KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
VERSLO INFORMATIKOS KATEDRA

Rasa Baužaitė

Agregatinių specifikacijų verifikavimas transformuojant jas į baigtinius automatus

Magistro darbas

Kalbos konsultantė
Lietuvių k. katedros lekt.
I. Mickienė
2006-05

Vadovas
prof. habil. dr. H. Pranevičius
2006-05

Recenzentas
dr. Antanas Mikuckas
2006-05

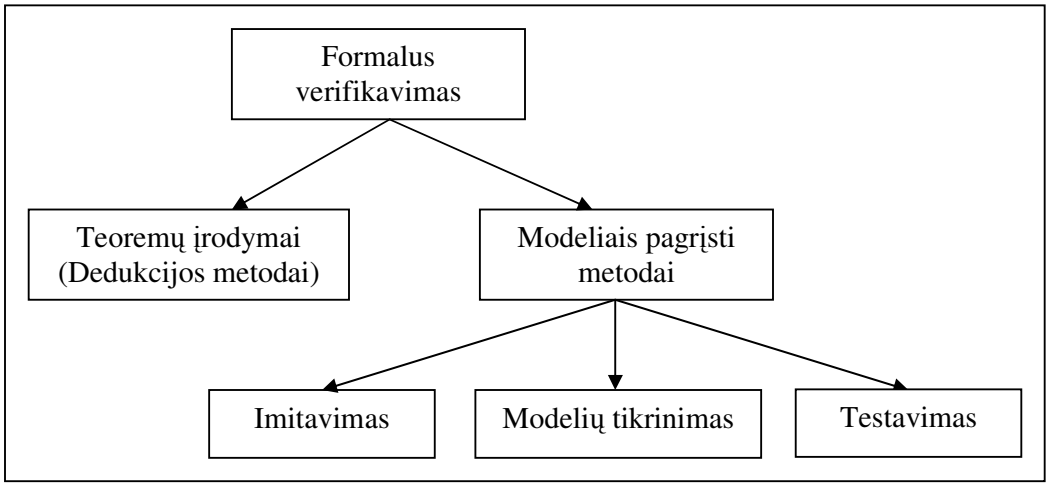
Atliko
IFM-0/1 gr. stud. Rasa Baužaitė
2006-05

Kaunas
2006

Turinys

ĮVADAS

Įvairia metodika siekiama sumažinti ir palengvinti verifikavimą tuo pačiu didinant jo efektyvumą. Formalūs metodai siūlo dideles galimybes siekiant greičiau integruotis į procesų verifikavimą, sumažinti verifikavimo laiką ir leidžia naudoti efektyvesnius verifikavimo būdus [12]. Formaliame verifikavime išskiriamos dvi verifikavimo kategorijos: dedukcijos ir modeliais pagrįsti metodai (1 pav.).



1 pav. Formalaus verifikaimo metodai [12]

Dedukcijos metodai – sistemos korektiškumas apibrėžiamas matematinėmis savybėmis. Šios savybės įrodomos didesnio tikslumo įrankiais, skirtais teoremų įrodymui ir įrodymo tikrinimui[18].

Modeliais pagrįsti metodai – galimas sistemos elgesys apibūdinamas matematiškai, tiksliai ir nedviprasmiškai. Algoritmai, kurie semantiškai išnagrinėja visas sistemos modelio būsenas lydi sistemų modelius. Tai sudaro visos verifikavimo metodologijos pagrindą, nuo nuodugnaus verifikavimo iki eksperimentų, su apribotais scenarijų rinkiniais modelyje (imitavimas) ir realybėje (testavimas)[18].

Vienas iš labiausiai naudojamų ir žinomų būdų yra imitavimas. Programinės įrangos įrankis – imitatorius, leidžia vartotojui analizuoti sistemos elgesį.

Modelių tikrinimas yra verifikavimo metodika, kuri tiria visas galimas sistemos būsenas [19]. Modelių tikrintojas nagrinėja visas tiesiogiai susijusias sistemos būsenas, ir tokiu būdu patikrina ar patenkinama norima savybė. Jei nustatoma, kad būsena pažeidžia tikrinamą savybę, modelių tikrintojas nurodo kaip pasiekti tą būseną. Taigi, pateikiamas kelias nuo pradinės būsenos iki būsenos, kuri pažeidžia savybę verifikuojant. Imitatoriaus pagalba vartotojas gali peržiūrėti būseną pažeidžiančią savybę, veiksmų seką iki jos, gaudamas naudinga tikrinimo informaciją, ir tuo pačiu, pakeisti modelį.

Formalaus verifikavimo metodai, tokie kaip imitavimas ir modelių tikrinimas yra pagrįsti modelio aprašu, iš kurio sugeneruojamos visos galimos sistemos būsenos, todėl naudojant gerai išnagrinėtą verifikavimo techniką, galima testuoti sistemą šiuo metodu, net ir tada, kai sunku ar net neįmanoma sudaryti sistemos modelį [30].

Analogiškai, kaip ir modelių tikrinime, pradžios taškas, modeliais grįstame testavime yra tiksli, vienareikšmiška sistemos specifikacija. Tradiciniuose testavimo metoduose to paprastai nėra. Paremti šiomis formaliomis specifikacijomis, testų generavimo algoritmai generuoja įrodomai pagrįstus testus, t.y., testus kurie ieško kas turėtų būti testuojama.

Norint pagerinti modelio kokybę, verta prieš modelio tikrinimą atlikti imitavimą. Imitavimas gali būti efektyviai panaudotas nesudėtingų modeliavimo klaidų radimui. Nesudėtingų klaidų pašalinimas atliktas prieš išsamų tikrinimą sumažina verifikavimo laiką ir sudėtingumą.

Norint atlikti tikslų verifikavimą, savybės turi būti apibrėžtos aiškiai ir nedviprasmiškai. Tai paprastai atliekama pasinaudojant specifikavimo kalba.

Sistemos modelis apibrėžia sistemos funkcionavimą tiksliai ir nedviprasmiškai. Dažniausiai jis vaizduojamas baigtiniu automatu, sudarytu iš baigtinės būsenų aibės ir perėjimų tarp jų. Būsenas sudaro informacija apie kintamųjų reikšme, prieš tai vykdytus sakinius, ir t.t. Perėjimai apibūdina kaip sistema pereina iš vienos būsenos į kita. Baigtiniai automatai apibūdinami naudojantis modelių aprašymo kalbomis kaip C, Java, VHDL, Promela ir t.t.

SPIN sistemoje verifikavimas atliekamas įrodinėjant procesų bendravimo korektiškumą, stengiamasi abstrahuotis nuo vidinių nuoseklių skaičiavimų.

SPIN įvesties kalba yra Promela (Process Meta Language) [4]. Ji iš kitų programavimo kalbų išsiskiria tuo, kad yra tinkama aukštam abstrakcijos lygiui. Promela kalba aprašytas modelis tampa efektyviai patikrinamas, jei laikomasi šių reikalavimų:

- Promela kalba galima specifikuoti tiktai baigtines sistemas, net jei taikomoji programa yra potencialiai nebaigtinė,
- Analizuojama sistema turi būti pilnai specifikuota, t.y., ji turi būti uždara savo aplinkai. Visi įvesties šaltiniai turi būti modelio dalis, bent abstrakčioje formoje.

Specifikacijos korektiškumo tikrinimo charakteristikos yra suskirstytos į dvi grupes: saugumo ir gyvybingumo. Saugumo charakteristikos rodo, kad sistemoje neįvyksta iš anksto apibrėžtų nepageidaujamų įvykių. Apibūdinant elgseną, naudojami du pagrindiniai metodai: konstruktyvus ir aksiomatinis. Konstruktyvusis metodas elgseną apibūdina programa, parašyta kokia nors programavimo kalba arba formalaus modeliavimo kalba, tokia kaip I/O

automatai, įvairioms Petri tinklų modifikacijoms, sąveikaujančių procesų algebra, agregatiniu metodu, LTL, Promela.

Lentelėje 1 yra pateikiamos paskirstytųjų sistemų specifikavimo kalbos ir atitinkami matematiniai metodai, kurie buvo panaudoti jas kuriant.

Lentelė Nr. 1 *Specifikavimo kalbose naudojami matematiniai metodai*

Specifikavimo kalba	**Matematinis metodas**
SDL, ESTELLE	Išplėsti baigtiniai automatai
LOTOS	Sąveikaujančių sistemų skaičiavimai
Z, VDM	Aibių teorija ir matematinė logika
ESTELLE/Ag	Atkarpomis-tiesiniai agregatai
Promela\SPIN	Baigtiniai automatai

Šio darbo tikslas yra agregatinių specifikacijų verifikavimas transformuojant jas į baigtinius automatus. Darbe turi būti sukurta transformavimo metodika, sudrytos konkrečių sistemų PLA (atkarpomis tiesiniais agregatais) specifikacijos, kurios būtų transformuojamos į baigtinius automatus, kuriuos aprašius Promela kalboje, būtų atliekamas verifikavimas naudojant SPIN sistemą.

Šis darbas susideda iš trijų pagrindinių skyrių. Pirmame skyriuje apžvelgiamos paskirstytų sistemų specifikavimo ir verifikavimo priemonės. Nagrinėjama SPIN sistema. Aptariama SPIN sistemoje specifikavimui naudojama Promela kalba, bei jos sintaksė. Pristatomi atkarpomis tiesiniai agregatai. Apibendrinimui pabrėžiamas abstrakcijos poreikis paskirstytų sistemų tyrimui. Antrame skyriuje pateikiamas atkarpomis tiesinių agregatų specifikacijų transformavimo į baigtinius automatus metodas. Formalizuojamas agregatinių specifikacijų transformavimas į baigtinius automatus ir automatų aprašymas Promela kalba. Trečiame skyriuje pateikiami atkarpomis tiesinių agregatų specifikacijų verifikavimo pavyzdžiai.

1. PASKIRSTYTŲ SISTEMŲ SPECIFIKAVIMO IR VERIFIKAVIMO PRIEMONĖS

1.1. SPIN sistema

SPIN yra modelių tikrinimo sistema, galinti patikrinti vartotojo sukurtos sistemos elgsenos modelio loginių sakinių teisingumą [20]. Šie modeliai paprastai imituoja procesų elgesį paskirstytose sistemose, o "loginiai sakiniai" atitinka klaidas, apie kurias spėja sistemos testuotojas.

Pritaikymo sritis – paskirstytų sistemų programinės įrangos testavimas. Ši sistema padeda rašant paskirstytų sistemų programinę įrangą ir atsako į klausimą: ar gali testuojama sistema sutrikti?

Šis įrankis gali būti naudojamas aklaviečių procesų lenktyniavimo ar sinchronizavimo klaidoms aptikti. Taip pat gali būti panaudotas ir daug sudėtingesnėms asinchroninių procesų sistemos laikinų savybių problemoms spręsti [21] [23].

SPIN programinė įranga parašyta standartine C kalba, ir yra suderinama su Unix, Linux, cygwin, Plan9, Inferno, Solaris, Mac, ir Windows. Ši programinė įranga, kaip ir UNIX, SmallTalk, TCP/IP, Tcl/Tk, ir World Wide Web sistemos, buvo apdovanota prestižiniu ACM (Association for Computing Machinery) programinės įrangos sistemų apdovanojimu (Software System Award) [22].

SPIN, kaip formaliųjų metodų taikymo priemonės, tikslai:

1. suteikti intuityvią, panašią į programavimo kalbą, anotaciją, skirtą vienareikšmiškai (ir be realizacijos detalių) aprašyti sistemos modeliui;
2. suteikti galingą, bet glaustą anotaciją pagrindiniams korektiškumo reikalavimams aprašyti;
3. suteikti metodologiją, kuri nustatytų loginį ryšį tarp projektavimo pasirinkimų iš 1 punkto ir korektiškumo reikalavimų iš 2 punkto.

SPIN pritaikyta efektyviam programinės įrangos verifikavimui. Šis įrankis palaiko aukšto lygio programavimo kalbą Promela, naudojamą specifikacijoms aprašyti. SPIN naudojama loginių klaidų sekimui paskirstyto projektavimo sistemose, tokiose kaip operacines sistemos, duomenų perdavimo protokolai, lygiagretaus programavimo algoritmai, geležinkelio signalų protokolai ir t.t. Šis įrankis tikrina loginį specifikacijos nuoseklumą. Praneša apie aklavietes, neapibrėžtus pranešimus, vėliavėlių nesuderinamumus.

SPIN palaiko programavimo kalbą C. Tai leidžia tiesiogiai verifikuoti realizavimo lygio programinės įrangos specifikacijas, naudojant SPIN kaip loginį variklį, verifikuojant aukšto lygio laikines savybes.

SPIN dirba "on-the-fly". Tai reiškia, kad išvengiama išankstinio globalaus būsenų grafo formavimo būtinybės, norint pradėti verifikavimą.

SPIN gali būti naudojamas kaip pilna LTL (Linear time Temporal Logic) [27] [35] modelio tikrinimo sistema, palaikanti visus korektiškumo reikalavimus, aprašytus LTL. Korektiškumo reikalavimai gali būti specifikuoti kaip sistemos ar procesų invariantai, kaip LTL reikalavimai, kaip formalus Büchi automatas, ar dar plačiau, kaip bendrosios savybės „never“ tvirtinimų sintaksėje.

SPIN palaiko ir „rendezvous**“** (kai žinute siunčiama ir gaunama nenaudojant buferių)**,** ir „buffered“ pranešimų perdavimą, bei komunikavimą naudojant paskirstytą atmintį. Taip pat palaikomos mišrios sistemos, naudojančios sinchroninį ir asinchroninį komunikavimą,. Pranešimų kanalo identifikatoriai abiems „rendezvous“ ir „buffered“ kanalams, gali būti perduoti iš vieno proceso į kitą per pranešimus.

SPIN sistemoje tikrinamos:

Saugumo savybės – ieškomas kelias vedantis į klaidas. Jei tokio kelio sistema neranda, savybė patenkinama:

- Invariantas *x visada mažiau už y;*
- aklaviečių nebuvimas – s*istema niekada nepasiekia būsenos kur veiksmai negalimi.*

Gyvybingumo savybės – rasti (begalinį) ciklą, kuriame niekas nevyksta. Jei tokio ciklo nėra, savybė patenkinama:

- Užbaigimas – *sistema pagaliau baigs darbą;*
- Reagavimas – *jei įvykis X įvyks tada pagaliau įvyks ir įvykis Y.*

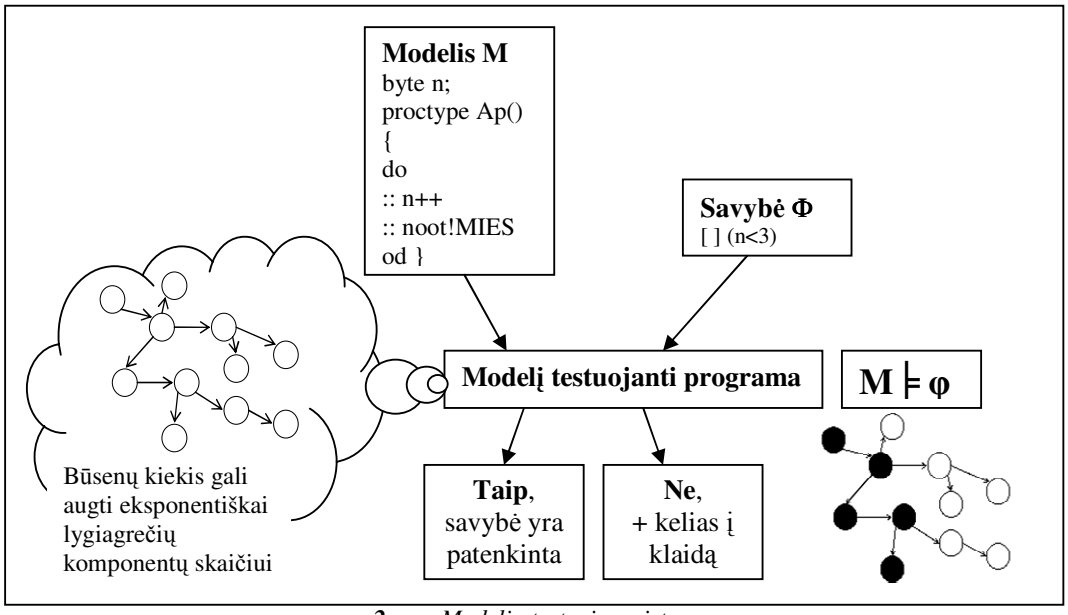
Dažniausiai pasitaikančios modelių klaidos kuriant paskirstytas sistemas (tinklo taikomosios programas, duomenų komunikavimo protokolus, daugelio gijų (multithreaded) programinę įrangą, kliento-serverio taikomąsias programas) gali būti aptiktos SPIN pagalba:

- Aklavietės (deadlocks)
- Amžini ciklai (livelocks), badavimas (starvation)
- Specifikacijos nepilnumas (Underspecification)
  - netikėtas pranešimo gavimas
- Specifikacijos perteklinis kodas (Overspecification)
  - „mirties“ kodas
- Apribojimų pažeidimai
  - buferio perpildymas
  - masyvų apribojimų pažeidimai
- Prielaidos apie laiką

- loginis teisingumas
- realaus laiko darbas

Programuoti lygiagretaus programavimo sistemas yra sudėtinga, nes dažnai tokio tipo, kaip anksčiau išvardinta, klaidos nepastebimos. Tačiau jos nesunkiai gali būti aptiktos naudojant medelių testavimo metodiką. Kaip jau minėta, vieną jų siūlo SPIN sistema.

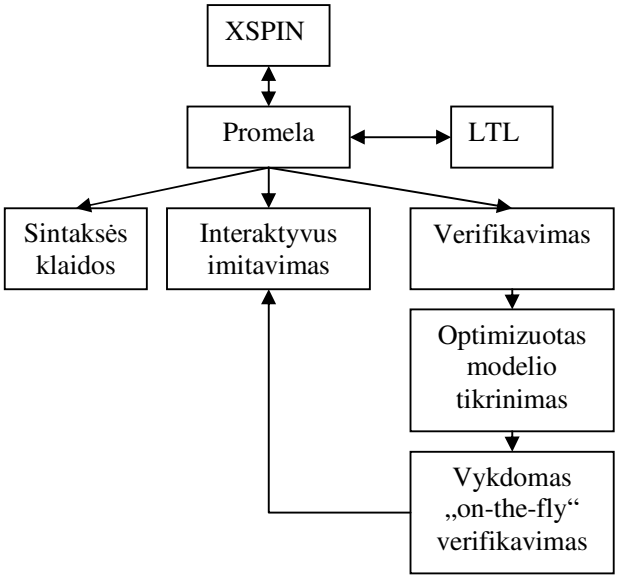
Modelių testavimas SPIN sistemoje grafiškai pavaizduotas paveikslėlyje (pav.2).



2 pav. *Modelių testavimo sistema*

SPIN veikia kitaip nei kiti testavimo įrankiai. Šios sistemos vartotojai pirmiausiai paprašomi sukurti palyginti tikslų testuojamos sistemos aprašymą vykdomo modelio pavidalu. Tada jau gali būti atliktas analizuojamos sistemos imitavimas ir nuodugnus verifikavimas.

Bendra SPIN modelių tikrinimo struktūra pateikta paveikslėlyje (pav.3) [8]. Tipinis darbo režimas prasideda nuo aukšto lygio lygiagretaus veikimo sistemos (arba paskirstyto algoritmo) specifikacijos aprašymo paprastai naudojant grafinį įrankį XSPIN. Po to, kai sintaksės klaidos jau yra pašalintos, atliekamas dialoginis (interaktyvus) imitacinis modeliavimas kol pasiekiamas pradinis įsitikinimas, kad sistema veikia kaip buvo suplanuota. Trečiame žingsnyje, atliekamas verifikavimas aukšto lygio specifikacijai (vykdymo metu - „on-the-fly“). Jei aptinkami kokie nors prieštaravimai teisingumo savybėms, grįžtama į imitavimo fazę kad detalizuoti ir surasti klaidos priežastis.

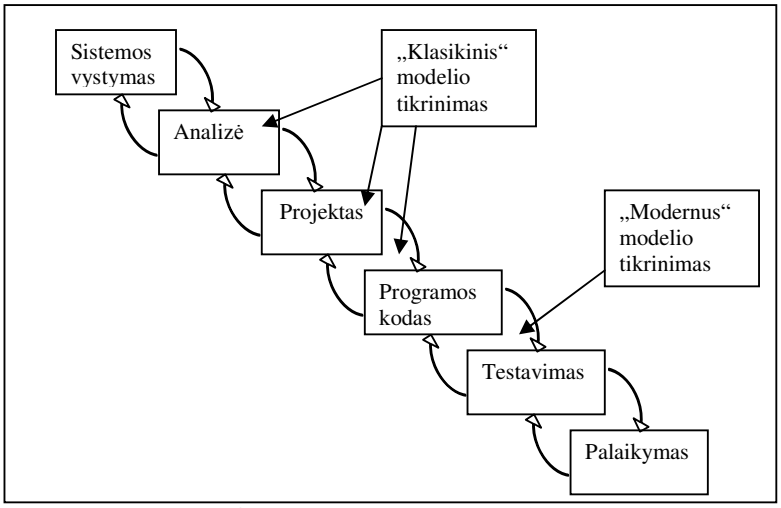


3 *pav.* *Imitavimo ir verifikavimo struktūra SPIN sistemoje [8]*

SPIN yra pakankamai galingas, kad apimtų ir labai detalius, artimus realizacijai modelius. Tačiau tai reikalauja atitinkamai didesnių kompiuterio našumo išteklių.

Dar vienas svarbus SPIN įrankio privalumas yra tas, kad jį gali naudoti ir reikalavimų nustatymo inžinieriai, sistemos projektuotojai, programuotojai bei testuotojų. Šio įrankio palaikomų analizės tipų gali prireikti dar prieš pradedant programuoti, nes jis gali dirbti su aukšto lygio detaliais “eskizais”.

Sistemos kūrimas prasideda nuo sistemos idėjos išvystymo, jos analizės ir projekto sudarymo (pav. 4). Dar šiose ankstyvose fazėse prasideda modelio testavimas. Toliau darbe šis etapas bus vadinamas klasikiniu modelio testavimu. Automatinis verifikavimas yra ne tik naudingas teisingumo įrodymui, bet jis dar pasižymi klaidų radimu ankstyvoje sistemos kūrimo stadijoje. Vėliau, po realizacijos programavimo kalba, prasideda modernus modelio tikrinimas – atliekamas verifikavimas. Šiame etape, SPIN dažniausiai naudojamas praktikoje, nors jis puikiai tinka ir ankstyvų fazių tikrinimui.



4 pav. *Klasikinis krioklio modelis*

SPIN siūlomi tikrinimo būdai M ⊨ φ (M – modelis, φ – tikrinamos savybės):

1. Interaktyvus, atsitiktinis ir valdomas imitavimas;
2. Dalinis tikrinimas – SPIN „bitstate hashing“ metodas (būsenos neišsaugomos) greitai pereinantis per visas būsenas;
3. Išsamus tikrinimas, kur atliekama pilna ir tiksli analizė, pagal nustatytus korektiškumo reikalavimus. Sukurta tikrinimo aproksimacijos sistema, kur gali būti verifikuoti netgi labai dideli modeliai su maksimaliu būsenų padengimu:
   - Suspaudimas (būsenų vektoriaus)
   - Optimizavimas (SPIN parinktys)
   - Nukreipimai (valdant SPIN „slicing“ algoritmui)

SPIN sistemoje naudojamas imitavimo algoritmas:

```
    while (! error & ! allBlocked) do                    // aklavietė ≡ allBlocked
        ActionList menu = getCurrentExecutableActions();  // Aplanko visus procesus ir
                                                          // surenka visus vykdomus
        allBlocked = (menu.size() == 0);                  // veiksmus
                if (! allBlocked)
                Action act = menu.chooseRandom();         // Interaktyvus imitavimas: act
                                                                  // valdomas vartotojo
                error = act.execute();                    // act yra įvykdomas ir
                        fi                                        // sistema pereina į kitą
būseną
        od
```

SPIN sistemoje naudojamas verifikavimo algoritmas:

SPIN naudoja grafo trasavimo gilyn [29] (depth first search DFS) algoritmą, būsenų generavimui ir tyrimui.

