Debreceni Egyetem
Informatika Kar

Információs Rendszerek Szoftveres Szimulációja

Témavezető:
Bérczes Tamás
egyetemi tanársegéd

Készítette:
Szabó Ferenc
Programtervező informatikus
szakos hallgató

Debrecen
2010

Tartalomjegyzék

1. Bevezetés

Az elmúlt évtizedekben az információs technológia robbanásszerű fejlődése és a komplexitás növekedése a rendszerfejlesztési módszertanok gyökeres megújulását eredményezte. Az alapos és átfogó tervezés, az ellenőrzés, a verifikáció és validáció fejlesztési folyamatban betöltött szerepe meghatározóvá vált. Ezzel párhuzamosan, illetve ennek következménye képpen a modellezési eljárások alkalmazása általánossá vált a rendszertervezés valamennyi szakaszában. Bár a modellezési paradigmák és az eszközök folyamatosan fejlődtek, megannyi területen és problémakörben az ismertebb és széles körben alkalmazott módszerek nem rendelkeztek megfelelő kifejező erővel és eszközkészlettel a rendszerek teljeskörű tervezésére és elemzésére. A skálázhatóság biztosítása,a küldetés-kritikus rendszerek megbízhatósága és a teljesítménytervezés is ilyen sarkalatos kérdéskör. A rendszerfejlesztési folyamat következetességének növelése és a kudarc kockázatának csökkentése megkívánta olyan eljárások megalkotását, amelyekkel már a tervezési stádiumban megbizonyosodhatunk a rendszer helyességéről. Ennek folyományaként jött létre a szimbolikus modellellenőrzés, amely egy formális verifikációs technika.

Ahogy az eljárás nevéből következik, a modellellenőrzést a szimulálandó rendszer egy modelljén végezzük. A modellt általában egy magas szintű leírásból generáljuk, mint például egy processzus algerbra vagy a Petri-háló. A létrehozott modell tipikusan egy nemdeterminisztikus véges állapotú automata, amely leírja a rendszer lehetséges viselkedését. Ha a rendszert reprezentáló modellt elkészítettük, leellenőrizhetjük, hogy a modell megfelel-e egy formális specifikációnak. Ezt a specifikációt általában valamilyen temporális logikával adjuk meg, mint az LTL (Linear Time Logic) vagy a CTL (Computation Tree Logic). Ezekkel a logikákkal állapotok és állapotátmenetek tulajdonságait tudjuk kifejezni. Amint a modellt és az elvárt,illetve szükséges tulajdonságokat formalizáltuk, a modellellenőrző eszköz felépíti az állapotteret, és megállapítja, hogy a modell rendelkezik e ezekkel a tulajdonságokkal. Az ilyen típusú tradícionális modellellenőrző technika 100%-os biztonsággal kiértékeli a rendszert.

A modellelenőrzési paradigma egyik alapvető jellemzője az, hogy a verifikáció eredménye csak a modell pontosságától függ. Ezért ahogy a verifikációs technikák hatékonyabbá és elterjedtebbé váltak, igény mutatkozott a modellek és specifikációs formalizmusok kiterjesztésére is. Egy ilyen terület a valószínűségi modellellenőrzés is, mivel rengeteg, a valós

életben található folyamat eredendően sztochasztikus viselkedéssel bír. Jól szemléltetik ezt a hálózati protokollok,hibatűrő rendszerek, vagy a számtógépes hálózatok, melyek kiszámíthatatlansága csak valószínűségi alapokon modellezhető akkurátusan.

E dolgozat témája egy ilyen sztochasztikus modellező eszköz, a PRISM bemutatása. Használatához elengedhetetlen a szoftver saját modellező és lekérdező nyelvének ismerete, így ezeket külön fejezetben tárgyaljuk. A rendszer helyességének bizonyítására egy egyszerű modellt adunk, és a kiértékelés eredményeit összevetjük az analítikus eredményekkel. Majd egy összetettebb példában átfogóbb képet nyújtunk a lehetőségekről.

2. Valószínűségi modell-ellenőrzés

A valószínűségi modellellenőrzés egy formális verifikációs technika olyan rendszerek elemzésére, amelyek viselkedése eredendően sztochasztikus. A módszer hasonló a bevezetésben említett hagyományos ellenőrző eljáráshoz,viszont a modell állapotátmenetei kiegészülnek valószínűségi vagy időzítési információkkal, vagyis sztochasztikus viselkedést modellez. Az eljárást alkalmazzák a véletlen valószínűségi algoritmusoktól egészen a mestrséges intelligencia tervezéséig, a biztonságtechnikában, vagy akár a biológiai folyamatok modellezésében.

2.1. Valószínűségi modellek

Hogy a technikát alkalmazni tudjuk sztochasztikus viselkedést mutató rendszereken, meg kell adnunk a rendszer formális modelljét. Ehhez néhány gyakran alkalmazott reprezentációt használhatunk, mint például:

- Diszkrét idejű Markov-láncok(DTMC)
- Folytonos idejű Markov-láncok(CTMC)
- Markov döntési folyamatok(MPD)
- Sztochasztikus Petri-hálók
- Bayes-hálók

Ezek a modellek tipikusan a valószínűségszámítás és gráfelmélet kombinációjaként jöttek létre. A továbbiakban csak a Markov-láncokkal foglalkozunk, amelyet a legtöbb valószínűségi modellelemző használ.

Definíció *Tekintsük egymás után végrehajtott kísérletek sorozatát. Legyen* $E_1, E_2, ..., E_i, ...$ *egy teljes eseményrendszer, a* $\xi_n (n = 0, 1, 2...)$ *valószínűségi változók pedig olyanok, hogy* $\xi_n = i$ *ha az* n*-edik kísérletnél az* E_i *esemény fordul elő. Ha fennáll minden* n*-re és a változók összes lehetséges értékeire, hogy*

$$P(\xi_n = j \mid \xi_1 = i_1, \xi_2 = i_2, ..., \xi_{n-1} = i_{n-1}) = P(\xi_n = j \mid \xi_{n-1} = i_{n-1})$$

úgy azt mondhatjuk, hogy az egymást követő kísérletek, illetve a (ξ_n) *valószínűségi változók* ***egyszerű Markov-láncot*** *alkotnak.*

A Markov-láncok fontos speciális esetét képezik a homogén Markov-láncok. Ezeknél a $P(\xi_n = j \mid \xi_{n-1} = i)$ átmenetvalószínűségek függetlenek az n-től, azaz

$$P(\xi_n = j \mid \xi_{n-1} = i) = p_{ij}$$

írható. A Markov-láncokkal kapcsolatban rendszerint a következő terminológia szokásos: az E_i eseményeket a rendszer **állapotainak** nevezzük. A ξ_0 változó $P(\xi_0 = i) = P_i(0)$ eloszlását **kezdeti eloszlásnak** és a $P(\xi_n = j \mid \xi_{n-1} = i)$ feltételes valószínűséget **átmenetvalószínűségnek** nevezzük. Ha pedig $\xi_{n-1} = i$ és $\xi_n = j$, akkor azt mondjuk, hogy a rendszer az n-edik lépésben **átmenetet tett**.

Definíció *Tekintsünk egy homogén Markov-láncot*

$$p_{ij} = P(\xi_n = j \mid \xi_{n-1} = i)$$

átmenetvalószínűségekkel. Legyen továbbá $P(\xi_n = j) = P_j(n)$*. Nyilvánvalóan* $\sum p_{ik} = 1$ *és* $p_{ij} \geq 0$*. A* p_{ij} *átmenetvalószínűségek elrendezhetők a következő mátrix alakban:*

$$\pi = \begin{pmatrix} p_{11} & p_{12} & p_{13} & \dots \\ p_{21} & p_{22} & p_{23} & \dots \\ p_{31} & p_{32} & p_{33} & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}.$$

π az úgynevezett *átmenetvalószínűségek mátrixa*. A π mátrix négyzetes, elemei nem negatívak és a sorok összege 1. Egy ilyen mátrixot *sztochasztikus mátrixnak* nevezünk. Egy Markov-lánc egyértelműen meg van határozva a π mátrix és a $(P_j(0))$ kezdeti eloszlás megadásával.

Markov láncok és állapotainak osztályozása

Definíció *Azt mondjuk, hogy az E_k állapot elérhető az E_j állapotból, ha létezik olyan n, hogy $p_{ij}^{(n)} > 0$. Egy Markov-láncot irreducibilisnek nevezünk akkor, ha minden állapota elérhető minden állapotából.*

Definíció *Egy Markov-lánc állapotainak C halmazát zártnak nevezzük, ha egylépéses átmenettel nem lehet kijutni ebből a halmazból, tehát $p_{jk} = 0$, ha $E_j \in C$, és $E_k \notin C$. Nyilvánvalóan ekkor tetszőleges n-re is fennáll $p_{jk}^{(n)} = 0$, ha $E_j \in C$, és $E_k \notin C$.*

Az irreducibilis Markov-láncok állapotai egyetlen zárt halmazt alkotnak. Ha csupán egy zárt C halmaz állapotait tekintünk, úgy egy rész Markov láncot nyerünk, amely a többi állapottól függetlenül vizsgálható. Egy Markov-láncot *szétbonthatónak* nevezünk, ha állapotai két vagy több zárt halmazra bonthatóak.

