KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Edgaras Augus

Lygiagrečių programų interaktyvaus vizualizavimo programinė įranga

Magistro darbas

Vadovas
doc. dr. R. Marcinkevičius

KAUNAS, 2011

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

PROGRAMŲ INŽINERIJOS KATEDRA

Edgaras Augus

Lygiagrečių programų interaktyvaus vizualizavimo programinė įranga

Magistro darbas

Recenzentas

doc. dr. S. Maciulevičius

2011 05 30

Vadovas

doc. dr. R. Marcinkevičius

2011 05 30

Atliko

IFM 9/2 grupės studentas

Edgaras Augus

2011 05 30

Kaunas, 2011

SUMMARY. This work presents design, features and assessment of the system, which was implemented during master project. The analytical part of this document firstly describes criteria to evaluate systems that visualize parallel programs later on alternative systems are evaluated in accordance with established criteria, further problems associated with such system realization are raised, and also other authors work, proposals and recommendations are discussed. Design part of this work contains requirement part and solutions related to the technical realization. In the part of study data for experimental part is prepared, which will be used for created system‘s assessment, it provides standard concurrent programming problems and their solutions. Experimental part uses data prepared in study part and evaluates the use of implemented master project while visualizing parallel programs, as well as how its functionality helps to detect errors in parallel programs and other adverse situations, and their origins.

Turinys

Paveikslėliai

Lentelės

1. ĮVADAS

1.1. Santrauka

Šame darbe pateiksime magistrinio darbo projekto metu sukurtos sistemos realizacijos ypatumus bei jos vertinimą. Analitinėje dalyje apibrėžiami vertinimo kriterijai, skirti vertinti sistemas, kurios vizualizuoja lygiagrečias programas, aptariamos rastos alternatyvos, jos įvertinamos pagal nustatytus kriterijus, taip pat iškeliamos problemos, susijusios su tokių sistemų realizacija, aptariami kitų autorių darbai bei pasiūlymai. Projektinėje dalyje pateikiami surinkti reikalavimai kuriamai sistemai, bei sprendimai, susiję su technine jos realizacija. Tyrimo dalyje parengiami duomenys, kurie bus naudojami eksperimento metu, tiriant magistrinio darbo projekto metu sukurtą sistemą - aptariamos tipinės lygiagretaus programavimo problemos, bei jų sprendimai. Eksperimentinėje dalyje, panaudojant tyrimo dalyje parengtus duomenis, įvertinsime sukurtos sistemos tinkamumą lygiagrečių programų vizualizacijai, bei kaip jos funkcionalumas padeda aptikti lygiagrečių programų klaidas ir kitas nepageidaujamas situacijas bei jų šaltinius.

2. ANALITINĖ DALIS

2.1. Įžanga

Vizualus informacijos pateikimas dažnai yra vienas geriausių būdų padėti suprasti žmogui naujus dalykus ar idėjas [1, 3, 5, 7, 8]. Informacinių technologijų pasaulyje nuo pat pradžių informaciją siekiama vizualizuoti, kaip pavyzdžius galima pateikti operacines sistemas ar televiziją – operacinių sistemų didžioji dauguma turi grafinę sąsają; radiją pakeitė televizija.

Lygiagrečių programų vizualizavimas taip pat yra svarbus [14, 15], nes jis padėtų komercinėms įmonėms greičiau analizuoti kaip veikia jų sukurtas produktas, įvertinti ar jis veikia taip kaip tikėtasi bei aptikti klaidas ar ištirti algoritmo našumą (angl. performance) [5, 6]. Toks įrankis taip pat padėtų studentams lengviau suvokti kaip veikia lygiagretūs procesai ir su kokiomis problemomis susiduria juos kuriantys programuotojai, kaip veikia programinės priemonės tokios kaip semaforai bei monitoriai.

Šis įrankis turi ilgalaikę perspektyvą, kadangi žmonių naudojamos sistemos plečiasi, apdorojami vis didesni informacijos kiekiai, reikalingi vis sudėtingesni skaičiavimai, o rezultatų gavimo pagreitinimas yra realizuojamas lygiagrečiai paskirstant skaičiavimus (stambūs www portalai veikia keliuose serveriuose, GRID sistemos ir pan.).

2.2. Tikslas

Projekto tikslas yra sukurti sistemą, kuri analizuotų lygiagrečios programos kodą (realizuotos C++ kalba, lygiagrečių gijų kūrimui panaudojant OpenMP biblioteką), jį vykdytų ir atvaizduotų jo veikimą vartotojui patogia forma. Sistema turėtų palaikyti programuotojams reikalingas funkcijas, kurios padėtų kurti ir tobulinti jau sukurtas lygiagrečias programas - palengvintų programuotojo darbą. Ne paslaptis, jog programavimo procesą sudaro dvi pagrindinės dalys: programos kodo rinkimas ir jo derinimas (angl. „debugging“). Norint, jog sistema palengvintų programuotojo darbą, ją kuriant reikia atsižvelgti į abi minėtas programavimo proceso dalis, jas tobulinti, kiek įmanoma automatizuoti, tai galima pasiekti:

- sukūrus patogų programos kodo redaktorių
- įdiegiant įvairias derinimo metu reikalingas funkcijas, tokias kaip:
  - kintamųjų reikšmių stebėjimas realiu laiku
  - kodo optimizacijos pasiūlymai
  - automatinis aklaviečių aptikimas

2.3. Lygiagrečių programų stebėjimo įrankių vertinimo kriterijai

Tam, kad palyginti kuriamą sistemą su egzistuojančiomis alternatyvomis ar pačia alternatyvas tarpusavyje reikia iškelti ir apibrėžti kriterijus, kuriais vadovaujantis jas vertinsime. Kiti autoriai [20, 21] savo darbuose taip pat nagrinėja įrankius, skirtus darbui su lygiagrečiomis programomis, jie siūlo tokius vertinimo kriterijus: programos stebėjimas realiu laiku, panaudojimo paprastumas, galimybė stebėti atskiras gijas, kovos dėl resursų aptikimas, aklavietės aptikimas, užsiciklinimų aptikimas, rezultatų pernešamumas, konfigūravimo paprastumas, rezultatų išsamumas, nepriklausomumas nuo aplinkos, C/C++ programų palaikymas, vartotojo nurodytų įvykių tyrimas, kelių bandymų palyginimas ir t.t. Tačiau ne visi kriterijai yra mums tinkami, kadangi minėtų autorių ir mūsų sistemos tikslai yra skirtingi – vieno autoriaus [20] darbe dėmesys skiriamas gijų ir procesų komunikavimui bei

lygiagrečios programos efektyvumui tirti. Mūsų atveju tikslinga būtų išskirti šiuos programinės įrangos vertinimo kriterijus:

- **Kintamųjų reikšmių stebėjimas** – galimybė stebėti kintamųjų reikšmes, esančias gijose, kaip jos kinta programos veikimo metu.
- **Programos vykdymo stebėjimas realiu laiku** – galimybė stebėti programos veikimą, kol dar ji nepabaigė darbo, tai yra svarbu, kadangi programa gali turėti klaidų ir jos darbas gali niekada nesibaigti, pvz. dėl amžinų ciklų.
- **Programos kodo redaktorius –** sistemos dalis, programuotojui suteikianti galimybę rinkti programos kodą, jį saugoti, keisti ir t.t. Šiuolaikiniai programų kodo redaktoriai turi daugybę funkcijų, palengvinančių programos kodo rinkimo (programavimo) procesą, tokių kaip automatinis užpildymas(angl. „auto complete“), raktinių žodžių žymėjimas, sintaksės klaidų aptikimas.
- **Interaktyvumas** – galimybė vartotojui įsiterpti į vizualizacijos procesą, t.y. vykdyti tokias komandas kaip vizualizacijos stabdymas, tęsimas (po sustabdymo), vizualizacijos nutraukimas (lygiagrečios programos darbo nutraukimas), vizualizacijos greičio reguliavimas ir t.t.
- **Galimybė stebėti atskiras gijas** – kriterijus nurodo, ar yra galimybė vartotojui stebėti kiekvienos gijos darbą atskirai nuo kitų.
- **Rezultatų saugojimas** – ar yra galimybė gautus rezultatus išsaugoti.
- **Kelių bandymų palyginimas** – šis kriterijus nurodo ar yra galimybė palyginti kelis tos pačios programos vykdymus, jei programos veikimas buvo stebėtas daugiau nei vieną kartą.
- **Profiliavimas** – informacijos apie gijų veikimą rinkimas, kaip kad: kurioje kodo eilutėje gijos veikimas ilgiausiai užtrunka, kurią kodo eilutę gija dažniausiai vykdo ir pan.
- **Aklaviečių aptikimas** – aklavietė - tai būsena, kai dvi gijos laukia viena kitos atliekant kokį nors veiksmą, kadangi abi gijos yra laukimo rėžime jokie veiksmai nevykdomi. Kriterijus išpildomas, kai įrankis automatiškai aptinka aklavietes.
- **Begalinių ciklų aptikimas** – kriterijus nurodo ar vartotojui yra pateikiama informacija apie potencialius begalinius ciklus ir ar jie aptinkami lygiagrečios programos vykdymo metu.

2.4. Alternatyvos

1. UPPAAL

Uppaal yra integruota priemonė modeliavimo, imitavimo ir tikrinimo realiuoju laiku sistemoms. Tipinės taikymo sritys apima realaus laiko valdiklius ir komunikacijos protokolus ypač tuos, kur laiko aspektai yra labai svarbūs. Ši sistema leidžia vartotojui sukurti imitacinį modelį realaus laiko sistemoms, kuriose yra neišvengiami lygiagretūs procesai, ji atlieka modelio patikrinimą, atranda modelio būsenų pasiekiamumo problemas, bei suteikia galimybė simuliuoti jo veikimą, stebėti modelio būsenų kitimą[10]. UPPAAL sistemoje modelio elgsena aprašoma JAVA programavimo kalba, tam tikslui ji turi integruotą bei patogų šios kalbos redaktorių, simuliacijos rezultatus galima saugoti vaizdiniais (pvz. JPEG) ir tekstiniais(pvz. XML) formatais.

2. Convit

Suomijos studentų darbas, skirtas atvaizduoti tam tikrų problemų, susijusių su lygiagrečiaisiais procesais, plačiai taikomus sprendimus. Šis projektas palaiko specializuotą programavimo kalbą, kuri yra paremta „Ada“ programavimo kalba, šis įrankis realizuoja tokius lygiagretaus programavimo problemų sprendimus kaip: Dekker‘io algoritmas, Patterson‘o algoritmas, gamintojo-vartotojo problema ir t.t. [11]. Sistema leidžia stebėti tik dvi gijas, veikiančias vienu metu, programos kodo redaktorius šioje sistemoje yra skurdokas, kita vertus, jis buvo kurtas specializuotai programavimo kalbai, kurios galimybės yra labai ribotos. Šis įrankis leidžia stebėti kintamųjų reikšmes programos vykdymo metu, bet palaikomi tik penki kintamųjų tipai, tačiau minėtus trūkumus kompensuoja interaktyvumo komandos, kurios leidžia stabdyti, tęsti, vykdyti pažingsniui lygiagrečią programą, taip pat leidžiama išsisaugoti visą gijų veikimo seką, su jose buvusia informacija.

3. PolkaW

Polka yra bendrosios paskirties animacinė sistema, kuri tinka konstruoti algoritmus ir programų animacijas. Ji buvo pradėta kurti kaip sistema, kuri padėtų kurti animaciją lygiagrečioms programoms ir skaičiavimams. Ši sistema yra paremta objektinio programavimo principais, kurie suteikia vartotojui galimybę atvaizduoti lygiagrečių programų veikimą 2D (dviejų dimensijų) vaizdu, vėliau ši sistema buvo išplėsta į Polka3D, kuri leidžia kurti tokius 3D grafikos primityvus kaip kūgis ar sfera [12]. Kadangi ši sistema buvo kurta kaip bendrosios paskirties sistema, skirta lygiagrečių programų vizualizacijai, ji nepasižymi itin plačiu funkcijų spektru, o koncentruojasi į programos veikimo atvaizdavimą grafiškai, tačiau tikslai, ties kuriais dirbo autoriai yra palyginti gerai išpildyti: šis įrankis palaiko kintamųjų reikšmių stebėjimą, realiu laiku grafiškai atvaizduoja neribotą kiekį veikiančių gijų, darbo rezultatus galima išsisaugoti.

4. Gthread

Šis projektas turi rinkinį priemonių, kurios yra skirtos atvaizduoti lygiagretiems procesams. Animacija vaizduoja atskiras gijas ir jų judėjimą per programos funkcijas, taip pat yra atvaizduojami tokie programos priedai kaip semaforai. Šios sistemos veikimas paremtas gijų veikimo ir komunikavimo stebėjimu ir surašymu į žurnalą, kurio duomenys vėliau bus atvaizduojami vartotojui [13].

5. Intel Parallel Studio

http://software.intel.com/en-us/articles/intel-parallel-studio-home/ (žiūrėta 2011)

Šis įrankis yra Microsoft Visual Studio papildinys (angl. „add-on“), jis siūlo platų spektrą įvairiausių funkcijų, kurios yra reikalingos programuotojams, kuriantiems lygiagrečias programas, tai būtų: profiliavimas(angl. „profiling“), programos veikimo stebėjimas realiu laiku, gijų veikimo analizė, aklaviečių aptikimas, begalinių ciklų aptikimas ir t.t., tačiau jis neturi savo programos kodo redaktoriaus, todėl norint jį naudoti būtina įsigyti ir Microsoft produktą, įrankis labiau yra panašus į derintuvę (angl. „debugger“), todėl čia nerasime tokios gijų veikimo animacijos, kokią turi PolkaW įrankis.

2.5. Alternatyvų palyginimas

Šioje dalyje atliksime sukurtų alternatyvų palyginimą, pagal jau aptartus kriterijus. Lentelėje „Alternatyvų palyginimas“ pateikiamos apžvelgtos sistemos, bei kaip jos išpildo nustatytus kriterijus. Kriterijų išpildymas bus vertinamas penkiabalėje sistemoje, nuo 1 iki 5. Įvertinimas „1“ reiškia, jog kriterijus prastai išpildomas, o 5 – kriterijus puikiai išpildomas Ženklu „-“ žymėsime, jei sistema neturi tokio funkcionalumo, kuris atitinka kriterijaus apibrėžimą.

1 lentelė. Alternatyvų palyginimas

	UPPAAL	Convit	PokaW	Gthread	Intel Parallel Studio
Kintamųjų reikšmių stebėjimas	5	3	4	3	5
Programos vykdymo stebėjimas realiu laiku	5	5	5	-	5
Programos kodo redaktorius	5	2	3	3	-
Interaktyvumas	-	4	-	-	5
Galimybė stebėti atskiras gijas	-	2	5	4	4
Rezultatų saugojimas	5	5	5	5	5
Kelių bandymų palyginimas	-	-	-	-	5
Profiliavimas	4	-	-	-	5
Aklaviečių aptikimas	3	-	-	-	4
Begalinių ciklų aptikimas	4	-	-	-	4

Apibendrinant alternatyvų įvertinimą pagal kriterijų sąrašą galime teigti, jog geriausiai kriterijus išpildo „Intel Parallel Studio“ įrankis, kuris visuose kriterijuose lenkia arba yra lygus kitoms alternatyvoms, tačiau tai dar nereiškia, jog minėtas įrankis yra geriausias šioje srityje, kadangi gali būti pasirinkta daugiau vertinimo kriterijų, kuriuos kiti alternatyvūs įrankiai išpildytų geriau.

2.6. Probleminė sritis ir siūlomi sprendimai

Su lygiagrečių programų vizualizacijos poreikiu yra susiduriama nuo pat lygiagretaus programavimo atsiradimo, dažniausi klausimai, su kuriais susiduriama, kuriant tokias sistemas, yra kaip organizuoti pačios sistemos darbą (įvykių sekimo, būsenų pasikeitimo stebėjimas, programos kodo analizavimas) bei kaip patraukliai ir suprantamai atvaizduoti lygiagrečios programos procesų veikimą ekrane vartotojui [8]. Šiame skyriuje aptarsime keletą, kitų autorių siūlomų sprendimų, pradedant klausimu kokią architektūrą pasirinkti kuriant vizualizacijos įrankį, kaip operuoti duomenimis ir baigiant animacijos koncepcija.

VISTOP pasiūlytas sprendimas į kokias atsakingas dalis padalinti sistemą.

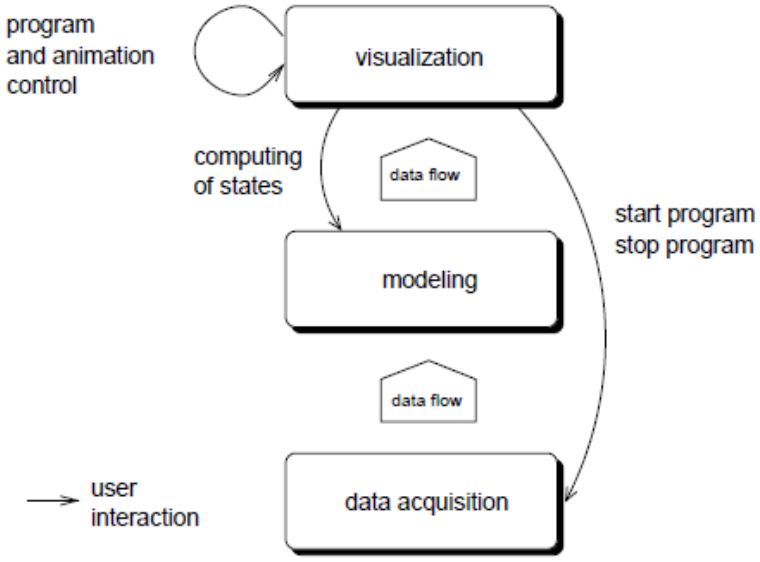
Šioje dalyje aptarsime VISTOP (angl. „Visualization Tool for Parallel systems) pasiūlytą architektūrinį sprendimą [1].

