséget jelenti)
- A nem hasznos előretüzelés esetén a $T2 \setminus S \neq \emptyset$ reláció áll fenn.



15. ábra: A tranzíció-előretüzelés során keletkező állapothalmazok egymáshoz való viszonyának szemléltetése

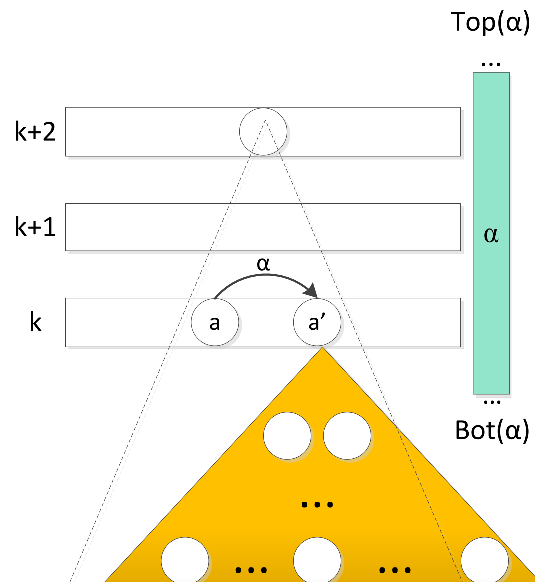
5.3 Az eltüzelendő tranzíció kiválasztása, a heurisztikák bemutatása

Az előretüzelés esetén kulcsfontosságú a megfelelő tranzíció kiválasztása, ehhez többféle heurisztikát dolgoztam ki. A heurisztikákat a *Strategy* [13] tervezési mintának megfelelően implementáltam, így akár futási időben is könnyen lehet váltani közöttük.

A jelenlegi implementációban több heurisztikát is kidolgoztam:

- *Naiv heurisztika*: A csomópont szintjéhez képest egy 2-vel feljebbi szintre hatással levő tranzíciót választ ki. Ez azt jelenti tehát, hogy ha az eltüzelendő tranzíció az α és a csomópont szintje a k, akkor teljesülnie kell a $Bot(\alpha) \leq k + 2 \leq Top(\alpha)$ egyenlőtlenségnek. A heurisztika ily módon történő megválasztásának oka, hogy az előretüzelést az eredeti csomóponthoz viszonylag közel hajtjuk végre és az esemény lokalitás miatt ezen tüzelés eredményére nagy valószínűséggel szükség lesz a későbbiekben, ugyanis ezt a lokalitást használja ki a szaturációs algoritmus is.

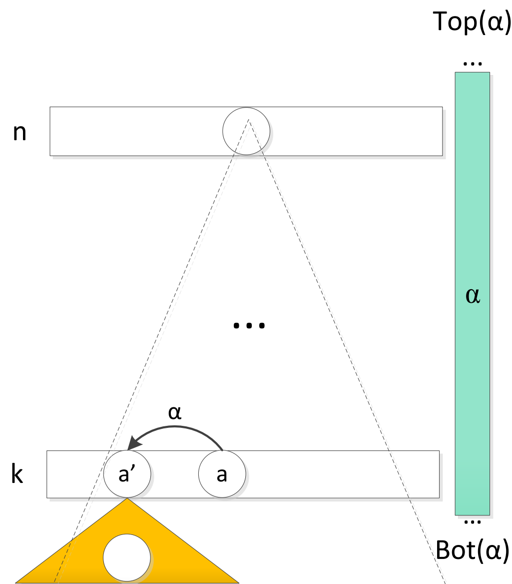
  A naív heurisztika működését illusztrálja a 16. ábra. Az *a* csomóponton eltüzeljük az α eseményt, ennek eredménye lesz az *a'* csomópont gyökerű részgráf (háromszög szemlélteti). Ezt az eredményt a szaturációs algoritmus viszonylag hamar fel fogja használni, hiszen csak két szinttel feljebbi tranzíció került eltüzelésre. A szaggatott vonallal jelölt háromszög azt a döntési diagramot ábrázolja, ami akkorra alakul ki, amikor a *k+2-ik* szinten levő csomópont válik szaturálttá. A kék téglalap azt mutatja, hogy az α esemény mely szintekre van hatással.



16. ábra: A naív heurisztika szemléltetése

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

  Az uppermost heurisztika működését illusztrálja a 17. ábra. A naív heurisztikával ellentétben itt a döntési diagramban egy távoli szintet is befolyásoló tranzíciót tüzelünk el, ezért ezt az eredményt csak később fogja felhasználni az algoritmus. Az ábrán az is látható, hogy az előretüzelés nem feltétlenül tökéletes, azaz van olyan részhalmaza az előretüzelésnek, amely nem része a tényleges állapottérnek.



17. ábra: Az uppermost heurisztika szemléltetése

- *T-invariánsokon alapuló heurisztikák:* A T-invariánsok a rendszer működésében egy ciklusnak felelnek meg, a kezdeti állapotból a ciklus elemein végigmenve visszajutunk a kezdeti állapotba. A T-invariánsokon alapuló heurisztikák célja egy ilyen ciklus valamely tranzíciójának eltüzelése és innen kiindulva egy állapottér-rész felderítése.
  A T-invariánsokra hasonlóan értelmezhető a Top és a Bot, mint a tranzíciókra. Jelölje T valamely T-invariánst, amelynek elemei a $\{t_1, t_2, \ldots, t_n\}$ tranzíciók, ekkor:

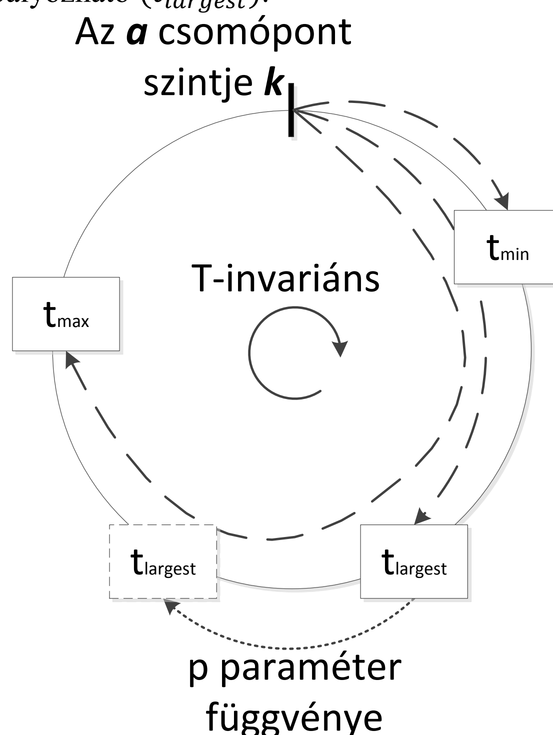
  - $Top(T) = max(Top(t_i)), ahol\ t_i \in T$ és $1 \leq i \leq n$
  - $Bot(T) = min(Bot(t_i)), ahol\ t_i \in T$ és $1 \leq i \leq n$

  - *MaxTop (T-invariáns) heurisztika:* A legnagyobb Top szinttel rendelkező T-invariánsból a legnagyobb Top szintű tranzíciót választja ki. Ha több T-invariáns vagy tranzíció is van ilyen tulajdonsággal, akkor a sorrendben az elsőt választja ki.

  - *MinTop (T-invariáns) heurisztika:* A legnagyobb Top szinttel rendelkező T-invariánsból a legkisebb Top szintű tranzíciót választja ki, amelyre teljesül az is, hogy ez még a csomópont szintje vagy a felett van. Ha nincs ilyen tulajdonságú tranzíció a T-invariánsban, akkor a heurisztika nem ad vissza tranzíciót, azaz ebben az esetben nem tüzelünk előre.
  - *Largest (T-invariáns) heurisztika:* A csomópont szintjét befolyásoló T-invariánsok közül a legtöbb tranzíciót tartalmazót veszi figyelembe (ha több ilyen is van, akkor a sorrendben az elsőt). A heurisztikához meg kell adni egy egész típusú paramétert (jelölje p). A kiválasztandó α tranzícióra teljesülnie kell, hogy $Top(\alpha) = k + p$, ahol k jelöli a csomópont szintjét. Ha nincs ilyen tulajdonságú tranzíció, akkor nem tüzelünk előre.

  A T-invariánsokon alapuló heurisztikák egymáshoz való viszonyát illusztrálja a 18. ábra. Fontos megjegyezni, hogy az ábra csupán illusztráció, a döntési diagram struktúrája és az, hogy az egyes invariánsokban előforduló

tranzíciók hol helyezkednek el jelentősen eltérhet az ábrán bemutatott esettől.
Az ábrán a kör szemlélteti a T-invariánst, mint egy ciklust a rendszerben. A T-invariáns tranzíciói is adott szinteken vannak értelmezve, ez alapján a ciklusban elhelyezhetjük valahol az a csomópont szintjét (k). Ez lesz az a csomópont, amelyen az előretüzelést végre fogjuk hajtani.
A *MinTop* heurisztika a csomópont szintjéhez közeli szintről, azaz a ciklusban is viszonylag közelről választja ki t_{min} -t mint eltüzelendő tranzíciót. A *MaxTop* heurisztika ezzel szemben egy viszonylag távoli szinten értelmezett tranzíciót választ ki, amely a ciklusban is távol helyezkedik el (t_{max}). A *Largest* heurisztika egy nagy ciklust választ ki, és a csomópont szintjéhez képest a ciklusban kiválasztott tranzíció a p paraméter értékével szabályozható ($t_{largest}$).



