Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matej Dolenc

Verifikacija komunikacijskih protokolov na osnovi barvnih Petrijevih mrež

DIPLOMSKO DELO
UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

prof. dr. Miha Mraz
MENTOR

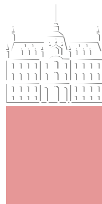
Ljubljana, 2015

© 2015, Univerza v Ljubljani, Fakulteta za računalništvo in informatiko

Rezultati diplomskega dela so intelektualna lastnina Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje Fakultete za računalništvo in informatiko ter mentorja.

Univerza
v Ljubljani

Fakulteta *za računalništvo*
in informatiko



Tematika naloge:

Kandidat naj v svojem delu predstavi problematiko verifikacije komunikacijskih protokolov, metode pristopa k verifikaciji in napravi pregled ustreznih programskih orodij. V nadaljevanju dela naj kandidat na osnovi barvnih Petrijevih mrež postavi model in opravi verifikacijsko analizo vzorčnega protokola vedra z žetoni. V okviru slednje naj se usmeri predvsem na analizo prostora stanj modeliranega sistema.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani izjavljam, da sem avtor dela, da slednje ne vsebuje materiala, ki bi ga kdorkoli predhodno že objavil ali oddal v obravnavo za pridobitev naziva na univerzi ali drugem visokošolskem zavodu, razen v primerih kjer so navedeni viri.
S svojim podpisom zagotavljam, da:

- sem delo izdelal samostojno pod mentorstvom prof. dr. Mihe Mraza,
- so elektronska oblika dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko in
- soglašam z javno objavo elektronske oblike dela v zbirki “Dela FRI”.

— Matej Dolenc, Ljubljana, avgust 2015.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matej Dolenc

Verifikacija komunikacijskih protokolov na osnovi barvnih Petrijevih mrež

POVZETEK

V diplomskem delu prikažemo različne metode verificiranja komunikacijskih protokolov. Vsaka izmed metod je na kratko predstavljena, dodatno pa predstavimo še nekaj programskih orodij, ki omenjene metode podpirajo.

V nadaljevanju se predvsem osredotočimo na metodo *barvnih Petrijevih mrež*, ki je grafično orientiran jezik za specifikacijo, simulacijo in verifikacijo komunikacijskih protokolov S pomočjo barvnih Petrijevih mrež lahko poljuben protokol predstavimo kot mrežo, ki je sestavljena iz pogojev in akcij, le-te pa so med seboj povezane z usmerjenimi povezavami. Sledi tudi pregled nekaterih programskih orodij za delo z barvnimi Petrijevimi mrežami, še posebej pa vzamemo pod drobnogled programsko orodje *CPN Tools*.

CPN Tools uporabimo za postavitev modela vedra z žetoni, ki predstavlja protokol, ki se uporablja pri oblikovanju prometa po omrežju. Ko imamo model postavljen, nad njim opravimo postopek verifikacije, v katerem preverimo nekaj najbolj pomembnih lastnosti protokola. Na podlagi pridobljenih rezultatov lahko nato model protokola popravimo in dodatno nadgradimo.

Ključne besede: verifikacija, barvne Petrijeve mreže, CPN Tools

University of Ljubljana
Faculty of Computer and Information Science

Matej Dolenc

Verification of communication protocols with coloured Petri nets

ABSTRACT

In this thesis we point out different methods for verification of communication protocols. Each method is presented in short. In addition, several tools which support mentioned methods are also presented.

Next, we primarily focus on a method called *coloured Petri nets* which is graphically oriented language for specification, simulation and verification of communication protocols. With the help of coloured Petri nets we can present a protocol as a net which consists of places and transitions. Places and transitions are then connected with directional arcs. Then, we take a look at a few tools for work with coloured Petri nets. We mainly focus on a tools called *CPN Tools*.

CPN Tools is used for constructing a model of a token bucket which is a traffic shaping protocol. When the model is constructed we can verify it. During verification we check some of the most important properties of a model. On the basis of results we get, we can then upgrade and fix the erros in the model.

Key words: verification, coloured Petri nets, CPN Tools

ZAHVALA

Zahvaljujem se mentorju prof. dr. Mihi Mrazu za nasvete in vodenje pri izdelavi diplomskega dela.

Zahvala gre tudi vsem mojim najbližjim za dolgoletno podporo v času študija.

— Matej Dolenc, Ljubljana, avgust 2015.

KAZALO

Povzetek i

Abstract iii

Zahvala v

1 Uvod 1

1.1 Motivacija . . . 1

1.2 Pregled diplomskega dela . . . 2

2 Verifikacija komunikacijskih protokolov 3

2.1 Definicija protokola . . . 3

2.2 Pomen verifikacije protokola . . . 4

2.3 Metode verificiranja komunikacijskih protokolov . . . 5

2.3.1 Algoritmični pristop . . . 6

2.3.2 Procesna algebra . . . 6

2.3.3 CFSM model . . . 7

2.3.4 ECFSM model . . . 7

2.3.5 Estelle metoda . . . 8

2.3.6 Petrijeve mreže . . . 8

2.3.7 Barvne Petrijeve mreže . . . 9

2.4 Pregled programskih verifikacijskih orodij . . . 9

2.4.1 Opisni jezik LOTOS . . . 9

2.4.2 Programsko orodje PSF . . . 10

2.4.3 Zbirka programskih orodij EDT . . . 10

2.4.4 Programsko orodje ITP . . . 10

2.4.5 Programsko orodje SPIN 10
2.4.6 Programsko orodje SMV 11

3 Barvne Petrijeve mreže **13**
3.1 Osnove CPN 14
3.2 Proženje CPN 14
3.3 Čuvaj 16
3.4 Hierarhične CPN 17
3.5 Pregled orodij 20
3.5.1 Design/CPN 20
3.5.2 CPN-AMI 21
3.5.3 INA 21
3.5.4 Snoopy 22
3.5.5 CPN Tools 23

4 Modeliranje in analiza vzorčnega protokola **29**
4.1 Opis izbranega protokola 30
4.1.1 Leaky Bucket algoritem 30
4.1.2 Token Bucket algoritem 30
4.2 Postavitev modela 32
4.3 Analiza modela ob nedeterminističnem generatorju števil 35
4.3.1 Statistika 37
4.3.2 Dosegljivost stanj 37
4.3.3 Domače označitve 38
4.3.4 Omejenost 38
4.3.5 Živost 40
4.3.6 Prostor stanj 40
4.3.7 Težave nedeterminizma 42
4.4 Odprava nedeterminizma in rezultati 44

5 Zaključek **49**

1 Uvod

Verificiranje predstavlja obvezen postopek pri razvoju računalniških sistemov. Ravno zato smo s to diplomsko nalogo želeli prikazati, zakaj je temu tako. Ko imamo postavljen model sistema, lahko na podlagi rezultatov verificiranja model dopolnemo in popravimo, in se tako postopoma bližamo končni rešitvi. Z uspešno zaključenim postopkom verificiranja dosežemo željen produkt.

1.1 Motivacija

Računalniški sistemi postajajo za razvoj vedno bolj zapleteni. Razlog za to je predvsem njihova naraščajoča kompleksnost. Problem nastane, ko je take sisteme potrebno verificirati. Kljub temu, da bodo načrtovalci bili kar se da pazljivi pri načrtovanju, obstaja velika možnost da ne pričakujejo določenega obnašanja, ki bo nato javil napako [11]. V postopku verifikacije želimo preveriti, ali sistem deluje tako, kot smo to specificirali. Postopek verificiranja želimo opraviti hitro in učinkovito. Za namen verificiranja zato obstaja množica metod kot so algoritmični pristop, opisovanje s procesno algebro, predstavitev sistema z grafom Petrijeve mreže, opisni jeziki ter predstavitev sistema z

avtomatom s končnim številom stanj. Za omenjene postopke obstajajo namenska programska orodja, s katerimi izvedemo postopek verificiranja. V tem diplomskem delu se osredotočimo na barvne Petrijeve mreže, ki so grafični jezik za predstavitev računalniških sistemov in protokolov [14]. Programskih orodij za namen kreiranja in analize Petrijevih mrež obstaja precej. Z njihovo pomočjo se nam izvajanje simulacije in verifikacije močno poenostavi. Eno izmed takih orodij je program CPN Tools, s katerim lahko opravimo omenjene postopke.

1.2 Pregled diplomskega dela

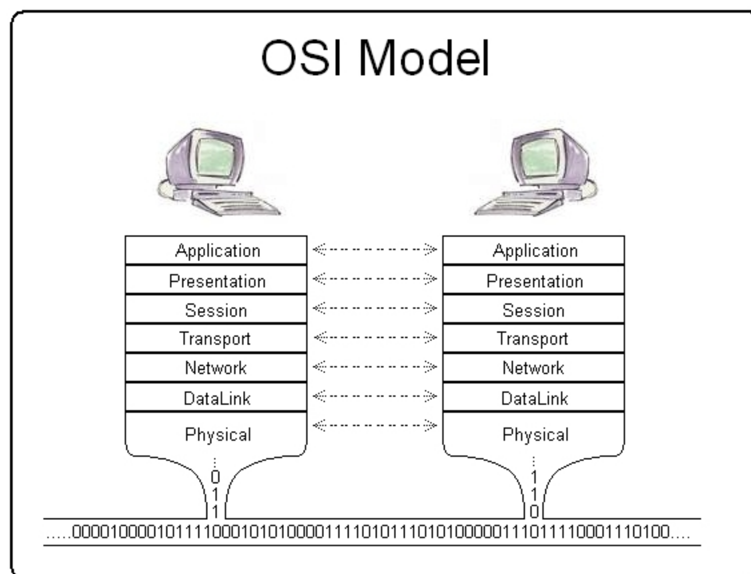
Namen diplomskega dela je prikaz postopka verificiranja izbranega protokola. Drugo poglavje je namenjeno teoriji. Najprej definiramo komunikacijski protokol in pomen verificiranja, zatem pa opišemo metode za verificiranje in orodja, ki opisane metode podpirajo. Tretje poglavje je v celoti namenjeno barvnim Petrijevim mrežam, kjer jih opišemo bolj podrobno. Pričnemo pri samih osnovah barvnih Petrijevih mrež, ne spuščamo se pa v osnovne Petrijeve mreže, ker predpostavljamo, da ima bralec že neka začetna znanja o njih. V nadaljevanju opišemo pravila proženja barvnih Petrijevih mrež in pomen čuvajev. Sledi še opis, kako lahko neko barvno Petrijevo mrežo pretvorimo v hierarhično, zatem pa predstavimo še nekatera orodja, ki so namenjena analizi, simulaciji, verifikaciji in modeliranju barvnih Petrijevih mrež. V četrtem poglavju predstavimo protokol vedra z žetoni, nato pa s pomočjo orodja CPN Tools zgradimo njegov model. S pomočjo slednjega opravimo postopek verifikacije na osnovi analize prostora stanj in opišemo rezultate, ki smo jih dobili ob samem postopku verificiranja.

2 Verifikacija komunikacijskih protokolov

V pričujočem poglavju si bomo najprej ogledali samo definicijo komunikacijskega protokola in pomen njegove verifikacije. Zatem bodo predstavljene nekatere izmed metod, ki se uporabljajo za verifikacijo in ustrezna programska orodja. Ker programskih orodij obstaja veliko, smo se omejili na izbiro le takih, ki podpirajo verifikacijo z omenjenimi metodami.

2.1 Definicija protokola

Prva stvar, ki jo je potrebno razumeti je ta, kaj sploh je komunikacijski protokol. Komunikacijski protokol je zbirka pravil, ki definira kako poteka komunikacija med dvema entitetama v omrežju [25]. Protokol je lahko implementiran strojno, programsko, ali pa na oba načina. Običajno je definiran glede na sloje ISO/OSI modela (glej sliko 2.1) ali TCP/IP modela. Tako lahko delimo komunikacijske protokole kar glede na sloje. Poznamo npr. transportne protokole (TCP, UDP) in aplikacijske protokole (HTTP). Vsaka skupina protokolov ima svoje funkcionalnosti, ki se delijo glede na sloj v katerem



Slika 2.1 Slika prikazuje ISO/OSI model po plasteh. Plasti si sledijo ena za drugo, zarisane pa so tudi P2P povezave med istima plastema [19].

nastopajo - vsak protokol bo rešil probleme na tistem sloju, na katerem deluje in nato svoje rezultate posredoval naprej.

2.2 Pomen verifikacije protokola

Verifikacija je potek opisovanja protokola v nekem jeziku (npr. končni avtomat, Petrijeva mreža) in nato primerjanje izvajanja sistema opisanega v tem jeziku z izvajanjem, ki smo ga definirali v postopku specifikacije [11]. V postopku verificiranja se generirajo stanja, v katerih se protokol trenutno nahaja. Ta stanja se sproti primerjajo s specifikacijami. V kolikor pride do inkonsistence med doseženimi in specificiranimi stanji javimo napako, nakar imamo možnost to napako odpraviti. Poglejmo si najprej terminologijo potrebno za samo razumevanje komunikacijskih protokolov in njihovo verificiranje.

V omrežjih, ki temeljijo na slojni sestavi (TCP/IP), so protokoli definirani kot zbirka pravil, ki povedo, kako naj se sporočila po omrežju izmenjujejo. S tem želimo doseči dva cilja [7]:

- pošiljanje paketov sosednjim plastem,
- ”peer-to-peer”povezavo med dvema vozliščema.

Prvi cilj imenujemo *specifikacija storitve*, drugi pa *specifikacija protokola*. Iz tega je razvidno, da sta za uspešno komuniciranje potrebni dve povezavi - prva povezava je povezava med istima slojema pri pošiljatelju in prejemniku (P2P). Primer za to je potrjevanje paketov pri TCP protokolu na transportnem sloju. Druga povezava je med parom zgornjega in spodnjega sloja (npr. transportna plast je odvisna od omrežne plasti). Nižji sloj prejme paket iz zgornjega sloja in ga enkapsulira s tem, da doda glavo in rep (angl. *header*, *trailer*), ki vsebujeta podatke potrebne za uspešno delovanje.

Nek protokol lahko verificiramo glede na njegovo *implementacijo* ali pa glede na njegovo *zgradbo*. Razlika je v tem, da verifikacija zgradbe poteka lahko že v fazi izdelave, verifikacija implementacije pa šele v fazi testiranja. Verifikacija zgradbe nam lahko močno zmanjša ceno razvoja in testiranja, saj lahko napake odkrijemo že zgodaj v času razvoja. S pomočjo verifikacije komunikacijskega protokola se želimo izogniti naslednjim tipom napak [7]:

- *Nedefiniran sprejem* (angl. *undefined reception*): to je sprejem, ki se lahko izvede, kljub temu, da ni bil specificiran v fazi načrtovanja;

- *Neizvršljivi deli* (angl. *dead code*): oddaja ali sprejem, ki sta bila specificirana v fazi načrtovanja, vendar se nikoli ne izvedeta;

- *Mrtva zanka* (angl. *deadlock*): gre za primer, ko se proces ujame v neko neskončno zanko in je ne more zapustiti; mrtvim zankam se je potrebno obvezno izogniti;

- *Livelock:* gre za stanje, v katerem si množica procesov konstantno izmenjuje sporočila brez kakršnegakoli efekta; tako kot cikel je to neskončna zanka, ki je procesi ne morejo zapustiti.