Definíció *Ha egyetlen E_{kk} állapot képez egy zárt halmazt, azaz $p_{kk} = 1$, akkor ezt az E_{kk} állapotot abszorbeáló állapotnak nevezzük.*

Tétel *Tekintsünk egy tetszőleges, de rögzített E_j állapotot. Annak a valószínűsége, hogy a rendszer valamikor visszatér az E_j állapotba,*

$$f_j = \sum_{n=1}^{\infty} f_j^{(n)}.$$

Ha $f_j = 1$, azaz a visszatérés biztos, úgy a visszatérésig megtett lépésszám várható értéke, az átlagos visszatérési idő:

$$\mu_j = \sum_{n=1}^{\infty} n f_j^{(n)}.$$

A Markov-láncok állapotait a következo képpen osztályozhatjuk:

- Az E_j állapotot rekurrens állapotnak mondjuk, ha az E_j állapotba való visszatérés biztos, azaz $f_j = 1$.

- Az E_j állapotot tranziens állapotnak mondjuk, ha az E_j állapotba való visszatérés nem biztos, azaz $f_j < 1$.

- Az E_j rekurrens állapotot zérus állapotnak nevezzük, ha az átlagos visszatérési idő végtelen, azaz $f_j = 1$ és $\mu_j = \infty$.

- Az E_j állapotot periodikusnak mondjuk, t periódussal, ha az E_j állapotba való visszatérés csupán a $t, 2t, 3t, \ldots$ lépésnél következhet be, és $t > 1$ a legnagyobb ilyen tulajdonsággal rendelkező szám.

Definíció *Tekintsük a t paraméter véges vagy végtelen intervallumba eső értékeire értelmezett valós ξ_t valószínűségi változók összességét.A (ξ_t) sztochasztikus folyamatot* ***Markov folyamatnak*** *nevezzük, ha fennáll*

$$P(\xi_t \leq x \mid \xi_{u_1} = y_1, \xi_{u_2} = y_2, ..., \xi_{u_n} = y_n) = P(\xi_t \leq x \mid \xi_{u_n} = y_n)$$

valamennyi $u_1 < u_2 < ... < u_n < t$-re és a szóban forgó változók összes lehetséges értékeire. Ilyen módon a Markov folyamatok a Markov-láncok közvetlen általánosításának tekinthetők.

A (ξ_t) Markov-folyamatot ergodikusnak nevezzük, ha létezik a $\lim_{t\to\infty} P(\xi_t \leq x \mid \xi_0 = y)$ határeloszlásfüggvény, és független y-tól. Ekkor $\lim_{t\to\infty} P(t, x)$ is létezik és az előzővel megegyezik. A ξ_0 változó $P(0, x)$ eloszlását stacionáriusnak mondjuk, ha $P(\xi_t \leq x) = P(t, x) = P(0, x)$. A (ξ_t) Markov-folyamatot osztályozhatjuk még aszerint is, hogy ξ_t értékkészlete diszkrét vagy ξ_t változása folytonos, vagy ξ_t változása folytonos és ugrásszerű lehet.

2.2. Eszközök

Ebben a fejezetben a leggyakrabban használt valószínűségi modellellenőrzőket és azok tulajdonságait, képességeit hasonlítjuk össze. Ezek a szoftverek az ETMCC, az MRMC, és az YMER. Az általam használt szoftver, a PRISM modellelenőrző tárgyalása a következő fejezetben található. Elemzésük főbb szempontjai az implementáció, a modellek és lekérdezések specifikációjának nyelve és a támogatott algoritmusok valamint a használt adatszerkezetek.

Az ETMCC

Az ETMCC-t az Erlangen-Nürnberg Egyetem Sztochasztikus Modellezési csoportja és a holland Twente-i Egyetem Formal Methods & Tools csoportja fejlesztette. Ingyenes és nyílt forráskódú eszköz. A szoftvert Java nyelven implementálták, futtatásához a JVM 1.5-ös vagy frissebb verziója szükséges, és használható Linux, Windows és Solaris operációs rendszereken is. Az ETMCC-nek nincs parancssori felülete, minden műveletet a grafikus felületen kell elvégeznünk. Ez a grafikus felület tartalmaz egy szövegszerkesztőt, és használható a szükséges állományok betöltésére és az elemzés eredményeinek megjelenítésére.

Az ETMCC csak a CTMC típusú modellek megadását támogatja. A modell leírását a *tra-format* formátumban tehetjük meg, amelyet a TIPPtool és a DaNAMiCS eszközök generálnak. Ebben a formátumban minden sor egy tranzakciót jelöl, amelyhez megadunk egy forrás- és egy célállapotot, valamint egy intenzitást. Az ETMCC esetében a lekérdezések specifikálásához a CSL és az akció-alapú CSL(aCSL) logikákat használhatjuk. A numerikus számításokhoz a Jacobi és Gauss-Seidel iterációs algoritmusokat alkalmazza, míg az adatokat ritka mátrix adatszerkezetben tárolja.

Az MRMC

Az MRMC szoftvert a németországi Aachen-i Egyetem MOVES csoportja és a holland Twente-i Egyetem Formal Methods & Tools csoportja közösen fejlesztette. GNU/GPL licenszelt termék. A szoftver fejlesztőit az ETMCC inspirálta, így tekinthetjük annak jogutódjaként is. Az MRMC egy parancssorra épülő eszköz, amelyet C nyelven fejlesztet-

tek, és csak Linux platformon futtatható. Mind DTMC és CTMC típusú modelleket is támogat, továbbá ezek költségekkel ellátott típusait is. A modell leírásának formalizmusa ugyan az, mint az ETMC esetében, azzal a kivétellel, hogy az aCSL típusú szintaxis nem támogatott. A valószínűségeket vagy intenzitásokat egy *.tra* fájlban kell elhelyeznünk, az állapotok címkéit pedig egy *.lab* fájlban. A PCTL és a CSL temporális logikákkal és ezek költségekkel kiterjesztett változataival adhatjuk meg lekérdezéseinket. A használt iterációs algoritmusok megegyeznek az ETMCC-ben alkalmazottakkal, az adatokat pedig az úgynevezett *Harwell-Boeing rita mátrix*ban tárolja.

Az YMER

Az YMER-t Hakan Younes, a Pittsburgi egyetem munkatársa fejlesztette. A szoftver ingyenes, GNU/GPL licenszelt, legfrisebb(2005, február 1) verziója a 3.0-ás. Támogatja a CTMC és az általános fél-Markov-folyamat(GSMP) típusú modellek elemzését is. Az YMER statisztikai modellellenőrző algoritmusokat implementál, amelyek a diszkrét esemény szimuláción és a statisztikai mintavételes ellenőrzésen alapulnak. Emellett numerikus algoritmusokat is használ CTMC típusú modellek ellenőrzéséhez. Az YMER parancssori eszköz, véletlen szám generátora C nyelven, míg többi része C++ nyelven készült. A szimbolikus adatszerkezetekhez a CUDD csomagot használja, amely nem része a disztribúciónak, és külön telepítendő. A CTMC modellek numerikus számításait végző modul a PRISM program hibrid ellenőrző egysége. A szoftver csak Linux operációs rendszerekre telepíthető. A modellt a PRISM modellelemző nyelv egy részhalmazával adhatjuk meg. Így néhány funkció nem érhető el benne, például nem adhatunk meg a modellben intenzitásnak más intenzitások összegét vagy különbségét, csak szorzatát vagy hányadosát. A lekérdezések specifikálásához a CSL temporális logika használatos. Ezek kiértékeléséhez statisztikai mintavételező algoritmusokat, a minta utak létrehozásához pedig diszkrét esemény szimulációt alkalmaz. A statisztikai elemzéshez és a minták tárolásához elhanyagolhatóan kevés memória szükséges a teljes állapottér felépítésével összehasonlítva. A program által használt adatszerkezet nincs dokumentálva, kivége a numerikus számításokat végző csomag, amely a PRISM ritka mátrix és MTBDD hibrid modulját használja az adatok reprezentálására.

3. A PRISM

3.1. A szoftver áttekintése

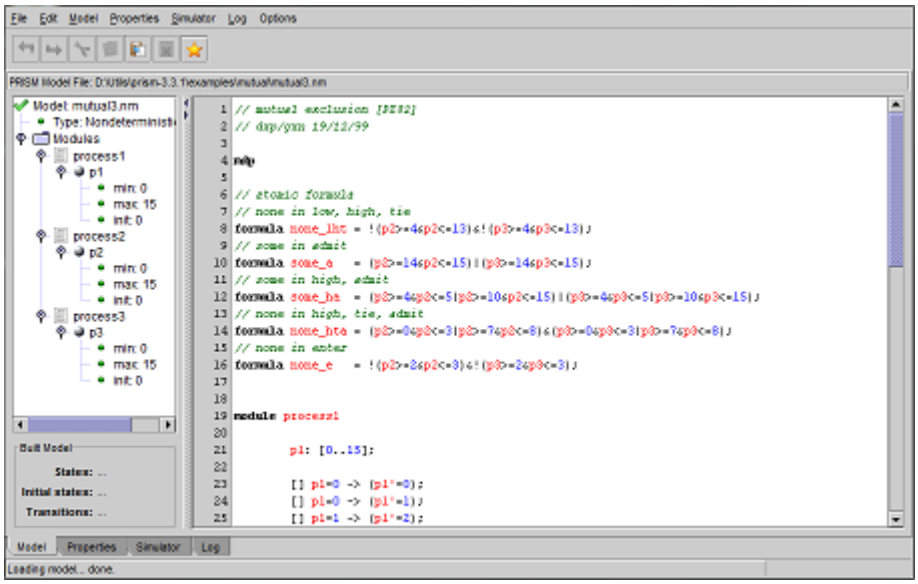
A PRISM egy széles körben használt sztochasztikus modellező és elemző eszköz, amellyel lehetőségünk van formális,valószínűségre épülő modellek megadására és elemzésére. A szoftvert 2001 és 2007 között a Birminghami Egyetem fejlesztette, 2007 júliusától viszont a fejlesztést végző csapat az Oxfordi Egyetemen folytatta munkáját. Ingyenes és nyílt forráskódú,GNU/GPL licenszelt termék. Fejlesztése Java és C++ nyelveken történt, pontosabban a felhasználói felület és a szintaktikai elemző Java nyelven, míg a sztochasztikus algoritmusok C++ nyelven. Az MTBDD adatszerkezetet a CUDD csomag egy átdolgozott verziója kezeli. A szoftver telepíthető Linux, Windows, Mac OS X és Solaris operációs rendszerekre is. A futtatáshoz a Java Runtime Environment(JRE) szükséges. A program használható grafikus felülettel és parancssori interfészen keresztül is. Funkcionalitásban teljesen megegyeznek, alattuk ugyanaz a modellelemző fut. Az utóbbi hasznosabb, ha nagy mennyiségű kötegelt feladatot bízunk a PRISM-re,vagy ha hosszú idejű elemzéseket végzünk a háttérben. A grafikus felületen elérhetőek a következő eszközök:

- -szövegszerkesztő ablak,amely a kiemeli a szintaktikai elemeket és jelzi a hibákat;
- -szerkesztő felület a modell lekérdezéseihez;
- -szimulációs eszköz a modell feltárásához és nyomkövetéséhez
- -rajzoló segédeszköz az analízis eredményeinek megjelenítéséhez.

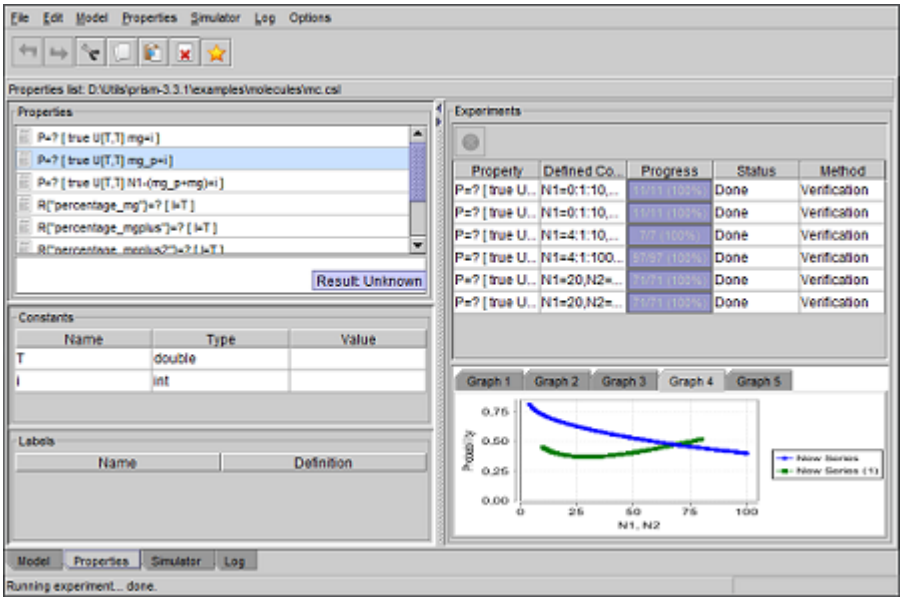
A szoftver grafikus intefészének szövegszerkesztő felülete az 1.ábrán látható. Parancssoros módban a PRISM kapcsolókkal vezérelhető. A teljes utasításlista lekérdezéséhez a `-help` kapcsolót adhatjuk meg:

```
prism -help
```

A program beállításait a *.prism* fájl tárolja, amelyet a grafikus és parancssoros felület is használ. Az alapértelmezett beállításokat a grafius interfész *Options* párbeszédablakának *Load Defaults* opciójával tehetjük meg, vagy a *.prism* fájl törlésével. A numerikus számításokat végző modul kiválasztása a grafikus felületen az *Options* párbeszédablak



1. ábra. A PRISM modell lerió felülete



2. ábra. A PRISM szimulációs felülete

PRISM fülének *Engine* opciójával történik, míg parancssorban a `-m`, `-s`, és `-h` kapcsolókkal, amelyek rendre az MTBDD, ritka mátrix és hibrid modulok betöltését végzik.

Modell típusok

A PRISM három különböző sztochasztikus modell leírására alkalmas:

- diszkrét-idejű Markov láncok(DTMCs,discrete-time Markov chains)
- Markov döntési folyamatok (MDPs, Markov decision processes)
- folyamatos-idejű Markov láncok(CTMCs,continous-time Markov chains).

A modellezés folyamán a PRISM először feldolgozza a modell leírását, amiből felépít egy belső reprezentációt. A modell megadására a PRISM nyelv használatos, de emellett a PEPA modellek egy részhalmazát is támogatja. Lehetőségünk van a feldolgozott modell exportálására több modellező nyelven is, amelyet használhat például az ETMCC és az MRMC. A modell megadását követi a lekérdezések feldolgozása, amelyből a megfelelő ellenőrzések elkészíthetőek. A lehetséges állapotok reprezentálásra több adatszerkezet kombinációját használja, mint az MTBDD (Multi-terminal binary decision diagram),ami a bináris döntési diagram(BDD) egy kiterjesztése,és konvencionális adatszerkezeteket mint a tömbök és a ritka mátrixok. Ezek az adatszerkezetek idő- és költséghatékonyak a sztochasztikus modellek elemzéséhez, amelyek gyakran nagy méretű állapottérrel rendelkeznek. Az állapotok elérhetőségének elemzésére, a modell temporális logikai vizsgálatához a PRISM gráfelméleti algoritmusokat használ. A számításokat három módszerrel végezheti a PRISM: Az MTBDD hasznos,ha nagy az állapottér, a ritka mátrix feldolgozó a kis állapotterű,de hosszú elemzési idejű modellekre optimalizált, míg a harmadik mód a hibrid reprezentáció, amely átlagos futási idővel és memória igénnyel dolgozik.

3.2. A PRISM nyelv

3.2.1. Modulok és változók

A PRISM nyelv egy magas szintű, állapot- és valószínűségalapú modellező nyelv, amely a *Reaktív Modulok* (Reactive Modules) formalizmusra épül. Egy PRISM nyelven specifikált modell egymással kapcsolatban álló modulokból épül fel. A modulok állapotát a bennük szereplő változók értéke határozza meg. Ezeket lokális változóknak hívjuk, értékük minden modulból olvasható, de csak abban módosítható, amelyikben szerepelnek. Lehetőség van globális változók megadására is, amelyeket bármelyik modul olvashat és írhat. A modell állapotát a lokális és globális változók együttes értéke határozza meg. Modul deklarációja bárhol előfordulhat, kivéve egy másik modul belsejében.
Modulokat a következő konstrukcióval hozhatunk létre:

```
module modul_név
...
endmodule
```

Lehetőségünk van modulok deklarálására átnevezéssel is abban az esetben, ha két modul teljesen megegyezik. Ilyenkor kötelezően át kell neveznünk az összes lokális változót, de megtehetjük ezt a címkékkel is:

```
module modul2 = modul1 [v1=v2,v2=v1,action1=action2] endmodule
```

Lokális változók deklarálása:

```
változó_név : [min..max] init konstans;
```

ahol az `min` és `max` a változó lehetséges értékeinek minimuma illetve maximuma. Az init kulcsszó opcionális, a kezdőállapot beállítására szolgál, a konstans pedig egy egész szám a $[min, max]$ intervallumból. Lehetőség van logikai változó deklarálásra is:

```
változó_név : bool init true;
```

alakban. A kezdőállapot megadása szintén opcionális.
Globális változók deklarálása a global kulcsszóval történik:

```
global változó_név : bool init true;
```

A nyelv lehetőséget biztosít konstansok használatára is. Konstansok lehetnek egész,valós vagy logikai típusúak, értéküket pedig literálokkal vagy konstans kifejezésekkel adhatjuk meg. Az értékadás nem kötelező, akár a szimulációk során is kaphatnak értéket. Deklarálásuk a const kulcsszóval lehetséges:

```
const int egesz = ertek;
const double valos = ertek;
```

Formulák és címkék

A forráskód egyszerűsítése céljából a nyelv bevezeti a formulák fogalmát, amelyek a kód duplikálását hivatottak elkerülni. Egy formula egy névből és egy kifejezésből áll, ahol a név egy azonosító lehet. Bárhol előfordulhatnak, ahol a kifejezések is. Deklarációjuk a `formula` kulcsszóval történik:

```
formula formula_név = kifejezés
```

Bár a címkék csak a lekérdezések specifikálásához használatosak, kényelmi szempontból a modell leírásában is szerepelhetnek. Segítségükkel állapothalmazokat tudunk azonosítani. A címkék abban különböznek a formuláktól, hogy egyrészt csak logikai értékűek lehetnek, másrészt nevüket kötelezően idézőjelek között írjuk:

```
label "címke_név" logikai_kifejezés
```

Kezdőállapotok

Mivel a modell állapotát a változók állapota határozza meg, így ha kezdőállapotot adunk meg a változóknak, a rendszer kezdőállapotát is beállítottuk. Ha nem adunk meg kezdőállapotot, a változó felveszi értelmezési tartománya minimumát. Ezekben az esetekben egyetlen kezdőállapottal rendelkezik a modell. Lehetséges azonban olyan modell leírása is, amelynek több kezdőállapota van. Ez az

```
init feltétel endinit
```

konstrukcióval lehetséges, amely bárhol előfordulhat a forrásban. A `feltétel` egy logikai feltételt jelöl, amelyben minden változónak szerepelnie kell.