Architektūrinio pasiūlymo esmė: trys pagrindinės sistemos dalys (1 pav.):

1. Duomenų (programos kodo) įsisavinimas (angl. data acquisition), ši dalis būtų atsakinga už svarbių įvykių suradimą programos kode ir jų perdavimą kitai daliai – modeliavimo sluoksniui (angl. modeling layer).

2. Modeliavimo sluoksnis (angl. modeling layer). Jis skaičiuotų globalias būsenas iš įvykių srauto, kurį jam siųstų duomenų įsisavinimo sluoksnis. Ši dali taip pat yra atsakinga už programinį modelį.

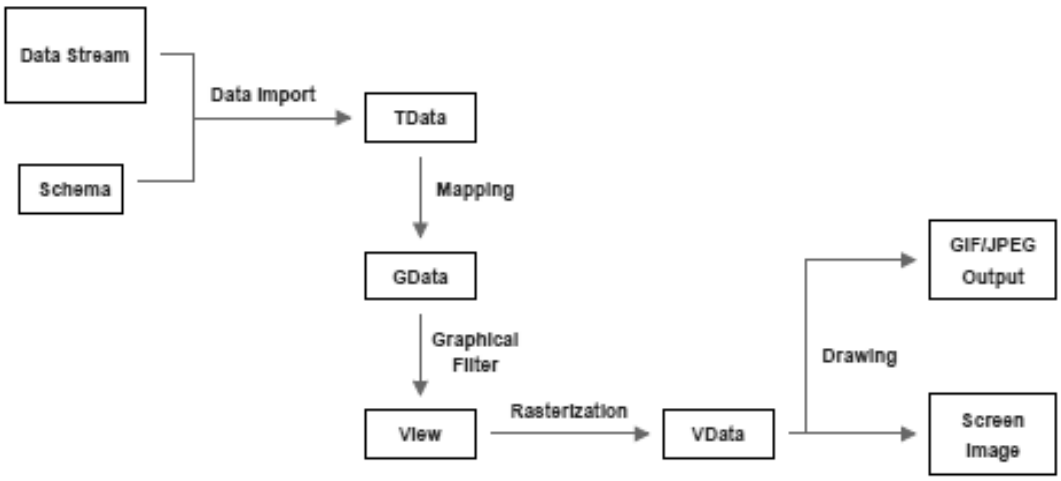
3. Vizualizacijos sluoksnis (angl. visualization layer). Vizualizuoja būsenas, apskaičiuotas modelio, jis taip pat kontroliuoja kitas dvi sistemos dalis.



1 pav. VISTOP pasiūlyta schema [1]

Penkių pakopų vizualizavimo procesas

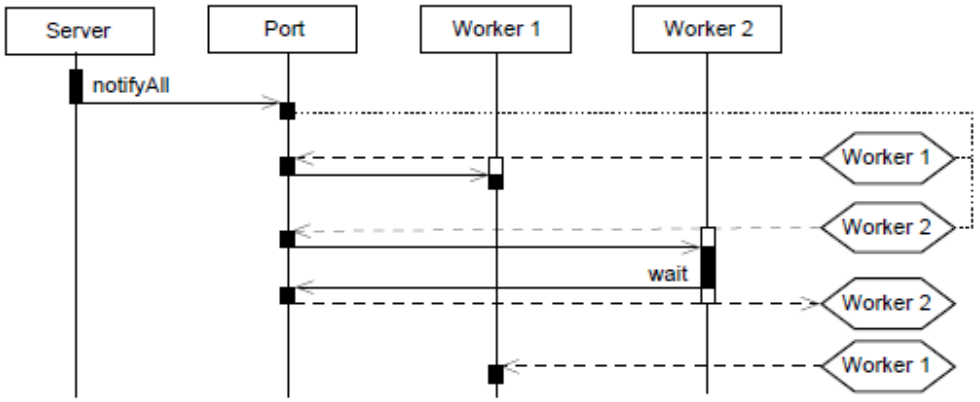
Žvelgiant giliau, į sistemos realizaciją, taip kai kurie autoriai [6] siūlo kaip programiškai galima būtų realizuoti lygiagrečių programų vizualizavimo sistemą (2 pav.). Pačioje pradžioje duomenų srautas (data stream) yra sudarytas iš dvejetainių arba ASCII tipo įrašų. TData objektas sukomponuoja duomenų schemą ir duomenų srauto įrašus. TData objektas (įrašas) yra sujungiamas su GData objektu, kuris turi grafinius atributus, tokius kaip koordinatės, spalva, figūra, pasukimo kampas ir pan. Kitas žingsnis yra GData objektą paversti į pikselinį atvaizdą (pixel image), kurs schemoje figūruoja kaip VData objektas. Paskutinis žingsnis – VData objektų atvaizdavimas ekrane.



2 pav. Penkių pakopų vizualizavimas [6]

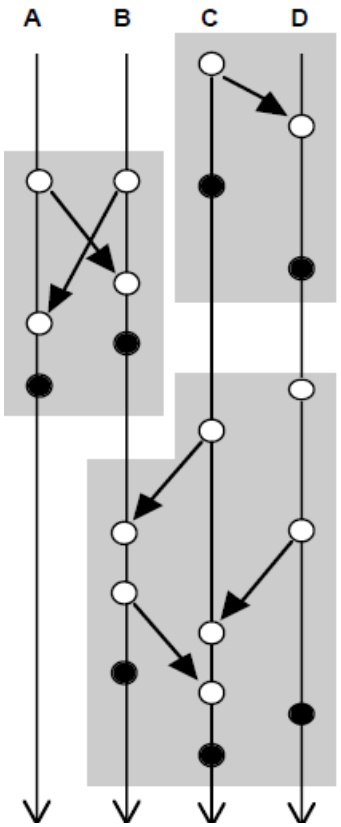
Vizualizacijos sprendimas UML diagramomis.

Kuriant tokias sistemas dažnai susiduriama su problema kaip intuityviai ir informatyviai atvaizduoti lygiagrečių procesų ar gijų veikimą. Šiai problemai spręsti galima pasirinkti UML diagramomis paremtą animaciją, nes šios diagramos yra pripažintos ir plačiai naudojamos IT sektoriuje, taip pat žmogui jos yra lengvai suprantamos[7]. Tačiau jos negali pilnai atvaizduoti svarbių procesų, vykstančių lygiagrečiose programose (pvz. kritinių sekcijų), todėl idėjos autoriai praplėtė UML tiek, kad būtų galima atvaizduoti lygiagrečius įvykius sekų diagramomis (3 pav.).



3 pav. Atvaizdavimas UML Diagramomis [7]

Dar vienas, dažnai kylantis klausimas, yra kaip sutalpinti pakankamai didelį kiekį informacijos, kurį grąžina veikianti lygiagreti programa[16]. Ji kyla iš to, kad vykdant lygiagrečią programą yra sukuriamos kelios gijos(ar procesai), kurių kiekvieno savybės(kintamieji, kodo eilutė, gijų komunikacija ir pan.) bet kuriuo laiko momentu gali kisti ir minėtų savybių kitimas yra svarbus, kadangi būtent pagal juos galime spręsti kaip veikia programa: kokios operacijos buvo atliktos kiekvienoje gijoje, kokia buvo konkrečios gijos būsena tam tikru laiko momentu (dirbanti, laukianti kol atsilaisvins kritinė sekcija ir pan.), kokia buvo gijų vykdymo seka, kuri gija įtakojo bendrų kintamųjų pasikeitimą ir t.t. Paveikslėlyje nr.4 matome autorių D.Hart'o, E.Kraemer'io ir G-C.Roman'o pasiūlymą kaip galima atvaizduoti gijų tarpusavio komunikaciją: veikia keturios gijos (A, B, C, D), pilkame fone išskirtos trys transakcijos, rodyklėmis nurodyta komunikacijos kryptis. Tie patys autoriai, remdamiesi jau atliktais savo darbais, sistemos darbą siūlo atvaizduoti trimačiame modelyje, jie siūlo išskirti šias ašis: laiko(X ašis), gijų (Y ašis) ir gijos duomenų konkrečiu laiko momentu (Z ašis).



4 pav. Gijų komunikacijų atvaizdavimas

Taip pat susiduriama su problema kaip geriau rinkti informaciją apie lygiagrečios programos veikimą. Visi siūlymai susiveda į du bendrus: būsenos stebėjimas tam tikrais lako

momentais(angl. „sampling“), informacijos rinkimas visu programos veikimo metu [18] bei modeliavimas [19]. Pirmasis būdas yra naudingas tuo, kad programos veikimo stebėjimas nereikalauja tiek daug kompiuterio resursų kaip antrasis, tačiau jis neužtikrina, kad bus surinkta visa dominanti informacija, nes tarp informacijos rinkimo laiko intervalų gali įvykti ne viena svarbi stebėtojui operacija. Pasiūlymas rinkti informaciją visu jos veikimo metu yra patrauklus tuo, kad galime kaupti visą mus dominančią informaciją, kaip kad kokiu laiku, kažkokia gija buvo tam tikroje eilutėje ir kokios buvo lokalių bei globalių kintamųjų reikšmes tuo laiko momentu, tačiau toks stebėjimas yra iš dalies kišimasis į lygiagrečios programos darbą, kuris gali „iškreipti“ jos veikimą, be to toks metodas kainuoja daugiau kompiuterio resursų(atmintis, procesoriaus apkrovimas), nei pirmasis. Trečiasis būdas (modeliavimas) apima lygiagrečios programos kodo analizavimą ir jo veikimo imitavimą. Tačiau šis būdas yra labai sudėtingas, kadangi jam realizuoti reikia puikiai išmanyti konkrečios programavimo kalbos veikimą, procesoriaus darbą, gijų darbo paskirstymo logiką ir t.t. Šis metodas niekada neatspindės realaus veikimo, trumpa jo esmė: išanalizuojamas lygiagrečios programos kodas, sukuriamas jos modelis ir pagal modelį imituojamas programos veikimas, t.y. kaip ji turėtų veikti, o pati programa nėra kompiliuojama ar paleidžiama.

Grįžtant prie klausimo kaip visgi tinkamai ir patraukliai atvaizduoti lygiagrečios programos veikimą turime nustatyti kokiai auditorijai bus skirtas vizualizavimo įrankis: ar tai bus patyrę programuotojai, kuriems svarbios visos techninės detales ar tik lygiagretųjį programavimą mokytis pradedantys asmenys, nes tuomet bus aiškesnė įrankio paskirtis: ar tai bus mokymosi, ar darbo įrankis [17].

2.7. Išvados

Kaip matome buvo rastos ir įvertintos pakankamai įvairios kuriamos sistemos alternatyvos, taip pat aptarti galimi techniniai sistemos įgyvendinimo sprendimai, tą įvairovę lemia tai, jog su lygiagrečių programų vizualizacijos poreikiu susiduria ne viena suinteresuotų asmenų grupė (programuotojai, studentai ir t.t.).

Galime daryti prielaidą, jog sukurti, visų poreikius tenkinančią, universalią lygiagrečių programų vizualizavimo programinę įrangą nėra taip paprasta, todėl šioje srityje yra kur tobulėti.

3. PROJEKTINĖ DALIS

3.1. Reikalavimų specifikacija

3.1.1. Sistemos paskirtis

Vizualus informacijos pateikimas dažnai yra vienas geriausių būdų padėti suprasti žmogui naujus dalykus ar idėjas. Informacinių technologijų pasaulyje nuo pat pradžių informaciją siekiama vizualizuoti, kaip pavyzdžius galima pateikti operacines sistemas ar televiziją – operacinių sistemų didžioji dauguma turi grafinę sąsają; radiją pakeitė televizija.

Taigi lygiagrečių programų vizualizavimas taip pat yra svarbus, nes jis padėtų studentams lengviau suvokti kaip veikia lygiagretūs procesai ir su kokiomis problemomis susiduria juos kuriantys programuotojai, kaip veikia programinės priemonės tokios kaip semaforai bei monitoriai. Toks įrankis taip pat padėtų komercinėms įmonėms greičiau analizuoti kaip veikia jų sukurtas produktas, įvertinti ar jis veikia taip kaip tikėtasi bei aptikti klaidas.

3.1.2. Projekto kūrimo pagrindas (pagrindimas)

Šis įrankis turi ilgalaikę perspektyvą, kadangi žmonių naudojamos sistemos plečiasi, apdorojami vis didesni informacijos kiekiai, reikalingi vis sudėtingesni skaičiavimai, o rezultatų gavimo pagreitinimas yra realizuojamas lygiagrečiai paskirstant skaičiavimus (stambūs www portalai veikia keliuose serveriuose, GRID sistemos ir pan.).

3.1.3. Sistemos tikslai (paskirtis)

Projekto tikslas yra sukurti sistemą, kuri analizuotų lygiagrečios programos kodą, jį vykdytų ir atvaizduotų jo veikimą vartotojui patogia forma.

3.1.4. Užsakovai, pirkėjai ir kiti sistema suinteresuoti asmenys

Būtina turėti informaciją apie šiuos asmenis, kad išvengti skaudžių netikėtumų.

1. Užsakovas. Tai Kauno Technologijos Universitetas, programų inžinerijos katedra yra įsikūrusi adresu Studentų gatvė 50-415a, LT-51368 Kaunas. Šios įstaigos veikla yra mokymas ir moksliniai tyrimai.
2. Pirkėjas. Užsakovo atstovas yra Romas Marcinkevičius, Kauno technologijos universiteto informatikos fakulteto programų inžinerijos katedros darbuotojas (magistrinio darbo vadovas).
3. Kiti sprendimus priimantys asmenys. Projekto vadovas yra dr. doc. Romas Marcinkevičius. Už projekto vykdymo priežiūrą yra atsakingi dėstytojai Kęstutis Motiejūnas bei Virginija Limanauskienė. Projekto vykdytojas yra magistrantas Edgaras Augus. Sukurtas produktas gali būti platinamas mokymo tikslams, moksliniams tyrimams. Sukurta sistema skirta panaudoti kaip prototipas, kuriant tobulesnę, galutinę, daugiau funkcijų atliekančią sistemą.

3.1.5. Vartotojai

Sistemos plėtimo vartotojų lentelė

2 lentelė. Sistemos plėtimo vartotojų lentelė

Vartotojo kategorija:	Universitetai, mokslinės grupės, programinės įrangos projektavimo ir kūrimo įmonės
Vartotojo sprendžiami uždaviniai:	Sistemos prototipo tobulinimas, sistemos realizavimas
Patirtis dalykinėje srityje:	Patyręs
Patirtis informacinėse technologijose:	Patyręs
Prioritetas	Svarbus vartotojas

Sistemos vartotojas – studentas

3 lentelė. Sistemos vartotojas – studentas

Vartotojo kategorija:	Mokinys
Vartotojo sprendžiami uždaviniai:	Lygiagrečių programų specifikos įsisavinimas
Patirtis dalykinėje srityje:	Naujokas arba patyręs
Patirtis informacinėse technologijose:	Naujokas arba patyręs
Prioritetas	Antraeilis vartotojas

Sistemos vartotojas – komercinės įmonės darbuotojas

4 lentelė. Sistemos vartotojas – komercinės įmonės darbuotojas

Vartotojo kategorija:	Inžinierius, programuotojas
Vartotojo sprendžiami uždaviniai:	Programavimas, sistemų derinimas
Patirtis dalykinėje srityje:	Patyręs
Patirtis informacinėse technologijose:	Patyręs
Prioritetas	Svarbus vartotojas

3.1.6. Apribojimai sprendimui

Pagrindinis sistemos apribojimas yra operacinė sistema – sistema privalės dirbti Linux operacinės sistemos aplinkoje, taigi visi pasiūlymai, dėl kurių reikėtų keisti nustatytąją operacinę sistemą privalo būti atmesti.