Iz povedanega je razvidno, da je glavni cilj verificiranja zagotavljanje, da protokol izpolnjuje vse pogoje, ki so bili določeni v fazi njegovega načrtovanja in izdelave specifikacij.

2.3 Metode verificiranja komunikacijskih protokolov

Za verificiranje obstaja več različnih metod. Vsaka ima svoje prednosti in slabosti. Ogledali si bomo nekaj najbolj pogostih, na koncu tega razdelka pa bo predstavljena

metoda *barvnih Petrijevih mrež* (angl. *CPN - Coloured Petri Nets*), ki jo bomo kasneje uporabili za verifikacijo vzorčnega protokola.

2.3.1 Algoritmični pristop

Pri algoritmičnem pristopu si bomo na kratko ogledali metodo verificiranja protokolov, ki temelji na Hoareovi logiki. Predstavljena je bila leta 1969 s strani britanskega znanstvenika C.A.R. Hoarea [23]. Metoda opisuje protokole kot skupek procesov, ki med seboj na nek način komunicirajo. Osnova za Hoareovo logiko je *Hoareov trojček*, ki ga zapišemo z izrazom

$$\{P\}C\{Q\}, \tag{2.1}$$

pri čemer je C ukaz, P in Q pa sta dve trditvi - P je *predpogoj* (angl. *precondition*), Q pa *končni pogoj* (angl. *post condition*). Definiramo ga lahko na naslednji način: če začnemo program v pogoju P, ko je ta izpolnjen in nato izvedemo C, potem se C manifestira v izpolnjenosti pogoja Q.

Preverjanje s Hoareovo logiko je bilo popularno predvsem v šestdesetih letih prejšnjega stoletja. Uporabnik je moral sam sestavljati dokaze s pomočjo aksiomov, ki so mu na deduktiven način pomagali do končne rešitve, je pa bilo tako delo zelo zahtevno in dolgotrajno.

2.3.2 Procesna algebra

V računalništvu se procesna algebra uporablja predvsem pri opisovanju sistemov kot algebraičnih procesov [6]. Osnova za procesno algebro so *atomične akcije* (a, b, c, d, itd.). Določene akcije predstavljajo poseben pomen (δ - *mrtva zanka*). Obstaja več izvedenk procesne algebre. V našem delu je predstavljena izvedenka ACP (angl. *Algebra of Communicating Processes*).

Atomične operacije lahko združujemo v procese z operatorji kot so +, ki predstavlja izbiro in *, ki predstavlja zaporedje.

V primeru iz izraza 2.2 proces najprej izbere akcijo a ali b, nato sledi akcija c in zaključek.

$$(a + b) * c \tag{2.2}$$

V primeru iz izraza 2.3 bo proces vedno izbral akcijo x, saj nas δ pripelje v mrtvo zanko.

$$\delta + x = x \tag{2.3}$$

Izraz 2.4 prikazuje zaporedje. Ker se akcije izvedejo po vrsti, smo že takoj na začetku v mrtvi zanki. Akcija x tega ne more spremeniti.

$$\delta * x = \delta \tag{2.4}$$

2.3.3 CFSM model

Pri tej metodi so protokoli predstavljeni kot avtomati s končnim številom stanj, ki med seboj komunicirajo [27]. Ravno iz tega izhaja tudi ime metode (angl. *CFSM - Communicating Finite State Machine*). Tehnika verifikacije temelji na preverjanju dosegljivosti stanj. Vsa stanja, ki so dosegljiva iz nekega začetnega stanja, so preverjena za logične napake, nedefinirane sprejeme, nastop mrtvih zank, itd.

Največji problem, ki se pojavlja pri tem modelu, je *eksplozija stanj*, saj je lahko dosegljivih stanj iz nekega drugega stanja ogromno. Za rešitev tega problema obstaja množica rešitev, ki so sledeče:

- število procesov, ki med seboj komunicirajo je omejeno na dva,
- procesi ne vsebujejo ciklov, razen tistih, ki potekajo skozi začetno stanje,
- proces lahko naenkrat le oddaja ali sprejema.

2.3.4 ECFSM model

ECFSM (angl. *Extended CFSM*) je razširitev CFSM modela. Ta model vpelje v postopek verifikacije še spremenljivke. Te spremenljivke postanejo samosvoja implicitna stanja, ki lahko zasedajo določene vrednosti. Na začetku jim določimo njihove tipe in začetne vrednosti.

Model ECFSM je definiran [20] kot petorček ECFSM = (S, I, O, V, T), kjer so pomeni oznak sledeči:

- S - množica stanj,
- I - množica vhodov,
- O - množica izhodov,
- V - množica spremenljivk,
- T - množica povezav.

Za opisovanje ECFSM modelov se uporablja jezik SDL (Specification and Description Language). Prednost SDLja je ta, da je dobro sprejet in podprt s strani ISO in ITU-T. Razvit je bil pri ITU - International Telecommunication Union.

2.3.5 Estelle metoda

Metoda Estelle je bila razvita pri ISO (International Organization for Standardization) in temelji na razširjenem končnem avtomatu [18]. Sestavljata jo dva dela in sicer

- razširjeni končni avtomat (ECFSM),
- jezik za opisovanje sistema, ki temelji na programskem jeziku Pascal.

S pomočjo Estelle metode lahko sistem specificiramo kot hierarhično strukturo sestavljeno iz posameznih modulov. To so povezane komponente sistema, ki si med seboj s pomočjo akcij izmenjujejo sporočila po dvosmernih povezavah.

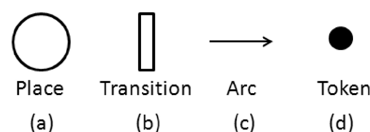
2.3.6 Petrijeve mreže

Petrijeve mreže (angl. *Petri Nets*) so orodje za simulacijo in modeliranje sistemov. Omogočajo nam, da za nek sistem postavimo njegov model, nato pa s simulacijo modela pridobimo informacije o pravilnosti delovanja sistema. Na podlagi pridobljenih informacij lahko nato sistem spremenimo ali dopolnimo.

Gradniki Petrijevih mrež so pogoji, akcije in povezave med njimi. Povezave so dovoljene le iz akcije v pogoje in obratno. Primer, ko povezava potuje iz akcije v akcijo ali iz pogoja v pogoj ni dovoljen. Zadnji izmed gradnikov Petrijev mrež so žetoni, ki so nastanjeni v pogojih. En žeton v pogoju pomeni enkratno izpolnjenost pogoja.

Žetoni so potrebni za uspešno izvajanje akcij. Neka akcija se lahko izvede, ko se po vsaki povezavi, ki pelje iz predpogoja v to akcijo pripelje en žeton. V tem primeru se akcija izvede, en žeton pa se ponovno prenese po vsaki povezavi, ki gre iz akcije v izhodne pogoje in se tam ustavi. Na sliki 2.2 so prikazani vsi elementi, ki sestavljajo grafično ponazoritev Petrijeve mreže.

Omenjene Petrijeve mreže so najbolj osnovne. Poznamo tudi barvne, mehke, stohastične in časovne Petrijeve mreže. V nadaljevanju si bomo natančneje ogledali barvne Petrijeve mreže.



Slika 2.2 Sestavni deli Petrijevih mrež: place - pogoj, transition - akcija, arc - povezava, token - žeton [8].

2.3.7 Barvne Petrijeve mreže

Barvne Petrijeve mreže (CPN - Coloured Petri Nets) predstavljajo razširitev Petrijevih mrež. So grafično orientiran jezik, ki se uporablja za specifiranje, simuliranje in verificiranje sistemov [14]. Predvsem je uporaben na področju modeliranja in verifikacije komunikacijskih protokolov.

Najbolj opazna razlika glede na običajne Petrijeve mreže je ta, da se v barvnih Petrijevih mrežah spremeni pomen žetona. Ta postane podatkovna struktura in na ta način omogoča prenos spremenljivk, katerim so določene neke vrednosti. Njihovo sestavo, uporabo in programska orodja si bomo natančneje ogledali v naslednjem poglavju.

2.4 Pregled programskih verifikacijskih orodij

Zaradi vse bolj kompleksnih sistemov je pomen verificiranja za uspešno delovanje sistema zelo pomembno. V tem razdelku si bomo pogledali programska orodja za izvedbo metod, ki smo jih predhodno že opisali. Orodij obstaja veliko, zato smo se pri izbiri omejili na nekaj najbolj tipičnih.

2.4.1 Opisni jezik LOTOS

LOTOS (Language Of Temporal Ordering Specification) je široko uporabljan jezik za opisovanje sistemov, ki temelji na procesni algebri. Razvit je bil pri ISO in predstavlja enega izmed standardov opisovanja sistemov [21]. Eno izmed orodij, ki za svoje delovanje uporablja LOTOS, je CADP (Construction and Analysis of Distributed Processes). Deluje tako, da specifikacijo spisano v jeziku LOTOS prevede v C program ali pa v t.i. LTS (Labelled Transition System) [1]. LTS je pravzaprav avtomat s končnim številom stanj, ki ima vse povezave poimenovane po svojih akcijah. Na ta način preprosto vemo katera akcija se je izvedla, ko smo po eni izmed povezav prišli iz prvotnega stanja v novo stanje. Za verificiranje se nad LTS opravi analiza dosegljivosti stanj.

2.4.2 Programsko orodje PSF

PSF (Process Specification Formalism) je prostodostopno orodje, ki za opisovanje sistemov uporablja procesno algebro, natančneje njeno izvedenko ACP (Algebra of Communicationg Processes), ki je bila predhodno predstavljena v tem poglavju. Procesi so v PSF opisano kot zaporedje atomičnih operacij, ki so nato povezane skupaj s pomočjo operatorjev [10]. S to združitvijo dobimo izraze, s katerimi definiramo procese.

2.4.3 Zbirka programskih orodij EDT

EDT (Estelle Development Toolset) je zbirka orodij za specikacijo in verifikacijo [2]. Temelji na metodi Estelle. Orodje je v svetu precej poznano. Uporablja ga več kot 30 institucij po celem svetu. Za uporabo EDT je na voljo enomesečna preizkusna verzija, sicer pa je za komercialno uporabo potrebno pridobiti licenco.

EDT sestavljajo naslednji gradniki:

- prevajalnik, ki Estelle specifikacijo prevede v programsko kodo C,
- simulator in razhroščevalnik, ki se uporabljata za odkrivanje napak,
- generator, ki lahko Estelle specifikacijo z n podsistemi razbije v n Estelle specifikacij,
- generator tabel za generiranje tabel stanj in tabel dogodkov,
- grafični urejevalnik za definiranje, urejanje in pregledovanje Estelle specifikacij.

2.4.4 Programsko orodje ITP

Programskih orodij, ki bi podpirali Hoareovo logiko, je malo, kar verjetno kaže tudi na to, da je metoda Hoareove logike zapletena in že precej stara. Eno izmed orodij, ki bi ga vseeno omenili, je ITP Tool. Deluje predvsem v kombinaciji s programskim jezikom Java. Glavni namen uporabe je preverjanje lastnosti programske kode pisane v jeziku Java (preverjanje pravilnosti delovanja zank, aritmetičnih operacij, prireditev, itd.).

2.4.5 Programsko orodje SPIN

Orodje SPIN temelji na metodi preverjanja dosegljivosti stanj. Razvit je bil pri Bell Labs. Jezik, ki se uporablja za opisovanje sistema, je Promela, ki je nedeteriminističen jezik s podporo za podatkovne strukture, prekinitve in omejevanje kode za atomično

izvajanje [12]. SPIN je odprtokodni program za verificiranje in je prosto dostopen iz uradne spletne strani [3]. Deluje "on-the-fly"kar pomeni, da ni potrebna predhodna izgradnja grafa globalnih stanj sistema. To je velika prednost pri verificiranju velikih sistemov.

2.4.6 Programsko orodje SMV

SMV (Simbolic Model Verifier) je orodje za avtomatsko preverjanje sistemov [12]. Sistem je predstavljen v obliki drevesa CTL (Computational Tree Logic). SMV program si lahko predstavljamo kot zaporedje računskih operacij. Rezultat vsake operacije predstavlja novo stanje, v katerem se sistem nahaja.

V tem razdelku smo si ogledali nekaj orodij, ki temeljijo na končnih avtomatih, Hoare-ovi logiki, metodi Estelle in procesni algebri. Preostanjejo nam še orodja za Petrijeve mreže, ki pa si jih bomo pogledali v naslednjem poglavju, ki je v celoti namenjem barvnim Petrijevim mrežam.

3 Barvne Petrijeve mreže

Razvoj sistemov je vedno bolj zapleten. Razlog za to je predvidena kompleksnost sistemov in hkratnost izvajanja procesov, kar pomeni, da lahko ob večkratnem izvajanju funkcij sistema dobimo popolnoma drugačne rezultate. Na to vpliva npr. razvrščanje procesov in izguba sporočil med komuniciranjem. Ravno zaradi teh razlogov je načrtovanje modernih sistemov zapleteno. Možnost človeške napake ob načrtovanju je velika, zato potrebujemo nekakšno rešitev, da se temu lahko izognemo. Eden izmed pristopov je izgradnja *modela* sistema, ki nastopi v zgodnjih fazah razvoja [15].

S postavitvijo modela sistema pridobimo dodaten vpogled v sistem. Razvijalec ima večji pregled nad sistemom in bolj celovito razume delovanje sistema. Model nas prav tako vodi do bolj celovite specifikacije sistema, s čimer lahko napake v sistemu odkrijemo hitreje. V postopku verificiranja se običajno vedno pojavijo napake. V primeru, ko do določenega dogodka v času simulacije ne more priti, bo razvijalec hitro odkril kaj je v sistemu šlo narobe ali kaj sistemu manjka, da bi dosegel željeno funkcionalnost.

Še ena izmed prednosti modela je ta, da napake lahko reproduciramo. To v realnih sistemih ni možno zaradi neponovljivosti bremena. S tem, ko določene napake repro-

duciramo, lahko potrdimo ali je sprememba v sistemu, ki je bila potrebna zaradi pojava napake, resnično odpravila napako ali ne.

Postavitev modela običajno pomeni večji vložek truda na začetku razvoja. To lahko upravičimo s tem, da se veliko več napak odkrije že v fazi razvoja, kot pa v fazi implementacije.

V tem poglavju si bomo bolj podrobno ogledali koncept *barvnih Petrijevih mrež* (v nadaljevanju CPN - Coloured Petri Nets), ki so le ena izmed mnogih metod za modeliranje sistemov.