***18*. ábra: A T-invariánsokon alapuló heurisztikák szemléltetése**

5.4 Az előretüzelés implementációja

Az előretüzelés végrehajtásához tehát szükség van egy csomópontra és egy tranzícióra, e két paraméter alapján kell elkészíteni az előretüzelés eredményét reprezentáló részgráfot. A *PreFire* metódus szolgál ezen feladat végrehajtására, melynek pszeudokódja alább látható.

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

1      ℒ: lokális állapotok halmaza; l: szint; g,j: lokális állapot;
f,u,s:index;
       sat:bool
```

```
2     l = q csomópont szintje
3     Lock(FC(l));
4     if Find(FC(l), Key(l, q, e), s, sat) then
5           Unlock(FC(l));
6           return;
7     s = NewNode(l);
8     actPreNodeId = s.Id
9     s.preFiring = true
10    s.Key = Key(l, q, e);
11    AddOp(l, s);
12    Insert(Fc(l), s.Key, s, false);
13    Unlock(FC(l));
14    ℒ= Locals(l, q, e);
15    while ℒ ≠ 0 do
16          g =Pick(ℒ);
17          f = RecFire(e, l-1, <l,q>[g], s,  g);
18          if f ≠ <l-1, 0> then
19                Lock(<l,s>);
20                j = GetTargetState(l, g, e);
21                u = Union(l-1, f, <l,s>[j]);
22                if u ≠ <l,s>[j] then
23                      <l,s>[j] = u;
24                      Unlock(<l,s>);
25                      Confirm(l, j);
26                else
27                      Unlock(<l,s>);
28    if RemoveOp(l, s) then
29          if DWarcs(l, s) then
30                QSaturate(s);
31          else
32                Remove(l, s);
33    return <l, 0>;
```

A metódus alapjául a korábban közölt *SatRecFire* szolgált. A lényeges különbség a kettő között, hogy míg a *SatRecFire-t* egy felsőbb szinten levő csomópontból hívjuk meg, az előretüzeléshez nincs ilyen csomópont, hiszen itt csak előre szeretnénk dolgozni. E miatt az előretüzelést végrehajtó *PreFire* metódus paraméterezése is más, egy csomópont-tranzíció párt kell neki átadni.

Mivel a *PreFire* hívásnál nem állt rendelkezésre felsőbb szinten levő csomópont, ezért minden olyan részt elhagytam a *SatRecFire* kódjából, mely felfelé mutató élek beállítására vonatkozik. Megmaradt azonban itt is az a rész, mely a tüzelési gyorsítótárban keresést hajt végre, annak elkerülésére, hogy ha esetleg az eredeti szaturációs algoritmus már elkezdte végrehajtani az adott csomóponton az adott tranzíció eltüzelését, akkor azt ne hajtsuk végre még egyszer.

A jelenlegi implementációban akkor van lehetőség előretüzelést elindítani, amikor egy csomópont szaturálttá vált (ez lesz a paraméterként átadandó csomópont). Azért csak szaturált csomópontból indítunk előretüzelést, mert az a csomópont biztosan jelen lesz az állapotteret reprezentáló döntési diagramban, egy nem szaturált csomópontból kiindulva rossz eredményt kaphatunk a tüzelés révén.

Az eltüzelendő eseményt heurisztikák alkalmazásával választjuk ki. A vizsgált heurisztikákról részletes információk találhatók az 5.3. fejezetben. A műveletek ilyen módon történő indításának azonban az a problémája, hogy kisméretű modellek esetén is nagyon sok csomópont lesz a végeredményként kapott döntési diagramban, azaz nagyon

sok csomópont válik szaturálttá a futás során. Ha minden ilyen esetben új előretüzelést indítunk új szálon, akkor a létrejövő sok szál jelentősen leronthatja a teljesítményt. A probléma úgy oldható meg, hogy

- nem indítunk minden előretüzeléshez új szálat, hanem egy előre létrehozott szálkészletből gazdálkodunk
- egy időben csak egy előretüzelés lehet folyamatban.

5.4.1 Saját threadpool implementáció bemutatása

A .NET keretrendszer által nyújtott *ThreadPool*-nak [33] (szálkészlet, pool) több problémája is van a jelenlegi alkalmazásban:

- A pool-ban levő szálak száma nem lehet kevesebb, mint a processzormagok száma, felső korlát pedig a processzormagok számának 250-szerese, ezért nem alkalmas az előretüzelés menedzseléséhez. A mérési eredmények alapján látható lesz az, hogy a sok szál indítása jelentősen rontja az algoritmus teljesítményét.
- Nem lehet állítani a pool-ban levő szálak prioritását, ezt azonban az általam kifejlesztett módszer igényli.
- Nem lehet állítani a pool-ban levő szálak veremterületének méretét. Ez azért jelent problémát, mert a szaturáció rekurzív jellege miatt a nagyobb modellekre hamar megtelik a .NET keretrendszer *ThreadPool*-jában levő szálak verme.
- Nem lehet több pool-t létrehozni. Ez érthető is, hiszen általában egy adott architektúrán egy folyamat egy szálkészlettel gazdálkodhat, de a jelenlegi implementáció igényli ezt.

Ezen problémák orvoslására saját *ThreadPool* implementációt hoztam létre (a projektben ez *CustomThreadPool* néven található meg). A mérések azt igazolták, hogy a .NET-es *ThreadPool* hatékonyságától kis mértékben elmarad a saját implementációé. Ez azzal magyarázható, hogy a .NET-es változat sok helyen natív kódban lett megírva szemben az általam tisztán C# nyelven megírt változattal szemben.

A *CustomThreadPool* jellemzői:

- A .NET-es változattal ellentétben ez nem statikus, így tetszőleges számú osztálypéldány létrehozható. Erre szükség is van az előretüzelés során.
- A pool-ban levő szálak száma / prioritása / veremmérete állítható. Ezek megoldást jelentenek a korábban vázolt problémákra.

A leírt tulajdonságok miatt volt szükség a saját threadpool implementációra, amely megfelelően tudja menedzselni az erőforrásokat az előretüzeléssel kiegészített párhuzamos algoritmus esetén.

A *CustomThreadPool* mellet a .NET-es *ThreadPool*-al is végeztem méréseket, ebben az esetben minden szálnak azonos volt a prioritása. A mérési eredmények a 0. fejezetben találhatók.

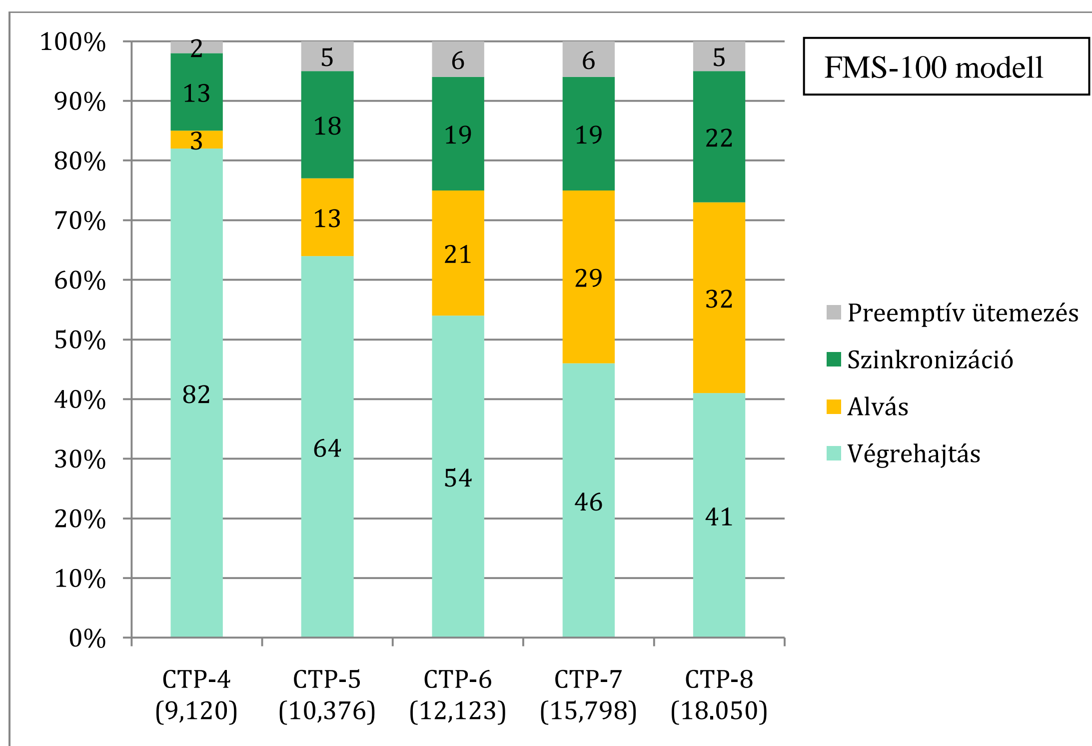
5.4.2 A szálak menedzselése

A mérések alapján elmondható az, hogy az erőforrás-kihasználtság problémáján a legtöbb esetben nem segít, ha a párhuzamos algoritmus futtatásához alkalmazott szálak számát növeljük. Például egy 4 processzormaggal rendelkező architektúrán a legkisebb

futási időt - a Slotted Ring modellt kivéve - minden esetben akkor kaptam, ha 4 szálat alkalmaztam a szaturáció végrehajtásához. Ennél több szál alkalmazása esetén már futási időben növekedés volt tapasztalható. Ezt a jelenséget szemlélteti a 19. ábra, amelyet a később bemutatásra kerülő FMS-100 modell vizsgálatával készítettem. A szaturációt végző szálak menedzseléséhez a saját *CustomThreadPool* (CTP) implementációt alkalmaztam, az ábrán látható az, hogy az egyes esetekben hány szál volt a pool-ban és mennyi lett a futási idő. Látható, hogy a 4 szálat tartalmazó pool esetén lett legnagyobb a futási időnek az állapottér felderítésre fordított része és ekkor lett legkisebb a futási idő. A szálak számát tovább növelve fokozatosan romlik ez az arány és növekedik az alvásra fordított idő. A szinkronizációra fordított idő lényegesen nem nő, e mellett az ábrán látható a preemptív ütemezés miatt elvett futási jogból eredő várakozás aránya is.

Egyedül a Slotted Ring modell esetén volt az tapasztalható, hogy a pool-ban levő szálak számának növelésével nem jelentkezett szignifikáns lassulás a futási időben.

A diagram adataiból jól látható, hogy a tartalék erőforrások kihasználtságán nem segít az, hogyha tovább növeljük a szálak számát a pool-ban, ezért ebben a formában az előretüzelés sem lehet hatékony. A megoldást az jelentheti, ha az előretüzelést végző szálak prioritását kisebbre állítjuk be, mint a ténylegesen szaturációt végző szálakét. Ez volt a fő motiváció a saját *ThreadPool* implementáció elkészítésekor, hiszen ebben paraméterezhető a pool-ban levő szálak prioritása. A prioritást a .NET keretrendszer ThreadPriority enumerációjából lehet kiválasztani; ebben öt érték található a szálak prioritásának beállítására.



19. ábra: A processzormagok kihasználtságának alakulása a CustomThreadPool-ban levő szálak számának függvényében (FMS-100 modell)

A vizsgált 4 processzormagos architektúrán a szálak menedzselése az alábbiak szerint történik:

- A szaturáció végrehajtásához létrehoztam egy pool-t 4 szállal, amelyek prioritása Normal (Pool-1).