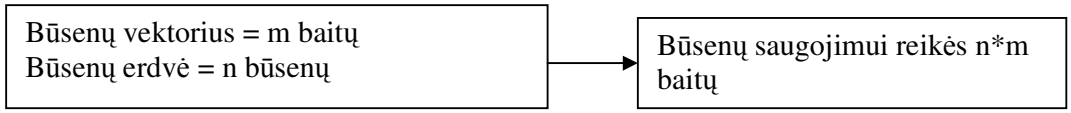
```
    procedure dfs(s: state)
        if error(s) then report error fi
        add s to Statespace                               // būsenos saugomos hash
                                                                  // lentelėje
        foreach successor t of s do
                if t not in Statespace then dfs(t) fi     // senos būsenos laikomos
                                                          // dfs - steke, kuris atitinka apeitų
        od                                                // būsenų kelią
```

end dfs

Atkreipkite dėmesį į tai, kad būsenų erdvės konstravimas ir klaidų ieškojimas vyksta tuo pačiu metu „on-the-fly". Būsenų vektorius skirtas sistemos būsenos identifikavimui. Jis sudarytas iš:

- Visų globalių kintamųjų
- Visų pranešimų kanalų
- Kiekvienam sistemos procesui:
  - visų lokalių kintamųjų
  - procesų skaitiklio procesams

Labai svarbu minimizuoti būsenų vektoriaus dydį, tam SPIN naudoja keletą algoritmų.



SPIN naudoja keletą optimizavimo algoritmų kad verifikavimas vyktų efektyviau:

- „partial order reduction" [24] [36]:
  jei globalioje būsenoje, procesas P gali vykdyti tik "lokalius" sakinius, tada visi kiti procesai gali būti atidėti vėlesniam laikui
- „bitstate hashing" (apytikslis) [25]:
  vietoj to, kad saugoti tiksliai kiekvieną būseną, tiktai vienas atminties bitas panaudojamas išsaugoti pasiektą būseną
- „hash compaction" (apytikslis) [26];
- „state vector compression" ("atskirų būsenų suspaudimas");
- duomenų srauto analizė: „mirusių" kintamųjų analizė, sakinių suliejimas.

1.2.Paskirstytų sistemų specifikavimo kalba Promela

Promela yra sutrumpinimas iš Process Meta-Language. Termino „meta" panaudojimas šiame kontekste yra reikšmingas. Kaip žinome, abstrakcija paprastai yra raktas į sėkmingą verifikavimą. Ši specifikavimo kalba skirta gero abstrakcijos lygio radimo palengvinimui sistemų specifikavime. Promela kalba skirta sistemų aprašymui, o ne jų realizavimui. Šios kalbos esminis skirtumas yra tas, kad ji skirta procesų sinchronizavimo ir koordinavimo modeliavimui.

Pagrindiniai Promela modelių struktūriniai blokai yra asinchroniniai procesai, buferizuoti ir nebuferizuoti pranešimų kanalai, sinchronizuojantys sakiniai, ir struktūriniai duomenys. Iš anksto numatyta, kad nėra galimybės specifikuoti laiko; nėra slankaus kablelio skaičių, ir yra tik keletas skaičiavimo funkcijų. Šie apribojimai netrukdo modeliuoti ir verifikuoti kliento – serverio protokolų darbo.

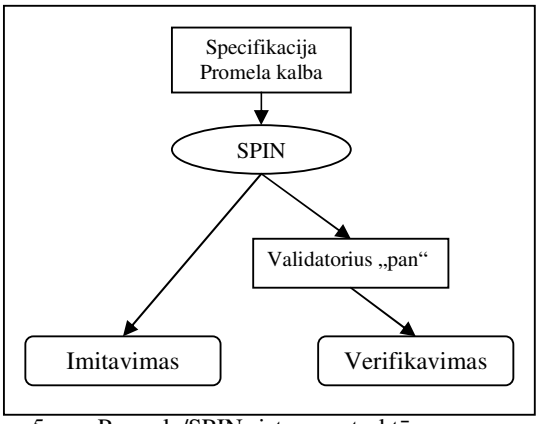
Promela modeliuose dėmesys sutelktas į paskirstytų sistemų koordinavimo ir sinchronizavimo aspektus, o ne į skaičiavimus. Tokiam pasirinkimui yra atitinkamos priežastys. Pirma, korektiškai koordinuotų struktūrų paskirstytų sistemų programinės įrangos modeliavimas ir verifikavimas paprastai daug sudėtingesnis nei neinteraktyvių nuoseklių skaičiavimų kūrimas. Antra, susidaro kuriozinė situacija, nes šiuo metu loginis verifikavimas paskirstytose sistemose, nors dažnai brangesnis resursų atžvilgiu, gali būti atliktas daug nuodugniau ir daug patikimiau nei paprasčiausios skaičiavimų procedūros verifikavimas. Specifikavimo kalba, naudojama sistemų verifikavimui, sąmoningai sukurta paskatinti vartotoją abstrahuotis nuo grynųjų skaičiavimų ir sutelkti visą dėmesį į specifikavimą procesų sąveikos sisteminiame lygyje.

Šio specializavimosi rezultatas – Promela turi daug savybių, kurių nėra daugelyje populiarių programavimo kalbų. Šios savybės skirtos darbo palengvinimui, kuriant aukšto lygio paskirstytų sistemų modelių konstrukcijas. Ši kalba palaiko nedeterministinių valdymo struktūrų specifikacijas, pagrindus procesų kūrimui, ir didelę aibę komunikavimo elementų procesų. Tačiau ji neturi savybių, kurių turi kitos programavimo kalbos, pavyzdžiui, funkcijų grąžinančių reikšmes. Priežastis yra ta, kad Promela nėra realizavimo kalba, ji skirta kurti verifikavimo modelius. Verifikavimo modelis skiriasi dviem pagrindiniais požymiais nuo programos parašytos programavimo kalba Java ar C:

- Verifikavimo modelis atitinka abstrahuotą modelį su išskirtomis sistemos savybėm, kurias ketinama verifikuoti
- Verifikavimo modelis dažnai turi tokias dalis, kurios netinkamos realizacijai. Tai gali būti neigiami spėjimai apie sistemos elgesį, kurie gali įvykti sąveikoje su aplinka ir svarbiausiai, modelis gali turėti tiesiogines arba netiesiogines specifikacijas teisingumui tikrinti.

Taigi, Promelą apžvelgsime kaip programavimo kalbą specifikuoti lygiagretaus veikimo elgesį ir procesų sąveikavimą paskirstytose sistemose.

Promela/SPIN sistemos struktūra pateikta paveikslėlyje 5. Kaip jau minėta, darbas su SPIN sistema prasideda nuo specifikacijos Promela kalba. Sukūrus specifikaciją jau gali būti atliekamas imitavimas ir verifikavimas.



5 ***pav.*** Promela/SPIN sistemos struktūra

Kiekviena sudėtingesnė užduotis gali būti išskaidyta lygiais. Kiekvieno lygio serviso nuosekli ir išbaigta procedūrinių taisyklių aibė, turi būti išvesta ir apibrėžta formalia kalba. Tai vienas sudėtingiausių uždavinių protokolo projektavimo procese.

Verifikuodami projektą, turime sugebėti tiksliai išdėstyti, kokie yra kriterijai, kuriais galėtume apibrėžti projekto korektiškumą. Protokolas turi būti specifikuotas, formaliai įrodant, jog mirties taškai (deadlocks), amžini ciklai (livelocks), neteisingi (nenumatyti) protokolo užsibaigimai yra negalimi.

Kalboje Promela yra numatyti formalizmai korektiškumo reikalavimams specifikuoti. Korektiškumo kriterijai specifikacijoje formuojami, verifikavimo modelio elgesį pavadinant negalimu; jei jis tampa galimas, pažeidžiama korektiškumo sąlyga.

Verifikavimo modelio elgesys yra pilnai apibrėžiamas vykdymo sekų, kurias jis gali atlikti, aibe. Vykdymo seka yra baigtinė, tvarkinga būsenų aibė. Būsena visiškai apibrėžiama visų lokalių ir globalių kintamųjų reikšmių, visų duomenų srautų taškų vykdomuose procesuose specifikacijomis ir visų pranešimų kanalų turiniu. Verifikavimo modelis gali būti imituojamas, nustatant konkrečias kintamųjų reikšmes, apibrėžiant duomenų srautų taškus ir kanalų būsenas.

Vykdymo seka laikoma teisinga duotam Promela modeliui M, jei

- pirma sekos būsena yra pradinė M būsena, kurioje visi kintamieji inicializuoti nuliui, visi pranešimų kanalai tušti ir vienas (init) procesas aktyvus savo pradinėje būsenoje;
- perėjus M į būsena su numeriu i, egzistuoja bent vienas vykdomas sakinys, kuris sistemą pervestų į būseną i+1.

Vykdomosios sekos gali būti užsibaigiančios (terminating) ir ciklinės (cyclic, potencialiai begalinės).

Konstantos apibrėžiamos dviem būdais:

- Taip kaip ir C kalboje naudojant #, pvz.: `#define NAME 5;`

- Raktiniu žodžiu mtype.

Šie operatoriai ir funkcijos naudojami aprašyti išraiškas:

+	-	*	/	%	>	>=	<	<=	==	!=	!
&	||	&&	|	~	>>	<<	^	++	--	len()	
empty()		nempty()		nfull()	full()	run	eval()	enabled()		pc_value()	

Promela kalboje yra apibrėžiami trys specifinių tipų objektai: procesai, pranešimų kanalai ir kintamieji. Procesai yra visada globalūs objektai, kanalai ir kintamieji gali būti ir globalūs, ir lokalūs. Procesų elgesys, kanalai ir globalūs kintamieji apibrėžia aplinką, kurioje vyksta procesai.

Promela kalboje nėra skirtumo tarp sakinių ir sąlygų. Net izoliuotos loginės sąlygos gali būti naudojamos kaip sakiniai. Sakinio vykdymas yra sąlyginis nuo jo įvykdomumo. Pavyzdžiui, abu sakiniai „while (a != b) skip“ ir „(a = = b)“ yra identiški. Paskutinis užrašas tarp skliaustų reiškia blokavimąsi iki tol, kol sąlyga nebus išpildyta.

Lentelėje pateikiami Promela kalboje apibrėžti duomenų tipai

Lentelė Nr. 2 *Duomenų tipai*

Pavadinimas	Ribos	Pavyzdžiai
bit	0 .. 1	bit turn = 1;
bool	false .. true	bool flag = true;
byte	0 .. 255	byte cnt;
chan	1 .. 255	chan q;
mtype	1 .. 255	mtype msg;
pid	0 .. 255	pid p;
short	$-2^{15}-1$ $2^{15}-1$	short s = 100;
int	$-2^{31}-1$ $2^{31}-1$	int x = 1;
unsigned	0 $2^{n}-1$	unsigned u : 3;

Dar vienas duomenų tipas chan bus aptariamas žemiau. Duomenų masyvų tipai taip pat galimi ir deklaruojami analogiškai kaip C kalboje.

Procesų tipai deklaruojami proctype raktiniu žodžiu. Pavyzdžiui,

```
byte state = 2;
proctype A(byte state; short set) { (state == 1) -> state = set }
proctype B() {  state = state - 1 }
```

deklaruojami du A ir B tipo procesai. Pirmasis iš jų blokuojamas kol state kintamas taps lygus 1, vėliau jam priskiriama reikšmė 3. Procesas B modifikuos kintamo reikšmę ir terminuosis. Sakinių atskyrimui naudojami operatoriai ; ir -> yra identiški.

Deklaruoti procesai gali būti paleidžiami elementariu operatoriumi run. Kadangi run yra apibrėžtas kaip operatorius, jis gali būti naudojamas bet kokiame sudėtiniame sakinyje ar

sąlygoje. Pavyzdžiui, i = run A() && (run B() || run C()) bus teisinga. Operatorius grąžina pid numerį, ir jis bus lygus nuliui, jei sistemoje einamu momentu neįmanoma sukurti procesų.

```
init {  run A(state, 3); run B() }
```

Šiame užrašyme lygiagrečiai paleidžiami du A ir B tipo procesai; beje, procesų parametrais negali būti nei masyvai, nei procesų tipai. Procesas init turi būti deklaruotas kiekvienoje Promela specifikacijoje, ir jį būtų galima palyginti su C kalbos analogu – main funkcija.

Sakinių seka, apskliausta skliaustais užrašyta po atomic raktinio žodžio, yra vykdoma kaip vienas nedalomas vienetas, nepertraukiamas kitų procesų. Sekos atomic apibrėžimas protokolo verifikavimo procese žymiai sumažina nagrinėjamų galimų veiksmų išsišakojimų skaičių. Tai vienas iš sinchronizacijos mechanizmų:

```
byte state = 1;
proctype A() { atomic { (state == 1) -> state = state + 1 } }
proctype B() { atomic { (state == 1) -> state = state – 1 } }
init { run A(); run B() }
```

Promela kalbos sintaksėje išskiriami šie meta terminai:

comments komentarai. Komentarų viduje galima rašyti bet kokį tekstą, šis tekstas nėra vykdomas. Komentarai prasideda simboliais „/*“ ir baigiasi – „*/“.

false loginis kintamasis, yra sinonimas konstantos reikšmei 0, kuri paprastai naudojama išraiškose.

true loginė konstanta. Yra sinonimas reikšmės 1, ir taip pat sinonimas skip.

inline procedūros mechanizmas. Apibrėžiama taip pat kaip ir proctype deklaravimas. Visada turi būti deklaruota globaliai. Inline iškvietimas gali būti rekursyvus, bet ne ciklinis. Sintaksė: `inline vardas ([argumentų sąrašas]) {sekos}`.

skip sutrumpinimas į tuščią, nulinį sakinį. Dažniausiai naudojamas perėjimui į sakinių sekos pabaigą.

```
proctype A()
{L0:    if
        :: cond1 -> goto L1     /* peršokama į sekos pabaigą */
        :: else -> skip         /* "-> skip" perteklinis */
        fi;
        ...
L1:     skip}
```

active proctype prefikso deklaravimas, kad inicijuoti pradinį procesą.

Sintaksė: active proctype [vardas] ([deklaravimo sąrašas]) {sakiniai}.

arrays masyvo deklaravimas.

`byte state[N],` kur N yra konstanta, apibrėžianti masyvą iš N baitų. Ši konstanta

gali būti sutartinė, neįtakojama išorės (side effect free) ar Promela išraiška. Anksčiau užrašytos išraiškos galima reikšmių aibė yra 0 .. N-1.

progress Dviejų rūšių ciklai, kurie įvardija nelauktas ir klaidingas būsenas, yra šie:

- neprogresyvūs (non-progress) ciklai – tai tokie ciklai, kurių viduje tam tikros veiksmų sekos ar būsenos yra nepasiekiamos neapibrėžtai ilgą laiko tarpą;
- amžini ciklai (livelocks) – ciklai, kurių viduje tam tikros veiksmų sekos ar būsenos kartojasi neapibrėžtai ilgą laiko tarpą.

Kalboje Promela numatytos priemonės korektiškumo kriterijams išreikšti abiems atvejams. Pirmuoju atveju, raktinis žodis **progress** (galūnė gali būti įvairi; pvz., progress0, progress_inc - šie žodžiai rodys teisingą žymos vartojimą) nurodys, kuris sakinys turi būti vykdomas išvengiant neprogresyvių ciklų. Pavyzdžiui, Dijkstra semaforui

```
proctype dijkstra()
{
end:        do
            :: sema!p ->
progress:   sema?v
            od
}
```

nurodoma, kad semaforo vedantis sakinys negali būti aplenkiamas neapibrėžtai dideliu iteracijų skaičiumi. Automatinis verifikatorius patvirtintų šį teiginį.

accept Amžinų ciklų išvengimui kalboje numatytos **accept** žymos, kurios nurodo, koks sakinys ar jų aibė negali būti vykdomi neapibrėžtai dažnai. Pavyzdžiui,

```
proctype dijkstra()
{end:   do
        :: sema!p ->
accept: sema?v
        od}
```

teigiama, kad neįmanoma iteruoti p ir v operacijomis nuolatos. Tai neteisingas teiginys. Pateiktame pavyzdyje sukuriamos trys būsenos, tai pradinė, būsena tarp siuntimo ir gavimo ir pabaigos (nepasiekiama) būsena iškart po ciklo kartojimo konstrukcijos.

xr Šis sakinys naudojamas aprašymui kai norime išskirtinių kanalo skaitymo teisių, pvz., pranešimų gavimui. Jei daugiau nei vienas procesas bando pasiekti pažymėtą kanalą, susidaro klaidinga situacija.

xs Šis sakinys naudojamas išskirtinio kanalo rašymo teisių aprašymui, pvz., pranešimų siuntimui. Susidaro klaidinga situacija jei daugiau nei vienas procesas bando pasiekti pažymėtą kanalą

hidden naudojamas duomenų paslėpimui, t.y. kad jie nebūtų rodomi būsenų deskriptoriuje verifikavimo metu. Pvz.:

```
hidden byte a;
hidden short p[3];
```

local paženklinti globalų kintamąjį, kuris naudojamas tik vienam procesui.

```
local byte a;
local short p[3];
```

mtype naudojamas konstantų simbolinius vardams apibrėžti.

Pavyzdys: `mtype = { ack, nak, err, next, accept }`

proctype proceso deklaravimas.

```
proctype A(mtype x) { byte state; state = x }
```

chan pranešimo kanalo deklaravimas ir inicijavimas.

chan a = [16] of { short };

/*kanalas gali talpinti iki 16 short tipo pranešimų*/

chan c[3] = [4] of { byte }; /*apibrėžto tipo ir apimties kanalų masyvas */

Pranešimai kanalu siunčiami pirmas įėjo, pirmas išėjo (first-in first-out) tvarka. Vienas ir daugiau išraiškų ir kintamųjų gali būti nurodyta kanalo priekyje, o siunčiama (!) ir priimama (?) atitinkamai, pvz.:

qname!expr1,expr2,expr3;

qname?var1,var2,var3;

qname!expr1(expr2,expr3);

qname?var1(var2,var3);

Abi užrašymų grupės ekvivalenčios: antruoju atveju expr1 ir var1 nurodytų siunčiamo ar priimamo pranešimo tipą, o likusieji kintamieji tarp skliaustelių siunčiamus ar priimamus duomenis. Siuntimas vykdomas, jei kanalas neperpildytas; duomenų priėmimas vykdomas, jei kanalas netuščias. Visi pertekliniai duomenys siuntimo ar priėmimo pusėje yra neapibrėžtos reikšmės. Kanalų siuntimo ir priėmimo operacijos negali būti naudojamos sąlyginiuose sakiniuose, tačiau priėmimo atveju numatyta galimybė pasitikrinti ar yra kanale duomenų – tai kanalo duomenys, nurodyti tarp laužtinių skliaustų. Sekančios dvi sakinių sekos yra ekvivalenčios:

```
(len(qname) > 0) -> qname?msgtype;
 qname?[msgtype] -> qname?msgtype;
```

Iki šiol nagrinėjome asinchroninius pranešimų siuntimus kanalu. Taip pat galimi ir sinchroniniai pranešimo metodai – tam tikslui kanalas aprašomas nuliniu ilgiu:

```
chan name = [0] of { byte, byte };
byte name;
proctype A()
{       name!msgtype(124);
        name!msgtype(121)
```

```
}
proctype B()
{       byte state;
        name?msgtype(state)
}
init
{       atomic { run A(); run B() }
}
```

Toks, nulinio ilgio, kanalo deklaravimas reiškia, kad kanalu siunčiami duomenys neišsaugomi, jei jų niekas neišskaito. Pavyzdyje procesas B nuskaitys proceso A išsiųstą reikšmę ir pasibaigs; proceso A antra siuntimo operacija nebus vykdoma. Jei tai būtų asinchroninis siuntimas, proceso A antra siuntimo operacija būtų įvykdyta ir kanale būtų palikti duomenys.

end Nenumatytos modelio būsenos gali apimti ne tik mirties taškų būsenas, tačiau ir daugelį kitų klaidingų būsenų, kurios yra protokolo specifikacijos loginio neišbaigtumo rezultatas. Apibrėžiama numatyta ir galima (proper) vykdymo sekos galinė būsena (end-state), jei tenkinama:

kiekvienas sukurtas procesas arba pasibaigė, arba pasiekė būseną, kuri pažymėta kaip galima galinė būsena;

visi pranešimų kanalai yra tušti.

Netenkinant bent vieno iš šių kriterijų, pabaigos būsena nelaikoma teisingai galima. Visai tikėtina ir leidžiama, kad procesas gali nesibaigti. Tokiu atveju, proceso tipo kūne nurodoma galimos galinės būsenos žyma – end. Pavyzdžiui, Dijkstra semaforo nesibaigiantis procesas, korektiškai pažymint jo galimą galinę būseną, užrašomas

```
proctype dijkstra()
{end:   do
        :: sema!p -> sema?v
        od }
```

Jei modelyje naudojama daugiau žymų, jų šaknis - end - turi išlikti ta pati, o galūnė skirtis, pvz., end0, end_war, endmod bus atpažįstamos galinės būsenos žymos.

priority naudojamas nustatyti proceso imitavimo prioritetą. Paprastai rašomas prie sakinio „run“ arba prie „active proctype“. Prie žodelio priority nurodomas skaičius, reiškiantis proceso prioritetą. Kuo didesnis skaičius tuo didesnis prioritetas. Pagal nutylėjimą prioritetas būna lygus vienam, jei konstanta nenurodoma.

```
run pavadinimas(...) priority P;
```

active proctype pavadinimas() priority P { sekos }, kur P yra konstanta >=1

never Sakykime, kad norime pateikti laikiną teiginį, kuris teigtų: “kiekviena būsena, kurioje savybė P tapati true, pereina į kitą būseną, kurioje savybė Q tapati true”, arba P -> Q. Kalboje Promela modelio korektiškumas įrodomas neiginiais, kitaip, teiginio

kenkiančio korektiškumui prieštaravimu. Todėl ir minėtas teiginys būtų užrašomas

```
never { do
:: skip
:: break od -> P -> !Q }
```

čia ciklas išreiškia bet kokio laiko tarpo vėlinimą. Visas šis never blokas reiškia, kad neįmanoma, savybei P esant teisingai, sistemai pereiti į būseną, kurioje savybė Q būtų neteisingai. Jei laikinas teiginys tarp figūrinių skliaustų yra tenkinamas (matched), gaunamas prieštaravimas, kuris prieštarauja korektiškumo kriterijui. Sakykime, kad norime išreikšti laikiną teiginį, kuriame būtų tvirtinama, jog sąlyga condition1 negali būti true neapibrėžtai ilgą laiko tarpą. Laikinajame teiginyje turime numatyti, kad sąlyga condition1 pradinėje būsenoje kažkurį laiką gali būti false, tačiau pasikeitus į true, toliau reikšmė nebesikeičia.

```
never {  do
        :: skip
        :: condition1 -> break
        od;
accept: do
        :: condition1
        od}
```

Šis teiginys tenkinamas tik tuomet, kai aptinkamas amžinas ciklas; tokiu atveju korektiškumo savybė būtų pažeista. Vienas esminis ypatumas skiriant never teiginius nuo proctype kūnų yra tas, kad kiekvienas laikinojo teiginio sakinys interpretuojamas kaip sąlyga. Tai reiškia, kad teiginių sakiniai negali turėti šalutinių efektų, kuriuos sukelia tokie Promela sakiniai, kaip priskyrimo, assert operatoriai, pranešimų siuntimai, priėmimai, printf sakiniai. Išsamus laikinų never teiginių mechanizmas įgalina tiesiogiai kreiptis į vykdomų procesų duomenų srautų būsenas ir kintamųjų reikšmes.

if struktūra apibrėžiama:

```
if
:: (cond1) -> option1
:: (cond2) -> option2
fi
```

Išrinkimo struktūra užrašoma už dvigubo dvitaškio, o jos vykdymas priklauso nuo kiekvienos sakinių sekos pirmo sakinio, vadinamo vedančiuoju (guard), įvykdomumo. Jei procesas sakiniu cond1 (cond2) nebus blokuojamas, vykdymas perduodamas sakiniui option1 (option2). Taip pat, jei išraiška expr1 (expr2) yra vykdoma, kuria procesas neblokuojamas, ji bus įvykdyta. Tik viena sakinių seka iš jų

sąrašo yra vykdoma. Jei daugiau nei vienas vedančiųjų sakinių yra vykdytini, atitinkama sakinių seka išrenkama atsitiktinai.

Ciklo struktūriniai blokai apibrėžiami

```
do
:: (cond1) -> option1
:: (cond2) -> option2
:: (cond3) -> break
od
```

Tai if struktūros praplėtimas, kuri vykdoma ciklišk ai. Čia taip pat vienu metu vykdoma viena sekanti už dvigubo dvitaškio sakinių seka. Operatorius break išeina iš ciklo vykdymo bloko.