A modulok és változók nevei szabványos azonosítók lehetnek, tartalmazhatnak kis- és nagybetűket,számokat és az _(aláhúzás) karaktert, de csak betűvel kezdődhetnek. Továbbá nem lehetnek azonosítók a következő, a PRISM által fenntartott kulcsszavak: **A, bool, const, ctmc, C, double, dtmc, E, endinit, endmodule, endrewards, endsystem, false, formula, func, F, global, G, init, I, int, label, max, mdp, min, module, X, nondeterministic, Pmax, Pmin, P, probabilistic, prob, rate, rewards, Rmax, Rmin, R, S, stochastic, system, true, U, W,**

3.2.2. Jutalmak és költségek

A jutalmak és költségek arra használatosak a modellezében, hogy segítségükkel nem csak a rendszer viselkedésébe nyerünk bepillantást, hanem számos mérhető, a viselkedéssel kapcsolatos mennyiséget tudunk meghatározni úgy, hogy állapotokhoz és állapotátmenetekhez valós értékeket rendelünk. Mivel nincs lényegbeli különbség a jutalmak és a költségek között, a PRISM csak a jutalmak megadását támogatja. Jutalmakat a `rewards` és `endrewards` kulcsszavak között adhatunk meg, legalább egyet de akár többet is. Jutalom deklarálása a következő formában történik:

```
feltétel : érték ;
```

illetve

```
[parancs_címke] feltétel : érték ;
```

Előbbi esetben állapotkhoz, utóbbiban átmenetekhez rendelünk jutalmat. A feltétel mindkét esetben egy állapothalmazt azonosít, az érték pedig az ezekhez az állapotokhoz rendelt jutalom mennyisége. Ha parancshoz társítunk jutalmat, a `parancs_címke` jelöli ki, hogy ez az adott állapotban mely állapotátmenetekhez tartozzék. A címke használata opcionális, az üres szögletes zárójelpár az összes tranzakcióhoz jutalmat társít. Például a

```
rewards
    x=0 : 100;
```

```
    x>0 & x<10 : 2*x;
    x=10 : 100;
endrewards
```

jutalmazási rendszer 100 jutalmat rendel minden olyan állapothoz, amelyben x értéke 0 vagy 10, és $2 \cdot x$ jutalmat azokhoz az állaptokhoz, amelyekben x nagyobb, mint 0 és kisebb, mint 10. A

```
rewards
    [] true : 1;
    [a] true : x;
    [b] true : 2*x;
endrewards
```

jutalomrendszer minden átmenethez 1 jutalmat rendel, ezen felül az `a` és `b` címkéjű átmenetekhez x illetve $2 \cdot x$ jutalmat társít. Mivel egy modellhez akár több jutalmazási mód is tartozhat, címkézhetjük is a blokkokat, amely a lekérdezések specifikálásánál bizonyul hasznosnak. A címkét idézőjelek között adhatjuk meg:

```
rewards "címke"
...
endrewards
```

Kifejezések és beépített függvények

Egy kifejezés a PRISM nyelven tartalmazhat literálokat, azonosítókat és az 1.táblázatban felsorolt operátorokat.

A precedencia a sorrendnek megfelelően fentről lefelé csökken, és az azonos sorban találhatóak precedenciája megegyezik. A számításokhoz használható beépített függvényeket a 2.táblázat tartalmazza.

Operátorok	
`-`	negatív előjel
`* , /`	szorzás, osztás
`+`	összeadás
`<, <= >=, >`	hasonlító operátorok
`=, !=`	egyenlő és nem egyenlő operátorok
`!`	negáció
`&`	konjukció
|	diszjunkció
`=>`	implikáció
`?`	feltételes operátor

1. táblázat. A PRISM nyelv operátorai

Beépített függvények	
`min(...), max(...)`	minimum,maximum függvények
`floor(x), ceil(x)`	fel- és lekerekítés egészre
`pow(x,y)`	hatványozás
`mod(i,n)`	maradékos osztás
`log(x,b)`	logaritmus függvény

2. táblázat. A PRISM nyelv beépített függvényei

3.2.3. Parancsok

A modulok viselkedését parancsokkal írhatjuk le, amelyek egy feltételből(guard) és egy vagy több átmenetből(update) állnak:

```
[parancs_címke] feltétel -> pr_1 : (u1) + ... + pr_n : (un);
```

ahol a feltétel logikai kifejezés, az úgynevezett átmenetek, az `u1,u2,...,un` kifejezések pedig állapotátmeneteket írnak le a változók értékeinek megváltoztatásával. DTMC és MDP esetében a `pr_1,pr_2,...,pr_n` mennyiségek valószínűségeket jelentenek, míg folytonos esetben paramétereket. A diszkrét-idejű modellek esetében a valószínűségek összegének kötelezően egynek kell lennie. A parancs azt határozza meg, milyen ál-

lapotátmenetek és milyen valószínűséggel menjenek végbe azokban az állapotokban, amelyekben a feltétel igaz. Például az

```
[] x=1 & y!=2 -> 0.3 : (x'=2) + 0.7 : (x'=y*x);
```

parancs olyan állapotokban lesz aktív, ahol az `x` változó egyenlő egyel és az `y` változó nem egyenlő kettővel. Ekkor az állapotátmenet következtében $0,3$ valószínűséggel `x` új értéke 2 lesz, $0,7$ valószínűséggel pedig $y \cdot x$. A parancs címkézése opcionális, jelentősége a szinkronizációban van. A modell összes állapotában a parancsok egy részhalmaza engedélyezett, tehát ahol a végrehajtási feltétele igaz. Ha egy állapotban több feltétele is igaz, a modell típusától függ, hogy melyik lesz aktív. Ha a modelltípus Markov döntési folyamat(MDP), a választás nemdeterminisztikus, míg diszkrét-idejű Markov láncok(DTMC) esetén egyenlő valószínűséggel aktiválódnak a parancsok.

Szinkronizáció

A legegyszerűbb szinkronizációs módszer,ha csak címkézzük a parancsainkat. Kettő vagy több megegyező címkéjű parancs átmenetei csak egyidőben mehetnek végbe, amelyek valószínűsége a különböző átmenetek valószínűségének szorzata. Ez azonban nem minden esetben igaz, így általánosan használt technika, hogy az egyik parancs paraméterének 1-nek adunk meg, ez lesz az úgynevezett passzív parancs, míg az aktív értékének a szinkronizált parancsok elvárt pramatérerét állítjuk be. Hogy ezt a modellt finomíthassuk, meghatározhatjuk azt is, mely modulok szinkronizáljanak egymással a parancsaikon keresztül. Ezt a `system` és `endsystem` kulcsszavak között tehetjük meg, a következő operátorok használatával:

- `M1 || M2`

  : a két modul közös parancsai alapján történik szinkronizáció,ez az alapértelmezett.

- `M1 ||| M2`

  : nincs szinkronizáció a modulok között.

- `M1 |[a,b, ....]| M2`

  : csak a közös `a,b,...` parancsok szinkroizálnak.

- `M / {a,b,....}`

  : az M modul `a,b,...` parancsainak elfedése.

- `M {a<-b,c<-d,...}`

  : az M modul parancsainak átnevezése.

Ebben a konstrukcióban minden modulnak pontosan egyszer kell szerepelnie. Az első két operátor asszociatív, és több modulra is alkalmazható. Az elfedés és átnevezés operátorai erősebben kötnek mint a kompozíciós operátorok, de a kifejezés zárójelezhető a kiértékelés sorrendiségének megváltoztatásához.

3.3. Lekérdezések specifikálása

Egy PRISM nyelven modellezett rendszer elemzéséhez definiálhatunk bizonyos lekérdezéseket, amelyeket a modell felépítése után a szoftver kiértékelhet. A PRISM lekérdező nyelve több jól ismert valószínűség-alapú temorális logika ötvözete illetve kiterjesztése, mint például a PCTL(Probabilistic Computation Tree Logic), amely a diszkrét idejű modelleknél használható, vagy a CSL(Continous Stochastic Logic), amely CTMC modelltípus esetén alkalmazott.

3.3.1. A P operátor

A **P** operátor alkalmazható mindhárom modelltípushoz megadott lekérdezésekhez. Segítségével meghatározhatjuk, hogy a modellezett rendszerben végbemennek e bizonyos események, és ha igen, milyen valósznűséggel. Egy **P** operátorral ellátott lekérdezés formája a következő:

```
P korlát [ útkifejezés ]
```

amely igaz értéket szolgáltat egy s állapotban, ha annak valószínűsége, hogy az s-ből induló utak kielégítik az *útkifejezés* formulát, megfelel a *korlát*nak. Például a

```
P<0.1  [ X z=2 ]
```

lekérdezés igaz egy állapotban, ha annak valószínűsége, hogy a következő állapotban `z` egyenlő lesz 2-vel,kisebb mint 0,1. Meg kell azonban említenünk, hogy a Markov döntési folyamatok nemdeterminisztikus volta következtében az ilyen típusú modelleknél az *útkifejezés* kielégíthetőségének minimális illetve maximális valószínűsége értelmezhető a vizsgálat során.

Útkifejezések

Az útkifejezések logikai formulák, amelyek segítségével a reprezentált modell bizonyos utait azonosíthatjuk. A kifejezések megadásához használhatóak a PCTL és CTL temporális logikákból ismeretes *temporális operátorok*. A **P** operátor útkifejezéseiben az alábbiakat alkalmazhatjuk:

Az X operátor: Az `[ X feltétel ]` alakú útkifejezés igaz egy útra, ha a rákövetkező állapotában igaz a `feltétel` formula.