- Az előretüzeléshez használt szálkészlet (Pool-2) szálainak prioritása BelowNormal. Ezzel a prioritással garantálható az, hogy az előretüzelés sosem fog erőforrást elvenni egy szaturációt végző száltól. Ha egy szaturációt végző szál futásra késszé válik, akkor az ütemező egyből elveszi a futási jogot az alacsonyabb prioritású előretüzelést végző száltól. Az előretüzeléses algoritmus futási idejét 1 és 4 darab szálat tartalmazó pool-al vizsgáltam. A 4 szálat tartalmazó szálkészletet azért vizsgáltam, mert bár egy időben egy előretüzelés hajtható végre, de a 4 szállal ezt az egy műveletet is párhuzamosan hajthatjuk végre.

6 Mérési eredmények

A szaturációs algoritmus implementációjának alapját jelentő [5]–ben nem számoltak be szignifikáns sebességnövekedésről, sőt sok esetben jelentősen elmaradt a párhuzamos algoritmus futási ideje a szekvenciális változatétól. A TDK dolgozat keretében elkészült saját implementációnk valós sebességnövekedést tudott felmutatni több modellre is. Ezt az algoritmust egészítettem ki az előretüzelést végző modullal a szakdolgozat keretében.

A különböző heurisztikákat mérésekkel hasonlítottam össze és elmondható az, hogy egyes esetekben további sebességnövekedés érhető el az előretüzelés alkalmazásával, illetve van olyan modell, amelyre a szekvenciális algoritmus futási idejéhez képest elmaradt a párhuzamos algoritmusé, de az előretüzelést is alkalmazva már lényeges különbség nem volt tapasztalható a két eset között.

6.1 A mérési környezet

A mérési infrastruktúra egy 4 processzormaggal és 4 GB memóriával rendelkező Intel számítógépből állt. A méréseket Windows 7 operációs rendszeren végeztem a .NET 4.0 keretrendszer 64 bites változatát felhasználva. A szálak menedzseléséhez a saját threadpool implementációt (*CustomThreadPool*, a mérések során a CTP rövidítéssel utalok rá), illetve a .NET keretrendszer által nyújtott *ThreadPool*-t használtam fel (a mérések során .NET TP rövidítéssel utalok rá).

6.2 A vizsgált modellek

A méréseket több valós életből vett rendszer Petri-háló alapú modelljén végeztem el, amelyek a B függelékben megtalálhatók. Az egyes modellekről részletes információk találhatók a [5]-ben. A vizsgált modellek:

- Slotted Ring (Réselt Gyűrű) [B1]: Egy hálózati protokoll modellje, amelyben a résztvevő felek gyűrű alakú topológián keresztül kommunikálnak egymással.
- FMS (Flexible Manufacturing System – Adaptív Gyártórendszer) [B2]: Egy olyan gyártósor modelljéről van szó, amely képes alkalmazkodni az idő közben bekövetkezett tervezett vagy véletlenül történt változásokhoz.
- Kanban [B3]: Egy termelési folyamat modellje, amely hatékonyan alkalmazható az olyan kérdések eldöntésében, hogy mikor és mennyi terméket kell előállítani.

6.3 A mért értékek

Minden egyes mérés esetén megtalálható az adott modell állapotterének felderítéséhez szükséges futási idő és ennek az időnek a relatív aránya a szekvenciális változat futási idejéhez képest (ez jelenti a 100%-ot). Az egyes előretüzeléses változatoknál ezeken felül a további értékek is fel lettek tüntetve:

- TA (Találati Arány, százalékos érték): Az előretüzelés során létrehozott csomópontok és ebből a szaturációs algoritmus által ténylegesen felhasznált csomópontok számának aránya.

- RÁ (Referencia Átlag): Az előretüzelés eredményei közül azon csomópontokra mutató referenciák átlaga, amelyeket a szaturációs algoritmus ténylegesen felhasznált.
- FM (Fa Magassága): Az előretüzelés során létrejövő azon részfák átlagos magassága, amelyeket a szaturációs algoritmus ténylegesen felhasznált. Az előretüzelés eredményeként létrejövő csomópont 1-1 ilyen fa gyökere.

A párhuzamos algoritmusok esetén mind a .NET *ThreadPool*-al mind pedig a saját threadpool implementációval le lettek mérve az egyes modellekre az értékek (ebben az esetben kétféle módon is az 5.4.2. fejezetben ismertetett pool-okban levő szálak számának állításával). Például a CTP 4-1 jelölés arra utal, hogy a *CustomThreadPool* lett felhasználva a szálak menedzseléséhez, a párhuzamos szaturációs algoritmust egy 4 szálat tartalmazó szálkészlet hajtotta végre, míg az előretüzelést egy 1 szálat tartalmazó szálkészlet.

A *PetriDotNet* keretrendszer többféle szintezési módot nyújt a döntési diagramok kialakítására:

- P-invariáns alapú szintezés: a döntési diagram szintjeit és azt, hogy az egyes szinteken mely helyek állapotai vannak elkódolva a Petri-háló P-invariánsai alapján kerül kialakításra.
- Manuális szintezés: Kézzel lehet beállítani azt, hogy az egyes szinteken hány hely állapotai legyenek elkódolva, de akár helyenként is meg lehet adni azt, hogy melyik szinten legyen elkódolva.

6.4 Slotted Ring modell

A Slotted Ring (továbbiakban SR) modellre vonatkozó mérési eredmények a 20. ábrán láthatók. A T-invariáns felderítő modul az SR modell esetén nem működik megfelelően, ezért nincsenek feltüntetve az egyes T-invariáns alapú előretüzeléses változatok.

Itt az SR-150 modell esetén P-invariánsos, SR-100 modellnél pedig manuális szintezést alkalmaztam. Az állapottér-felderítés végén kialakuló döntési diagramnak SR-100 esetén már $2{,}6 \cdot 10^{105}$ állapota van.

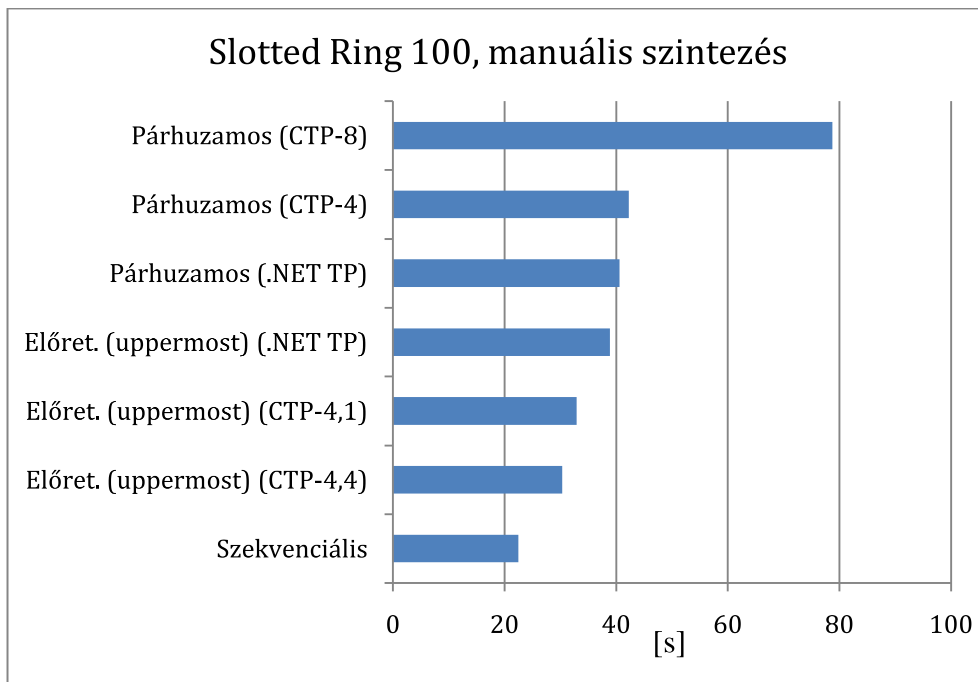
Látható az, hogy a SR-150 modellre jelentős, 30% körüli futási időbeli csökkenés volt tapasztalható. Ez már olyan állapottér méret, amely esetén a párhuzamos algoritmus hatékony tud lenni, a szinkronizációs költségekből jelentkező lassulást ellensúlyozni tudja az algoritmus párhuzamos volta. Azt is érdemes megfigyelni, hogy a párhuzamos algoritmus hatékonyságát az *Uppermost* heurisztika tudja tovább növelni a leginkább. Ennek oka az lehet, hogy a SR modellnél a sok szint miatt megfelelő lehet egy olyan heurisztika, amely távoli tranzíciót választ ki.

Az SR-100 modellre a manuális szintezés mellett a szekvenciális algoritmus hatékonyabb, mint a párhuzamos változatok. A párhuzamos algoritmus ebben az esetben nem tud igazán hatékony lenni, de az *Uppermost* heurisztika használatával azonban jól látható módon jelentősen lecsökken a hátrány a szekvenciális változathoz képest. Mindkét modell esetén megfigyelhető az is, hogy ott volt leghatékonyabb az előretüzelés ahol legnagyobb lett a találati arány.

Slotted Ring N							
Szint-ezés	N	Típus	Futási idő (s)	Arány (%)	TA (%)	RÁ	FM
P-invariáns	150	Szekvenciális	70,35	100,00	--	--	--
		Párhuzamos (.NET TP)	49,01	69,67	--	--	--
		Párhuzamos (CTP-4)	55,39	78,73	--	--	--
		Párhuzamos (CTP-8)	52,98	75,31	--	--	--
		Párhuzamos előret. (naiv) (.NET TP)	49,67	70,60	55,34	3,00	271,90
		Párhuzamos előret. (naiv) (CTP-4,1)	50,89	72,34	43,43	3,00	234,13
		Párhuzamos előret. (naiv) (CTP-4,4)	52,65	74,84	50,00	3,00	232,76
		Párhuzamos előret. (uppermost) (.NET TP)	**46,29**	**65,80**	**91,64**	**8,00**	**16,83**
		Párhuzamos előret. (uppermost) (CTP-4,1)	47,79	67,93	96,05	8,00	11,00
		Párhuzamos előret. (uppermost) (CTP-4,4)	49,69	70,63	88,05	8,00	12,11
Manuális (szintenként 8 hely)	100	Szekvenciális	22,50	100,00	--	--	--
		Párhuzamos (.NET TP)	40,61	180,52	--	--	--
		Párhuzamos (CTP-4)	42,25	187,81	--	--	--
		Párhuzamos (CTP-8)	78,73	349,97	--	--	--
		Párhuzamos előret. (naiv) (.NET TP)	39,73	176,61	37,18	22,00	78,83
		Párhuzamos előret. (naiv) (CTP-4,1)	34,23	152,16	11,59	22,00	30,63
		Párhuzamos előret. (naiv) (CTP-4,4)	43,74	194,43	33,33	21,00	66,80
		Párhuzamos előret. (uppermost) (.NET TP)	38,86	172,74	24,93	25,00	12,77
		Párhuzamos előret. (uppermost) (CTP-4,1)	32,90	146,25	21,11	26,00	25,83
		Párhuzamos előret. (uppermost) (CTP-4,4)	**30,32**	**134,78**	**44,77**	**26,00**	**17,33**

20. ábra: A Slotted Ring modellre vonatkozó mérési eredmények - 1

A futási időknek az alakulását szemlélteti a 21. ábra is az SR-100 modellre vonatkozóan. A diagramon az előretüzeléses változatok közül csak az *Uppermost* futási ideje van feltüntetve. Ezen az ábrán is jól látható az, hogy a szekvenciális algoritmus lett a leggyorsabb erre a modellre, de az *Uppermost* előretüzelést alkalmazva jelentősen lecsökkent a párhuzamos algoritmus hátránya az olyan változathoz képest, amely nem alkalmaz előretüzelést.



21. ábra: A Slotted Ring modellre vonatkozó mérési eredmények - 2

6.5 FMS modell