_last globalus tik skaitymui skirtas kintamasis.

_pid lokalus, tik skaitymui skirtas kintamasis, kuris saugoja procesų skaičių.

empty operatorius susidedantis iš vieno elemento, kanalo tuštumo patikrinimui.

full operatorius susidedantis iš vieno elemento, kanalo pilnumo patikrinimui.

len kintamasis saugantis pranešimų skaičių kanale.

np_ tik skaitymui skirtas loginis kintamasis.

run operatorius naudojamas, naujo proceso sukūrimui.

stdin tik skaitymui skirtas kanalas.

timeout išreiškia sąlygą, kuri tampa teisinga tada ir tik tada, kai sistemoje nėra kitų vykdytinų sakinių. Sakinio timeout įvykdymas nesukelia jokių šalutinių efektų; šie sakiniai gali būti įterpti į išraiškas.

assert Sakinys assert(condition) yra visada vykdytinas ir gali būti užrašomas bet kurioje modelio vietoje. Korektiškumo kriterijai dažnai gali būti išreiškiami loginėmis sąlygomis, kurios turi būti išpildytos, procesui pasiekus atitinkamą būseną. Jei sąlyga true, sakinys nieko nedaro, priešingu atveju demonstruojama protokolo klaida. Pavyzdžiui,

```
byte state = 1;
proctype A()
{       (state == 1) -> state = state + 1;
                assert(state == 2)
}
proctype B()
{       (state == 1) -> state = state – 1;
                assert(state == 0)}
init {  run A(); run B() }
```

toks tokiam modeliui automatinis validatorius greitai sugeneruos klaidą, kadangi baigiantis tiek procesui A, tiek procesui B kintamojo state reikšmė gali būti lygi 1. Tai įmanoma, jei abu procesai vedančiuosius sakinius įvykdė vienu metu. Bendresnis sakinio assert taikymas – formalizuoti sistemos invariantus; nurodyta sąlyga turi išlikti true visose sistemos pasiekiamose būsenose, nepriklausomai nuo vedančios į konkrečia būseną vykdymo sekos. Pavyzdžiui,

```
proctype monitor() { assert(count == 0 || count == 1) }
```

monitor tipo procesas nurodo, kad viso protokolo vykdymo metu kintamasis count negali būti kitoks nei 0 arba 1. Šis procesas turi būti aiškiai paleidžiamas run komanda, o jo vykdymas bus aktyvinamas, kiekvieną kartą sistemai perėjus į kitą būseną.

printf teksto spausdinimas imitavimo metu.

Promela neturi realaus laiko modeliavimo savybių, specifikuojamas tik funkcionavimas. Tačiau dauguma protokolų naudoja laikmačius arba laiko matavimo mechanizmus, kad persiųsti pranešimus ar patvirtinimus. Todėl yra išskirti keli būdai laikui skaičiuoti:

- timeout - specialus kintamasis Promeloje. Timeout reikšmė pereis į (true) būseną kai nebus jokių kitų galimų vykdyti sakinių. Timeout modeliuose yra globalus kintamasis, padedantis išvengti aklaviečių.
- Else - taip pat specialus kintamasis, kuris gali būti naudojamas if/do sakiniuose. Reikšmė pereis į (true) tik tada kai jokie kiti sąlygos sakiniai iš if/do ciklų nebus tenkinami.

Pagrindinė schema atstatymui po pranešimo pametimo:

```
proctype Siuntėjas() {
...
        do
        :: R ! MSG(duomenys, siusti_B) ->
                if
                :: S ? ACK(gauti_B) -> ...              /* normalus atvejis */
                :: timeout -> ...                       /* pranešimas pamestas */
                fi
        od}
```

Priešlaikinis timeout gali būti sumodeliuotas pakeičiant timeout kintamąjį sakiniu skip.

Paprasta schema sumažinti priešlaikinius laikmačius:

```
#define pirmalaikis_timeout
        if
        :: nr_ pirmalaikis _timeouts <= MAX_ pirmalaikis_ timeout ->
                if
                :: pirmalaikis_timeout=true ;
                nr_ pirmalaikis _timeouts++
                :: pirmalaiki _timeout=false
                fi
        :: else -> pirmalaikis_timeout=false
        fi
```

Įmanoma imituoti ir diskretųjį laiką.

```
byte time;
proctype Tick() {
        do
        :: timeout -> (time = time+1)%MAXTIME;
        od}
```

Jei nė vienas iš procesų neveikia, galima pailginti testavimo laiką. Naudojant šį laikmatį, galima ilgiau ieškoti aklaviečių.

Tokiu būdu imituojant laiką pastebimi ir minusai:

- Visi veiksmai, kuriems reikia laiko, turi sinchronizuotis laike – šuoliai laike veikia kaip tvarkaraštis modelyje ir tai daro modelį sunkiau suprantamą.
- Tai yra brangus būdas dirbant su laiku – šio proceso metu reikia atkreipti dėmesį į galinčias išsiplėsti būsenas.

Metodas TimeAdvance() – imituojamas laikas pereina į sekantį laiko vienetą, kuriame įvykiai sukelia būsenų perėjimą ir visas įsiterpiantis laikas praleidžiamas. Didesnis efektas pastebimas, kai laiko tarpas tarp šių momentų yra didelis.

Du sėkmingi laiko įgyvendinimai SPIN sistemoje.

- RT-SPIN – realus laikas [34];
- DT-SPIN – deterministinis laikas. Naudoja panašų būdą kaip procesas **Tick**, bet turi pakeistą „dalinės tvarkos“ algoritmą SPIN sistemoje, kad pasinaudoti specialiomis proceso **Tick** charakteristikomis.

1.3. Atkarpomis-tiesiniai agregatai

Atkarpomis-tiesinių agregatų (PLA) metodas yra išsamiai aprašytas G.Kovalenko bei N.Buslenko publikacijose. Šiame paragrafe yra pateikiami šio metodo pagrindiniai principai [1].

Aprašant sistemą, sistemos būsenų aibėje *S* yra išskiriama baigtinė pagrindinių būsenų aibė $I = \{0,1,2,\ldots s\}$. Šios aibės elementai $\nu \in I$ yra pagrindinės agregato būsenos. Kiekvienai pagrindinei būsenai yra priskiriamas sveikas neneigiamas skaičius $\|\nu\|$, kuris vadinamas būsenos rangu bei išgaubtas daugiakampis $Z^{(\nu)}$ užduotas $\|\nu\|$ išmatavimų euklido erdvėje. Laikoma, kad būsenų aibę $Z = \bigcup_{\nu \in I} Z^{(\nu)}$ sudaro poros $(\nu, z^{(\nu)})$, čia $\nu \in I$, o $z^{(\nu)} \in Z^{(\nu)}$. $z^{(\nu)}$ - vadinamos papildomomis agregato koordinatėmis.

Pradiniu laiko momentu t_0 agregatas būna būsenoje $z(t_0) = (\nu, z^{(\nu)}(0))$, čia $z^{(\nu)}(0) \in Z^{(\nu)}$. Kai nėra įėjimo signalo, kai $t > t_0$ taškas $z^{(\nu)}(t)$ juda srityje $Z^{(\nu)}$ iki bus

pasiektas šios srities kontūras. Laiko momentas t_1, kai pasiekiamas kontūras, vadinamas atraminiu.

Daugiakampio Z kontūras yra aprašomas lygtimis:

$$\sum_{i=1}^{\|\nu\|} \gamma_{ji}^{(\nu)} z_i^{(\nu)} + \gamma_{j0}^{(\nu)} = 0\,,\ \ j = 1,\ldots, m(\nu),$$

čia $m(\nu)$ – kontūrų skaičius;

$z_i^{(\nu)}$ - vektoriaus $z^{(\nu)}$ komponentės, $i = 1,\ldots,\|\nu\|$;

$\gamma^{(\nu)}$ - sistemos parametrais apibūdinami dydžiai.

Atraminiu laiko momentu agregato pagrindinė būsena kinta iš ν į ν', o agregato papildomos koordinatės įgyja reikšmę $z^{(\nu')}(t_1) \in Z^{(\nu')}$.

Toliau papildomos koordinatės kinta srityje $Z^{(\nu')}$ iki nepateks ant šios srities kontūro. Tam įvykus agregatas vėl keičia būseną.

Atkarpomis-tiesinio agregato būsenai patekus į srities kontūrą yra išduodamas išėjimo signalas $y \in Y$, čia Y – išėjimo signalų aibė, kuri yra analogiška aibei Z. Išėjimo signalo struktūra

$$y = \left(\lambda, y^{(\lambda)}\right),$$

čia λ - išėjimo signalo diskretinė dalis;

$y^{(\lambda)}$ - išėjimo signalo papildomų koordinačių vektorius, priklausantis nuo λ

$$y^{(\lambda)} = \left(y_1^{(\lambda)},\ldots, y_{r(\lambda)}^{(\lambda)}\right).$$

Jei laiko momentu t^* į agregatą yra paduodamas įėjimo signalas, tai agregato papildomos koordinatės nustoja kitusios ir agregato būsena momentaliai pereina į kitą tos pačios srities ar kitos srities $Z^{(\nu'')}$ tašką.

Įėjimo signalas turi struktūrą analogišką išėjimo signalo struktūrai, t.y.

$$x = \left(\mu, x^{(\mu)}\right).$$

Įėjimo signalo atėjimo momentu agregatas išduoda išėjimo signalą ir šis momentas taip pat yra atraminis. Toliau taškas $z^{(\nu'')}(t)$ srityje $Z^{(\nu'')}$ juda taip pat, kaip buvo aprašyta anksčiau.

Kai nėra įėjimo signalų atkarpomis-tiesinio agregato būsenos kitimas apsirašo tokiomis lygtimis:

$$\nu(t) = \nu = const\,;\ \frac{dz^{(\nu)}}{dt} = \alpha^{(\nu)},$$

čia $\alpha^{(\nu)}$ - pastovus vektorius, turintis pavidalą

$$\alpha^{(\nu)} = \left(\alpha_1^{(\nu)}, \ldots, \alpha_{m(\nu)}^{(\nu)}\right).$$

Vektorinėje formoje diferencialinės lygties sprendinys $z^{(\nu)}(t)$ gali būti užrašytas

$$z^{(\nu)}(t) = z^{(\nu)}(0) + \alpha^{(\nu)}(t - t_0).$$

Spręsdami papildomų koordinačių kitimo ir sričių kontūrų lygtis kartu galima išskaičiuoti laiko momentus, kai papildomų koordinačių reikšmės patenka į kontūrus. Pvz., pirmas atraminis laiko momentas

$$t_1 = \min\left[t : z^{(\nu)}(0) + \alpha^{(\nu)}(t - t_0) \in \bigcup_{i=1}^{m(\nu)} Z_j^{(\nu)}, t > t_0\right]$$

arba

$$t_1 = \min\left\{t : t > t_0, \sum_{i=1}^{\|\nu\|} \gamma_{ji}^{(\nu)}\left[z_{ji}^{(\nu)}(0) + \alpha_i^{(\nu)}(t - t_0)\right] + \gamma_{j0}^{(\nu)} = 0\right\}.$$

Minimumas yra ieškomas indeksų $j = 1, \ldots, m(\nu)$ aibėje.

Jeigu pažymėsime

$$\tau_j = -\left(\sum_{i=1}^{\|\nu\|} \gamma_{ji}^{(\nu)} z_i^{(\nu)}(0) + \gamma_{j0}^{(\nu)}\right) \Bigg/ \sum_{i=1}^{\|\nu\|} \gamma_{ji}^{(\nu)} \alpha_i^{(\nu)}$$

ir

$$\tau = \min\{\tau_j : \tau_j > 0\},$$

tai laiko momentas, kai pirmą kartą pasiekiamas kontūras

$$t_1 = t_0 + \tau.$$

1.4. Abstrakcija SPIN sistemoje ir PLA modelyje

SPIN verifikavimo sistemoje visas dėmesys sutelkiamas į valdymo aspektus paskirstytose sistemose. O programavimo kalba Promela leidžia išreikšti prielaidas apie sąveiką su kitais moduliais. Ši programavimo kalba neskatina detalios procesų vidinių skaičiavimų specifikacijos. Specifikacija Promela kalba tampa efektyviai patikrinama jei laikomasi šių reikalavimų:

- Modelis gali specifikuoti tiktai baigtines sistemas, netgi jei taikomoji programa yra potencialiai nebaigtinė,
- Modelis turi būti pilnai specifikuotas, t.y., jis turi būti uždaras savo aplinkai (closed to its environment). Visi įvesties šaltiniai turi būti modelio dalis, bent abstrakčioje formoje.

Pavyzdžiui, programa, kuri leidžia neribotą rekursiją, ar turi neribotus buferius, nėra baigtinė. Ši programa negali būti verifikuota automatinėmis technologijomis nebent įvedus atitinkamas abstrakcijas. O tokias galimybes suteikia PLA modelis ir SPIN sistema.

Abstrakcijos svarbumas įtakoja įvesties kalbą. Jei įvesties kalba per daug detalizuota, tai varžo abstrakciją, kuri galų gale užgriozdina verifikavimo procesą. SPIN sistemoje daug programavimo savybių buvo atsisakyta. Tarp jų atminties valdymo palaikymo ir skaitmeninės analizės. Kitos verifikavimo sistemos yra dar labiau ribotos, netgi atsisako struktūrizuotų duomenų tipų, pranešimų perdavimo, ir procesų kūrimo. Tačiau yra mažiau ribotų sistemų, bet jos sudėtingumo kaina palaiko: realaus laiko verifikavimą, tikimybes, ir procedūras. Mažiausiai apribotoje sistemoje lengva kurti modelius, bet verifikuoti sunku. SPIN bando rasti balansą tarp lengvo naudojimosi ir modelio tikrinimo efektyvumo.

Pirmiausiai reikia nuspręsti kurios modelio dalys yra svarbiausios ir reikalauja verifikavimo. Reikia išskirti sistemos reikalavimų rinkinį. Reikalavimai turi būti testuotini. Turi būti įmanoma aiškiai nustatyti, kokiomis aplinkybėmis reikalavimas pažeidžiamas. Taip pat sistema neturėtų sutrikti nepažeidus bent vieno iš tų reikalavimų. Šių reikalavimų formavimas jau pats savaime yra tyrimo procesas. Kiti modelio reikalavimai ar apribojimai iš pradžių gali būti visiškai nežinomi, kol pradinė analizė neparodo jų reikšmingumo.

Toliau, reikia apsvarstyti tuos aspektus, kurie yra pagrindiniai apsaugant sistemos elgesį, ir padeda sistemai įvykdyti savo reikalavimus.

Abstrakcija turėtų būti tik tiek detali, kad išlaikyti sprendimo esmę, ir ne daugiau. Tikslas yra mažiausias pakankamas modelis kuris leidžia atlikti verifikavimą.

Naudinga galvoti apie verifikavimo modelį, kaip apie sistemą iš procesų modulių ir sąsajų. Kiekvienas proceso modulis atstoja asinchroninę esybę paskirstytose sistemose arba agregatą PLA modelyje. Kiekvienas procesas turi būti detalizuotas tik tiek, kad atliktų minimalų savybių rinkinį. Minimizuoti padeda PLA aprašymo struktūra. Šis aprašymas tampa sąsajos apibrėžimu. Verifikavimo tikslas yra patikrinti ar visi sąsajos apibrėžimai, formalizuojantys savybes, kurias procesai sistemoje atlieka tarpusavyje, yra logiškai atitinkantys. Tai galima patikrinti be kiekvieno modulio detalizavimo, tiesiog juos laikant juodosiomis dėžėmis arba minimaliai detalizuojant, kaip agregatai PLA specifikacijoje.

Verifikavimo modelis turi leisti paneigti projekto sakinius. Elementai kurie negali būti paneigti, turi būti ištrinami. Tai padidina sistemos verifikavimo efektyvumą. Imre Lakatos yra pasakęs: "Analizės tikslas yra ne priversti tikėti bet sukelti abejones."

Verifikavimo įrankiai tapo galingos tikrinimo sistemos. Tačiau sudėtingumo problema išliko. Su šia problema galima kovoti tik abstrakcijas palaikančiomis sistemomis kaip SPIN ir matematiniais metodais kaip PLA.

2. Atkarpomis tiesinių agregatų specifikacijų transformavimas į baigtinius automatus ir Promela kalbą

2.1. Agregatinių specifikacijų transformavimas į automatus

Būsenų grafas yra puiki priemonė specifikavimui, analizavimui, dizainui, ir sudėtingų sistemų dokumentacijai. Grafinė kalba tarnauja aprašant valdymą, duomenų transformavimą, ir laikinius aspektus kuriant sistemą. Automatų kalba apjungia keletą abstrakcijų, ir tai ją kvalifikuoja kaip reaktyviųjų sistemų kalbą. Vertimas į būsenų grafus gali būti pavaizduotas įvykių išraiškomis, sąlygomis ir veiksmų išraiškomis [11] [31].

Bendro pavidalo agregatas nėra baigtinė sistema, t.y. jo būsenų skaičius nėra baigtinis. Akivaizdu, jog taip yra, jei nors vienos diskrecinės koordinatės reikšmių aibė begalinė. Antra vertus, tolydinių koordinačių reikšmės – tai atsitiktiniai dydžiai su nurodyta pasiskirstymo funkcija, t.y. tolydinės koordinatės įgyja reikšmes iš realių teigiamų skaičių aibės intervalo, o tai reiškia, jog jų galimų reikšmių aibė taipogi begalinė. Kadangi operacijų trukmės – atsitiktiniai dydžiai, agregato funkcionavimas tampa atsitiktiniu procesu, kuris kiekvienu atveju gali vykti skirtingai. Priešingai, negu automatai, agregato būsenos nėra aiškiai išskaičiuojamos jį aprašant, jos gali būti gaunamos sekant agregato funkcionavimą ir fiksuojant kiekvieną skirtingą būsenos koordinačių kombinaciją [1]. Visa tai neleidžia tiksliai apibrėžti sistemos verifikavimui SPIN.

Taigi, konceptualaus modelio pagrindu siūloma apibūdinti sistemos funkcionavimą baigtiniais automatais. Baigtinis automatas siūlomas kaip tarpinis formatas perėjimui iš PLA specifikacijos prie verifikavimo SPIN sistema.

Baigtinio būsenų automato A veikimas yra baigtinis tada, kai galinis perėjimas (s_{n-1}, l_{n-1}, s_n) turi savybę s_n, kuri $s_n \in F$ (F yra pabaigos būsenų aibė). Automato veikimas laikomas baigtiniu tada ir tik tada kai sustoja veikti galinėje būsenoje [4].

Dauguma realių objektų modeliuojama ne vienu atskiru agregatu, bet jų sistema, susidedančia iš kelių tarpusavyje susietų agregatų. Globalinė tokios sistemos būsena – tai atskirų agregatų būsenų sujungimas. Perėjimai iš agregato pradinės būsenos galimi įvykus vidiniam įvykiui. Agregatinėje sistemoje vienu metu gali vykti kelios operacijos, kurių trukmės atsitiktiniai dydžiai. Vadinasi, negalima nustatyti, kuri iš šių operacijų pasibaigs pirma, ir atitinkamai – kuris įvykis įvyks pirmas. Norint išsamiai išnagrinėti agregatinės sistemos funkcionavimą, reikia peržiūrėti visus galimus variantus. Įvykus vidiniam įvykiui, gali būti perduodami signalai, kuriuos apdorojus pereinama į tolesnę stabilią būseną, t.y. būseną, kurioje nėra likę neapdorotų signalų. Šioje būsenoje vėl gali vykti viena ar kelios operacijos, kurioms pasibaigus įvyksta vidiniai įvykiai, ir sistema (pasibaigus signalų

apdorojimui) pereina į tolesnę stabilią būseną. Tai tęsiama tol, kol yra generuojamos būsenos, besiskiriančios nuo jau esančių. Šį procesą siūloma vaizduoti baigtiniu automatu, sudarytu iš būsenų ir perėjimų tarp jų. Kai perėjimo metu perduodami signalai, virš atitinkamo lanko užrašomi perduodamų signalų pažymėjimai.

Taigi, sistemai specifikuotai agregatiniu metodu, automatai sudaromi taip:

1. Kiekvienas agregatas nagrinėjamas atskirai. Agregatinėje specifikacijoje tolydusis laikas panaikinamas: visos tolydžiosios dedamosios w(e, t_m) keičiamos į w(e)=1, jei operacija aktyvi, ir w(e)=0, jei operacija neaktyvi. Po tokių pokyčių agregato tolydžiosios dedamosios tampa nepriklausomos nuo laiko. Ši transformacija leidžia naudoti nelaikinius automatus agregatinių specifikacijų verifikavimui.

$$w(e_i'', t_m) = \begin{cases} 1, \text{ jei operacija aktyvi,} \\ 0, \text{ kitu atveju.} \end{cases}$$

2. Nusakoma agregato pradinė būsena.
3. Apibrėžiama galutinė agregato būsena.
4. Automatas susideda iš baigtinės būsenų aibės S ir perėjimų aibės T. Pradinėje S_0 agregato būsenoje apibrėžiama pradinė būsena. O perėjimas T yra trejetas: būsena z, iš kurios išeina lankas, įvykis, kuris sukelia perėjimą e, būsena z′, t.y. (z, e, z′), čia z′=H(z, e).
5. Sudaroma sistemos būsenų aibė S ir perėjimų tarp būsenų aibė T. Būsenų ir perėjimų aibes sudaro įvykiai ir jų sąlygos.

Taip gaunamas automatas artimas naudojamam SPIN sistemoje [28]:

A(S, S_0, L, T, F), kur

S yra baigtinė būsenų aibė,

S_0 yra pradinė būsena, $S_0 \in S$,

L yra baigtinė žymių (labels) aibė,

T yra perėjimų aibė,

F yra pabaigos būsena, $F \subseteq S$.

Šis automatas turi paprastą prioritetų koncepciją, kuri palengvina sekančio žingsnio apskaičiavimą. SPIN sistemoje automato perėjimai turi išplėstines žymes (labels), kas leidžia apibrėžti perėjimo tikslo konfigūraciją.

Remiantis prieš tai pateikta automato sudarymo metodika, bus sudarytas automatas alternuojančio bito protokolo agregatui „*Siuntėjas*“. Šio protokolo formalus aprašymas yra ateiktas 3.1 poskyryje.

Automatą sudaro keturios būsenos:

S_1: paketo formavimas su alternuojančio bitu 0;

S_2: paketas išsiųstas ir *Siuntėjas* laukia patvirtinimo su bitu 0;

S_3 – paketo formavimas su alternuojančio bitu 1;

S_4 – paketas išsiųstas ir *Siuntėjas* laukia patvirtinimo su bitu 1.

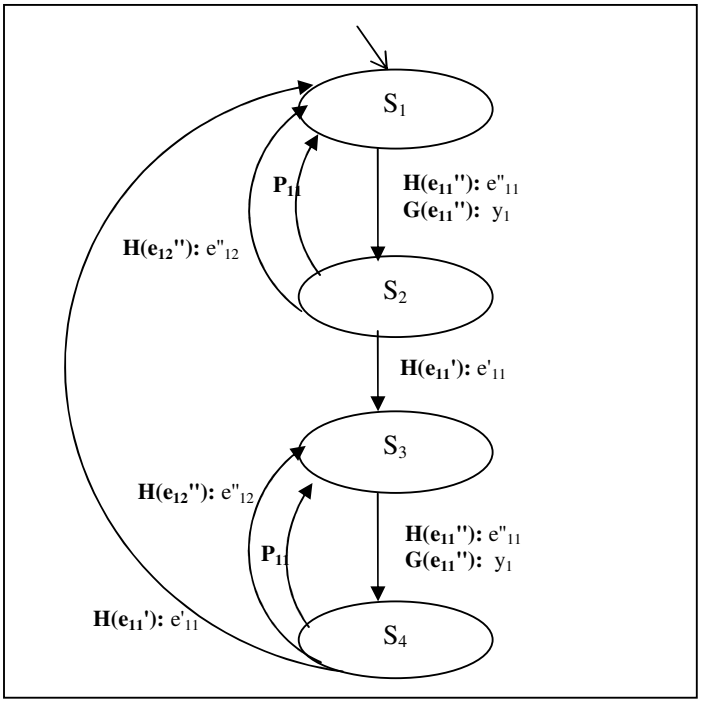
Perėjimai tarp būsenų inicijuojami šie:

e''_{11} – paketas suformuotas ir signalas į *Kanalą* išsiųstas,

e'_{11} – gautas patvirtinimas iš *Kanalo*,

e''_{12} – paketas pametamas.

