KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

PROGRAMŲ INŽINERIJOS KATEDRA

Tomas Krivoūsas

Verifikavimo algoritmų panaudojimas analizuojant formalių PLA specifikacijų teisingumą

Magistro darbas

Darbo vadovas

prof. habil. dr. H. Pranevičius

Kaunas 2008

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

PROGRAMŲ INŽINERIJOS KATEDRA

Tomas Krivoūsas

Verifikavimo algoritmų panaudojimas analizuojant formalių PLA specifikacijų teisingumą

Magistro darbas

Recenzentas

doc. dr. Tomas Blažauskas
2008-05-26

Vadovas

prof. habil. dr. H.Pranevičius
2008-05-26

Atliko

IFM-2/2 gr. stud.
Tomas Krivoūsas
2008-05-26

Kaunas 2008

Turinys

1. Įvadas 7
2. Agregatinių specifikacijų teisingumo tikrinimo metodai ir algoritmai 9
2.1. Agregatinis paskirstytųjų sistemų formalizavimo metodas 9
2.2. Agregatinių specifikacijų saugumo ir gyvybingumo tyrimo metodai 10
2.2.1. Pasiekiamų būsenų metodas. 10
2.2.1.1. Pilna paieška 11
2.2.1.2. Kontroliuojama dalinė paieška.. 11
2.2.2. Invarianto metodas 12
2.3. Stipriai susijusių grafo komponenčių radimo algoritmai. 13
2.3.1. Kosarajaus algoritmas. 14
2.3.2. Tarjano algoritmas 16
2.3.3. Gabovo algoritmas. 19
2.3.4. DCSC algoritmas 19
2.4. Saugumo ir gyvybingumo savybių tyrimo algoritmai 20
2.4.1. Aklavietčių radimas.. 20
2.4.2. Uždarų ciklų radimas... 21
2.4.3. Būsenų pasiekiamumo tikrinimas 22
2.4.4. Būsenos koordinačių apribojimų tikrinimas.. 22
2.4.5. Invarianto tikrinimas 22
2.5. Problemos sprendimas pasaulyje 22
2.5.1. SPIN 23
2.5.2. VIS 24
2.5.3. SMV 24
2.5.4. NuSMV.. 24
2.5.5. Uppaal.. 24
2.5.6. Pranas-2 25
2.5.7. VALSYS... 25
2.5.8. SLAM 26
2.5.9. TLC 26
2.5.10. ADPRO.. 26
3. Projektinė integruotos formalių specifikacijų analizės sistemos dalis 27
3.1. Programų sistemos funkcijos.. 27
3.2. Sistemos sudėtis.. 28
3.3. Validavimo posistemės panaudojimo atvejai 28

3.4. Validavimo posistemės klasių diagrama 29
3.5. Objektinio modelio klasių diagrama 30
3.6. Validavimo posistemės darbo langas 32
3.7. Sistemos duomenų vaizdas 32
4. Pasiekiamų būsenų grafo sudarymo metodai 34
4.1. Pasiekiamų būsenų grafo sudarymo algoritmas 34
4.2. Lygiagretus pasiekiamų būsenų grafo sudarymo algoritmas 35
4.2.1. Lygiagretus programavimas 35
4.2.2. Pasiekiamų būsenų grafo sudarymo algoritmas 36
4.2.3. Būsenų grafo sudarymo algoritmo gijų skaičiaus parinkimas 39
5. Eksperimentai su FSA sistema 41
5.1. Validavimo posistemės veikimo eksperimentai 41
5.2. Validavimo posistemės greitaveikos eksperimentai. 46
5.2.1. Grafo generavimo trukmės priklausomybė nuo būsenų skaičiaus. 47
5.2.2. Grafo generavimo pagreitėjimas naudojant lygiagretų programavimą 48
5.2.3. Grafo validavimo laiko priklausomybė nuo būsenų skaičiaus 48
5.2.4. Grafo analizės laiko priklausomybė nuo būsenų skaičiaus 49
5.2.5. Santykinis grafo generavimo ir validavimo laiko įvertinimas 50
5.2.6. Generavimo laiko priklausomybė nuo naudojamų gijų skaičiaus. 51
5.3. Greitaveikos eksperimentų rezultatai 52
6. Išvados 53
7. Santrumpų ir terminų žodynas 54
8. Literatūros sąrašas 55
9. Priedai 57
9.1. PLA formalizavimo kalbos aprašymas 57

Lentelių sąrašas

Lentelė 2.1 Paskirstytųjų sistemų analizės metodai ir analizuojamos charakteristikos 10
Lentelė 2.2. Grafo viršūnių požymių reikšmės Tarjano algoritmo vykdymo pabaigoje..........18
Lentelė 5.1 Grafo generavimo priklausomybė nuo būsenų skaičiaus duomenų lentelė 47
Lentelė 5.2 Grafo validavimo laiko priklausomybės nuo būsenų skaičiaus duomenų lentelė .48
Lentelė 5.3 Grafo analizės priklausomybės nuo būsenų skaičiaus duomenų lentelė...............49
Lentelė 5.4 Santykinio grafo generavimo ir validavimo laiko įvertinimo duomenų lentelė....50
Lentelė 5.5 Generavimo laiko priklausomybės nuo gijų skaičiaus duomenų lentelė 51

Paveikslų sąrašas

Pav. 2.1 Orientuotas grafas, kuriame ieškoma stipriai susijusių komponenčių.......................14
Pav. 2.2 Grafas su surastais viršūnių apdorojimo pradžios ir pabaigos laikais 15
Pav. 2.3 Transponuotas grafas su viršūnių apdorojimo pabaigos laikais 15
Pav. 2.4 Transponuotas grafas su surastomis stipriai susijusiomis komponentėmis 15
Pav. 2.5 Jungiųjų komponenčių grafas...16
Pav. 2.6 Tarjano algoritmo vykdymas po 4 žingsnių...17
Pav. 2.7 Šakninė stipriai susijusios komponentės viršūnė..17
Pav. 2.8 Algoritmo žingsnis po viršūnių pašalinimo iš steko..18
Pav. 2.9 Pasiekiamų būsenų grafo aklavietė..21
Pav. 2.10 Uždari ciklai pasiekiamų būsenų grafe 21
Pav. 3.1 Sistemos sudėtis..28
Pav. 3.2 Validavimo posistemės panaudojimo atvejai 28
Pav. 3.3 Validavimo posistemės klasių diagrama..30
Pav. 3.4 Objektinio modelio klasių diagrama..31
Pav. 3.5 Pagrindinis validavimo posistemės langas 32
Pav. 3.6 XML faile aprašomos duomenų struktūros ir ryšiai tarp jų. 33
Pav. 4.1 Bendrai naudojamo resurso problema. 37
Pav. 4.2 Bendrai naudojamo resurso problemos sprendimo būdas..38
Pav. 5.1 Vienkanalė aptarnavimo sistema su šakotuvu 41
Pav. 5.2 Pasiekiamų būsenų grafas 44
Pav. 5.3 Pasiekiamų būsenų grafas po specifikacijos pakeitimo 46
Pav. 5.4 Specifikacijos klaida, aptikta sistemos validavimo metu..46
Pav. 5.5 Grafo generavimo trukmės priklausomybės nuo būsenų skaičiaus grafikas.............47
Pav. 5.6 Grafo generavimo pagreitėjimo grafikas 48
Pav. 5.7 Grafo validavimo laiko priklausomybės nuo būsenų skaičiaus grafikas 49
Pav. 5.8 Bendro analizės algoritmo vykdymo laiko grafikas 49
Pav. 5.9 Nuoseklaus grafo generavimo ir validavimo laiko santykio grafikas........................50
Pav. 5.10 Lygiagretaus grafo generavimo ir validavimo laiko santykio grafikas 51
Pav. 5.11 Generavimo laiko priklausomybės nuo gijų skaičiaus grafikas 52
Pav. 5.12 Generavimo laiko priklausomybės nuo gijų skaičiaus logaritminis grafikas..........52

Summary

Usage of verification algorithms for analyzing the correctness of formal PLA specifications.

Arguably the most important task in creation of software is user requirement specification. Accurate requirement specification allows avoidance of errors in late stages of software development. This is extremely important in critical systems, where even vague error can cause great financial losses or even human victims. One of the methods used for precise user requirement specification is use of formal specifications.

Formal specification is a mathematical method for describing of software or hardware, which might be suitable for system realization. Nevertheless, the construction of formal specifications does not guarantee the correctness of specification. For this reason formal specification validation is necessary.

In this paper methods of formal specification validation are discussed. Two most popular methods of formal specification validation are reachable state graph analysis and invariant checking. Reachable state graph analysis consists of graph generation and graph analysis. Graphs can be analyzed for dead-ends, closed loops, state reach ability checking, coordinate restriction checking or invariant checking. Traditional reachable graph generation algorithm uses unanalyzed states queue to produce reachable state graph. Each step single state is analyzed and depending on results new vertex or edge is added to state graph.

An improvement to the algorithm to consider is usage of parallel programming to process multiple states simultaneously. This allows increasing the speed of algorithm execution, since multiple states will be processed in time just one state was processed.

Experiments with single channel processing systems showed, that usage of parallel reachable state generation algorithm for solution of this problem increased analysis performance by up to 35%, depending on number of states.

1. Įvadas

Vienas svarbiausių uždavinių kuriant sudėtingas programinės įrangos sistemas – tai teisingas vartotojo reikalavimų specifikavimas. Kuo tiksliau specifikuojami vartotojų reikalavimai, tuo mažesnė klaidų tikimybė realizacijos metu. Be to, pataisymų kaina specifikuojant sistemas yra žymiai mažesnė nei atliekant sistemos realizaciją. Tai ypač svarbu kritinėse sistemose, kuriose net menkiausias netikslumas sistemos realizacijoje gali atnešti milžiniškų nuostolių arba net žmonių aukų. Vienas iš metodų tiksliai specifikuoti vartotojo reikalavimus yra formaliosios specifikacijos. Formalios specifikacijos – tai matematinis programinės ar techninės įrangos aprašymas, kurį galima naudoti sistemos realizacijai.

Formalių specifikacijų sudarymas nėra paprastas procesas, be to, labai sunku sudarinėjant specifikaciją patikrinti jos teisingumą. Formalūs metodai negali pakeisti tradicinių PĮ kūrimo metodų, o greičiau bus naudojami integruoti su kitais PĮ kūrimo metodais, siekiant gauti papildomos informacijos apie sistemos veikimą, galimybę anksti pastebėti klaidas ir lengvai jas ištaisyti, galimybę specifikuoti ir dokumentuoti reikalavimus be dviprasmybių, esant reikalui sistemos veikimą pagrįsti matematiškai.

Formali sistemos specifikacija dar negarantuoja jos teisingumo, todėl prieš pradedant sistemos realizaciją būtina verifikuoti specifikaciją. Tam naudojami formalaus verifikavimo metodai. Dažniausiai nagrinėjamos dvi sistemos teisingumo charakteristikų grupės – saugumo ir gyvybingumo charakteristikos. Saugumo charakteristikos parodo, kad sistemoje nevyksta nepageidaujamų įvykių. Gyvybingumo charakteristikos parodo, kad sistemoje įvyksta tam tikri pageidaujami įvykiai.

<u>Darbo tikslas</u>

Išanalizuoti teisingumo tikrinimo metodus ir algoritmus, naudojamus formalių specifikacijų analizei, bei surastus algoritmus pritaikyti ir panaudoti sukurtoje formalių specifikacijų analizės sistemos validavimo posistemėje. Ši posistemė analizuoja atkarpomis tiesinių agregatų (PLA) teisingumą, verifikuodama specifikacijų saugumo ir gyvybingumo charakteristikas.

<u>Uždaviniai</u>

- Išanalizuoti formalių specifikacijų validavimo algoritmus.
- Parinkti ir realizuoti teisingumo tikrinimo algoritmus formalių specifikacijų analizės (FSA) sistemos validavimo posistemėje.
- Atlikti eksperimentus su sukurtu įrankiu.
- Įvertinti realizuotų algoritmų greitaveiką.

Dokumento struktūra

Šiame darbe aprašomi agregatinių specifikacijų saugumo ir gyvybingumo tyrimo metodai, bei sukurta formalių specifikacijų analizės sistema. Antrame dokumento skyriuje analizuojami agregatinių specifikacijų teisingumo tikrinimo metodai. Trečias skyrius skirtas pristatyti sukurtą formalių specifikacijų analizės sistemą FSA. Ketvirtame skyriuje aprašyti algoritmai, skirti būsenų grafo generavimui. Penktame skyriuje pateikiami eksperimentų, atliktų su formalių specifikacijų analizės sistema, rezultatai. Šeštame skyriuje pateikiami darbo rezultatai ir išvados.

2. Agregatinių specifikacijų teisingumo tikrinimo metodai ir algoritmai

2.1. Agregatinis paskirstytųjų sistemų formalizavimo metodas

Agregatinio paskirstytųjų sistemų formalizavimo ir analizės metodo teorinis pagrindas – atkarpomis tiesinių agregatų formalizmas. Ši matematinė schema leidžia bendrosios formaliosios specifikacijos pagrindu validuoti sudarytą specifikaciją (atlikti teisingumo analizę) bei sudaryti analizuojamos sistemos imitacinį modelį. Atkarpomis tiesiniai agregatai priklauso automatų modelių klasei. Skiriamasis šių modelių bruožas yra sistemos būsenai aprašyti naudojamos diskrečiosios ir tolydžiosios koordinatės. Atkarpomis tiesiniai agregatai aprašomi valdymo sekų metodu [1].

Agregatinių specifikacijų teisingumo tikrinimo metodai leidžia analizuoti specifikuotos sistemos saugumo ir gyvybingumo savybes. Pasiekiamų būsenų metodo esmę sudaro tai, kad, turint analizuojamos sistemos agregatinę specifikaciją, generuojama visų galimų sistemos būsenų trajektorijų aibė. Paskui šios trajektorijos analizuojamos sistemos tiriamų savybių atžvilgiu. Taikant invariantų metodą, reikia sudaryti sistemos invariantą ir patikrinti, ar jis yra teisingas visose galimose sistemos būsenose.

Paskirstytų sistemų elgsenos analizės metu tiriamos visos galimos sistemos trajektorijos, o tai leidžia patikrinti, ar sudaryta specifikacija teisinga. Paskirstytų sistemų teisingumas tikrinamas įvairiais validavimo bei verifikavimo (teisingumo tikrinimo ir patvirtinimo) metodais.

2.1 lentelėje pateiktos pagrindinės elgsenos bei funkcionavimo analizės charakteristikos.

Elgsenos analizės požiūriu analizuojamos sistemos charakteristikos suskirstomos į dvi grupes: *saugumo* ir *gyvybingumo.* Saugumo charakteristikos rodo, kad sistemoje neįvyksta iš anksto apibrėžtų nepageidaujamą įvykių. Tokių įvykių pavyzdžiai gali būti: statinės ir dinaminės aklavietės, kintamųjų neapibrėžtumas, invariantinės savybės ir pan. Gyvybingumo charakteristikos rodo, kad sistemoje įvyksta tam tikrų pageidaujamų įvykių. Tokių įvykių pavyzdys: jei predikatas P yra teisingas tam tikroje būsenoje, tai sistema po tam tikro laiko pasieks kitą būseną, kurioje yra teisingas predikatas Q. Tai žymima P~> Q. *Pabaigiamumas* reiškia, kad sistema pasieks galinę būseną.

Lentelė 2.1 Paskirstytųjų sistemų analizės metodai ir analizuojamos charakteristikos

Analizės rūšis	
Elgsenos	*Funkcionavimo*
Metodai	
Validavimas ir verifikavimas	*Imitacinis modeliavimas*
Charakteristikos	
Saugumas • statinės ir dinaminės aklavietės • kintamųjų apibrėžtumas • invariantinės savybės *Gyvybingumas* • $P \sim > Q$ • pabaigiamumas	Eilių ilgiai • Pranešimų perdavimo laikai • Laukimo laikai • Įrenginių panaudojimo koeficientai

2.2. Agregatinių specifikacijų saugumo ir gyvybingumo tyrimo metodai

Agregatinių specifikacijų saugumo bei gyvybingumo tikrinimui naudojami pasiekiamų būsenų ir invariantų metodai [2]. Pasiekiamų būsenų metodo esmę sudaro tai, kad turint sistemos agregatinę specifikaciją generuojama visų galimų sistemos trajektorijų aibė. Tada šios trajektorijos nagrinėjamos sistemos tiriamų savybių atžvilgiu. Taikant invariantų metodą, reikia sudaryti sistemos invariantą ir patikrinti, ar jis yra teisingas visose galimose sistemos būsenose.

Toliau nagrinėjamos pasiekiamų būsenų analizės rūšys bei invariantinis tikrinimas.

2.2.1. Pasiekiamų būsenų metodas.

Sistemos būsena galima vadinti sistemos kintamųjų reikšmes bet kuriuo laiko momentu. Bet kuri sistema gali turėti tūkstančius ar net šimtus tūkstančių būsenų, iš kurių ne visos gali būti pasiekiamos iš pradinės būsenos. Aišku, kad bet kuriuo laiko momentu sistema yra tik vienoje būsenoje, be to gali pereiti tik į tam tikras kitas būsenas. Formalaus sistemos modelio būsenų išskyrimo algoritmo užduotis yra sugeneruoti ir patikrinti visas sistemos būsenas, kurios gali būti pasiekiamos iš pradinės sistemos būsenos [3].

Priklausomai nuo nagrinėjamos sistemos sudėtingumo naudojami trys pagrindiniai pasiekiamų būsenų analizės algoritmai [4]:

- Pilnos paieškos.
- Kontroliuojamos dalinės paieškos [5].
- Atsitiktinio modeliavimo.

Pilnos paieškos algoritmas taikomas sistemoms, kurios turi palyginti mažą būsenų skaičių(iki 10^5 eilės būsenų). Kontroliuojamą dalinę paiešką tikslinga atlikti sistemoms, kurios turi iki 10^8 būsenų. Jei būsenų skaičius viršija 10^8 naudojamas atsitiktinio modeliavimo algoritmas.

2.2.1.1. Pilna paieška

Išsami analizė reiškia, kad bus nagrinėjamos visos galimos situacijos, kuriose gali atsidurti sistema vykdymo metu. Šis algoritmas yra paprasčiausias iš anksčiau išvardintų algoritmų, tačiau tuo pačiu gali būti pritaikytas tik žemiausio lygio sistemoms [24]. Išsami pasiekiamumo analizė nustato, kurios sistemos būsenos yra pasiekiamos, o kurios – ne.

Kiekviena pasiekiama būsena ir pasiekiamų būsenų seka gali būti patikrinta pagal kokį nors korektiškumo kriterijų. Daugeliu atvejų reikalavimai sistemai gali būti aprašyti sistemos invariantais, kurie galėtų būti patikrinti loginiu testu kiekvienoje pasiekiamoje sistemos būsenoje.

Šis algoritmas tinkamas tik žemiausio lygio sistemoms, ir yra apribotas sistemos sudėtingumo laipsniu

2.2.1.2. Kontroliuojama dalinė paieška.

Jeigu analizuojamų būsenų aibė yra didesnė nei prieinami kompiuterio resursai gali apdoroti, tai pilna paieška sumažėja iki dalinės, be garantijos, kad svarbiausios sistemos dalys bus išanalizuotos [3]. Dėl šios priežasties buvo sukurta nauja algoritmų klasė, kuri naudoja dalinės paieškos pranašumus. Tokie algoritmai remiasi prielaida, jog daugelyje atvejų projektuotojus dominanti pasiekiamų būsenų aibė *A* yra tiktai dalis visų pasiekiamų būsenų *R*. Taigi, dalinės paieškos tikslas yra:

- Išanalizuoti būsenų aibę *A,* kurią sudaro *M/S* būsenų (čia S – atminties talpa, reikalinga vienai būsenai saugoti, *M* – atminties talpa, skirta vienam baigtiniam automatui. Tokiu būdu mes galime sugeneruoti ir išanalizuoti ne daugiau kaip *M/S* automato būsenų).
- Parinkti būsenas šiai aibei *A* iš pilnos pasiekiamų būsenų aibės *R,* kad būtų išanalizuotos visos pagrindinės sistemos funkcijos.
- Parinkti tokias būsenas aibei *A,* kad *kokybės* paieška (t. y. tikimybė rasti klaidą) būtų geresnę už *stebėjimo sritį A/R.*

Dalinės paieškos algoritmo mechanizmas yra toks pat kaip ir pilnos paieškos, tik skirtumas tas, kad yra analizuojamos ne visos būsenos einančios po duotos pasiekiamos būsenos.