3.1.7. Diegimo aplinka

Operacinė sistema: Linux

Įvesties įrenginiai: pelė ir klaviatūra

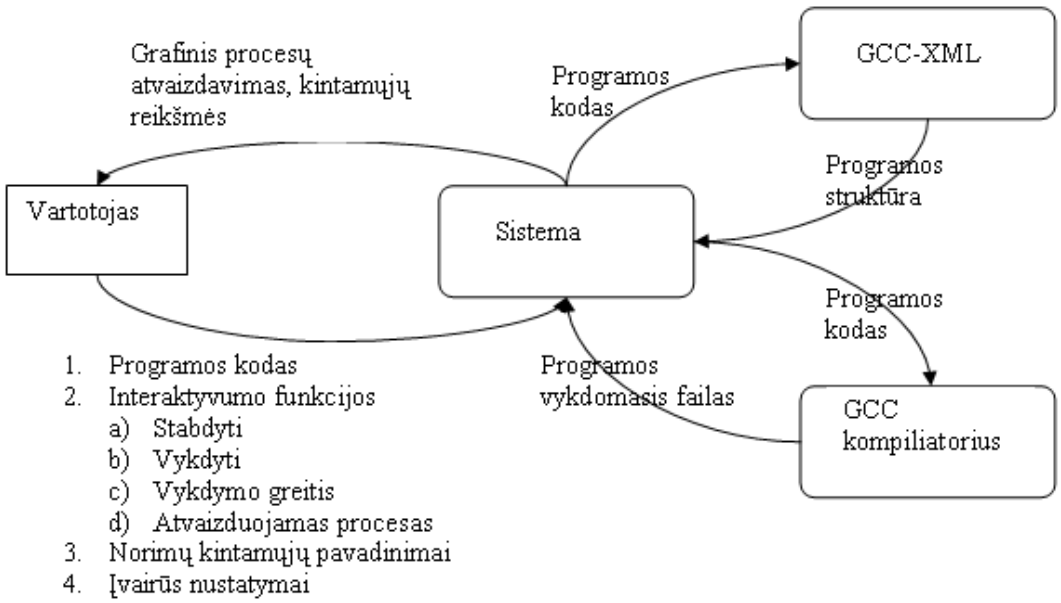
Išvesties įrenginiai: monitorius

3.1.8. Bendradarbiaujančios sistemos

- „Boost“ biblioteka (http://www.boost.org (žiūrėta 2010))
- C++ kalbos kompiliatorius (http://gcc.gnu.org/ (žiūrėta 2010))
- GCC-XML - C++ kalbos nagrinėtojas (angl. „parser“) (http://www.gccxml.org/ (žiūrėta 2010))
- OpenMP – lygiagrečių programų kūrimo sąsaja (http://openmp.org (žiūrėta 2010))
- GDB – C++ kalbos derintuvė (angl. „debugger“)

3.2. Funkciniai reikalavimai

3.2.1. Veiklos kontekstas (pateikiama konteksto diagrama)



5 pav. Konteksto diagrama

3.2.2. Veiklos padalinimas

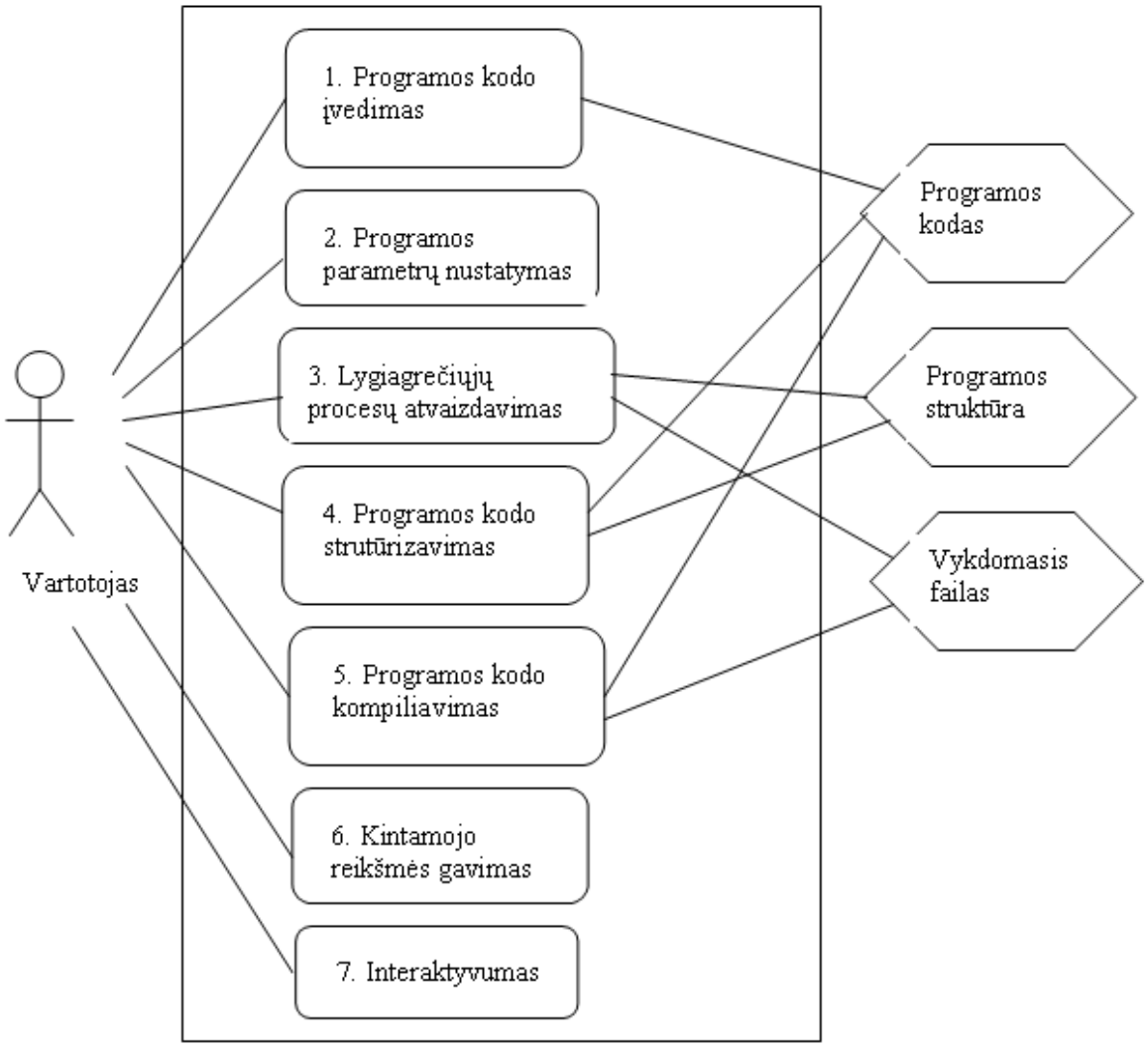
Sudaromas veiklos įvykių sąrašas, kuris apima visus veiklos įvykius, už kuriuos yra atsakinga nagrinėjama veikla. Veiklos įvykiai - tai vartotojo išskiriami veiksmai, atliekami veiklos metu. Reakcija (atsakymas) į kiekvieną įvykį atvaizduoja veiklos dalį, įeinančią į bendras veiklą sudarančias funkcijas.

Įvykių sąrašą sudaro: 1) Įvykio pavadinimas; 2) Įeinantys ir išeinantys informacijos srautai, kurie “lydi” įvykį. veiklos padalinimo paskirtis - identifikuoti veiklos “gabaliukus” , kurių pagrindu būtų galima nustatyti reikalavimus. Veiklos įvykių pagrindu toliau galima remtis atliekant sistemos detalią analizę ir projektavimą.

5 lentelė. Veiklos įvykių sąrašas

Eil.nr.	Įvykio pavadinimas	Įeinantys/išeinantys informacijos srautai
1	Vartotojas įveda programos kodą	Programos kodas(in)
2	Vartotojas kviečia programos paleidimo funkciją	Paleidimo trigeris(in)
3	GCC kompiliatorius kompiliuoja programos kodą	Programos kodas(out) Kompiliavimo rezultatas(in)
4	GCC-XML analizatorius analizuoja programos kodo struktūrą	Programos kodas(out) Programos struktūra(in)
5	Vartotojas vykdo interaktyvumo funkcijas (atvaizdavimo greitis, stabdyti ir t.t.)	Funkcijos paleidimo trigeris (in)
6	Vartotojas derina sistemos nustatymus	Programos nustatymai(in)
7	Sistema atvaizduoja programos darbą	Grafinis dinamiškas atvaizdavimas(out)
8	Kintamojo reikšmės gavimas	Kintamojo pavadinimas(in) Kintamojo reikšmė(out)

3.3. Sistemos ribos



6 pav. Sistemos ribos

3.3.1. Panaudojimo atvejų sąrašas

- Programos kodo įvedimas
- Programos parametrų nustatymas
- Lygiagrečiųjų procesų atvaizdavimas
- Programos kodo struktūrizavimas
- Programos kodo kompiliavimas
- Kintamojo reikšmės gavimas
- Interaktyvumas

Panaudojimo atvejų aprašai:

1. Programos kodo įvedimas

Tikslas: programos kodo įvedimas reikalingas tam, kad turėtume ką atvaizduoti.

Aktoriai: vartotojas, sistema.

Ryšiai su kitais PA: vartotojo lygiagrečiųjų procesų atvaizdavimas.

Sužadinimo sąlyga: iš vartoto sąsajos meniu punkto „įvesti programos kodą“ pasirinkimas ir programos kodo įvedimas teksto redaktoriaus pagalba.

Po-sąlyga: programos kodas įvestas ir išsaugotas.

2. Programos parametrų nustatymas

Tikslas: Vizualizavimo moduliui suteikti tam tikrą informaciją apie įvestą programą (semaforai, monitoriai, būsenos ir pan.), bei nurodyti vizualizavimo parametrus (pvz. procesų veikimo greitį).

Aktoriai: vartotojas, sistema.

Prieš-sąlygos: Įvestas ir išsaugotas programos tekstas

Sužadinimo sąlyga: iš vartoto sąsajos meniu punkto „nustatyti parametrus“ pasirinkimas

Po-sąlyga: parametrai įvesti ir išsaugoti.

3. Lygiagrečiųjų procesų atvaizdavimas

Tikslas: Atvaizduoti lygiagrečios programos procesų veikimą.

Aktoriai: vartotojas, sistema.

Prieš-sąlygos: Įvestas programos kodas, nustatyti programos veikimo parametrai.

Sužadinimo sąlyga: iš vartoto sąsajos meniu punkto „pradėti vizualizaciją“ pasirinkimas.

Po-sąlyga: Grafiškai atvaizduojamas programos veikimas.

4. Programos kodo struktūrizavimas

Tikslas: Pagaminti XML dokumentą, kuris aprašytų įvestos programos struktūrą

Aktoriai: sistema.

Prieš-sąlygos: Įvestas ir išsaugotas programos kodas.

Sužadinimo sąlyga: iš vartoto sąsajos meniu punkto „pradėti vizualizaciją“ pasirinkimas.

Po-sąlyga: Gautas XML dokumentas su programos struktūra.

5. Programos kodo kompiliavimas

Tikslas: Pagaminti vykdomąjį failą

Aktoriai: sistema.

Prieš-sąlygos: Įvestas ir išsaugotas programos kodas.

Sužadinimo sąlyga: iš vartoto sąsajos meniu punkto „pradėti vizualizaciją“ pasirinkimas.

Po-sąlyga: Gautas vykdomasis failas.

6. Kintamojo reikšmės gavimas

Tikslas: Parodyti vartotojui tam tikro kintamojo reikšmę.

Aktoriai: vartotojas, sistema.

Prieš-sąlygos: Vykdoma programos vizualizacija.

Sužadinimo sąlyga: iš vartoto sąsajos meniu punkto „parodyti kintamojo reikšmę“ pasirinkimas.

Po-sąlyga: Ekrane atvaizduojama kintamojo reikšmė.

7. Interaktyvumas

Tikslas: Valdyti vizualizacijos procesą.

Aktoriai: vartotojas, sistema.

Prieš-sąlygos: Vykdoma programos vizualizacija.

Sužadinimo sąlyga: iš vartoto sąsajos meniu punkto pasirinkti komandą (stabdyti, lėčiau, greičiau, ir pan.).

Po-sąlyga: Įvykdoma interaktyvumo komanda.

3.4. Funkciniai reikalavimai ir reikalavimai duomenims

3.4.1. Funkciniai reikalavimai

Reikalavimas Nr.		Reikalavimo tipas	9.1	Įvykis/PA NR.	5
Aprašymas	Programos kodo kompiliavimas yra negali būti nutrauktas				
Pagrindimas	Kompiliavimas yra sudėtingas procesas, todėl reikalinga, kad kompiliatorius pats baigtų kompiliavimą, nors ir programos kode yra klaidų				
Šaltinis	Programuotojas				
Tinkamumo kriterijus	Neįgyvendinti kompiliavimo stabdymo vartotojo meniu				
Užsakovo patenkinimas	0		Užsakovo netenkinimas		4
Priklausomybės	Nėra		Konfliktai		Nėra
Papildoma medžiaga	Nėra				
Istorija	2010 kovo 1d.				

Reikalavimas Nr.		Reikalavimo tipas	9.1	Įvykis/PA NR.	6

Aprašymas	Vizualizacija gali būti nutraukta bet kuriuo metu		
Pagrindimas	Programos vykdymas gali tęstis labai ar net amžinai ilgai dėl jos veikimo scenarijaus arba dėl programoje įsivėlusios klaidos		
Šaltinis	Užsakovas		
Tinkamumo kriterijus	Realizuoti stabdymo funkciją vartotojo meniu		
Užsakovo patenkinimas	3	Užsakovo netenkinimas	5
Priklausomybės	Nėra	Konfliktai	Nėra
Papildoma medžiaga	Nėra		
Istorija	2010 kovo 1d.		

Reikalavimas Nr.		Reikalavimo tipas	9.1	Įvykis/PA NR.	1
Aprašymas	Programos kodo įkėlimas iš failo ir išsaugojimas jame				
Pagrindimas	Kiekvieną kartą įvedinėti programos kodą ar jį kopijuoti į programos tekstinį redaktorių vartotoją veiktų neigiamai				
Šaltinis	Užsakovas				
Tinkamumo kriterijus	Realizuoti saugojimo/įkėlimo funkciją vartotojo meniu				
Užsakovo patenkinimas	3		Užsakovo netenkinimas		5
Priklausomybės	Nėra		Konfliktai		Nėra
Papildoma medžiaga	Nėra				
Istorija	2010 kovo 1d.				

3.5. Nefunkciniai reikalavimai

Nusako sistemos savybes, kuriomis ji turi pasižymėti. Tai kokybinės funkciniuose reikalavimuose numatytų funkcijų vykdymo charakteristikos.

3.5.1. Reikalavimai sistemos išvaizdai

Bendri reikalavimai vartotojo sąsajai.

- Intuityvi, lengvai skaitoma sąsaja;
- paprastas (nesudėtingas) panaudojimas;
- spalvotas ir patrauklus lygiagrečių procesų atvaizdavimas

3.5.2. Reikalavimai panaudojamumui

Panaudojimo paprastumas (lengvumas), kuris gali būti vertinamas konkrečiais kriterijais.

- Greitas reagavimas į interaktyvumo komandas
- paprastas naudotis inžinieriams – vizualizavimas vykdomas UML sekų diagramomis
- paprastai panaudojamas bet kokio asmens be apsimokymo

3.5.3. Reikalavimai vykdymo charakteristikoms

Interaktyvumo komandos turi būti vykdomos nedelsiant (įvykdoma tuojau pat po to, kai bet kuris procesas baigia vykdyti kodo eilutę).

3.6. Architektūros specifikacija

3.6.1. Dokumento paskirtis

Dokumentas pateikia išsamų architektūrinį kuriamos sistemos vaizdą. Jam pateikti naudojami keletas skirtingų architektūrinių vaizdų, kurie parodo skirtingus kuriamos sistemos architektūrinius aspektus. Šio dokumento tikslas surinkti ir pateikti svarbius architektūrinius sprendimus, kurie buvo atlikti, projektuojant sistemą. Šis dokumentas tarnauja kaip bendravimo medžiaga tarp programinės įrangos architekto ir kitų komandos narių dėl architektūrinių sistemos kūrimo sprendimų.

Šis dokumentas bus pagrindas sudarant sistemos detalią architektūrą, bei bus naudingas koduojant sistemą.

3.6.2. Apžvalga

Dokumentas aprašo lygiagrečių programų interaktyvaus vizualizavimo programinės įrangos sistemos architektūrą. Sistemos nefunkciniai reikalavimai ir apribojimai pateikiami skyriuje „Architektūros tikslai ir apribojimai“. Sistemos panaudojimo atvejai pateikiami skyriuje „Panaudojimo atvejų vaizdas“. Sistemos išskaidymas ir statinė struktūra pateikta skyriuje „Loginis vaizdas“. Sistemos procesai ir jų aprašymai pateikiami skyriuje „Procesų vaizdas“. Sistemos išdėstymas, ir techninė įranga, kurioje bus realizuota sistema, pateikiama skyriuje „Išdėstymo vaizdas“. Skyriuje „Duomenų vaizdas“ pareikiama sistemos duomenų bazės struktūra. Skyriuje „Kokybė“ aprašoma kaip architektūra įtakoja sistemos išplečiamumą, pernešamumą, patikimumą ir pan.

3.6.3. Architektūros pateikimas

Architektūros specifikacija yra parengta, naudojant Rational Unified Process (RUP) projektavimo metodiką, ir CASE priemonę „Rational Rose 2002 Enterprise Edition“, kuri leido greičiau ir efektyviau sudaryti bei analizuoti reikiamas UML diagramas. Sistemos architektūrai pavaizduoti yra naudojami keturi vaizdai: panaudojimo atvejų, loginis, procesų ir išdėstymo. Duomenų vaizdas nebus naudojamas, kadangi kuriamai sistemai nėra reikalinga duomenų bazė. Visi vaizdai ir juos sudarantys modeliavimo elementai yra pateikti 7 paveiksle.

Vaizdas	Diagramos
Panaudojimo atvejų	Panaudojimo atvejų diagramos.
Loginis	Klasių diagramos; Sistemos išskaidymas į paketus.
Procesų	Būsenų diagramos; Sekų diagramos; Bendradarbiavimo diagramos; Veiklos diagramos.
Išdėstymo	Išdėstymo diagramos.

7 pav. Sistemos specifikacijos vaizdai

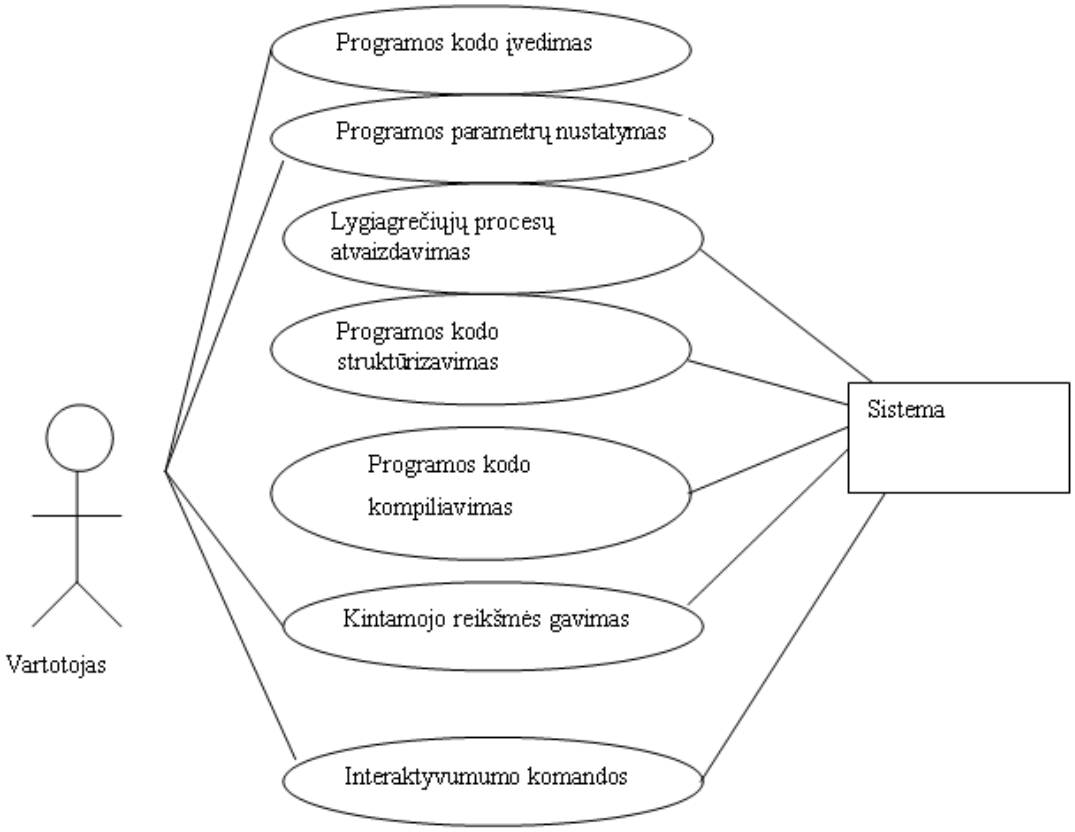
3.6.4. Architektūros tikslai ir apribojimai

Yra keletas reikalavimų ir apribojimų, kurie turi įtaką sistemos architektūrai.