1. Automatą pradedame sudarinėti nuo pradinės būsenos, kuri 6 paveikslėlyje pažymėta rodykle.



6 pav. Automatas „*Siuntėjas*“

Tolesniu žingsniu nustatomi perėjimai į kitas būsenas.

2. Kadangi operacija būsenoje S_1 yra aktyvi, jai pasibaigus, įvyks įvykis e''_{11}. Atlikus veiksmus remiantis operatoriumi $H(e''_{11})$, pereiname į būseną S_2.
3. Būsenoje S_2 galimi šie funkcionavimo atvejai: pasibaigus operacijai įvyks įvykis e'_{11} arba įvykis e''_{12}. Atlikus veiksmus remdamiesi perėjimo operatoriumi $H(e_{11}')$ pereiname į būseną S_3 arba remdamiesi perėjimo operatoriumi $H(e_{12}'')$ grįžtame į būseną S_1.
4. Būsenoje S_3 galimas vienas atvejis: pasibaigus operacijai įvyks įvykis e''_{11}. Atlikus veiksmus remiantis operatoriumi $H(e''_{11})$, pereiname į būseną S_4.
5. Būsenoje S_4 galimi šie funkcionavimo atvejai: pasibaigus operacijai įvyks įvykis e'_{11} arba įvykis e''_{12}. Atlikus veiksmus remdamiesi perėjimo operatoriumi $H(e_{11}')$ pereiname į būseną S_1 arba remdamiesi perėjimo operatoriumi $H(e_{12}'')$ grįžtame į būseną S_3.

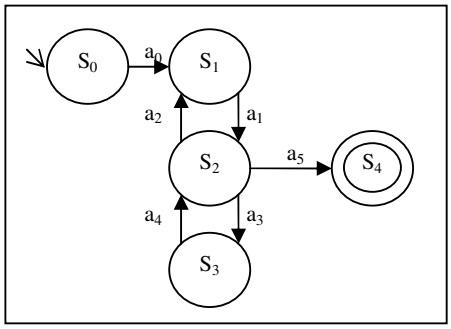
Sekantis paveikslėlis rodo paprastą baigtinį būsenų automatą su penkiomis būsenomis. Jis SPIN sistemoje apibrėžiamas:

$S = \{ S_0, S_1, S_2, S_3, S_4 \}$

$L = \{ a_0, a_1, a_2, a_3, a_4, a_5 \}$

$F = \{ S_4 \}$

$T = \{ (S_0, a_0, S_1), (S_1, a_1, S_2), (S_2, a_2, S_1), (S_2, a_3, S_3), (S_3, a_4, S_2), (S_2, a_5, S_4)\}$



7 pav. *Baigtinis automatas*

Pradinė būsena S_0 tradiciškai pažymėta rodykle, o pabaigos būsenų aibės F elementas pažymėtas dvigubu apskritimu, kaip ir pavaizduota paveikslėlyje (Pav.7) Pradiniu laiko momentu t_0 automatas yra pradinėje būsenoje S_0. Laiko momentu t_{n+1} pradinė būsena kinta iš S_0 į S_1, o papildomos koordinatės įgyja reikšmes. Perėjimų ryšiai T apibrėžia srauto valdymą. Perėjimų žymių aibė L riša kiekvieną aibės T perėjimą su atitinkama valdymo seka, kuri apibrėžia įvykdomumą ir perėjimo poveikį.

PLA modelyje sistemos koordinatės kinta su kiekvienu įvykiu pagal atitinkamą programos fragmentą ir įvyksta perėjimas nuo vienos būsenos į kitą. Kitimas priklauso nuo tikimybinių sistemos charakteristikų ir sistemos būsenos prieš įvykstant įvykiui.

Galime teigti, kad kiekvienas agregatas atitinka baigtinį būsenų automatą A(S, S_0, L, T, F) SPIN sistemoje. Šio automato būsenų aibė S atitinka galimas baigtinio automato būsenas, kurias galima išskirti iš perėjimo ir išėjimo operatorių agregtiniame modelyje (3 lentelė). Perėjimų ryšiai T apibrėžia srauto valdymą tarp būsenų ir juos atitinka išoriniai ir vidiniai įvykiai agregatiniame modelyje. Perėjimų žymių aibė L riša kiekvieną aibės T perėjimą su atitinkama valdymo seka, kuri apibrėžia įvykdomumą ir perėjimo poveikį. Pradinės būsenos S_0 pažymimos abiejuose modeliuose.

Lentelė Nr. 3 Atitikmenys tarp agregatinio aprašymo ir baigtinio automato

Agregatas	**Automatas A(S, s_0, L, T, F)**
1. Įėjimo signalų aibė	L baigtinė žymių (labels) aibė
2. Išėjimo signalų aibė	
3. Išorinių įvykių aibė	Įvykiai įtakojantys sistemos koordinačių kitimą,

4. Vidinių įvykių aibė	perėjimų aibę T
5. Valdymo sekos	yra
6. Diskrečioji būsenos dedamoji	yra
7. Tolydžioji būsenos dedamoji	nėra
8. Parametrai	yra
9. Pradinė būsena	S_0 pradinė būsena, $s0 \in S$
10. Perėjimo ir išėjimo operatoriai	formuojama baigtinė būsenų aibė S

2.2. Automatų aprašymas Promela kalba

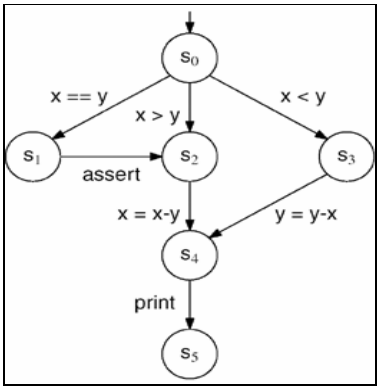
Tikslinė sistema yra paskirstytų sistemų formalaus verifikavimo paketas SPIN. Norint pradėti verifikuoti, šiuo atveju turimus baigtinius automatus reikia pervesti į Promela kalbą. Promela siūlo platų pasirinkimą abstrakčių konstrukcijų, kurios gali būti panaudotos modelių kūrimui:

- Lygiagretūs procesai su persipinančiu lygiagretumo modeliu;
- Globalūs ir lokalūs kintamieji;
- Sakiniai aprašyti užklausas ir sąlygas, ciklus;
- Lygiagrečių procesų sinchronizavimas, sinchroninis pranešimų siuntimas.

SPIN sistemoje kiekviena Promela „proctype“, atitinka baigtinį būsenų automatą A(S, s_0, L, T, F). Šio automato būsenų aibė S atitinka tam tikrus „proctype“ valdymo taškus. Perėjimų ryšiai T apibrėžia srauto valdymą (flow of control). Perėjimo žymių aibė L riša kiekvieną perėjima iš aibės T su atitinkamu aprašu, kuris apibrėžia įvykdomumą ir perėjimo rezultatą. Baigtinių būsenų aibė F, apibrėžiama su Promela žymėmis: end-state, accept-state, ir progress-state.

Pagrindinių sakinių rinkinys Promela kalboje yra labai mažas. Jų yra tik šeši: assignments, assertions, print, send ar receive sakiniai, ir Promela kalbos išraiškos. Visi kiti Promela kalbos elementai tik padeda apibrėžti srauto valdymą (flow of control) proceso vykdymo metu, t.y., padeda apibrėžti perėjimų ryšių detales aibėje T. Pavyzdžiui, sakinys „goto“ nėra vienas iš pagrindinių Promela kalbos sakinių. Sakinys „goto“ yra tokio pat lygio kaip ir kabliataškis, tiktai apibrėžia srauto valdymą (control-flow).

Pavyzdys kaip SPIN sistemoje automatai verčiami į Promela sakinius, parodytas paveikslėlyje 8. Sakinys „goto“ įtakoja kad assertion sakinio vykdymas veda į būseną s_2, vietoj s_4. Taigi jis pakeičia perėjimo tikslinę būseną, bet jis pats jokių perėjimų neprideda. Kitais žodžiais, „goto“ įtakoja perėjimo ryšį T, bet jis pats, neatsiranda aibėje L.



8 pav. *Perėjimų ryšiai*

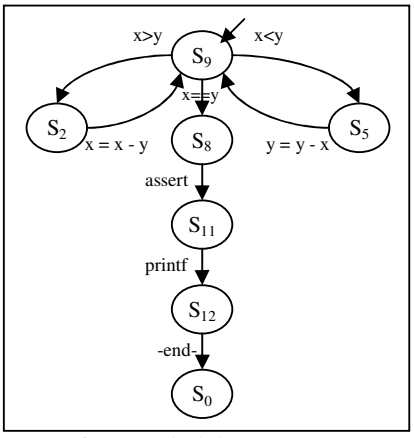
Promela modelio pavyzdys:

```
active proctype not_euclid(int x, y)
{
        if
        :: (x >  y) -> L: x = x - y
        :: (x <  y) -> y = y - x
        :: (x == y) -> assert(x!=y); goto L
        fi;
        printf(";%d\n", x)
}
```

Kalbos elementai tokie kaip if, goto, sakinių skyrikliai, kabliataškiai ir rodyklės, bei do, break, unless, atomic, ir d_step, negali atsirasti žymėmis ant perėjimų. Tik šeši pagrindiniai anksčiau minėti Promela sakiniai gali priklausyti aibei L. Promela kalboje kiekvienas sakinys turi sąlygą, kuri apibrėžia kada jis vykdomas ir reikšmę kas bus kai jis bus įvykdytas. Kiekvienas procesas turi pabaigos perėjimą kuriame jis miršta.

Dar vienas pavyzdys (9 pav.) kaip susiję Promela ir automatinis aprašymas.

```
proctype gcd(int x, y) {
L: if
:: (x > y) -> x = x-y; goto L
:: (x < y) -> y = y-x; goto L
:: (x == y) -> assert(x == y);
fi;
printf("gcd = %d\n", x)
}
```



9 pav.Baigtinis automatas

3. Atkarpomis tiesinių agregatų specifikacijų verifikavimo pavyzdžiai

3.1. Alternuojančio bito protokolas

Alternuojančio bito protokolas – tai tam tikras protokolas, kuriuo bendrauja siuntėjas (sender) ir gavėjas (receiver) [1], [13].

Alternuojančio bito protokolas užtikrina patikimą informacijos perdavimą per nepatikimą perdavimo terpę. Siuntėjas siunčia duomenų blokus, kurie perduodami gavėjui. Siuntėjas tolesnį duomenų bloką siunčia tik tada, kai gauna patvirtinimą, žymintį, kad anksčiau siųstas pranešimas sėkmingai perduotas. Perduodamieji duomenys numeruojami moduliu 2 (0 arba 1). Perdavus kadrą pažymėtą numeriu 0, sekantis bus perduodamas kadras pažymėtas numeriu 1, bet tik po to, kai bus gautas iš gavėjo patvirtinimas, jog šis gavo prieš ta siustą kadrą. Gavėjas saugo kontrolinį skaičių (0 arba 1), kuris rodo, koks kadras yra laukiamas. Jei kadras perduotas be iškraipymų (tai nustatoma palyginus gauto kadro numerį su gavėjo saugomu kontroliniu skaičiumi), jis perduodamas sekančiam vartotojui. Gavėjas siunčia patvirtinimus, kurių numeriai sutampa su gauto kadro numeriu. Procesui – siuntėjui gavus patvirtinimą, jis perduodamas pirmam vartotojui. Kartu tai ženklas šiam vartotojui, jog galima siųsti tolesnį duomenų bloką. Duomenų kadrai perdavimo metu gali būti pamesti arba iškraipyti. Tokiu atveju siuntėjas turi pakartoti šio kadro siuntimą, t.y. tam tikrą laiką negavus patvirtinimo, suveikia taimeris ir pakartotinai perduodamas kadras su tuo pačiu numeriu.

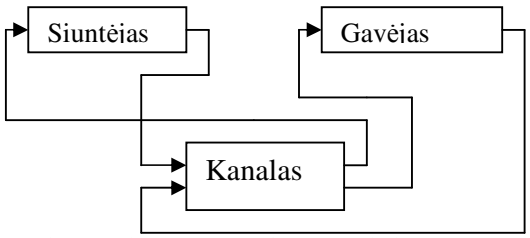
Parinkus per trumpą laikrodžio (timer) laiką, galima situacija, kai dubliuojamas ne tik perduodamas duomenų blokas, bet ir perduodamas patvirtinimas. Jei siuntėjas reaguoja ne tik į pirmą, bet ir į visus tolesnius patvirtinimus (siunčia duomenų bloko kopijas), galimas atvejis, kai gavėjui du kartus perduodamas tas pats duomenų blokas. Ši klaida ištaisoma, uždraudžiant siuntėjui kartoti duomenų bloko perdavimą reaguojant į pakartotinai gautus patvirtinimus.

3.1.1. Alternuojančio bito protokolo agregatinė specifikacija

Dauguma realių objektų modeliuojama ne vienu atskiru agregatu, bet jų sistema, susidedančia iš kelių tarpusavyje susietų agregatų (modulių) [1]. Globalinė tokios sistemos būsena – tai atskirų būsenų sujungimas. Perėjimai iš pradinės būsenos galimi įvykus vidiniam įvykiui kuriame nors modelyje. Agregatinėje sistemoje vienu metu gali vykti kelios operacijos, kurių trukmės, kaip žinia – atsitiktiniai dydžiai. Vadinasi, mes negalime nustatyti, kuri iš šių operacijų pasibaigs pirma, ir atitinkamai – kuris įvykis įvyks pirma. Norint išsamiai išnagrinėti agregatinės sistemos funkcionavimą, reikia peržiūrėti visus galimus variantus. Įvykus vidiniam įvykiui, gali būti perduodami signalai, kuriuos apdorojus sistema pereina į tolesnę stabilią būseną, t.y. būseną, kurioje nėra likę neapdorotų signalų. Šioje būsenoje vėl

gali vykti viena ar kelios operacijos, kurioms pasibaigus įvyksta vidiniai įvykiai, ir sistema (pasibaigus signalų apdorojimui) pereina į tolesnę stabilią būseną. Tai tęsiama tol, kol yra generuojamos būsenos, besiskiriančios nuo jau esančių.

Bendro pavidalo agregatas nėra baigtinė sistema, t.y. jo būsenų skaičius nėra baigtinis. Akivaizdu, jog taip yra, jei nors vienos diskretinės koordinatės reikšmių aibė begalinė. Antra vertus, pusės tolydinių koordinačių reikšmės – tai atsitiktiniai dydžiai su nurodyta pasiskirstymo funkcija, t.y. tolydinės koordinatės įgyja reikšmes iš realių teigiamų skaičių aibės intervalo, o tai reiškia, jog jų galimų reikšmių aibė taipogi begalinė. Kadangi operacijų trukmės – atsitiktiniai dydžiai, agregato funkcionavimas tampa atsitiktiniu procesu, kuris kiekvienu atveju gali vykti skirtingai. Priešingai, negu automatai, agregato būsenos nėra aiškiai išskaičiuojamos jį aprašant, jos gali būti gaunamos sekant agregato funkcionavimą ir fiksuojant kiekvieną skirtingą būsenos koordinačių kombinaciją. Visa tai neleidžia apibrėžti bendro pavidalo agregatiniam modeliui.



10 pav. *Alternuojančio bito protokolo agregatai*

Agregato Siuntėjas aprašymas:

1. Įėjimo signalų aibė:

$X = \{x_1\}$, $x_1 = B$, $B \in \{0;1\}$

x_1 – signalas apie perduoto paketo patvirtinimą;

B – alternuojančio bito reikšmė patvirtinime.

2. Išėjimo signalų aibė

$Y = \{y_1\}$, $y_1 = B$, $B \in \{0;1\}$

y_1 – signalas naujo protokolo perdavimui;

B – alternuojančio bito reikšmė perduodamame pakete.

3. Išorinių įvykių aibė

$E' = \{e_{11}'\}$

e_{11}' – priimtas signalas x_1.

4. Vidinių įvykių aibė

(Visi įvykiai, kurie keičia sistemos būseną)

$E'' = \{e_{11}'', e_{12}''\}$

e_{11}'' – paketas suformuotas siųsti;

e_{12}'' – baigėsi laikmačiui nustatytas laikas.

5. Valdymo sekos:

$e_{11}'' \rightarrow \{\eta_{1j}\}$, $j = 1,\infty$

$e_{12}'' \rightarrow \{\tau_{1j}\}$, $j = 1,\infty$

η_{1j} – j-ojo paketo formavimo trukmė;

$\tau_{1j} = const$ – laikmačiui nustatyta trukmė

6. Diskrečioji būsenos dedamoji

$\upsilon(t_m) = \{PSK(t_m), Bit_1(t_m)\}$;
$PSK(t_m)$ – priimtų patvirtinimų skaičius;
$Bit_1(t_m)$ – alternuojančio bito reikšmė paskutiniame išsiųstame pakete.

7. Tolydžioji būsenos dedamoji

$z_\upsilon(t_m) = \{\omega(e_{11}'',t_m), \omega(e_{12}'', t_m)\}$
$\omega(e_{11}'',t_m)$ – eilinio paketo formavimo pabaigos momentas;
$\omega(e_{12}'', t_m)$ - laikmačiui nustatytas pabaigos momentas.

8. Parametrai:

P_{11} – patvirtinimo klaidingo perdavimo tikimybė;
RND(1) – atsitiktinė reikšmė, tolygiai pasiskirsčiusi intervale [0,1].

9. Pradinė būsena

$z(t_0) = \{0, 1, \eta_{11}, \infty\}$.

10. Perėjimo ir išėjimo operatoriai:

$H(e_{11}')$: /Priimtas signalas x_1/
if $(B = Bit1(t_m)) \wedge (P_{11} < RND(1))$
then
$Bit1(t_{m+1}) = 1 - Bit1(t_m)$;
$PSK(t_{m+1}) = PSK(t_m) + 1$;
$\omega(e_{11}'',t_{m+1}) = t_m + \eta_{1j}$
else
$\omega(e_{11}'',t_{m+1}) = t_m + \eta_{1j-1}$;
$\omega(e_{12}'', t_{m+1}) = \infty$;
fi
$H(e_{11}'')$: /Paketas suformuotas siųsti/
$\omega(e_{11}'', t_{m+1}) = \infty$;
$\omega(e_{12}'', t_{m+1}) = t_m + \tau_i$.
$G(e_{11}'')$:
$y_1 = (Bit1(t_m))$ – išduotas signalas;
$H(e_{12}'')$: /Baigėsi laikmačiui nustatytas laikas/
$\omega(e_{11}'',t_{m+1}) = t_m + \eta_{1j-1}$;
$\omega(e_{12}'', t_{m+1}) = \infty$.

Agregato Kanalas aprašymas:

1. Įėjimo signalai

$X = \{x_1, x_2\}$, $x_i = (B)$, $i = 1,2$, $B \in \{0;1\}$
x_1 – pranešimas gautas iš siuntėjo;
x_2 – pranešimas gautas iš gavėjo;
B – alternuojančio bito reikšmė perduodamame pakete ar patvirtinime.

2. Išėjimo signalai

$Y = \{y_1, y_2\}$, $y_i = (B)$, $i = 1,2$, $B \in \{0;1\}$
y_1 – pranešimas siunčiamas gavėjui;
y_2 – pranešimas siunčiamas siuntėjui;
B – alternuojančio bito reikšmė perduodamame pakete ar patvirtinime.

3. Išorinių įvykių aibė

$E' = \{e_{21}', e_{22}'\}$
e_{21}' – signalas x_1 atėjo iš Siuntėjo;
e_{22}'– signalas x_2 atėjo iš Gavėjo.

4. Vidinių įvykių aibė

$E'' = \{e_{21}'', e_{22}''\}$
e_{21}'' – patvirtinimo perdavimas baigėsi;
e_{22}'' – paketo perdavimas kanalu baigėsi.

5. Valdymo sekos

e_{21}" -> $\{\xi_{1j}\}$, j = 1,∞; e_{22}" -> $\{\xi_{2j}\}$, j = 1,∞;
ξ_{1j} - j-ojo paketo patvirtinimo perdavimo trukmė;
ξ_{2j} – j-ojo paketo perdavimo trukmė.

6. Agregato būsena

$\nu(t_m) = \{Bit2(t_m)\}$, $z_\upsilon(t_m) = \{\omega(e_{21}",t_m), \omega(e_{22}", t_m)\}$;
$\omega(e_{21}",t_m)$ – patvirtinimo perdavimo kanalu pabaigos momentas;
$\omega(e_{22}", t_m)$ - paketo perdavimo kanalu pabaigos momentas.
$Bit2(t_m)$ – alternuojančio bito reikšmė perduodamame pakete/patvirtinime;

7. Parametrai

P_{21} – paketo praradimo tikimybė kanale;
P_{22} – patvirtinimo praradimo tikimybė kanale;
RND(2) – atsitiktinė reikšmė, tolygiai pasiskirsčiusi intervale [0,1].

8. Pradinė būsena

$Z(t_0) = \{0, \infty,\infty\}$

9. Perėjomo ir išėjimo operatoriai:

H(e_{21}'): /Signalas x_1 atėjo iš Siuntėjo/
If $(\omega(e_{21}", t_m) = \infty) \wedge (\omega(e_{22}", t_m) = \infty)$ then
    $Bit2(t_{m+1}) = B$;
    If $P_{21} \geq$ RND(2) then $\omega(e_{22}", t_{m+1}) = \infty$
    else
        $\omega(e_{22}",t_{m+1}) = t_m + \xi_{2j}$;
    fi
fi.
H(e_{22}'): /Signalas x_2 atėjo iš Gavėjo/
If $(\omega(e_{21}", t_m) = \infty) \wedge (\omega(e_{22}", t_m) = \infty)$ then
    $Bit2(t_{m+1}) = B$;
    If $P_{21} \geq$ RND(2) then $\omega(e_{22}", t_{m+1}) = \infty$
    else
        $\omega(e_{22}",t_{m+1}) = t_m + \xi_{1j}$;
    fi
fi.
H(e_{21}"): /Patvirtinimo formavimo pabaiga/
    $\omega(e_{21}", t_{m+1}) = \infty$.
G(e_{21}"): $y_1 = Bit2(t_{m+1})$ – išduotas signalas.

Agregato Gavėjas aprašymas:

1. Įėjimo signalai

$X = \{x_1\}$, $x_1 = (B)$, $B \in \{0;1\}$
x_1 –gautas pranešimas;
B – alternuojančio bito reikšmė priimamame pakete.

2. Išėjimo signalai

$Y = \{y_1\}$, $y_1 = (B)$, $B \in \{0;1\}$
y_1 – perduotas patvirtinimas;
B – alternuojančio bito reikšmė perduodamame patvirtinime.

3. Išorinių įvykių aibė

E' = $\{e_{31}'\}$
e_{31}' – signalo x_1 atėjimas.

4. Vidinių įvykių aibė

E" = $\{e_{31}"\}$
e_{31}" – patvirtinimo formavimo pabaiga.

5. Valdymo sekos

e_{31}" -> $\{\xi_{3j}\}$, j=1, ∞;

ξ_{3j} – j-ojo patvirtinimo formavimo trukmė.

6. Agregato būsena

$\nu(t_m) = \{KSK(t_m), Bit3(t_m), z_v(t_m) = \{w(e_{31}'', t_m)\}$;

$Bit3(t_m)$ – alternuojančio bito reikšmė paskutiniame suformuotame patvirtinime;

$KSK(t_m)$ – priimtų paketų skaičius;

$w(e_{31}'', t_m)$ – laiko momentas, kada baigsis patvirtinimo formavimas.

7. Parametrai:

P_{31} – paketo klaidingo perdavimo tikimybė;

$RND(1)$ – atsitiktinė reikšmė, tolygiai pasiskirsčiusi intervale [0,1].

8. Pradinė būsena

$z(t_0) = \{0, \infty, \infty\}$.

9. Perėjimo ir išėjimo operatoriai

$H(e_{31}')$: /Signalo x_1 atėjimas/

If $\omega(e_{31}'', t_m) = \infty$ then

if $(B \neq Bit3(t_m)) \wedge (P_{31} < RND(1))$

then

$KSK(t_{m+1}) = KSK(t_m) + 1$;

$Bit3(t_{m+1}) = B$;

$\omega(e_{31}'', t_{m+1}) = t_m + \eta_{3j}$

else

$\omega(e_{31}'', t_{m+1}) = t_m + \eta_{3j-1}$;

fi

fi.