Az FMS modellre vonatkozó mérési eredmények a 22. és 23. ábrán láthatók. A 22. ábra az FMS-8 modell, P-invariánsos szintezés esetén tartalmazza a mérési eredményeket. A párhuzamos változatok futási idejei közül a *MinTop* heurisztikával kiegészített lett a leggyorsabb (.NET TP). Ezzel a változattal körülbelül 7%-kal csökkent a futási idő a szekvenciális algoritmushoz képest. Megfigyelhető azonban, hogy nem ennek a heurisztikának volt a legnagyobb a találati aránya. A *MinTop* heurisztikával létrejövő fáknak a magassága és a referenciák száma nem mondható soknak a többi heurisztikához viszonyítva.

FMS 8						
Szint-ezés	Típus	Futási idő (s)	Arány (%)	TA (%)	RÁ	FM
P-invariáns	Szekvenciális	157,32	100,00	--	--	--
	Párhuzamos (.NET TP)	147,76	93,92	--	--	--
	Párhuzamos (CTP-4)	149,11	94,78	--	--	--
	Párhuzamos (CTP-8)	146,89	93,37	--	--	--
	Előret. (naiv) (.NET TP)	146,44	93,08	21,62	702,00	3,20
	Előret. (naiv) (CTP-4,1)	147,95	94,04	10,71	683,00	3,33
	Előret. (naiv) (CTP-4,4)	147,97	94,06	27,78	618,00	3,33
	Előret. (uppermost) (.NET TP)	148,18	94,19	14,58	205,00	4,00
	Előret. (uppermost) (CTP-4,1)	146,21	92,94	32,50	295,00	2,33
	Előret. (uppermost) (CTP-4,4)	146,87	93,36	22,45	214,00	3,00
	Előret. (tinv - maxTop) (.NET TP)	147,23	93,59	16,67	1378,00	3,00

	Előret. (tinv - maxTop) (CTP-4,1)	148,28	94,25	20,51	829,00	3,67
	Előret. (tinv - maxTop) (CTP-4,4)	147,46	93,73	7,84	340,00	5,00
	Előret. (tinv - minTop) (.NET TP)	**145,74**	**92,64**	**13,33**	**356,00**	**3,33**
	Előret. (tinv - minTop) (CTP-4,1)	146,64	93,21	7,69	2926,00	3,00
	Előret. (tinv - minTop) (CTP-4,4)	147,31	93,64	15,56	125,00	4,67
	Előret. (tinv - largest) (.NET TP)	146,29	92,99	23,08	2289,00	3,33
	Előret. (tinv - largest) (CTP-4,1)	148,28	94,25	38,46	1716,00	3,00
	Előret. (tinv - largest) (CTP-4,4)	145,95	92,77	15,38	3432,00	2,00

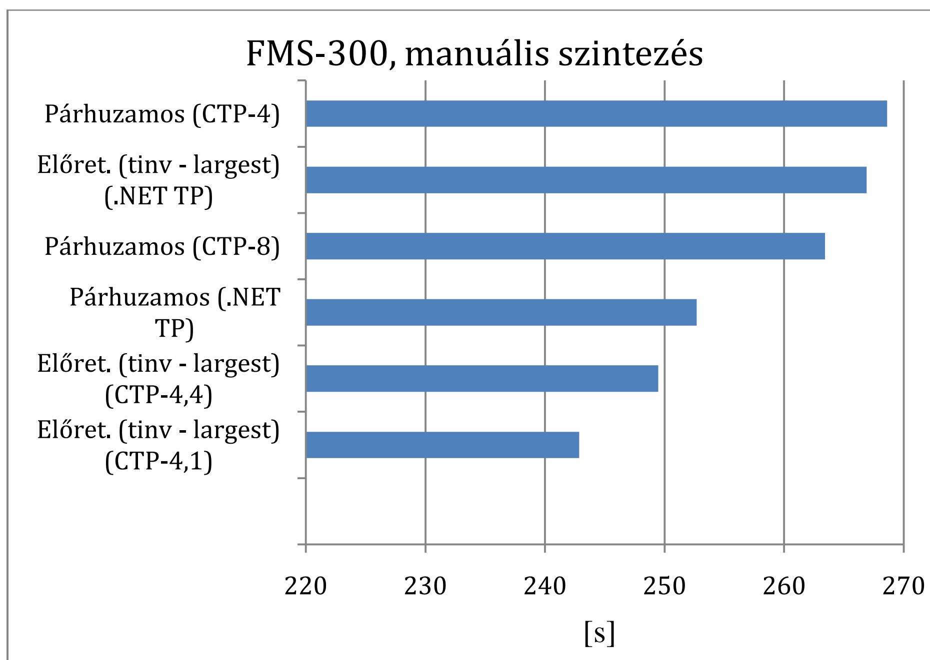
22. ábra: FMS-8, P-invariánsos szintezés mérési eredmények

A 23. ábrán az FMS-300 modellre vonatkozó mérési eredmények láthatók (az állapottere $\sim 10^{25}$ nagyságrendű). Ebben az esetben csak a futási idők lettek feltüntetve a mérések viszonylag hosszú ideje miatt. Az általam vizsgált modellek közül erre tapasztaltam a legnagyobb mértékű gyorsulást a szekvenciális algoritmushoz képest (átlagosan ~70%-al csökkent a futási idő a párhuzamos változatok alkalmazásával). Fontos megjegyezni, hogy a *Naív* és *Largest* heurisztika (CTP 4-1 ütemezéssel) a leggyorsabb párhuzamos algoritmushoz képest (.NET TP) további ~10 másodperces futási időbeli csökkenést produkált (CTP-4,1).

FMS N			
Szint-ezés	Típus	Futási idő (s)	Arány (%)
Manuális (szintekként 1 hely)	Szekvenciális	939,05	100,00
	Párhuzamos (.NET TP)	252,67	26,91
	Párhuzamos (CTP-4)	268,61	28,60
	Párhuzamos (CTP-8)	263,41	28,05
	Előret. (naiv) (.NET TP)	273,57	29,13
	Előret. (naiv) (CTP-4,1)	**242,46**	**25,82**
	Előret. (naiv) (CTP-4,4)	246,05	26,20
	Előret. (uppermost) (.NET TP)	295,48	31,47
	Előret. (uppermost) (CTP-4,1)	292,65	31,16
	Előret. (uppermost) (CTP-4,4)	306,75	32,67
	Előret. (tinv - maxTop) (.NET TP)	284,09	30,25
	Előret. (tinv - maxTop) (CTP-4,1)	295,60	31,48
	Előret. (tinv - maxTop) (CTP-4,4)	300,30	31,98
	Előret. (tinv - minTop) (.NET TP)	288,98	30,77
	Előret. (tinv - minTop) (CTP-4,1)	306,27	32,61
	Előret. (tinv - minTop) (CTP-4,4)	291,69	31,06
	Előret. (tinv - largest) (.NET TP)	266,90	28,42
	Előret. (tinv - largest) (CTP-4,1)	**242,86**	**25,86**
	Előret. (tinv - largest) (CTP-4,4)	249,46	26,57

23. ábra: FMS-300, manuális szintezés mérési eredmények - 1

A 24. ábra a párhuzamos algoritmusok és a *Largest* heurisztika változatainak futási idejeit szemlélteti az FMS-300 modellre vonatkozóan, manuális szintezés esetén. Ebben az esetben a szekvenciális algoritmus jelentősen lassabb lett a párhuzamos változatoknál, ezért nem lett feltüntetve. Ezen az ábrán is látható az, hogy a *Largest* heurisztikával a leggyorsabb előretüzelés nélküli párhuzamos változathoz képest is ~10 másodperces gyorsulás volt tapasztalható (ez további 4%-os csökkenés a futási időben).



24. ábra: FMS-300, manuális szintezés mérési eredmények - 2

6.6 Kanban modell

A Kanban-100 modellre vonatkozó mérési eredmények a 25. ábrán láthatók. A Kanban modell esetén a szekvenciális algoritmus futási ideje lett a legkevesebb. A párhuzamos algoritmus leggyorsabb változatával is 14%-al több lett a futási idő ehhez képest. A táblázat adataiból azonban jól látható, hogy az *Uppermost* heurisztika alkalmazásával ezt a 14%-os hátrányt sikerült ~1,5%-ra lecsökkenteni. A T-invariánsos heurisztikák a Kanban modell esetén nem bizonyultak hatékonynak, az esetek többségében lassítottak a párhuzamos algoritmus futási idején.

Kanban 100						
Szint-ezés	Típus	Futási idő (s)	Arány (%)	TA (%)	RÁ	FM
Manuális (szintenként 2 hely)	Szekvenciális	48,17	100,00	--	--	--
	Párhuzamos (.NET TP)	57,86	120,12	--	--	--
	Párhuzamos (CTP-4)	54,92	114,01	--	--	--
	Párhuzamos (CTP-8)	56,44	117,17	--	--	--
	Előret. (naiv) (.NET TP)	54,54	113,22	100,00	1761	5
	Előret. (naiv) (CTP-4,1)	55,03	114,24	100,00	1761	5
	Előret. (naiv) (CTP-4,4)	56,31	116,90	100,00	1677	5
	Előret. (uppermost) (.NET TP)	53,38	110,82	63,64	80	2
	Előret. (uppermost) (CTP-4,1)	50,3	104,42	78,51	36	2
	Előret. (uppermost) (CTP-4,4)	**48,92**	**101,56**	**80,55**	**41**	**2**
	Előret. (tinv - maxTop) (.NET TP)	62,01	128,73	44,56	79	2
	Előret. (tinv - maxTop) (CTP-4,1)	59,83	124,21	80,47	43	2
	Előret. (tinv - maxTop) (CTP-4,4)	62,6	129,96	77,54	43	2
	Előret. (tinv - minTop) (.NET TP)	56,41	117,11	2,08	156	0,2
	Előret. (tinv - minTop) (CTP-4,1)	56,04	116,34	5,77	152	0,6
	Előret. (tinv - minTop) (CTP-4,4)	58,06	120,53	3,85	174	0,4
	Előret. (tinv - largest) (.NET TP)	59,01	122,50	44,16	101	2
	Előret. (tinv - largest) (CTP-4,1)	62,04	128,79	68,08	39	2
	Előret. (tinv - largest) (CTP-4,4)	61,93	128,57	79,17	43	2

25. ábra: Kanban modellre vonatkozó mérési eredmények

6.7 A mérési eredmények értékelése

A mérések alapján megállapítható, hogy:

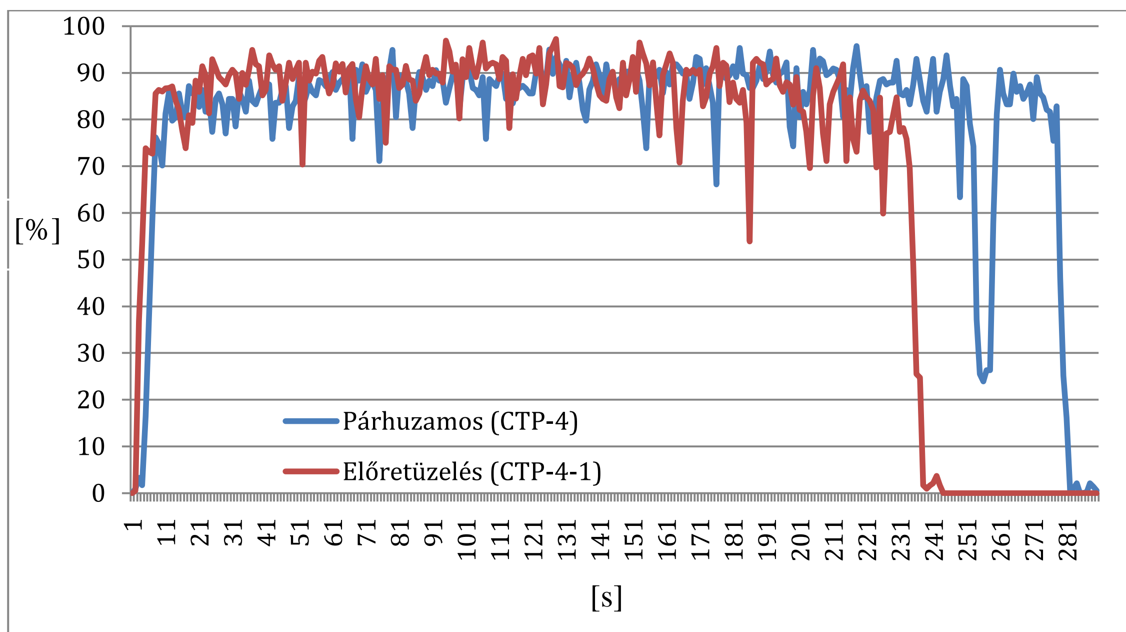
- A heurisztikák hatékonysága nagymértékben modellfüggő. Az esetek többségében jelentős további gyorsulás érhető el alkalmazásukkal (például SR modell esetén az *Uppermost*, FMS modell esetén a *Largest*), de olyan esetre is van példa, hogy a párhuzamos szaturáció algoritmus futási idején rontott az előretüzelés (például Kanban modell, T-invariánsos heurisztikák). Jogosan merül fel a kérdés, hogy ennek mi lehet az oka, ha az előretüzelés csak kisebb prioritású szálon fut. Ez lehet a szál kontextus váltások magas száma miatt, illetve az előretüzeléses algoritmus is abba a gyorsítótárban rakja az értékeket, amelyet a szaturációs algoritmus használ. Ebből következik, hogy elkerülhetetlen a zárkezelés miatti többletköltség, amely ha sok előretüzelést indítunk, akkor jelentősen ronthat az algoritmus hatékonyságán.

- Az eredményekből látszódik, hogy nem feltétlenül abban az esetben a leghatékonyabb az előretüzelés, ahol 100%-os lett a találati arány (például Kanban modell). Előfordulhat, hogy egy olyan heurisztika sokkal hatékonyabb, amelynek ugyanis kisebb a találati aránya, de az előretüzelés során kisebb is a létrejövő fáknak az átlagos magassága. Ennek oka, hogy az előretüzelés

eredményére esetleg várni kell és mivel az kisebb prioritású szálon fut, ezért a nagyobb fa létrehozásával megnő a futási idő is.

- A Kanban modell esetén volt látványos az a jelenség, hogy a párhuzamos algoritmus futási ideje még jelentősen elmaradt a szekvenciális algoritmus futási idejéhez képest, de az előretüzelés alkalmazásával ez a hátrány jelentősen lecsökkent.

- Az előretüzelés alkalmazásával ténylegesen növelni lehet a számítási erőforrások kihasználtságát, ezt szemlélteti a 26. ábra. Az ábrán a processzormagok kihasználtsága látható az idő függvényében az FMS-100 modell állapotterének felderítése alatt. Kék szín szemlélteti a párhuzamos algoritmust, míg a piros színnel a *Naív* heurisztikával kiegészített változat esetén láthatók az értékek. Az alkalmazott szálak száma az 5.4.2. fejezetben leírtaknak megfelelően lett beállítva. A diagramon több dolog figyelhető meg:
    - Az előretüzeléssel kiegészített algoritmus futási ideje kevesebb lett, mint a párhuzamos változaté
    - A processzor kihasználtság kis mértékben ugyan, de javult a párhuzamos változathoz képest (átlagosan 4-5%)