- Sistema turi veikti Linux operacinėje sistemoje
- Sistema naudos GCC kompiliatorių
- Lygiagrečių gijų kūrimui bus naudojamas OpenMP sąsaja
- Vartotojo sąsajos kūrimui bus naudojama QT karkasas (angl. Framework)
- Programos lygiagretaus veikimo atvaizdavimui bus naudojamos UML diagramos
- Vizualizacija bei interaktyvumo komandos turi būti aiškio ir suprantamos net ir nepatyrusiems vartotojams

3.6.5. Panaudojimo atvejų vaizdas

Pateikiami esminiai panaudojimo atvejai.

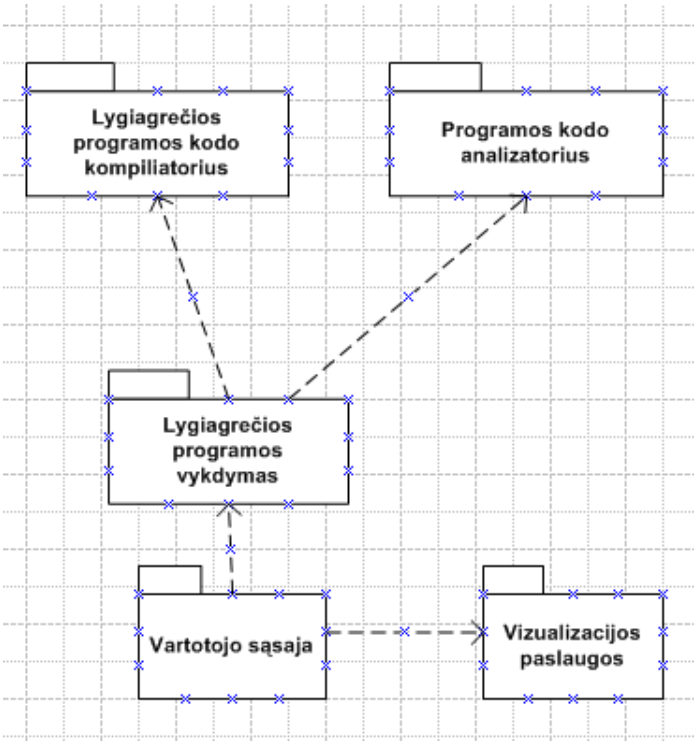


8 pav. Panaudojimo atvejų diagrama

3.7. Sistemos statinis vaizdas

3.7.1. Apžvalga

Sistema suskaidyta į paketus, kurie pateikti paveikslėlyje 9 „Paketų diagrama".

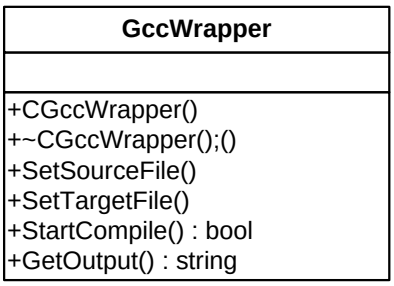


9 pav. Paketų diagrama

3.8. Paketų detalizavimas

3.8.1. Paketas „Lygiagrečios programos kodo kompiliatorius"

Paketas „Lygiagrečios programos kodo kompiliatorius" yra skirtas nustatyti ar vartotojo įvesta programa yra sintaksiškai teisinga, taip pat jis sukompiliuoja ir paruošia vykdomąjį failą.

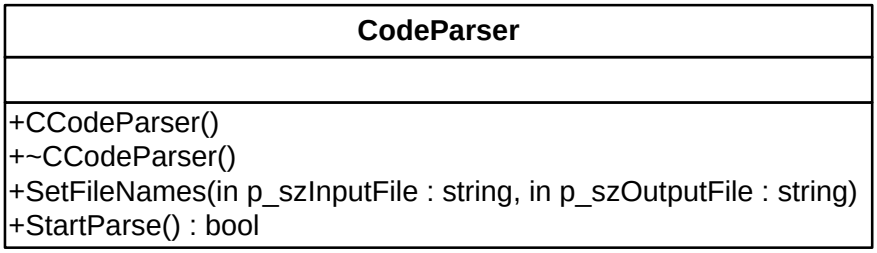


10 pav. Paketo „Lygiagrečios programos kodo kompiliatorius" klasių diagrama

3.8.2. Paketas „Programos kodo analizatorius"

Paketas „Programos kodo analizatorius" atlieka bene esminį vaidmenį visame programos veikime. Jo paskirtis yra analizuoti ir keisti vartotojo įvestos programos kodo turinį, ir sudėti žymes apie kiekvienos kodo eilutės informaciją. Jis rinks kokia informacija

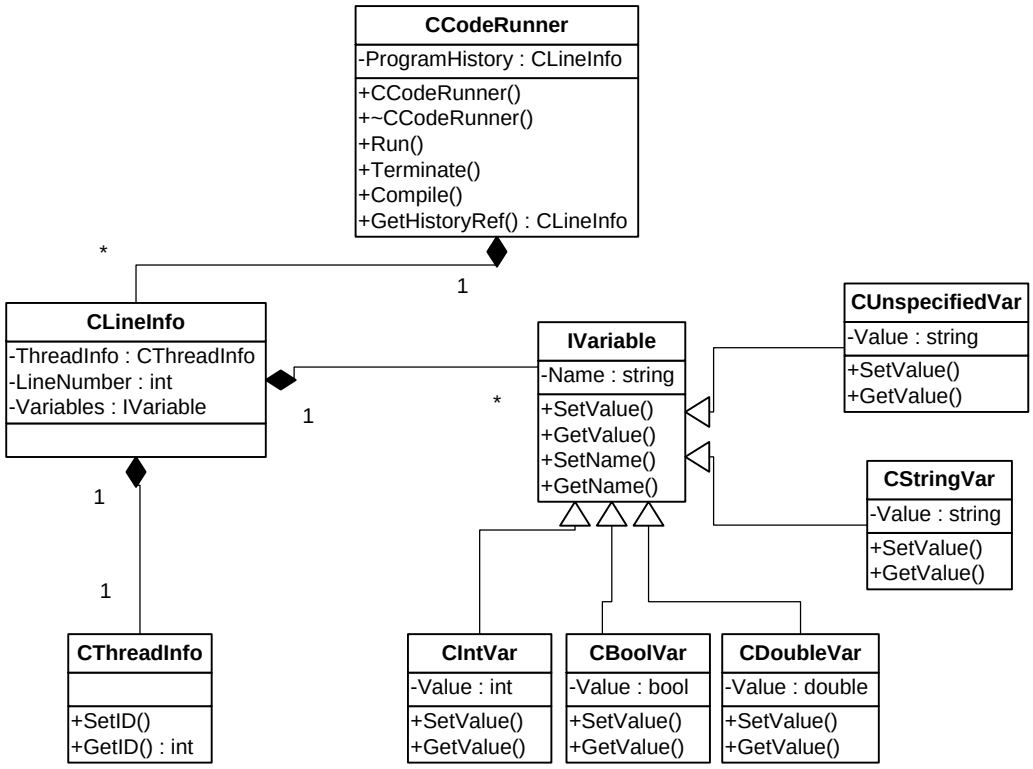
yra pasiekiama kiekvienoje vykdomo kodo eilutėje, t.y. kokie kintamieji yra pasiekiami, gijos identifikacinis numeris, kodo eilutės numeris, ar kodo eilutė priklauso kritinei sekcijai ir t.t. Surinkęs informaciją apie eilutę jis įterps komandą, kuri vykdant vartotojo programą visą surinktą informaciją išspausdins į duomenų srautą „std::out (c++)", duomenų srauto informaciją rinks ir apdoros paketas „Programos vykdymo paslaugos".



11 pav. Paketo „Programos kodo analizatorius" klasių diagrama

3.8.3. Paketas „Lygiagrečios programos vykdymas"

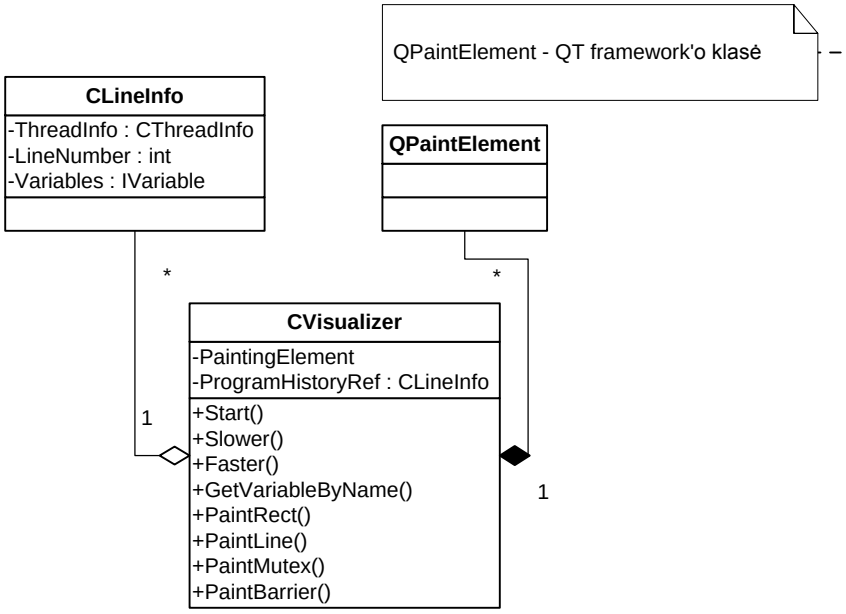
Paketas „Lygiagrečios programos vykdymas" kompiliuoja, vykdo ir renka informaciją bei ją struktūrizuoja apie vartotojo įvestos lygiagrečios programos veikimą, tai yra duomenų šaltinis paketui „Vizualizacijos paslaugos". Žemiau pateikta šio paketo klasių diagrama bus plečiama priklausomai nuo poreikių (pvz. bus pridėta informacija apie funkciją, kurioje šiuo metu veikia gija ir pan.)



12 pav. Paketo „Lygiagrečios programos vykdymas" klasių diagrama

3.8.4. Paketas „Vizualizacijos paslaugos“

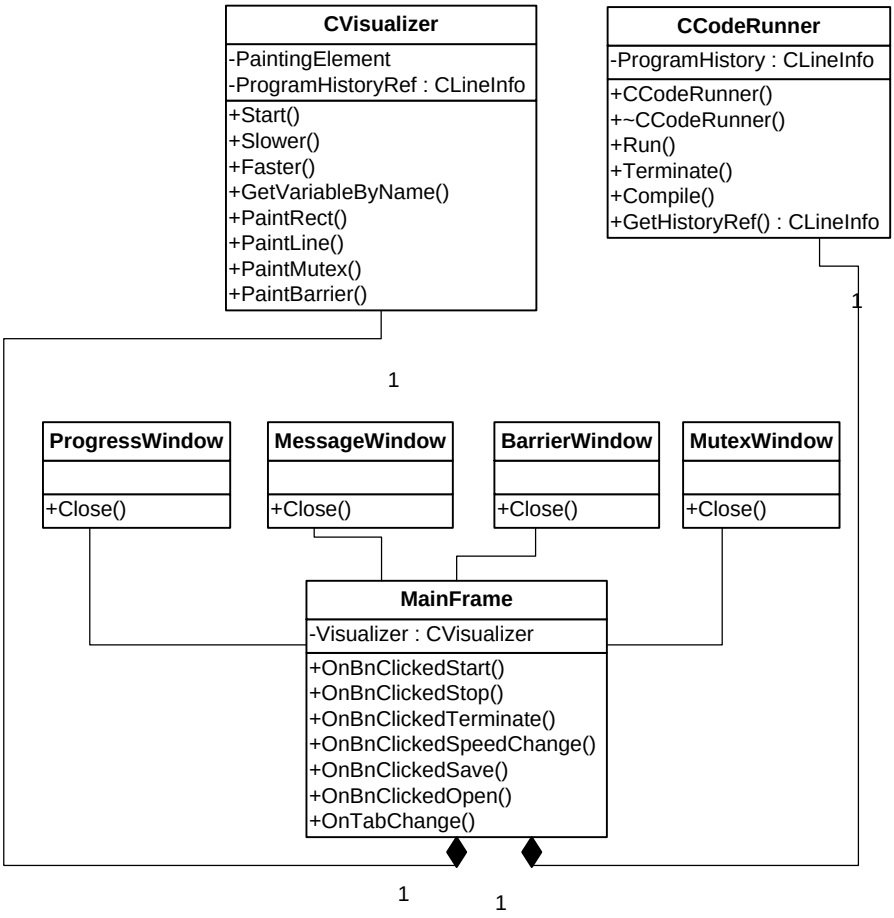
Paketo „Vizualizacijos paslaugos“ paskirtis yra atvaizduoti gijų veikimo istoriją, kurią per vartotojo sąsają ji gaus iš paketo „Lygiagrečios programos vykdymas“, taip pat reaguoti į vartotojo iškviestas interaktyvumo komandas, tokias kaip: pradėti, sustoti, nutraukti vizualizaciją, vykdyti vizualizaciją greičiau/lėčiau, gauti kintamojo reikšmę. Šis paketas tiesiogiai dirbs su vartotojo sąsajos elementu, kuriame bus piešiamas programos veikimo vaizdas.



13 pav. Paketo „Vizualizacijos paslaugos“ klasių diagrama

3.8.5. Paketas „Vartotojo sąsaja“

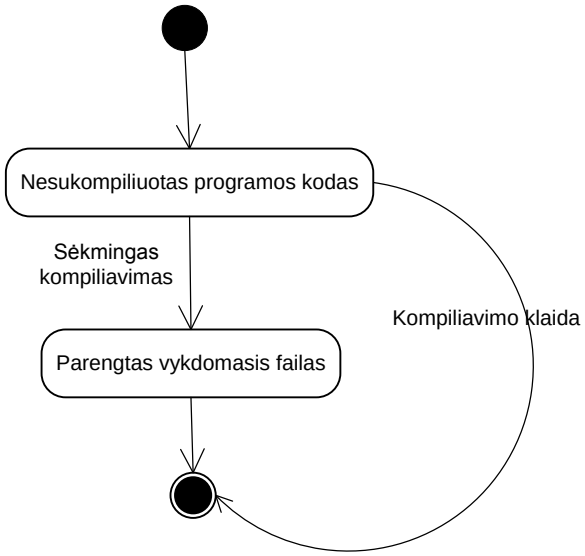
Paketas „Vartotojo sąsaja“ leidžia vartotojui valdyti sistemos darbą, šio paketo pagalba vartotojas turi galimybę vykdyti interaktyvumo komandas, įvesti savo lygiagrečios programos kodą, keisti jį ir išsaugoti pakeitimus.



14 pav. Paketo "Vartotojo sąsaja" klasių diagrama

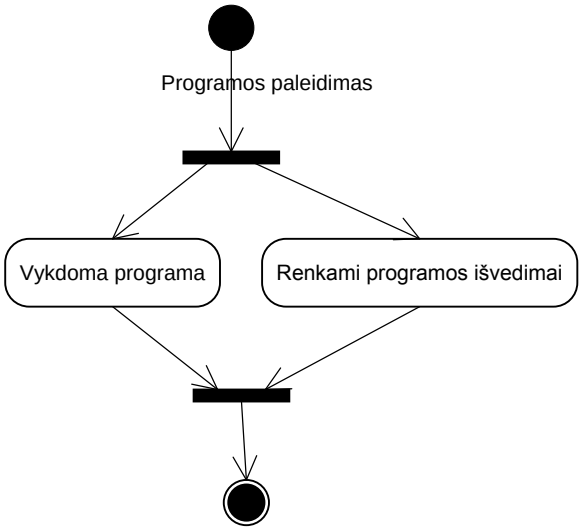
3.9. Sistemos dinaminis vaizdas

Paveikslėlyje 15 pateikiama kompiliavimo proceso būsenų diagrama. Šio proceso užduotis yra paleisti GCC kompiliatorių ir gauti programos vykdomąjį failą, taip pat gauti suformuoti pranešimą apie kompiliavimo rezultatą ir klaidas (jei tokių yra).