$H(e_{31}'')$: /Patvirtinimo formavimo pabaiga/

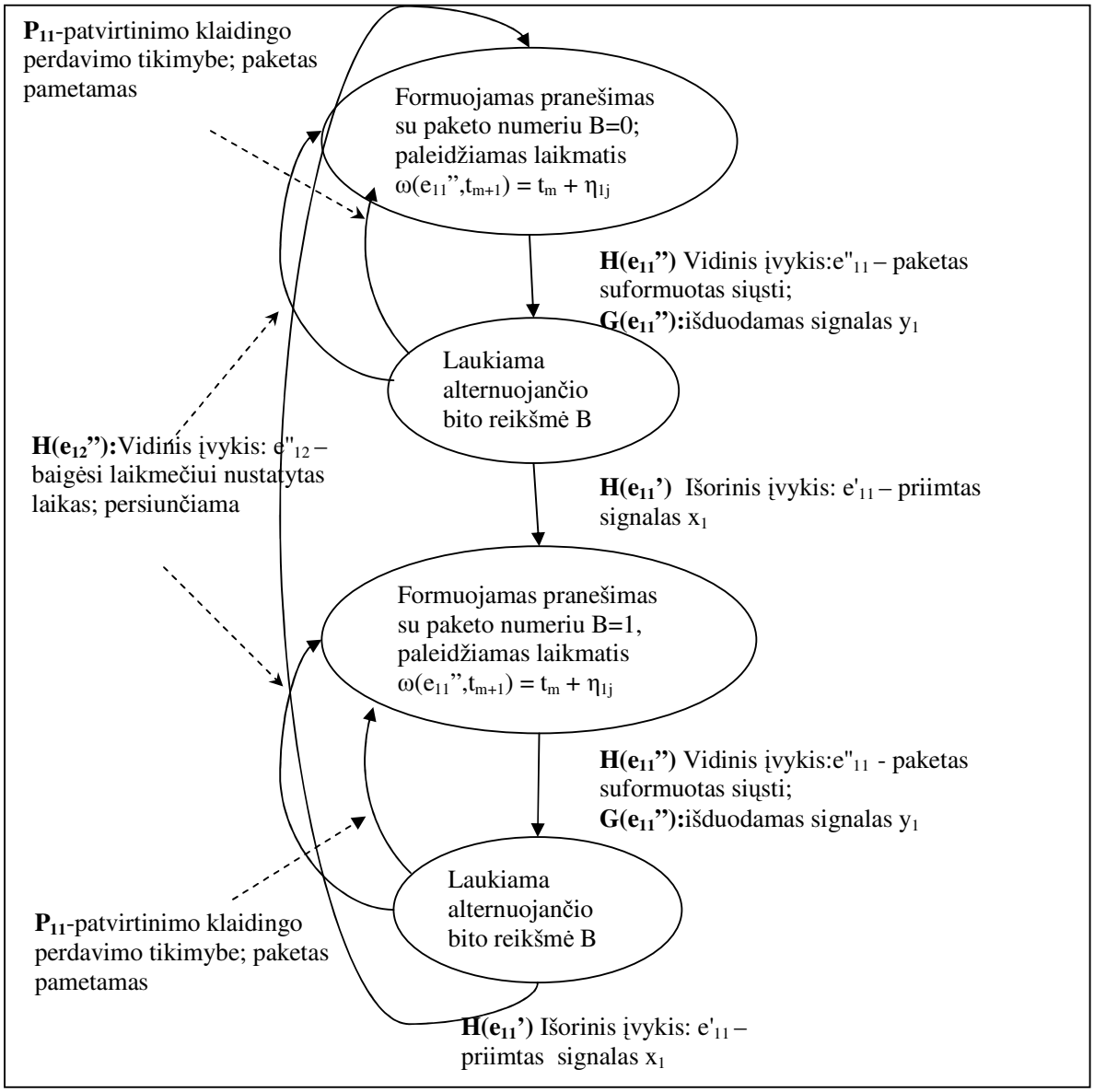
$\omega(e_{31}'', t_{m+1}) = \infty$.

$G(e_{31}'')$:

$y_1 = Bit3(t_{m+1})$ – išduotas signalas;

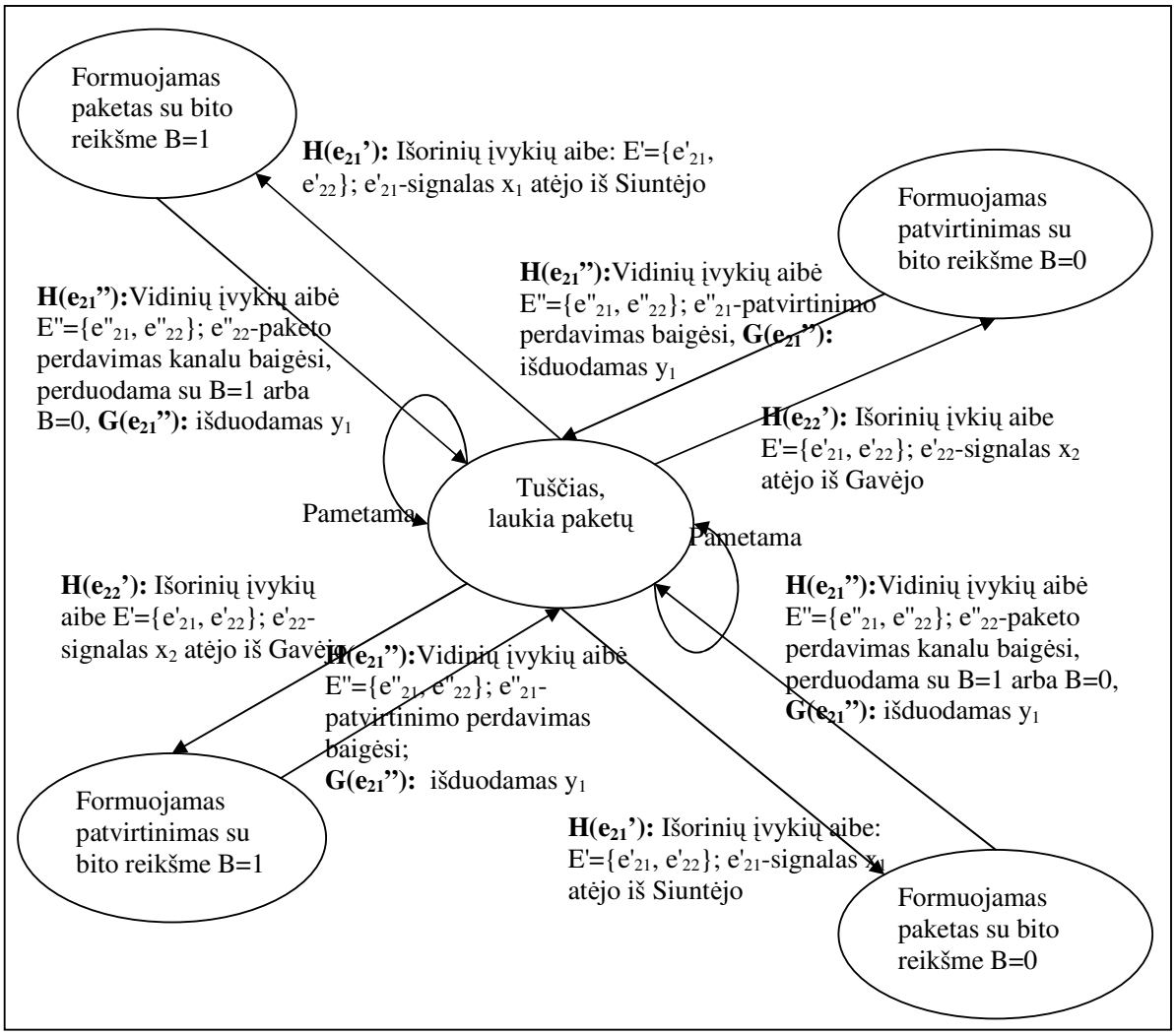
3.1.2. Alternuojančio bito protokolo automatų modelis

Alternuojančio bito protokolas gali būti aprašytas trimis automatais: *Siuntėjas*, *Kanalas*, *Gavėjas*. Šiame skyriuje jie ir pateikiami.



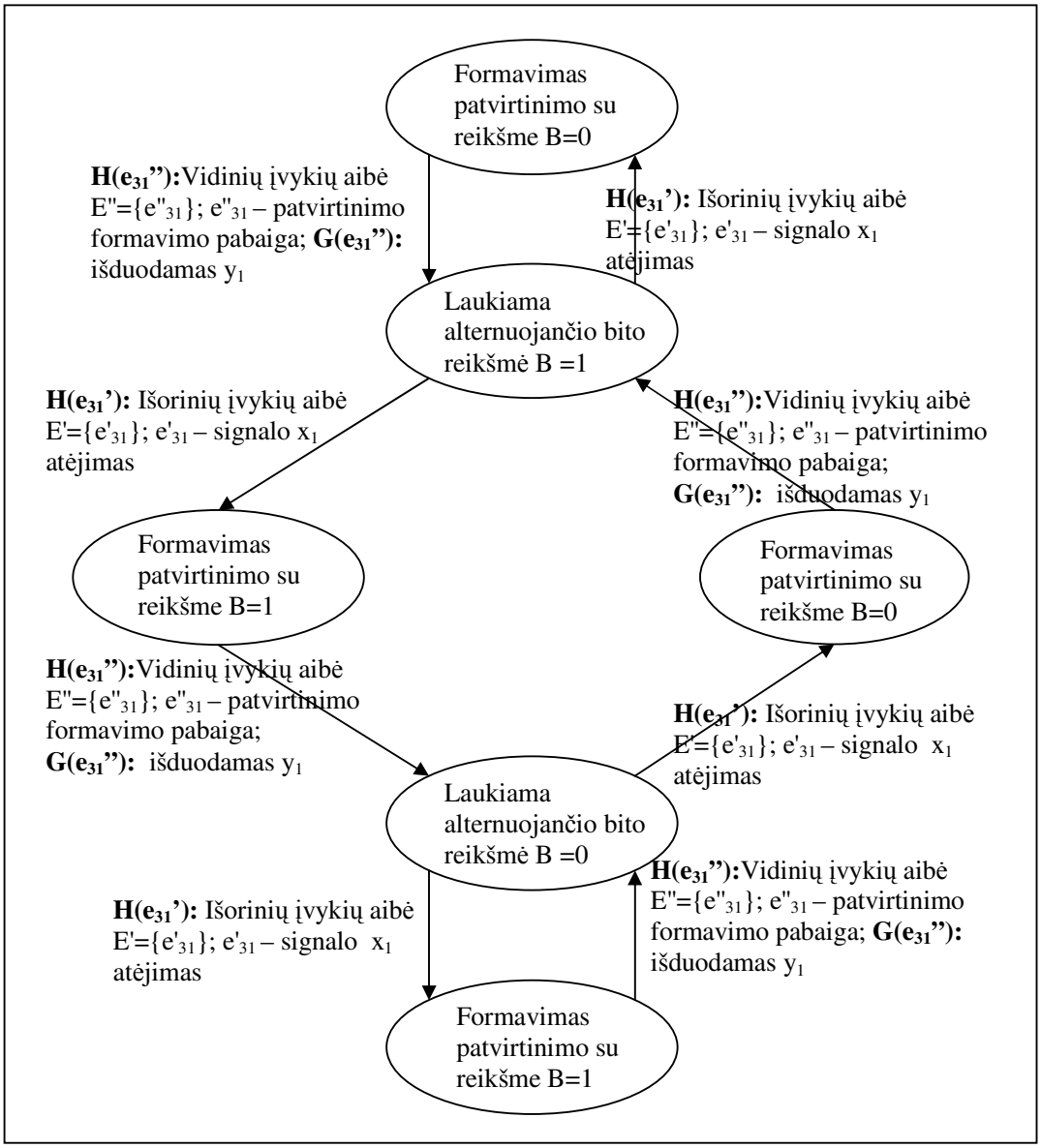
11 pav. Agregato *Siuntėjas* automatas

Agregato *Siuntėjas* automatinis modelis (11 pav.) suformuojamas iš keturių būsenų. Pirmoje būsenoje atliekamas pranešimo formavimas ir paleidžiamas laikmatis. Sekančioje būsenoje laukiamas paketas su atitinkamu numeriu. Perėjimą tarp šių būsenų įtakoja vidinis įvykis e"$_{11}$ (paketas suformuotas siųsti). Grįžimas į pradinę būseną gali būti įtakotas dviejų įvykių. Tai laiko pasibaigimas arba paketo pametimas. Sekančios dvi būsenos ekvivalenčios pirmosioms, tik skiriasi alternuojančio bito reikšmė.



12 pav. Agregato *Kanalas* automatas

Agregato *Kanalas* automatinis modelis pavaizduotas 12 paveikslėlyje. Pradinėje būsenoje *Kanalas* būna tuščias, laukia paketų iš *Siuntėjo* arba *Gavėjo.* Į paketo formavimo būseną pereinama atėjus signalui iš *Siuntėjo*. O grįžtama į pradinę būseną, kai paketas jau perduotas *Gavėjui*. Perėjimą į patirtinimo formavimo būseną įtakoja signalo atėjimas iš *Gavėjo.* Perdavus patvirtinimą *Siuntėjui* grįžtama į pradinę būseną. Paveikslėlyje pavaizduotos būsenos ir perėjimai su abiem (0 ir 1) alternuojančio bito reikšmėm.



13 pav. Agregato *Gavėjas* automatas

Agregato *Gavėjas* automatiniame modelyje (13 pav.) pradinėje būsenoje laukiamas signalas iš *Kanalo* su atitinkama reikšme. Atėjus signalui, pereinama į patvirtinimo formavimo būseną. Kadangi šiuo atveju kanalas modeliuojamas nepatikimas, tai signalas gali ateiti iškraipytas. Taigi, iš laukimo būsenos pereinama į patvirtinimo būseną su tokia alternuojančio bito reikšme, kokia gauta iš kanalo. Perdavus *Kanalui* patvirtinimą, vėl pereinama į laukimo būseną.

3.1.3. Programos tekstas Promela kalba

Ankstesniame skyriuje aprašyti alternuojantį bito protokolą apibrėžiantys automatai. Norint imituoti SPIN sistema, reikia automatus aprašyti Promela kalba. Tai pateikiama 4 lentelėje.

Lentelė Nr. 4.

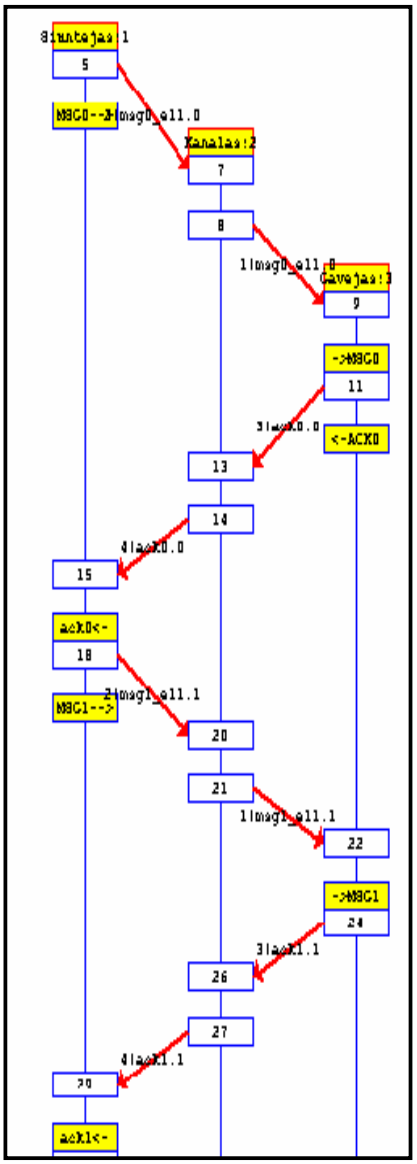
Programos kodas Promela kalba	Komentarai
proctype **Siuntejas**() { bit B; byte any_P11; again: do :: is_siuntejo_aibeY!msg0_e11(B); if :: i_siunteja_aibeX?ack0(B) -> break; :: i_siunteja_aibeX?any_P11(B); ::timeout fi od; do :: is_siuntejo_aibeY!msg1_e11(B); if :: i_siunteja_aibeX?ack1(B) -> break; :: i_siunteja_aibeX?any_P11(B) ; :: timeout fi od; goto again }	Alternuojančio bito reikšmė, B={0,1} Vidinis įvykis: e"$_{11}$-paketas suformuotas siųsti; siunčiamas pranešimas su bit=0 Išorinis įvykis: e'$_{11}$-priimtas signalas x1; gaunamas patvirtinimas su bit=0 P_{11}-patvirtinimo klaidingo perdavimo tikimybe; pametama Vidinis įvykis: e"$_{12}$-baigėsi laikmečiui nustatytas laikas; persiunčiama Vidinis įvykis: e"$_{11}$-paketas suformuotas siusti; siunčiamas pranešimas su bit=1 Išorinis įvykis: e'$_{11}$-priimtas signalas x_1; gaunamas patvirtinimas su bit=1 P_{11}-patvirtinimo klaidingo perdavimo tikimybe; pametama Vidinis įvykis: e"$_{12}$-baigėsi laikmečiui nustatytas laikas; persiunčiama
proctype **Kanalas** () { byte any_P21, any_P22; bit B; do :: is_siuntejo_aibeY?msg0_e11(B); if :: i_gaveja_x1!msg0_e11(B); :: i_gaveja_x1!msg1_e11(B); fi :: is_gavejo_y1?ack0(B) -> i_siunteja_aibeX!ack0(B) :: is_gavejo_y1?any_P22(B); :: is_siuntejo_aibeY?any_P21(B); :: is_siuntejo_aibeY?msg1_e11(B); if :: i_gaveja_x1!msg0_e11(B); :: i_gaveja_x1!msg1_e11(B); fi	Paketo ir patvirtinimo praradimo tikimybes Alternuojančio bito reikšmė, B={0,1} Išorinių įvykių aibe: E'={e'$_{21}$, e'$_{22}$}; e'$_{21}$-signalas x_1 atėjo iš Siuntėjo Vidinių įvykių aibė E"={e"$_{21}$, e"$_{22}$}; e"$_{22}$-paketo perdavimas kanalu baigėsi Išorinių įvykių aibe E'={e'$_{21}$, e'$_{22}$}; e'$_{22}$-signalas x_2 atėjo iš Gavėjo Vidinių įvykių aibė E"={e"$_{21}$, e"$_{22}$}; e"$_{21}$-patvirtinimo perdavimas baigėsi P_{22}-patvirtinimo praradimo tikimybe kanale; pametama P_{12}-paketo praradimo tikimybe kanale; pametama Išorinių įvykių aibe E'={e'$_{21}$, e'$_{22}$}; e'$_{21}$-signalas x_1 atėjo iš Siuntėjo Vidiniu įvykių aibe E"={e"$_{21}$, e"$_{22}$}; e"$_{22}$-paketo perdavimas kanalu baigėsi

:: is_gavejo_y1?ack1(B) ->
 	
i_siunteja_aibeX!ack1(B);
od
}
	Išorinių įvykių aibe E'={e'$_{21}$, e'$_{22}$}; e'$_{22}$-signalas x_2 atėjo iš Gavėjo
	Vidinių įvykių aibė E''={e''$_{21}$, e''$_{22}$}; e''$_{21}$-patvirtinimo perdavimas baigėsi

proctype **Gavejas**() { byte any; bit B;bit B0=0; bit B1=1; again: do :: i_gaveja_x1?msg1_e11(B) -> is_gavejo_y1!ack1(B) ; break; :: i_gaveja_x1?msg0_e11(B)-> is_gavejo_y1!ack0(B) od; do :: i_gaveja_x1?msg0_e11(B) -> is_gavejo_y1!ack0(B); break; :: i_gaveja_x1?msg1_e11(B)-> is_gavejo_y1!ack1(B) od; goto again}	Alternuojančio bito reikšmė, B={0,1} Laukiamas bit =1 Išorinių įvykių aibė E'={e'$_{31}$}; e'$_{31}$-signalo x_1 atėjimas; gaunamas pranešimas su bit=1 Vidinių įvykių aibė E''={e''$_{31}$}; e''$_{31}$-patvirtinimo formavimo pabaiga; patvirtinimas su bit=1 Alternuoja laukiamas bit Išorinių įvykių aibe E'={e'$_{31}$}; e'$_{31}$-signalo x_1 atėjimas; gaunamas pranešimas su bit=0 Vidinių įvykių aibe E''={e''$_{31}$}; e''$_{31}$-patvirtinimo formavimo pabaiga; patvirtinimas su bit=0 Laukiamas bit= 0 Išorinių įvykių aibė E'={e'$_{31}$}; e'$_{31}$-signalo x_1 atėjimas; gaunamas pranešimas su bit=0 Vidiniu įvykių aibe E''={e''$_{31}$}; e''$_{31}$-patvirtinimo formavimo pabaiga; patvirtinimas su bit=0 Alternuoja laukiamas bit Išorinių įvykių aibe E'={e'$_{31}$}; e'$_{31}$-signalo x_1 atėjimas; gaunamas pranešimas su bit=1 Vidiniu įvykių aibe E''={e''$_{31}$}; e''$_{31}$-patvirtinimo formavimo pabaiga; patvirtinimas su bit=1
#define N 1 mtype = {msg0_e11, msg1_e11, ack0, ack1}; chan i_siunteja_aibeX =[N] of {mtype,bit} chan is_siuntejo_aibeY =[N] of {mtype,bit} chan i_gaveja_x1 =[N] of { mtype,bit} chan is_gavejo_y1 =[N] of {mtype,bit}	Įėjimo signalų aibė X={x_1}, x_1=B; B-alternuojančio bito reikšmė patvirtinime, B={0,1} Įsėjimo signalų aibė Y={y_1}, y_1=B; B-alternuojančio bito reikšmė perduodamame pakete, B={0,1] Įėjimo signalai x_1=(B); B – alternuojančio bito reikšmė priimtame pakete Įsėjimo signalai y_1=(B); B – alternuojančio bito reikšmė perduodamame patvirtinime

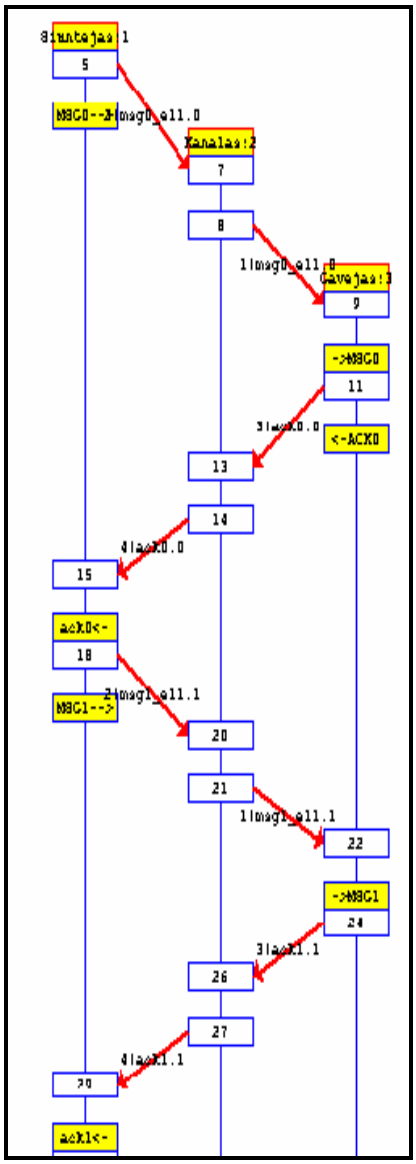
init {
atomic{
run Siuntejas();
run Kanalas();
run Gavejas() }}
	Inicijuojami *Siuntėjo*, *Kanalo* ir *Gavėjo* procesai