    - Az ábrán látható letörések oka az automatikus szemétgyűjtés (garbage collection), ilyenkor kis mértékben visszaesik a processzormagok kihasználtsága. Mindkét változat esetén megfigyelhető egy nagy letörés is a futás vége felé, amelynek oka, hogy akkor történt meg a fölöslegessé vált csomópontok eltávolítása a döntési diagramból.



26. ábra: Az előretüzeléssel kiegészített és az önmagában futtatott párhuzamos algoritmus cpu-kihasználtságának szemléltetése

7 Összefoglalás

A szakdolgozat keretében implementáltam a heurisztikán alapuló tranzíció-előretüzelés módszerét, amely kiegészítője a TDK dolgozatban [2] elkészített párhuzamos szaturációs algoritmusunknak. Használatával a párhuzamos algoritmus által felhasznált számítási erőforrások aránya tovább növelhető és ezzel több esetben is további teljesítmény növekedést sikerült elérnem az állapottér-felderítés során.

A szakdolgozat keretében elkészült eredmények két nagy csoportra bonthatók. Az elméleti eredményeket tekintve:

- Egy TDK dolgozat keretében kidolgoztuk a párhuzamos szaturációs algoritmust, amely az alapjául szolgált a heurisztika alapú fejlesztéseknek.
- Kifejlesztettem a heurisztikán alapuló tranzíció előretüzelés algoritmusát és hogy azt hogyan lehetne hozzáilleszteni a TDK dolgozatban elkészült párhuzamos szaturációs algoritmusunkhoz.
- Több heurisztikát is kifejlesztettem, amelyeknek az a célja, hogy az előretüzeléshez kiválasszák az eltüzelendő tranzíciót a döntési diagram csomópontjának tulajdonságai alapján.

A gyakorlati eredmények az alábbiak szerint foglalhatók össze:

- Egy TDK dolgozatban elkészült a párhuzamos szaturációs algoritmus implementációja.
- Az előretüzeléses algoritmus implementációja C# nyelven és integrálása a *PetriDotNet* modellellenőrző keretrendszerbe.
- A kialakított öt heurisztika implementálása és összehasonlítása különböző méréseken és statisztikákon keresztül.
- Saját threadpool implementációt dolgoztam ki, amely az előretüzelést végző szálak hatékony menedzseléséhez szükséges.

A TDK dolgozat keretében elkészült párhuzamos szaturációs algoritmus mérési eredményeihez képest az előretüzelés alkalmazásával több modellre is sikerült valós sebességnövekedést elérni. Olyan esetre is volt példa, hogy a párhuzamos algoritmus futási ideje egy-egy modellre elmaradt a szekvenciális változatéhoz képest, de az előretüzeléssel kiegészítve ez a hátrány jelentősen lecsökkent. Fontos azonban megjegyezni azt, hogy az előretüzelés hatékonysága nagyban függ a vizsgált rendszer struktúrájától és jelentős különbségek is lehetnek két heurisztika között egy adott modellre.

Az elkezdett munka számos továbbfejlesztési lehetőséget hordoz magában:

- Adaptív heurisztikák megalkotása, amely valós időben a rendszer struktúrájától függően képes a legmegfelelőbb heurisztikát kiválasztani és az alapján végezni az előretüzelést.
- Elosztott modellellenőrző algoritmus létrehozása, amely munkaállomások hálózatán tud hatékonyan működni. Egy ilyen rendszer esetén a párhuzamosság eléréséhez valamilyen üzenetváltáson alapuló konkurencia modell bevezetésére is szükség lehet.

Irodalomjegyzék

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

[2] Jámbor Attila, Szabó Tamás: Aszinkron rendszerek modellellenőrzése párhuzamos technikákkal, TDK dolgozat, 2010

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

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

[5] Ezekiel J, Lüttgen G, Siminiceanu R. Can saturation be parallelised? On the parallelisation of a symbolic state-space generator. PDMC conference on Formal methods: Applications and technology. 2006:331-346

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

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

[8] Murata, T.; , Petri nets: Properties, analysis and applications, Proceedings of the IEEE , vol.77, no.4, pp.541-580, Apr 1989 doi: 10.1109/5.24143

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

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

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

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

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

[14] Oriol Roig , Jordi Cortadella , Enric Pastor, Verification of Asynchronous Circuits by BDD-based Model Checking of Petri Nets, Proceedings of the 16th International Conference on Application and Theory of Petri Nets, p.374-391, June 26-30, 1995

[15] Bryant, R.E.; , "Graph-Based Algorithms for Boolean Function Manipulation," *Computers, IEEE Transactions on* , vol.C-35, no.8, pp.677-691, Aug. 1986
doi: 10.1109/TC.1986.1676819

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

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

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

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

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

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

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

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

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

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

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

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

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

[29] A Performance Comparsion of ReaderWriterLockSlim with ReaderWriterLock, MSDN Magazine, 2007, http://blogs.msdn.com/b/pedram/archive/2007/10/07/a-performance-comparison-of-readerwriterlockslim-with-readerwriterlock.aspx

[30] Amdahl Gene, Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities, AFIPS Conference Proceedings (30): 483-485, 1967

[31] SPIN Model Checker: http://spinroot.com/spin/whatispin.html

[32] HSF-SPIN heurisztikán alapuló modellellenőrző kiegészítés a SPIN keretrendszerhez: http://www.albertolluch.com/research/tools

[33] Jeffrey Richter, The CLR's Thread Pool, MSDN Magazine, 2003, http://msdn.microsoft.com/en-us/magazine/cc164139.aspx

Ábrák jegyzéke

1. A tesztelés, szimuláció, modellellenőrzés és tételbizonyítás szemléltetése 5
2. Egy egyszerű Petri-háló 8
3. Termelő-fogyasztó rendszer Petri-háló alapú modellje 9
4. Petri-háló T-invariánsainak szemléltetése 10
5. Az $f = (x \land y) \lor (\neg x \land y)$ karakterisztikus függvényhez tartozó döntési fa 12
6. Karakterisztikus függvény döntési fa alapú reprezentációja 13
7. Karakterisztikus függvény BDD alapú reprezentációja... 13
8. Karakterisztikus függvény ROBDD alapú reprezentációja 14
9. Kvázi-redukált MDD 15
10. A szaturáció folyamata 19
11. Az MDD zárolási módszerek szemléltetése 30
12. Az írási-olvasási zárak zár-kompatibilitási mátrixa... 31
13. Architektúrális ábra a heurisztikus előretüzeléshez... 33
14. A tranzíció-előretüzelés szemléltetése.. 36
15. A tranzíció-előretüzelés során keletkező állapothalmazok egymáshoz való viszonyának szemléltetése 36
16. A *Naív* heurisztika szemléltetése 37
17. Az *Uppermost* heurisztika szemléltetése 38
18. A T-invariánsokon alapuló heurisztikák szemléltetése 39
19. A processzormagok kihasználtságának alakulása a CustomThreadPool-ban levő szálak számának függvényében (FMS-100 modell) 42
20. A Slotted Ring modellre vonatkozó mérési eredmények – 1 47
21. A Slotted Ring modellre vonatkozó mérési eredmények – 2 48
22. FMS-8, P-invariánsos szintezés mérési eredmények.. 49
23. FMS-300, manuális szintezés mérési eredmények – 1 49
24. FMS-300, manuális szintezés mérési eredmények – 2 50
25. Kanban modellre vonatkozó mérési eredmények 51
26. Az előretüzeléssel kiegészített és az önmagában futtatott párhuzamos algoritmus cpu-kihasználtságának szemléltetése... 52
27. Slotted Ring modell Petri-hálója.. 65
28. FMS modell Petri-hálója.. 66
29. Kanban modell Petri-hálója 66

Függelék A

[A1] A szekvenciális szaturáció SatFire függvénye

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

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

[A2] A szekvenciális szaturáció SatRecFire függvénye

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

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