Būdai dalinei paieškai organizuoti [3][4]:

- *Gylio apribojimas.* Analizuojamoms vykdomosioms sekoms uždedama ilgio riba, tai apriboja paieška iki rezultatyvaus sistemos elgsenų poaibio (pvz., eliminuoja daugkartinių persidengimų atvejus).
- *Išsidėstymo (išsibarstymo) analizė.* Išrenkamos tokios vykdomosios sekos,

kurios potencialiai veda į aklavietes. Vienas iš aklavietės atpažinimo požymių yra signalų nebuvimas perdavimo kanaluose.

- *Valdomoji paieška.* Projektuotojas pats pasirenka kitą sistemos būseną.
- *Tikimybinė paieška.* Pasiekiamos būsenos analizuojamos jų pasirodymo tikimybės mažėjimo tvarka. Pranešimai turi "aukštos" ar "žemos" tikimybės žymeklius, kurie pasitarnauja kaip atrankos kriterijus.
- *Atsitiktiniai rinkiniai.* Nededama pastangų į būsenos tipo prognozavimą (klaida, aklavietė). Si priemonė vienintelė atitinka visus tris kontroliuojamos dalinės paieškos tikslus.

2.2.2. Invarianto metodas

Sistemos invariantas I yra loginis teiginys, kuris aprašo teisingą sistemos funkcionavimą ir kuris turi išlikti teisingas nepaisant įvykių sekos ir sistemos perėjimo iš vienos būsenos į kitą [1][6].

Norint įrodyti, kad I yra sistemos invariantas, reikia:

- Įrodyti, kad *I* yra teisingas pradinei būsenai.
- Įrodyti, kad teisingi teiginiai:

$$\forall e_i : (EP_i \wedge I) H_i(e_i)(I), i = 1,2,...,n;$$

čia EP_i – įvykio e_i galimumo predikatas, $H_i(e_i)$ — agregatinės specifikacijos fragmentas, aprašantis koordinačių kaitą įvykus įvykiui e_i, ir n – įvykių skaičius.

Remiantis konceptualiuoju modeliu galima aprašyti sistemos funkcionavimą įvykių seka, pateikiama grafu G(V), jeigu V – viršūnių aibė, $V = \{e_1, e_2, ..., e_n\}$, čia ei yra i –tasis įvykis, n — įvykių skaičius; $A = \{a_{ij}\}$ – matrica, apibrėžianti ryšius tarp įvykių, kai:

$a_{ij} = 1$, jei e_j gali įvykti įvykus įvykiui e_i,

$a_{ij} = 0$, priešingu atveju

$(e_i e_j) \neq (e_j e_i)$, t.y. grafas yra orientuotas.

Agregato būsenų poaibis, į kurį patenka sistema įvykus įvykiui e_t, vadinamas simboline būsena ir žymimas SS_t.

$$SS_i = \{z \in Z | (\exists z')((z' \in Z) \wedge EP_i(z') \wedge (z = H_i(z', P)))\};$$

čia Z – sistemos būsenų aibė, $EP_i(z')$ – įvykio e_i būsenoje z' galimumo predikatas, P – sistemos tikimybiniai parametrai ir H_t – perėjimo operatorius, apibrėžiantis naują sistemos būseną įvykus įvykiui e_i.

Sakoma, kad sistema yra simbolinėje būsenoje SS_i tada ir tik tada, jeigu ji yra būsenoje z ir $z \in SS_i$. Atsižvelgiant į simbolinės būsenos SSi apibrėžimą, kiekvienas įvykis

ei yra susijęs su simboline būsena SS_i, todėl viršūnių V aibė grafe G(V) gali būti pakeista aibe V'= $\{SS_1,SS_2,...,SS_n\}$. Matrica A, apibrėžianti ryšius tarp gretimų simbolinių būsenų, išlieka nepakitusi. Gaunamas simbolinių būsenų grafas G(V'),kuris apibūdina sistemą apibrėždamas galimų simbolinių būsenų aibę ir perėjimus iš vienos simbolinės būsenos į kitą. Norint atskleisti ypatingas konkretaus modelio savybes, simbolinė būsena SS_i gali būti suskaidyta į poaibius SSi_1, SSi_2 ,...,SSi_m su sąlyga, kad $SS_i = \bigcup_{j=1}^{m} SS_{i_j}$ (tokiu būdu simbolinių būsenų padaugėja). Be to, jeigu skirtumas tarp galimų skirtingų simbolinių būsenų yra neesminis, tyrinėtojui analizuojant modelį, būsenos SSi1 ,SSi2 ,...,SSiq gali būti sujungtos į naują būseną– $SS_N = \bigcup_{j=1}^{m} SS_{i_j}$

Predikatas P_i, apibrėžiantis simbolinę būseną SS_i, nustato sistemos būsenų koordinačių apribojimus būsenoje SS_i. Invarianto I struktūra yra tokia

$$I = \bigvee_{i=1}^{n} P_i$$

čia n — simbolinių būsenų skaičius.

Predikato P_i struktūra yra tokia

$$P_i = \bigwedge_{j=1}^{p} K_{ij}$$

čia K_{ij} – j–asis predikatas, aprašantis simbolinės būsenos SS_i apribojimus, p – K_{ij} predikatų skaičius. Jeigu vartotojo nedomina konkreti koordinatė, ji gali būti neapribota nei vienu iš K_{ij} predikatų. Kiekvienas K_{ij} formuluojamas remiantis konceptualiuoju modeliu.

Norint įrodyti predikato teisingumą, būtina įrodyti, kad I yra teisingas kiekvienoje simbolinėje būsenoje, be to, kad simbolinių būsenų seka, gaunama iš konceptualiojo modelio, sutampa su įvykių seka, gaunama iš specifikacijos. Pažymėkime, kad PIS_i – tai aibė predikatų P– indeksų, kurie gali tapti teisingais pereinant iš simbolinės būsenos SS_i. Taip pat ir PIS_i yra aibė predikatų P_j indeksų, kurie gaunami iš specifikacijos. Kad simbolinių būsenų sekos, gautos iš konceptualiojo modelio atitiktų formaliąją specifikaciją, įrodant I teisingumą įvykiui e_i, sudaroma aibė PIS'_i ir tikrinama, ar $PIS_i = PIS'_i$.

2.3. Stipriai susijusių grafo komponenčių radimo algoritmai.

Stipriai susijusi grafo komponentė – tai toks grafo viršūnių poaibis, kuriame tarp bet kurių dviejų komponentės viršūnių u ir v yra kelias. Stipriai susijusi komponentė pasižymi tokiomis savybėmis:

- Dvi orientuoto grafo viršūnės priklauso tai pačiai jungiajai grafo komponentei tada ir tik tada, kai tarp šių viršūnių yra kelias.

- Bet kurį grafą galima išskaidyti į aibę jungiųjų komponenčių su tarp jų esančiais keliais.
- Tarp dviejų jungiųjų komponenčių negali būti abipusio kelio.
- Algoritmai gali bet kurį grafą išskaidyti į jungiąsias komponentes.

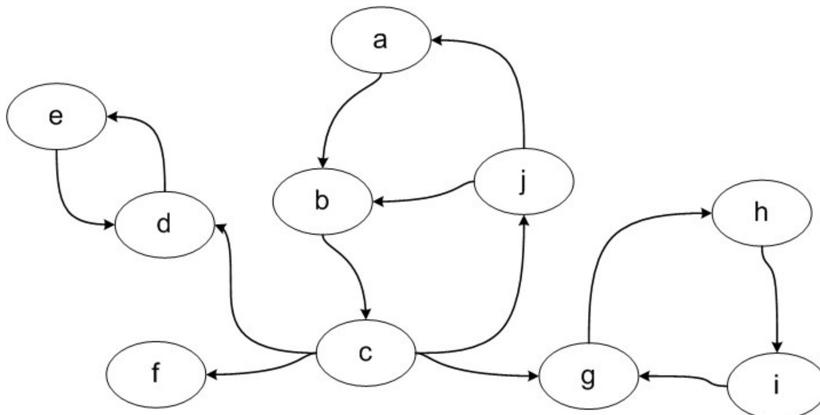
Labiausiai paplitę ir dažniausiai naudojami šie grafo susijusių komponenčių radimo algoritmai:

- Kosarajaus
- Tarjano
- Gabovo
- DCSC

2.3.1. Kosarajaus algoritmas.

Tai paprasčiausias ir lengviausiai realizuojamas stipriai susijusių komponenčių radimo algoritmas [7].

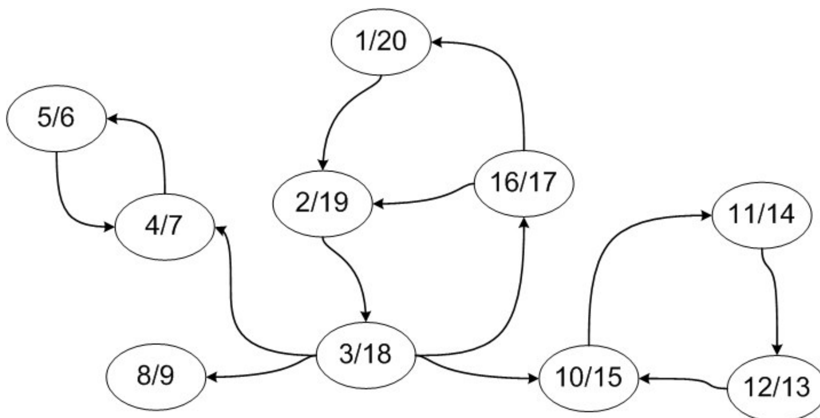
Tarkime turime grafą, pavaizduotą 2.1 paveiksle. Tikslas – surasti šio grafo stipriai susijusias komponentes.



Pav. 2.1 Orientuotas grafas, kuriame ieškoma stipriai susijusių komponenčių

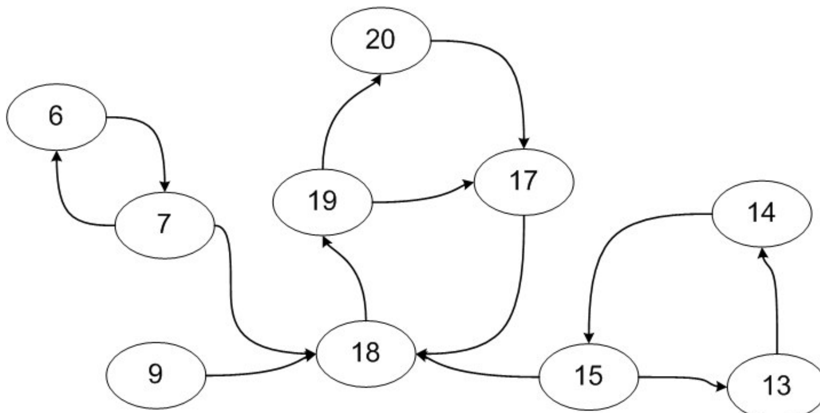
Kosarajaus algoritmas turi tokius etapus:

1. Naudojant grafo paieškos gilyn algoritmą (*angl.* DFS) surandame viršūnių apdorojimo pradžios ir pabaigos laikus. Viršūnės apdorojimo pradžios laikas – tai laiko momentas, kada paieškos gilyn algoritmas pirmą kartą aplankė viršūnę. Viršūnės apdorojimo pabaigos laikas – tai laiko momentas, kada paieškos gilyn algoritmas baigė apdoroti viršūnę. Grafas su surastais apdorojimo laikais pateiktas 2.2 paveiksle. Čia pirmas skaičius grafo viršūnėje reiškia apdorojimo pradžios laiką, o antras – pabaigos.



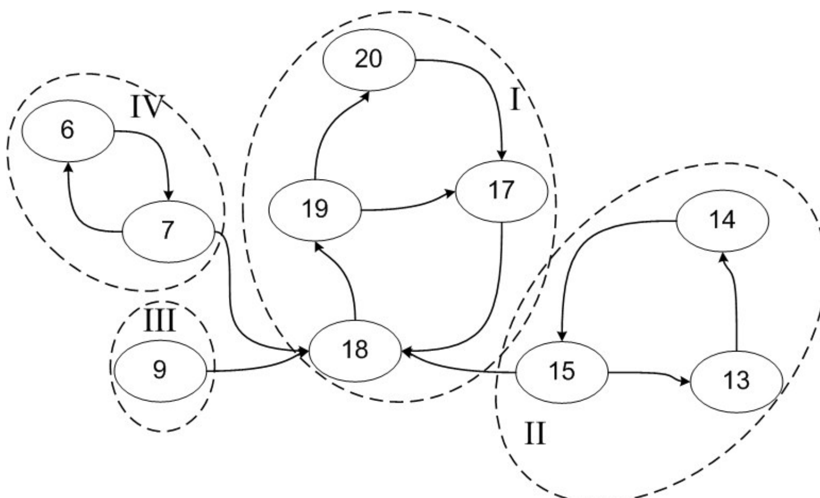
Pav. 2.2 Grafas su surastais viršūnių apdorojimo pradžios ir pabaigos laikais

2. Sudarome transponuotą grafą, t.y. grafą, kuriame pakeistos visos lankų kryptys. Transponuotas nagrinėjamo pavyzdžio grafas pavaizduotas paveiksle 2.3. Transponuoto grafo viršūnėse surašyti pradinio grafo apdorojimo pabaigos laikai.



Pav. 2.3 Transponuotas grafas su viršūnių apdorojimo pabaigos laikais

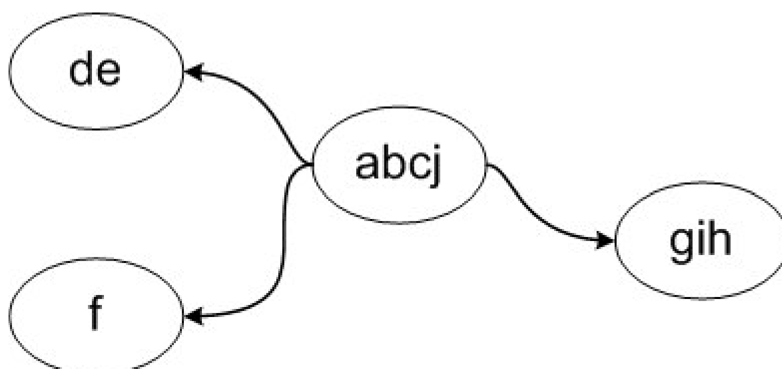
3. Naudojant paieškos gilyn algoritmą apeinamos visos grafo viršūnės jų apdorojimo pabaigos laiko mažėjimo tvarka. Algoritmo metu gauti viršūnių medžiai yra surastos stipriai susijusios komponentės. Pavyzdiniame grafe surastos jungiosios komponentės parodytos paveiksle 2.4.



Pav. 2.4 Transponuotas grafas su surastomis stipriai susijusiomis komponentėmis

Šiame pavyzdyje yra 4 stipriai susijusios komponentės – {abcj}, {de}, {f} ir {gih}.

4. Iš surastų jungiųjų komponenčių sudaromas jungiųjų komponenčių grafas, kuris pavaizduotas paveiksle 2.5.



Pav. 2.5 Jungiųjų komponenčių grafas

Šio algoritmo vykdymo laikas priklauso nuo viršūnių skaičiaus *V* ir briaunų skaičiaus *E*. Algoritmo vykdymo metu užtenka du kartus naudoti paieškos gilyn algoritmą, kurio vykdymo laikas tiesiškai priklauso nuo *V* ir *E*. Kosarajaus algoritmo sudėtingumas yra $O(V+E)$.

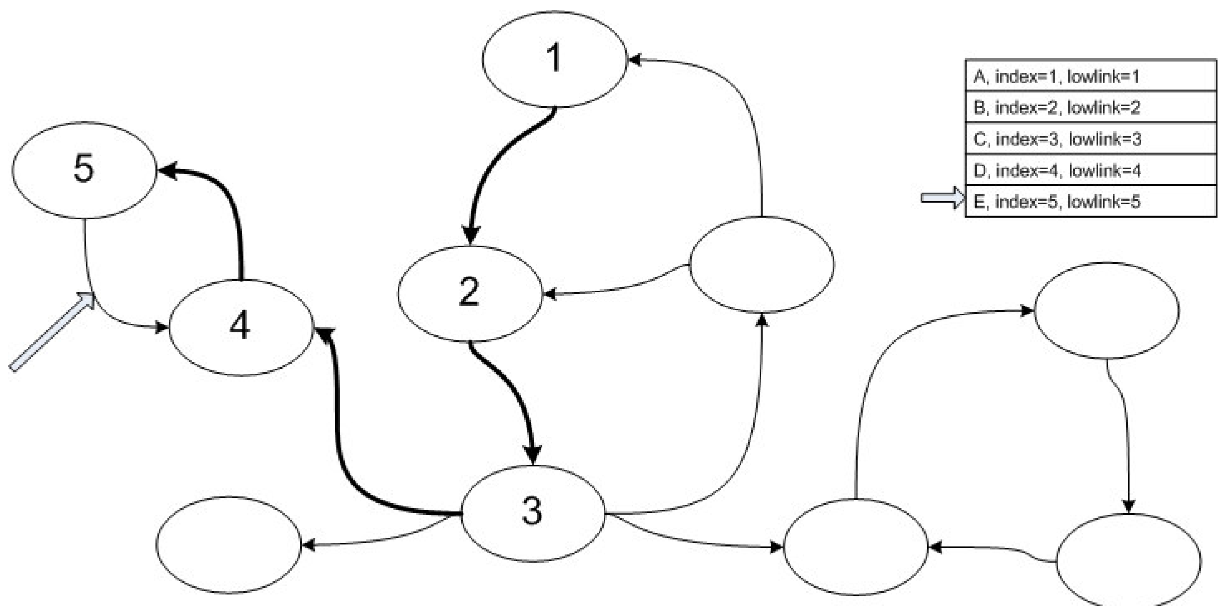
2.3.2. Tarjano algoritmas

Tarjano algoritmas – tai patobulintas Kosarajaus algoritmas, kuris paieškos gilyn algoritmą naudoja vieną kartą.

Tarjano algoritmas [8], kiekvienai grafo viršūnei aprašyti naudoja du papildomus požymius – *index*, nurodantį viršūnės apdorojimo eilės numerį arba indeksą, bei *lowlink*, reiškiantį mažiausią viršūnės indeksą, kurią galima pasiekti iš nagrinėjamos viršūnės. Stipriai susijusios komponentės šaknine viršūne (*angl.* root) čia bus vadinama pirma paieškos gilyn metu surasta stipriai susijusios komponentės viršūnė. Papildomai algoritme naudojamas stekas, į kurį talpinamos visos apdorotos viršūnės.

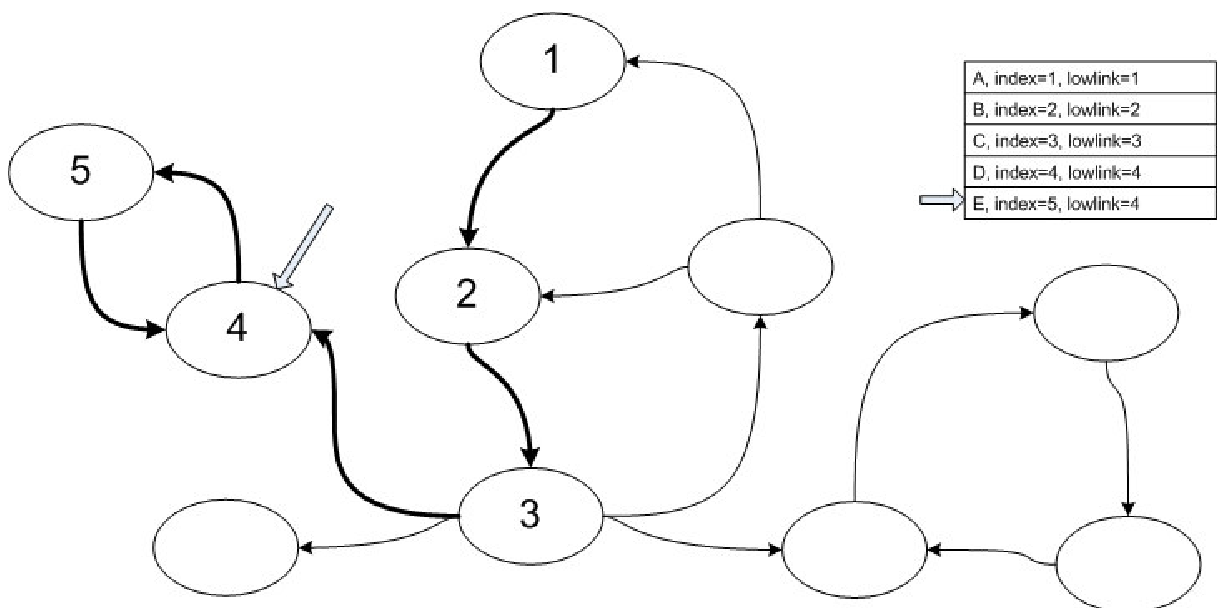
Algoritmą sudaro tokie etapai:

1. Rekursiškai kviečiamas paieškos gilyn algoritmas. Kol apdorojamos naujos viršūnės kiekvienos viršūnės požymiai *index* ir *lowlink* įrašomi tokie, koks yra apdorojamos viršūnės eilės numeris. Pradėta nagrinėti viršūnė rašoma į steką. Paveiksle 2.6 parodytas nagrinėjamas grafas. Viršūnėje įrašyta *index* požymio reikšmė. Taip pat parodytas stekas su jame esančiomis nagrinėtomis viršūnėmis. Tamsiau pažymėtos išnagrinėtos grafo briaunos, o rodykle pažymėta grafo briauna, kuri bus nagrinėjama sekančiame algoritmo žingsnyje.