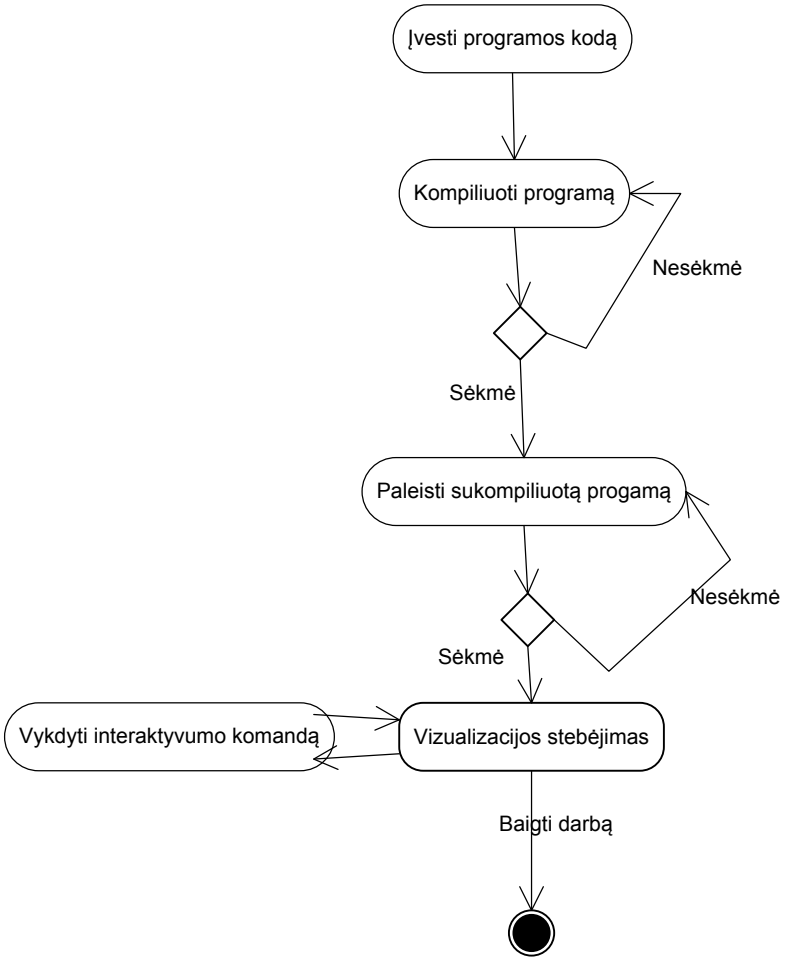
15 pav. Vartotojo programos kompiliavimo būsenų diagrama

Paveikslėlyje 16 pateikiama vartotojo įvestos lygiagrečios programos vykdymo, bei jos duodamų rezultatų stebėjimo būsenų diagrama. Šio proceso tikslas yra paleisti vykdyti programą ir surinkti jos išvedamus duomenis.



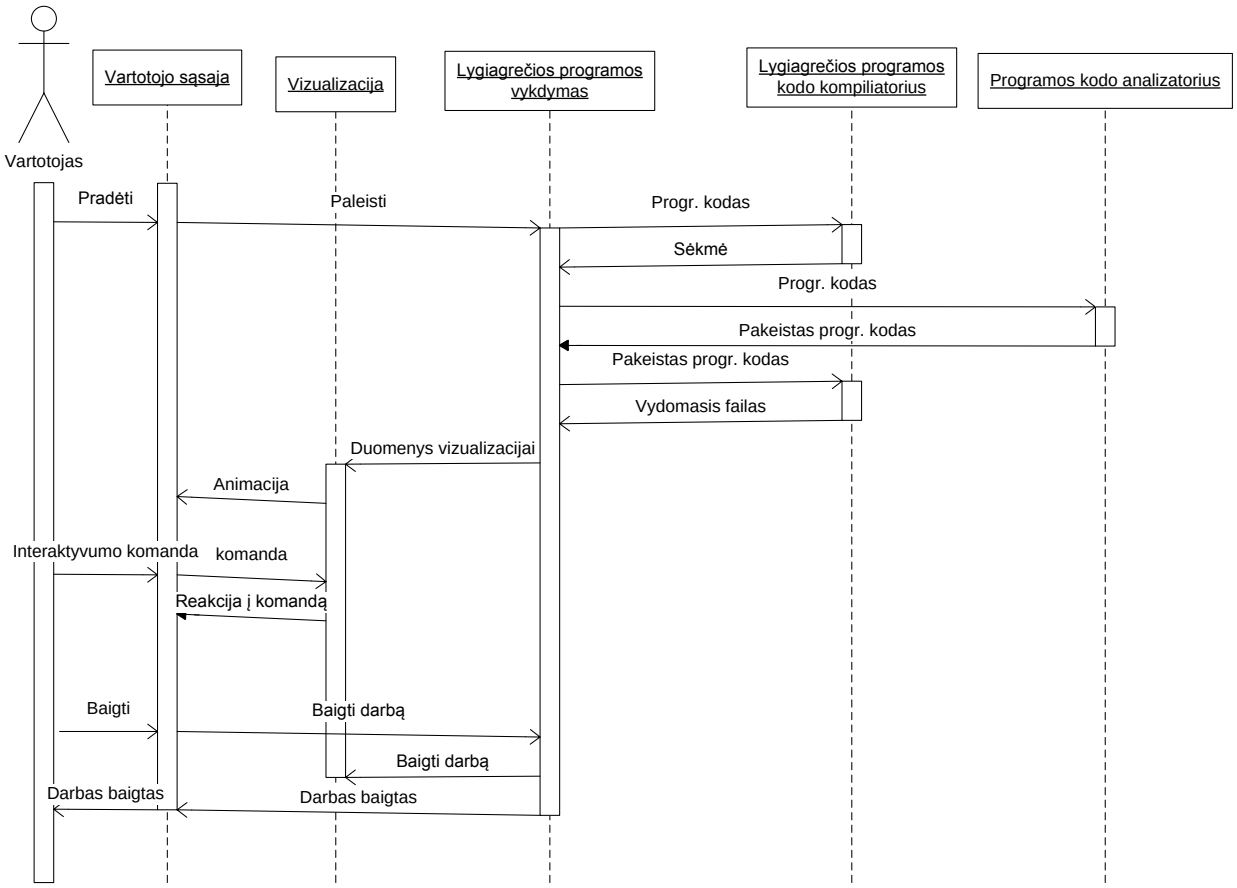
16 pav. Vartotojo programos paleidimo būsenų diagrama

Paveikslėlyje 17 atvaizduota darbo su sistema veiklos diagrama.



17 pav. Darbo su sistema veiklos diagrama

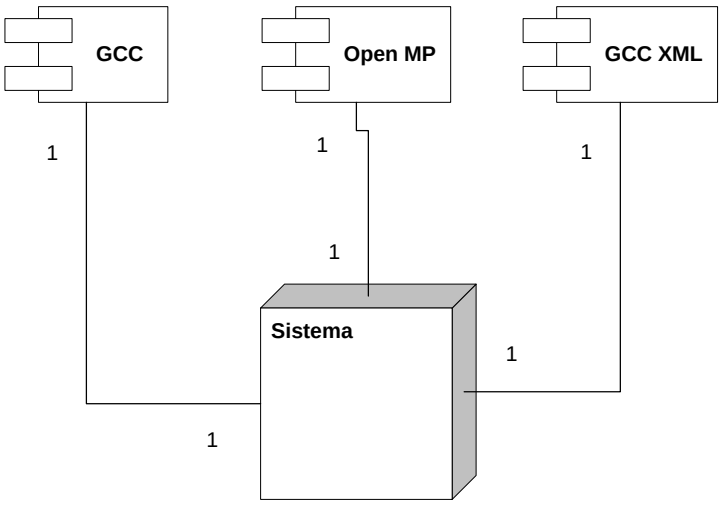
Paveikslėlyje 18 atvaizduota darbo su sistema sekų diagrama.



18 pav. Darbo su sistema sekų diagrama

3.10. Išdėstymo vaizdas

Sistema veiks viename kompiuteryje, nebus jokių nutolusių mazgų. Žemiau pateikti sistemos komponentai.



19 pav. Išdėstymo vaizdas

3.11. Kokybė

Sistemos patikimumas yra teigiamai įtakojamas to, kad į programą įtraukti kitų autorių sukurti komponentai, tokie kaip GCC, GCC XML, tai yra didelio sudėtingumo komponentai, todėl jų panaudojimas leis išvengti klaidų, su kuriomis būtų susidurta, jei būtų nutarta sukurti savo analogus. Paketų bei klasių diagramos buvo projektuojamos taip, kad prireikus būtų galima be didelių pakeitimų išplėsti sistemą. Sistemos kokybę taip pat užtikrins tai, jog ji buvo projektuojama panaudojant profesionalius projektavimo įrankius, atskiros sistemos dalys bus ištestuotos vienetų testavimo metodu.

4. TYRIMO DALIS

4.1. Tyrimo tikslas

Tyrimo dalies tikslas yra parengti duomenis magistrinio darbo sistemai, kad galėtume įvertinti jos funkcionalumą bei naudą vartotojui. Kadangi sistemos duomenys yra programos kodas (parašytas C++ kalba), šioje dalyje aptarsime keletą tipinių lygiagretaus programavimo problemų, kurios vėliau bus suprogramuotos ir įvykdytos naudojant magistrinio darbo sistemą (OMP Visual).

4.2. Tipinės lygiagretaus programavimo problemos

Tipinės lygiagretaus programavimo problemos iliustruoja lygiagrečių programų veikimo situacijas, kurios nėra pageidaujamos ir jas reikia šalinti, vengti arba kitaip spręsti, trumpai apžvelgsime tipines problemas.

Prie dažniausių nepageidaujamų situacijų, su kuriomis susiduria lygiagrečių programų kūrėjai, galime paminėti tokias kaip riboto masyvo – situacija, kai gijos dalijasi bendru riboto dydžio masyvu ir reikia užtikrinti, jog jis nebus perpildytas, aklavietės – situacija, kai visos programos gijos laukia tokio įvykio, kuris niekada neįvyks, badavimo – tai situacija, kai dėl netinkamos algoritmo realizacijos viena ar kelios gijos yra laukimo rėžime, nors nėra jokių apribojimų jų veikimui, gijų sinchronizacijos – situacija, kai reikia užtikrinti, jog gijos bus vykdomos tinkamu eiliškumu, problemas. Riboto masyvo problemą iliustruoja gamintojo-vartotojo pavyzdys. Skaitytojo-rašytojo problema iliustruoja bendros atminties dalinimosi situaciją. Rūkoriaus problema aprašo situaciją, kai gijos dalijasi ne vienu bendros atminties bloku. Pietaujantys filosofai – bene populiariausia lygiagrečių gijų sinchronizacijos problemų iliustracija, ji taip pat kelia aklaviečių ir badavimo problemas.

4.2.1. Gamintojo-vartotojo problema

Gamintojo-vartotojo problema (dar žinoma kaip riboto masyvo (buferio) problema) – tai klasikinis, lygiagrečiai veikiančių gijų, sinchronizacijos problemos pavyzdys. Problema apibrėžia du lygiagrečiai veikiančius procesus (vartotoją ir gamintoją), kurie vienu metu dalinasi bendru riboto dydžio masyvu. Problemos esmė – užtikrinti, kad gamintojas nebandys įdėti duomenų į pilną masyvą, o gamintojas pasiimti duomenų iš tuščio masyvo.

Problemos sprendimas yra „užmigdyti“ gamintojo procesą, jei masyvas yra pilnas, o vartotojui pasiėmus duomenis iš masyvo jis turi „pažadinti“ gamintoją, kad jis pradėtų pildyti masyvą. Iš kitos pusės – vartotojas yra „užmigdomas“, jei masyvas yra tuščias, o gamintojas „prižadina“ vartotoją, kai masyvas papildomas duomenimis. Dažniausiai ši problema yra išsprendžiama panaudojant semaforus, netinkamas problemos sprendimas gali lemti, jog programa paklius į aklavietę (angl. „deadlock“). Netinkamo problemos sprendimo realizacija pateikta 1 priede (realizuota JAVA programavimo kalba). Minėto sprendimo problema yra ta, kad nėra išvengta aklavietės galimybė, jei programos veikimas vyktų pagal tokį scenarijų:

1. Vartotojas nuskaito „itemCount“ kintamąjį, „pastebėjo“, jog jo reikšmė yra „0” ir jau yra bepereinantis į „if“ bloką
2. Prieš pat iškviečiant funkciją „sleep“ vartotojo procesas yra pertraukiamas, o gamintojo vykdymas tęsiamas
3. Gamintojas papildo masyvą ir padidina kintamojo „itemCount“ reikšmę
4. Kadangi masyvas buvo tuščias prieš jį papildant gamintojas bando „pažadinti“ vartotoją

5. Deja, vartotojas dar nebuvo „užmigęs“, todėl „žadinimo“ funkcija yra iškviečiama be reikalo. Tuomet vartotojo procesas yra tęsiamas, jis „užmiega“ ir niekada nebebus pažadintas
6. Gamintojas pildys masyvą, kol jis netaps pilnas ir „užmigs

Šioje situacijoje abu procesai „miega“, kas nuveda programos veikimą į aklavietę.

Problemos sprendimas, realizuotas JAVA kalba, pateikiamas 1 priede. Šiame sprendime panaudoti du semaforai, kurie neleidžia iškviesti „pažadinimo“ funkcijos be reikalo.

4.2.2. Skaitytojo-rašytojo problema

Skaitytojo-rašytojo problema yra dar vienas dažnas lygiagretaus programavimo problemų pavyzdys. Šiame pavyzdyje veikia daug gijų, kurios turi prieiti prie bendros atminties vienu metu, vienos iš jų rašo į atmintį, kitos skaito, natūralus apribojimas yra tai, kad gijai būtų draudžiama skaityti ar rašyti į bendrą atmintį, kol kažkuri kita gija vykdo rašymą į bendrą atmintį, tačiau yra leidžiama dviem skaitančioms gijoms prieiti prie bendros atminties. Kritinę sekciją galima būtų apsaugoti binariniu semaforu, tačiau šiuo atveju išskiriamos dvi pagrindinės problemos:

Tarkime turime bendrąją atmintį, apsaugotą, binariniu semaforu. Šis sprendimas nėra optimalus, kadangi šiuo atveju jokia kita gija negali prieiti prie apsaugotos atminties, o tai reiškia, jog skaitanti gija, užrakinus bendrąją atmintį, neleis prie jos prieiti kitai skaitančiai gijai, nors tokia situacija yra leidžiama, todėl nei viena skaitanti gija neturėtų laukti bendros atminties atlaisvinimo, jei ją užrakinus turi kita skaitanti gija.

Dar viena problema susidaro tuomet, jei bendrąją atmintį yra užrakinus skaitanti gija, o rašanti laukia, kol ji bus atrakinta. Tuo tarpu, pirmajai skaitančiai gijai, bebaigiant darbą, prie bendros atminties „prileidžiama“ kita skaitanti gija (leidžiama situacija) ir tai kartojasi nuolat – skaitančios gijos nuolat laiko užrakintą bendrąją atmintį, o rašančioji gija „badauja“. Sprendimas būtų neleisti kitai skaitančiai gijai prieiti prie bendros atminties, jei eilėje jau laukia rašančioji.

4.2.3. Rūkoriaus problema

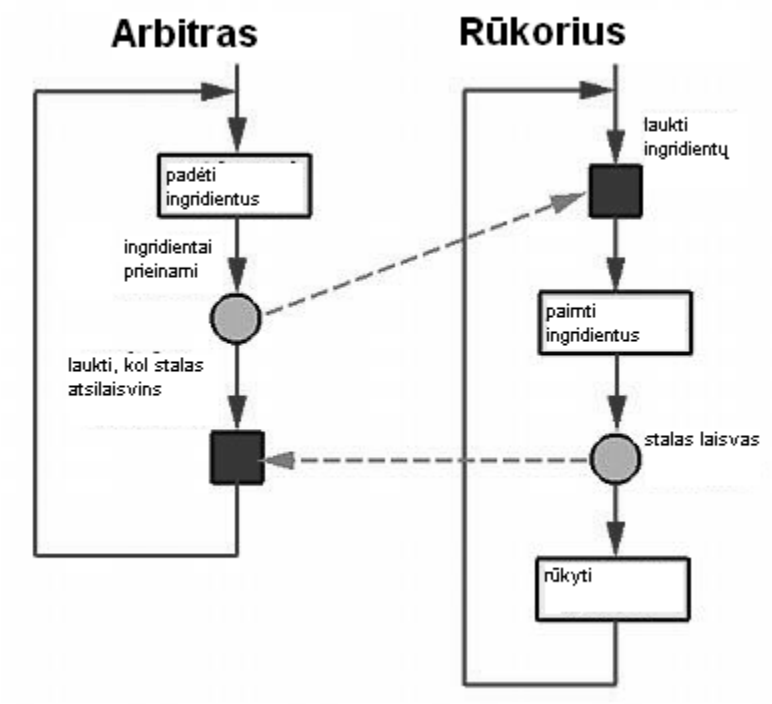
Rūkoriaus problema yra lygiagretumo problema, kuri kompiuterių moksle pirmą kartą buvo suformuluota dar 1971 metais, jos autorius S.S.Patil‘as.

Problemą galime apibrėžti taip: Tarkime, jos rūkorius rūko cigaretes, kuriai pagaminti ir surūkyti yra reikalingo tokios trys pagrindinės sudedamosios dalys:

- Tabakas
- Popierius
- Degtukas

Tarkime, jog prie stalo ratu sėdi trys rūkoriai, iš kurių kiekvienas turi begalinį kiekį vieno iš trijų sudedamųjų dalių – vienas turi begalinį kiekį tabako, kitas – popieriaus, o trečiasis begalinį kiekį degtukų.

Tarkime, jog dar yra ir nerūkantis arbitras. Arbitras leidžia rūkoriams gamintis cigaretes pasirinkdamas du iš jų laisva valia, paimdamas po vieną ingredientą iš kiekvieno „tiekėjo“ ir padėdamas juos ant stalo. Tuomet arbitras liepa trečiajam rūkoriui paimti du ingredientus nuo stalo ir kartu su savo ingredientu pagaminti cigaretę, kurią ji rūkys kurį laiką. Tuo metu, kai arbitras pamato tuščią stalą, vėl atsitiktinai pasirenka du rūkorius ir padeda jų rūkymo ingredientus ant stalo. Šis procesas tęsiamas amžinai, šį procesą galime atvaizduoti diagrama, pateikta 20 paveikslėlyje.



20 pav. Rūkoriaus problemos scenarijus

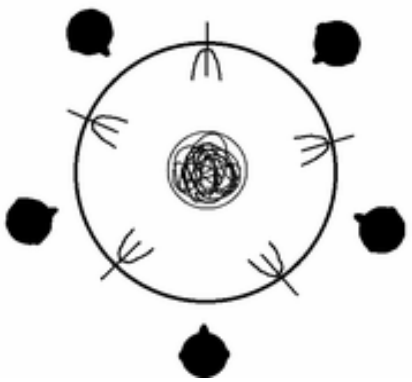
Rūkoriai nekaupia ingredientų ant stalo; rūkorius pradeda gaminti cigaretę tik tuomet, kai jis baigia rūkyti paskutiniąją. Jei arbitras padeda tabaką ir popierių ant stalo, kol rūkorius, turintis degtukus rūko, tabakas ir popierius yra neliečiami, kol rūkantysis nebaigė rūkyti ir jų nepasiima.