Az U operátor: A `[ feltétel_1 U feltétel_2 ]` útkifejezés igaz egy útra, ha egy állapotában a `feltétel_2` formula igaz, és az ezt az állapotot megelőző összes állapotra igaz a `feltétel_1` feltétel.

Az F operátor: Az `[ F feltétel ]` útkifejezés igaz egy útra, ha van az útnak egy olyan állapota, melyre a `feltétel` feltétel igaz.

A G operátor: Amíg az **F** operátor az elréhetőséghez kapcsolódik, addig a **G** operátor az invarianciához köthető: A `[ G feltétel ]` útkifejezés igaz, ha a `feltétel` formula igaz az út összes állapotára.

A W operátor: Egy `[ feltétel_1 W feltétel_2 ]` útkifejezés igaz egy útra, ha az út minden állapotában igaz a `feltétel_1` feltétel, vagy ha valamely állapotában igaz a `feltétel_2` formula, akkor az azt megelőző összes állapotban igaz a `feltétel_1`. Ez egy származtatott operátor, mivel a `[ feltétel_1 W feltétel_2 ]` és a `[ (feltétel_1 U feltétel_2)` | `G feltétel_1 ]` formulák ekvivalensek.

Az R operátor: Az **R** operátor szintén egy, a fentiekből származtatott operátor, a `[ feltétel_1 R feltétel_2 ]` és a `[ !( !feltétel_1 U !feltétel_2) ]` útkifejezések ekvivalensek.

Az **X** operátor kivételével a fent említett operátoroknak létezik az idő függvényében kötött variánsa is. A diszkrét idejű modelleknél csak felső korlátot adhatunk az `operátor <= időkorlát` formában, ahol az `időkorlát` egész szám vagy egész értékű kifejezés lehet. Utóbbi esetben a kifejezés kötelezően kerek zárójelek között írandó. Folytonos idejű modelleknél mind a `<=`, mind a `>=` hasonlító operátorok megengedettek, vagy akár időintervallumot is megadhatunk az `operátor[alsó_korlát ,felső_korlát]` alakban. Az ilyen típusú modellnél a korlátok valós értéket is felvehetnek.Például a

```
P>=0.98  [ F<=7  y=4 ]
```

lekérdezés igaz egy s állapotra, ha annak valószínűsége, hogy az `y` változó értéke az s-ből kiinduló utakon 7 időegységen belül 4 lesz, nagyobb, mint $0,98$. Hasonlóan a

```
P>0 [ G[5,6] y>1 ]
```

lekérdezés igaz egy útra, ha annak valószínűsége, hogy az `y` változó értéke az $[5,6]$ időintervallumban meghaladja az 1-et, nagyobb, mint 0.

Valószínűségek számítása

A **P** operátor segítségével a modellezett rendszerben lehetséges események bekövetkezésének valósznűségét is meghatározhatjuk. Ebben az esetben a fent említett `korlát` helyett a `=?` használatos:

```
P=? [ útkifejezés ]
```

A Markov döntési folyamatok nemdeterminisztikus jellegéből adódóan azonban az ilyen modellekben lehetséges a valószínűségek minimum illetve maximum értékének számítása a következő módon:

```
Pmin=? [ útkifejezés ]
Pmax=? [ útkifejezés ]
```

A fenti lekérdezések mindegyike egyetlen numerikus adatot szolgáltat, és alapértelmezetten a kezdőállapotból induló utakra vonatkozik. Lehetőségünk van viszont megadni egy kitüntetett állapotot a kezdőállapot helyett úgynevezett szűrő segítségével:

```
P=? [ útkifejezés {szűrő} ]
```

Természetesen a `szűrő` feltétel több állapotban is igaz lehet, ebben az esetben a PRISM a lexikografikus sorrendben legelső állapotot választja. Ilyenkor a rendszer figyelmeztető üzenetet küld, és megjeleníti az említett állapotot. Viszont ha egyetlen állapot sem elégíti ki a szűrőfeltételt, a lekérdezés hibaüzenettel zárul. Létezik ellenben olyan megoldás is, amely több, a szűrőfeltételnek megfelelő állapot esetén a valószínűségek minimum és maximum értékét szolgáltatja:

```
P=? [ útkifejezés {szűrő}{min} ]
P=? [ útkifejezés {szűrő}{max} ]
```

Például a

```
P=? [ F x=5&y=5 {y=2}{min} ]
P=? [ F x=5&y=5 {y=2}{max} ]
```

lekérdezések rendre a legkisebb és legnagyobb valószínűségét szolgáltatják annak az eseménynek, hogy az `x` és `y` változó értéke is 5 lesz az összes olyan állapotból kiindulva, amelyben `y` értéke 2. Ezenfelül a PRISM azt is jelzi, mely állapotokból érhetőek el a ezek a minimum illetve maximum valószínűségek.

3.3.2. Az S operátor

Az **S** operátor az egyensúlyi állapot-valószínűségek számítására alkalmas. Az

```
S korlát [ feltétel ]
```

lekérdezés igaz értékű egy s állapotban, ha ebből az állapotból indulva a `feltétel` formulát kielégítő állapotok egyensúlyi állapot-valószínűsége megfelel a `korlát` korlátnak. A **P** operátorhoz hasonlóan az **S** operátornak is használható olyan formája, mellyel ezt a valószínűséget számíthatjuk. Ennek általános alakja a következő:

```
S=? [feltétel]
```

Az előzőekhez hasonlóan a szűrőket is írhatunk a lekérdezéshez.

3.3.3. Az R operátor

Az **R** operátort akkor alkalmazzuk a lekérdezésekben, ha a modellben megadtunk jutalmazási rendszert. Ilyenkor az operátor e jutalmak várható értékének számítására használható. Az előbbiekhez hasonlóan az **R** operátornak is van korlátos és lekérdező formája is:

```
R korlát [ jutalom_feltétel ]
R=? [ jutalom_feltétel ]
```

Az első esetben a korlát tartalmazhatja a `<, <=, >, >=` operátorokat. A Markov döntési folyamatok esetében az előzőekhez hasonlóan a nemdeterminisztikus viselkedésből adódóan a várható értékek minimuma illetve maximuma számítható:

```
Rmin=? [ jutalom_feltétel ]
Rmax=? [ jutalom_feltétel ]
```

A jutalmatási feltételek az útkifejezésekhez hasonló formulák, megadásukhoz az alábbi temporális operátorokat használhatjuk:

Az F operátor: Egy elérhető állapothoz rendelt jutalom mennyiségéhez kapcsolható. Egy `R korlát [ F feltétel ]` kifejezés igaz egy s állapotban, ha egy a `feltétel` feltételnek megfelelő állapotba történő eljutás jutalma kielégíti a `korlát` feltételt.

A C operátor: A **C** operátor a költségek időben korlátozott mennyiségének várható értékére ad feltételt. Ezzel az operátorral ellátott **R** operátornak csak a lekérdező formája használható. Egy `R=? [ C<= időkorlát ]` alakú lekérdezés az `időkorlát` idő alatt lejátszódó állapotátmenetek várható jutalmát szolgáltatja. A korábbiakhoz hasonlóan a diszkrét idejű modelleknél a korlát egész szám illetve egész értékű kifejezés lehet, míg folytonos esetben valós szám illetve kifejezés.

Az I operátor: Az **I** operátor segítségével egy adott időpillanatban a jutalom várható értékét számíthatjuk, illetve adhatunk rá feltételt. Az `R feltétel [ I=időkorlát ]` alakú lekérdezés igaz értéket szolgáltat egy s állapotban, ha pontosan `időkorlát` idő elteltével a jutalom várható értéke kielégíti a `feltétel` feltételt. Az `R=? [ I=t ]` feltétel a i-edik időpillanatban várható jutalom értékének számítását végzi.

Az S operátor: Az előzőekkel ellentétben az **S** operátor nem bizonyos utak jutalmainak várható értékéhez kapcsolható. Az `R feltétel [ S ]` lekérdezés igaz értéket szolgáltat egy s állapotban, ha az s-ből indulva a jutalmak várható értékének határértéke kielégíti a `feltétel` feltételt.

Ha egy modellhez több jutalmazási rendszert is specifikáltunk, akkor a lekérdezésekben meg kell adnunk, melyikre hivatkozunk. Ezt úgy tehetjük meg, hogy kapcsos zárójelek között megadjuk a utalmazási rendszer címkéjét, vagy a modell forrásában elfoglalt sorrendjét:

```
R{"címke"} korlát [feltétel]
R{"2"}=? [feltétel]
```

4. A rendszer helyessége

4.1. Analítikus eredmények

Az alábbiakban egy egyszerű M/M/1/K típusú klasszikus sorbanállási rendszer a PRISM nyelven elkészített, és a szoftverrel elemzett modelljét vetjük össze az alapvető sorbanállási elméletek eredményeivel. Ilyen rendszerek esetében egyetlen kiszolgáló csatorna van, az igények Poisson-folyamat szerint érkeznek, azonban csak azok az igények léphetnek be a rendszerbe, melyek érkezésekor kevesebb, mint K igény tartózkodik a sorban. Ekkor az alábbi érkezési intenzitásokat kapjuk:

$$\lambda_k = \begin{cases} \lambda & \text{,ha } k < K, \\ 0 & \text{,ha } k \geq K \ , \end{cases}$$

$$\mu_k = \mu, \qquad k = 1, 2, ..., K.$$

A forgalmi intenzitás ekkor:

$$\varrho = \frac{\lambda}{\mu}$$

Annak valószínűsége, hogy a rendszerben pontosan k igény tartózkodik:

$$P_k = P_0 \prod_{j=0}^{k-1} \frac{\lambda}{\mu} = P_0 \left(\frac{\lambda}{\mu}\right)^k, \qquad \text{ha } k \leq K.$$

$$P_0 = \frac{1 - \lambda/\mu}{1 - (\lambda/\mu)^{K+1}}$$

A rendszerben tartózkodó igények átlagos száma, vagyis az átlagos sorhossz:

$$\overline{L} = \begin{cases} \dfrac{\varrho[1 - (K+1)\varrho^K + K\varrho^{K+1}]}{(1-\varrho)(1-\varrho^{K+1})} & , ha \lambda \neq \mu \ , \\ \dfrac{K}{2} & , ha \lambda = \mu. \end{cases}$$