3.1 Osnove CPN

CPN je grafično orientiran jezik, ki se uporablja za modeliranje, specificiranje, simuliranje in verificiranje dinamičnih sistemov [14]. Za nadaljnje razumevanje spodbujamo bralca, da si prebere osnove Petrijevih mrež, ki jih lahko najde v viru [22].

Tako kot osnovne Petrijeve mreže, so tudi CPN sestavljene iz *pogojev*, ki jih označujemo s krogi ali elipsami, iz *akcij*, ki jih označujemo s pravokotniki in iz *povezav*. Povezave so usmerjene črte, ki potekajo iz akcije v pogoj in obratno. Povezava nikoli ne sme potekati iz akcije v akcijo, ali iz pogoja v pogoj. Zadnji sestavni del CPN so *žetoni*, ki jih najdemo v pogojih. Na vhodnih povezavah v akcije je vedno zapisan *izraz*, ki pove kakšni podatki se iz vhodnega pogoja prenesejo v akcijo, na izhodni povezavi iz akcije pa izraz, ki pove kakšni podatki se prenesejo v izhodni pogoj.

V žetonih tiči tudi glavna razlika med CPN in osnovnimi Petrijevimi mrežami. V CPN dobi žeton pomen sestavljene spremenljivke, ki lahko prenaša podatke določenega tipa. Prav po tej lastnosti so CPN tudi dobile svoje ime. Žetoni lahko prenašajo različne podatke in se na ta način razlikujejo med seboj.

3.2 Proženje CPN

Da pride do proženja akcije morata biti izpolnjena naslednja dva pogoja:

- Vhodni pogoji (to so tisti, katerih povezave peljejo v akcijo) morajo vsebovati zadostno količino žetonov.

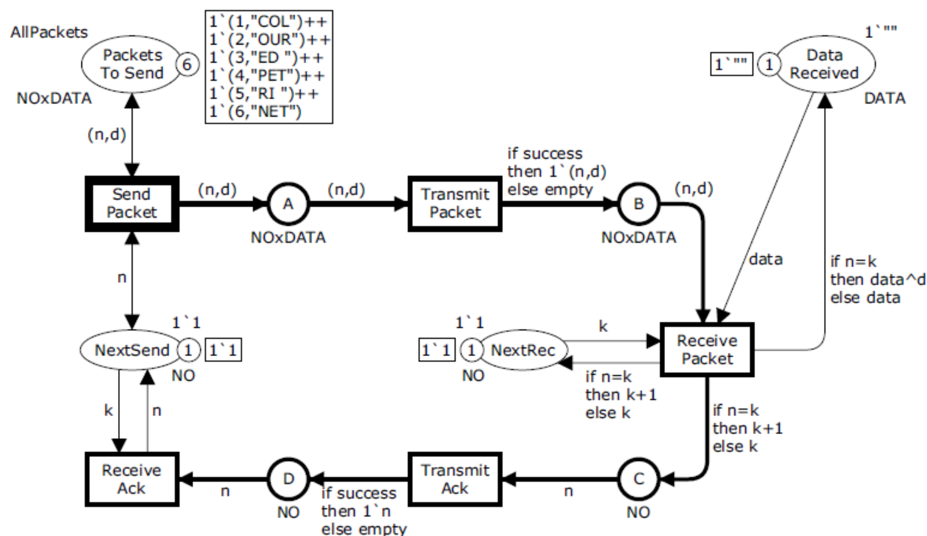
- Žetoni, ki so v vhodnih pogojih, morajo imeti vrednosti take, kot jih zahteva izraz na povezavi.

Poglejmo si primer enostavnega oddajno sprejemnega protokola, povzetega po viru [15], ki je predstavljen na sliki 3.1. Prva akcija, ki se bo izvedla v tem modelu, je akcija *SendPacket*. Da se ta akcija izvede, morata biti izpolnjena njena vhodna pogoja, to sta pogoja *PacketsToSend* in *NextSend*. Naslednja stvar, ki mora biti izpolnjena je ta, da so tipi žetonov enaki tipom, ki so navedeni na povezavah, po katerih bodo te žetoni odšli do akcije. V žetonih se nahajata dve spremenljivki n in d (na sliki označena z *NO* in *DATA*) tipa *integer* in *string*. Pri akciji *NextSend* imamo trenutno le en žeton z vrednostjo 1 tipa *NO*. Torej jasno je, da je začetna vrednost spremenljivke n v tem žetonu enaka 1. Iz tega lahko izpeljemo, da mora biti vrednost spremenljivke d enaka *"COL"*, saj imamo v pogoju *PacketsToSend* le en žeton z vrednostjo $n = 1$. Prvi žeton, ki se bo odpeljal do akcije *SendPacket* je zato žeton z vrednostjo $[n = 1, p = \text{“}COL\text{”}]$. Ko se akcija *SendPacket* izvede, se oba žetona odstranita iz vhodnih pogojev, ampak takoj zatem vrneta, saj imamo dvosmerne povezave. Razlog za to je dejstvo, da se žeton (paket) lahko v omrežju izgubi. To nam omogoča ponovno pošiljanje izgubljenega paketa.

Ko žeton $[n = 1, p = \text{“}COL\text{”}]$ prispe do pogoja A lahko rečemo, da je paket v omrežju. Naslednja akcija, ki se bo izvedla, je akcija *TransmitPacket*. Ta akcija se lahko izvede na dva načina in sicer z žetonom $[n = 1, p = \text{“}COL\text{”}, success = true]$ ali z žetonom $[n = 1, p = \text{“}COL\text{”}, success = false]$. V prvem primeru se žeton uspešno prenese iz pogoja A v pogoj B preko akcije *TransmitPacket* . V drugem primeru, ko je $success = false$, pride do izgube žetona. Potrebno je ponovno pošiljanje od samega začetka.

S tem, ko je žeton prispel do pogoja B, je postala omogočena akcija *ReceivePacket*. Vrednost spremenljivke n v žetonu se primerja z vrednostjo spremenljivke k v pogoju *NextRec*, ki predstavlja število naslednjega prejetega paketa. V kolikor se vrednosti ujemata, se vsebina spremenljivke d doda vrednosti spremenljivke *data*, ki se nahaja v pogoju *DataReceived*. Naslednja stvar, ki se osveži, je vrednost žetona v pogoju *NextRec*. Poveča se za 1, saj pričakujemo naslednji žeton. Preko pogoja C pošljemo potrditev o prejetem žetonu v spremenljivki, ki nosi številko naslednjega pričakovanega žetona.

S tem smo omogočili akcijo *TransmitAck*, ki deluje na podoben način kot *TransmitPacket*. V primeru, da je $success = true$, se potrditev prenese naprej do pogoja D, sicer pa se izgubi.



Slika 3.1 Primer enostavnega oddajno-sprejemnega protokola v CPN obliki [15].

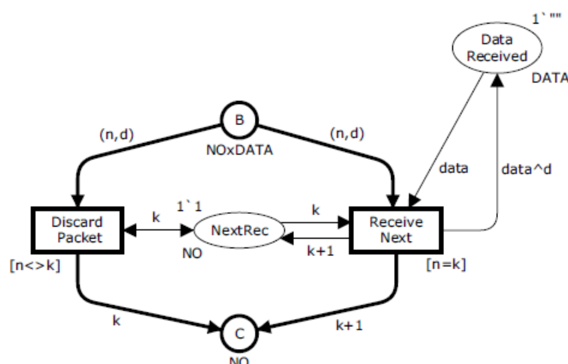
Iz pogoja D sledi še prenos žetona do akcije *ReceiveAck*, ki osveži vrednost žetona v pogoju *NextSend*. Na ta način je vse pripravljeno za prenos naslednjega žetona v vrsti pogoja *PacketsToSend*.

3.3 Čuvaj

Vpeljimo v koncept CPN še pojem *čuvaja*. Čuvaj predstavlja izraz, ki omejuje izvajanje akcij [15]. Zapisan je v oglatih oklepajih poleg akcije (glej sliko 3.2). V primeru CPN se pred izvajanjem akcije vedno preveri vrednost izraza v čuvaju. V kolikor dobimo rezultat izraza v čuvaju *true*, potem je ta akcija omogočena. Če bi bila vrednost *false*, potem se ta akcija ne bi mogla izvesti.

Na sliki 3.2, ki je povzeta po viru [15], je predstavljen del modela iz slike 3.1. Akcijo *ReceivePacket* smo razbili v dve posamezni: *DiscardPacket*, ki bo žeton zavrgla in *ReceiveNext*, ki bo žeton sprejela. *DiscardPacket* predstavlja primer, ko prejmemo napačni paket, *ReceieveNext* pa primer, ko smo prejeli paket v pravilnem zaporedju. Ko prispe žeton v pogoj B, postaneta omenjeni akciji omogočeni. Sedaj sledi preverjanje izrazov v čuvajih. Čuvaj pogoja *ReceiveNext* $[n = k]$ zahteva, da je vrednost n v žetonu enaka vrednosti k v pogoju *NextReceive*, čuvaj pogoja *DiscardPacket* $[n <> k]$ pa zahteva, da

se vrednosti n in k razlikujeta. V tem primeru bi bil žeton zavržen.



Slika 3.2 Delovanje čuvajev na primeru sprejemanja paketov [15].

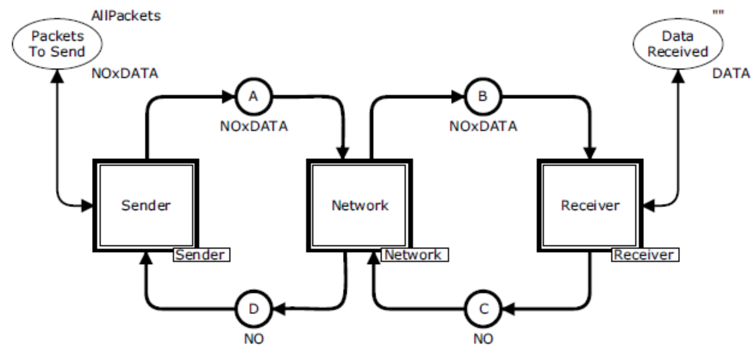
3.4 Hierarhične CPN

Klasične CPN (lahko bi rekli tudi nehierarhične CPN) je možno razbiti v posamezne *module*. Na ta način pridemo do *hierarhičnih CPN* [15]. Vsak modul si lahko predstavljamo kot zaključeno celoto. V primeru hierarhičnih CPN lahko sistem modeliramo s pomočjo modulov. Module modeliramo neodvisno drug od drugega. Načrtovalci modela si lahko module predstavljajo kot črne škatle. Vsebina modulov, katerim ob nekem trenutku ne posvečajo pozornosti, jih ne zanima in to tudi ne vpliva na samo modeliranje sistema kot celote.

Prednost takega pristopa se pokaže pri modeliranju velikih sistemov, ki so zelo kompleksni. Na ta način pridobimo na preglednosti modela sistema in na času, ki ga vložimo v samo modeliranje. Poglejmo si kako lahko primer, predstavljen v razdelku 3.1, zgradimo s pomočjo modulov [15].

Iz slike 3.1 lahko sklepamo, da je model sestavljen iz treh delov in sicer iz pošiljatelja, prejemnika in omrežja. Vsak del lahko prikažemo s posameznim modulom, pri čemer pogoji A, B, C in D delujejo kot povezave med njimi. Rezultat je prikazan na sliki 3.3.

Poglejmo si kako bi izgledal vsak modul posebej. To je vidno na slikah 3.4, 3.5 in 3.6. Pogoj A pri modulu "pošiljatelj"deluje kot izhodna vrata, pogoj D kot vhodna vrata, pogoj *PacketsToSend* pa kot vhodno-izhodna vrata. Ti trije pogoji predstavljajo *vmesnik*, preko katerega si modul izmenjuje žetone z ostalimi moduli. Modul "pošiljatelj"bo žetone sprejemal preko vrat D, pošiljal pa jih bo preko vrat A. Preko vhodno-izhodnih

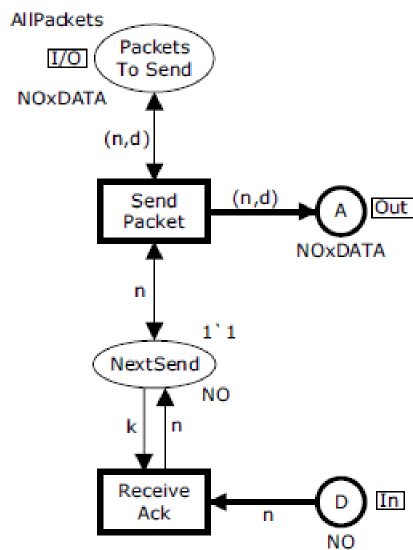


Slika 3.3 Pošiljatelj, prejemnik in omrežje so predstavljeni kot posamezni moduli [15].

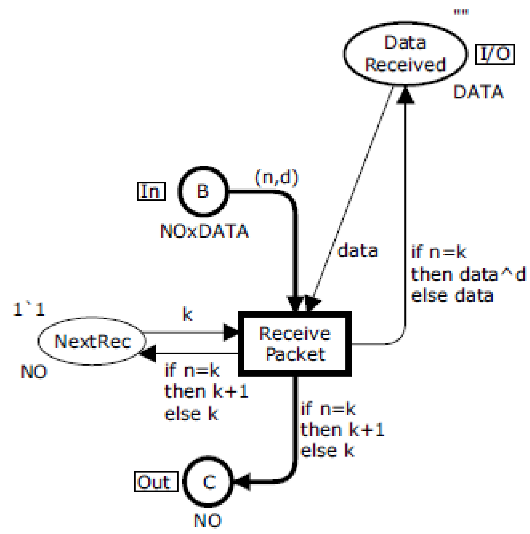
vrat *PacketsToSend* lahko modul prejema in pošilja žetone.

Podoben primer velja za modul "prejemnik". Preko vrat B modul prejema žetone, preko vrat C pa jih pošilja. Pogoj *DataReceived* predstavlja vhodno-izhodna vrata.

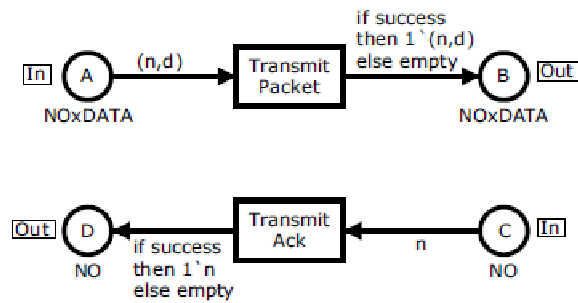
Modul "omrežje" vsebuje dvoje vrat, ki delujejo kot vhodna vrata, to sta pogoja A in C, ter dvoje vrat, ki delujejo kot izhodna vrata, to sta pogoja B in D.



Slika 3.4 Sestavni deli modula "pošiljatelj" [15].



Slika 3.5 Sestavni deli modula "prejemnik"[15].



Slika 3.6 Sestavni deli modula "omrežje"[15].