```
                                                            állapotra
11                u = Union(l-1, f,  ⟨l|s⟩[j])
12                if (u ≠ ⟨l|s⟩[j]) then //ha az unió új csomópontot
                                                      eredményezett
13                     ⟨l|s⟩[j] = u //beállítjuk az élet
14 s = CheckIn(l, s)
15 PutInCache(FIRE, l, e, q, s)
16 return s
```

[A3] A párhuzamos szaturáció Saturate függvénye

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

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

[A4] A párhuzamos szaturáció SatFire függvénye

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

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

[A5] A párhuzamos szaturáció SatRecFire függvénye

```
SatRecFire(in: e: esemény, l: szint, q: index, p: index, i: lokális
állapot): index
// Egy új, ⟨l|s⟩ gyökerű MDD-t hoz létre, amelyre teljesül, hogy
$\mathcal{N}^*_{\leq l}(\mathcal{N}_e(\mathcal{B}(< l, q >)))$.

1  ℒ: lokális állapotok halmaza; g,h,j: lokális állapot; f,u,s:index; sat:bool
2  if l < Bot(e) then // az esemény nincs hatással erre a szintre
3     return q;
4  Lock(FC(l)); // a módosítás erejéig zároljuk az FC-t
5  if Find(FC(l), Key(l, q, e), s, sat) then
6     if !sat then // ha a megtalálta csomópont még nem szaturált
7           j = GetTargetState(l+1, i, e); // az e esemény i-ből j
                                            // állapotba visz
8           SetUpArc(l, s, p, j);
9           s = ⟨l|0⟩; // mivel s még nem szaturált, ZeroNode a
visszatérés
10    Unlock(FC(j));
11    return s;
12 s = NewNode(l); // ha nem volt az FC-ben találat, új csomópontot
                   //hozunk létre
13 s.Key = Key(l, q, e);
14 j = GetTargetState(l+1, i, e); // az e esemény i-ből j állapotba
visz
15 SetUpArc(l, s, p, j);
16 AddOp(l,s); // feljegyezzük, hogy egy szál éppen dolgozik a
csomóponton
17 Insert(Fc(l), s.Key, s, false);
18 Unlock(FC(l));
19 ℒ = Locals(l, q, e); // azok az állapotok, amelyek e-re
engedélyezettek
20 while ℒ ≠ 0 do
21    g = Pick(ℒ);
22    f = SatRecFire(e, l-1, ⟨l|q⟩[g], s,  g); // g állapotra eltüzeljük
e-t
23    if f ≠ ⟨l − 1|0⟩ then // ha nem ZeroNode volt a RecFire
visszatérése,
                         // azaz az FC-ben már benne volt a tüzelés
                         // eredménye
24          Lock(⟨l|s⟩.dw); // a módosítás erejéig zároljuk a
csomópontot
25          j = GetTargetState(l, g, e);
26          u = Union(l-1, f, ⟨l|s⟩[j]); // unió az él korábbi értékével
27          if u ≠ ⟨l|s⟩[j] then
28                ⟨l|s⟩[j] = u;
29                Unlock(⟨l|s⟩.dw);
30                Confirm(l, j);
31          else
32                Unlock(⟨l|s⟩.dw);
33 if RemoveOp(l, s) then // ha más nem dolgozik s-n, és nem mutat rá
                         // felfelé él
34    if DWarcs(l, s) then
35          QSaturate(s); // ha volt tüzelhető esemény, akkor
szaturáljuk
36    else
37          Remove(l, s); // egyetlen eseményt sem tudtuk eltüzelni,
akkor
```