3.1.4. Imitavimo eksperimentai



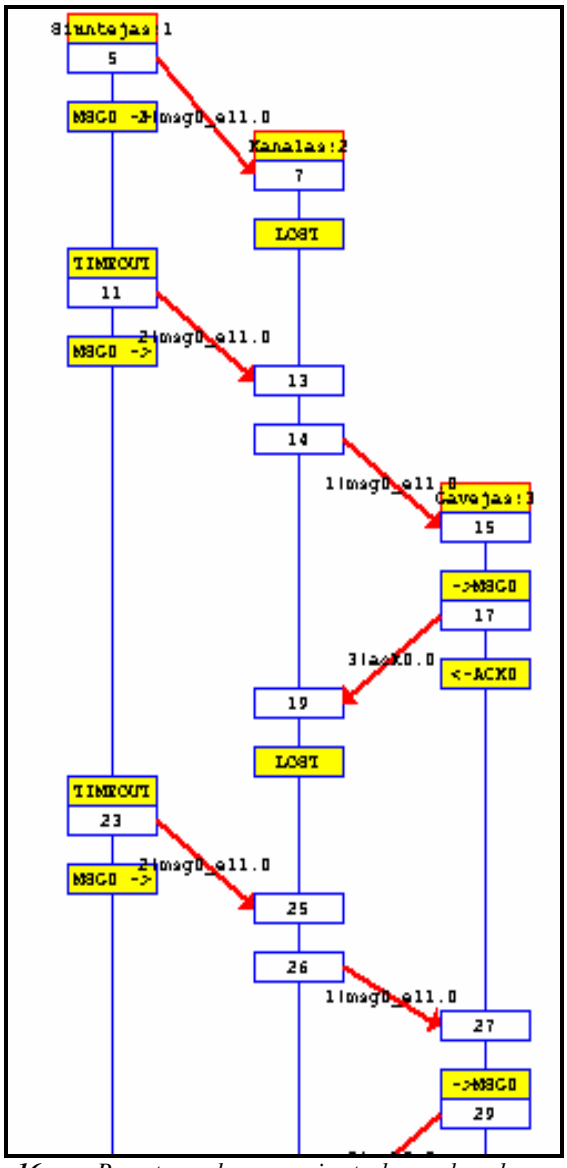
14 pav. *Siuntimas ir gavimas be pametimų.*

Paveikslėlyje (14 pav.) pavaizduota idealiai veikianti sistema. *Siuntėjas* būsenoje numeriu 5 siunčia pranešimą su bitu 0 į *Kanalą. Kanalas* būsenoje numeriu 7 priima pranešimą iš Siuntėjo ir pradeda duomenų formavimą. Iš būsenos numeriu 8 siunčia pranešimą i *Gavėją. Gavėjas* priima duomenis būsenoje numeriu 9 ir pradeda formuota atsakymą. Būsenoje numeriu 11 išsiunčia atsakymą *Kanalui* apie gautus duomenis. *Kanalas* priima atsakymą būsenoje 12 ir suformavęs persiunčia *Siuntėjui*. *Siuntėjas* gavęs patvirtinimą apie sėkmingą duomenų gavima formuoja ir siunčia nauja duomenų paketą bet jau su bitu 1. Vėl visas procesas kartojasi. Ant lankų galime matyti su kokiu bitu (0 ar 1) siunčiamas pranešimas. Patamsinto fono



15 pav. *Siuntimas ir gavimas be pametimų.*

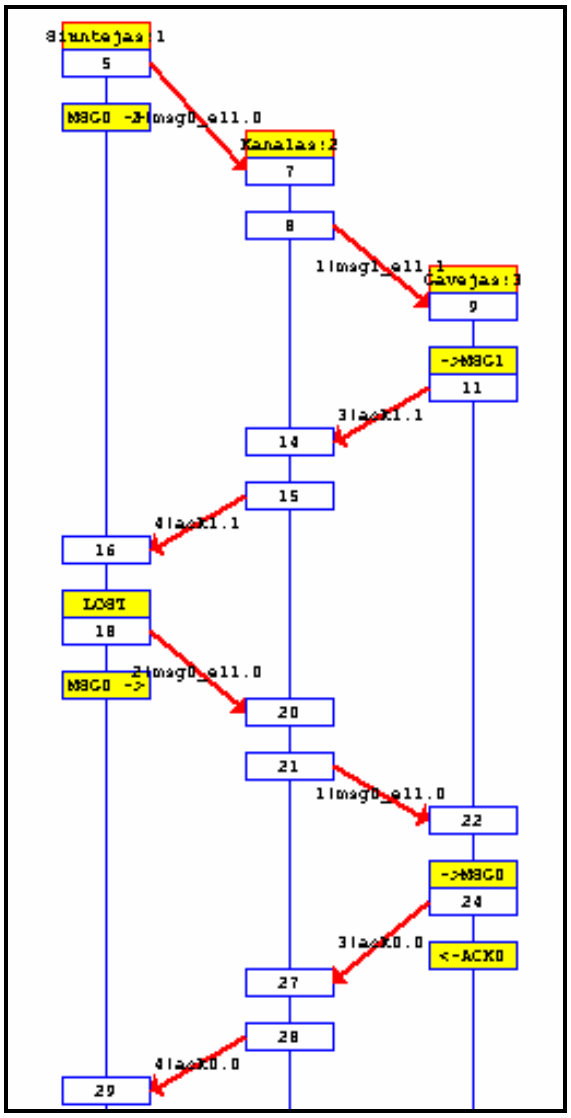
Paveikslėlyje (14 pav.) pavaizduota idealiai veikianti sistema. *Siuntėjas* būsenoje numeriu 5 siunčia pranešimą su bitu 0 į *Kanalą. Kanalas* būsenoje numeriu 7 priima pranešimą iš Siuntėjo ir pradeda duomenų formavimą. Iš būsenos numeriu 8 siunčia pranešimą i *Gavėją. Gavėjas* priima duomenis būsenoje numeriu 9 ir pradeda formuota atsakymą. Būsenoje numeriu 11 išsiunčia atsakymą *Kanalui* apie gautus duomenis. *Kanalas* priima atsakymą būsenoje 12 ir suformavęs persiunčia *Siuntėjui*. *Siuntėjas* gavęs patvirtinimą apie sėkmingą duomenų gavima formuoja ir siunčia nauja duomenų paketą bet jau su bitu 1. Vėl visas procesas kartojasi. Ant lankų galime matyti su kokiu bitu (0 ar 1) siunčiamas pranešimas. Patamsinto fono kvadratukai rodo komentarus.



16 pav. *Pametama duomenys ir atsakymas kanale*

Paveikslėlyje (15 pv.) matome kaip atrodo duomenų perdavimas nepatikimu kanalu kai ir duomenys ir atsakymas gali būti pamesti.

Siuntėjas būsenoje numeriu 5 suformuoja ir siunčia parnešimą su bitu 1 į *Kanalą*. *Kanalas* priima pranešimą ir pradėjęs formavimą jį pameta. Formavimas nutrūksta. Suveikia laikmatis. *Siuntėjas* kartoja tų pačių duomenų siuntimą. Šį kartą *Kanalas* priima duomenis į būseną 13 ir sėkmingai persiunčia *Gavėjui* iš būsenos 14. *Gavėjas* priima duomenis būsenoje 15 ir formuoja atsakymą. Atsakymas išsiunčiamas *Kanalui* iš būsenos 17. *Kanalas* priima atsakymą ir formuoja jo perdavimą siuntėjui. Atsakymas pametamas *Kanale*. Suveikia laikmatis. *Siuntėjas* kartoja tų pačių duomenų siuntimą su tuo pačiu kontroliniu bitu.

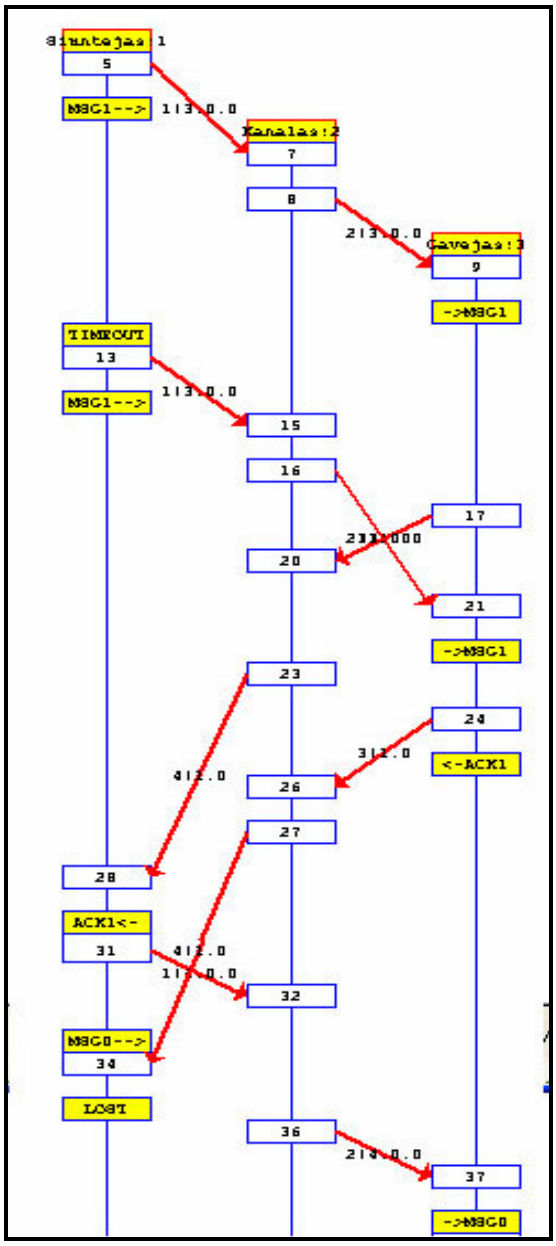


17 pav. *Duomenys iškraipomi.*

Paveikslėlyje (16 pav.) pavaizduota kaip duomenys gali būti iškraipomi nepatikimoje perdavimo terpėje.

Siuntėjas siunčia duomenų paketą su kontroliniu bitu 1 iš būsenos 5 į *Kanalą. Kanalas* priima duomenis ir pradeda formavimą. Formuojant duomenis *Kanale* jie iškraipomi ir dabar jau siunčiami su kontroliniu bitu 0. *Gavėjas* priima duomenis ir suformuoja atsakymą su kontroliniu bitu 0. *Kanalas* priėmęs atsakymą jį persiunčia *Siuntėjui. Siuntėjas* siuntė duomenis su kontroliniu bitu 1, o gavo atsakymą su kontroliniu bitu 0. Šiuo atveju laikoma kad duomenys gauti iškraipyti, todėl *Siuntėjas* pakartotinai persiunčia tuos pačius duomenis.

Iš būsenos 17 išsiunčiami duomenys su kontroliniu bitu 1. Šį karta *Kanalas* priima ir persiunčia teisingus duomenis *Gavėjui*. *Gavėjas* suformuoja atsakymą su reikiamu kontroliniu bitu 1 ir išsiunčia *Kanalui*.



18 pav. *Suveikia pirmalaikis laikmatis*

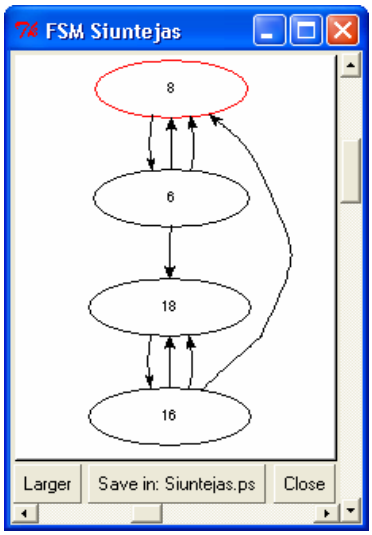
Paveikslėlyje (17 pav.) matome situaciją, kai laikmatis suveikia ankščiau nei gaunamas patvirtinimas apie duomenų gavimą.

Siuntėjas iš būsenos 5 išsiunčia duomenis su kontroliniu bitu 1 į *Kanalą*. *Kanalas* priima duomenis būsenoje 7 ir formuoja persiuntimą *Gavėjui* būsenoje 8. *Gavėjas* būsenoje 9 priima duomenis su laukiamu kontroliniu bitu 1 ir formuoja atsakymą.

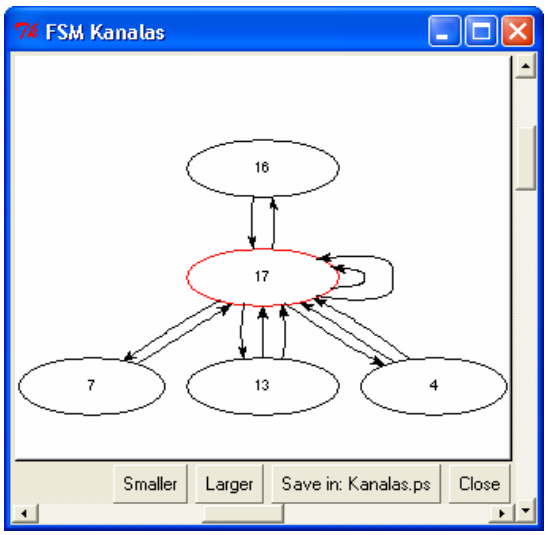
Suveikia *Siuntėjo* laikmatis. *Siuntėjas* kartoja duomenų siuntimą su tuo pačiu kontroliniu bitu 1 iš būsenos 13. *Kanalas* priima ir formuoja duomenų persiuntimą *Gavėjui* būsenose 15 ir 16. *Gavėjas* tuo metu baigia formuoti prieš tai gautų duomenų atsakymą ir jį siunčia *Kanalui* iš būsenos 17. *Kanalas* priima atsakymą į būseną 20. Tuo pačiu *Gavėjas*

priima naujai atėjusius duomenis į būseną 21. O *Kanalas* suformuoja atsakymą ir išsiunčia *Siuntėjui* iš būsenos 23. Tuo pačiu *Gavėjas* baigia formuoti persiųstų duomenų atsakymą būsenoje 24. *Kanalas* priima iš *Gavėjo* siunčiamą atsakymą būsenoje 26 ir formuoja persiuntimą būsenoje 27. Tuo pačiu *Siuntėjas* gauna atsakymą iš *Kanalo* su lauktu kontroliniu bitu 1. Iš karto formuojamas naujų duomenų siuntimas su kontroliniu bitu 0. Tada *Siuntėjas* gauna dar vieną atsakymą apie tų pačių duomenų gavimą. Šis pakartotinis atsakymas ignoruojamas

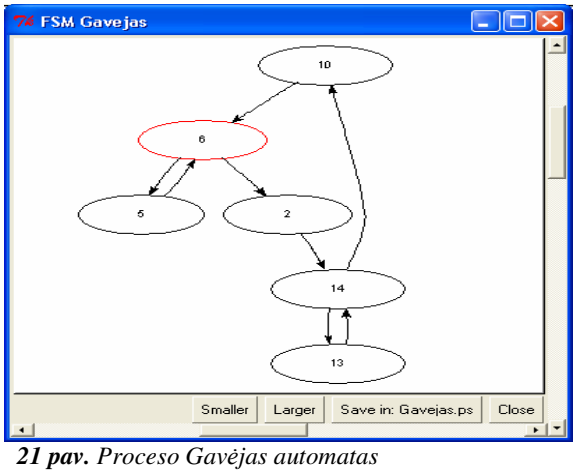
SPIN sistemos automatų peržiūros parinktis leidžia pamatyti struktūrą automato, kurį SPIN naudoja verifikavimo metu. Kiekvienas procesas atvaizduojamas atskiru automatu. Pasirinkus automatų peržiūrą, pirmiausiai SPIN sugeneruoja verifikatorių, sukompiliuoja ir paleidžia vykdymą. Tokiu būdu gaunami procesų vardai ir atitinkamas automatas.



19 pav. *Proceso Siuntėjas automatas*



20 pav. *Proceso Kanalas automatas*



21 pav. *Proceso Gavėjas automatas*

3.1.5. Verifikavimo eksperimentai

Norint susidaryti pirmą įspūdį apie nagrinėjamo protokolo sudėtingumą, galima atlikti išsamų verifikavimą su SPIN. Šio verifikavimo metu gali būti patikrintos pagrindinės saugomo savybės, tokios kaip:

– sistemos apribojimų pažeidimai,

– specifikacijos nepilnumas (specification incompleteness),

– specifikacijos pertekliškumas-mirties kodas(specification redundancy),

– aklavietės(deadlocks), amžini ciklai(livelocks), badavimas(starvation).

Modelių logikos tikrinimo programa SPIN skirta rasti daugybes loginių ir funkcinių klaidų:

– lenktynių padėtis(race conditions),

– blokavimo problemos(locking problems), pirmumo problemos(priority problems),

– resursų paskirstymo klaidos,

– loginės problemos: trūksta priežastinių ar laikinų ryšių(missing causal or temporal relations).

Alternuojančio bito protokolo verifikavimas įvairiais aspektais

Tikrinsime teisingumo savybes:

Saugumo būsenų savybes (Safety state properties);

Neteisingos pabaigos būsenas(Invalid end states);

Nepasiekiamas būsenas(Report Unreachable Code).

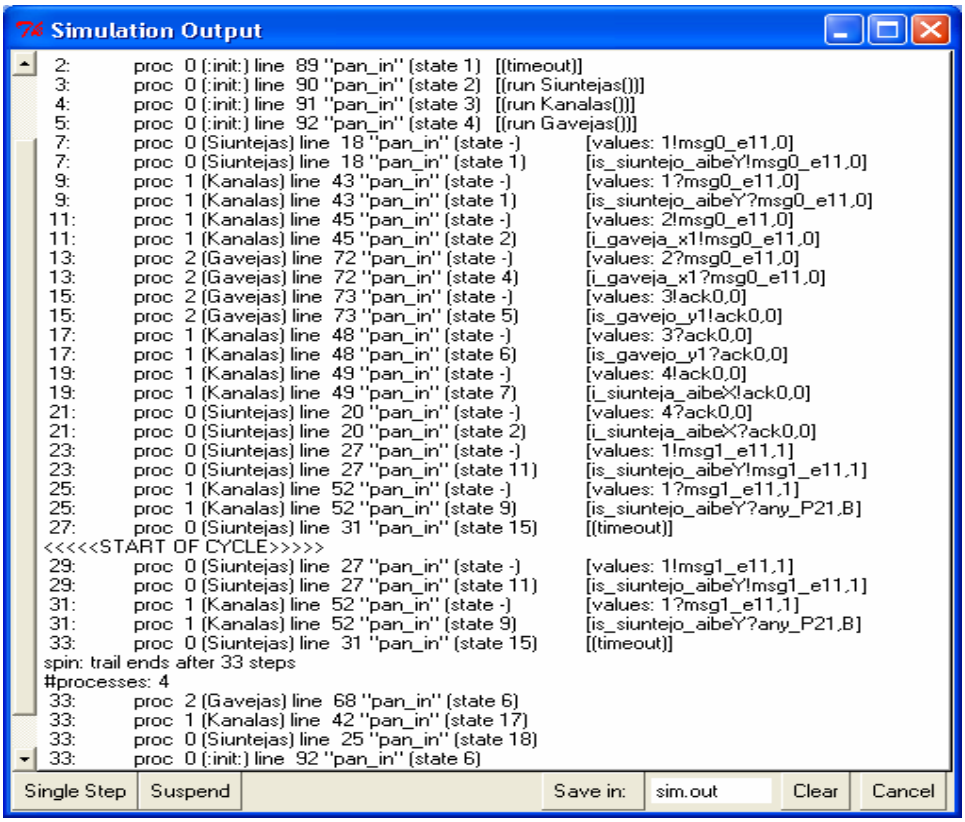
Verifikavimas bus vykdomas nuodugniu paieškos metodu(Exhaustive), o esant pilnai eilei pranešimai bus blokuojami(Blocks New Msgs).

Lentelė Nr. 5. Vrifikavimo eksperimento analizė

Verifikavimo rezultatai	Paaiškinimai
(SPIN Version 3.4.14 -- 6 April 2002	SPIN versija kuri sugeneruoja failą pan.c. Iš šio failo vykdomas verifikavimas.
+ Partial Order Reduction	Tikrinimo būdas. Pliuso ženklas reiškia, kad šis algoritmas buvo naudojamas pagal nutylėjimą. Minuso ženklas reikštų kompiliavimą nuodugnia, nesumažinta paieška su parinktimi -DNOREDUCE .
Full statespace search for:	Parodo paieškos metodą. Pagal nutylėjimą atliekama pilna būsenų paieška. Labai dideli modeliai gali būti tikrinami Bitstate paieška, kuri yra apytikslė.
never-claim - (not selected)	Niekada nesikreipiama. Minuso ženklas reiškia, kad nebuvo tikrinama. Jei „never claim“ buvo modelio dalis, gali būti kad buvo nuslopinta su parinktimi -DNOCLAIM .
assertion violations - (disabled by -A flag)	Tvirtinimų pažeidimai. Minusas reiškia, kad paieškoje nebuvo tikrinama ar yra neprogresyvių ciklų. Norint tai atlikti reikia nustatyti ”run-time” parinktį **-a** arba kompiliuoti -DNP sujungtą su ”run-time” parinktimi **-l**.

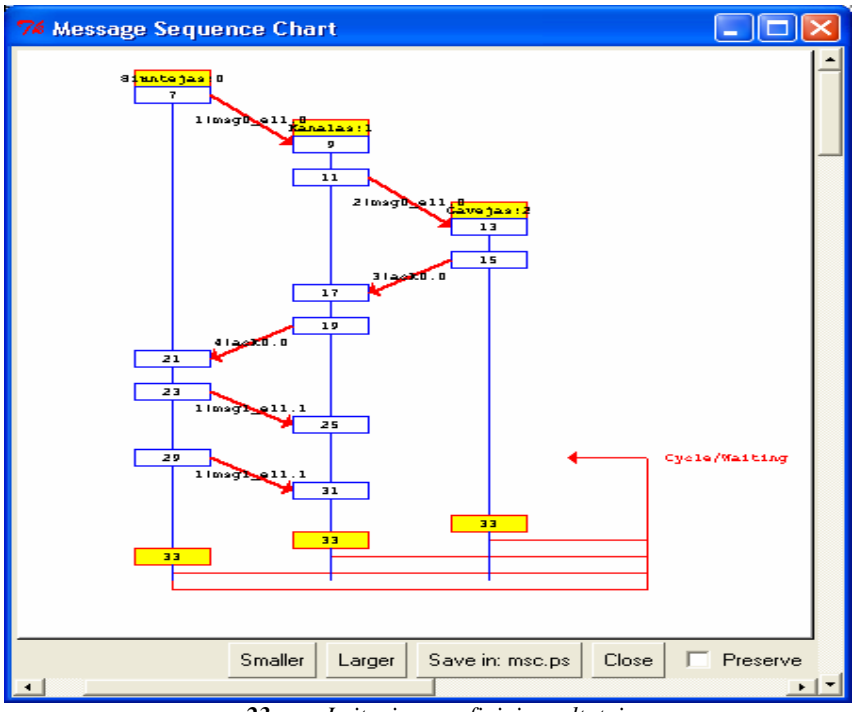
cycle checks - (disabled by -DSAFETY)	Ciklų paieška-netikrinta
invalid endstates +	Neteisingos pabaigos būsenų nerasta
State-vector 64 byte, depth reached 27, errors: 0	Visos sistemos pilnas testavimas pareikalavo 64 baitų atminties(kiekvienai būsenai). Ilgiausias paieškos gylio kelias turėjo 27 perėjimus nuo pradinės sistemos būsenos. Klaidų nerasta.
45 states, stored	Sukurtos 45 globalios būsenos
15 states, matched	15 būsenų atitiko, t.y paieškoje grįžo i jau tikrintas būsenas paieškos medyje.
60 transitions (= stored+matched)	Viso buvo ištirta 60 perėjimų.
3 atomic steps	3 nedalomi (atomic) žingsniai.
hash conflicts: 7 (resolved)	7 atvejais „hash“ schema aptiko procesų susidūrimus ir buvo priversta perkelti būsenas į nuorodų sąrašą „hash“ lentelėje.
(max size 2^19 states)	Argumentas(pagal nutylėjimą), kuris nusako „hash“ lentelės dydį. Atitinka „run-time“ parinktį **-w18**.
2.542 memory usage (Mbyte)	Šiai paieškai panaudota 2.542 Mb atminties resursų įtraukiant steką, „hash“ lentelę ir visas duomenų struktūras. Su parinktimis **-m** ir **–w** galima sumažinti reikalaujamą atminties kiekį.

Kitu atveju, kai randame klaidų, imitavimo(simulation) rezultatuose galime matyti SPIN testavimo žingsnius ir šiuo atveju (21 pav.) pažymėtą neprogresyvaus ciklo pradžią.



22 pav. *Imitavimo rezultatai*

Grafiškai taip pat galime matyti kad kanale sistema aptiko neprogresyvų ciklą.