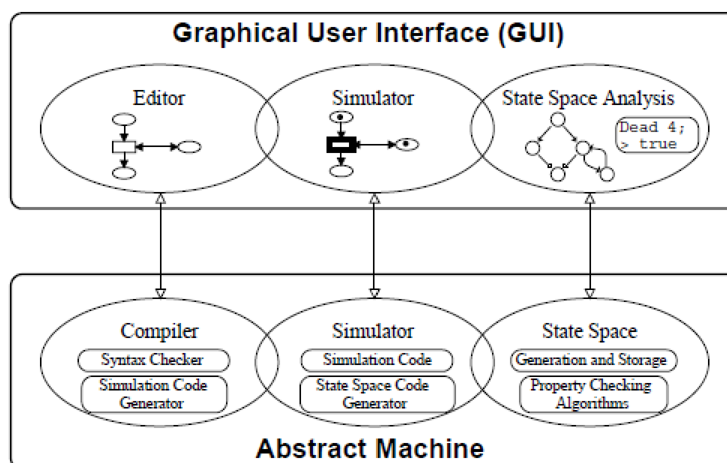
3.5 Pregled orodij

Orodij za načrtovanje in urejanje Petrijevih mrež obstaja zelo veliko. V našem delu smo se omejili na orodja, ki dobro podpirajo urejanje CPN. Orodja za vse tipe Petrijevih mrež si bralec lahko ogleda na spletni strani http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html.

3.5.1 Design/CPN

Design/CPN je bil prvi grafično podprt program za delo s CPN [9]. Prvič se je pojavil leta 1989 kot skupno delo Meta Software Corporation in univerze v Aarhusu na Danskem.

Orodje je sestavljeno iz dveh glavnih delov - *grafičnega uporabniškega vmesnika* in iz *abstraktnega dela* (abstraktnega stroja), kar je vidno na sliki 3.7.



Slika 3.7 Arhitektura programskega orodja Design/CPN [9].

Model sistema se najprej zgradi v *urejevalniku*, nato pa takoj sledi zagon preverjanja sintakse, ki se nahaja v *prevajalniku*. Preverjanje sintakse zagotavlja, da je uporabnik zgradil CPN, ki se podreja vsem pravilom o CPN. Ko je mreža preverjena sledi generiranje simulacijske kode v *simulatorju*. Omogočanje in izvajanje akcij se vedno preverja v simulatorju, ki se nahaja v abstraktnem delu, nadzor simulacije pa v simulatorju, ki se nahaja v grafičnem uporabniškem vmesniku. Generator prostora stanj omogoča izgradnjo funkcij in podatkovnih struktur, ki se bodo uporabljale pri *analizi prostora stanj*. Abstraktni del *prostor stanj* (angl. *State Space*) vsebuje generiranje in hranjenje pros-

tora stanj in algoritme za preverjanje lastnosti prostorov stanj. Ti dve funkciji sta na voljo uporabniku pri analizi prostora stanj, ki se izvaja v grafičnem delu programa.

Za opisovanje modelov se uporablja jezik CPN ML. CPN ML je razširitev jezika SML (Standard ML).

Design/CPN se v praksi ne uporablja več. Nasledilo ga je orodje CPN Tools, ki si ga bomo ogledali v kratkem.

3.5.2 CPN-AMI

CPN-AMI je še eno izmed orodij, ki podpirajo modeliranje, simuliranje in verificiranje sistemov. Zanimivost CPN-AMI je ta, da je v bistvu skupek različnih orodij za modeliranje in verificiranje Petrijevih mrež [4]. Vsa orodja povezuje integracijska platforma FrameKit, ki se uporablja za hitro implementacijo CASE (Computer Aided Software Engineering) okolij. Nekatera izmed orodij, ki so implementirana v CPN-AMI, so:

- LoLa: analiza prostora stanj, iskanje mrtvih zank,
- dot: specifikacija objektov v usmerjen graf,
- PROD: analiza dosegljivosti stanj,
- PeP: orodje za modeliranje, simulacijo in verifikacijo sistemov, ki bazira na Petrijevih mrežah.

CPN-AMI se večinoma uporablja za verificiranje sistemov. Je prosto dostopen z uradne spletne strani [4] za Windows, Linux in Mac platforme.

3.5.3 INA

INA (Integrated Net Analyzer) je razvil Peter H. Starke z namenom urejanja, analize in proženja Petrijevih mrež [5]. Orodje INA je sestavljeno iz naslednji delov:

- *Tekstovni urejevalnik* omogoča ustvarjanje novih ali urejanje obstoječih mrež. INA ne podpira grafičnega urejevalnika, je pa možno Petrijeve mreže urediti v kateremkoli grafičnem urejevalniku, ki zna pretvoriti mrežo v format, ki ga uporablja INA (.pnt format za osnovne Petrijeve mreže ali .cnt format za CPN).
- *Simulator* omogoča poljubno proženje akcij.

- *Redukcijski del* omogoča zmanjšati velikost mreže in obenem ohranjati vsa pravila pri gradnji Petrijevih mrež (ali se mreža proži pravilno, omejenost, itd.).
- *Analitični del* preiskuje sistem za pasti, konflikte v strukturi mreže in mrtve zanke.

INA je za nekomercialne namene prosto dostopen z uradne spletne strani [5] za Windows in Linux platforme.

3.5.4 Snoopy

Snoopy je orodje, ki omogoča gradnjo in analizo različnih vrst Petrijevih mrež. Ogrodje Snoopy se deli na dva dela in sicer na barvni in nebarvni del. Vsak del vsebuje različne tipe Petrijevih mrež. Nebarvni del tako vsebuje časovne, stohastične in hibridne Petrijeve mreže. Barvni del vsebuje popolnoma enake tipe Petrijevih mrež le, da jim je dodana še barva, torej žetoni omogočajo prenos spremenljivk. Snoopy se uporablja predvsem za raziskovanje velikih bioloških sistemov in modeliranje računalniških sistemov [13].

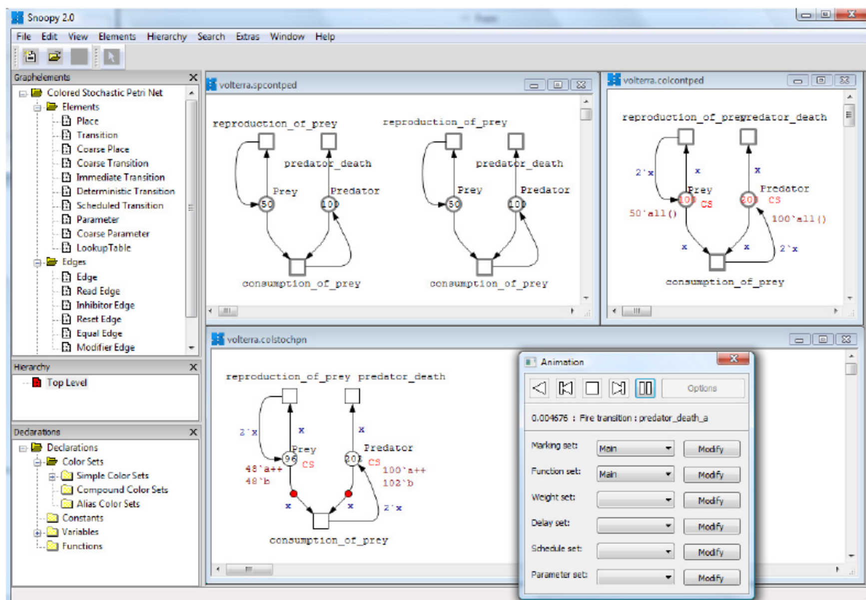
Snoopy ponuja možnost pretvorbe mrež iz barvnih v nebarvne in obratno. S prehodom v drug tip se spremeni le izgled mreže, sama struktura pa ostane popolnoma enaka.

Za urejanje Petrijevih mrež Snoopy ponuja grafični urejevalnik, viden na sliki 3.8, ki ga sestavljajo:

- *okno vseh elementov*, ki vsebuje spisek vseh grafičnih elementov v mreži (pogoji, akcije, povezave),
- *hierarhija*, ki prikazuje hierarhijo modela,
- *deklaracije*, ki vsebujejo spisek vseh spremenljivk in konstant za potrebe CPN,
- *okno za risanje*, kjer rišemo model sistema.

Za potrebe modeliranja so na voljo naslednje osnovne funkcije:

- *deklariranje spremenljivk* (samo za potrebe CPN),
- *dodajanje grafičnih elementov* (akcij, pogojev, povezav),
- *urejanje lastnosti vozlišč* (imena elementov, začetna označitev, število povezav, itd.).



Slika 3.8 Grafični uporabniški vmesnik orodja Snoopy [13].

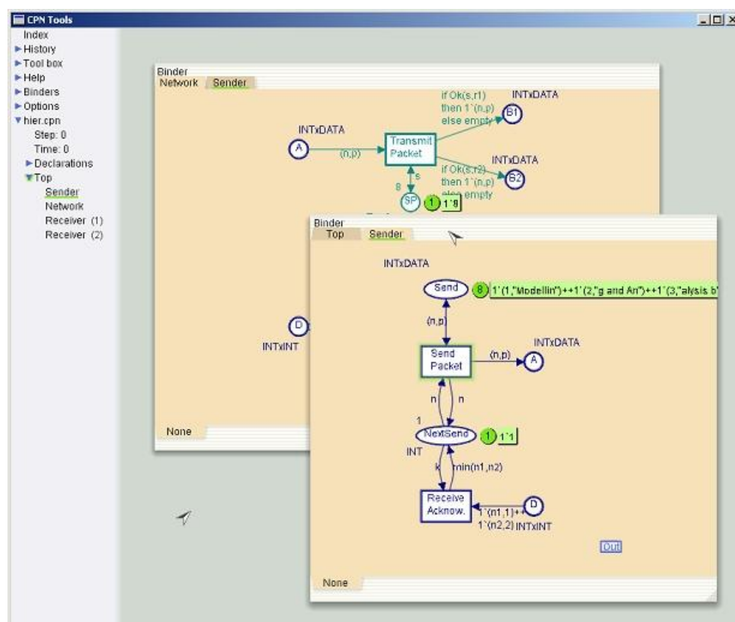
Snoopy nam prav tako omogoča grajenje hierarhičnih CPN, kot smo razložili v razdelku 3.4, kjer lahko posamezne dele mreže združimo v logične enote - podmreže. Za ta namen so na voljo t.i. *grobe akcije* in *grobi pogoji* (angl. *coarse transitions, places*), ki predstavljajo mejo med posameznimi podmrežami.

Snoopy ponuja animacijo pri izvrševanju klasičnih Petrijevih mrež in njihovih barvnih sorodnikov (CPN). Animacija predstavlja pot žetonov po mreži, s čimer lahko uporabnik vidi kako se bo sistem obnašal in na podlagi tega že ugotovi, kje pride do napak pri izvajanju. Mrežo je možno izvrševati v poljubnem številu korakov (en korak, večje število korakov, maksimalno število korakov). Po zaključku izvršitve nam Snoopy poda simulacijske rezultate. Za bolj podrobno analizo Snoopy omogoča izvoz mrež v različne formate. Primer orodij, ki se uporabljata za temeljitejšo analizo, sta Charlie in Marcie.

Orodje Snoopy je na voljo za Windows, Linux in Mac platforme. Za potrebe učenja je prosto dostopen z uradne spletne strani `http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Snoopy`.

3.5.5 CPN Tools

Zadnje orodje, ki si ga bomo pogledali, je CPN Tools. Predstavlja še eno izmed orodij za kreiranje in analizo Petrijevih mrež, bolj natančno za časovne in nečasovne CPN. CPN



Slika 3.9 Grafični uporabniški vmesnik orodja CPN Tools. Prikazano je glavno okno z delovnim prostorom in seznamom orodij [24].

Tools je naslednik orodja Design/CPN, ki smo si ga predhodno že ogledali.

CPN Tools je rezultat skupnega projekta s strani univerze v Aarhusu, univerze George Mason in podjetij Nokia, Microsoft ter Hewlett-Packard [24]. Namen tega projekta je bil popolna sprememba obstoječega orodja Design/CPN. Vse modele, ki so izdelani v orodju Design/CPN, je možno pretvoriti v format za CPN Tools, obratno pa to ni mogoče.

Zgradba

Zgradba CPN Tools se močno razlikuje od prej že opisanih orodih. Prva opazka je ta, da ne vsebuje klasičnega izvlečnega seznama. Večino okna zavzema *delovni prostor* (angl. *Workspace*), ki lahko vsebuje več različnih oken, ki jih imenujemo *binders*. Na njih se lahko nahaja različno število *strani*. Na straneh najdemo mrežni model, deklaracije ali zbirko orodij. Strani, ki so med seboj povezane, lahko združimo v eno okno in si na ta način olajšamo delo. Na levi strani glavnega okna tiči *seznam* (angl. *index*), ki vsebuje spisek vseh orodij in elementov, ki so na voljo za uporabo v CPN Tools. Izgled uporabniškega vmesnika je prikazan na sliki 3.9.

Urejevalnik

Kreiranje in urejanje Petrijevih mrež v CPN Tools je za uporabnika močno poenostavljeno. Že ob sami gradnji se preverja sintaktična pravilnost mreže, obenem pa se tudi že generira simulacijska koda. V kolikor pride do napake pri gradnji, orodje sproti javlja napake.

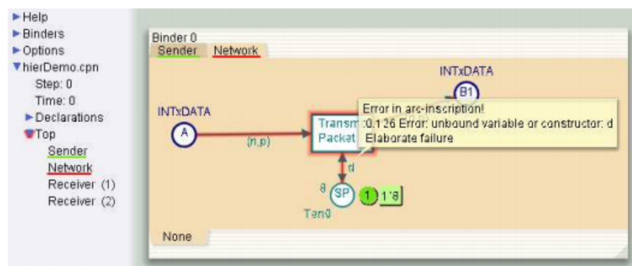
Orodja

Večino orodij, ki se uporabljajo za kreiranje ali manipuliranje Petrijevih mrež najdemo v predelu seznama imenovanem *Tool Box*. Orodja so sledeča:

- *Create Tools*: To so orodja, ki se uporabljajo za kreiranje mrež. Tu so vsi glavni elementi (pogoji, akcije, povezave). CPN Tools ima implementiranega pomočnika, ki skrbi za lep izgled mreže ob postavljanju elementov. Za dodajanje opisa elementa je potrebno preprosto klikniti element, ki ga želimo spremeniti. V kolikor potrebujemo več enakih objektov, nam je v pomoč kloniranje oz. kopiranje elementov. S tem dobimo nov objekt, ki ima vse lastnosti enake staremu.
- *Style Tools*: Naloga teh orodij je samoumevna. Z njihovo pomočjo lahko elementom spreminjamo velikost, barvo, debelino in stil obrobe.
- *View Tools*: Ta orodja uporabljamo za definiranje skupin in za približevanje in oddaljevanje pogleda. Skupine definiramo za boljši pregled nad modelom. View Tools lahko uporabljamo v kombinaciji s Style Tools. Če spremenimo stil enemu objektu v skupini, bo končni rezultat sprememba stila vsem relevantnim članom skupine.
- *Hierarchy Tools*: Uporabljamo jih, ko ustvarjamo hierarhične CPN. Na ta način lahko nek model sistema razbijemo v več logičnih modelov, ki so med seboj povezani. V samem orodju to izgleda tako, da ustvarimo nove strani kot *podstrani* (angl. *subpage*), ki so podrejene *superstrani* (angl. *superpage*).