Az átlagos várakozási idő, feltéve hogy a kiszolgálás FIFO(First In First Out) rendszerben történik:

$$\overline{W} = \overline{L}\frac{1}{\mu}.$$

4.2. A rendszer PRISM modellje

A modell mindössze egy modulból áll. A sor maximális hosszát a `K` konstans határozza meg, míg az érkezési és kiszolgálási intenzitásokat rendre `lambda` és `mu` valós konstansok tárolják. Ezek értékeit nem definiáltuk, és csak kísérletek folyamán adjuk meg. A sorban várakozó igények számát az `n` változó reprezentálja. Két különböző jutalom rendszert adtunk meg, amelyek az átlagos sorhossz és az átlagos várakozási idő számításában segítenek.

```
ctmc
const int N;
const double lambda;
const double mu;
const int k;

module MM1K_sor
   n : [0..N] init 0;
   [erkezes]     (n<N) -> lambda : (n'=n+1);
   [kiszolgalas] (n>0) -> mu : (n'=n-1);
endmodule

rewards "atlagos_sorhossz"
   true : n;
endrewards

rewards "atlagos_varakozas"
   true: n/mu;
endrewards
```

A fenti eredmények igazolására a rendszer lekérdezéseit a következőképpen definiáljuk:

```
S=? [n=k]
//Annak valószínűsége, hogy k igény van a rendszerben
R{"atlagos_sorhossz"}=? [S]
```

```
//Az átlagos sorhossz várható értéke
R{"atlagos_varakozas"}=? [S]
//Az átlagos várakozási idő várható értéke
```

4.3. Az eredmények összehasonlítása

A fenti modellhez kapcsolódó lekérdezéseket különböző maximális sorhosszok és forgalmi intezitások mellett is kiértékeltük. Ezekből a kísérletekből jól látszik, hogy a matematikai számítások és a szoftver által szolgáltatott eredmények megegyeznek, illetve különbségük elhanyagolható. A számításokat a PRISM hibrid numerikus moduljával végeztük.

Elsőként azt vizsgáljuk, mekkora a valószínűsége annak, hogy egy időpillanatban a rendszerben k darab igény várakozik. A kísérletet két különböző forgalmi intenzitással is elvégeztük, de a sor maximális hossza mindkét esetben 5 volt. Az alábbi táblázat tartalmazza a kézzel számolt eredményt és a PRISM által szolgáltatott értékeket is. Jól látható, hogy a végeredmények teljesen identikusak.

k	Analítikus eredmények		PRISM eredmények	
	$\varrho = 0.72$	$\varrho = 0.83$	$\varrho = 0.72$	$\varrho = 0.83$
0	0,285745963	0,192586495	0,285745963	0,192586495
1	0,206372084	0,160488746	0,206372084	0,160488746
2	0,149046505	0,133740622	0,149046505	0,133740622
3	0,107644698	0,111450518	0,107644698	0,111450518
4	0,077743393	0,092875432	0,077743393	0,092875432
5	0,056148006	0,077396193	0,056148006	0,077396193

3. táblázat. Annak valószínűsége, hogy k igény várakozik a rendszerben

A következő kísérletben a rendszerben tartózkodó igények átlagos számát vetettük össze az elemi sorbanállási elmélet eredményeivel. A 4.táblázatban az átlagos sorhosszok láthatóak a forgalmi intenzitás függvényében.

ϱ	Analítikus eredmények	PRISM eredmények
0.78	2.780509	2.780510
0.8	2.966314	2.966316
0.82	3.158292	3.158294
0.84	3.355592	3.355594
0.86	3.557254	3.557256
0.88	3.762236	3.762239

4. táblázat. Átlagos sorhossz várható értéke rögzített, N=10 maximum sorhossz esetében

Az utolsó kísérletben a várható várakozási idő értékét értékeltük ki állandó,$\varrho = 0.77$ forgalmi inenzitás és növekvő kiszolgálási intenzitás mellett. Az eredmények az 5.táblázatban láthatóak.

μ	Analítikus eredmények	PRISM eredmények
0.7	3,94325399	3,94325399
0.72	3,83371915	3,83371915
0.74	3,73010512	3,73010512
0.76	3,63194446	3,63194446
0.78	3,53881768	3,53881768
0.8	3,45034724	3,45034724

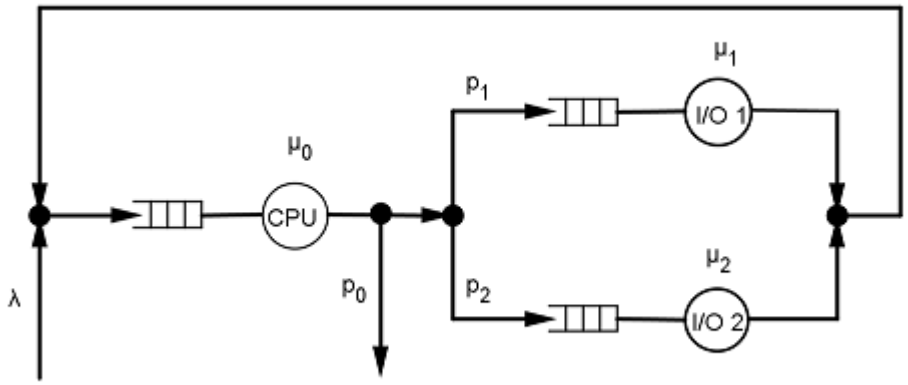
5. táblázat. Átlagos várakozási idő $\varrho = 0.77$ forgalmi intenzitás mellett

Amint azt a fenti szimulációkból láthatjuk, a PRISM számításai igazolják az analítikus módszerek eredményeit. Továbbá számos esettanulmány[1] készült komplexebb rendszerek modellezésével, amelyek bemutatják, hogy az eszköz helyesen működik nagyméretű állapottér és bonyolult viselkedés mellett is.

[1]Lsd. http://www.prismmodelchecker.org/casestudies/index.php

5. Egy összetett rendszer ellenőrzése

Ebben a fejezetben egy nem triviális sorbanállási rendszert modellezve szemléltetjük a szoftver lehetőségeit és hatékonyságát. Látni fogjuk, hogy ilyen összetettebb példa esetében is egyszerű a modell és a lekérdezések megadása, és ami még lényegesebb, az ellenőrzés gyors, és jól közelíti az analítikus eredményeket. A Rendszer grafikus reprezentációját a 3.ábrán láthatjuk.



3. ábra. Központi szerveres nyílt sorbanállási rendszer

A programok λ intenzitással lépnek be a rendszerbe, és ha még van szabad hely a sorban, várakoznak processzoridőre. Ha a sor elérte maximális kapacitását, az ekkor érkező igények elvesznek. Amint az ábrán is látható, a processzor μ_0 átlagos kiszolgálási idővel dolgozik, és ha végzett egy programmal, az p_0 valószínűséggel terminál. Ellenkező esetben valamelyik I/O sorba áll, rendre p_1 és p_2 valószínűséggel, feltéve hogy a sorban van még szabad hely. Az Input/Output eszközök átlagos kiszolgálási ideje μ_1 illetve μ_2. Ha a ki- vagy bemenet lejátszódott, a program visszakerül a processzorra várakozók sorába, ismét csak akkor, ha a sor nincs még tele. Vizsgálatunk tárgyát legfőképpen a folyamat időtartama, vagyis az átlagos kiszolgálási idő képezi, de kitérünk arra is, hogy bizonyos intenzitások és kihasználtsági ráták mellett mekkora lesz a sorok várható hossza. A számításokat hybrid adatreprezentáció és JOR iteratív metódussal végeztük 10^{-7} terminálási pontossággal, maximum 100000 iterációig. A lekérdezések ideje még 30

sorhossz esetében sem haladta meg a 15 percet.