23 pav. *Imitavimo grafiniai rezultatai*

Nagrinėjamas alternuojančio bito protokolo modelis teiginiais (šiuo atveju „never“).

```
1 /* alternuojančio bito protokolas su nepatikimu kanalu */
2
3  #define N 1
4
5  mtype = {msg0_e11, msg1_e11, ack0, ack1};
6  chan i_siunteja_aibeX =[N] of {mtype,bit}
7  chan is_siuntejo_aibeY =[N] of {mtype,bit}
8  chan i_gaveja_x1 =[N] of { mtype,bit}
9  chan is_gavejo_y1 =[N] of {mtype,bit}
10 proctype Siuntejas()
11 {     bit B; byte any_P11;
12 again:
13       do
14       :: is_siuntejo_aibeY!msg0_e11(B);
15                if
16                :: i_siunteja_aibeX?ack0(B) -> break;
17                :: i_siunteja_aibeX?any_P11(B);
18                :: timeout
19                fi
20       od;
21       do
22       :: is_siuntejo_aibeY!msg1_e11(B);
23                if
24                :: i_siunteja_aibeX?ack1(B) -> break;
25                :: i_siunteja_aibeX?any_P11(B) ;
26                :: timeout
27                fi
28       od;
```

```
29      goto again
30 }
31
32 proctype Kanalas ()
33 {    byte any_P21, any_P22; bit B;
34
35      do
36      ::  is_siuntejo_aibeY?msg0_e11(B);
37              if
38              :: i_gaveja_x1!msg0_e11(B);
39              :: i_gaveja_x1!msg1_e11(B);
40              fi
41      ::  is_gavejo_y1?ack0(B) -> i_siunteja_aibeX!ack0(B)
42      ::  is_gavejo_y1?any_P22(B);
43      ::  is_siuntejo_aibeY?any_P21(B);
44      ::  is_siuntejo_aibeY?msg1_e11(B);
45              if
46              :: i_gaveja_x1!msg0_e11(B);
47              ::i_gaveja_x1!msg1_e11(B);
48              fi
49      ::  is_gavejo_y1?ack1(B) ->i_siunteja_aibeX!ack1(B);
50      od
51 }
52 proctype Gavejas()
53 {     byte any; bit B;
54 again:
55      do
56      :: i_gaveja_x1?msg1_e11(B) -> is_gavejo_y1!ack1(B) ;
57         break;
58      :: i_gaveja_x1?msg0_e11(B)-> is_gavejo_y1!ack0(B)
59      od;
60      do
61      :: i_gaveja_x1?msg0_e11(B) -> is_gavejo_y1!ack0(B);
62         break;
63      :: i_gaveja_x1?msg1_e11(B)->is_gavejo_y1!ack1(B)
64      od;
65      goto again
66 }
67 init { atomic{run Siuntejas(); run Kanalas(); run Gavejas() }}
```

Pranešimų praradimas šiame modelyje sumodeliuotas aiškiai siuntėjo (Sender) ir priėmėjo (Receiver) procesuose (16, 24, 57, 62 eilutės) su prielaida, kad pranešimai gali būti "pamesti". Norėdami patikrinti teiginį: "visada yra teisinga tai, kad siuntėjui išsiuntus pranešimą, priėmėjas jį gaus" – formuojame laikiną never teiginį:

```
never {
        do
        :: skip
        :: i_gaveja_x1?[msg0_e11] -> goto accept0
        :: i_gaveja_x1?[msg1_e11] -> goto accept1
        od;
```

```
accept0:
        do
        :: !i_gaveja_x1?[msg0_e11]
        od;
accept1:
        do
        :: !i_gaveja_x1?[msg1_e11]
        od
}
```

Kai tik pereinama į būsena accept0 (accept1), teiginys gali likti šioje būsenoje, jei priėmėjo procesas niekada nesulauks pranešimo su tuo pačiu eilės numeriu. Laikinam never teiginiui bus gaunamas prieštaravimas, tokiu būdu užtikrinant minėtos savybės korektiškumą.

3.2. Duomenų perdavimo protokolo su adaptyviu komutavimo metodu verfikavimas

Perdavinėjami dviejų tipų srautai: failiniai seansai ir paketai.

Šis protokolas skirsto aptarnavimo kanalus į: failų aptarnavimo kanalus nf(t), paketų aptarnavimo kanalus np(t) ir bendrus kanalus nfp(t) kur aptarnaujami ir failai, ir paketai.

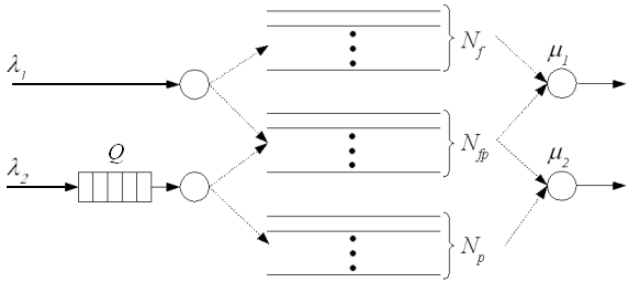
Jei trūksta kanalų failams, tai jie gali užimti visus bendrus kanalus, taip pat jei trūksta kanalų paketams, tai jie gali užimti visus bendrus kanalus.

Failai neefektyviai išnaudoja kanalus, todėl pauzių metu failiniais kanalais gali būti perduoti paketai.

Failai aptarnaujami, jei yra laisvų kanalų nf(t) arba nfp(t), kitaip failai prarandami. Kai visi kanalai nf(t) užimti, failai užima laisvus nfp(t) kanalus tik tuomet, kai paketų skaičius eilėje Q<L1 (0<L1<L2<L).

Paketai aptarnaujami np(t) kanalais, laisvais kanalais nfp(t). Tam, kad neprarastume ateinančių paketų jie buferizuojami į eilę Q, kurios dydis ribojamas L. Kai buferis prisipildo, siekiant išvengti paketų praradimų, paketų aptarnavimui suteikiamas didesnis prioritetas ir jie gali užimti daugiau bendrų kanalų nfp(t). Taigi, kai paketų eilės Q ilgis viršija L2, atėjęs paketas atima vieną kanalą nfp(t) išstumdamas failą.

3.2.1. Duomenų perdavimo protokolo su adaptyviu komutavimo metodu agregatinė specifikacija



24 pav. *Duomenų perdavimo protokolo sistema*

Agregato Siuntėjas aprašymas:

1. Įėjimo signalų aibė:

$X = \varnothing$

2. Išėjimo signalų aibė

$Y = \varnothing$

3. Išorinių įvykių aibė

$E' = \varnothing$

4. Vidinių įvykių aibė

(Visi įvykiai, kurie keičia sistemos būseną)

E" = {e_1", e_2", e_3", e_4"}

e_1'' – atėjo failinė sesija;

e_2'' – atėjo paketas;

e_3'' – baigėsi failinės sesijos aptarnavimas;

e_4'' – baigėsi paketo aptarnavimas.

5. Valdymo sekos:

$e_1'' \rightarrow \lambda_1$, $e_2'' \rightarrow \lambda_2$, $e_3'' \rightarrow \mu_1$, $e_4'' \rightarrow \mu_2$

6. Diskrečioji būsenos dedamoji

$\upsilon(t_m)$ = { Nfpf(t_m), Nfpp(t_m), n(t_m) };

Nf(t_m) – f.ailinės sesijos kanalo užimtumas;

Nfp(t_m) – paketų kanalo užimtumas;

n(t_m) – paketų eilė Q laiko momentu t.

7. Tolydžioji būsenos dedamoji

$z_\upsilon(t_m)$ = {w(e_1",t_m), w(e_2", t_m), w(e_3",t_m), w(e_4",t_m)}

8. Pradinė būsena

$z(t_0)$ = {0, 0, 0, ∞, ∞, ∞, ∞ }.

9. Perėjimo ir išėjimo operatoriai:

$H(e_1'')$:

$$n_f(t+0) = \begin{cases} n_f(t)+1, \text{if } n_f(t) < n_f, \\ n_f(t), \text{kitu atveju;} \end{cases}$$

$$n_{fp}(t+0) = \begin{cases} n_{fp}(t)+1, \text{if } (n_f(t) = N_f) \wedge (n_{fp} < N_{fp}) \wedge (n < l_1), \\ n_{fp}, \text{kitu atveju;} \end{cases}$$

$$n(t+0) = n(t);$$

;

;

$$w(e_4'', t+0) = \begin{cases} 1, if \;\; n(t) > 0 \\ 0, kitu\; atveju \end{cases}$$

$H(e_2'')$:

$$n_f(t+0) = n_f(t);$$

$$n_{fp}(t+0) = \begin{cases} n_{fp}(t)-1, \text{if } (n_{fp} > 0) \wedge (n < l_2), \\ n_{fp}(t), \text{kitu atveju;} \end{cases}$$

$$n(t+0) = n(t)+1;$$

;

;

$$w(e_4'', t+0) = \begin{cases} 1, if \;\; n(t) > 0 \\ 0, kitu\; atveju \end{cases}$$

$H(e_3'')$:

$$n_f(t+0) = \begin{cases} n_f(t)-1, \text{if } (n_f > 0) \wedge n_{fp}(t) = 0, \\ n_f(t), \text{kitu atveju;} \end{cases}$$

$$n_{fp}(t+0)=\begin{cases} n_{fp}(t)-1, \text{if } n_{fp}>0, \\ n_{fp}(t), \text{kitu atveju;} \end{cases}$$

$$n(t+0)=n(t);$$

;

;

$$w(e_4'',t+0)=\begin{cases} 1, if \;\; n(t)>0 \\ 0, kitu\; atveju \end{cases}$$

$$H(e_4''):$$

$$n_f(t+0)=n_f(t);$$

$$n_{fp}(t+0)=n_{fp}(t);$$

$$n(t+0)=\begin{cases} n(t)-1, \text{if } n(t)>0, \\ 0, \text{kitu atveju;} \end{cases}$$

;

;

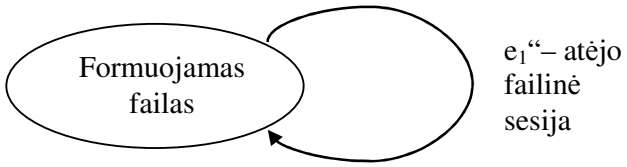
;

$$w(e_4'',t+0)=\begin{cases} 1, if \;\; n(t)>0 \\ 0, kitu\; atveju \end{cases}$$

3.2.2. Duomenų perdavimo protokolo su adaptyviu komutavimo metodu automatų modelis

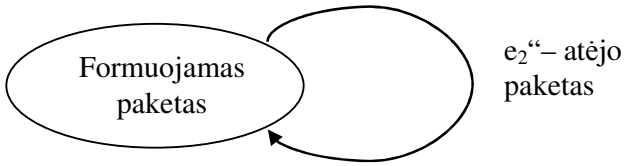
Duomenų perdavimo protokolas, su adaptyviu komutavimo metodu, gali būti aprašytas automatais: Siuntėjas1, Siuntėjas2, Kanalas1, Kanalas2. Šie automatai čia ir pateikiami.

Siuntėjas1

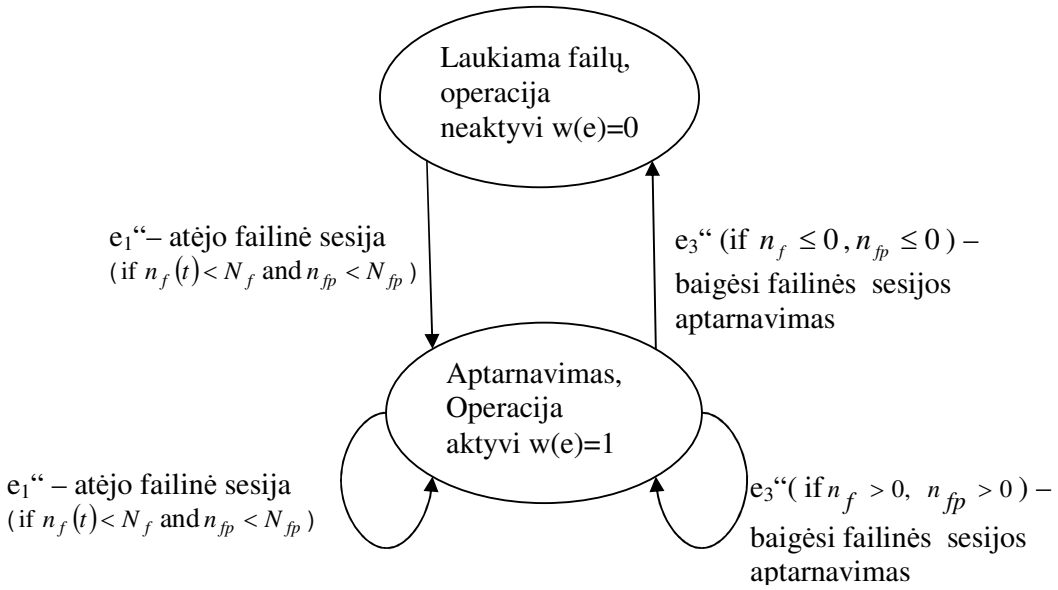


25 pav. *Siuntėjo siunčiančio failus automatinis modelis*

Siuntėjas2

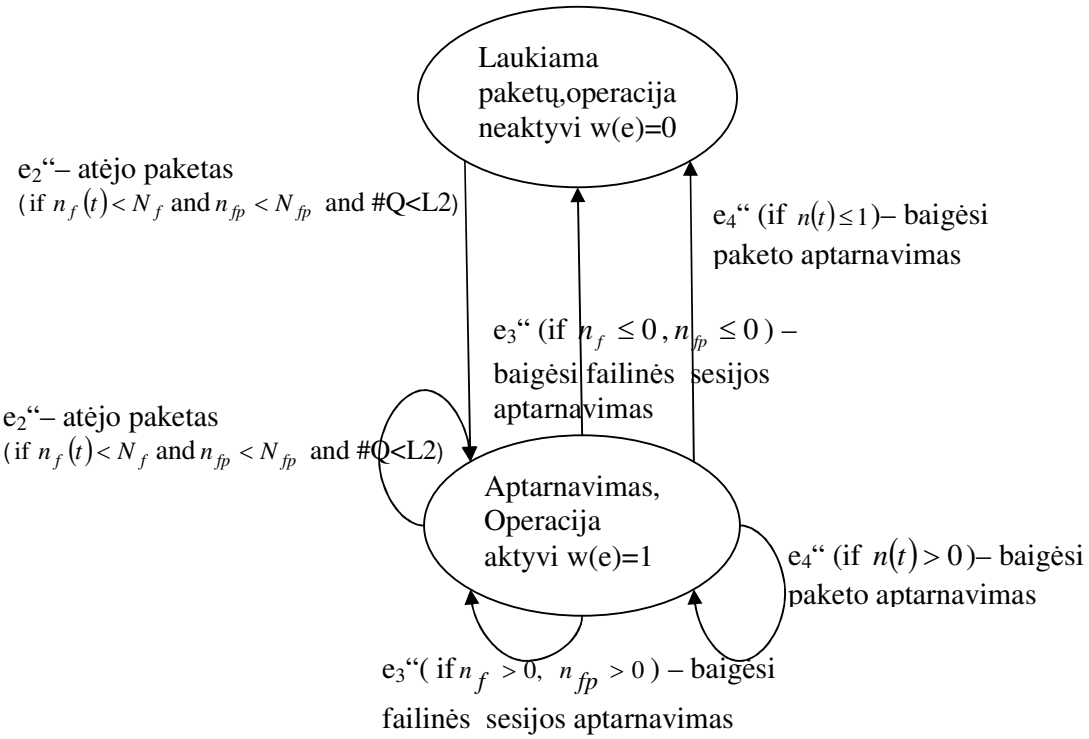


26 pav. *Siuntėjo siunčiančio paketus automatinis modelis*



27 pav. *Kanalo1 automatinis modelis*

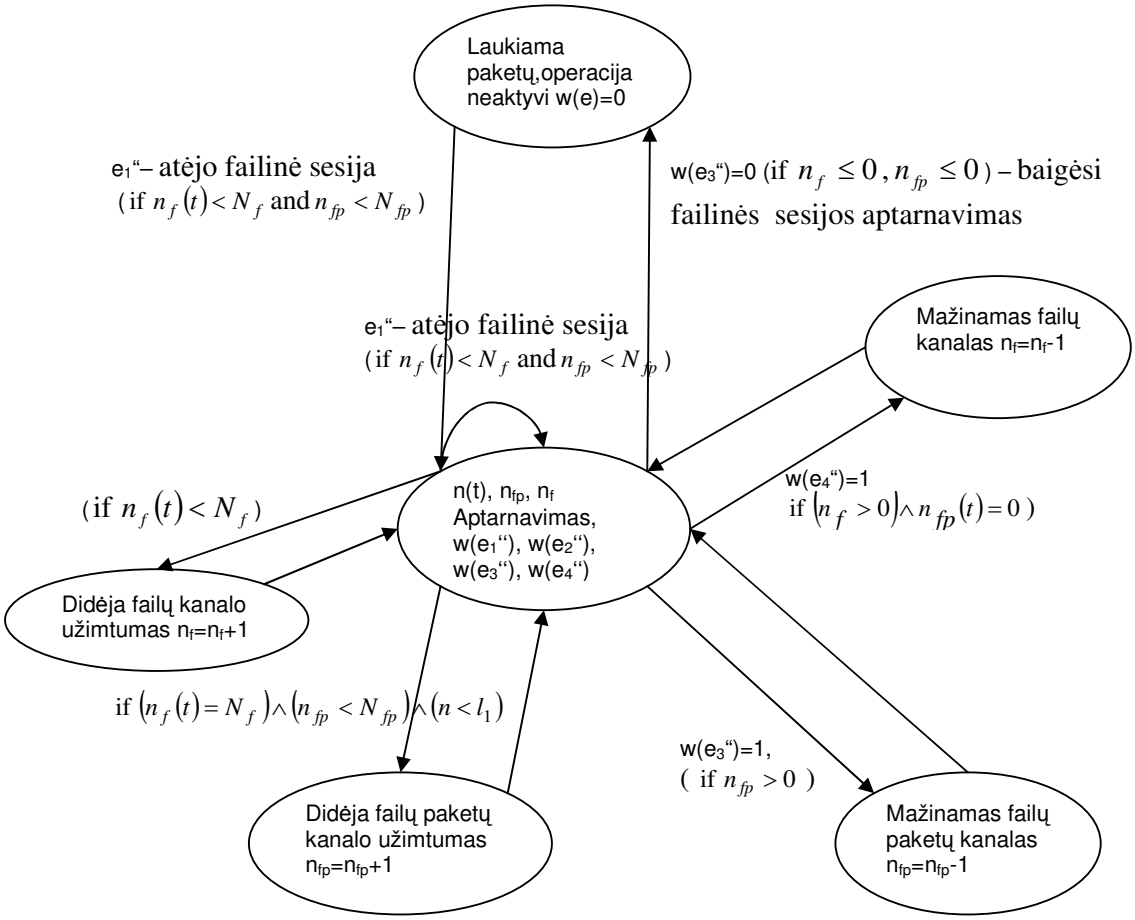
Automatą (27 pav.) sudaro dvi būsenos: pradinėje būsenoje laukiama failų ir operacija neaktyvi, tolydi koordinatė w(e)=0; antroje būsenoje atliekamas aptarnavimas, tolydi koordinatė tampa w(e)=1. Įvykus įvykiui e_1“ (atėjo failinė sesija) pereinama iš laukimo būsenos į aptarnavimo būseną. Šio įvykio metu tikrinamos kanalų pildymo sąlygos ir atitinkamai užpildomi kanlai, šiuo atveju arba failų kanalas, arba failų ir paketų kanalas. Aptarnavimo būsenoje taip pat gali įvykti įvykis e_1“, t.y. aptarnavimo metu gali ateiti failinė sesija. Įvykus įvykiui e_3“ (baigėsi failinės sesijos aptarnavimas) grįžtama į pradinę būseną kai tenkinama sąlyga1. Taip pat ir aptarnavimo metu gali įvykti įvykis e_3“ jei tenkinama sąlyga.



28 pav. *Kanalo2 automatinis modelis*

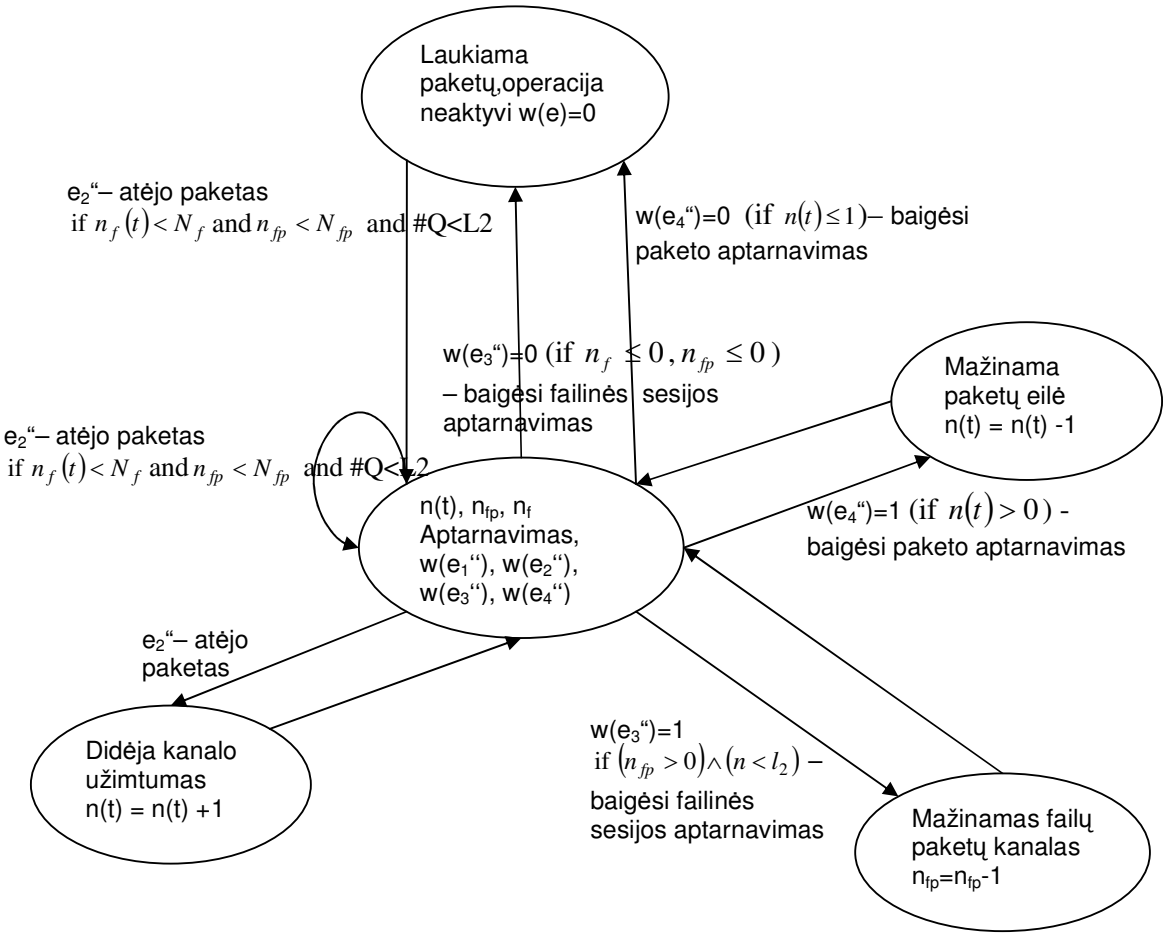
Automatą (28 pav.) sudaro dvi būsenos: pradinėje būsenoje laukiama paketų ir operacija neaktyvi, tolydi koordinatė w(e)=0; antroje būsenoje atliekamas aptarnavimas, tolydi koordinatė tampa w(e)=1. Įvykus įvykiui e_2“ (atėjo paketas) pereinama iš laukimo būsenos į aptarnavimo būseną. Šio įvykio metu tikrinamos kanalų pildymo sąlygos ir atitinkamai užpildomi kanlai, šiuo atveju arba paketų kanalas arba failų ir paketų kanalas. Aptarnavimo būsenoje taip pat gali įvykti įvykis e_2“, t.y. aptarnavimo metu gali ateiti paketai. Įvykus įvykiui e_4“ (baigėsi paketo aptarnavimas) grįžtama į pradinę būseną kai tenkinama sąlyga2. Taip pat ir aptarnavimo metu gali įvykti įvykis e_3“ jei tenkinama sąlyga3. Kadangi pauzių metu paketai gali būti aptarnaujami ir failų kanale, tai automate matomas įvykis e_3“.