Preverjanje sintakse

Kot je že bilo omenjeno, se preverjanje sintakse konstantno izvaja v ozadju. Uporabnik dobi povratno informacijo o pravilnosti mreže v obliki barv, ki so aplicirane na elemente. Oranžna barva pomeni, da element še ni bil preverjen. Rumena barva pove, da je element



Slika 3.10 Primer sintaktične napake v CPN Tools. Element se obarva rdeče, obenem pa se pojavi še oblaček z opisom napake [24].

v fazi preverjanja. V kolikor so elementi preverjeni in med preverjanjem ni prišlo do napak, barva izgine. V primeru sintaktične napake, vidne na sliki 3.10, se element obarva rdeče, v bližnji okolici pa se dodatno pojavi še oblaček s sporočilom o napaki.

Analiza mrež

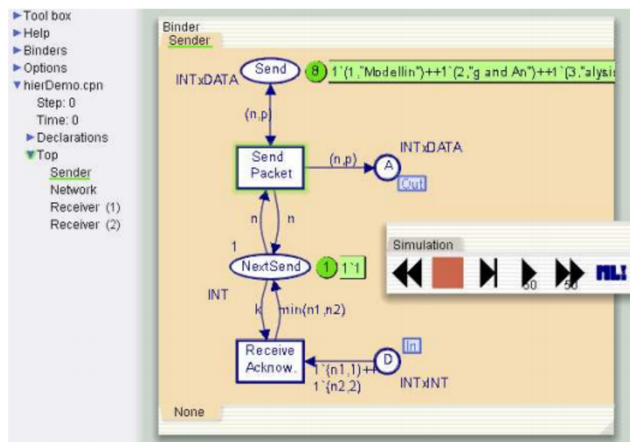
CPN Tools podpira dva tipa analize in sicer na podlagi *simulacije* in *analize prostora stanj*. Obe orodji najdemo v že znanem Tool Box predelu.

Simulacija Simulacijo izvajamo s pomočjo simulacijskega orodja imenovanega *Simulation Tools*. Omogoča izvajanje po korakih. Izvedemo lahko en korak, pri čemer se izvede ena akcija. Izvedemo lahko tudi vnaprej definirano število korakov, ki jih uporabnik poda sam, pri tem pa se grafični vmesnik posodobi ob vsakem koraku. Zadnja možnost je izvajanje v hitrem načinu. Deluje enako kot v navadnem načinu, le da se grafični vmesnik posodobi šele ob koncu izvajanja simulacije.

Povratna informacija pri simulaciji je predstavljena z zeleno barvo, kar se vidi na sliki 3.11. Vse akcije, ki imajo zeleno obrobo, so *omogočene*, kar pomeni da se lahko izvedejo, saj so vhodni pogoji izpolnjeni. V bližnji okolici pogojev je v zelenih krogcih predstavljeno število žetonov v pogoju, zraven pa je v zelenih pravokotnikih predstavljena trenutna označitev pogoja.

Analiza prostora stanj To je še ena izmed možnosti analize CPN v orodju CPN Tools. Za izvajanje analize prostora stanj je pomembno, da imajo vse akcije in pogoji unikatna imena, če ne bi prišlo do inkonsistence pri analizi.

Analizo izvajamo s pomočjo orodja imenovanega *State Space Tools*. Vsebuje orodja, ki so potrebna za generiranje kode, ki bo nadalje poskrbela za generiranje prostora stanj.



Slika 3.11 Prikazan je način povratne informacije med simulacijo. Akcija "Send packet" je omogočena, zato je obarvana zeleno. Ob pogojih "Send" in "Next send" je še informacija o številu žetonov in trenutni označitvi [24].

Prisotna so tudi orodja, ki sam prostor stanj dejansko kreirajo (*CalcSS*) in orodja za ustvarjanje grafa povezav (*CalcSCC*). Tukaj bi opomnili tudi na razliko med simulacijskim pristopom analize in analizo prostora stanj: simulacijska koda nastaja sproti z gradnjo modela sistema, kodo za generiranje prostora stanj pa je potrebno kreirati posebej šele potem, ko imamo model že postavljen.

S tem zaključujemo predstavitev orodja CPN Tools in orodij za delo z CPN. Za zaključek povejmo še, da je orodje CPN Tools prosto dostopno s spletne strani `http://cpntools.org/` za Windows, Linux in Mac platforme. V bližnji prihodnosti pa naj bi bilo na voljo v celoti kot odprtokodno orodje.

4 Modeliranje in analiza vzorčnega protokola

V tem poglavju si bomo ogledali model primera problematike *oblikovanja prometa* (angl. *traffic shaping*). Oblikovanje prometa predstavlja prilagajanje povprečne hitrosti podatkovnega prenosa [25]. Zamašitve v omrežju se pogosto pojavijo zato, ker podatki potujejo v *rafalih* (angl. *burst*). Do tega ne bi prihajalo, če bi *vozlišča* oz. *gostitelji* (angl. *hosts*) pošiljali podatke z neko konstantno hitrostjo.

Ko se vzpostavi povezava se uporabnik in omrežje dogovorita, kako bo med njima potekal promet. V kolikor se uporabnik drži pravil, ki sta si jih zastavila oba, potem bo tudi omrežje dostavilo vse podatke (pakete) v nekem predvidenem času. S tem dogovorom se učinkovito izognemo zamašitvam v omrežju. Dogovori takega tipa so še posebej pomembni pri prenosu podatkov v realnem času, kot so audio in video podatki. Realno-časovni podatki zelo slabo tolerirajo zamašitve. V primeru, da do njih pride, to uporabnik občuti kot izgubo slike, zvoka, trganja slike, itd.

4.1 Opis izbranega protokola

V tem razdelku bosta predstavljena dva sorodna protokola za oblikovanje prometa po omrežju. To sta *algoritem puščajočega vedra* (angl. *leaky bucket algorithm*) in *algoritem vedra z žetoni* (angl. *token bucket algorithm*). Algoritem puščajočega vedra je predstavljen zgolj za potrebo primerjave z algoritmom vedra z žetoni.

4.1.1 Leaky Bucket algoritem

Kot že pove samo ime, si lahko ta algoritem predstavljamo kot vedro, ki ima na dnu majhno luknjo. Ne glede na to, s kakšno hitrostjo voda priteka v vedro, bo iz njega kapljala skozi luknjo z neko konstantno hitrostjo, če je voda prisotna oz. s hitrostjo 0, če vode v vedru ni. Težava se pojavi, ko je vedro polno, voda pa še vedno priteka. V tem primeru se odvečna voda razlije čez rob vedra [25].

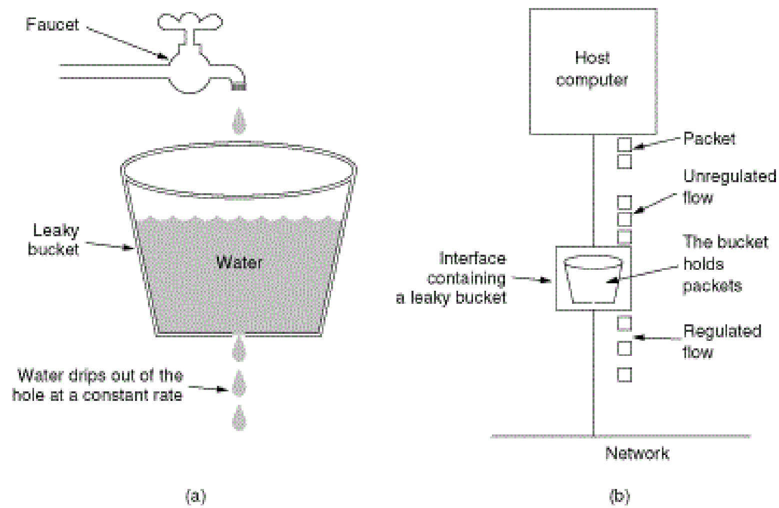
Enak princip velja za pakete, ki potujejo po omrežju. Vedro je v omrežju predstavljeno kot končna čakalna vrsta z enim strežnikom, v katero prihajajo paketi iz omrežja. Strežba se izvaja s konstantno hitrostjo. Na vsako urino periodo [25] bo v omrežje poslan natanko en paket. Če paket prispe do čakalne vrste, ko je ta polna, bo zavržen. V nasprotnem primeru se vanjo uvrsti. To je ena izmed slabosti tega algoritma. Primer delovanja algoritma je prikazan na sliki 4.1.

Gostitelj lahko pošlje v omrežje le en paket na vsako urino periodo. Na ta način se neenakomeren pritok paketov spremeni v enakomeren odtok paketov. S tem zmanjšamo možnost pojava zamašitve v omrežju.

V omrežjih se ponavadi pojavljajo paketi različnih velikosti. V tem primeru spremenimo algoritem puščajočega vedra tako, da namesto le enega paketa omogoča pretok določenega števila bajtov vsako urino periodo. Če dovolimo prenos 1024 bajtov se lahko prenese en paket velikosti 1024 bajtov, dva paketa velikosti 512 bajtov, itd. Če je velikost naslednjega paketa večja od preostale kapacitete za prenos, mora ta paket počakati na naslednjo urino periodo.

4.1.2 Token Bucket algoritem

Za mnoge aplikacije je bolje, da dovolimo povečanje izhodne hitrosti, ko pride do prihoda večjega števila paketov v rafalih. To nam omogoča algoritem vedra z žetoni. V vedru se nahajajo žetoni, ki jih generira ura vsakih ΔT sekund. Da paket lahko odpotuje naprej,

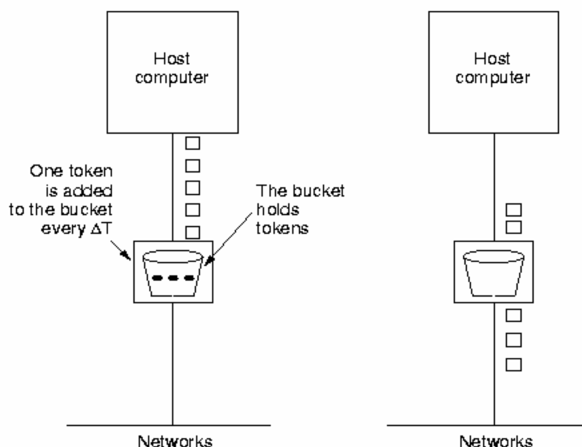


Slika 4.1 Na sliki (a) je prikazan primer iz realnega življenja. Iz vedra s konstantno hitrostjo kaplja voda, ne glede na to s kakšno hitrostjo priteka v vedro. Na sliki (b) pa je prikazan še primer s paketi v omrežju. Paketi prihajajo z neenakomerno hitrostjo, odhajajo pa enakomerno porazdeljeno [25].

mora iz vedra vzeti en žeton in ga uničiti [25]. Pri tem se količina žetonov v vedru zmanjša za ena. Primer je prikazan na sliki 4.2.

Vedro z žetoni predstavlja drugačno oblikovanje prometa, kot puščajoče vedro. Ko je vedro z žetoni nedejavno (ni paketov za pošiljanje), lahko shranjuje žetone, ki se bodo kasneje uporabili za pošiljanje paketov. Na ta način se bo lahko določeno število paketov poslalo takoj, ko pridejo do vedra, saj jim ne bo potrebno čakati, da se žetoni zgenerirajo. Število žetonov, ki se lahko shranijo, je omejeno s kapaciteto vedra C. Puščajoče vedro ne omogoča shranjevanja, ko paketov za pošiljanje ni, zato v tem času na opravlja nobene naloge.

Tudi tu je možno žetone predstaviti tako, da omogočajo pošiljanje k bajtov namesto enega paketa. Ob vsaki urini periodi (vsakih ΔT sekund) se v vedro doda žeton, ki omogoča prenos k bajtov podatkov. Paket se pošlje, ko je na voljo zadostno število žetonov, ki pokrijejo njegovo velikost. Možna je tudi delna poraba žetona. V kolikor se za uspešno pošiljanje paketa potrebuje le del žetona, se neporabljeni del ohrani za bodoče prenose (npr. za uspešen prenos potrebujemo le $k/2$ bajtov, preostanek, torej polovica žetona, pa se ohrani).



Slika 4.2 Primer algoritma vedro z žetoni. V vedru se nahajajo trije žetoni. To omogoča trem paketom, ki čakajo v čakalni vrsti, da odpotujejo naprej v omrežje [25].

4.2 Postavitev modela

Za vzorčni primer postavitve modela in analize smo si izbrali algoritem vedra z žetoni. Zgradili smo ga v orodju CPN Tools. Za to orodje smo se odločili zato, ker ponuja množico verifikacijskih postopkov, kar je tudi cilj te diplomske naloge: postaviti model in ga verificirati. S pomočjo orodij za verificiranje lahko na modelu preverimo dosegljivost stanj, mrtve zanke, omejenost, živost in pravičnost modela. Dodatno lahko tudi izrišemo prostor stanj za naš model in v njem opravljamo poizvedbe za prej omenjene verifikacijske postopke nad poljubnimi stanji.

Da bo naš model uspešno deloval, moramo vpeljati določene deklaracije, s katerimi določimo barvne tipe, spremenljivke in funkcije. Deklaracije so vidne na sliki 4.3. Barvni tip (*angl. color set*) je v deklaraciji označen kot *colset*. Barvni tip INT predstavlja cela števila, TOKEN predstavlja klasične črne žetone, ki ne prenašajo podatkov, barvni tip smallREAL pa predstavlja realna števila. Za uporabo teh barvnih tipov nato deklariramo še spremenljivke, ki jim določimo enega izmed barvnih tipov (spremenljivke n, k, i, sR) ali pa direktno neko vrednost (spremenljivka c). Med deklaracijami lahko definiramo tudi funkcije. V našem primeru sta to funkciji delay() in fillRate(), ki sta razloženi v nadaljevanju.

Tukaj bi razložili še naslednje rezervirane besede, ki se pojavijo v naših deklaracijah. Prva je beseda timed, s katero barvni tip označimo dodatno še za časovnega. To nam

```
▼TokenBucket.cpn
   Step: 0
   Time: 0
  ▶Options
  ▶History
  ▼Declarations
    ▶Standard priorities
    ▶Standard declarations
    ▼colset INT = int;
    ▼colset TOKEN = unit timed;
    ▼colset smallREAL = real with 0.0..1.0;
    ▼var n,k,i : INT;
    ▼var sR : smallREAL;
    ▼val c = 10;
    ▼fun fillRate() = 4;
    ▼fun delay() = poisson(4.0);
```

Slika 4.3 Spisek deklaracij, ki se uporabljajo v modelu algoritma z žetoni. Vsebuje barvne tipe, spremenljivke in funkcije.