```
                              // fölösleges a csomópont, és törölhető
35    return ⟨l|0⟩;
```

[A6] A párhuzamos szaturáció NodeSaturated függvénye

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

1  q: index, i: lokális állapot
2  q = p; // elmentünk egy hivatkozást a csomópontra
3  p = CheckIn(k, p);
4  if k = K then // ha a legfölső szintű csomópontot szaturáltuk,
                 // akkor elkészült a szaturáció
5     Terminate();
6     return;
7  Lock(FC(k));
8  Update(FC(k), ⟨k|p⟩.Key, p, true); // frissítjük az FC-t
9  Unlock(FC(k));
10 while GetUpArc(k, p, r, i) do
11    Lock(⟨k+1|r⟩.dw);
12    u = Union(k, p, ⟨k+1|r⟩[i]); // a föntebbi csomópontokban
beállítjuk
                                 // az elkészült csomópontot
13    if u ≠ ⟨k+1|r⟩[i] then
14          ⟨k+1|r⟩[i] = u;
15          Unlock(⟨k+1|r⟩.dw);
16          Confirm(k+1, i);
17          if ⟨k+1|r⟩.saturating then
18                SatFire(k+1, r, i); //az új állapotokban is
eltüzeljük
                                        //az összes eseményt
19    else
20          Unlock(⟨k+1|r⟩.dw);
21    if RemoveOp(k+1, r) then // ha már más nem dolgozik a fölső
cs.ponton
22          if ⟨k+1|r⟩.saturating then // és már elkezdtük szaturálni
23                NodeSaturated(k+1, r); // akkor most be is fejeződött
24          else
25                QSaturate(k+1, r); // egyébként elindítjuk a
szatuációját
26 if q ≠ p then
27    delete(⟨k|p⟩);
```

[A7] A párhuzamos szaturáció Remove függvénye

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

1  i: lokális állapot; q: index
2  Lock(FC(k));
3  Update(FC(k), ⟨k|p⟩.Key, ⟨k|p⟩, true); // bejegyezzük az FC-be a
törlést
4  Unlock(FC(k));
5  while GetUpArc(k, p, q, i) do
6     if RemoveOp(k+1, q) then // ha már más nem dolgozik a fölső
cs.ponton 7           if ⟨k+1|q⟩.saturating then // és már elkezdtük
```