Pav. 2.6 Tarjano algoritmo vykdymas po 4 žingsnių.

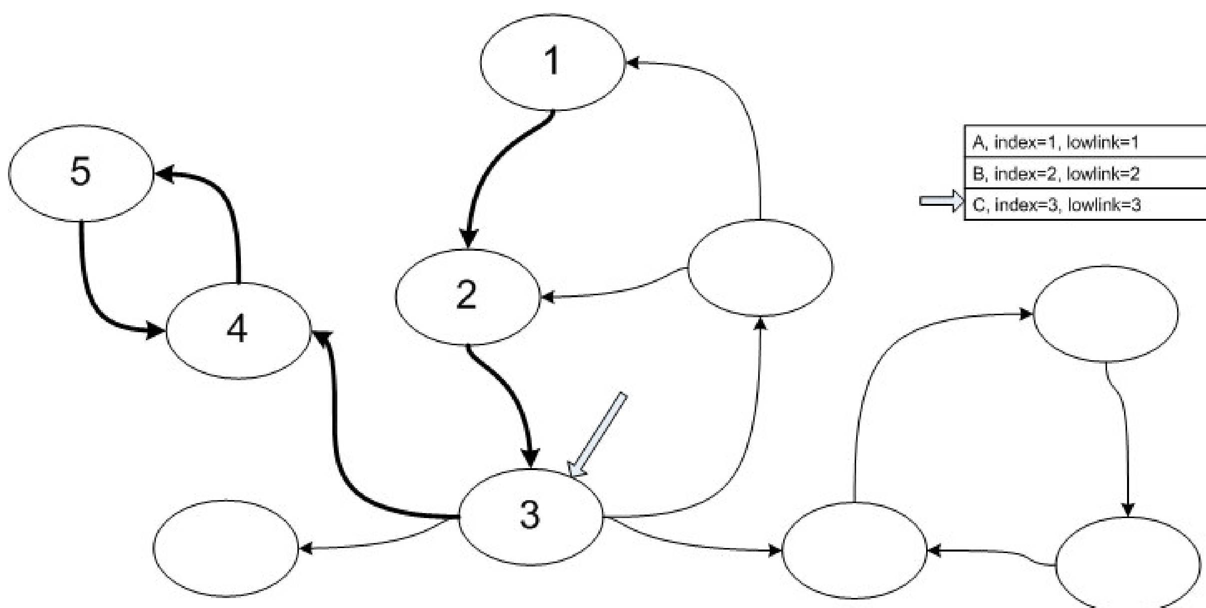
2. Jei bet kurios nagrinėjamos briaunos abi viršūnės *v* ir *v‘* jau buvo pradėtos nagrinėti (yra steke arba požymis *index* nėra tuščias), tuomet nagrinėjamos viršūnės *v* požymis *lowlink* surandamas pagal formulę *v.index = min(v.index, v'.index)*. Nagrinėjamame pavyzdyje viršūnės e požymis *lowlink* = 5, o viršūnės d – 4, todėl *e.lowlink* įrašoma reikšmė 4.
3. Jei aptinkama, kad viršūnė yra pilnai apdorota, be to yra šakninė stipriai susijusios komponentės viršūnė (*v.index = v.lowlink*), tai reiškia, kad aptikta stipriai susijusi grafo komponentė. Paveiksle 2.7 parodyta surasta šakninė stipriai susijusios komponentės viršūnė.



Pav. 2.7 Šakninė stipriai susijusios komponentės viršūnė

4. Aptikus šakninę stipriai susijusios komponentės viršūnę iš steko šaliname visas viršūnes, turinčias tokį patį *lowlink* požymį. Pašalinamos viršūnės yra vienos stipriai

susijusios komponentės viršūnės. Paveiksle 2.8 parodyta grafo būsena po viršūnių pašalinimo iš steko.



Pav. 2.8 Algoritmo žingsnis po viršūnių pašalinimo iš steko.

5. Algoritmas baigiamas, kai visos viršūnės yra apdorotos ir steke nebelieka viršūnių. Lentelėje 2.2 pateikiamos visų viršūnių požymių *index* ir *lowlink* reikšmės algoritmo vykdymo metu. Kaip ir ieškant stipriai susijusių komponenčių Kosarajaus algoritmu grafas turi 4 stipriai susijusios komponentes – {abcj}, {de}, {f} ir {gih}.

Lentelė 2.2. Grafo viršūnių požymių reikšmės Tarjano algoritmo vykdymo pabaigoje.

Viršūnė	***index***	***lowlink***
a	1	1
b	2	1
c	3	1
d	4	4
e	5	4
f	6	6
g	7	7
h	8	7
i	9	7
j	10	1

Toliau pateiktas Tarjano algoritmo veikimas pseudokodu:

```
Input: Graph G = (V, E), Start node v0

index = 0                       // Aplankytos viršūnės numeris
S = empty                       // Tuščias viršūnių stekas
tarjan(v0)                      // Pradėti Tarjano algoritmą pradinėje viršūnėje

procedure tarjan(v)
  v.index = index               // viršūnės indekso nustatymas
  v.lowlink = index
  index = index + 1
  S.push(v)                     // Įrašyti viršūnę v į steką
  forall (v, v') in E do        // Nagrinėjamos visos išeinančios briaunos
    if (v'.index is undefined)  // Ar buvo aplankyta gretima viršūnė
      tarjan(v')                // Rekursija
```

```
      v.lowlink = min(v.lowlink, v'.lowlink)
    elseif (v' in S)            // Ar gretima viršūnė v' jau yra steke?
      v.lowlink = min(v.lowlink, v'.index)
  if (v.lowlink == v.index)     // Ar v yra stipriai jungios komponentės šaknis
    print "SCC:"
    repeat
      v' = S.pop
      print v'
    until (v' == v)
```

Tarjano algoritmas veikia šiek tiek greičiau nei 2.3.1 skyriuje aprašytas Kosarajaus algoritmas, nes algoritmo vykdymo metu kiekviena viršūnė aplankoma po kartą (Kosarajaus algoritme – po du kartus). Visgi algoritmo vykdymo laikas lygiai taip pat priklauso nuo viršūnių ir briaunų skaičiaus, todėl algoritmo sudėtingumas toks pats – $O(E+V)$.

2.3.3. Gabovo algoritmas.

Algoritmas yra modifikuota Tarjano algoritmo versija. Šis algoritmas [9] labai panašus į Tarjano algoritmą, tačiau vietoje viršūnės požymių, nustatančių jungiąsias komponentes, jis naudoja dar vieną steką, kurio pagalba nustato, kada iš viršūnių steko pašalinti tos pačios jungiosios komponentės viršūnes.

Gabovo algoritmo vykdymo greitis labai panašus į Tarjano algoritmo, o sudėtingumas kaip ir anksčiau minėtų algoritmų – $O(V+E)$.

2.3.4. DCSC algoritmas

Šis algoritmas yra skirtas greitam stipriai susijusių komponenčių radimui daugiaprocesorinėse sistemose. Algoritmas buvo sukurtas naudoti programose, kurios naudoja labai didelius grafus, ir kurioms reikia greito stipriai susijusių komponenčių radimo [10].

Standartiniai jungiųjų komponenčių algoritmai remiasi grafo paieškos gilyn algoritmu, o šio algoritmo veikimo praktiškai neįmanoma išskaidyti į lygiagrečius procesus.

DCSC (*angl.* Divide-and-conquer strong component) algoritmas remiasi keliomis lemomis, kuriomis įrodoma, kaip greitai surasti visas viršūnes esančias toje pačioje stipriai susijusioje komponentėje, kaip ir nagrinėjama viršūnė.

Nagrinėjamos viršūnės, kurią algoritmo kūrėjai vadina ašine, atžvilgiu visos kitos grafo viršūnės yra suskirstomos į protėvius (*angl.* predessesor), palikuonis (*angl.* descendant) bei visas likusias (*angl.* remainder). Algoritmo kūrėjai įrodo, kad ašinės viršūnės protėvių ir palikuonių sankirta sudaro vieną stipriai susijusią grafo komponentę. Kitaip sakant

Desc(G; v) ∩ Pred(G; v) = SCC(G; v),

kur *Desc(G; v)* – ašinės viršūnės *v* palikuonių aibė, *Pred(G; v)* – ašinės viršūnės *v* protėvių aibė, o *SCC(G; v)* – ašinės viršūnės stipriai susijusi komponentė.

Tokiu būdu suradus vieną stipriai susijusią komponentę galima rekursiškai naudoti tą patį algoritmą visoms likusioms jungiosioms komponentėms surasti.

```
DCSC(G)
  If G is empty then Return.
  Select v uniformly at random from V.
  SCC <- Pred(G; v) ∩ Desc(G; v)
  Output SCC.
  DCSC(<Pred(G; v) \ SCC>)
  DCSC(<Desc(G; v) \ SCC>)
  DCSC(<Rem(G; v)>)
```

Šio algoritmo vykdymo laikas greitesnis nei kitų algoritmų, ir jo vykdymo laiką galima pagreitinti skaičiavimams naudojant daugiau nei vieną procesorių. Deja, bet šis algoritmas reikalauja sudėtingos grafo saugojimo duomenų struktūros, nes grafą būtina saugoti tokiu formatu, kad būtų galima greitai nustatyti bet kurios viršūnės protėvius bei palikuonius. Tai apsunkina algoritmo realizaciją, todėl šis algoritmas turėtų naudojamas tada, kai greitas jungiųjų komponenčių suradimas yra kritiškai svarbus uždavinys. Kitais atvejais pilnai pakanka naudoti bet kurį iš skyriuose 2.3.1 – 2.3.3 aptartų tiesinių algoritmų, skirtų orientuoto grafo stipriai susijusių komponenčių suradimui.

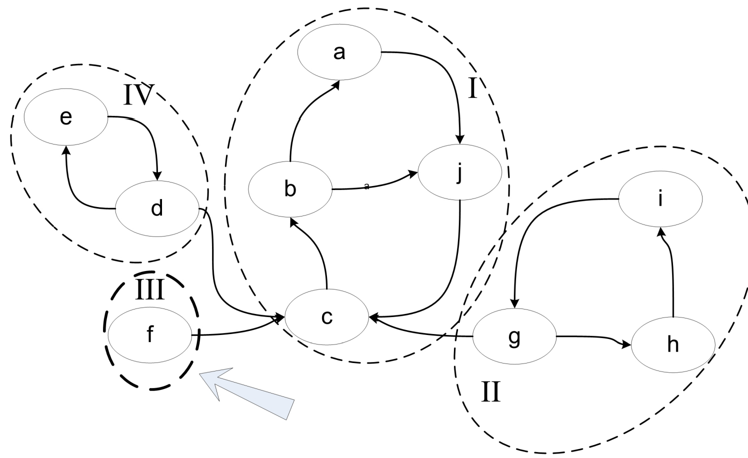
2.4. Saugumo ir gyvybingumo savybių tyrimo algoritmai

2.4.1. Aklaviečių radimas

Aklavietė – tai sistemos būsena, į kurią galime patekti, tačiau iš jos niekada nepereinama į jokią kitą sistemos būseną. Būsenų grafe tai būsena, turinti įeinančią briauną ar briaunas, tačiau neturinti išeinančių briaunų.

Aklaviečių radimo algoritmas:

1. Naudojant vieną iš aprašytų algoritmų surandamos stipriai susijusios būsenų grafo komponentės.
2. Suformuojamas stipriai susijusių komponenčių grafas.
3. Kiekviena jungiosi komponentė, kurią sudaro lygiai viena viršūnė, tačiau ši neturi išeinančių briaunų, yra aklavietė.



Pav. 2.9 Pasiekiamų būsenų grafo aklavietė

Paveiksle 2.9 parodytas būsenų grafas su surastomis stipriai susijusiomis komponentėmis. Komponentę III, sudaro viena viršūnė – f, be to ši komponentė neturi išeinančių briaunų, todėl tai yra aklavietė.

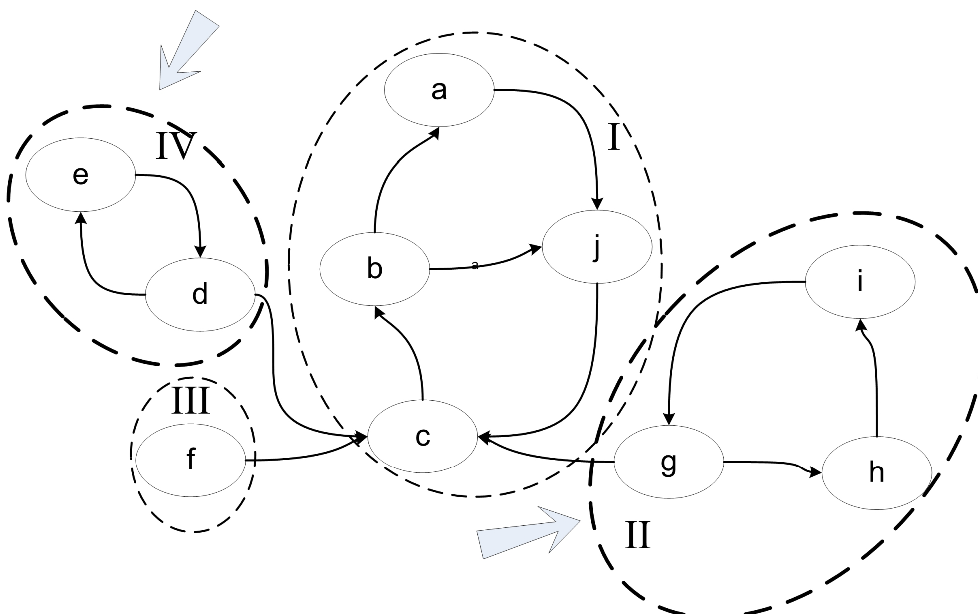
2.4.2. Uždarų ciklų radimas

Uždaras ciklas – tai sistemos būsenų aibė, į kurią patekus neegzistuoja perėjimas į jokias kitas sistemos būsenas. Kitaip sakant, jei sistema patenka į šias būsenas, ji patena į amžiną ciklą. Būsenų grafe tai stipriai susijusi komponentė, turinti daugiau nei vieną viršūnę ir neturinti išeinančių briaunų į kitas stipriai susijusias komponentes.

1 Naudojant vieną iš aprašytų algoritmų surandamos stipriai susijusios būsenų grafo komponentės.

2 Suformuojamas stipriai susijusių komponenčių grafas.

3 Kiekviena stipriai susijusi komponentė, kurią sudaro daugiau nei viena viršūnė, ir kuri neturi išeinančių briaunų, yra uždaras ciklas.



Pav. 2.10 Uždari ciklai pasiekiamų būsenų grafe

Paveiksle 2.10 parodytas būsenų grafas su surastomis stipriai susijusiomis komponentėmis. Šiame grafe yra 2 uždari ciklai – komponentė IV, kurią sudaro viršūnės e ir d, bei komponentė II, kurią sudaro viršūnės g, h ir i. Šias stipriai susijusias komponentes sudaro daugiau nei viena viršūnė, be to iš šių komponenčių nėra išeinančių briaunų į kitas komponentes.

2.4.3. Būsenų pasiekiamumo tikrinimas

Tiriant sistemą kartais yra naudinga sužinoti į kurias būsenas galima patekti, o į kurias – ne. Tai gali būti naudojama norint patikrinti, ar iš nagrinėjamos būsenos galima patekti į kitą norimą būseną.

Būsenų pasiekiamumo tikrinimo algoritmas:

1. Naudojant vieną iš aprašytų algoritmų surandamos stipriai susijusios būsenų grafo komponentės.
2. Suformuojamas stipriai susijusių komponenčių grafas.
3. Jei abi nagrinėjamos būsenos yra toje pačioje stipriai susijusioje komponentėje, arba egzistuoja kelias tarp pradinės ir galinės būsenos komponenčių, tuomet galinė būsena yra pasiekiama iš pradinės būsenos. Priešingu atveju būsenos yra nepasiekiamos.

2.4.4. Būsenos koordinačių apribojimų tikrinimas

Būsenos koordinatės – tai būsenos vektoriaus reikšmės. Patikrinus, ar būsenos koordinatės patenka į numatytus rėžius galima aptikti klaidas specifikacijoje.

Norint patikrinti būsenų koordinačių apribojimus reikia kiekvienai grafo viršūnei patikrinti būsenos koordinačių reikšmes.

2.4.5. Invarianto tikrinimas

Invariantas – tai predikatas, aprašytas sistemos būsenos vektoriaus kintamaisiais, matematiniais veiksmais ir loginėmis išraiškomis. Invariantas turi būti teisingas visoms sistemos būsenoms, nepriklausomai nuo įvykių.

Invarianto tikrinimas atliekamas kiekvienai grafo viršūnei tikrinant, ar užrašyta predikato išraiška yra teisinga.

2.5. Problemos sprendimas pasaulyje

Šiuo metu rinkoje egzistuoja keletas sistemų, atliekančių formalių specifikacijų integruotą analizę. Labiausiai paplitusios ir plačiausiai naudojamos yra sistemos *SPIN*, *VIS*, *SMV*, *NuSMV*, *Uppaal*, *PRANAS-2*, *VALSYS*. [11][12]

2.5.1. SPIN

Specifikavimo, imitacinio modeliavimo ir validavimo sistema *SPIN* yra plačiai naudojama kompiuterių protokolams tirti. SPIN buvo sukurtas kompanijos „Bell Labs“ dar 1980 metais. 2002 metais įrankis gavo prestižinį „System Software Award“ apdovanojimą. *SPIN* naudoja *C* specifikacijos notaciją, kuri padidina jo pritaikomumą pirmoje kūrimo stadijoje [13]. *SPIN* leidžia simuliavimą ir validavimą *PROMELA* kalba parašytai specifikacijai.

Analizė atliekama su atsitiktiniu arba interaktyviu (dialoginiu) imitavimu. Detalesniam sistemos nagrinėjimui validavimo įrankis patikrina specifikaciją aklaviečių, ciklų be išėjimo ir kt. atžvilgiu. Jeigu sistema yra tokia didelė, kad nepakanka sisteminių resursų (kompiuterio atminties), validavimas atliekamas su atsitiktinai parinktų būsenų aibėmis. Tokios priemonės yra pakankamos korektiškumui ir funkciniams reikalavimams, kurie gali būti realizuoti sistemos prototipe, įvertinti.

Sistemai, specifikuotai *PROMELA* kalboje, *SPIN* gali atlikti tos sistemos vykdymo imitacinį modeliavimą, arba generuoti programą *C* kalba, kuri vykdo sistemos savybių teisingumo patikrinimą realiu laiku [25]. Tikrintojas taip pat gali būti naudojamas tikrinti sistemos kintamųjų teisingumui, jis gali rasti nereikalingus ciklus, patikrinti sekančio laiko momento loginių formuluočių teisingumą. Tikrinimas turi būti naudingas ir naudoti minimalų atminties kiekį. Išsamus tikrinimas gali su matematine tikimybe nustatyti ar aprašytas sistemos elgesys yra be klaidų. Labai didelės tikrinimo problemos, kurios negali būti išspręstos su esama kompiuterine technika, gali būti bandomos spręsti su ekonomiška „būsenos saugojimo bitu“ technika. Šiuo metodu būsenos užimama vieta suskirstoma į mažą bitų skaičių pasiekiamoje sistemos būsenoje, su minimaliu pašaliniu efektu.

SPIN programos kalba PROMELA susideda iš procesų, pranešimų kanalų bei kintamųjų. Procesai yra globalūs objektai kurie atvaizduoja paskirstytos sistemos esybes. Procesai aprašo elgseną, kanalai ir globalūs kintamieji aprašo aplinką, kurioje procesas veikia [13][14].