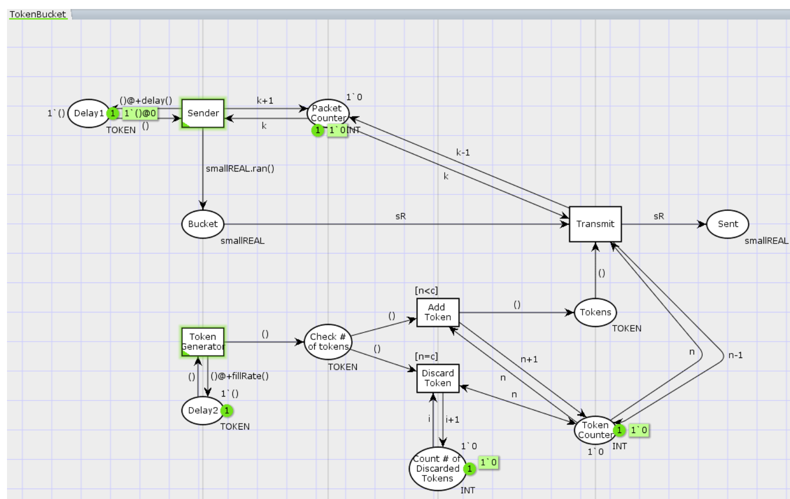
omogoči uporabo časa v našem modelu. Druga beseda je with, s katero označimo barvni tip za majhnega (angl. *small color set*). To pomeni le, da je obseg barvnega tipa omejen na nek določen interval. V našem primeru to obvezno potrebujemo, saj CPN Tools omogoča uporabo naključne funkcije ran() (random funkcija) le z majhnimi barvnimi tipi.

V našem modelu, ki je prikazan na sliki 4.4, pošiljamo naključna realna števila. Ta števila se generirajo s pomočjo akcije *Sender* in potujejo po povezavi v čakalno vrsto oz. vedro, ki je predstavljeno kot pogoj *Bucket*. Na povezavi od akcije *Sender* do pogoja *Bucket* vpeljemo funkcijo ran(), ki nam zgenerira naključno realno število v obsegu od 0.0 do 1.0. Generiranje realnih števil je porazdeljeno po Poissonovi porazdelitvi. To smo dosegli z vpeljavo pogoja *Delay1*. Na povezavi, ki pelje iz akcije *Sender* v pogoj *Delay1* smo določili časovni zamik s pomočjo funkcije poisson(). Kot argument ji podamo število pričakovanih paketov, ki pridejo do vedra v časovni enoti. Ob vsakem generiranju realnega števila prav tako povečamo števec paketov, ki ga predstavlja pogoj *Packet Counter*.

Da se paketi oz. realna števila sploh lahko pošljejo, morajo ob vsakem pošiljanju iz vedra vzeti en žeton in ga uničiti. Pri algoritmu vedro z žetoni se žetoni generirajo z določeno frekvenco (npr. r žetonov v sekundi). Ker izvajanje simulacije v CPN Tools ne podpira realnega časa temveč le simulacijskega, smo pri naši izbiri frekvence generiranja bili primorani skleniti kompromis. Žetoni se generirajo v akciji *Token Generator* na

vsakih t časovnih korakov. To smo dosegli z vpeljavo pogoja *Delay2*. Na povezavi od akcije *Token Generator* do pogoja *Delay2* smo vpeljali novo funkcijo fillRate(), ki vsakih t časovnih korakov zgenerira nov žeton. Ko je žeton zgeneriran se prenese v pogoj *Check # of Tokens*, ki preveri ali je kapaciteta vedra že dosežena. Kapaciteto smo omejili na c žetonov, kjer je c pozitivno naravno število. Tukaj nastopita dva scenarija, ki jih ponazorimo z uporabo čuvajev. Vpeljali smo akcijo *Add Token* s čuvajem $[n < c]$ in akcijo *Discard Token* s čuvajem $[n{=}c]$. Število n predstavlja trenutno število žetonov v vedru, c pa kapaciteto vedra. V kolikor vedro še ni polno ($n < c$), se izvede akcija *Add Token* in žeton se doda v vedro oz. v pogoj *Tokens*, kjer hranimo vse žetone, ki so že na voljo v vedru. Obenem se poveča števec žetonov *Token Counter* za 1. Če pa je vedro polno ($n = c$), se izvede akcija *Discard Token*. V tem primeru žeton preprosto zavržemo. Pogoj *Count # of Discarded Tokens* služi zgolj informativno in šteje število zavrženih žetonov. Na tem mestu bi opozorili na naslednjo stvar. V našem modelu se nahajata pogoja *Tokens* in *Token Counter*. Za pravilno delovanje modela pogoj *Tokens* ni potreben, saj število žetonov štejemo tudi v pogoju *Token Counter*. Ker smo modelirali algoritem vedro z žetoni, smo želeli imeti pogoj, ki bi hranil dejanske žetone. S tem želimo omogočiti bralcu lažje razumevanje.

V tem trenutku je vse nared za pošiljanje. Kot smo predhodno povedali, se števila hranijo v pogoju *Bucket*, žetoni pa v pogoju *Tokens*. Sledi jima akcija *Transmit*, v katero peljejo povezave iz pogojev *Bucket*, *Tokens*, *Packet Counter* in *Token Counter*. Da se pošlje eno število, moramo za to imeti v pogoju *Tokens* vsaj en žeton. Za pošiljanje smo predpostavili, da se izvede hipno, zato tukaj ni nobenega časovnega zamika. V akcijo se zato pripeljeta eno realno število ter en žeton. Na tem mestu tudi zmanjšamo števca paketov in žetonov za 1. Iz akcije *Transmit* pelje le ena povezava v pogoj *Sent*, kjer se hranijo poslana realna števila, ki so odšla naprej v omrežje.



Slika 4.4 Model algoritma vedra z žetoni, postavljen v programskem orodju CPN Tools.

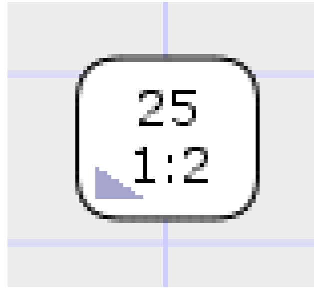
4.3 Analiza modela ob nedeterminističnem generatorju števil

V analizi se bomo posvetili še verificiranju modela. V našem primeru bomo analizirali omejenost mreže modela (angl. *boundedness*), dosegljivost stanj (angl. *reachability analysis*), domače označitve (angl. *home markings*) ter živost modela (angl. *liveness*) [16]. Pri analizi omejenosti modela preverjamo pogoje, ali presegajo največje dovoljene vrednosti števila žetonov v pogojih. To nas zanima za pogoje, kjer določenega števila žetonov ne želimo preseči (npr. pogoj *Tokens*, kjer smo kapaciteto omejili na c žetonov). Domača označitev nam predstavlja označitev, ki je vedno dosegljiva. Pri analizi živosti modela iščemo označitve, ki nimajo nobene omogočene akcije. Takim označitvam pravimo mrtve označitve (angl. *dead markings*). Akcija ni omogočena, ko se ne more sprožiti [16].

Za vse opisane postopke obstaja v CPN Tools priročno orodje imenovano State Space [17]. S pomočjo tega orodja najprej izračunamo prostor stanj, nato pa še graf povezanih komponent (angl. *Strongly Connected Component Graph - SCC Graph*). Izračun tega grafa je pomemben za pridobitev informacije o živosti modela in domači označitvi. Naslednji korak je generiranje poročila, ki vsebuje podatke o lastnostih modela, ki smo jih predstavili. Na podlagi poročila analiziramo model.

V orodju State Space obstajajo še dodatne funkcije za bolj podrobno analizo:

- Prikaz poljubnega stanja v modelu (npr. mrtvo označitev): To nam omogoči orodje State Space to Simulator. Na podlagi podanega stanja model prikaže trenutno



Slika 4.5 Vozlišče, ki predstavlja eno stanje v programu CPN Tools. Zgornje število predstavlja indeks opazovanega stanja, spodnji dve pa število stanj na predhodnem koraku in število stanj na naslednjem koraku.

označitev mreže v tem stanju.

- Izris prostora stanj z orodjem Display State Space: Izris prostora stanj lahko začnemo iz poljubnega stanja, nato pa izrisujemo predhodnike in naslednike tega stanja. Stanje je predstavljeno kot vozlišče s tremi števili (glej sliko 4.5). Vsako vozlišče vsebuje opis trenutne označitve, povezava pa opis elementa v katerega potuje.
- Uporaba poizvedb v prostoru stanj: Poizvedbe uporabljamo za preiskovanje dosegljivosti, omejenosti, živosti in domače označitve modela. Izvedemo jih preprosto z uporabo pomožnega teksta, ki ga vpišemo na poljubno mesto znotraj strani, v kateri se nahaja naš model. Nato nad tekstom uporabimo orodje Evaluate ML, ki nam vrne informacijo, ki smo jo želeli. Poudarili bi, da se večina poizvedb opravi že med samim računanjem prostora stanj, rezultati pa se zapišejo v poročilo. Za opisovanje poizvedb uporabljamo nabor funkcij, ki so na voljo v programu CPN Tools.

Verificiranje z opisanimi postopki bomo sedaj prikazali na primeru našega modela vedro z žetoni. Na tem mestu želimo opomniti še na eno izmed opcij orodja State Space. Naš model vedra z žetoni je z vidika generiranja žetonov, ki so ključni del za pošiljanje paketov, in realnih števil neskončen, saj se žetoni in paketi generirajo ves čas z določenimi zakasnitvami. Zato bomo uporabili možnost zaustavitve izračunavanja prostora stanj s pomočjo pogoja. Kot pogoj za ustavitev izračuna prostora stanj bomo določili čas, ki bo namenjen simulaciji. Ko bo čas pretekel, se bo simulacija ustavila. Čas, ki ga podamo, predstavlja realni čas in ne simulacijskega, tako kot pri izvajanju simulacije. Analizo bomo opravljali s pomočjo poročila, ki nam ga zgenerira CPN Tools.

Pred samo analizo je bilo potrebno nastaviti še določene parametre, ki bistveno vplivajo na izvajanje našega modela. To so kapaciteta vedra, frekvenca generiranja žetonov in intenzivnost prihajanja paketov. Kapaciteto vedra smo omejili na 10 žetonov, žetoni se generirajo na vsake 4 časovne korake, paketi pa v povprečju prihajajo prav tako na vsake 4 časovne korake. Simulaciji smo namenili 30 sekund. Pakete za pošiljanje generiramo s pomočjo nedeterminističnega generatorja števil.

4.3.1 Statistika

V statistiki najprej dobimo podatke o poteku izračunavanja prostora stanj (glej sliko 4.6). V našem primeru opazimo, da je bilo izračunano 14573 vozlišč med katerimi poteka 16535 povezav. Parameter *secs* pove, koliko časa je bilo namenjeno izračunu prostora. Pomemben podatek se nahaja v parametru *status*, kjer je navedeno *partial*. Status *partial* smo dobili zato, ker je potekel ves čas namenjen računanju (30 sekund), v tem času pa ni uspel celotni izračun prostora stanj. V kolikor bi izračunali celoten prostor stanj, bi dobili status *full*. Pridobili smo tudi podatke o grafu povezanih komponent. Rezultati so identični prostoru stanj, kar pomeni, da v našem modelu nimamo nobenega cikla [16].

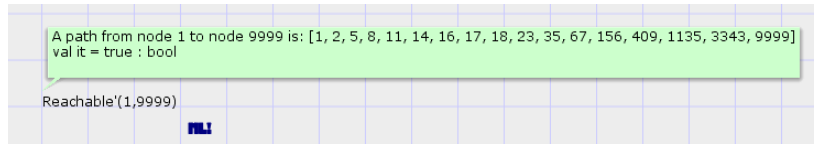
```
Statistics
---------------------------

 State Space
    Nodes:  14355
    Arcs:   16014
    Secs:   30
    Status: Partial

 Scc Graph
    Nodes:  14355
    Arcs:   16014
    Secs:   1
```

Slika 4.6 Statistika pri računanju prostora stanj za Petrijevo mrežo algoritma vedro z žetoni.

4.3.2 Dosegljivost stanj

Dosegljivost modela preverjamo s pomočjo poizvedb. V ta namen imamo na voljo že definirane funkcije, kot je npr. Reachable(x,y). Za x in y podamo dve vozlišči, nato pa funkcijo evaluiramo s pomočjo orodja Evaluate ML. V našem primeru želimo preveriti ali obstaja pot od začetnega vozlišča do vozlišča, ki predstavlja mrtvo označitev. Za ta namen uporabljamo funkcijo Reachable'(). Funkciji smo za imenom dodali še ', kar nam

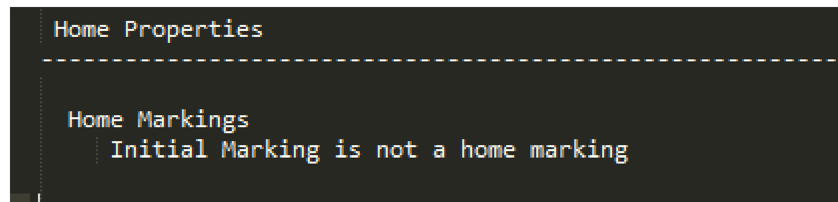
vrne lepše izpisan rezultat. Rezultat poizvedbe je prikazan na sliki 4.7.



Slika 4.7 Primer poizvedbe za preverjanje dosegljivosti stanj. Zanima nas, ali obstaja pot od začetnega vozlišča do ene izmed mrtvih označitev. Poizvedba vrne rezultat true in izpiše pot, ki jo je potrebno opraviti, da pridemo do izbrane mrtve označitve.

4.3.3 Domače označitve

Domača označitev je označitev, ki je vedno dosegljiva iz kateregakoli dosegljivega stanja [16]. Ker izračun prostora stanj ni uspel v celoti, ne dosežemo nobene domače označitve (glej sliko 4.8). Tukaj tiči težava delnega prostora stanj, saj bi za domačo označitev pričakovali označitev, pri kateri se izvajanje algoritma ustavi. Ker je prostor stanj našega modela neskončen, do tega nikoli ne pride.



Slika 4.8 Mreža ne vsebuje nobene domače označitve. Začetno stanje prav tako ni domače stanje, saj se vanj ne moremo vrniti.