Problema kyla bandant parašyti minėtą situaciją simuliuojančią programą, yra jei kuriamas semaforas kiekvienam iš ingredientų, sprendimas būtų naudoti semaforus ingredientų kombinacijoms. Sprendimo realizacija pateikta C++ kalba 2 priede.

4.2.4. Pietaujančių filosofų problema

Pietaujančių filosofų problema – tai klasikinė lygiagrečių programų procesų patekimo į aklavietes ir jų badavimo iliustracija, ji pateikta žemiau.

Penki, tarpusavyje nebendraujantys, filosofai sėdi už apvalaus stalo. Sėdėdami jie daro tik du dalykus: valgo arba galvoja. Kol jie valgo – negalvoja, kol galvoja – nevalgo. Tarp kiekvieno greta sėdinčio filosofo yra šakutė. Tarp kiekvieno filosofo yra šakutė - kitaip tariant yra 5 šakutės. Filosofas galvoja tol, kol išalksta. Tam, kad jis galėtų valgyti reikalingos dvi šakutės. Taigi išalkęs jis pasiima dvi šakutes (tik esančias jam iš kairės ir dešinės), iš pradžių, esančią jam iš kairės, o paskui dešinės, ir pradeda valgyti. Jei šakutė, kurią jis nori paimti yra užimta, jis laukia, tačiau nepadeda atgal jau



21 pav. Pietaujantys filosofai

paimtos šakutės. Kai filosofas pavalgo, jis padeda šakutes atgal ir ima vėl galvoti.

Aprašytoje situacijoje gali susidaryti aklavietė, jei visi filosofai vienu metu išalks ir paims po šakutę iš kairės – iš dešinės šakutės jau bus paimtos ir visi ims laukti, kol atsilaisvins antroji šakutė, o tai, pagal esamą modelį, niekada neįvyks.

Badavimas gali susidaryti, jei visą laiką valgys vienas prieš kitą sėdintys filosofai – tuomet kiti trys niekuomet nepasiims šakučių ir negalės valgyti.

4.2.5. Apibendrinimas

Aptartų tipinių lygiagretaus programavimo problemų apibendrinimui pateikiame lentelę, kurioje matosi kokias probleminės situacijas jos padengia. Ženklu „+“ žymėsime, jei problema iliustruoja atitinkamą probleminę situaciją.

6 lentelė. Probleminių situacijų padengimas

	Gamintojo-vartotojo problema	Skaitytojo-rašytojo problema	Rūkoriaus problema	Pietaujančių filosofų problema
Riboto masyvo situacija	+			
Aklavietės situacija	+		+	+
Badavimo situacija		+	+	+
Gijų sinchronizavimo problemos	+			+
Bendros atminties apsauga	+	+	+	+

Pateiktoje probleminių situacijų padengimo lentelėje matome, jog visos tipinės problemos susiduria su bendros atminties dalijimosi tarp gijų klausimu. Kitos probleminės situacijos taip pat yra iliustruojamos ne viena tipine problema, tačiau beveik kiekvienu atveju probleminių situacijų priežastis yra skirtinga.

5. EKSPERIMENTINĖ DALIS

5.1. Eksperimento tikslas

Eksperimentinėje dalyje yra atliekamas magistratūros studijų metu sukurtos ir įdiegtos programinės įrangos eksperimentinis tyrimas. Eksperimentų duomenys (lygiagrečių programų išvesties tekstai) pateikiami prieduose. Skyriaus pradžioje bus aptariamas bendras sistemos funkcionalumas, pilnas vartotojo vadovas pateiktas 3 priede. Vėliau bus pademonstruota kaip ši sistema pritaikoma, dirbant su tipinėmis lygiagretaus programavimo problemomis, kurios buvo aptartos ankstesniame skyriuje. Skyriaus pabaigoje pateiksime eksperimento rezultatus.

5.2. Sistemos funkcijos

Sistema, analizuoja lygiagrečios programos (parašytos C++ kalba, lygiagretumui naudojant OpenMP technologiją) kodą, ją vykdo ir atvaizduoja jos veikimą vartotojui patogia forma.

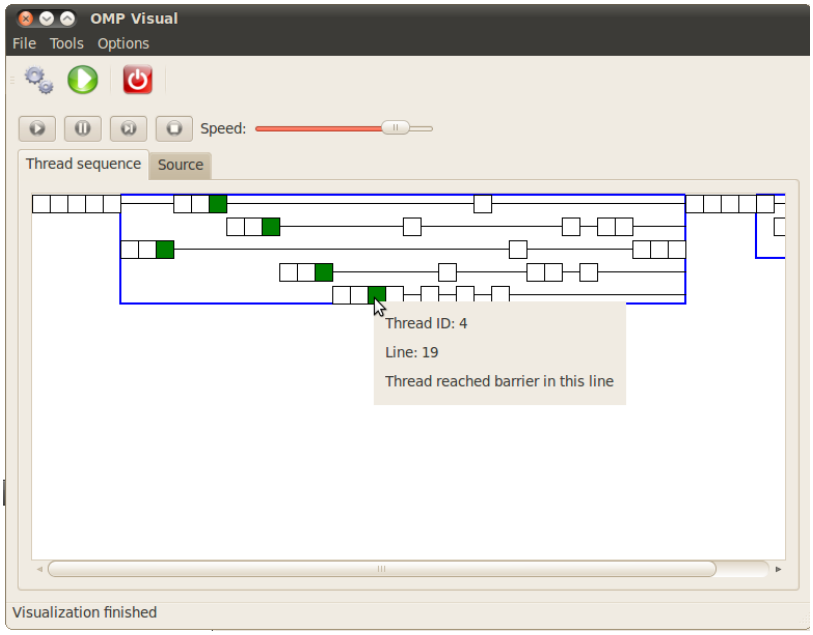
Pagrindinė šios sistemos funkcija, kaip ir aukščiau yra paminėta, yra atvaizduoti lygiagrečios programos kodo veikimą, šiuo atveju buvo atvaizdavimui buvo pasirinkta modifikuota UML sekų diagrama, kuri pakankamai aiškiai demonstruoja atsirandančias gijas, jų veikimą laike viena kitos atžvilgiu.

5.2.1. Informacijos apie gijas stebėjimas

Vykstant vizualizacijos procesui ar jai baigus darbą, galime stebėti informaciją apie gijų veikimą, ją apima:

- Gijos numeris
- Kodo eilutės numeris
- Kintamųjų sąrašas
- Kritinių sekcijų ir kitų reikšmingų įvykių informaciją

Šis funkcionalumas pasiekiamas tiesiog užvedus pelės žymeklį ant reikiamos gijos veikimo vietos (kvadratėlio), tai yra atvaizduota paveikslėlyje nr. 26.



22 pav. Informacijos apie gijas stebėjimas

5.3. Gamintojo – vartotojo problemos modeliavimo programa ir jos tyrimas

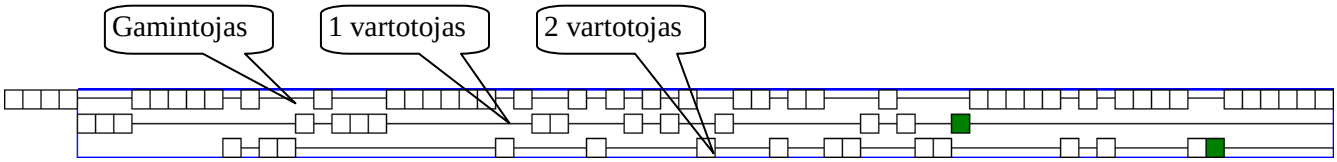
Šioje dalyje tirsime kaip sukurtas įrankis padeda analizuoti gamintojo-vartotojo problemos modeliavimo programą, bei aptikti programos klaidas.

Aptariamos problemos modeliavimo programos yra pateiktos 4 priede, yra pateikta programa su klaidomis, kurios buvo sąmoningai padarytos, bei pateikta gerai veikianti programa.

Pirmiausiai aptarsime programą, kurioje yra klaidų. Svarbiausi programos veikimo aspektai:

- Dirba trys gijos: viena gamintojo ir dvi vartotojo
- Gijos dalijasi bendrą masyvą, kurio dydis yra 3
- Gamintojas baigia darbą, pagaminęs 10 gaminių
- Gamintojo gija baigia darbą, sunaudojus 3 gaminius

Šios programos veikimo vizualizacija pateikta 23 paveikslėlyje, matome tris horizontalias sekas, kurios atspindi tris gijas. Pirmoji nuo viršaus yra gamintojo gija, likusios dvi – vartotojai.

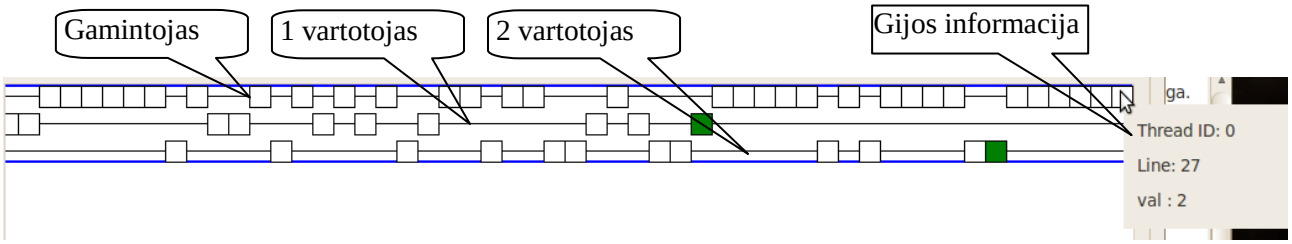


23 pav. Gamintojo-vartotojo programos aklavietė

Galime stebėti, jog neteisingai susinchronizavus gijų veikimą gamintojas negali baigti darbo, jis laukia, kol bus atlaisvintas dar vienas masyvo elementas, tačiau tai niekada neįvyks,

nes vartotojai jau yra baigę darbą ir laukia gamintojo gijos darbo pabaigos ties barjeru – programa patenka į aklavietę, seka nėra papildoma, o informacija apie vizualizacijos pabaigą negavome.

Jei panaudotume kintamųjų stebėjimo funkciją (žr. 3 priede), galėtume nustatyti aklavietės priežastį – masyvas yra užpildytas, tai pateikta 24 paveikslėlyje (kintamojo „val“ reikšmė yra 2).



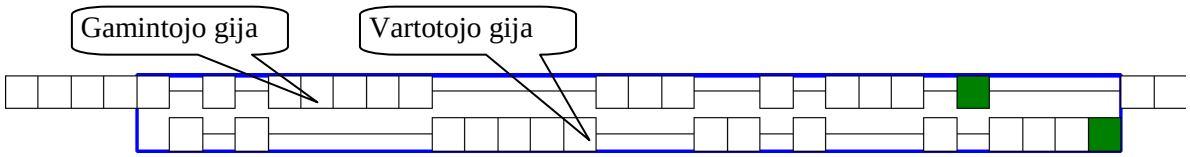
24 pav. Gamintojo-vartotojo programos aklavietės priežastis

Taigi programuotojas gali daryti išvadą, jog modeliuojant šią programą reikia atkreipti dėmesį į tai, kiek gijų veikia, kiek jos pagamina ir sunaudoja masyvo elementų, o taip pat į tai, koks yra masyvo dydis, kad programa nepakliūtų į aklavietę.

Į tai atsižvelgus parašome teisingai veikiančią programą, kuri pateikta 4 priede, svarbiausi programos aspektai:

- Lygiagrečiai dirba dvi gijos: viena gamintojo ir viena vartotojo
- Gijos dalijasi masyvą, kurio dydis 1
- Gamintojas pagamina 3 gaminius
- Vartotojas sunaudoja taip pat 3 gaminius

Šios veikimo vizualizacija pateikta 25 paveikslėlyje.



25 pav. Teisinga gamintojo-vartotojo problemos realizacija

Paveikslėlyje matome dvi horizontalias sekas (gijas), iš kurių pirmoji nuo viršaus yra gamintojo, o antroji – vartotojo. Akivaizdu, jog dabar programa yra tinkamai realizuota, pirmoji darbą baigė gamintojo gija, paskui ją – vartotojo, gijos viena kitos sulaukė ties barjeru ir valdančioji gija (angl. „master thread“) užbaigė programos darbą.

5.4. Skaitytojo - rašytojo problemos modeliavimo programa ir jos tyrimas

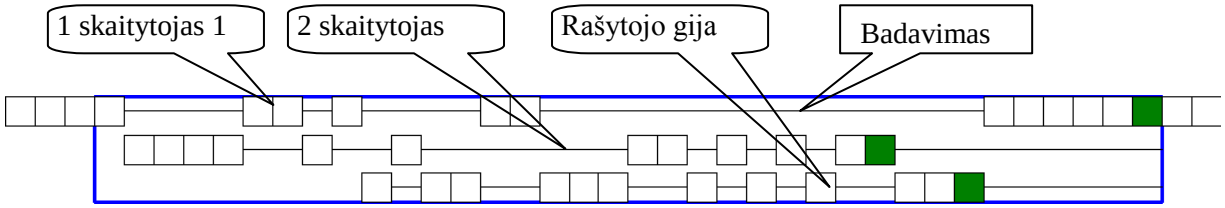
Šioje dalyje tirsime kaip sukurtas įrankis padeda analizuoti skaitytojo-rašytojo problemos modeliavimo programą, bei aptikti programos klaidas. Aptariamos problemos modeliavimo programos yra pateiktos 5 priede.

Kaip ir ankstesniame skyrelyje, šiame taip pat pateiksime programas, kuriose sąmoningai buvo padarytos klaidos, bei aptarsime kaip tas klaidas padeda aptikti sukurtas įrankis.

Pirmoji gamintojo-vartotojo problemą modeliuojanti programa, pateikta 5 priede yra realizuota bendrąją atmintį apsaugant binariniu semaforu, svarbiausi programos aspektai yra šie:

- Lygiagrečiai dirba trys gijos: dvi skaitytojo ir viena rašanti
- Skaitančios gijos po tris kartus skaito bendrąją atmintį
- Rašytojo gija tris kartus rašo į bendrąją atmintį
- Bendroji atmintis apsaugota binariniu semaforu

Programos vizualizaciją pateikiame 26 paveikslėlyje.



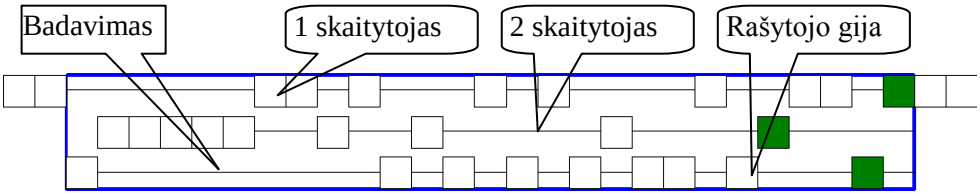
26 pav. Skaitytojo-rašytojo problema. Binarinis semaforas

Paveiksle matome tris lygiagrečiai veikiančių gijų sekas, dvi pirmosios nuo viršaus yra skaitančios gijos, trečioji – rašytojo gija. Matome, jog programa sėkmingai baigia darbą, į aklavietę nepatenka, tačiau, kaip ir minėta skyrelyje 4.1.2. binarinis semaforas nėra optimalus sprendimas, apsaugant bendrąją atmintį – galime pastebėti pirmosios skaitančios gijos badavimą, ji kiek paveikia programos pradžioje, o didžiąją dalį darbo padaro tik kai antroji skaitančioji ir rašančioji gijos baigia dirbti. Taigi pirmoji gija pralaimi kovą dėl resursų ir ilgą laiką neveikia, nors ji galėtų veikti lygiagrečiai su kita skaitančiąja gija.

Antrojoje, skaitytojo-rašytojo problemą modeliuojančioje programoje (žr. 5 priedas), bandome spręsti aukščiau iškilusią problemą – leidžiame skaitytojo gijoms vienu metu skaityti iš bendrosios atminties. Svarbiausi programos aspektai yra šie:

- Lygiagrečiai dirba trys gijos: dvi skaitytojo ir viena rašanti
- Skaitančios gijos po tris kartus skaito bendrąją atmintį
- Rašytojo gija tris kartus rašo į bendrąją atmintį
- Bendroji atmintis apsaugota „omp critical“ regionu

Programos vizualizacija pateikta 27 paveiksle.



27 pav. Skaitytojo-rašytojo problema. "Omp critical" regionas

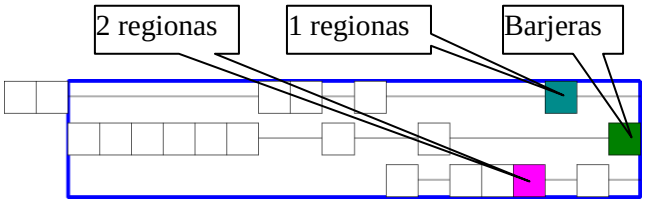
Paveiksle matome tris lygiagrečiai veikiančių gijų sekas, dvi pirmosios nuo viršaus yra skaitančios gijos, trečioji – rašytojo gija. Kaip ir aukščiau buvusi programa, ši taip pat

sėkmingai baigia darbą, tačiau šiuo atveju pačioje programos veikimo pradžioje galime stebėti rašančios gijos badavimą – bendrąją atmintį dalijasi abi skaitančios gijos ir neprileidžia rašančiosios (ši problema jau aptarta 4.1.2. skyrelyje).

Bandome spręsti šia problemą, modifikuodami programą dar kartą, kad skaitančiosios ir rašančios gijos būtų apsaugotos skirtingais užraktais ir taip galėtume prileisti skaitančiąsias gijas vienu metu kreiptis į bendrąją atmintį, o rašančioji naudotų kitą užraktą (žr. 5 priede). Svarbiausi programos aspektai yra šie:

- Lygiagrečiai dirba trys gijos: dvi skaitytojo ir viena rašanti
- Skaitančios gijos po tris kartus skaito bendrąją atmintį
- Rašytojo gija tris kartus rašo į bendrąją atmintį
- Bendroji atmintis apsaugota skirtingais „omp critical“ regionais

Programos vizualizacija pateikta 28 paveiksle.