SPIN gali būti naudojamas kaip:

- Imitatorius, leidžiantis greitą analizę naudojant atsitiktinę, valdomą arba dialoginę simuliaciją.
- Nuodugnus verifikatorius, galintis kruopščiai įrodyti vartotojo aprašytų reikalavimų specifikacijos teisėtumą.
- Apytikrio skaičiavimo sistema, leidžianti validuoti net labai didelius sistemų modelius maksimaliai išnaudojant būsenos vietą.

2.5.2. VIS

VIS (*Verification Interacting with Synthesis*) yra sistema, leidžianti atlikti sistemų, aprašytų formaliomis specifikacijomis, verifikavimą, sintezę ir simuliaciją. Pagrindinis reikalavimas aprašomai sistemai – ji turi turėti baigtinį skaičių būsenų.

VIS gali susintetinti baigtinio būsenų skaičiaus sistemą ir/arba verifikuoti sistemos formaliai aprašytų savybių teisingumą.

Pagrindinės *VIS* sistemos atliekamos funkcijos:

- Loginių ciklų simuliavimas ir aklaviečių bei uždarumo tikrinimas;
- Sudėtingų ir nuoseklių ciklų verifikavimas. [15]

2.5.3. SMV

SMV yra formaliai aprašytų sistemų validavimo ir verifikavimo įrankis. Pasinaudojant šiuo įrankiu galima verifikuoti ir validuoti sistemas, aprašytas išplėsta *SMV* formalizavimo kalba. *SMV* turi lengvai naudojamą ir nesunkiai įsimenamą grafinę vartotojo sąsają, bei leidžia dalinį aprašytos sistemos trasavimą. [16]

2.5.4. NuSMV

NuSMV yra patobulintas ir pagerintas *SMV* įrankis. *NuSMV* buvo sukurtas kaip atviros architektūros įrankis formaliai aprašytų sistemų verifikavimui, validavimui. Taip pat šį įrankį galima naudoti kaip branduolį kuriant savo verifikavimo ir validavimo sistemas. [17]

2.5.5. Uppaal

Uppaal yra integruotų įrankių aplinka, skirta formaliai aprašytų sudėtingų sistemų simuliavimui, validavimui ir verifikavimui. Ši sistema dažniausiai naudojama realaus laiko kontroleriams, komunikavimo protokolams tirti. Įrankis sukurtas bendradarbiaujant Danijos „Aalborg“ ir Švedijos „Uppsala“ universitetams.

Uppaal susideda iš trijų pagrindinių dalių: kalbos aprašymo, simuliatoriaus ir modelio tikrintojo. Aprašymo kalba yra nedeterminuota kalba su paprastais duomenų tipais (sveiki skaičiai, masyvai ir kt.). Ji naudojama formaliai aprašant tiriamą sistemą.

Imitatorius leidžia tikrinti sistemos veikimą, tikrinant visus galimus veikimo kelius. Imitacinis modeliavimas gali būti atliekamas jau ankstyvojo modeliavimo etapuose ir tokiu būdu suteikia galimybę aptikti ir ištaisyti galimas klaidas pačioje pradžioje [18]. Validatorius gali tikrinti ar sistema nepatenka į aklavietes, ar nesusidaro amžinų ciklų ir pan.

Pagrindinės UPPAAL savybės:

- Grafinis redaktorius sistemos aprašymui įvesti.
- Grafinis imitatorius vizualiai pateikia galimas sistemos veikimo kryptis. Taip pat gali būti vizualizuojama trasavimo informacija gauta iš validatoriaus.

- Reikalavimų specifikacijos redaktorius.
- Validatorius sistemos gyvybingumui tikrinti.
- Trasavimo informacijos generavimas, su galimybe grafiškai atvaizduoti rezultatus.

2.5.6. Pranas-2

Pranas-2 – tai KTU Verslo informatikos katedroje sukurta protokolų modeliavimo ir validavimo sistema. Ši sistema sukurta agregatinio metodo pagrindu ir ji naudojama *ESTELLE/Ag* kalbą. *ESTELLE/Ag* kalba yra *ESTELLE* ISO standarto modifikacija, specialiai pritaikyta *PLA* formalizavimui. Naudojant tokią kalbos modifikaciją galima sukurti formalias specifikacijas, kurios tinka validavimui ir imitaciniam modeliavimui.

Pranas-2 analizės sistemą sudaro tokios dalys:

- Specifikacijų redaktorius;
- Validavimo posistemis;
- Imitacinio modeliavimo posistemis.

Validavimo posistemis leidžia sugeneruoti pasiekiamumo grafą ir atlikti tokius patikrinimus:

- Pasiekiamumą – kokiu būdu iš pradinės būsenos galima pasiekti galinę būseną;
- Būsenų koordinačių apribojimus – galima nustatyti, ar būsenos koordinatės neišeina už iš anksto nustatytų ribų;
- Tolydžiųjų koordinačių reikalingumą – reikia nustatyti, ar specifikacijoje nėra aprašytų tolydžiųjų dedamųjų, kurios niekada negeneruoja vidinių įvykių;
- Aklaviečių paiešką – galima nustatyti, ar yra būsenų, iš kurių niekur neišeinama;
- Ciklų paiešką – galima nustatyti, ar nepatenkama į uždarą ciklą, kuriame kartojasi tam tikra įvykių seka. [20].

2.5.7. VALSYS

VALSYS – tai KTU Verslo informatikos katedroje sukurta agregatinių specifikacijų saugumo ir gyvybingumo tyrimo sistema. Sistemos paskirtis yra pasinaudojant pagal agregatines specifikacijas gautais tiriamos sistemos būsenų grafais ištirti jos veikimo korektiškumą bei specifikacijos pilnumą. Tai automatinio tikrinimo priemonė, turinti užtikrinti, jog kuriama agregatinėmis specifikacijomis aprašyta sistema nepateks į aklavietes, uždarus ciklus ar iš anksto nenumatytas, neapibrėžtas būsenas. Tai įgyvendinama analizuojant būsenų grafą ir ieškant būsenų, netenkinančių specifikacijos.

Pagrindinės šios sistemos funkcijos:

- Aklaviečių paieška;
- Uždarų ciklų paieška;

- Invarianto tikrinimas;
- Būsenų pasiekiamu tikrinimas.

2.5.8. SLAM

SLAM projektas – tai „Microsoft Research“ sukurtas įrankis, skirtas sistemos automatiniam validavimui [23]. Šiuo metu SLAM sėkmingai naudojamas Windows operacinės sistemos įrenginių tvarkyklių veikimo validavimui. Pagrindinės savybės:

- Dėmesys koncentruojamas ne į funkcinio programinės įrangos veikimo teisingumo tikrinimą, bet siekiama įrodyti, kad programa nepažeidžia kritinių sistemos savybių.
- SLAM tinka C kalboje parašytoms sistemoms. Sistemos validavimas paremtas baigtinių automatų teorija.
- SLAM atlieka sistemos gyvybingumo tikrinimą – randa amžinus ciklus, aklavietes, nustato teisingo/klaidingo veikimo įėjimus ir išėjimus.

2.5.9. TLC

TLC – tai įrankis, skirtas TLA+ kalba specifikuotų sistemų automatiniam validavimui ir imitavimui [21]. Taip pat šis įrankis naudojamas specifikacijos pilnumui ir vientisumui tikrinti. TLC pagalba gali būti valiuojamos ir imituojamos sąlyginai didelių sistemų specifikacijos. Esant didelei sistemai neužtektų resursų, analizuojant sistemos būsenas, todėl TLC analizuoja specifikaciją būsenos erdvėje, susiaurindamas analizuojamų būsenų skaičių, o vėliau pereinama prie kitų būsenų erdvių. Tokiu būdu pažingsniui išanalizuojamos visos būsenos. Kita savybė, dėl kurios TLC tinka didelėms sistemoms – simetrinės būsenos. TLC aptinka ir neanalizuoja simetrinių būsenų.

2.5.10. ADPRO

ADPRO – tai KTU sukurta agregatinėmis specifikacijomis aprašytų sistemų imitavimo įrankis. Šis įrankis praplečia kito, taip pat KTU sukurto įrankio SIMAS funkcionalumą dinaminio imitavimo galimybėmis [19].

3. Projektinė integruotos formalių specifikacijų analizės sistemos dalis

3.1. Programų sistemos funkcijos

Sudėtingų formalių specifikacijų integruotos analizės automatizavimo sistema susideda iš tokių posistemių:

- objektinis modelis;
- grafinis redaktorius.
- imitacinio modeliavimo posistemė;
- trasavimo posistemė;
- validavimo posistemė.

PLA metodu specifikuotos sistemos modelis aprašomas grafiniu redaktoriumi, o turinys saugomas *XML* failuose.

Objektinis modelis transliuoja ir saugo formalų sistemos aprašą į kompiuteriui suprantamą formatą. Objektinis modelis yra visų kitų posistemių pagrindas.

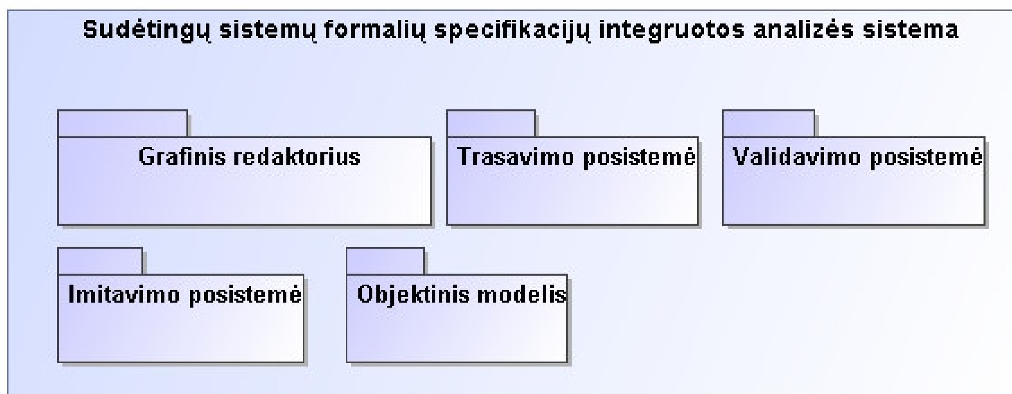
Grafinis redaktorius leidžia sukurti formalų sistemos aprašą naudojant vartotojo sąsają.

Trasavimo posistemė realizuoja modelio trasavimo ir verifikavimo funkcijas. Posistemės pagalba galima nesunkiai ir informatyviai atlikti specifikacijų teisingumo tikrinimą, išsamiai išanalizuoti saugumo ir gyvybingumo charakteristikas. Taip pat galima dirbtinu būdu sukelti nepageidaujamus įvykius (aklavietes, neapibrėžtų kintamųjų buvimą ir kt.) ir stebėti tolimesnę sistemos reakciją, tokiu būdų išanalizuojant beveik visas sistemos grėsmes ir labai ženkliai pagerinant sistemos kokybę.

Imitacinio modeliavimo posistemėje realizuojamas imitacinis *PLA* metodas. Validavimo posistemė atliekama specifikuotos sistemos validavimą.

Sukurta sistema leidžia ne tik patikrinti specifikacijos teisingumą atliekant sistemos validavimą bei verifikavimą, tačiau taip pat ir leidžia atlikti specifikacijos imitacinį modeliavimą bei žingsninį imitavimą.

3.2. Sistemos sudėtis

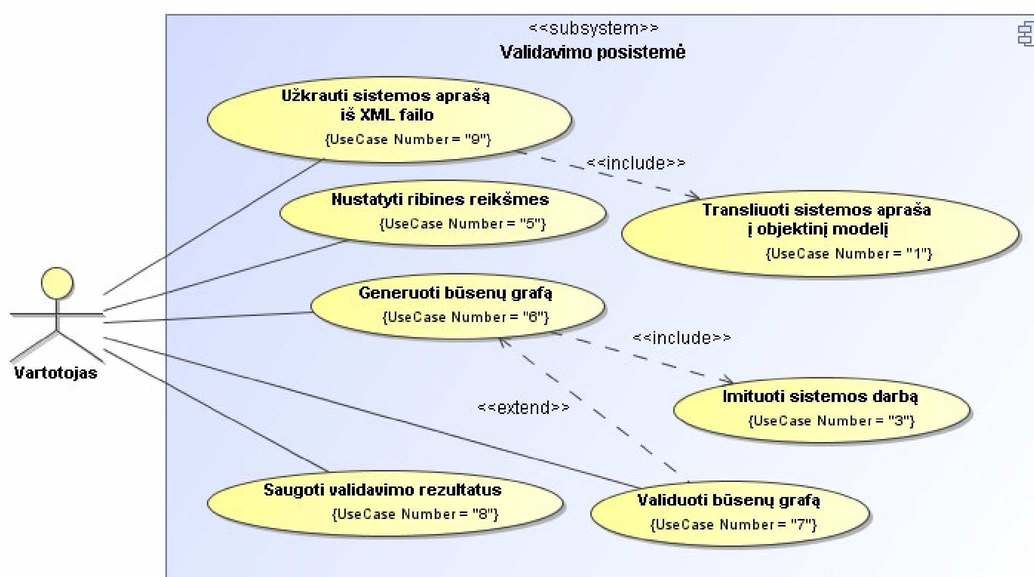


Pav. 3.1 Sistemos sudėtis

Paveiksle 3.1 parodyta formalių specifikacijų integruotos analizės sistemos sandara. Ši sistema susideda iš penkių pagrindinių dalių:

- Grafinis redaktorius – sistemos dalis, skirta analizuojamų sistemų aprašų kūrimui, redagavimui ir saugojimui *XML* formatu.
- Objektinis modelis – sistemos dalis, skirta analizuojamos sistemos aprašo saugojimui kompiuterio atmintyje darbo metu. Tai pagrindinė sistemos dalis, be kurios negalimas nei vienos kitos posistemės veikimas
- Imitavimo posistemė – sistemos dalis, skirta analizuojamos sistemos veikimo imitavimui.
- Trasavimo posistemė – sistemos dalis, skirta analizuojamos sistemos trasavimui.
- Validavimo posistemė – sistemos dalis, skirta analizuojamos sistemos aprašo validavimui.

3.3. Validavimo posistemės panaudojimo atvejai



Pav. 3.2 Validavimo posistemės panaudojimo atvejai

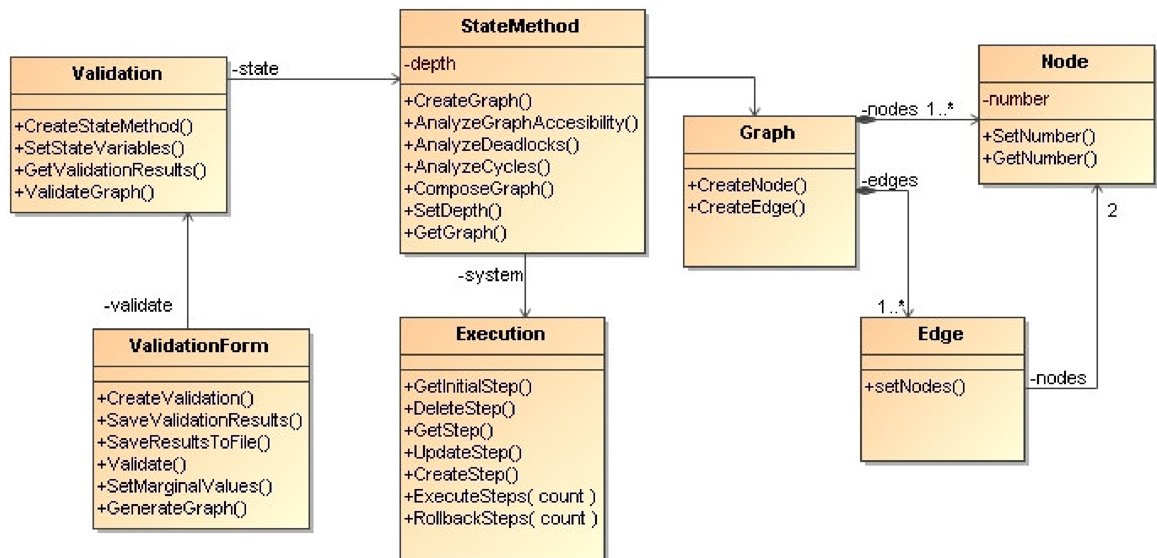
Paveiksle 3.2 parodyti formalių specifikacijų integruotos analizės sistemos validavimo posistemės panaudojimo atvejai. Ši posistemė turi tokius panaudojimo atvejus:

- **Užkrauti sistemos aprašą iš *XML* failo.** Apima procesą, kurio metu sistemos vartotojas atidaro analizuojamos sistemos aprašą, saugomą kompiuterio kietajame diske *XML* formatu
- **Transliuoti sistemos aprašą į objektinį modelį.** Apima procesą, kurio metu užkrautas analizuojamos sistemos aprašas sutransliuojamas į analizuojamos sistemos objektinį modelį, saugomą kompiuterio atmintyje
- **Nustatyti ribines reikšmes.** Apima procesą, kurio metu sistemos vartotojas nustato naujas arba palieka standartines analizuojamos sistemos aprašo validavimo ribines reikšmes.
- **Generuoti būsenų grafą.** Apima procesą, kurio metu generuojamas analizuojamos sistemos būsenų grafas.
- **Imituoti sistemos darbą.** Apima procesą, kurio metu yra vykdomas analizuojamos sistemos veikimo imitavimas.
- **Validuoti būsenų grafą.** Apima procesą, kurio metu atliekamas sugeneruoto sistemos būsenų grafo validavimas.
- **Saugoti validavimo rezultatus.** Apima procesą, kurio metu vartotojas saugo būsenų grafo validavimo rezultatus pasirinktame faile kompiuterio kietajame diske.

3.4. Validavimo posistemės klasių diagrama

Validavimo posistemė yra skirta formalaus aprašo teisingumo tikrinimui. Posistemė leidžia patikrinti sudaryto formalaus sistemos aprašo teisingumą prieš pradedant darbą su sistemos aprašu. Validavimo posistemė leidžia aptikti tokias klaidas, kaip nepasiekiamos aprašo vietos, susidarančios aklavietės, arba susidarantys amžini ciklai. Ši posistemė beveik pilnai automatizuota, todėl vartotojui užrenka paleisti posistemę, ir jai baigus vykdyti skaičiavimus parodomos visos formalaus aprašo klaidos.

Paveiksle 3.3 parodyta validavimo posistemės klasių diagrama.



Pav. 3.3 Validavimo posistemės klasių diagrama

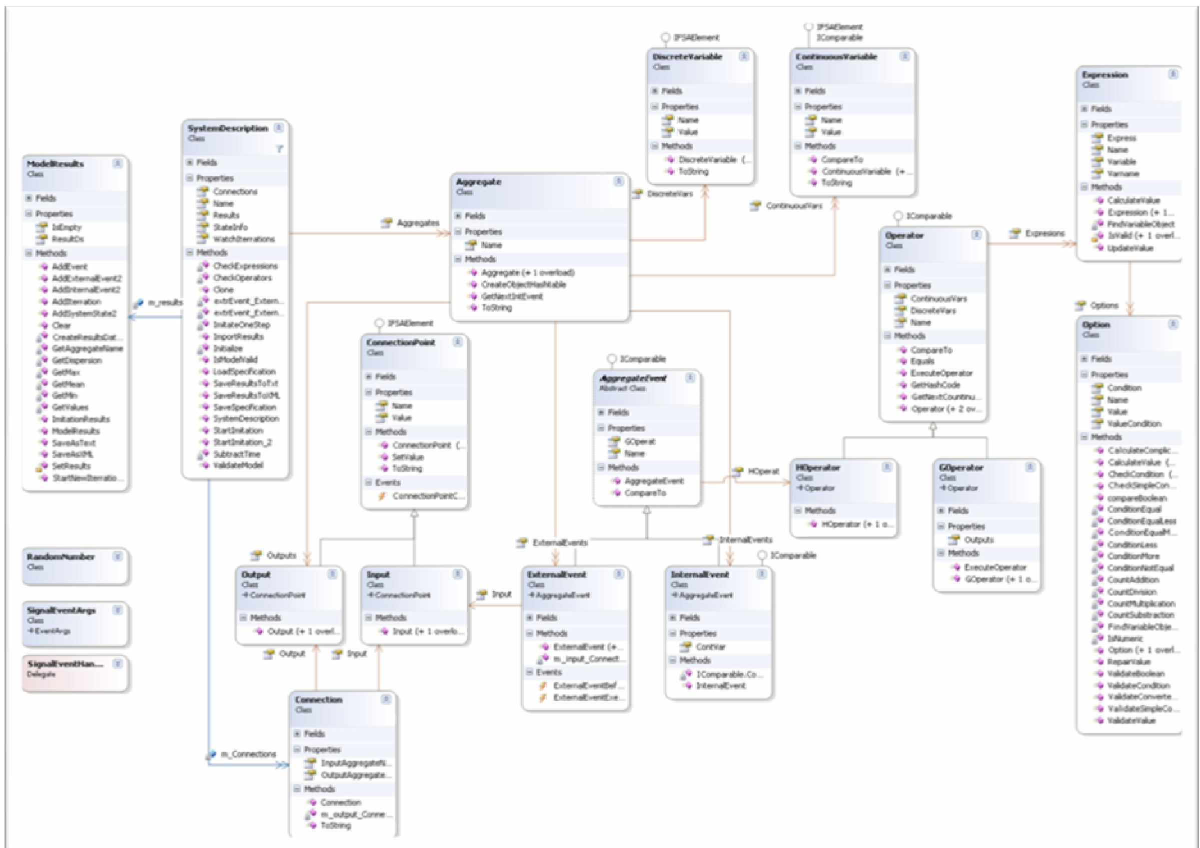
Posistemėje yra tokios klasės:

- **ValidationForm**. – Sąsaja skirta darbui su validavimo posisteme. Jos pagalba gali būti validuojamas formalus aprašas ir gaunami validavimo rezultatai.
- **Validation**. – Klasė atsakinga už validavimo posistemės darbą.
- **StateMethod** – Klasė atsakinga už pagrindinių operacijų vykdymą.
- **Graph** – Klasė skirta sistemos būsenų grafui saugoti.
- **Edge** – Klasė skirta grafo briaunos saugojimui.
- **Node** – Klasė skirta būsenų grafo viršūnėms saugoti.
- **Execution** – Klasė atsakinga už sistemos pažingsninį vykdymą.