4.3.4 Omejenost

Pri omejenosti želimo preveriti predvsem pogoje, katerim smo omejili kapaciteto. Na sliki 4.9 so vidni podatki o največjem in najmanjšem številu žetonov v posameznih pogojih. Vzemimo pod drobnogled pogoja *Delay1* in *Delay2*. Služita le kot zakasnitev za izvajanje akcij in zato želimo, da je v njem prisoten vedno le en žeton. V primeru, da bi bil v pogojih več kot en žeton, bi se akcije prožile ena za drugo, česar naš algoritem ne predvideva. V poročilu je vidno, da sta največje in najmanjše število žetonov vedno 1, kar je bila naša zahteva. Poglejmo si še pogoj *Tokens*. Na začetku smo povedali, da smo mu omejili kapaciteto na $c = 10$ žetonov. V poročilu vidimo, da je največje število žetonov v tem pogoju bilo 4, kar sovpada z našo zahtevo največ 10 žetonov.

```
Boundedness Properties
------------------------------------------------

 Best Integer Bounds
                             Upper      Lower
    TokenBucket'Bucket 1     6          0
    TokenBucket'Check 1      9          0
    TokenBucket'Count 1      1          1
    TokenBucket'Delay1 1     1          1
    TokenBucket'Delay2 1     1          1
    TokenBucket'Packet_Counter 1
                             1          1
    TokenBucket'Sent 1       3          0
    TokenBucket'Token_Counter 1
                             1          1
    TokenBucket'Tokens 1     4          0
```

Slika 4.9 Podatki o omejenosti vseh pogojev v mreži modela. Nobeden izmed omejenih pogojev ne preseže dovoljene kapacitete. Dosegli smo željene rezultate.

```
     TokenBucket'Check 1 9`()
     TokenBucket'Count 1 1`0
     TokenBucket'Delay1 1
                         1`()
     TokenBucket'Delay2 1
                         1`()
     TokenBucket'Packet_Counter 1
                         1`0++
1`1++
1`2++
1`3++
1`4++
1`5++
1`6
    TokenBucket'Token_Counter 1
                             1`0++
1`1++
1`2++
1`3++
1`4
     TokenBucket'Tokens 1
                         4`()
```

Slika 4.10 Omejenost pogojev pri največjem številu žetonov, kjer je prikazana njihova dejanska vsebina.

Analizo omejenosti pogojev smo do tega trenutka opravili le na podlagi števila žetonov v pogojih, ne pa tudi na podlagi vsebine žetonov v pogojih. Našo predstavitev si lahko razširimo tako, da si ogledamo, kaj dejansko se v pogojih nahaja. Primer je predstavljen na sliki 4.10, kjer je prikazana vsebina pogojev pri največjem številu in na sliki 4.11, ki prikazuje vsebino pogojev pri najmanjšem številu žetonov. Vzemimo pod drobnogled pogoj *Packet Counter*. Na sliki 4.9 vidimo, da je največje število žetonov v pogoju *Bucket* bilo 6. Torej mora za pogoj *Packet Counter* veljati, da ima največjo vrednost števila v pogoju 6, najmanjšo pa 0. Obe trditvi držita, kar lahko vidimo na slikah 4.10 in 4.11.

```
    TokenBucket'Token_Counter 1
                        1`0++
1`1++
1`2++
1`3++
1`4
    TokenBucket'Tokens 1
                        4`()

  Best Lower Multi-set Bounds
    TokenBucket'Bucket 1
                        empty
    TokenBucket'Check 1 empty
    TokenBucket'Count 1 1`0
    TokenBucket'Delay1 1
                        1`()
    TokenBucket'Delay2 1
                        1`()
    TokenBucket'Packet_Counter 1
                        empty
    TokenBucket'Sent 1  empty
    TokenBucket'Token_Counter 1
                        empty
    TokenBucket'Tokens 1
                        empty
```

Slika 4.11 Omejenost pogojev pri najmanjšem številu žetonov, kjer je prikazana njihova dejanska vsebina.

4.3.5 Živost

Pri preverjanju živosti modela imamo podatke o mrtvih označitvah, mrtvih akcijah in živih akcijah. Izpis iz poročila je predstavljen na sliki 4.12. V našem primeru ima model mrtvo akcijo *Discard Token*, kar pomeni, da se ni nikoli izvedla.

V modelu se nahaja 10020 mrtvih označitev. Vsak žeton v našem modelu dobi ob kreiranju časovni žig, ki predstavlja del označitve. Zaradi množice časovnih žigov dobimo tako veliko količino mrtvih označitev.

S pomočjo orodja State Space to Simulator smo nato preverili kakšna je dejanska označitev mrtvih označitev. Nobena izmed mrtvih označitev ni predstavljala uspešnega zaključka algoritma, kar je bilo tudi pričakovano, saj se naš model izvaja neskončno. Razlog za take mrtve označitve je potek časa, ki smo ga namenili simulaciji (30 sekund).

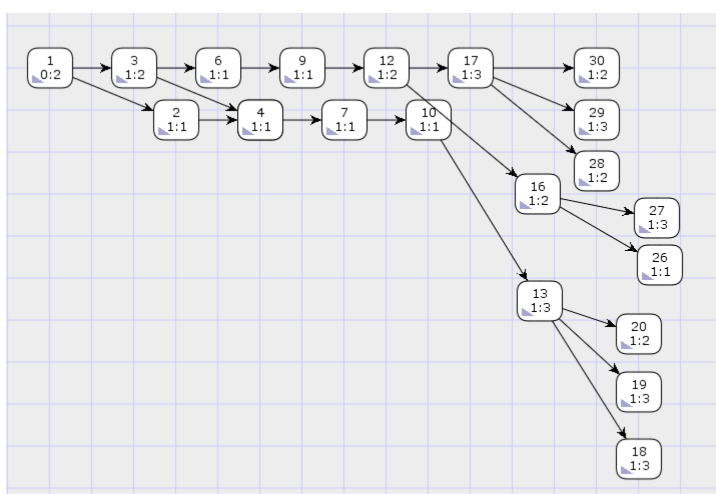
Živost lahko dodatno preverimo še s poizvedbami. Na voljo so nam funkcije kot so DeadMarking(), ki za podano označitev pove ali je mrtva, ListDeadTIs(), ki izpiše vse mrtve akcije, ListDeadMarkings(), ki izpiše vse mrtve označitve, itd.

4.3.6 Prostor stanj

Za konec želimo prikazati še začetni del prostora stanj našega modela. Delni prostor stanj je viden na sliki 4.13. Kot prvo vozlišče je predstavljeno začetno vozlišče s številko 1. V opisu vozlišča vidimo, da prednikov nima, ima pa dva naslednika, to sta vozlišči številka 2 in 3. Pri obeh se število prednikov ujema, torej en prednik in sicer vozlišče

```
Liveness Properties
------------------------------------------------------------

 Dead Markings
    10020 [9999,9998,9997,9996,9995,...]

 Dead Transition Instances
    TokenBucket'Discard_Token 1

 Live Transition Instances
    None
```

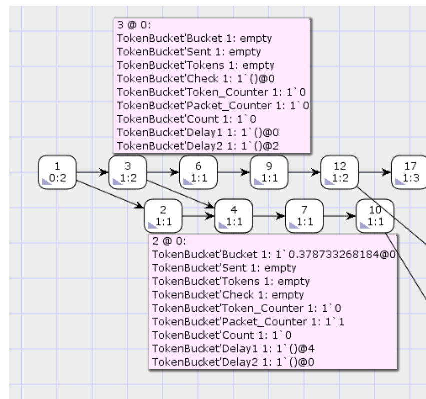
Slika 4.12 Podatki o živosti modela vsebujejo spisek mrtvih označitev, mrtvih akcij in živih akcij. Pri mrtvih označitvah imamo najprej izpisano število mrtvih označitev, v oglatih oklepajih pa še stanja, ki predstavljajo mrtve označitve.



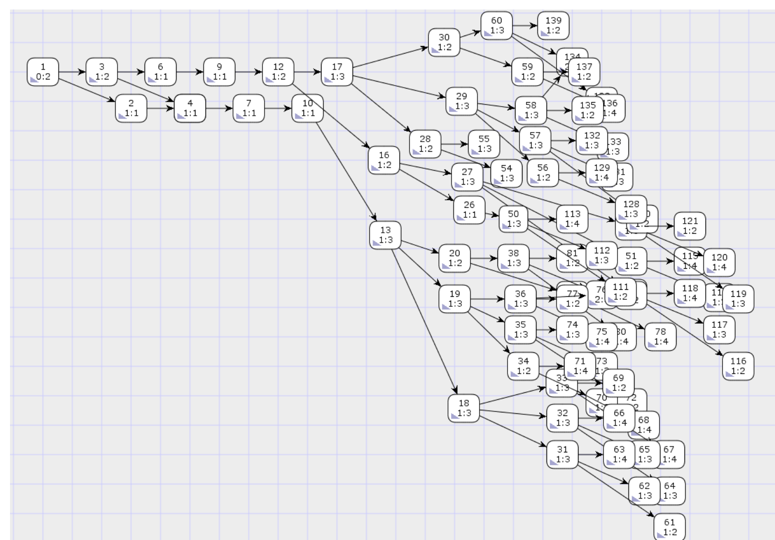
Slika 4.13 Prikaz delno razvitega prostora stanj za algoritem vedro z žetoni.

številka 1. Vsako vozlišče si lahko podrobneje ogledamo s klikom na sivi trikotnik v levem spodnjem kotu vozlišča. Pri vozlišču 2 vidimo, da se je zgeneriral en paket za pošiljanje (eno realno število), ki se je prenesel v pogoj *Bucket*, pri vozlišču številka 3 pa en žeton, ki se trenutno nahaja v pogoju *Check # of Tokens*, kjer čaka na preverjanje zasedenosti pogoja *Tokens*. Oba primera sta vidna na sliki 4.14 v roza oblačkih.

Eden izmed problemov, ki se pojavlja pri analizi prostora stanj, je eksplozija stanj. Število stanj narašča zelo hitro (eksponentno), zato analiza postane zelo otežena že po nekaj korakih. Primer na sliki 4.15 je nadaljevanje primera iz slike 4.13. Dodatno smo razvili še nekaj stanj in že prišli do občutno povečanega števila stanj.



Slika 4.14 Prikaz dodatne informacije za stanji številka 2 in 3.



Slika 4.15 Prikazan je primer eksplozije stanj, kjer število stanj narašča eksponentno. Kljub dodatnem razvoju le nekaj stanj pride zelo hitro do velike nepreglednosti.

4.3.7 Težave nedeterminizma

Petrijeve mreže, ki vsebujejo nedeterministične dele, niso primerne za analizo prostora stanj [26]. V CPN Tools povzročijo nedeterministično obnašanje pri izvajanju naslednji primeri:

- funkcija ran() za generiranje časovnih zamikov in naključnih števil,
- naključne porazdelitvene funkcije za generiranje časovnih zamikov in naključnih števil.

Vsakič, ko izračunamo prostor stanj, bo orodje State Space uporabilo različne vrednosti za naključne porazdelitvene funkcije. Iz tega lahko izpeljemo, da bomo vedno dobili drugačen prostor stanj [26]. Poglejmo si, kje v našem modelu naletimo na omenjene težave. Oba predhodno omenjena primera se pojavita tudi v pri nas. Funkcijo ran() uporabljamo pri generiranju realnih števil, ki jih nato pošiljamo po omrežju. Uporabljamo tudi naključno porazdelitveno funkcijo poisson() in sicer pri generiranju časovnega zamika prihajanja paketov v vedro oz. v pogoj *Bucket*. Ob večkratni uporabi orodja State Space dobimo različen prostor stanj. Za test smo trikrat izračunali prostor stanj. Pričakovano so bili vsi različni po velikosti in številu povezav, kar je prikazano na sliki 4.16.

```
Statistics
------------------------
State Space
   Nodes:  14103
   Arcs:   15644
   Secs:   30
   Status: Partial

Scc Graph
   Nodes:  14103
   Arcs:   15644
   Secs:   1

Statistics
------------------------
State Space
   Nodes:  14103
   Arcs:   15644
   Secs:   30
   Status: Partial

Scc Graph
   Nodes:  14103
   Arcs:   15644
   Secs:   1

Statistics
------------------------
State Space
   Nodes:  14103
   Arcs:   15644
   Secs:   30
   Status: Partial

Scc Graph
   Nodes:  14103
   Arcs:   15644
   Secs:   1
```

Slika 4.16 Na prvotnem modelu smo trikrat izračunali prostor stanj in trikrat dobili različen prostor stanj. Razlog za to je nedeterminizem, ki je v naši mreži predstavljen z uporabo funkcij ran() in poisson().

Drugi problem, ki ga ima naš model, je ta, da je neskončen z vidika generiranja žetonov in realnih števil za pošiljanje. Zaradi tega smo bili primorani omejiti simulacijo na 30 sekund. Posledica tega je, da ne moremo izračunati celotnega prostora stanj. Zaradi nepopolnega prostora stanj se zato pojavijo določene težave.

- Domača označitev: Če bi bil naš model končen, bi pričakovali, da bo domača označitev tista, v kateri se izvajanje algoritma ustavi. V primeru več mrtvih označitev domače označitve ni [16].

- Dosegljivost stanj: Obstajati mora pot iz začetnega stanja do stanja, v katerem se izvajanje algoritma zaključi [16].

- Mrtve označitve: Če bi imeli končen model, bi mrtva označitev morala biti predstavljena kot označitev, pri kateri ni omogočene nobene akcije. Predpostavljamo, da je to označitev, pri kateri smo poslali vse podatke in se je algoritem uspešno zaključil [16].

4.4 Odprava nedeterminizma in rezultati

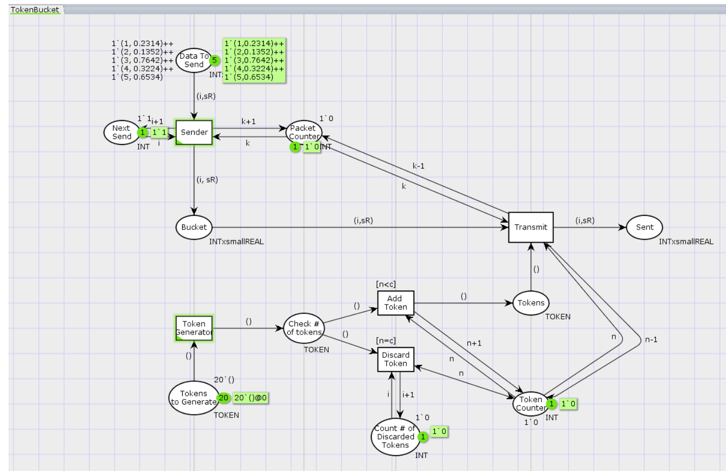
Rešitev težav, opisanih v prejšnjem razdelku, bomo predstavili v tem razdelku. Prva stvar, ki jo je potrebno urediti je ta, da omejimo izvajanje našega algoritma, torej, da ga naredimo za končnega. V realnosti vedro z žetoni tega ne predvideva, saj konstantno generira žetone. V našem primeru smo to primorani spremeniti, če želimo dobiti končen prostor stanj. Izvajanje algoritma bomo omejili tako, da bomo generatorju žetonov predpisali število žetonov, ki mu jih je dovoljeno generirati. To dosežemo z vpeljavo novega pogoja *Tokens to Generate*, ki vsebuje predpisano število žetonov. Podatke, ki jih bomo pošiljali po omrežju bomo predhodno inicializirali v novem pogoju *Data to Send*. Podatki so v tem primeru sestavljeni malo drugače, kot v prejšnjem primeru. Dodali smo jim še zaporedno številko, ki bo služila pošiljanju v vrstnem redu, zato smo na tem mestu vpeljali še tretji pogoj *Next Send*, ki vsebuje število naslednjega paketa, ki je na vrsti za pošiljanje. Na ta način smo se znebili funkcije ran(), ki je predstavljala nedeterminizem v našem modelu. Naslednja stvar je, da odstranimo funkcijo poisson(), ki nam prav tako predstavlja nedeterminizem v mreži. Najdemo jo v sklopu funkcije delay(). S tem smo se znebili še zadnjega primera nedeterminizma. Ker smo odstranili časovni zamik pri prihajanju paketov v vedro, odstranimo tudi časovni zamik pri generiranju žetonov (fillRate() funkcija).