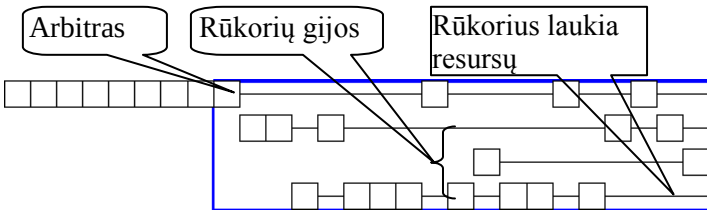
28 pav. Skaitytojo-rašytojo problema. Skirtingi "Omp critical" regionai

Paveiksle matome tris lygiagrečiai veikiančių gijų sekas, dvi pirmosios nuo viršaus yra skaitančios gijos, trečioji – rašytojo gija. Pirmoji gija – mėlynas kvadratėlis, trečioji gija – rožinis kvadratėlis, skirtingos spalvos žymi darbą skirtinguose regionuose. Antroji gija (žalia spalva) – laukia ties barjeru likusiųjų gijų. Akivaizdu, kad sprendimas bendrąją atmintį apsaugoti skirtingais „omp critical“ regionais buvo klaidingas, nes pastebime, jog programos vykdymo metu susidaro tokia situacija, kai skaitančioji ir rašančioji gijos vienu metu dirba „omp critical“ skirtinguose regionuose.

5.5. Rūkoriaus problemos modeliavimo programa ir jos tyrimas

Šioje dalyje tirsime kaip sukurtas įrankis padeda analizuoti rūkoriaus problemos modeliavimo programą, bei aptikti programos klaidas. Aptariamos problemos modeliavimo programos yra pateiktos 6 priede.

Kaip ir minėta skyrelyje 4.1.3, realizuojant šią problemą modeliuojančią programą, susiduriama su aklaviete, jei resursų (tabako, popieriaus ir degtukų) apsauga įgyvendinama panaudojant binarinį semaforą. Lygiagrečiai veikia keturios gijos: arbitras, ir trys rūkoriai su skirtingais ingredientais. Programos vizualizacija pateikta 29 paveikslėlyje.



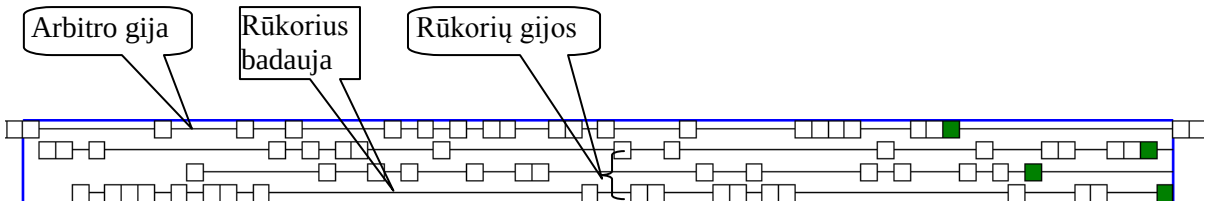
29 pav. Rūkoriaus problema. Binarinis semaforas

Paveiksle matome keturias veikiančias gijas, kurios atspindi tokius „dalyvius“, vardijant nuo viršaus:

- Arbitras
- Rūkorius su popierium
- Rūkorius su tabaku
- Rūkorius su degtukais

Galime stebėti aklavietės būseną, kuomet arbitras ant stalo padeda tabaką ir popierių, nors šie ingredientai yra reikalingi rūkoriui su degtukais, tačiau pirmiau už jį tabaką pasiima rūkorius, kuris turi popieriaus, o popierių pasiima rūkorius, kuris turi tabako. Arbitras nėra pažadinamas ir nededa naujų ingredientų ant stalo, kadangi rūkorius su degtukais nepranešė apie tai, jog jis pasiėmė abu ingredientus (jie buvo paimti kitų) ir pradėjo rūkyti.

Skyrelyje 4.1.3 pateiktas pasiūlymas kaip išspręsti šia problemą – naudojant užraktus ne konkretiems ingredientams, o jų kombinacijoms. Antroji šios problemos realizacija pateikta 6 priede, jos vizualizaciją pateikiame 30 paveikslėlyje.



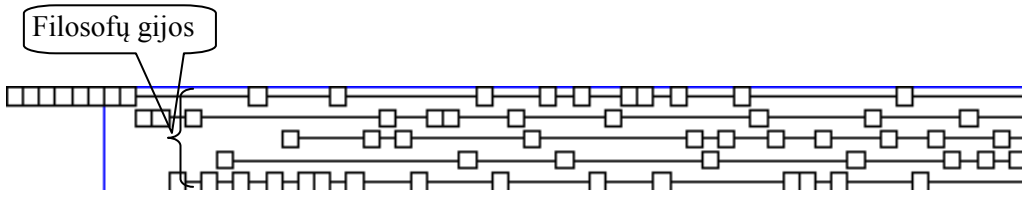
30 pav. Rūkoriaus problema. Ingridientų kombinacijos apsaugotos binariniais semaforais

Paveikslėlyje stebime tas pačias gijas kaip ir pirmame pavyzdyje, tačiau šiuo atveju įvestas programos patobulinimas padeda išvengti aklavietės. Taip pat galime pastebėti, jog rūkorius, turintis popieriaus beveik nerūko, tačiau tai nereiškia, jog vyksta gijos badavimas. Šiuo atveju arbitras, kuris atsitiktinai parenka ingredientų komplektus, retai išrenka jam tinkamus ingredientus. Norint, kad visi rūkoriai dirbtų vienodu tempu reikėtų atsisakyti atsitiktinio ingredientų parinkimo funkcijos.

5.6. Pietaujančių filosofų problemos modeliavimo programa ir jos tyrimas

Šioje dalyje tirsime kaip sukurtas įrankis padeda analizuoti pietaujančių filosofų problemos modeliavimo programą, bei aptikti programos klaidas. Aptariamos problemos modeliavimo programos yra pateiktos 7 priede. Kaip ir ankstesniuose skyreliuose, šiame taip pat pateiksime kelias pietaujančių filosofų problemą modeliuojančias programas bei jų vizualizacijas.

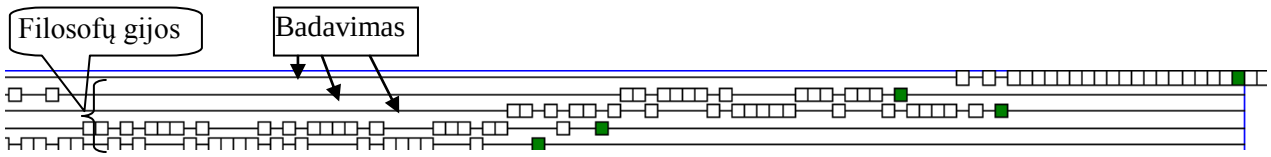
Pirmoji realizacija duoda visišką laisvę veikti filosofų gijoms, nėra vengiama badavimo ar aklaviečių situacija, šios programos vizualizaciją pateikiame 31 paveikslėlyje.



31 pav. Pietaujantys filosofai. Aklavietė

Iš paveikslėlio aišku, jog labai ilgai laukt nereikia, kuomet programa atsidurs aklavietėje – situacija tokia, kad visi filosofai vienu metu pasiėmė kairiąją šakutę, dešinioji niekada neatsilaisvins, o filosofai visada to lauks.

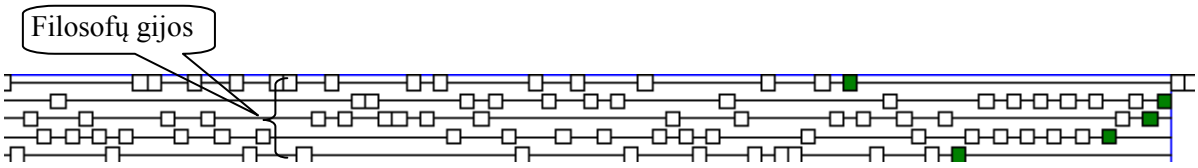
Antroji problemos realizacija leidžia tik vienam filosofui pietauti, akivaizdu, jog tokiu atveju problemą modeliuojanti programa niekada nepateks į aklavietę, programos vizualizacija pateikta 32 paveiksle.



32 pav. Pietaujantys filosofai. Prie stalo valgo tik vienas

Paveikslėlyje matome penkias horizontalias sekas, kurios atspindi gijų (filosofų) veiklą. Programa kiekvienam filosofui papietavus po tris kartus baigia darbą, aklavietės šiuo atveju išvengiame, tačiau galime pastebėti, jog pietaujant tik vienam filosofui kiti badauja – negauna priėjimo prie stalo, nors pietauti vienas kitam netrukdant galėtų du filosofai (sėdintys vienas prieš kitą).

Problemą modeliuojančia programą pakeičiame taip, kad prie stalo galėtų sėdėti ir pietauti trys filosofai, t.y. trys filosofai turėtų priėjimą prie resursų – šakučių. Programos vizualizacija pateikta 33 paveiksle.



33 pav. Pietaujantys filosofai. Trys sėdi prie stalo

Paveikslėlyje matome, kad programa taip pat nepatenka į aklavietę, o ir pakankamai tolygiai laike pasiskirstęs gijų veikimas rodo, jog badavimo situacijos taip pat buvo išvengta.

5.7. Eksperimento rezultatai

Šio skyrelio pradžioje aptarsime kaip sukurtas įrankis padėjo aptikti, bei spręsti eksperimento metu sukurtų programų klaidas, skyrelio pabaigoje palyginsime sukurtą sistemą alternatyvų, aprašytų 2.4. skyrelyje kontekste.

Buvo aptartos keturios klasikinės lygiagretaus programavimo problemos bei išanalizuoti įvairių jų realizacijų veikimai, panaudojant magistrinio darbo metu sukurtą įrankį. Atliktas eksperimentas parodė, jog magistrinio darbo metu sukurtą įrankį galima panaudoti sprendžiant problemas, susijusias su lygiagrečiu programavimu, vizualizacija, paremta sekų diagramomis leido pakankamai greitai įsigilinti į konkrečios problemos realizacijos ypatumus ir aptikti trūkumus. Įrankio pagalba galėjome identifikuoti:

- Programos aklavietes
- Gijų badavimus
- Neteisingą bendrosios gijų atminties apsaugą
- Aukščiau paminėtų problemų šaltinius

Programų aklavietės aiškiai pasimatydavo, kuomet vizualizacijos lange nebūdavo atnaujinama gijų veikimo informacija, o sistema nepranešdavo apie sėkmingą vizualizacijos

pabaigą. Gijų badavimai buvo akivaizdūs tuomet, kai programos veikimo metu gija būdavo laukimo rėžime (grafiškai pažymėta kaip horizontali linija), nors apie kitų gijų veikimą įrankis informuodavo (grafiškai pažymėta stačiakampiais). Neteisinga bendrosios atminties apsaugą galime pastebėti programos veikimo metu, kuomet dvi gijos vienu metu patenka į kritinę sekciją, kuri turėtų saugoti nuo tokių situacijų. Problemų šaltinius identifikavome sekdami kintamųjų reikšmių kitimą, sekdami gijų veikimo chronologiją bei tai, kaip jos patenka į kritines sekcijas.

Žemiau aptarsime kaip sistema atitinka kriterijus (7 lentelė), iškeltus 2.3. skyrelyje, bei pateiksime sistemos vertinimą (8 lentelė) penkiabalėje sistemoje, šalia aptartų alternatyvų įvertinimų.

7 lentelė. Kriterijų išpildymas

Kriterijus	**Trumpas aptarimas, kaip įrankis išpildo kriterijų**
Kintamųjų reikšmių stebėjimas	Sukurtas įrankis leidžia stebėti visų nesudėtingų tipų kintamųjų reikšmes (int, bool, string, char ir t.t.) , tačiau kompleksinių kintamųjų (list, struct ir pan.) jis neapdoroja.
Programos vykdymo stebėjimas realiu laiku	Programos veikimą galima stebėti iš karto po jos sėkmingo kompiliavimo, vartotojui pateikiama vizualizacija, paremta sekų diagramų principu.
Programos kodo redaktorius	Sistema turi kodo redaktorių, kuris orientuotas į „Open MP“ bibliotekos raktinių žodžių žymėjimą, taip pat žymi specialius C++ kalbos žodžius, yra kodo eilučių numeravimas, pažymima aktyvi(redaguojama) eilutė.
Interaktyvumas	Sukurtame įrankyje įgyvendinta galimybė vartotojui įsiterpti į vizualizacijos procesą, t.y. vykdyti tokias komandas kaip vizualizacijos stabdymas, tęsimas (po sustabdymo), vizualizacijos nutraukimas (lygiagrečios programos darbo nutraukimas), vizualizacijos greičio reguliavimas.
Galimybė stebėti atskiras gijas	Lygiagrečios programos vizualizacijos metu sekų diagramoje yra išskiriamos visos veikiančios gijos, galima stebėti kiekvienos gijos būseną, jai prieinamų kintamųjų reikšmes.
Rezultatų saugojimas	Įrankis leidžia vizualizacijos rezultatus išsaugoti PNG formatu.

Kelių bandymų palyginimas	Tokios galimybės nėra.
Profiliavimas	Tokios galimybės nėra.
Aklaviečių aptikimas	Sistema automatiškai neaptinka aklaviečių, tačiau vartotojas stebėdamas lygiagrečios programos veikimą gali suprasti kada ji pakliuvo į tokią situaciją.
Begalinių ciklų aptikimas	Sistema automatiškai neaptinka begalinių ciklų, tačiau vartotojas stebėdamas lygiagrečios programos veikimą gali suprasti kada ji pakliuvo į tokią situaciją – gija nuolat veikia tose pačiose kodo eilutėse.

Sukurtos sistemos (OMP Visual) skaitinis įvertinimas pagrįstas lyginimu su išnagrinėtomis alternatyviomis sistemomis:

- Kintamųjų reikšmių stebėjimą sukurta sistema išpildo geriau, nei „Convit“ alternatyva, kadangi ji palaiko daugiau kintamųjų tipų, tačiau kriterijų ji išpildo prasčiau, nei „Intel Parallel Studio“, kadangi pastaroji sistema palaiko visus C++ kalbos tipus.
- „OMP Visual“ visiškai išpildo programos vykdymo stebėjimo kriterijaus apibrėžime esančius reikalavimus, todėl kaip ir kitos alternatyvos yra įvertinta aukščiausiu balu.
- Sukurta sistema turi integruotą programos kodo redaktorių, kuris žymi kai kuriuos C++ kalbos rezervuotus žodžius, turi kitas programos teksto rinkimą palengvinančias funkcijas (aptarta anksčiau), tačiau „UPPAAL“ alternatyva turi funkcionalesnį programos teksto redaktorių.
- Begalinių ciklų ir aklaviečių aptikimas, naudojantis „OMP Visual“ nėra automatizuotas, programuotojas jį gali pastebėti iš grafinės vizualizacijos, o alternatyvos „UPPAAL“ ir „Intel Parallel Studio“ yra išpildžiusios šiuos kriterijų – sistemos automatiškai aptinka galimus begalinius ciklus ir potencialias aklavietes.

Vertinimui apibendrinti 8 lentelėje pateikiami sukurtos sistemos skaitiniai įvertinimai, šalia pateikti alternatyvų įvertinimai, kurie buvo aptarti anksčiau.

8 lentelė. Sukurtos sistemos palyginimas su alternatyvomis

	UPPAAL	Convit	PokaW	Gthread	Intel Parallel Studio	OMP Visual
Kintamųjų reikšmių stebėjimas	5	3	4	3	5	4
Programos vykdymo stebėjimas realiu laiku	5	5	5	-	5	5
Programos kodo redaktorius	5	2	3	3	-	4
Interaktyvumas	-	4	-	-	5	4
Galimybė stebėti atskiras gijas	-	2	5	4	4	4
Rezultatų saugojimas	5	5	5	5	5	5
Kelių bandymų palyginimas	-	-	-	-	5	-
Profiliavimas	4	-	-	-	5	-
Aklaviečių aptikimas	3	-	-	-	4	1
Begalinių ciklų aptikimas	4	-	-	-	4	1

Įvertinimai rodo, kad sukurtas įrankis yra gana vykęs – savo galimybėmis jis nenusileidžia, o kai kur ir lenkia aptartas alternatyvias sistemas, tačiau kaip ir kiekviena sistema turi savo trūkumų.

5.8. Siūlomi patobulinimai

Šiame skyrelyje trumpai aptarsime kaip būtų galima praplėsti bei patobulinti magistro darbo metu sukurtą sistemą, kokie galėtų būti tolimesni darbai ties šia sistema. Pirmiausiai aptarsime kaip galima būtų pagerinti jau esamą funkcionalumą, vėliau aptarsime kokiomis naujomis funkcijomis būtų galima papildyti sukurtą įrankį ir kokią naudą tai atneštų vartotojui:

- **Kintamųjų reikšmių stebėjimas** turėtų būti praplėstas tiek, kad vartotojas galėtų stebėti ir sudėtingų tipų kintamųjų reikšmes, tai tam tikrais atvejais paspartintų klaidų šaltinių aptikimą
- **Programos kodo redaktorius** galėtų būti patobulintas tokiu funkcionalumu kaip automatinis užpildymas (angl. „auto complete“), galimybe keliais paspaudimais patekti į kintamųjų deklaravimo vietas, programos teksto rinkimo metu sintaksės teisingumo tikrinimas ir pan. Tobulesnis redaktorius paspartintų programavimo procesą.
- **Profiliavimas.** Statistikos rinkimas apie gijų veikimą būtų naujas funkcionalumas, kuris leistų sistemos vartotojui optimizuoti programą, jis galėtų greitai įvertinti kuriose kodo eilutėse gija praleidžia daugiausiai laiko, kurias kodo eilutes dažniausiai vykdo, kiek resursų (pvz. operatyviosios atminties) ji naudoja ir t.t.

- **Metodų kvietimo medis.** Dar vienas naujas funkcionalumas, kuris paspartintų programos klaidų šaltinių aptikimą. Ši funkcija leistų stebėti kokiu eiliškumu tam tikra gija vykdė konkrečius metodus, kiek juose užtruko ir pan.