5.1. Elméleti háttér

Ha megfigyeljük egy program futását, akkor megkaphatjuk, várhatóan hányszor kerül a rendszer egy bizonyos csomópontjába, a processzorhoz vagy az I/O eszközökhöz. Ekkor V_j, feltéve hogy $p_0 \neq 0$:

$$V_j = \begin{cases} \dfrac{1}{p_0}, \text{ ha } j = 0 \\ \dfrac{p_1}{p_0}, \text{ ha } j = 1 \\ \dfrac{p_2}{p_0}, \text{ ha } j = 2. \end{cases}$$

Vagyis egy program a j-edik csomópontot várhatóan V_j-szer látogatja meg. Mivel pedig egységnyi időközönként λ program lép be a rendszerbe, az érkezési intenzitás a j-edik csomópontban:

$$\lambda_j = \begin{cases} \dfrac{1}{p_0}\lambda, \text{ ha } j = 0 \\ \dfrac{p_1}{p_0}\lambda, \text{ ha } j = 1 \\ \dfrac{p_2}{p_0}\lambda, \text{ ha } j = 2. \end{cases}$$

A csomópontok kihasználtsága $\varrho_j = \lambda_j \mu_j = \lambda V_j \mu_j$, és feltesszük, hogy $\varrho_j < 1$, $j = 1, 2, 3$. Így a várható sorhosszok($\overline{L}_j$) és kiszolgálási idők($\overline{W}_j$) a következőképpen adódnak:

$$\overline{L}_j = \frac{\varrho_j}{1 - \varrho_j} \quad \text{,illetve} \quad \overline{W}_j = \frac{1}{\lambda} \frac{\varrho_j}{1 - \varrho_j}.$$

Ebből adódóan a rendszerben tartózkodó igények számának várható értéke és az átlagos kiszolgálási idő:

$$\overline{L} = \frac{\varrho_0}{1 - \varrho_0} + \frac{\varrho_1}{1 - \varrho_1} + \frac{\varrho_2}{1 - \varrho_2}$$

és

$$\overline{W} = \frac{\overline{B}_0}{1 - \lambda \overline{B}_0} + \frac{\overline{B}_1}{1 - \lambda \overline{B}_1} + \frac{\overline{B}_2}{1 - \lambda \overline{B}_2},$$

ahol $\overline{B}_j = V_j \mu_j$ a kiszolgálási idők összege a j-edik csomópontban.

5.2. A PRISM reprezentáció

A modellezéshez első lépésként modulok kompozíciójává absztraháljuk a rendszert, így egy modult fog alkotni a processzor és kettőt a két I/O csatorna. Az utasítások címkézésével valósítjuk meg közöttük a szinkronizációt. Az érkezési intezitást, a kiszolgálási időket és a sorhosszokat konstansokkal adjuk meg, amelyeket nem inicializálunk, csak a kísérletek folyamán. Az első modult, vagyis a processzort szimuláló modult a következőképpen konstruáltuk meg:

```
module cpu
   queue : [0..N] init 0;
   accepted  : bool init false;

   [arrive] (queue<N) -> lambda : (queue'=queue+1) & (accepted' = true) ;
   [arrive] (queue=N) -> lambda : (accepted' = false);

   [repeat1] (queue<N) -> 1 : (queue'=queue+1) & (accepted' = true);
   [repeat1] (queue=N) -> 1 : (accepted' = false);

   [repeat2] (queue<N) -> 1 : (queue'=queue+1) & (accepted' = true);
   [repeat2] (queue=N) -> 1 : (accepted' = false);

   [terminate] (queue>0) -> ((1-p1-p2)/muProc) : (queue' = queue-1);
   [serve1] (queue>0) -> (p1/muProc) : (queue' = queue-1);
   [serve2] (queue>0) -> (p2/muProc) : (queue' = queue-1);

endmodule
```

A `queue` változó tárolja a sor pillanatnyi hosszát, az `accepted` logikai változó pedig egy *flag*, amely azt jelöli, hogy az utolsó igény elveszett-e vagy sem. A két `arrive` címkéjű parancs a λ intenzitással érkező programokat generálja, első esetben ha van hely a sorban, utóbbiban maximális sorhossznál. Ekkor vagy nő a sorhossz egyel és igazzá válik a feltétel, vagy jelezzük, hogy nem léphetett be az program a rendszerbe. A `repeat1` és `repeat2` címkéjű parancsok a két I/O modullal szinkronizálnak, és logikai felbontásuk

analóg az `arrive` parancséval. Mivel a szinkronizált parancsok intenzitása a parancsok intenzitásának szorzata, így ebben a modulban 1 rátával adtuk meg meg őket, tehát ezek a passzívak, míg az aktívak az I/O modulokban foglalnak helyet. A `terminate`, `serve1` és `serve2` parancsok pedig a CPU kiszolgálását szimulálják, intenzitásuk összege $\frac{1}{muProc}$, és utóbbi kettő a két I/O modullal szinkronizál. A két Input/Output eszközt modellező modult az alábbi formában írtuk le:

```
module IO1
   IO1 : [0..Nio] init 0;
   accepted1 : bool init false;
   [serve1] (IO1 < Nio) -> 1:(IO1'=IO1+1) & (accepted1' = true);
   [serve1] (IO1 = Nio) -> 1:(accepted1'=false);
   [repeat1] (IO1 > 0) -> 1/mu1 : (IO1'=IO1-1);
endmodule

module IO2 = IO1
[IO1=IO2,accepted1=accepted2,repeat1=repeat2,serve1=serve2,mu1=mu2]
endmodule
```

Láthatjuk, hogy mivel a két eszköz valós viselkedése megegyezik és csak paramétereikben térnek el, a második modult az első átnevezésével hoztuk létre. A processzort leíró modulhoz hasonlóan az `IO1` és `IO2` a sorok aktuális hosszát tárolja, az `accepted1` és `accepted2` logikai változók pedig jelzik, hogy az utolsóként érkezett igény elveszett-e. Ebben a két modulban értelemszerűen a `serve1` és `serve2` parancsok passzívak, tehát intenzitásuk 1, míg a `repeat1` és a `repeat2` aktívak, rátájuk $\frac{1}{mu1}$ és $\frac{1}{mu2}$.

Annak érdekében, hogy lekérdezhessük a rendszerben tartózkodó összes igény számának várható értékét és az átlagos kiszolgálási időt, a következő jutalmazási konstrukciókat alkalmazzuk:

```
rewards "jobs"
 true : (queue + IO1 + IO2);
endrewards
```

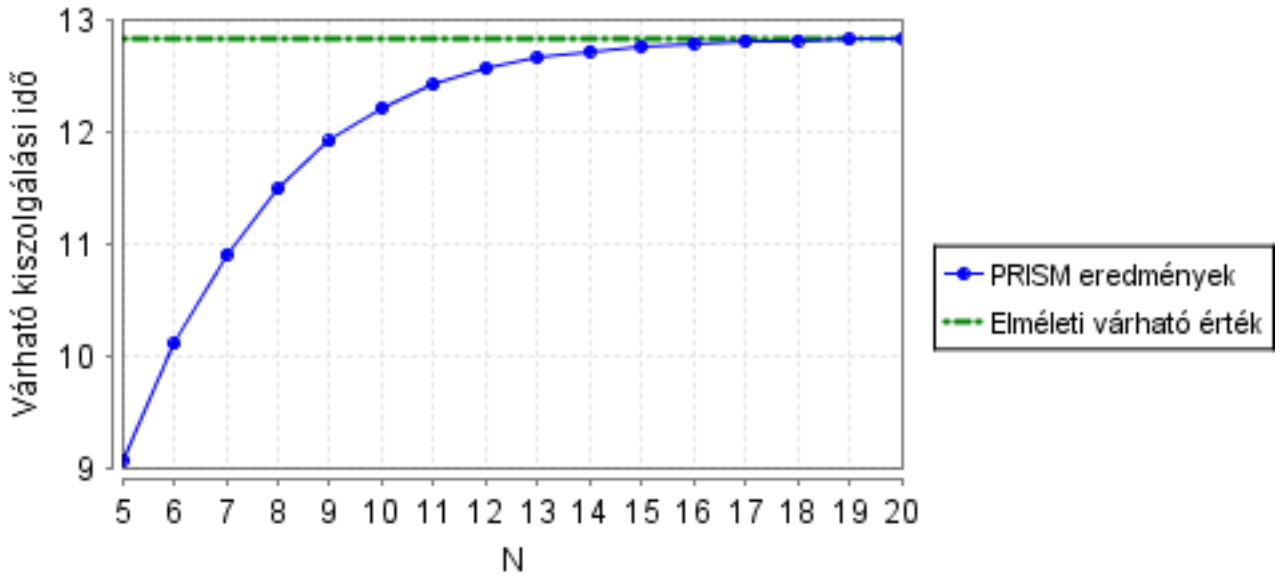
```
rewards "time"
  true : (queue + IO1 + IO2)/lambda;
endrewards

rewards "lost"
  !accepted : 1;
  !accepted1 : 1;
  !accepted2 : 1;
endrewards
```

Az első, `jobs` címkéjű jutalmazási rendszer minden állapothoz annyi jutalmat rendel, amennyi a három sor pillanatnyi hosszának összege, míg a második a *Little formula* alapján a várakozó igények és az érkezési intenzitás hányadosaként szolgáltatja a várható kiszolgálási időt. Az utolsó, `lost` címkéjű jutalmazási mód 1 értéket rendel minden olyan állapothoz, melyben egy igény elveszlett valamelyik sorban. Ennek segítségével lekérdezhetjük azoknak a programoknak az arányát, amelyek akkor érkeztek egy sorhoz, amikor az éppen elérte maximális kapacitását.