Če povzamemo popravljeno mrežo vidimo, da smo se znebili obeh pogojev *Delay1* in *Delay2*, namesto njiju pa imamo nove pogoje *Tokens to Generate*, ki generira žetone, *Data to Send*, ki predstavlja podatke, ki se bodo pošiljali in *Next Send*, ki pove, kateri paket je naslednji na vrsti za pošiljanje. Model popravljene mreže je viden na sliki 4.17.

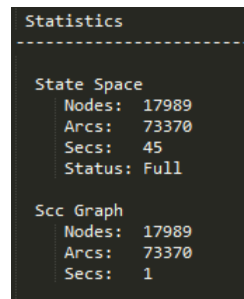
Rezultate analize bomo ponovno interpretirali s pomočjo poročila, ki se generira pri računanju prostora stanj. Opozorili bomo le na pomembne razlike, ki se pojavijo glede na prejšnjo analizo. Prva razlika se vidi že v statistiki, saj je parameter status tokrat enak *full* (glej sliko 4.18), kar pomeni, da se je prostor stanj izračunal v celoti.

Definicija domače označitve [16] pravi, da je domača označitev označitev, ki je dosegljiva iz katerekoli dosegljive označitve. V našem primeru imamo šest mrtvih označitev (glej sliko 4.22), ki pa med seboj niso dosegljive. Iz tega sledi, da nobena izmed mrtvih označitev ne more biti domača označitev. Posledično naš model nima domače označitve (glej sliko 4.19).

Pri omejenosti bi pod drobnogled vzeli pogoj *Tokens*. Ponovno je bila njegova ka-



Slika 4.17 Popravljen model mreže za algoritem vedro z žetoni. Vpeljali smo tri nove pogoje, ki služijo kot medpomnilnik za pakete in žetone in kot indikator naslednjega paketa, ki se bo poslal v omrežje. Znebili smo se dveh pogojev, ki sta služila za časovni zamik. S tem smo se znebili nedeterminizma v naši mreži.



Slika 4.18 Statistika za popravljen model. Tokrat je prostor stanj izračunan v celoti.

paciteta omejena na 10 žetonov. Na sliki 4.20 opazimo, da kapacitete 10 žetonov res ni presegel, kar pomeni, da se omejen pogoj obnaša tako, kot smo želeli.

Pri živosti modela opazimo, da tokrat nimamo nobene akcije, ki bi bila mrtva. To pomeni, da se je tokrat izvedla tudi akcija *Discard Token*, kar lahko preverimo tako, da v omejenosti preverimo pogoj *Count # of Discarded Tokens*. Opazimo, da se v tem pogoju nahaja žeton z vrednostjo 10. Iz tega sklepamo, da se je akcija *Discard Token* izvedla desetkrat (glej sliko 4.21).

Naslednja pomembna stvar pri živosti modela so mrtve označitve, ki jih je šest. Vsa stanja izpišemo s pomočjo funkcije ListDeadMarkings(), s katero opravimo poizvedbo. Kot rezultat dobimo mrtve označitve, ki so vidne na sliki 4.22.

Vsa stanja smo nato šli preveriti z orodjem State Space to Simulator, ki prikaže

```
Home Properties
------------------------------------------------

 Home Markings
    None
```

Slika 4.19 Mreža ne vsebuje nobene domače označitve, saj imamo več mrtvih označitev, ki pa med seboj niso dosegljive.

```
Boundedness Properties
------------------------------------------

 Best Integer Bounds
                          Upper      Lower
    TokenBucket'Bucket 1    5          0
    TokenBucket'Check 1     20         0
    TokenBucket'Count 1     1          1
    TokenBucket'Data_To_Send 1
                            5          0
    TokenBucket'Next_Send 1 1          1
    TokenBucket'Packet_Counter 1
                            1          1
    TokenBucket'Sent 1      5          0
    TokenBucket'Token_Counter 1
                            1          1
    TokenBucket'Tokens 1    10         0
    TokenBucket'Tokens_to_Generate 1
                            20         0
```

Slika 4.20 Podatki o omejenosti vseh pogojev v mreži modela. Pod drobnogled smo vzeli pogoj *Tokens*, ki ima omejeno kapaciteto na 10 žetonov. Te kapacitete ni presegel, kar pomeni da se obnaša tako, kot smo to predvideli.

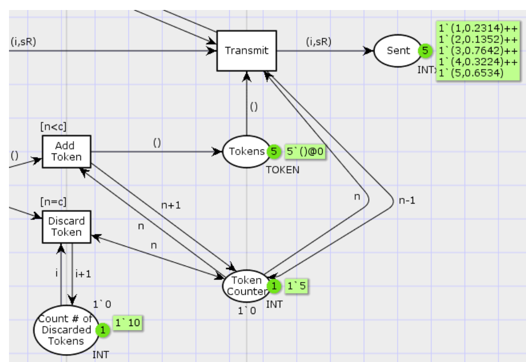
označitev mreže glede na podano stanje. Ugotovili smo, da so vsa stanja, ki jih predstavljajo mrtve označitve, resnično mrtva (nimajo omogočene nobene akcije). Razlog, da jih je šest, tiči v razporedu žetonov v mreži. Podatki, ki smo jih želeli poslati so v vseh šestih primerih bili poslani (se nahajajo v pogoju *Sent*). Pojavilo se je le šest različnih razporeditev žetonov v pogojih *Tokens* in *Count # of Discarded Tokens*, kar je popolnoma pričakovano. To je zaradi narave pošiljanja paketov. Primer: imamo pogoj *Tokens*, v katerem je deset žetonov (kapaciteta je zasedena v celoti). V pogoju *Bucket* se nahaja paket, kar pomeni, da se lahko pošlje. En žeton se prav tako nahaja v pogoju *Check # of Tokens*. V koliko se bo najprej poslal paket z akcijo *Transmit*, se bo žeton v pogoju *Check # of Tokens* uvrstil v pogoj *Tokens* preko akcije *Add Token*, saj se je sprostilo eno mesto v tem pogoju in čuvaj $[n < c]$ omogoči akcijo *Add Token* namesto *Discard Token*. V koliko pa se prej izvede akcija *Discard Token*, bo žeton zavržen. Primer dveh takih označitev je prikazan na slikah 4.23 in 4.24.

```
TokenBucket'Count 1 1`0++
1`1++
1`2++
1`3++
1`4++
1`5++
1`6++
1`7++
1`8++
1`9++
1`10
```

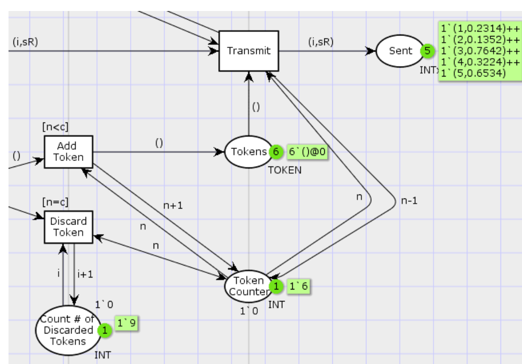
Slika 4.21 Vrednost žetona v pogoju *Count # of Discarded Tokens* doseže 10, kar pomeni da se je akcija *Discard Token* izvedla desetkrat. Mrtvih akcij zato tokrat ni.

```
val it = [17989,17988,17987,17986,17985,17984] : Node list

ListDeadMarkings()
```

Slika 4.22 S pomočjo funkcije ListDeadMarkings() izpišemo vse mrtve označitve.



Slika 4.23 V tem primeru mrtve označitve imamo pet žetonov v pogoju *Tokens*, deset pa v pogoju *Count # of Discarded Tokens*. Podatki so bili poslani v celoti.



Slika 4.24 V tem primeru mrtve označitve imamo šest žetonov v pogoju *Tokens*, devet pa v pogoju *Count # of Discarded Tokens*. Podatki so bili poslani v celoti.

Na tem mestu zaključujemo z analizo popravljenega modela algoritma vedro z žetoni. S popravki smo odpravili težave, ki jih je predstavljal nedeterministični model, in opravili analizo algoritma.

5 Zaključek

V diplomskem delu smo prikazali potek verificiranja izbranega protokola za oblikovanje prometa po omrežju. Pred tem je bilo potrebno model postaviti v obliki grafa barvne Petrijeve mreže v programskem orodju CPN Tools. S pomočjo tega orodja smo nato izvedli postopek verificiranja. Preverili smo vse pomembne lastnosti modela (živost, omejenost, dosegljivost stanj, domače označitve). Na podlagi pridobljenih rezultatov smo odkrili določene napake in probleme obstoječega modela, ki smo jih nato odpravili. Ob ponovnem postopku verificiranja smo ugotovili, da smo se teh napak uspešno znebili. Lahko smo potrdili, da smo model uspešno izdelali, saj se je izvajal tako, kot smo specificirali.

Največ pozornosti smo namenili barvnim Petrijevim mrežam, predstavljene pa so bile tudi nekatere druge metode verificiranja računalniških sistemov in protokolov ter njihova pripadajoča programska orodja. Posameznik ima tako na voljo resnično pestro izbiro pri izbiri metod verificiranja, v nadaljevanju pa se mu še dodatno razširi z izbiro programskega orodja, ki ga bo uporabil. Za CPN Tools lahko rečemo, da je preprost za uporabo, kljub njegovemu netipičnem izgledu. Ponuja veliko možnosti za simulacijo

in verifikacijo. Z njegovo pomočjo smo uspešno rešili tudi zadani problem te diplomske naloge.

Literatura

[1] CADP tools overview, dostopno 24. april, 2015.
url: http://cadp.inria.fr/tools.html

[2] EDT: The Estelle Development Toolset, dostopno 24. april, 2015.
url: http://www-lor.int-evry.fr/edt/

[3] Spin - formal verification, dostopno 24. april, 2015.
url: http://spinroot.com/spin/whatispin.html

[4] CPN-AMI web site, dostopno 28. april, 2015.
url: http://move.lip6.fr/software/CPNAMI/index.html

[5] INA Intergrated Net Analyzer, dostopno 28. april, 2015.
url: http://www2.informatik.hu-berlin.de/ starke/ina.html

[6] J. Bergstra, J. Klop, Process algebra: specification and verification in bisimulation semantics, Math. & Comp. Sci. II 2 (4) (1986) 61–94.

[7] M. V. Bharti, S. Kumar, Survey of network protocol verification techniques, International Journal of Scientific and Research Publications 2 (4) (2012) 228–231.

[8] G. Callou, P. Maciel, D. Tutsch, J. Araújo, J. Ferreira, R. Souza, A Petri net-based approach to the quantification of data center dependability, dostopno 8. maj, 2015.
url: http://www.intechopen.com/books/petri-nets-manufacturing-and-computer-science/a-petri-net-based-approach-to-the-quantification-of-data-center-dependability

[9] S. Christensen, J. B. Jørgensen, L. M. Kristensen, Design/CPN—A computer tool for coloured Petri nets, Springer, 1997.

[10] M. H. V. G. Diertens Bob, Sjouke Mauw, PSF - Process Specification Formalism, dostopno 24. april, 2015.
url: https://staff.fnwi.uva.nl/b.diertens/psf/whatispsf.html

[11] D. L. Dill, A. J. Drexler, A. J. Hu, C. H. Yang, Protocol verification as a hardware design aid., in: ICCD, Vol. 92, Citeseer, 1992, pp. 522–525.

[12] Y. Dong, X. Du, Y. Ramakrishna, C. Ramakrishnan, I. Ramakrishnan, S. A. Smolka, O. Sokolsky, E. W. Stark, D. S. Warren, Fighting livelock in the i-protocol: A comparative study of verification tools, in: Tools and Algorithms for the Construction and Analysis of Systems, Springer, 1999, pp. 74–88.

[13] M. Heiner, M. Herajy, F. Liu, C. Rohr, M. Schwarick, Snoopy–a unifying Petri net tool, in: Application and Theory of Petri Nets, Springer, 2012, pp. 398–407.

[14] K. Jensen, A brief introduction to coloured Petri nets, in: Tools and Algorithms for the Construction and Analysis of Systems, Springer, 1997, pp. 203–208.

[15] K. Jensen, L. M. Kristensen, Coloured Petri nets: modelling and validation of concurrent systems, Springer Science & Business Media, 2009.

[16] K. Jensen, L. M. Kristensen, Coloured Petri nets - modelling and validation of concurrent systems, chapter 7: State spaces and behavioural properties, University Lecture.

[17] K. Jensen, S. Christensen, L. M. Kristensen, CPN tools state space manual, Department of Computer Science, Univerisity of Aarhus.

[18] A. Jirachiefpattana, R. Lai, Verifying Estelle specifications: Numerical Petri nets approach, in: Network Protocols, 1993. Proceedings., 1993 International Conference on, IEEE, 1993, pp. 334–341.

[19] K. Kirbas, Open System Interconnection (OSI), IT blog, dostopno 8. maj, 2015.
url: http://www.kayhankirbas.com/2011/06/osi/

[20] G. Kovács, Z. Pap, G. Csopaki, Automatic test selection based on CEFSM specifications, Acta Cybern. 15 (4) (2002) 583–599.

[21] G. Leduc, Research @ RUN, LOTOS, dostopno 24. april, 2015.
url: http://www.run.montefiore.ulg.ac.be/Research/Topics/index.php?topic=Lotos

[22] M. Mraz, M. Moškon, Modeliranje računalniških omrežij, Založba FE in FRI, 2015.

[23] A. Møller, Program verification with Hoare logic, dostopno 22. april, 2015.
url: http://cs.au.dk/ amoeller/talks/hoare.pdf

[24] A. V. Ratzer, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, K. Jensen, CPN tools for editing, simulating, and analysing coloured Petri nets, in: Applications and Theory of Petri Nets 2003, Springer, 2003, pp. 450–462.

[25] A. S. Tanenbaum, Computer Networks, 3rd Edition, 3rd Edition, Prentice Hall, 1996.

[26] E. U. of Technology, CPN Tools homepage, dostopno 3. junij, 2015.
url: http://www.cpntools.org/

[27] K. Özdemir, H. Ural, Deadlock detection in CFSM models via simultaneously executable sets, in: Proc. 6th Conference on Communication and Information, Citeseer, 1994, pp. 673–688.