```
szaturálni
8               NodeSaturated(k+1, q); // akkor most be is fejeződött
          else // egyébként
9               if DWarcs(k+1, q) then // ha van nem ZeroNode-ba
mutató
                                       // éle
10                    QSaturate(k+1, q); // akkor elindítjuk a
                                         // szaturációját
                else                   // különben
11                    Remove(k+1, q); // fölösleges a fölső csomópont
is
12 delete(⟨k|p⟩);
```

[A8] A párhuzamos szaturáció Initialize függvényének definíciója

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

[A9] A párhuzamos szaturáció Union függvényének definíciója

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

[A10] A párhuzamos szaturáció DWarcs függvényének definíciója

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

[A11] A párhuzamos szaturáció SetUpArc függvényének definíciója

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

[A12] A párhuzamos szaturáció GetUpArc függvényének definíciója

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

[A13] A párhuzamos szaturáció AddOp függvényének definíciója

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

[A14] A párhuzamos szaturáció RemoveOp definíciója

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

[A15] A párhuzamos szaturáció Find függvényének definíciója

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

[A16] A párhuzamos szaturáció Insert függvényének definíciója

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

[A17] A párhuzamos szaturáció Update függvényének definíciója

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

[A18] A párhuzamos szaturáció Locals függvényének definíciója

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

[A19] A párhuzamos szaturáció Pick függvényének definíciója

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

[A20] A párhuzamos szaturáció NewNode függvényének definíciója

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

[A21] A párhuzamos szaturáció CheckIn függvényének definíciója

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

[A22] A párhuzamos szaturáció Key függvényének definíciója

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

[A23] A párhuzamos szaturáció QSaturate függvényének definíciója

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

[A24] A párhuzamos szaturáció Terminate függvényének definíciója

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

[A25] A párhuzamos szaturáció Confirm függvényének definíciója

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

[A26] A PreFire metódus pszeudokódja

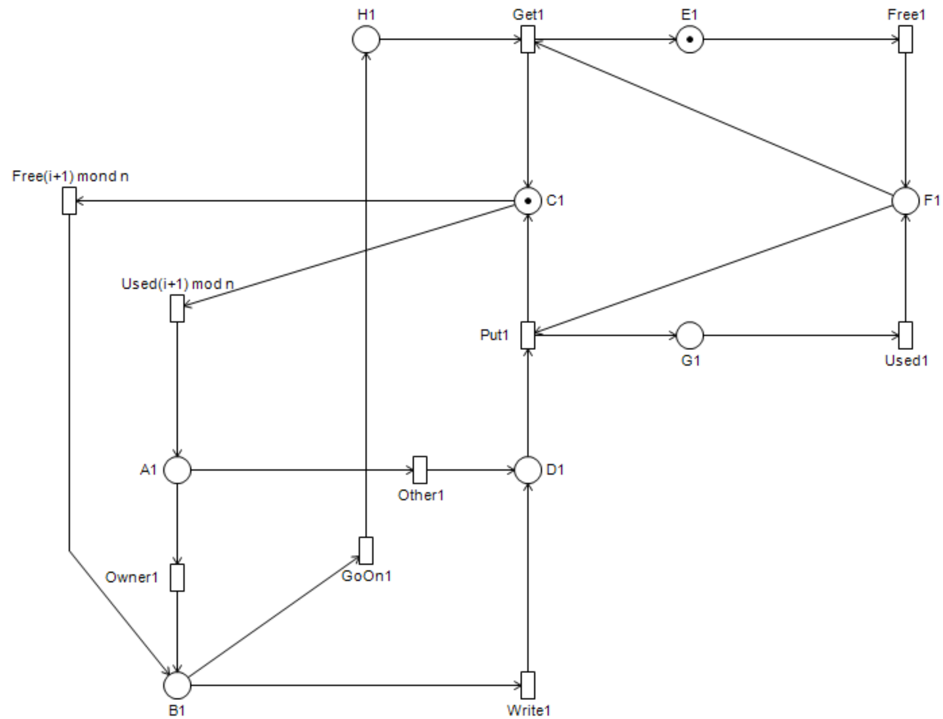
```
PreFire(in: q: csomópont, e: esemény)
Az n csomóponton az e esemény előretüzelését végzi el, az eredményt
berakja a tüzelési gyorsítótárba.

1      ℒ: lokális állapotok halmaza; l: szint; g,j: lokális állapot;
f,u,s:index;
       sat:bool
2      l = q csomópont szintje
3      Lock(FC(l));
4      if Find(FC(l), Key(l, q, e), s, sat) then
5            Unlock(FC(l));
6            return;
7      s = NewNode(l);
8      actPreNodeId = s.Id
9      s.preFiring = true
10     s.Key = Key(l, q, e);
11     AddOp(l, s);
12     Insert(Fc(l), s.Key, s, false);
13     Unlock(FC(l));
14     ℒ= Locals(l, q, e);
15     while ℒ ≠ 0 do
16           g =Pick(ℒ);
17           f = RecFire(e, l-1, <l,q>[g], s,  g);
18           if f ≠ <l-1, 0> then
19                 Lock(<l,s>);
20                 j = GetTargetState(l, g, e);
21                 u = Union(l-1, f, <l,s>[j]);
22                 if u ≠ <l,s>[j] then
23                       <l,s>[j] = u;
24                       Unlock(<l,s>);
```

```
25                      Confirm(l, j);
26                else
27                      Unlock(<l,s>);
28    if RemoveOp(l, s) then
29          if DWarcs(l, s) then
30                QSaturate(s);
31          else
32                Remove(l, s);
33    return <l, 0>;
```

Függelék B

[B1] A Slotted Ring modell Petri-hálója

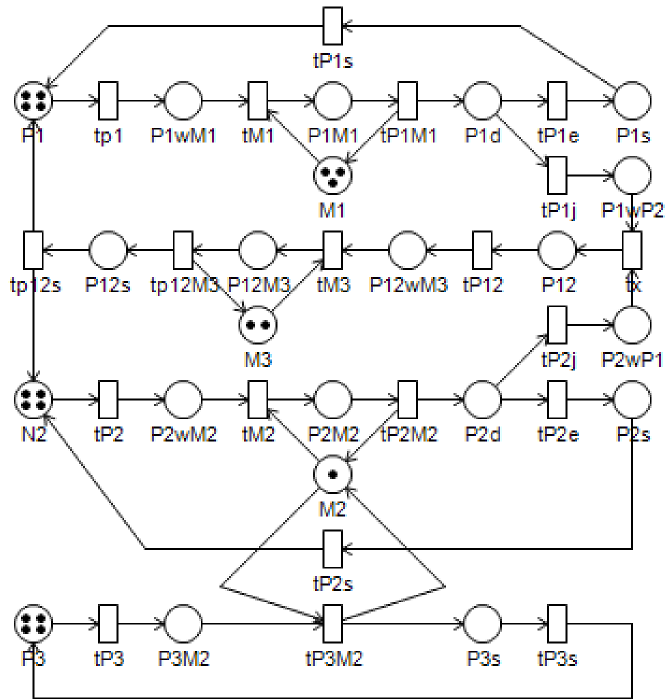
Az 27. ábrán látható Petri-háló a kommunikáció egy résztvevőjét reprezentálja. A Slotted Ring N jelölés arra utal, hogy a kommunikációnak N darab résztvevője van.



27. ábra: Slotted Ring modell Petri-hálója

[B2] Az FMS modell Petri-hálója

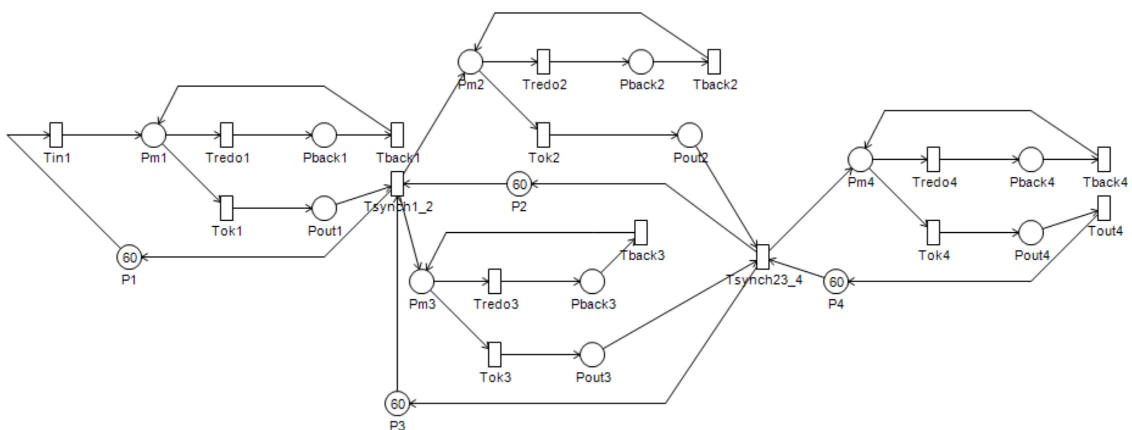
A modell paramétere a p_1, p_2 és p_3 helyeken található tokenek száma, erre utal az FMS N jelölés. Az 28. ábrán az FMS 4 modell látható.



28. ábra: FMS modell Petri-hálója

[B3] A Kanban modell Petri-hálója

A modell paramétere a p_1, p_2, p_3 és p_4 helyeken levő tokenek száma, erre utal a Kanban N jelölés. A 29. ábra a Kanban-60 modellt mutatja.



29. ábra: Kanban modell Petri-hálója