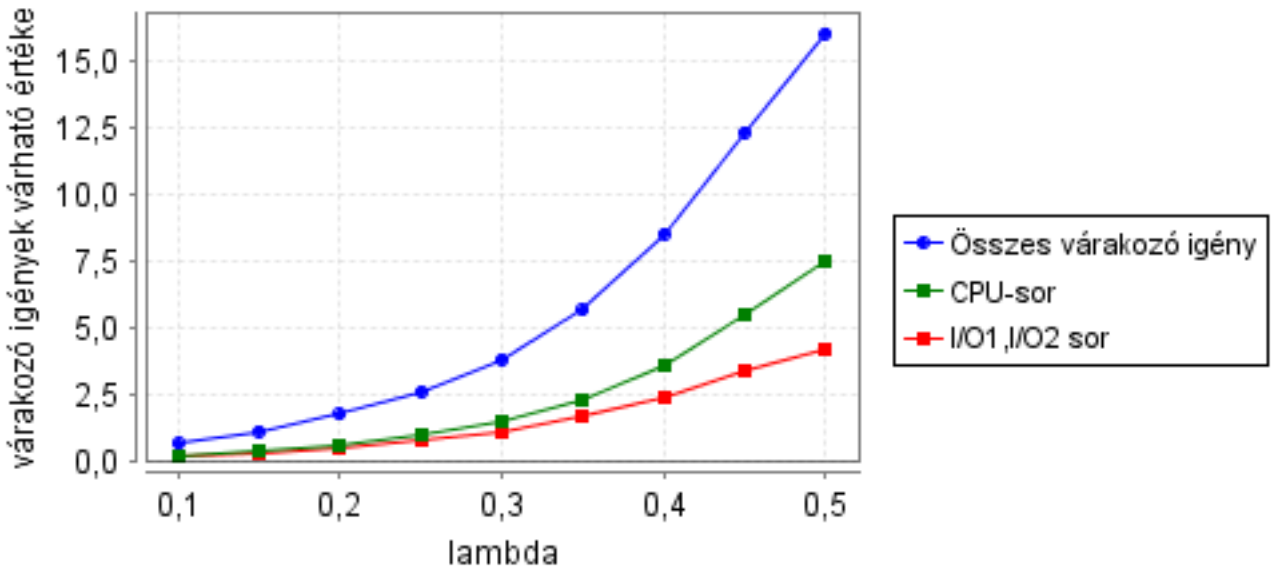
Elemzési eredmények

Vizsgálatunk fő tárgya a várható teljes kiszolgálási idő hossza, vagyis amennyi idő alatt egy program terminál. Mivel azonban az elméleti összefüggések végtelen sorhosszokat feltételeznek, és a PRISM csak véges sorhosszokat értelmez, a közelítő eljárások valamelyest eltérnek az analítikus értékektől. Így ha a sorok maximális hosszát növeljük, a PRISM által számított érték egyre jobban konvergál a várt értékhez. Ezt a konvergenciát láthatjuk a PRISM grafikus eszközével készített 4.ábrán, konstans $\varrho = 0.6$ kihasználtsági arány mellett. Azonban ha a kihasználtság növekszik, a számítási hiba nő, ezért szimulációinkban viszonylag hosszú, legalább $N = 20$ sorhosszokat adunk meg.



4. ábra. A számítás pontossága növekvő sorhossz esetén ($\varrho = 0.6$)

Az első szemléltetett szimulációban az érkezési intenzitás növekedése és a rendszerben tartózkodó összes igény számának várható értéke közötti kapcsolatot vizsgáljuk. Nyilvánvalóan ha az igények gyakrabban érkeznek, a kiszolgálási idők viszont nem csökkennek, a sorok hossza megnő, egészen addig, amíg a processzor kihasználtsága el nem éri az 1-et, ezen felül viszont a programok nem tudnak belépni a rendszerbe. Az 5.ábrán a várakozó igények számának várható értékét láthatjuk konstans $\mu_0 = 0,2$, $\mu_1 = 0,3$, $\mu_2 = 0,6$, $p_1 = 0,6$ és $p_2 = 0.3$ paraméterek mellett. Láthatjuk, hogy a két I/O csatornánál várakozó programok átlagos száma megegyezik. Ez annak köszönhető, hogy az elsőhöz



5. ábra. A rendszerben tartózkodó igények számának várható értéke N=20 sorhossz esetén

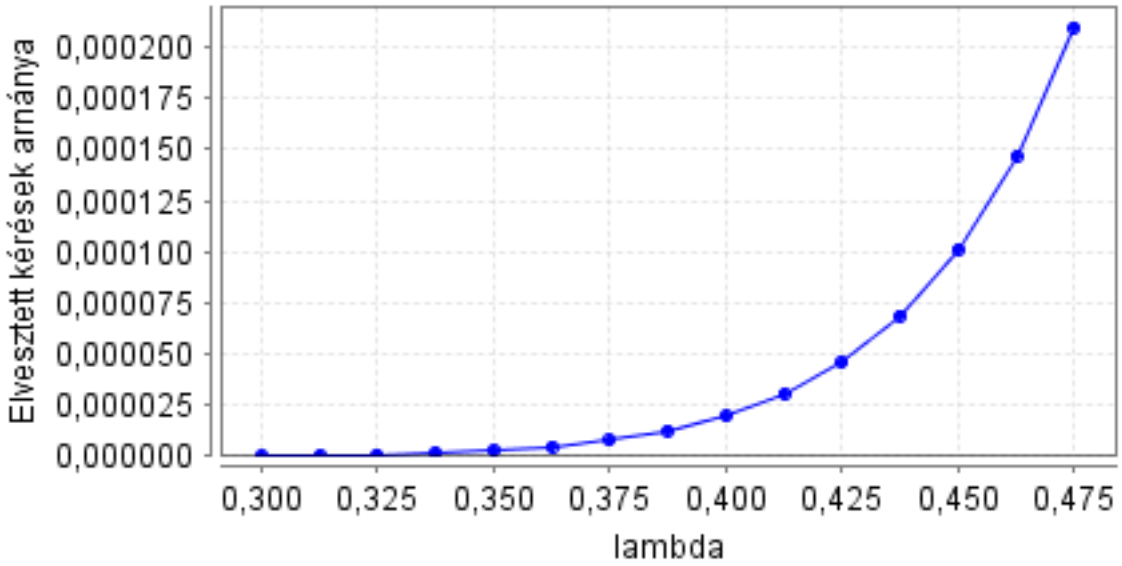
történő érkezés valószínűsége kétszerese a másodikhoz érkezésnek, viszont kiszolgálási ideje is csak a fele, mint utóbbi esetében.

Most tekintsük a rendszer egy olyan realizációját, ahol $0,1$ annak valószínűsége, hogy egy program a processzor használata után terminál, függetlenül a korábbi tevékenységétől, vagyis $p_0 = 0,1$. Továbbá $p_1 = 0,6$ és $p_2 = 0,3$, vagyis kétszer akkora valószínűséggel kerül a program az első I/O csatornához, mint a másodikhoz. Alkalmas kiszolgálási idők és növekvő érkezési intenzitású igények megadásával jól szemléltethetjük, mennyire rohamosan nő a kiszolgálási idő az intenzitás függvényében, ha a processzor kihasználtsága magas. Az analítikus és PRISM által számított értékeket a 6.táblázatban találjuk. A kísérletben a csomópontok átlagos kiszolgálási idejének $\mu_0 = 0,15$, $\mu_1 = 0,3$, $\mu_2 = 0,6$ értékeket adtunk meg, a maximális sorhossz pedig 30 volt.

λ	Analítikus eredmények	PRISM eredmények
0,3	10,55335968	10,55327938
0,325	11,60152806	11,60119949
0,35	12,88762447	12,88743754
0,375	14,50549451	14,50538639
0,4	16,60714286	16,60709231
0,425	19,45707997	19,45697619
0,45	23,56275304	23,56226941
0,475	30,04497751	30,04364432

6. táblázat. Átlagos várakozási idő növekvő érkezési intenzitás és N=30 sorhossz mellett

Mivel ebben az esetben a csomópontok kihasználtsága nő, és értelemszerűen a sorhosszok is, az elveszített igények várható aránya megnő, amely a 6.táblázatban látható különbségeket okozza az elméleti számítások és a PRISM eredmények között. A 6.ábrán az intenzitás növekedése és elvesztett igények aránya közötti kapcsolatot láthatjuk.



6. ábra. Elvesztett igények aránya az érkezési intenzitás függvényében

6. Összegzés

E dolgozat kitűzött célja a valószínűségi modellellenőrzés technikájának és módszerének ismertetése, illetve az erre használatos PRISM eszköz használatának és lehetőségeinek bemutatása volt. A legfőbb konklúzió mindenképpen az, hogy a szoftver méltán a legelterjedtebb és legszélesebb körben használt eszköz a sztochasztikus jellegű rendszerek kvalitatív és kvantitatív elemzésére. Az eredmények tükrében bizton állíthatjuk, a modellek ellenőrzése gyors és akkurátus, és a modellezhető rendszerek skálája széles. Emellett bepillantást nyertünk a rendszer funkcióiba, a kísérletek, lekérdezések és szimulációk segítségével. Példát mutattunk egy egyszerű és egy összetettebb rendszer modelljére, amelyek eredményeit a PRISM-el ismerkedő felhasználó beépíthet és felhasználhat a való életben számtalan helyen előforduló, valószínűségen alapuló rendszerek viselkedésének modellezése folyamán.

Végezetül, de nem utolsó sorban szeretnék köszönetet nyilvánítani témavezetőmnek, Bérczes Tamásnak, munkám során nyújtott segítségéért és iránymutatásáért.

Irodalomjegyzék

[1] Pataricza András,Formális módszerek az informatikában, Typotex Kiadó, 2006.

[2] Dr. Sztrik János, Bevezetés a sorbanállási elméletbe és alkamazásaiba, Debreceni Egyetem Kossuth Egyetemi Kiadója, 2000.

[3] Kishor S. Trivedi,Probability and Statistics with Reliability, Queueing and Computer Science Applications, John Wiley and Sons, New York, 2001

[4] Christel Baier, Joost-Pieter Katoen, Principles of Model Checking, MIT Press, 2008.

[5] Ward Whitt, *Continuity of generalized semi-markov processes.*, Mathematics of Operations Research, **5**(1980) 494–501

[6] http://www.tempastic.org/ymer/

[7] http://osl.cs.uiuc.edu/~ksen/vesta2/

[8] http://www.prismmodelchecker.org/

[9] http://www.mrmc-tool.org/trac/