3.5. Objektinio modelio klasių diagrama

Objektinis modelis skirtas analizuojamos sistemos aprašo saugojimui kompiuterio atmintyje darbo metu. Objektiniame modelyje saugoma visa *PLA* formalizavimo kalba aprašyta sistemos informacija: sistemos sujungimai, signalai, agregatai ir su jais susijusi informacija (įėjimai, išėjimai, diskretieji ir tolydieji kintamieji, H ir G operatoriai). Taip pat objektinis modelis atsakingas ir už aprašytos sistemos Objektinis modelis yra visų sistemos skaičiavimų pagrindas, todėl ją naudoja visos kitos posistemės.

Paveiksle 3.4 parodyta objektinio modelio klasių diagrama.



Pav. 3.4 Objektinio modelio klasių diagrama

Objektinį modelį sudaro tokios klasės:

- **Aggregate** – klasė aprašanti agregatą;
- **AggregateEvent** – abstrakti klasė aprašanti agregato įvykį;
- **Connection** – klasė aprašanti sujungimą, kuris susideda iš įėjimo ir išėjimo taškų;
- **ConnectionPoint** –klasė aprašanti sujungimą;
- **ContinuousVariable** – klasė aprašanti tolydų kintamąjį;
- **DiscreteVariable** – klasė aprašanti diskretų kintamąjį;
- **Expression** – klasė aprašanti išraišką;
- **ExternalEvent** – klasė aprašanti išorinį įvykį, paveldi klasę *AggregateEvent*;
- **GOperator** – klasė aprašanti G operatorių, paveldi Operator klasę;
- **HOperator** – klasė aprašanti H operatorių, paveldi Operator klasę;
- **Input** – klasė aprašanti įėjimą, paveldi *ConnectionPoint* klasę;
- **InternalEvent** – klasė aprašanti vidinį įvykį, paveldi klasę *AggregateEvent*;
- **ModelResults** – klasė atsakinga už imitacinio modeliavimo rezultatų saugojimą, eksportavimą;

- **Operator** – klasė aprašanti visus operatorius;
- **Option** – klasė aprašanti išraiškos alternatyvą;
- **Output** – klasė aprašanti išėjimą, paveldi *ConnectionPoint* klasę;
- **RandomNumber** – klasė generuojanti atsitiktinius skaičius;
- **SystemDescritpion** – sistemos aprašas.

3.6. Validavimo posistemės darbo langas

Pav. 3.5 Pagrindinis validavimo posistemės langas

Paveiksle 3.5 parodytas pagrindinis validavimo posistemės. Šis langas susideda iš trijų dalių. Viršuje esanti meniu juosta valdo programos langą, leidžia iškviesti vykdymo veiksmus. Kairėje esančiame lange rodomas sugeneruotas pasiekiamų būsenų grafas. Dešinėje pusėje rodomos klaidos, surastos validavimo metu.

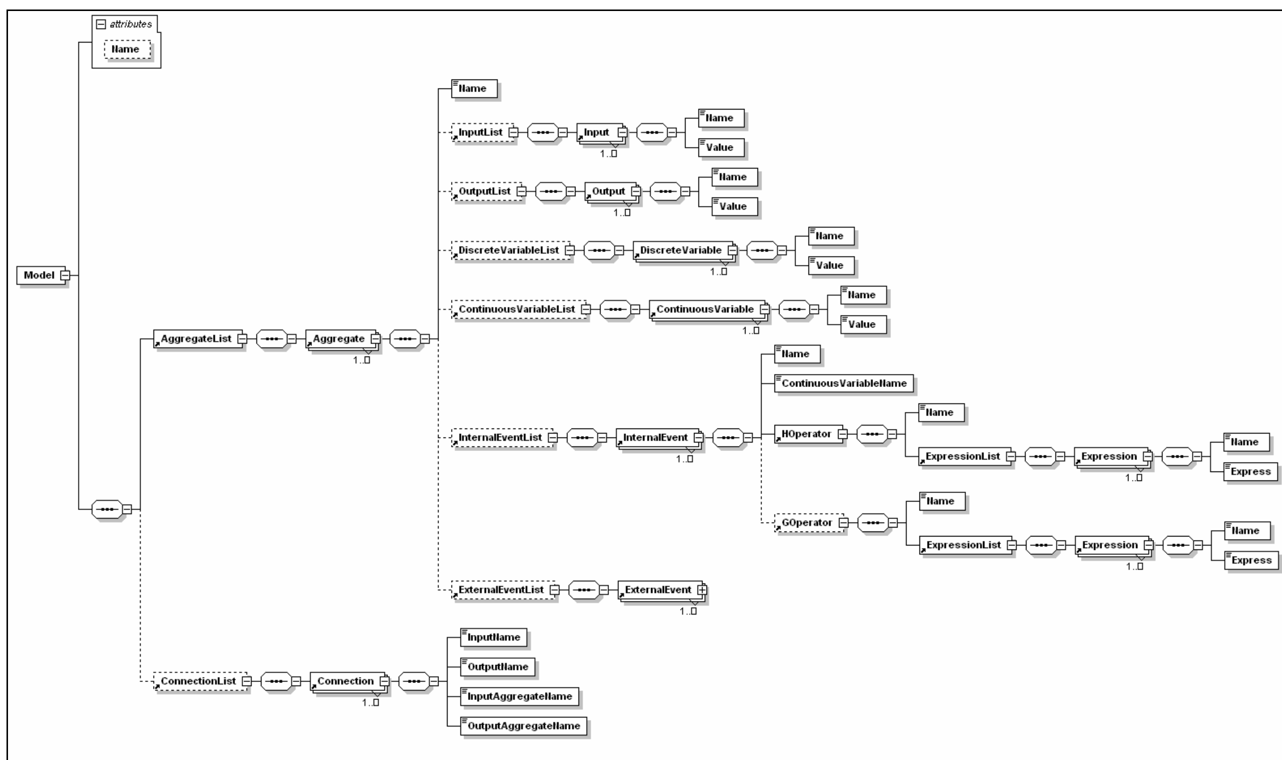
3.7. Sistemos duomenų vaizdas

Sistema naudoja labai daug pradinių duomenų (priklausomai nuo nagrinėjamos sistemos). Vis dėlto dėl specifinės duomenų paskirties ir struktūros duomenims saugoti nenaudojama reliacinių duomenų bazių valdymo sistema (RDBVS). RDBVS naudojimas šiuo atveju tik sulėtintų sistemos veikimą, kadangi prieš naudojant duomenis sistemoje reiktų juos išgauti iš duomenų bazės, bei papildomai paruošti, o tam būtų sugaištamas bereikalingas laikas.

Pagal duomenų specifiką, sistemos duomenis patogiausia saugoti XML formato faile. Dėl savo universalumo XML leis paprastai ir patogiai išsaugoti sistemos aprašą, kurį bus lengva sukurti ar redaguoti be papildomų priemonių. Be to, kadangi XML formatas yra

nepriklausomas nuo technologijos, sistemos duomenys būtų tinkami naudoti netgi tuo atveju, jei sistema būtų perkurta naudojant kitą technologiją nei pradinėje realizacijoje.

XML faile aprašytos tokios analizuojamos sistemos dalys bei ryšiai tarp jų pavaizduoti paveiksle 3.6



Pav. 3.6 XML faile aprašomos duomenų struktūros ir ryšiai tarp jų.

4. Pasiekiamų būsenų grafo sudarymo metodai

4.1. Pasiekiamų būsenų grafo sudarymo algoritmas

Svarbiausias formalių agregatinių specifikacijų validavimo metodas – tai pasiekiamų būsenų grafo sudarymas ir analizė. Sprendžiant validavimo uždavinį pasiekiamų būsenų grafo metodu svarbu sėkmingai išspręsti dvi užduotis – sudaryti pasiekiamų būsenų grafą, bei atlikti grafo saugumo ir gyvybingumo savybių tikrinimą.

Agregatinių specifikacijų pasiekiamų būsenų grafo generavimui naudojamas toks algoritmas [1]:

1. Atliekama visų sistemos agregatų kompozicija. Atlikus šią kompoziciją gaunamas agregatas, kuris neturi įėjimo ir išėjimo signalų.
2. Nusakoma pradinių būsenų aibė.
3. Apibrėžiama galutinių būsenų aibė.
4. Pasiekiamų būsenų grafas susideda iš viršūnių aibės V ir lankų aibės L.
5. Sudaroma nenagrinėtų būsenų aibė V_N, į kurią pradžioje dedamos visos pradinės būsenos. Paskui imama būsena $z \in V_N$ ir sudaroma tiek lankų, kiek yra aktyvių operacijų. Nenagrinėtų viršūnių aibė pildoma būsenomis į kurias pereinama iš būsenos z, bet kurios dar nebuvo nagrinėtos.
6. Grafas baigiamas sudaryti, kai nelieka nenagrinėtų viršūnių, t.y. būsenų V_N aibė tampa tuščia

Pasiekiamų būsenų grafo generavimo algoritmą pseudokodu galima aprašyti taip:

```
Input: SystemState s = state;       //pradinė sistemos būsena

Queue Q = empty                     //neapdorotų būsenų eilė
Graph G = empty                     //suformuotas būsenų grafas
Vector v = s.GetStateVector();      //sistemos būsenos vektorius
G.AddNode(v);                       //prie grafo pridedame pradinę viršūnę
Q.Push(s);                          //sistemos būseną įrašome į eilę
While(Q <> empty)                   //kol eilė ne tuščia
{
   SystemState s1 = Q.Pop();
   V1 = s1.GetStateVector();
   Forall (AvailableEvent) in s1.AvailableEvents();         //kiekvienas įvykis
   Begin
      SystemState s2 = s1.Execute(AvailableEvent);          //įvykdomas įvykis
      Vector v2 = s2.GetStateVector();
      If(not G.Has(v2))             //Jei viršūnės nėra grafe
         begin
            G.AddNode(v2);          //Įtraukiame viršūnę
            Q.Push(s2);             //Įtraukiame sistemos būseną į eilę
         end
      G.AddEdge(v1, v2);            //Įtraukiame briauną į grafą
   End
}

Return G;
```

Algoritme naudojama abstrakti duomenų struktūra – eilė – kurioje saugomos nagrinėjamos sistemos būsenos. Kiekviena sistemos būsena saugo visą agregatinės sistemos informaciją – agregatų aprašymus, kintamųjų reikšmes, būsimus įvykius. Algoritmo pradžioje į būsenų eilę įrašomos pradinės sistemos būsenos. Vykdant ciklą, iš būsenų eilės imamos nenagrinėtos sistemos būsenos ir kiekvienai būsenai nustatomi įvykiai, kurie gali įvykti sistemoje. Kiekvienam įvykiui, kuris sistemos būsenoje gali įvykti, sukuriama po naują sistemos būseną. Jei tokia sistemos būsena dar nebuvo apdorota, ji įrašoma į būsenų eilę ir apdorojama vėlesniuose algoritmo etapuose, o nauja grafo viršūnė ir/ar briauna įtraukiamos į būsenų grafą.

Toks algoritmas leidžia suformuoti pilną būsenų grafą. Deja, aprašytas algoritmas veikia pakankamai lėtai, nes kiekvienu momentu atliekami skaičiavimai tik su viena sistemos būsena. Šio algoritmo veikimą galima pagerinti naudojant lygiagretų programavimą.

4.2. Lygiagretus pasiekiamų būsenų grafo sudarymo algoritmas

4.2.1. Lygiagretus programavimas

Šiais laikais kompiuterio procesorių greitaveikos didinimas siejamas daugiausiai ne su procesorių greitaveikos, o su jų kiekio didinimu. Prieš keletą metų namų rinkai skirtuose kompiuteriuose pradėjo masiškai atsirasti kelių fizinių branduolių procesoriai, leidžiantys vienu metu vykdyti kelias instrukcijas. Lygiagretaus programavimo naudojimas ypač išpopuliarėjo paplitus masinei kelių branduolių procesorių gamybai, kai galingi kelių branduolių procesoriai tapo prieinami ne tik superkompiuteriuose ar serveriuose, tačiau ir kasdieninio naudojimo asmeniniuose kompiuteriuose.

Lygiagretus programavimas, kurio metu kelios instrukcijos vykdomos vienu metu, jau seniai naudojamas kurti programoms, kurios vienu metu turi atlikti daug užduočių. Lygiagretus programavimas remiasi principu, kad dideles užduotis dažnai galima išskaidyti į keletą smulkių, kurias galima vykdyti vienu metu (lygiagrečiai).

Programų greitaveikos ir našumo padidėjimas kompiuteriuose, didžiąja dalimi priklauso ne nuo techninės, o nuo programinės įrangos kūrėjų. Šiuo klausimu techninės įrangos gamintojai smarkiai lenkia programinės įrangos gamintojus – nors daugelio procesorių sistemos prieinamos jau gana seniai, tik nedidelė dalis šiandien naudojamų programų pilnai išnaudoja galimybes skaičiavimus vykdyti lygiagrečiai.

Neatsargus lygiagretaus programavimo naudojimas gali tapti programos problemų priežastimi. Dažniausiai pasitaiko tokios lygiagretaus programavimo problemos [22]:

- Netinkamas gijų skaičiaus pasirinkimas. Pasirinkus per didelį gijų kiekį bus eikvojama per daug sisteminių resursų ir vietoje norimo programos veikimo pagreitėjimo galime gauti sulėtėjimą.
- Lygiagretaus programavimo naudojimas netinkamiems uždaviniams spręsti. Gijų sukūrimui bei paruošimui taip pat reikalingi sisteminiai resursai, todėl reikia įvertinti, ar mažų uždavinių sprendimas naudojant lygiagretų programavimą yra tinkamas.
- Sinchronizacijos problemos. Tai dažniausiai pasitaikanti problema, kuri pasireiškia nekorektišku lygiagrečiai veikiančių gijų ar procesų bendravimu. Dažnai pasitaiko situacija, kad lygiagrečiai veikiančios užduotys turi naudoti tuos pačius resursus (pavyzdžiui vieną failą, objektą ar kitą sisteminį resursą). Tokiu atveju būtina užtikrinti, kad lygiagrečios gijos veiksmus atliktų tik tokia tvarka, kokia turėtų, taip pat, turėtų savalaikį priėjimą prie visų reikalingų resursų. Taip galima išvengti susidarančių sistemos veikimo aklavietčių.

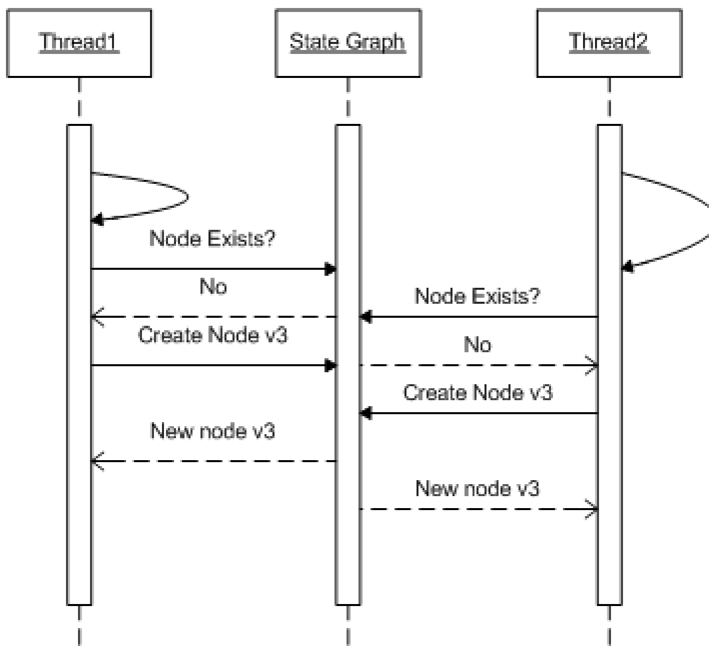
4.2.2. Pasiekiamų būsenų grafo sudarymo algoritmas

4.1. skyriuje aptarto pasiekiamų būsenų grafo algoritmo greitaveiką galima pagerinti panaudojus lygiagretų programavimą. Naudojant tradicinį algoritmą vienu metu apdorojama tik viena sistemos būsena, tuo tarpu kitos yra įrašytos į eilę ir laukia apdorojimo.

Sinchronizacijos problemos lygiagrečiame programavime dažniausiai iškyla dėl to, kad lygiagretūs procesai ar gijos bando vienu metu prieiti prie to paties bendrai naudojamo resurso. Algoritmo vykdymo metu naudojami du bendri resursai, kurių sinchronizacijos problemas reikia įvertinti – tai algoritmo metu formuojamas būsenų grafas, bei sistemos būsenų eilė.

Visi eilės elementai įrašomi į eilės pradžią, o skaitomi iš eilės pabaigos. Net jei tuo pačiu metu bandytume ir įrašyti elementą, ir nuskaityti, vis tiek gautume tokį patį rezultatą nepriklausomai nuo to, kuris veiksmas buvo atliktas pirmiau. Dėka šių eilės savybių, realizuojant lygiagretų pasiekiamų būsenų grafo sudarymo algoritmą galima nesirūpinti sistemos būsenų eilės priėjimo kontrole iš skirtingų gijų.

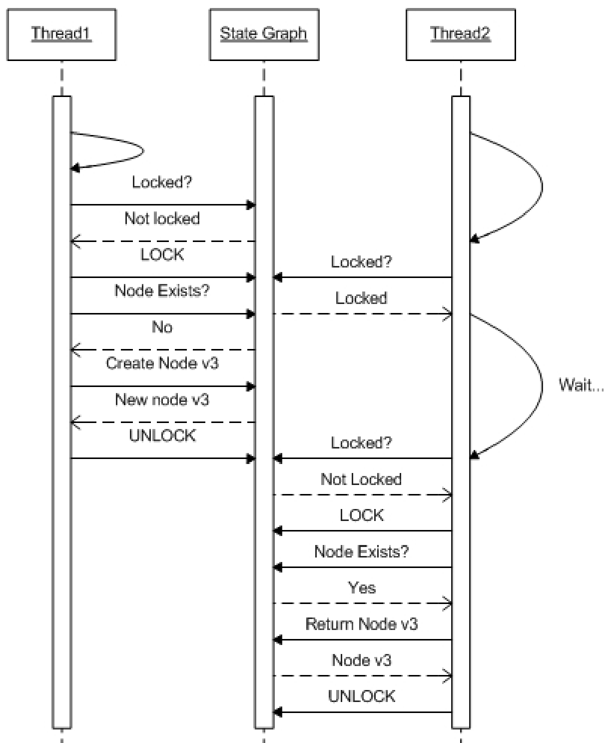
Kitas bendrai naudojamas resursas – tai būsenų grafas. Skirtingos gijos vienu metu gali tiek rašyti į grafą (įtraukti naujai suformuotą viršūnę), tiek skaityti iš grafo informaciją (tikrinti, ar viršūnė jau yra grafe). Paveiksle 4.1 parodyta situacija, kuri gali iškilti, jei algoritme nebus atsižvelgta į būsenų grafo naudojimą keliose gijose.