Automatai (*Siuntėjas1, Siuntėjas2, Kanalas1, Kanalas2*) yra aukšto abstrakcijos lygio, nes sudaryti tik iš operacijų ir įvykių, nevertinant likusių kintamųjų naudojamų PLA aprašyme. Taip galima verifikuoti procesų komunikavimą. Vėliau automatai gali būti palaipsniui detalizuojami, įvedant naujas būsenas ir perėjimus. Šis detalizavimas vykdomas kol pasiekiamas PLA abstrakcijos lygis t.y., PLA specifikacija pilnai aprašoma automatais.
Sekančiame paveikslėlyje pateiktas papildytas automatas *Kanalas1*, įvedant būsenas ir perėjimus, kurie formuoja paketinius ir failinius kanalus.



29 pav. *Kanalo1 išplėstas automatinis modelis*

Sekančiame paveikslėlyje pateiktas papildytas automatas *Kanalas2*.



30 pav. *Kanalo2 išplėstas automatinis modelis*

3.2.3. Programos tekstas Promela kalba

Norint imituoti SPIN sistema, reikia aukščiau sudarytus automatus aprašyti Promela kalba. Tai pateikiama šioje lentelėje kartu su agregatiniu metodu. Datalizutų automatų realizacija Promela kalba pateikta pirmame priede.

Lentelė Nr. 6.

Programos kodas Promela kalba	Komentarai

proctype **Siuntejas1**()
{
again: do
:: is_siuntejo_aibeF!f;
goto again;
od
}
proctype **Siuntejas2**()
{
again: do
:: is_siuntejo_aibeP!p;
goto again;
od;
}
	Vidinių įvykių aibė E” = {e1”, e2”, e3”, e4”}; e1“– atėjo failinė sesija
	
	Vidinių įvykių aibė E” = {e1”, e2”, e3”, e4”}; e2“– atėjo paketas

proctype Kanalas1 () { a: do :: is_siuntejo_aibeF?f ;	

do
:: is_siuntejo_aibeF?f ;
 	
:: nf<=1 && nfpf<=1; goto a
 	
:: nf>1 || nfpf>1;
od;
od
}
proctype Kanalas2 ()
{
a: do
:: is_siuntejo_aibeP?p;
do
:: is_siuntejo_aibeP?p;
 	
:: Q<=1;goto a
 	
:: Q>1 ;
 	
:: Q<=0;goto a
 	
:: Q>0;
od;
od
}
	e_1"-signalas f atėjo iš Siuntėjo1 kai operacija neaktyvi w(e)=0
	e_1"-signalas f atėjo iš Siuntėjo1 kai operacija aktyvi w(e)=1
	e_3" (sąlyga1)– baigėsi failinės sesijos aptarnavimas; w(e)=0
	e_3" (sąlyga1)– baigėsi failinės sesijos aptarnavimas; w(e)=1
	
	e_2"-signalas p atėjo iš Siuntėjo2 kai operacija neaktyvi w(e)=0
	e_2"-signalas p atėjo iš Siuntėjo2 kai operacija aktyvi w(e)=1
	e"$_3$- baigėsi paketo aptarnavimas; w(e)=0
	e"$_3$- baigėsi paketo aptarnavimas; w(e)=1
	
	e"$_4$- baigėsi failinės sesijos aptarnavimas; w(e)=0
	e"$_4$- baigėsi failinės sesijos aptarnavimas; w(e)=1

#define N 1 #define M 6 #define MAX 3 mtype = { f, p}; chan is_siuntejo_aibeF =[N] of {mtype} chan is_siuntejo_aibeP =[N] of {mtype} int nf,nfpf,nfpp,np; int Q=0,L2=2,L1=1,L=3;	Globalūs kintamieji pranesimu tipas siunčiamų failų kanalas siunčiamų paketų kanalas

init {
atomic{
run Siuntejas1();
run Siuntejas2();
run Kanalas1();
run Kanalas2();
}
	Paleidžiami dirbti agregatai

3.2.4. Imitavimo eksperimentai

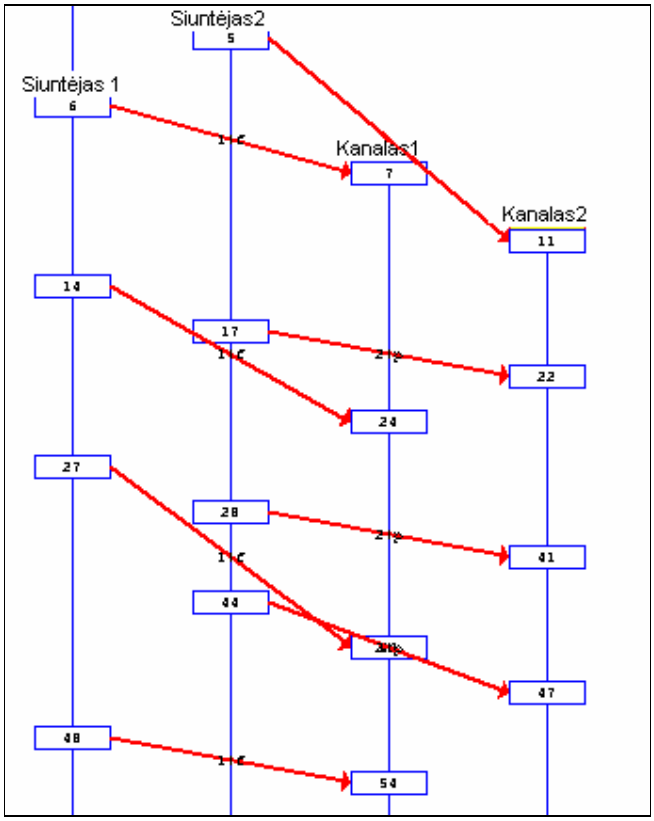
Imitavimo metu galima įsitikinti ar protokolas veikia taip kaip ir planuota, taip pat galima klaidos atveju pasižiūrėti koks kelias veda į klaidingą būseną.

Lentelė Nr. 7. Imitavimas

Žingsnis	Proceso numeris	Proceso pavadinimas	Kodo eilutės nr	Failas iš kurio vykdoma	Būsenos numeris imitavimo metu	Vykdomas programos tekstas ir kintamųjų reikšmės
0:	proc - (:root:) creates proc 0 (:init:)					
1:	proc 0 (:init:) creates proc 1 (Siuntejas1)					
1:	proc 0 (:init:) line 57 "pan_in" (state 5)					[(run Siuntejas1())]
2:	proc 0 (:init:) creates proc 2 (Siuntejas2)					
2:	proc 0 (:init:) line 59 "pan_in" (state 2)					[(run Siuntejas2())]

3:	proc 0 (:init:) creates proc 3 (Kanalas1)	
3:	proc 0 (:init:) line 60 "pan_in" (state 3)	[(run Kanalas1())]
4:	proc 0 (:init:) creates proc 4 (Kanalas2)	
4:	proc 0 (:init:) line 61 "pan_in" (state 4)	[(run Kanalas2())]
5:	proc 2 (Siuntejas2) line 22 "pan_in" (state -)	[values: 2!p]
5:	proc 2 (Siuntejas2) line 21 "pan_in" (state 3)	[is_siuntejo_aibeP!p]
6:	proc 1 (Siuntejas1) line 14 "pan_in" (state -)	[values: 1!f]
6:	proc 1 (Siuntejas1) line 13 "pan_in" (state 3)	[is_siuntejo_aibeF!f]
7:	proc 3 (Kanalas1) line 31 "pan_in" (state -)	[values: 1?f]
7:	proc 3 (Kanalas1) line 29 "pan_in" (state 9)	[is_siuntejo_aibeF?f]
8:	proc 3 (Kanalas1) line 37 "pan_in" (state 7)	[.(goto)]
9:	proc 3 (Kanalas1) line 32 "pan_in" (state 6)	[(((nf<=1)&&(nfpf<=1)))]
10:	proc 1 (Siuntejas1) line 15 "pan_in" (state 2)	[goto again]
11:	proc 4 (Kanalas2) line 46 "pan_in" (state -)	[values: 2?p]
11:	proc 4 (Kanalas2) line 44 "pan_in" (state 12)	[is_siuntejo_aibeP?p]
12:	proc 2 (Siuntejas2) line 23 "pan_in" (state 2)	[goto again]
13:	proc 3 (Kanalas1) line 34 "pan_in" (state 4)	[goto a]
14:	proc 1 (Siuntejas1) line 14 "pan_in" (state -)	[values: 1!f]
14:	proc 1 (Siuntejas1) line 13 "pan_in" (state 3)	[is_siuntejo_aibeF!f]
15:	proc 4 (Kanalas2) line 54 "pan_in" (state 10)	[.(goto)]
16:	proc 4 (Kanalas2) line 47 "pan_in" (state 9)	[((Q<=0))]
17:	proc 2 (Siuntejas2) line 22 "pan_in" (state -)	[values: 2!p]
17:	proc 2 (Siuntejas2) line 21 "pan_in" (state 3)	[is_siuntejo_aibeP!p]

Pagal sistemos prototipo vykdymo veiksmų seką galima išskirti apačioje pavaizduotose veiksmų sekų diagramose tuos pačius veiksmus.



31 pav. *Veiksmų sekų diagrama*

3.2.5. Verifikavimo eksperimentai

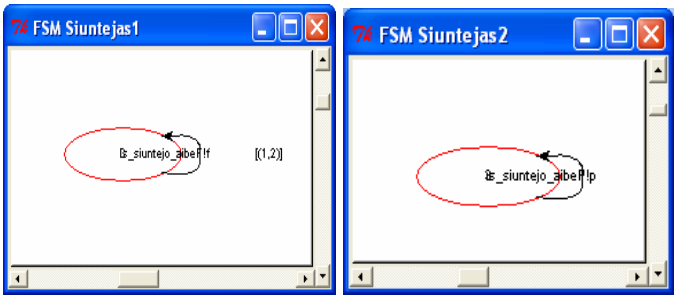
Tikrinsime teisingumo savybes: Saugumo būsenų savybes (Safety state properties); Neteisingos pabaigos būsenas(Invalid end states); Nepasiekiamas būsenas(Report Unreachable Code). Verifikavimas bus vykdomas nuodugniu paieškos metodu(Exhaustive).

Lentelė Nr. 8. Verifikavimas

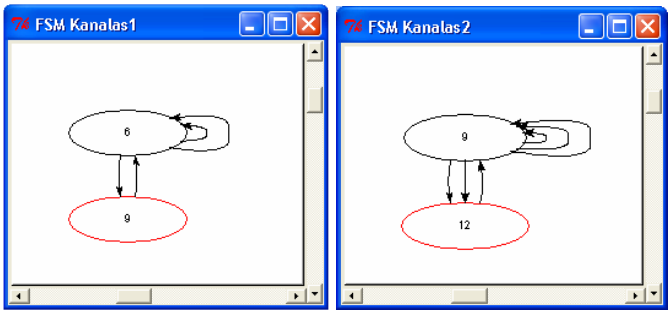
Verifikavimo rezultatai	Paaiškinimai
(Spin Version 3.4.14 -- 6 April 2002	Spin versija, kuri sugeneruoja failą pan.c. Iš šio failo vykdomas verifikavimas.
+ Partial Order Reduction	Tikrinimo būdas. Pliuso ženklas reiškia, kad šis algoritmas buvo naudojamas pagal nutylėjimą. Minuso ženklas reikštų kompiliavimą nuodugnia, nesumažinta paieška su parinktimi -DNOREDUCE .
Full statespace search for:	Parodo paieškos metodą. Pagal nutylėjimą atliekama pilna būsenų paieška. Labai dideli modeliai gali būti tikrinami Bitstate paieška, kuri yra apytikslė.
never-claim - (not selected)	Niekada nesikreipiama. Minuso ženklas reiškia kad nebuvo tikrinama. Jei „never claim“ buvo modelio dalis, gali būti kad buvo nuslopinta su parinktimi -DNOCLAIM .
assertion violations - (disabled by -A flag)	Tvirtinimų pažeidimai. Minusas reiškia kad paieškoje nebuvo tikrinama ar yra neprogresyvių ciklų. Norint tai atlikti reikia nustatyti ”run-time” parinktį **-a** arba kompiliuoti -DNP sujungtą su ”run-time” parinktimi **-l**.
cycle checks - (disabled by -DSAFETY)	Ciklų paieška-netikrinta
invalid endstates +	Neteisingos pabaigos būsenų nerasta
State-vector 64 byte, depth reached 9039, errors: 0	Visos sistemos pilnas testavimas pareikalavo 64 baitų atminties(kiekvienai būsenai). Ilgiausias paieškos gylio kelias turėjo 9039 perėjimus nuo pradinės sistemos būsenos. Klaidų nerasta.
17131 states, stored	Sukurtos 17131 globalios būsenos
51014 states, matched	51014 būsenų atitiko, t.y paieškoje grįžo i jau tikrintas būsenas paieškos medyje.
68145 transitions (= stored+matched)	Viso buvo ištirta 68145 perėjimų.
2 atomic steps	2 nedalomi (atomic) žingsniai.
hash conflicts: 486264 (resolved)	486264 atvejais „hash“ schema aptiko procesų susidūrimus ir buvo priversta perkelti būsenas į nuorodų sąrašą „hash“ lentelėje.
(max size 2^19 states)	Argumentas(pagal nutylėjimą) kuris nusako „hash“ lentelės dydį. Atitinka „run-time“ parinktį **-w18**.
3.566 memory usage (Mbyte)	Šiai paieškai panaudota 2.542 Mb atminties resursų įtraukiant steką, „hash“ lentelę ir visas duomenų struktūras. Su parinktimis **-m** ir **–w** galima sumažinti reikalaujamą atminties kiekį.

Baigtinių būsenų automatai

Spin sistemos automatų peržiūros parinktis leidžia pamatyti struktūrą automato, kurį Spin naudoja verifikavimo metu. Kiekvienas procesas atvaizduojamas atskiru automatu. Pasirinkus automatų peržiūrą, pirmiausiai Spin sugeneruoja verifikatorių, sukompiliuoja ir paleidžia vykdymą. Tokiu būdu gaunami procesų vardai ir atitinkamas automatas. Kaip matome, automatai tokie pat kaip ir susiprojektavome.



32 pav. *SPIN sugeneruoti Sintėjas1 ir Siuntėjas2 automatai*



33 pav. *SPIN sugeneruoti Kanalas1 ir Kanalas2 automatai*

IŠVADOS

Šiame darbe yra tiriamos protokolų, aprašytų PLA, verifikavimo ir imitavimo galimybės SPIN sistemoje:

- Transformavus PLA specifikacijas į baigtinius automatus galima atlikti imitavimą ir verifikavimą SPIN sistema.
- Taigi, tam kad būtų galima panaudoti SPIN, reikia PLA transformuoti į baigtinį automatą. Tai galima atlikti PLA tolydines koordinates aproksimuojant baigtinėmis.
- Parodyta, kad |PLA specifikaciją galima verifikuoti skirtinguose abstrakcijos lygiuose. Tam reikia konstruoti atitinkamus baigtinius automatus.
- Magistriniame darbe sukurta PLA verifikavimo metodika, naudojant SPIN sistemą, išbandyta imituojant ir verifikuojant alternuojančio bito protokolą ir duomenų perdavimo protokolą su adaptyviu komutavimo metodu.

LITERATŪRA

[1] Pranevičius, H. *Kompiuterinių tinklų protokolų formalusis specifikavimas ir analizė: agregatinis metodas*. Kaunas, 2003.

[2] Pranevičius, H.; Pilkauskas, V.; Chmieliauskas, A. *Agregate aproach for specification and anglysis of computer network protocols*. Kaunas, 1994.

[3] Holzman, G. J. *Design and validation of computer protocols*. New Jersey, 1991.

[4] Holzman, G. J. *SPIN Model Checker, The: Primer and Reference Manual.* New Jersey, 2003.

[5] Holzmann, G. J.; Smith, M.H. *An automated verification method for distributed systems software based on model extraction*. IEEE Trans. on Software Engineering. 2002. Nr. 4, p. 364–377.

[6] Holzmann, G. J. *Designing executable abstractions*. Formal Methods in Software Practice. FL. ACM Press. p. 103–108.

[7] Holzmann, G. J. Smith, M. H. *Software model checking: Extracting verification models from source code*. Formal Methods for Protocol Engineering and Distributed Systems. 1999. London, England. Kluwer Publ. p. 481–497.

[8] Holzmann, G. J. *The Model Checker SPIN.* IEEE Trans. on Software Engineering. 1997. Nr. 5, p. 279–295.

[9] Holzmann, G. *The theory and practice of a formal method.* NewCoRe. In Proceedings IFIP World Congress, 1994. p. 35-44.

[10] Holzmann, G.J.; Najm, E.; Serhrouchini, A. *SPIN model checking: an introduction.* Journal on Software Tools and Technology Transfer, 2000. Nr 2(4), p. 321-327.

[11] Mikk E.; Lakhnech Y.; Siegel M. *Hierarchical automata as model for statecharts.* London, 1997, p. 11-16.

[12] Katoen, J. P. *Principles of model checking*. Lecture notes 2002. 220p.

[13] Wang, K.; Pranevičius, H. *Aplications of AI to Production Engineering*. Kaunas, 1997.

[14] SPIN Home Page. Language Reference. New Jersey: Bell Labs. 2006 m. sausis. – [žiūrėta 2006-02-27]. Prieiga per internetą: http://SPINroot.com/SPIN/Man/Promela.html

[15] Barland, I. Promela and SPIN Reference. 2004, Rugpjūtis [žiūrėta 2006-01-12]. Prieiga per internetą: http://cnx.rice.edu/content/m12318/latest/

[16] Promela Grammar Definition. 1997, Rugpjūtis [žiūrėta 2006-01-12]. Prieiga per internetą: http://www.cis.ksu.edu/VirtualHelp/Info/SPIN/grammar.html

[17] Verification Tools Database. Brno, Czech Republic: Faculty of Informatics. 2004 m. spalis. – [žiūrėta 2006-02-01].
Prieiga per internetą: http://anna.fi.muni.cz/yahoda/

[18] Katoen, J.P. Formal methods and tools. University of Twente. Lecture notes. 2002, [žiūrėta 2006-02-01]. Prieiga per internetą: www.cs.auc.dk/~kgl/DAT4F02/

[19] Klarke, E.M.; Wing, J.M. Formal methods: state of the art and future directions. Carnegie Mellon university. [žiūrėta 2006-03-11].
Prieiga per internetą: www.cs.cmu.edu/afs/cs/usr/wing/www/mit/paper/paper.ps

[20] Holzmann, G. J.; Godefroid, P.; Pirottin, D. *Coverage precerving reduction strategies for reachability anglysis. Protocol specification, testing and verification*. North Holland 1992, p. 349-364

[21] Clarke, E.M.; Emerson, E.A. *Synthesis of synchronization skeletons for branching time temporal logic*. Lecture notes in computer science. Springer Verlag. 1990

[22] Kurshan, R.P. Computer aid verification of coordinating processes. Princeton university press, 1994

[23] McMillan, K.L. *Symbolic model checking*. Boston, 1993

[24] Peled, D. *Combining partial order reductions with on-the-fly model checking*. Lecture notes. 1994

[25] Holzmann, G.J. *An analysis of bitstate hashing.* Formal Methods in Systems Design, 1998

[26] Wolper, P.; Leroy, D. *Reliable hashing without collision detection.* Proc. 5th Int. Conference on Computer Aided Verification, 1993, Elounda, Greece, p. 59-70

[27] Gerth, R.; Peled, D.; Vardi, M.; Wolper, P. *Simple on-the-fly automatic verification of linear temporal logic.* Proc. PSTV Conference, Warsaw, Poland, 1995

[28] Puri, A.; Holzmann, G.J. *A Minimized automaton representation of reachable states.* Software Tools for Technology Transfer, Nr. 1, Springer Verlag

[29] Holzmann, G.J.; Peled, D.; Yannakakis, M. *On nested depth-first search. The Spin Verification System, American Mathematical Society, 1996, p. 23-32*

[30] Holzmann, G.J.; Peled, D. *An Improvement in Formal Verification.* Proc. FORTE Conference, Bern, Switzerland 1994

[31] Moshe Y. V.; Wolper, P. *An automata-theoretic approach to automatic program verification. Proc. First IEEE Symp. on Logic in Computer Science, 1986, p. 322-331*

[32] Rahardio, B. *SPIN as a hardware design tool*. First SPIN workshop. Kanada 1995

[33] Chan T.S.; Gorton, I. Formal validation of a high performance error control protocol using SPIN. Software, practice and experience. Nr. 26, 1996 p. 105-124

[34] Tripakis, S.; Courcoubetis, C. *Extending Promela and SPIN for real time*. Proc. tools and algorithms for the construction and anglysis of systems. Germany, 1996 p. 329-348

[35] Godefroid, P.; Holzmann, G.J. *On the verification of temporal properties*. Symp. Protocol specification, testing and verification. Belgium, 1993. p. 109-124

[36] Godefroid, P. Partial order methods for the verification of concurrent systems. Lecture notes. Springer Verlag, 1996

Verification of Aggregate specifications transforming them in to finite-state automata

Summary

The ultimate goal of SPIN and indeed of all testing or validation methodology is to demonstrate, with a certain degree of confidence, that a proposed design or implementation meets its requirements.

SPIN is a tool to simulate and validate Protocols. Promela, its source language, is a formal description technique like SDL and Estelle that is based on communicating state atomata. Unlike most other tools, SPIN is in the public domain and therefore is one of the most widely used formal verification tools today.

Paper presents a formalization method of piece linear aggregates (PLA) and an investigation of possibilities to use SPIN system for validation of PLA specifications of distributed systems. It is shown that in order to write Promela specifications, processes used in the PLA model should be represented by finite state automata. PLA specification of two protocols, its description by finite state automata and its verification results in SPIN system are presented.

It is shown in this paper that the SPIN system can be used for the verification of aggregate specifications. Using the SPIN system the finite state graphs of the processes used in the formal specification have to be formed. Then they have to be described in the Promela language.

PRIEDAI

Priedas 1

Duomenų perdavimo protokolo su adaptyviu komutavimo metodu programos tekstas Promela kalba. Pilna specifikacija.

```
#define N 1
#define M 6
#define MAX 3
mtype = { f, p};
chan is_siuntejo_aibeF =[N] of {mtype}
chan is_siuntejo_aibeP =[N] of {mtype}
int nf,nfpf,nfpp,np;
 int Q=0,L2=2,L1=1,L=3;
proctype Siuntejas1()
{
again:   do
         :: is_siuntejo_aibeF!f;
         goto again;
         od
}
proctype Siuntejas2()
{
again:   do
         :: is_siuntejo_aibeP!p;
         goto again;
         od;
}
proctype Kanalas1 ()
{
a: do
:: is_siuntejo_aibeF?f ;
b:       do
         :: if :: nfpf>1;nfpf--; fi;
         :: is_siuntejo_aibeF?f ;
         :: goto a;
         :: if :: nf<N;nf++; fi
         :: if :: nf=N&&nfpf<M&&Q<L1;nfpf++; fi;
         :: if :: nf>0&&nfpf==0;nf--; fi;
         od;
od
}
proctype Kanalas2 ()
```

```
{
a: do
:: is_siuntejo_aibeP?p;
b:      do
        :: is_siuntejo_aibeP?p;
        do ::Q++; goto b od
        :: Q<=1;goto a
        :: if ::Q>0;Q--; fi
        :: is_siuntejo_aibeP?p;
        :: if :: nfpf>0&&Q<L2; nfpf--; fi
        :: Q<=0;goto a
        od;
od
}
init {
atomic{
        run Siuntejas1();
        run Siuntejas2();
        run Kanalas1();
        run Kanalas2();
      }}
```

Priedas 2

Publikacijos:

Baužaitė, R.; Pranevičius, H. *SPIN sistemos panaudojimas agregatinių specifikacijų validavimui.* Informacinės technologijos 2005. Konferencinių pranešimų medžiaga. Kaunas 2005.

Baužaitė, R.; Pranevičius, H. *Simbolinių būsenų panaudojimas agregatinių specifikacijų validavimui.* Informacinės technologijos 2006. Konferencinių pranešimų medžiaga. Kaunas 2006. p. 521.

Baužaitė, R.; Pranevičienė, I.; Budnikas, G. *Verification of Agregate Specifications using SPIN System.* International conference on operational research: simulation and optimisation in business and industry. Tallin, Estonia 2006. p. 59.