Pav. 4.1 Bendrai naudojamo resurso problema.

Pavyzdyje turime dvi gijas (Thread1 ir Thread2), kurios nagrinėja skirtingas sistemos būsenas v1 ir v2. Abi gijos vienu metu pradeda skaičiuoti sekančią sistemos būseną. Thread1 apskaičiuoja, kad sekanti sistemos būsena – tai viršūnė v3. Gija tikrina, ar būsenų grafe jau yra tokia viršūnė. Kadangi tokios viršūnės nėra, Thread1 pateikia užklausą sukurti grafo viršūnę v3. Tuo tarpu skaičiavimus baigia gija Thread2, ir sekanti sistemos būsena pasirodo esanti tokia pati, kaip ir pirmos gijos – v3. Gija užklausia grafo, ar jame jau yra viršūnė v3. Kadangi grafas dar nespėjo sukurti viršūnės v3, kai to paprašė pirma gija, jis neranda šios viršūnės grafe ir grąžina gijai Thread2 neigiamą atsakymą. Tuomet antra gija siunčia grafui užklausą sukurti viršūnę v3. Šiuo atveju grafas gavo dvi užklausas sukurti identišką viršūnę, ir kadangi pirma užklausa nebuvo apdorota iki to laiko, kai atėjo nauja užklausa, grafe bus sukurtos dvi identiškos viršūnės.

Tokią problemą galima išspręsti naudojant „užraktą“. Šiuo atveju gija, kuri naudoja bendrą resursą – grafą – jį „užrakina“ savo naudojimui, o „atrakina“ tada, kai resursą panaudoja ir šio nebereikia. Galimas supaprastintas problemos sprendimas pavaizduotas paveiksle 4.2.



Pav. 4.2 Bendrai naudojamo resurso problemos sprendimo būdas

Šiuo atveju prieš atlikdama grafo užklausas gija Thread1 jį „užrakina", ir tokiu būdu tik ši gija turi priėjimą prie grafo. Jei tuo metu, kol grafas „užrakintas" užklausas bando pateikti kitos gijos, jos to padaryti negali, ir privalo laukti, kol grafą „atrakins" jį naudojanti gija.

Nors paveiksluose 4.1 ir 4.2 parodyta, kad gijos didesnę laiko dalį laukia arba bando gauti priėjimą prie grafo, tai nėra tiesa. Paveiksluose būsenos apskaičiavimo trukmė parodyta ypač maža, nors iš tikrųjų ji užims didžiąją laiko dalį, o priėjimas prie resursų užims tik mažą laiko tarpą.

Modifikuotas algoritmas pseudokodu galėtų būti aprašytas taip:

```
Input: SystemState s = state;       //pradinė sistemos būsena

Queue Q = empty                     //neapdorotų būsenų eilė
Graph G = empty                     //suformuotas būsenų grafas
Vector v = s.GetStateVector();      //sistemos būsenos vektorius
G.AddNode(v);                       //prie grafo pridedame pradinę viršūnę
Q.Push(s);                          //sistemos būseną įrašome į eilę
While(Q <> empty and ANY WORKING THREADS IN Threads)        //kol eilė ne tuščia
{
   SystemState s1 = Q.Pop();
   Threads.Add(new Thread(ProcessState(s1));    //Paleisti naują giją
}

ProcessState(SystemState ss)
{
   v1 = ss.GetStateVector();
   Forall (AvailableEvent) in ss.AvailableEvents();         //kiekvienas įvykis
   Begin
      SystemState s1 = ss.Execute(AvailableEvent);          //įvykdomas įvykis
      Vector v2 = s1.GetStateVector();
      LOCK_RESOURCES();             //"Užrakinami" resursai
      If(not G.Has(v2))             //Jei viršūnės nėra grafe
         begin
            G.AddNode(v2);          //Įtraukiame viršūnę
            Q.Push(s2);             //Įtraukiame sistemos būseną į eilę
         end
```

```
      G.AddEdge(v1, v2);              //Įtraukiame briauną į grafą
      UNLOCK_RESOURCES();             //"Atrakinami" panaudoti resursai
   End

}

Return G;
```

Šio algoritmo veikimas yra panašus į aprašytą anksčiau. Esminis skirtumas – sistemos būsenų eilės apdorojimas. Jei anksčiau sistemos būsenų aibė buvo apdorojama cikle ir vienu metu buvo galima apdoroti tik vieną sistemos būseną, dabar kiekvieną viršūnę apdoroja kita gija. Reikia pabrėžti, kad šiuo atveju negalima algoritmo baigti tada, kai sistemos būsenų eilė yra tuščia, kadangi tikėtina, kad tuo metu skaičiavimus atlikinės bent viena gija. Skaičiavimus galima baigti tik tada, kai eilė tuščia, bei visos grafo generavimo gijos baigė darbą.

Taip pat svarbu įvertinti laiką, kurį gija gali naudoti grafą. Jei atsitiktų nenumatytų problemų ir gijos, kuri turi „užrakinusi" bendrai naudojamą resursą – grafą – vykdyme atsirastų klaidų, sistema privalo atpažinti, jog gija per ilgai naudoja resursą ir šios gijos darbą reikia nutraukti, kad bendrais resursais galėtų naudotis kitos programos dalys.

4.2.3. Būsenų grafo sudarymo algoritmo gijų skaičiaus parinkimas

Lygiagretaus algoritmo naudojimas gali ženkliai pagreitinti būsenų grafo sudarymą. Siekiant pagerinti algoritmo greitaveiką būtina nustatyti, kiek gijų naudojant pasiekiami geriausi rezultatai.

Aptarsime kelis galimus variantus:

- Gijų skaičius neribojamas. Tokiu atveju kiekvienai naujai nenagrinėtai sistemos būsenai iš karto sukuriama nauja gija. Atrodo, kad toks gijų skaičius turėtų smarkiai pagerinti algoritmo greitaveiką. Deja, bet tai bus teisinga tik sistemoms, turinčioms labai mažą būsenų kiekį. Teoriškai Windows operacinės sistemos neturi apriboto proceso gijų skaičiaus. Iš tikrųjų gijų skaičių riboja naudojamos techninės įrangos pajėgumai – kiekviena gija privalo gauti dalį sistemos resursų. Jei šių resursų nebepakanka – gijų būsenos saugojimas ženkliai sulėtėja ir jų saugojimas ne pagreitina, o priešingai – sulėtina – programos darbą. Dėl šios priežasties neriboto gijų skaičiaus naudoti nerekomenduojama.
- Naudojamas dalinai apribotas gijų skaičius. Šiuo atveju, kaip ir pirmuoju, kiekvienai naujai sistemos būsenai sukuriama nauja gija. Visgi šį kartą apribotas maksimalus gijų skaičius. Šis sprendimas leidžia greitai pradėti naudoti maksimalų sistemai apibrėžtą gijų kiekį ir pagerinti programos greitaveiką, ypač daugelio procesorių sistemose. Deja, bet tokio metodo priežiūra yra gana sudėtinga, nes gijų kūrimas yra hierarchinis

(t.y. sukurta gija gali kurti naujas gijas) ir kai kuriais atvejais gali iškilti problemų norint suvaldyti visas sukurtas gijas.

- Naudojamas ribotas, iš anksto nustatytas gijų skaičius. Šis skaičius turi būti ne per didelis, kitaip susidursime su pirmame variante aprašytomis problemomis. Realizavus sprendimą šiuo variantu, visos sistemos būsenos būtų saugomos eilėje, kurią nuolat naudotų ribotas gijų skaičius. Tokiu atveju gijas gana lengva valdyti, tuo pačiu galima naudoti pakankamą jų kiekį, kad žymiai pagerinti algoritmo greitaveiką.

Sistemos realizacijai pasirinktas paskutinis variantas. Tikslus gijų skaičiaus pasirinkimas priklauso nuo konkretaus nagrinėjamo formalių specifikacijų uždavinio. Jei kiekvienos iteracijos metu eilėje atsiranda vos kelios naujos sistemos būsenos, tuomet net ir kelios gijos duoda pakankamai didelį greitaveikos padidėjimą. Tuo tarpu jei įvykdžius eilinę iteraciją atsiranda dešimtys naujų būsenų, reikėtų pagalvoti apie didesnį gijų skaičių, nes tokiu atveju galima efektyviau išnaudoti lygiagretaus programavimo privalumus

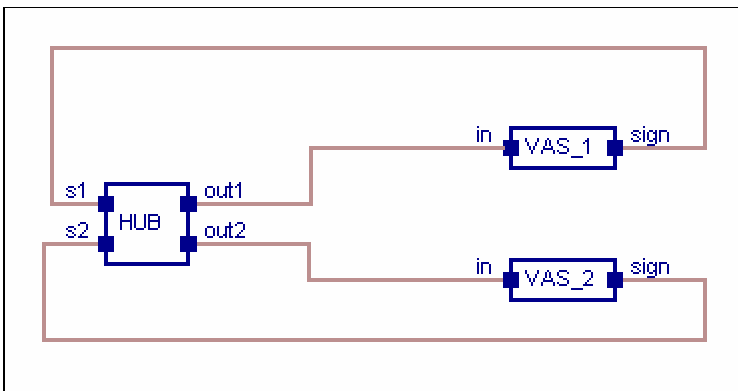
Eksperimentinėje darbo dalyje bus praktiškai išbandytas lygiagretus grafo generavimo algoritmas bei atlikti matavimai, nustatantys grafo generavimo pagreitėjimą naudojant šį algoritmą.

5. Eksperimentai su FSA sistema

5.1. Validavimo posistemės veikimo eksperimentai

Šiais eksperimentais bus siekiama parodyti, kad sukurta formalių specifikacijų analizės sistemos validavimo posistemė sėkmingai aptinka specifikacijose esančias klaidas. Taip pat bus siekiama parodyti, kad tiek naudojant nuoseklų grafo generavimo algoritmą, tiek lygiagretų, gaunami vienodi pasiekiamų būsenų grafai.

Sistemos veikimo patikrinimui naudosime sistemą iš dviejų vienkanalių aptarnavimo sistemų bei paraiškų generatoriaus. Ši sistema parodyta paveiksle 5.1.



Pav. 5.1 Vienkanalė aptarnavimo sistema su šakotuvu

Ši sistema susideda iš trijų agregatų. Agregatas HUB atsitiktiniu laiko tarpu generuoja paraiškas ir siunčia agregatams VAS_1 ir VAS_2. Jei neperpildytas agregato VAS_1 buferis, paraiška siunčiama agregatui VAS_1, priešingu atveju – agregatu VAS_2. Jei agregato VAS_1 buferis persipildo, šis siunčia pranešimą agregatui HUB, kad jo buferis persipildė – tuomet agregatas jam paraiškų nebesiunčia. VAS_1 agregatas atsitiktiniais laiko tarpais apdoroja paraiškas – taip mažėja buferyje esantis paraiškų skaičius. Agregatas VAS_2 veikia analogiškai agregatui VAS_1.

Šios sistemos formali specifikacija:
Agregatas HUB:

1. Įėjimo signalų aibė: $X = \{s1, s2\}$
2. Išėjimo signalų aibė: $Y = \{out1, out2\}$
3. Išorinių įvykių aibė: $E' = \{atejo_s1, atejo_s2\}$,
   atejo_s1 – į agregatą atėjo pranešimas apie pirmo VAS apdorojimo pabaigą,
   atejo_s2 – į agregatą atėjo pranešimas apie antro VAS apdorojimo pabaigą.
4. Vidinių įvykių aibė: $E'' = \{gen_par\}$
   gen_par – į agregate HUB sugeneruota nauja paraiška.
5. Diskrečioji agregato būsenos dedamoji: $v(t_m) = \{state1, state2\}$
   state1 – pirmas VAS perpildytas,
   state2 – antras VAS perpildytas.
6. Tolydžioji agregato būsenos dedamoji: $z_v(t_m) = \{par_gen\}$
   par_gen – paraiškos generavimo laikas.

7 Pradinė būsena: *state1=0, state2=0, gen_par = 0,1*
8 Perėjimo ir išėjimo operatoriai:
H(gen_par):

$$par_gen = random;$$

$$state1 = state1;$$

$$state2 = state2;$$

G(gen_par):

$$out1 = \begin{cases} 1, jei \quad state1 = 0 \\ 0, jei \quad state1 = 1 \end{cases};$$

$$out2 = \begin{cases} 1, jei \quad (state1 \neq 0) AND (state2 = 0) \\ 0, jei \quad (state2 = 1) OR (state1 = 0) \end{cases};$$

Agregatai VAS_1 ir VAS_2:

1 Įėjimo signalų aibė: $X = \{in\}$
2 Išėjimo signalų aibė: $Y = \{sign\}$
3 Išorinių įvykių aibė: $E' = \{atejo_par\}$,
atejo_par – į agregatą atėjo nauja paraiška,
4 Vidinių įvykių aibė: $E'' = \{apt_par\}$
apt_par – agregatas aptarnavo paraišką.
5 Diskrečioji agregato būsenos dedamoji: $v(t_m) = \{buf, buf_\max, dirba\}$
buf – paraiškų skaičius buferyje,
buf_max – maksimalus leidžiamas buferio užpildymas,
dirba – agregatas apdoroja paraiškas.
6 Tolydžioji agregato būsenos dedamoji: $z_v(t_m) = \{par_apt\}$
par_apt – apdorojama paraiška
7 Pradinė būsena: *buf=0, buf_max=3, dirba= 0, par_apt=∞*
8 Perėjimo ir išėjimo operatoriai:
H(apt_par):

$$par_apt = \begin{cases} random, jei \quad buf \neq 0 \\ \infty, jei \quad buf = 0 \end{cases};$$

$$dirba = \begin{cases} 1, jei \quad buf \neq 0 \\ 0, jei \quad buf = 0 \end{cases};$$

$$buf = \begin{cases} buf - 1, jei \quad buf \neq 0 \\ buf, jei \quad buf = 0 \end{cases};$$

G(apt_par):

$$sign = \begin{cases} 0, jei \quad buf \neq buf_\max \\ 1, jei \quad buf = buf_\max \end{cases};$$

H(atejo_par):

$$par_apt = \begin{cases} random, \quad jei \quad (buf = 0) \quad AND \quad (sign = 0) \quad AND \quad (dirba = 0) \quad AND \quad (in = 1) \\ par_apt, \quad jei \quad (buf \neq 0) \quad OR \quad (sign = 1) \quad OR \quad (dirba = 1) \quad OR \quad (in = 0) \end{cases};$$

$$buf = \begin{cases} buf + 1, \quad jei \quad (buf \neq buf_\max) \quad AND \quad (sign = 0) \quad AND \quad (dirba = 1) \quad AND \quad (in = 1) \\ buf, \quad jei \quad (buf = buf_\max) \quad OR \quad (sign = 1) \quad OR \quad (dirba = 0) \quad OR \quad (in = 0) \end{cases};$$

$$dirba = \begin{cases} dirba, \quad jei \quad in = 0 \\ 1, \quad jei \quad in = 1 \end{cases};$$

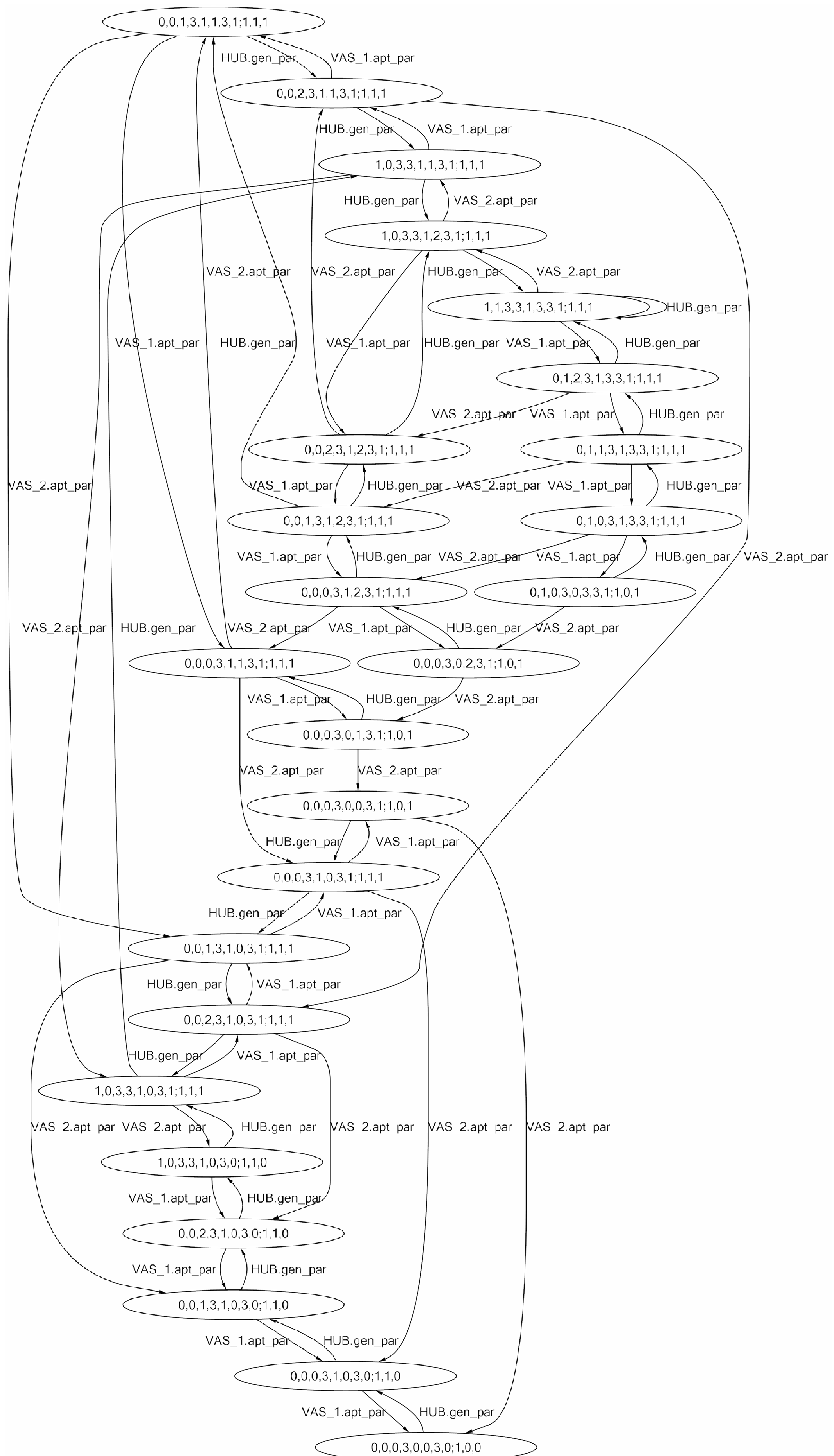
G(atejo_par):

$$sign = \begin{cases} 1, & jei \quad (sign = 0) \quad AND \quad (in = 1) \quad AND \quad (buf = buf_\max) \\ sign, & jei \quad (sign \neq 0) \quad OR \quad (in = 0) \quad OR \quad (buf \neq buf_\max) \end{cases};$$

Validuojant šią sistemą sugeneruojamas būsenų grafas, parodytas paveiksle 5.2.

Būsenų grafo viršūnėse yra sistemos būsenos, o grafo briaunos – tai įvykiai, kuriems įvykus įvyksta perėjimai tarp būsenų. Sistemoje galimi 3 skirtingi įvykiai – sugeneruota paraiška šakotuve (HUB.gen_par), arba apdorota paraiška vienoje iš aptarnavimo sistemų (VAS_1.apt_par ir VAS_2.apt_par). Sistemos būsenų grafe svarbiausios dvi koordinatės – eilių ilgiai vienkanalėse aptarnavimo sistemose (diskretūs kintamieji *buf*). Kiekvienoje grafo viršūnėje sugeneravus naują paraišką padidėja eilės ilgis vienoje iš aptarnavimo sistemų, todėl koordinatė padidėja vienetu. Atitinkamai jei viena iš aptarnavimo sistemų apdoroja užklausą, jos eilės ilgis sumažėja vienetu.

Iš sugeneruoto grafo galime vaizdžiai matyti, kaip keičiasi sistemos būsena įvykus vienam ar kitam galimam įvykiui. Deja, bet vizualus grafo atvaizdavimas galimas tik tuomet, kai nagrinėjamas mažas grafo viršūnių skaičius. Jei grafo viršūnių skaičius yra didelis, tuomet dažniausiai būsenų grafai dėl savo sudėtingumo būna sunkiai skaitomi ir suprantami.



Pav. 5.2 Pasiekiamų būsenų grafas

Atlikus būsenų grafo generavimą atliekamas jo validavimas. Sistemos specifikacija buvo aprašyta teisingai, todėl grafo validavimas neaptiko jokių pasiekiamų būsenų grafo klaidų.

Norint parodyti, kad sistema aptinka nekorektiškumus ir klaidas, pakeisime specifikaciją, tikslingai įrašydami klaidą. Pakeiskime agregato VAS_1 specifikaciją taip, kad jei agregato buferis persipildo, šis nesiunčia pranešimo agregatui HUB apie įvykusį perpildymą, ir tuo pačiu išjungia ir paraiškų apdorojimą. Tokiu atveju kiekviena nauja paraiška bus siunčiama į agregatą VAS_1, tačiau šis paraiškų neapdoros, tuo pačiu nesikeis ir buferyje esančių paraiškų skaičius. Po pakeitimo agregato VAS_1 perėjimo ir išėjimo operatoriai atrodys taip:

<u>*H(atejo_par):*</u>

$$par_apt = \begin{cases} \infty, & jei \quad buf = buf_\max \\ par_apt, & jei \quad (buf \neq 0) \quad OR \quad (sign = 1) \quad OR \quad (dirba = 1) \quad OR \quad (in = 0) \\ random, & jei \quad (buf = 0) \quad AND \quad (sign = 0) \quad AND \quad (dirba = 0) \quad AND \quad (in = 1) \end{cases};$$

$$buf = \begin{cases} buf + 1, & jei \quad (buf \neq buf_\max) \quad AND \quad (sign = 0) \quad AND \quad (dirba = 1) \quad AND \quad (in = 1) \\ buf, & jei \quad (buf = buf_\max) \quad OR \quad (sign = 1) \quad OR \quad (dirba = 0) \quad OR \quad (in = 0) \end{cases};$$

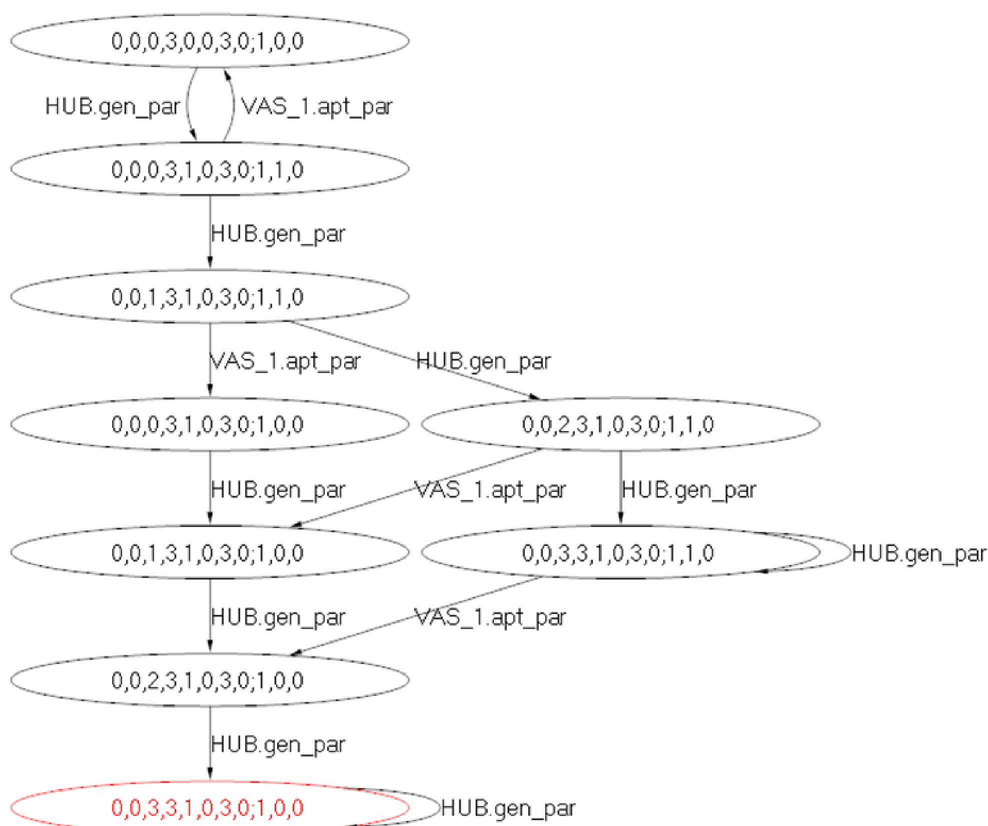
$$dirba = \begin{cases} dirba, & jei \quad in = 0 \\ 1, & jei \quad in = 1 \end{cases};$$

<u>*G(atejo_par):*</u>

$$sign = \begin{cases} sign, & jei \quad (sign = 0) \quad AND \quad (in = 1) \quad AND \quad (buf = buf_\max) \\ sign, & jei \quad (sign \neq 0) \quad OR \quad (in = 0) \quad OR \quad (buf \neq buf_\max) \end{cases};$$

Vykdome grafo validavimą su pakeista specifikacija. Šiuo atveju sugeneruotas pasiekiamų būsenų grafas parodytas paveiksle 5.3.

Šiuo atveju sugeneruotas būsenų grafas skiriasi nuo ankstesnio, pavaizduoto paveiksle 5.2. Iš šio grafo matyti, kad jei pirmos aptarnavimo sistemos eilės ilgis pasiekia 3 (maksimalus nustatytas buferio ilgis), vienintelis galimas įvykis sistemoje lieka naujos paraiškos generavimas. Šiuo atveju susidaro amžinas ciklas, nes paraiškų generatorius nuolat generuoja paraišką pirmai aptarnavimo sistemai, o ši neapdoroja paraiškos, tuo pačiu ir nepraneša generatoriui apie perpildytą buferį.



Pav. 5.3 Pasiekiamų būsenų grafas po specifikacijos pakeitimo

Šiuo atveju programa aptinka specifikacijos klaidą – aklavietę vienoje iš sugeneruoto grafo viršūnių. Aptikta klaida matoma programos lange, kuris parodytas paveiksle 5.4. Nors galinėje būsenoje nuolat kartojama ta pati operacija ir susidaro amžinas ciklas, grafo analizės požiūriu – tai aklavietė. Taip yra, todėl, kad ši būsena yra vienintelė grafo jungiosios komponentės viršūnė, kuri neturi išeinančių viršūnių, todėl atitinka 2.4.1 skyriuje aprašytas aklavietės savybes.

Id	Type	Vertices
1	Deadend	(0,0,3,3,1,0,3,0;1,0,0)

Pav. 5.4 Specifikacijos klaida, aptikta sistemos validavimo metu.

Surasta klaida atitinka specifikacijoje padarytą klaidą – specifikacija buvo pakeista taip, kad sistemos veikimo metu susidarytų aklavietė.

Abiem atvejais – tiek naudojant nuoseklų, tiek lygiagretų algoritmą, gaunami tokie patys pasiekiamų būsenų grafai. Šiais eksperimentais parodyta, kad sukurtas lygiagretus grafo generavimo algoritmas sėkmingai sugeneruoja pasiekiamų būsenų grafą. Taip pat pademonstruota, kad sistemoje sėkmingai realizuoti grafo analizės algoritmai – specifikacijoje palikus klaidą, validavimo metu ji buvo nesunkiai aptikta.

5.2. *Validavimo posistemės greitaveikos eksperimentai.*

Šių eksperimentų tikslas – nustatyti, ar sukurtas lygiagretus pasiekiamų būsenų grafo generavimo algoritmas veikia greičiau už nuosekliai veikiantį savo analogą. Taip pat siekiama

eksperimentiniu būdu išsiaiškinti, kelių gijų naudojimas duoda daugiausiai naudos generuojant pasiekiamų būsenų grafą.

Šiam eksperimentui naudota tokia pati sistemos specifikacija kaip ir 5.1. skyriuje vykdytiems veikimo eksperimentams, todėl pati specifikacija čia neaptariama.

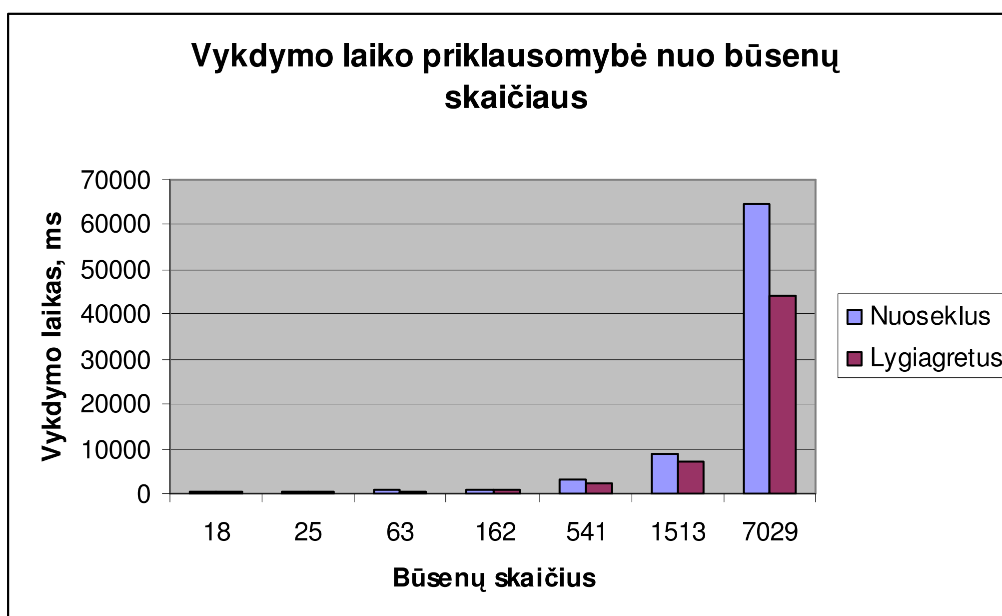
Keisdami keletą specifikacijos parametrų (VAS agregatų skaičių, buferių dydžius) gauname įvairių būsenų skaičių grafus, kuriems matuosime greitaveikos laikus.

5.2.1. Grafo generavimo trukmės priklausomybė nuo būsenų skaičiaus.

Išmatavus grafo generavimo trukmę naudojant abu realizuotus algoritmus nustatyta, kad nagrinėjamame pavyzdyje visais atvejais greičiau veikia lygiagretus algoritmas. Kai būsenų skaičius mažas (iki 100 būsenų), skaičiavimų laikas yra labai mažas, todėl veikimo laiko skirtumas irgi yra mažas, tačiau kai nagrinėjamas 7000 būsenų grafo sudarymas, lygiagretus algoritmas generavimą atlieka net 15 sekundžių greičiau. Lentelėje 5.1 pateikti eksperimentų metu gauti laikai naudojant nuoseklų ir lygiagretų algoritmus. Paveiksle 5.5 pateiktas skaičiavimo laiko grafikas, kuriame parodyta, kaip pasikeičia vykdymo laikas naudojant lygiagretų algoritmą vietoje nuoseklaus.

Lentelė 5.1 Grafo generavimo priklausomybė nuo būsenų skaičiaus duomenų lentelė

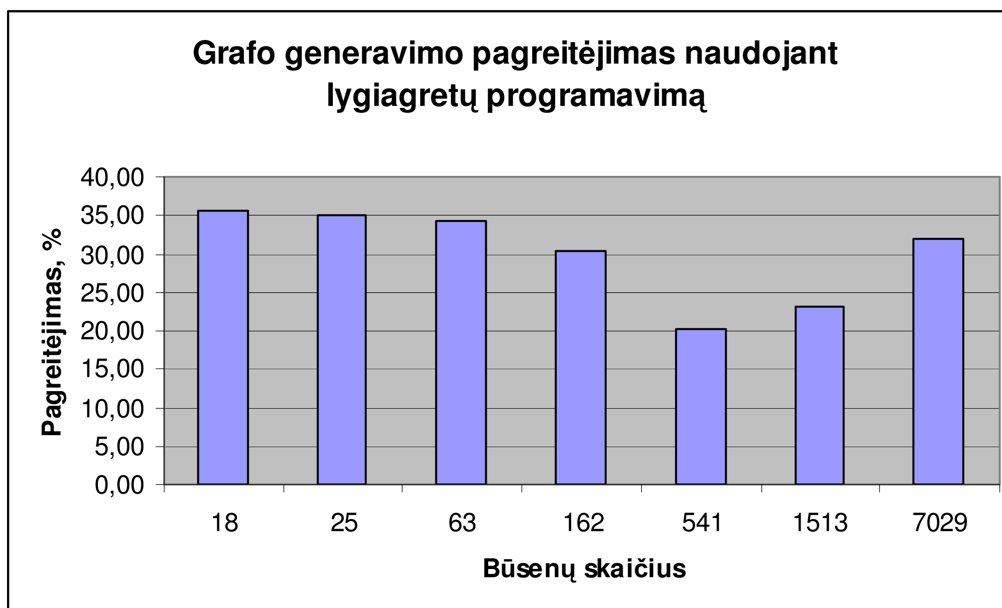
Būsenų skaičius	**18**	**25**	**63**	**162**	**541**	**1513**	**7029**
Nuoseklus, ms	512	570	702	992	3015	9123	64642
Lygiagretus, ms	330	370	461	690	2403	7007	44032



Pav. 5.5 Grafo generavimo trukmės priklausomybės nuo būsenų skaičiaus grafikas

5.2.2. Grafo generavimo pagreitėjimas naudojant lygiagretų programavimą

Atsižvelgus į anksčiau gautus rezultatus apskaičiuojamas skaičiavimo pagreitėjimas, kuris atsiranda nuoseklų grafo generavimo algoritmą pakeitus į lygiagretų. Paveiksle 5.6 parodytas vykdymo pagreitėjimas procentais, naudojant lygiagretų programavimą.



Pav. 5.6 Grafo generavimo pagreitėjimo grafikas

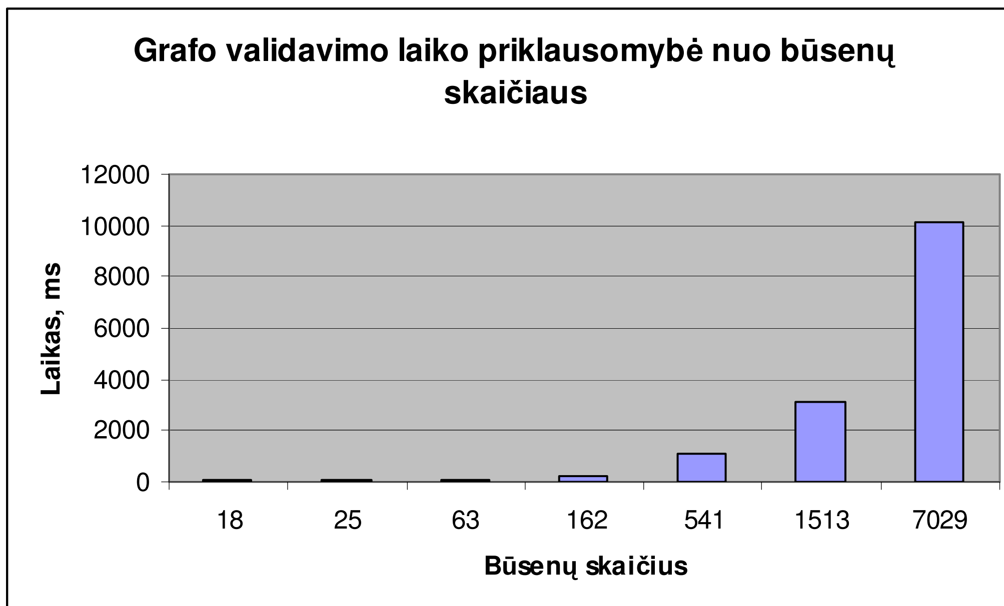
Eksperimento rezultatai parodė, kad nagrinėjamame pavyzdyje vykdymo laikas priklausomai nuo generuojamo grafo būsenų skaičiaus pagreitėja nuo 20 iki 35%.

5.2.3. Grafo validavimo laiko priklausomybė nuo būsenų skaičiaus

Grafo validavimui naudojamas tik vienas – nuoseklus algoritmas. 2.4. skyriuje buvo parodyta, kad šio algoritmo sudėtingumas yra tiesinis, todėl aišku, kodėl eksperimentuose grafo validavimo laikas yra daug mažesnis nei grafo generavimo laikas. Lentelėje 5.2 pateiktos nagrinėto uždavinio grafo validavimo laiko trukmės esant skirtingiems grafo būsenų skaičiams. Paveiksle 5.7 parodytas validavimo trukmės grafikas, esant skirtingiems būsenų skaičiams.

Lentelė 5.2 Grafo validavimo laiko priklausomybės nuo būsenų skaičiaus duomenų lentelė

Būsenų skaičius	**18**	**25**	**63**	**162**	**541**	**1513**	**7029**
Laikas, ms	62,5	70	101	230	1056	3099	10148



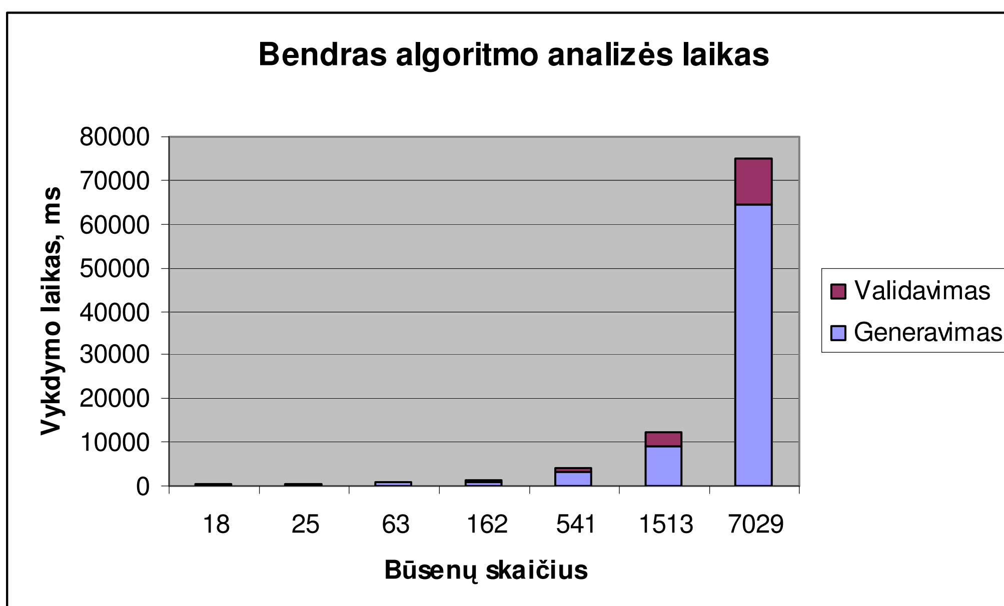
Pav. 5.7 Grafo validavimo laiko priklausomybės nuo būsenų skaičiaus grafikas

5.2.4. Grafo analizės laiko priklausomybė nuo būsenų skaičiaus

Grafo analizės uždavinys susideda iš dviejų dalių – grafo generavimo, bei grafo analizės. Eksperimento metu nustatyta, kad grafo generavimas užima daug daugiau laiko, nei grafo analizė. Lentelėje 5.3 parodyta nagrinėto uždavinio sprendimo – grafo generavimo ir verifikavimo – trukmė. Paveiksle 5.8 parodytas sprendimo trukmės grafikas.

Lentelė 5.3 Grafo analizės priklausomybės nuo būsenų skaičiaus duomenų lentelė

Būsenų skaičius	**18**	**25**	**63**	**162**	**541**	**1513**	**7029**
Grafo generavimas, ms	512	570	702	992	3015	9123	64642
Grafo validavimas, ms	62,5	70	101	230	1056	3099	10148



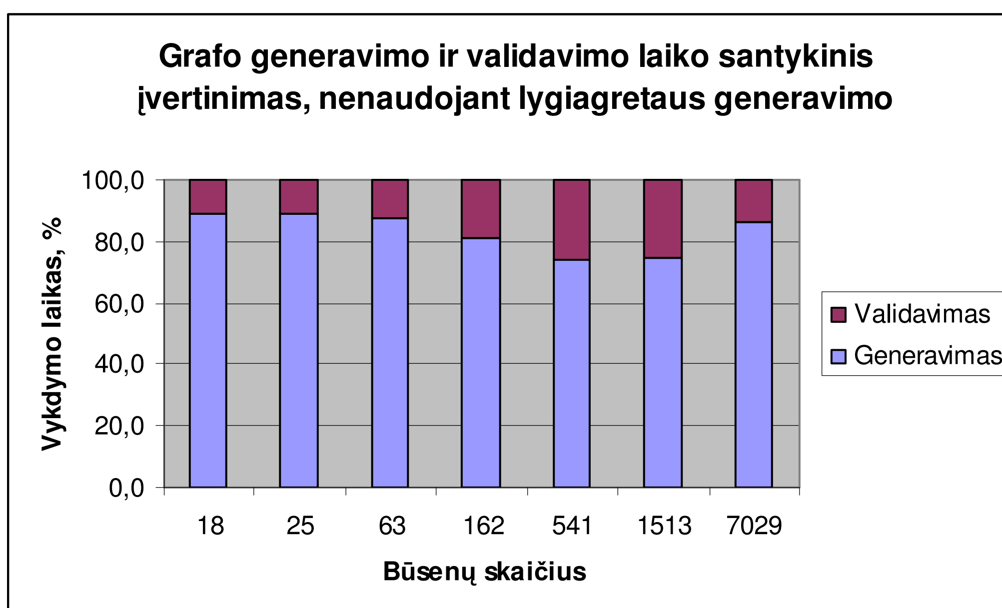
Pav. 5.8 Bendro analizės algoritmo vykdymo laiko grafikas

5.2.5. Santykinis grafo generavimo ir validavimo laiko įvertinimas

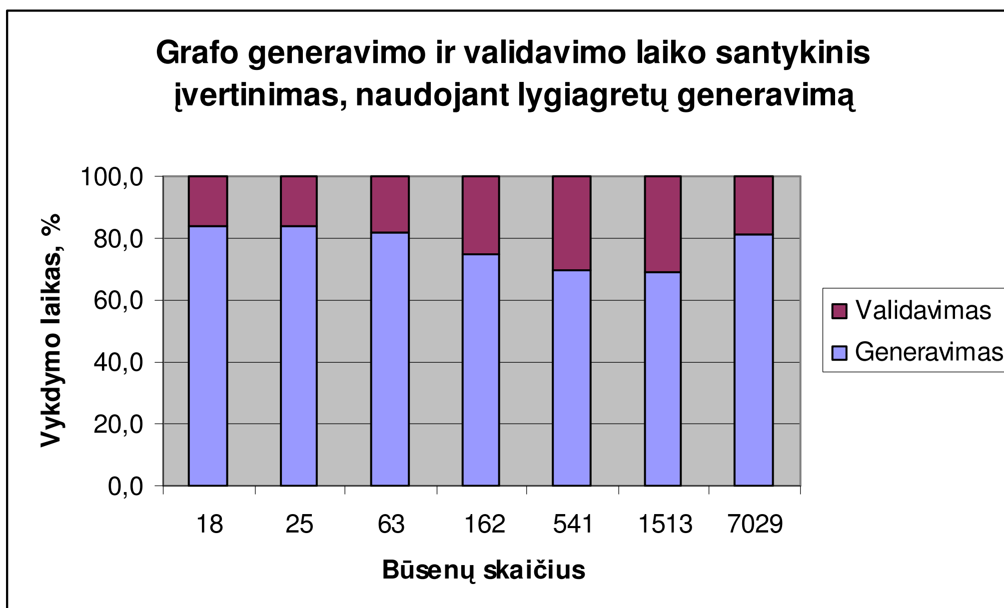
Šio eksperimento metu norėta išsiaiškinti, kiek procentų laiko iš viso analizės uždavinio užima grafo generavimas, o kiek validavimas. Įvertinimas atliekamas tiek naudojant nuoseklų grafo generavimo algoritmą, tiek lygiagretų. Lentelėje 5.4 parodyta, kokią analizės laiko dalį sudaro grafo generavimas ir validavimas, kai grafo generavimui naudojamas nuoseklus ir lygiagretus algoritmai. Paveiksle 5.9 parodytas santykinės generavimo ir validavimo trukmės grafikas, naudojant nuoseklų algoritmą. Paveiksle 5.10 parodytas santykinės generavimo ir validavimo trukmės grafikas, naudojant lygiagretų algoritmą.

Lentelė 5.4 Santykinio grafo generavimo ir validavimo laiko įvertinimo duomenų lentelė

Algoritmas	Būsenų skaičius	18	25	63	162	541	1513	7029
Nuoseklus	Generavimas, %	89,1	89,1	87,4	81,2	74,1	74,6	86,4
	Validavimas, %	10,9	10,9	12,6	18,8	25,9	25,4	13,6
Lygiagretus	Generavimas, %	84,1	84,1	82,0	75,0	69,5	69,3	81,3
	Validavimas, %	15,9	15,9	18,0	25,0	30,5	30,7	18,7



Pav. 5.9 Nuoseklaus grafo generavimo ir validavimo laiko santykio grafikas



Pav. 5.10 Lygiagretaus grafo generavimo ir validavimo laiko santykio grafikas

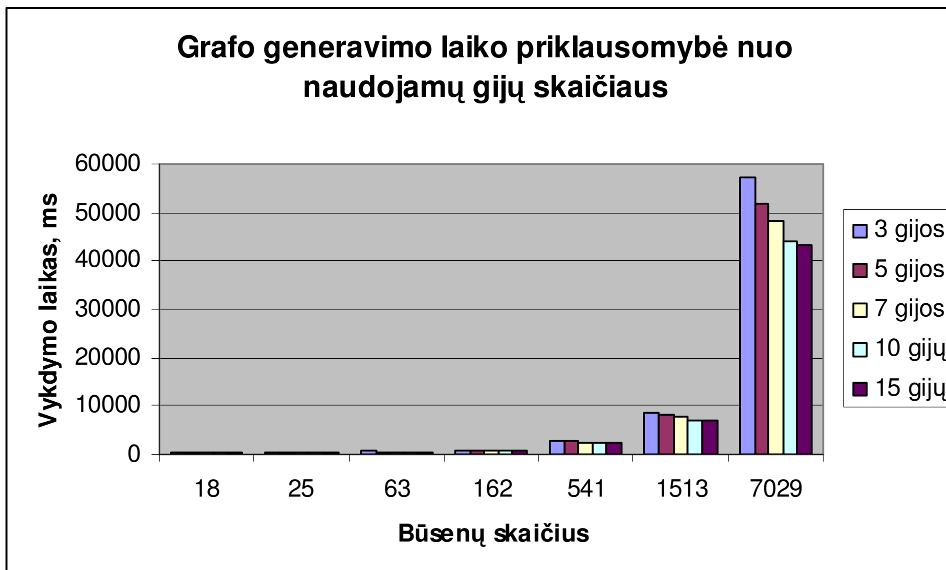
Kaip matoma iš eksperimento rezultatų, prieš realizuojant lygiagretų grafo generavimo algoritmą grafo generavimas sudarydavo beveik iki 90% viso algoritmo vykdymo laiko. Panaudojus lygiagretų skaičiavimo algoritmą grafo generavimas sudaro daugiausiai 84,1%, o esant dideliems būsenų skaičiams – dar mažiau.

5.2.6. Generavimo laiko priklausomybė nuo naudojamų gijų skaičiaus.

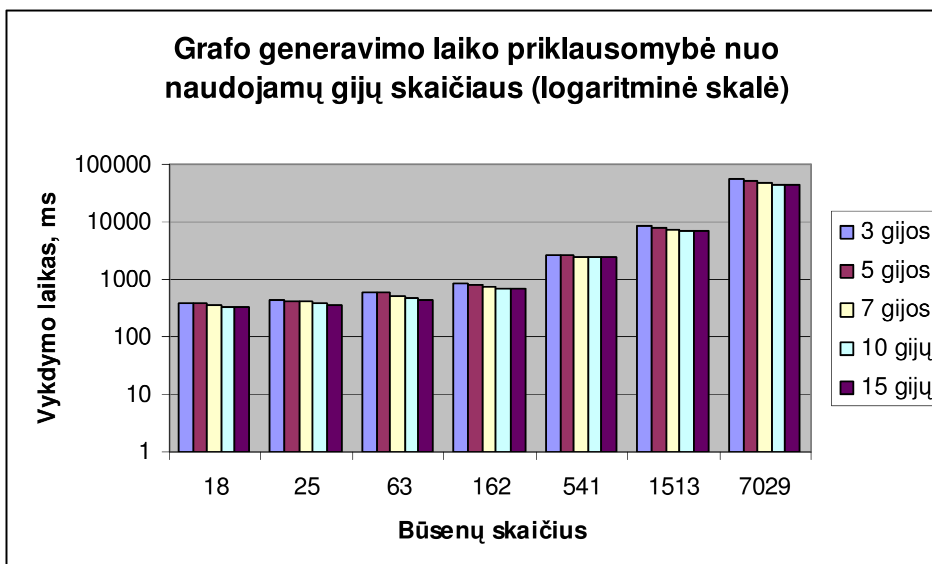
Šio eksperimento tikslas yra nustatyti, kaip naudojamų lygiagrečiame algoritme gijų skaičius įtakoja algoritmo greitaveiką. Buvo atlikti eksperimentai su 3 – 15 gijų, bei nustatyta, kuriuo atveju gaunami geriausi rezultatai. Lentelėje 5.5 pateikta generavimo laiko trukmė esant įvairiam būsenų skaičiui bei naudojamam gijų skaičiui. Paveiksle 5.11 pateiktas šio eksperimento rezultatų grafikas, o paveiksle 5.12 – tas pats grafikas logaritminėje skalėje.

Lentelė 5.5 Generavimo laiko priklausomybės nuo gijų skaičiaus duomenų lentelė

Būsenų skaičius	18	25	63	162	541	1513	7029
3 gijos	390	440	612	832	2612	8688	57109
5 gijos	375	424	577	789	2556	8111	51970
7 gijos	349	399	522	747	2492	7643	48478
10 gijų	330	370	461	690	2403	7007	44032
15 gijų	321	361	447	675	2377	6949	43320



Pav. 5.11 Generavimo laiko priklausomybės nuo gijų skaičiaus grafikas



Pav. 5.12 Generavimo laiko priklausomybės nuo gijų skaičiaus logaritminis grafikas

Eksperimentų metu nustatyta, kad didinant gijų skaičių algoritmo vykdymo laikas sumažėja. Nagrinėjamame uždavinyje didinant naudojamų gijų skaičių pagreitėjimas buvo nežymus, todėl tikėtina, kad ir didesnis gijų skaičiaus didinimas neduotų didesnio generavimo spartos padidėjimo.

5.3. Greitaveikos eksperimentų rezultatai

Greitaveikos eksperimentų metu buvo nagrinėjama, kaip pasikeičia sistemos pasiekiamų būsenų grafo generavimo laikas, analizuojant dviejų vienkanalių aptarnavimo sistemų su paraiškų generatoriumi sistemą. Eksperimentų metu nustatyta, kad naudojant lygiagretų programavimą nagrinėjamą uždavinį galima išspręsti 20-35% greičiau. Veikimo pagreitėjimas mažai priklauso nuo naudojamų gijų skaičiaus – net 15 gijų šio uždavinio neišsprendžia daug greičiau nei 3 gijos.

6. Išvados

Validavimo posistemė yra sukurtos formalių specifikacijų analizės (FSA) sistemos dalis. Formalių PLA specifikacijų įvedimui sukurtas interaktyvus grafinis redaktorius, imitaciniam modeliavimui – imitavimo posistemė, pažingsniniam sistemos imitavimui – trasavimo posistemė, o specifikacijų teisingumo tikrinimui – validavimo posistemė.

Sukurta validavimo posistemė realizuoja pasiekiamų būsenų metodą, kurio pagalba galima tikrinti agregatinės specifikacijos saugumo bei gyvybingumo savybes. Pasiekiamų būsenų metodo realizavimui buvo sukurtas pasiekiamų būsenų grafo generavimo algoritmas, realizuoti grafo analizės algoritmai, bei panaudotas grafo vizualaus vaizdavimo komponentas.

Pasiekiamų būsenų grafo analizės algoritmas tikrina sistemos saugumo ir gyvybingumo charakteristikas – ieško grafo aklavietčių, ciklų, nepasiekiamų būsenų, tikrina koordinačių apribojimus bei atlieka invarianto tikrinimą.

Sistema ištestuota atliekant vienkanalės aptarnavimo sistemos formalios specifikacijos validavimą. Ištirta pavyzdinė specifikacija ir patikrintas jos teisingumas leido įsitikinti, kad grafo validavimas atliekamas teisingai.

Pasiekiamų būsenų grafo generavimui sukurtas lygiagretaus generavimo algoritmas, kurio dėka grafo viršūnių generavimą galima vykdyti naudojant skirtingas gijas. Atliktų eksperimentų su vienkanale aptarnavimo sistema metu nustatyta, kad lygiagretaus algoritmo naudojimas šio pavyzdžio validavimui leidžia pagreitinti pasiekiamų būsenų grafo analizę iki 35%. Eksperimentų metu taip pat nustatyta, kad sistemos veikimo sparta mažai priklauso nuo grafo generavimui naudojamų lygiagrečių gijų skaičiaus.

7. Santrumpų ir terminų žodynas

DFS (**D**epth **F**irst **S**earch) – grafo, medžio ar hierarchinės struktūros viršūnių aplankymo algoritmas.

Stipriai susijusi komponentė – tai orientuoto grafo viršūnių poaibis, kuriame tarp bet kurių dviejų viršūnių egzistuoja kelias.

Stekas – abstrakti duomenų struktūra, kuri remiasi LIFO (*angl.* **L**ast **I**n, **F**irst **O**ut) principu.

Eilė – abstrakti duomenų struktūra, kuri remiasi FIFO (angl. **F**irst **I**n, **F**irst **O**ut) principu.

Lygiagretus programavimas – viena iš programavimo krypčių

PLA *(**P**iece **L**inear **A**ggregate)* – atkarpomis tiesinis agregatas.

FSA (**F**ormal **S**pecification **A**nalysis) – formalių specifikacijų analizė.

DCSC (**D**ivide-and-**C**onquer **S**trongly **C**onnected Component) – lygiagretus stipriai susijusių komponenčių orientuotame grafe radimo algoritmas

SPIN - įrankis, skirtas analizuoti loginį paskirstytų sistemų nuoseklumą.

XML – praplečiama žymėjimo kalba (e**X**tensible **M**arkup **L**anguage).

8. Literatūros sąrašas

[1] Pranevičius, H. *Kompiuterių tinklų protokolų formalusis specifikavimas ir analizė: agregatinis metodas.* Monografija. Kaunas, 2005.

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

[3] Holzmann, G. J. *Design and validation of computer protocols.* Prentice Hall, 1990.

[4] Holzman, G. J. Algorithms for automated protocol validation. *AT&T Technical Journal*, Vol.69, No.2, 1988.

[5] Lluch-Lafuente, A.; Leue, S.; Edelkamp, S. *Partial order reduction in directed model checking*. In SPIN Workshop on Model Checking Software, Lecture Notes in Computer Science 2318, Springer, 2002.

[6] Tiwari, A.; Rueß, H.; Saįdi, H.; Shankar, N. *A Technique for Invariant Generation*. Springer–Verlag, Genova, Italy, 2001.

[7] Aho, A.; Hopcroft, J.; Ullman, J. The *Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[8] Tarjan, R. *Depth-first search and linear graph algorithms.* SIAM Journal on Computing. Vol. 1, 1972

[9] Cheriyan, J.; Mehlhorn, K. *Algorithms for dense graphs and networks on the random access computer*, Algorithmica, volume 15, 1996

[10] Fleisher, L.; Hendrickson, B.; Pinar, A. *On Identifying Strongly Connected Components in Parallel*, 1999

[11] *Formal Systems Verification LAB* [interaktyvus] [žiūrėta 2008-05-07]. Prieiga per internetą <http://fsv.dimi.uniud.it/software>.

[12] *Carnegie Mellon Software Engineering Institute* [interaktyvus] [žiūrėta 2008-05-07]. Prieiga per internetą <http://www.sei.cmu.edu/vtu/tools.html>.

[13] „ON-THE-FLY, LTL MODEL CHECKING with SPIN“ [Žiūrėta 2008-04-12]. Prieiga per internetą <http://spinroot.com/spin/whatispin.html>.

[14] The PROMELA Language [Žiūrėta 2008-04-16]. Prieiga per internetą

<http://www.dai-arc.polito.it/dai-arc/manual/tools/jcat/main/node168.html>

[15] *VIS (Verification Interacting with Synthesis)* [interaktyvus] [žiūrėta 2008-04-27]. Prieiga per internetą <http://vlsi.colorado.edu/~vis/>.

[16] *SMV* [interaktyvus] [žiūrėta 2008-04-27]. Prieiga per internetą <http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/>.

[17] *NuSMV* [interaktyvus] [žiūrėta 2008-05-07]. Prieiga per internetą <http://nusmv.irst.itc.it/NuSMV/index.html>.

[18] *Uppaal* [interaktyvus] [žiūrėta 2008-05-12]. Prieiga per internetą <http://www.uppaal.com/>.

[19] Alzbutas R., Janilionis V. *Dynamic systems simulation using APL2*

[20] Pranevičius, H.; Pranevičienė, I.; Benkunskis, R. *Formalization and Simulation Telecommunication Protocols and Systems using PLA Method*. Kaunas: Technologija, 2005. – Nr. 4(60).

[21] Lamport, L.; *Specifying Systems The TLA+ Language and Tools for Hardware and Software Engineers,* Addison-Wesley, 2003

[22] Grama, A.; Gupta, A.; Vipin, K.; Karyapis, G; *Introduction to Parallel Computing – Design and Analysis of Algorithms,* Pearson Education, 2003.

[23] T. Ball, B. Cook, V. Levin, S. K. Rajamani *SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft* 2004

[24] Gao, Q.; Groz, R.; Bochmann, G. V.; Dargham, J.; Htite, E.H.; *Validation of Distributed Algorithms and Protocols,* 1995

[25] Holzmann, G.; Peled, D.; Yannakakis, M. *On Nested Depth First Search*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 00, 1996

9. Priedai

9.1. PLA formalizavimo kalbos aprašymas

Aprašinėjant sistemą *PLA* kalba, pirmiausiai sistema suskaidoma į atkarpomis tiesinius agregatus. Atkarpomis tiesiniai agregatai priklauso automatų modelių klasei. Kaip ir automatas, atkarpomis tiesinis agregatas aprašomas nurodant būsenų aibė *Z*, įėjimo signalų aibė *Y* bei perėjimo operatorių (atvaizdavimą) *H* ir išėjimo operatorių *G*, tačiau agregatas turi nemažai ypatybių, skiriančių šį modelį nuo automatinių modelių.

Agregato funkcionavimas stebimas laiko momentu $t \in T$ aibėje, t.y. agregato būsena $z \in Z$ yra laiko funkcija $z(t)$. Atkarpomis tiesinio agregato būsenos struktūra yra tokia pat kaip ir atkarpomis tiesinio Markovo proceso, t.y.

$$z(t) = (v(t), z_v(t));$$

čia $v(t)$ - diskrečioji būsenos dedamoji,

$z_v(t)$ - tolydi būsenos dedamoji.

Bendru atveju

$$v(t) = \{v_1(t), v_2(t), ..., v_m(t)\},\ z_v(t) = \{z_{v1}(t), z_{v2}(t), ..., z_{vk}(t)\};$$

čia $v_i(t)$ - i-oji diskrečios dedamosios koordinatė,

$z_{vi}(t)$ - i-oji tolydžiosios dedamosios koordinatė.

Kai nėra įėjimo signalų, agregato būsena kinta taip:

$$v(t) = const,\ \frac{dz_v(t)}{dt} = -\alpha_v;$$

čia $\alpha_v = (\alpha_{v1}, \alpha_{v2}, ..., \alpha_{vk})$ - pastovusis vektorius.

Agregato būsena gali pakisti tik dviem atvejais: kai į agregatą siunčiamas įėjimo signalas arba kai viena iš tolydžiųjų dedamosios koordinačių įgyja tam tikrą reikšmę.[19][20]

Aprašinėjant agregatus kompiuteryje, naudojama *PLA-CA* kalba. *PLA-CA* – tai *PLA* formalizavimo kalbos versija, pritaikyta kompiuteriams. Kiekvienas agregatas aprašomas atskirai pagal tokius požymius:

1. Įėjimų signalų aibė *X*;
2. Išėjimo signalų aibė *Y*;

3. Išorinių įvykių aibė E';
4. Vidinių įvykių aibė E'';
5. Valdymo sekos;
6. Diskrečiosios agregato būsenų dedamosios;
7. Tolydžiosios agregato būsenų dedamosios;
8. Pradinė agregato būsena;
9. Perėjimų iš išėjimų operatoriai.[1].