Taigi įdiegus siūlomus patobulinimus bei naujas funkcijas sistema iškeltus kriterijus (žr. 2.3. Lygiagrečių programų stebėjimo įrankių vertinimo kriterijai) išpildytų dar geriau, taptų patogesne vartotojui.

6. IŠVADOS

Šio darbo analitinėje dalyje išnagrinėti literatūros šaltiniai, kurių autoriai tyrė, kaip turėtų būti suprojektuota lygiagrečias programas vizualizuojanti sistema bei kaip turėtų atrodyti pati vizualizacija. Sudarytas sistemų, vizualizuojančių lygiagrečias programas, vertinimo kriterijų sąrašas. Rastos tokių sistemų alternatyvos išanalizuotos ir įvertintos penkiabalėje sistemoje pagal sudarytą kriterijų sąrašą.

Projektinėje darbo dalyje buvo surinkti reikalavimai kuriamai sistemai ir atlikta jų analizė. Vadovaujantis išnagrinėtos literatūros rekomendacijomis bei surinktais reikalavimais, buvo suprojektuota ir realizuota magistrinio darbo sistema „OMP Visual“. Sistema buvo realizuota bei ištestuota. Testavimo rezultatai parodė, jog sistema funkcionuoja tinkamai.

Tyrimo dalyje atlikta tipinių lygiagretaus programavimo problemų analizė, aprašyta, kokias problemines situacijas jos iliustruoja. Aprašytos probleminės situacijos, susijusios su lygiagrečiu programavimu, pristatyti jų sprendimo būdai. Parengti duomenys sukurtos sistemos funkcionalumui bei jos teikiamai naudai įvertinti.

Eksperimentinėje dalyje, naudojant sukurtą sistemą, buvo patikrintos klaidingos ir teisingos tipinių lygiagretaus programavimo problemų realizacijos. Eksperimentas parodė sukurtos sistemos tinkamumą lygiagrečių programų vizualizacijai bei klaidų jose aptikimui. Sistemos funkcionalumas suteikė galimybę aptikti klaidų ir nepageidaujamų situacijų šaltinius.

Darbo pabaigoje, remiantis sudarytais vertinimo kriterijais, nustatyti tobulintini sistemos komponentai. Numatytos sukurtos sistemos plėtros galimybės.

7. LITERATŪRA

[1] P.Braun, R. Wismuller. Visualization of Parallel Program Execution, 1995.

[2] J.T.Stasko, E.Kraemer. A Methodology for Building Application-Specific Visualisations of parallel Programs, 1992.

[3] J.T.Stasko, E.Kraemer. The Visualization of Parallel Systems: An overview, 1992.

[4] J.Brown, A.Geist, C. Pancake, D.Rover. Software Tool for Developing Parallel Applications Part2: Interactive Control and Performance Tuning, 1997.

[5] M.T.Heath, J.A.Etheridge. Visualizing the Performance of Parallel Programs, 1991.

[6] K.L.Karavanic, J.Myllymaki, M. Livny, B.P.Miller. Integrated Visualization of Parallel Program Performance Data, 1997.

[7] C.Artho, K.Havelund, S.Honiden. Visualization of Concurrent Executions, 2007.

[8] B.P.Miller. What to Draw? When to Draw? An Essay on parallel Program Visualization, 1992.

[9] Q.A.Zhao, J.T.Stasko. Visualizing the Execution of Threads-based Parallel Programs, 1995.

[10] UPPAAL sistema. http://www.uppaal.com/ (žiūrėta 2009)

[11] CONVIT sistema. http://www.cs.tut.fi/~convit/ (žiūrėta 2009)

[12] PPOLKAW sistema. http://www.cc.gatech.edu/gvu/softviz/parviz/polka.html (žiūrėta 2009)

[13] GThread įrankis. http://www.cc.gatech.edu/gvu/softviz/parviz/gthread/gthread.html (žiūrėta 2009).

[14] Lygiagretus programavimas. http://en.wikipedia.org/wiki/Concurrent_computing (žiūrėta 2009).

[15] Lygiagretūs procesai http://en.wikipedia.org/wiki/Concurrency_%28computer_science%29 (žiūrėta 2009).

[16] D. Hart, E.Kraemer and G-C.Roman “Interactive Visual Exploration of Distributed Computations”. In Proceedings of 11th International Parallel Processing Symposium, 1997.

[17] C.D.Hundhausen, S.A. Douglas, J.T.Stasko. A Meta-Study of Algorithm Visualization Effectiveness, 2001

[18] R.F.Erbacher. Visual debugging of data and operations for concurrent programs. SPIE '97 Conference on Visual Data Ecploration, 1997.

[19] R.F.Erbacher. Visual Assistance for Concurrent Processing, 2000

[20] V. Šeinauskas. Lygiagrečių programų efektyvumo tyrimo programinė įranga. Magistro darbas, 2008

[21] N. Cibulskytė. Grafinė sistema lygiagrečioms programoms stebėti. Magistro darbas, 2010

8. TERMINŲ IR SANTRUMPŲ ŽODYNAS

- boost biblioteka – C++ kalbos biblioteka
- OpenMP – lygiagrečių programų kūrimo sąsaja
- Linux – Unix tipo operacinė sistema naudojanti Linux branduolį
- GCC - LINUX sistemos C\C++ kalbos kompiliatorius
- QT – Aplikacijų ir vartotojo sąsajos karkasas (angl. „cross-platform application and UI framework“)
- UML – unifikuota modeliavimo kalba (angl. „Unified modelling Language“)
- GCC XML - C++ kalbos nagrinėtojas (angl. „parser“)
- VISTOP – vizualizacijos įrankis lygiagrečioms sistemoms (angl. „Visualization Tool for Parallel Systems“)
- UML – unifikuota modeliavimo kalba (angl. „Unified Modelling Language“)

9. PRIEDAI

1 priedas.

Netinkamas „Vartotojo-gamintojo“ problemos sprendimas

```
int itemCount;
procedure producer() {
    while (true) {
        item = produceItem();

        if (itemCount == BUFFER_SIZE) {
            sleep();
        }

        putItemIntoBuffer(item);
        itemCount = itemCount + 1;

        if (itemCount == 1) {
            wakeup(consumer);
        }
    }
}

procedure consumer() {
    while (true) {

        if (itemCount == 0) {
            sleep();
        }

        item = removeItemFromBuffer();
        itemCount = itemCount - 1;

        if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer);
        }

        consumeItem(item);
    }
}
```

„Vartotojo-gamintojo“ problemos sprendimas, panaudojant semaforus

```
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

procedure producer() {
    while (true) {
        item = produceItem();
        down(emptyCount);
            putItemIntoBuffer(item);
        up(fillCount);
    }
}

procedure consumer() {
    while (true) {
        down(fillCount);
            item = removeItemFromBuffer();
        up(emptyCount);
         consumeItem(item);
    }
}
```

2 priedas.
Rūkoriaus problemos sprendimas.

```
//--------------------------------------------------------------------------
// Filename:
//    Smoker.h
// PROGRAM DESCRIPTION
//    class definition file for Smokerthread and Agentthread class
//--------------------------------------------------------------------------

#ifndef _SMOKER_H
#define _SMOKER_H

#include "ThreadClass.h"

#define NUM_OF_SMOKERS      3        // number of Smoker

//--------------------------------------------------------------------------
// Smokerthread class definition
//--------------------------------------------------------------------------

class Smokerthread: public Thread
{
     public:
          Smokerthread(int Number);  // constructor
     private:
          void ThreadFunc();
          int No;
};

//--------------------------------------------------------------------------
// Agentthread class definition
//--------------------------------------------------------------------------

class Agentthread: public Thread
{
     public:
          Agentthread(char *ThreadName, int iter);  // constructor
     private:
          void ThreadFunc();
          int Iterations;
};

#endif
//--------------------------------------------------------------------------
// Filename:
//    smoker.cpp
// PROGRAM DESCRIPTION
//    Implementation of smoker threads and agent thread
//--------------------------------------------------------------------------

#include <iostream>
#include <stdlib.h>
#include <time.h>

#include "smoker.h"

// local macro defines
```

```
#define MATCH_PAPER         0
#define PAPER_TOBACCO       1
#define TOBACCO_MATCH       2

// global data variables

static char *materials[]={ "paper", "match", "tobacco"};
static char *names[]  =  { "Smoker(Tobacco)", "Smoker(Paper)",
"Smoker(Match)"};

// semaphores
static Semaphore PaperMatch("PaperMatch");
static Semaphore MatchTobacco("MatchTobacco");
static Semaphore TobaccoPaper("TobaccoPaper");
static Semaphore *Sem[NUM_OF_SMOKERS] = {&PaperMatch, &MatchTobacco,
&TobaccoPaper};
static Semaphore Table("Table", 0);// control the sharing of the table

// ---------------------------------------------------------------------
// FUNCTION  Filler():
//    This function fills a strstream with spaces.
// ---------------------------------------------------------------------

strstream *Filler(int n)
{
     int  i;
     strstream *Space;

     Space = new strstream;
     for (i = 0; i < n; i++)
        (*Space) << ' ';
     (*Space) << '\0';
     return Space;
}

//----------------------------------------------------------------------
// Smokerthread::Smokerthread()
//      constructor for Smokerthread class
//----------------------------------------------------------------------

Smokerthread::Smokerthread(int Number)
{
     No = Number;
     ThreadName.seekp(0, ios::beg);
     if (Number == 0)
        ThreadName << "Smoker(Tobacco)" << '\0';
     else if (Number == 1)
          ThreadName << "Smoker(Paper)" << '\0';
     else
          ThreadName << "Smoker(Match)" << '\0';
}

//----------------------------------------------------------------------
// Smokerthread::ThreadFunc()
//      implement a smoker thread
//----------------------------------------------------------------------

void Smokerthread::ThreadFunc()
{
     Thread::ThreadFunc();
     strstream *Space;
```

```
     Space = Filler(4);            // build leading spaces
     while (1) {
          Sem[No]->Wait();         // smoker waits for materials
          cout << Space->str() << ThreadName.str() << " is smoking" <<
endl;
          Table.Signal();          // take ingredients and release the
table
          Delay();                 // smoking for a while
     }
}

//--------------------------------------------------------------------------
// Agentthread::Agentthread()
//      constructor for Agentthread class
//--------------------------------------------------------------------------

Agentthread::Agentthread(char *ThreadName, int iter)
            : Thread(ThreadName)
{
     Iterations = iter;
     srand((unsigned int) time(NULL));
}

//--------------------------------------------------------------------------
// Agentthread::ThreadFunc()
//      implement a agent thread
//--------------------------------------------------------------------------

void Agentthread::ThreadFunc()
{
     Thread::ThreadFunc();
     int RandomNo;

     for (int i = 0; i < Iterations; i++) {
          Delay();
          RandomNo = rand() % NUM_OF_SMOKERS;
          cout << "Agent prepares " << materials[RandomNo] << " and "
               << materials[(RandomNo+1) % NUM_OF_SMOKERS] << endl;
          Sem[RandomNo]->Signal();        // randomly provide material to
one smoker
          Table.Wait();                   // waiting for smoker finish
smoking
          cout << "Agent finishes serving "  << names[RandomNo] << endl;
     }
     cout << "Agent quits." << endl;
     Exit();
}

// end of smoker.cpp
//--------------------------------------------------------------------------
// Filename:
//     Smoker-main.cpp
// PROGRAM DESCRIPTION
//     This program use the semaphore to solve smoker problem
//--------------------------------------------------------------------------

#include <iostream>
#include <stdlib.h>

#include "smoker.h"

//--------------------------------------------------------------------------
```

```
// main() function
//--------------------------------------------------------------------------

int main(int argc, char *argv[])
{
     Smokerthread SmokerTobacco(0);  // smoker thread with "tabacco"
     Smokerthread SmokerMatch(1);    // smoker thread with "match"
     Smokerthread SmokerPaper(2);    // smoker thread with "paper"
     int Iterations;

     if (argc != 2) {
          cout << "Use " << argv[0] << " #-of-iterations of agent." <<
endl;
          exit(0);
     }
     Iterations = abs(atoi(argv[1]));

     // create the agent thread and fire it up
     Agentthread Agent("Agent", Iterations);
     Agent.Begin();

     // start smoker threads
     SmokerTobacco.Begin();
     SmokerMatch.Begin();
     SmokerPaper.Begin();

     // wait for all child threads
     Agent.Join();
     cout << "Game is over." << endl;
     Exit();

     return 0;
}
```

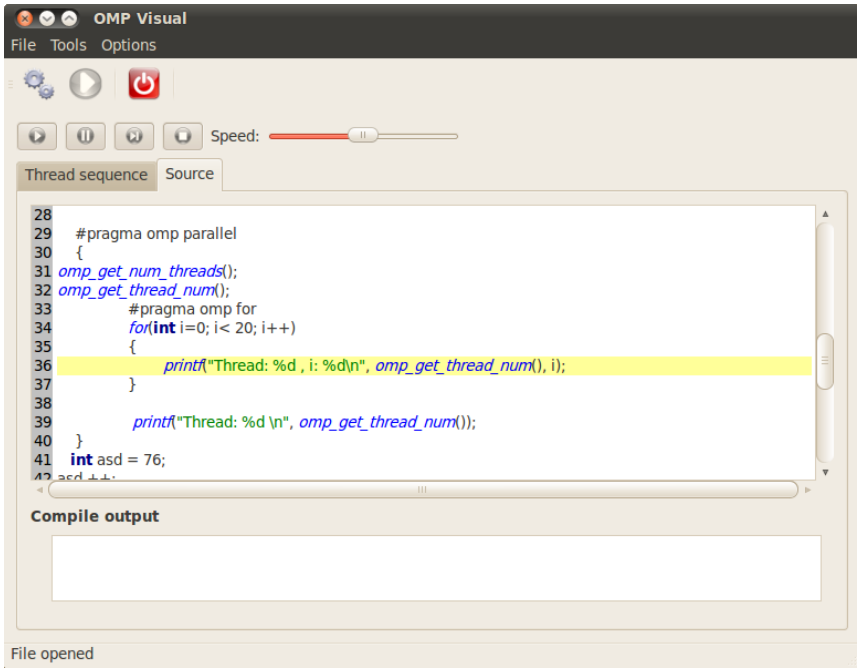
3 priedas.

Vartotojo atmintinė

Dirbti su sistema galima iš karto, joks diegimas ar prisijungimas prie sistemos nereikalingas, tereikia paleisti programos vykdomąjį failą, įvesti (arba atsidaryti jau parašytą) programos kodą, jį išsaugoti, jei jis dar nėra išsaugotas, sukompiliuoti suvesti reikalingus stebėjimui kintamuosius ir paleisti vizualizaciją. Vizualizacijos metu galima keisti vizualizavimo greitį, stabdyti ir pratęsti vizualizavimo procesą, bei jį nutraukti.

Programos kodo įvedimas

Primas žingsnis prieš pradedant dirbti su sistema yra programos kodo įvedimas. Programos kodo redaktorius pavaizduotas 34 paveikslėlyje, kuris paryškina kai kuriuos c++ kalbos raktinius žodžius, taip pat komentarus. Programa gali būti įvesta ir išsaugota arba tiesiog atidaryta jau parašyta programa per pagrindinį meniu (File -> Open).



34 pav. Programos kodo redaktorius

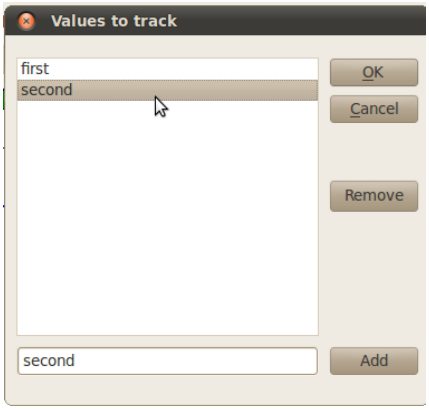
Programos kompiliavimas

Tam, kad programą galėtume vykdyti reikia ją sukompiliuoti. Sistema turi integruotą g++ kompiliatorių, todėl vartotojui yra lengva šį tikslą pasiekti, tereikia paspausti „Compile“ mygtuką, kuris vartotojo sąsajoje yra atvaizduotas paveikslėliu, kuriame matomi krumpliaračiai, 34 paveikslėlyje galime jį matyti po meniu juosta kairėje pusėje. Apie kompiliavimo klaidą arba sėmę bus pranešta vartotojui. Jei kompiliavimas nepavyks, kompiliatoriaus klaidas galima bus pamatyti „Compile output“ lange.

Stebimų kintamųjų suvedimas

Vizualizacijos metu yra suteikta galimybe vartotojui stebėti vartotojui svarbių kintamųjų reikšmes. Vartotojas jam svarbius kintamuosius gali suvesti atidaręs langą per pagrindinį meniu „Tools-> Variables to track“, iškvietus šią meniu funkciją pamatome kintamųjų suvedimo langą (35 pav.), jame yra išpildytos šios funkcijos:

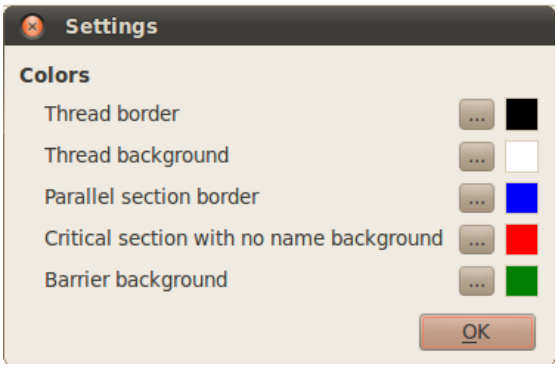
- Pridėti naują kintamąjį
- Pašalinti kintamąjį
- Atšaukti pakeitimus
- Priimti pakeitimus



35 pav. Kintamųjų suvedimo langas

Spalvų redagavimas

Vartotojo patogumui yra sukurta rei6minių vizualizacijos spalvų keitimo funkcija, pasiekiama per meniu „Options->Color settings“ (36 pav.). Ši funkcija leidžia nustatyti kokiomis spalvomis sistema žymės veikiančias gijas, jų buvimą kritinėse sekcijose, laukimą ties barjeru (angl. „barrier“, vieta, kurią turi pasiekti visos gijos, kad būtų vykdomas tolimesnis programos kodas) ir t.t..

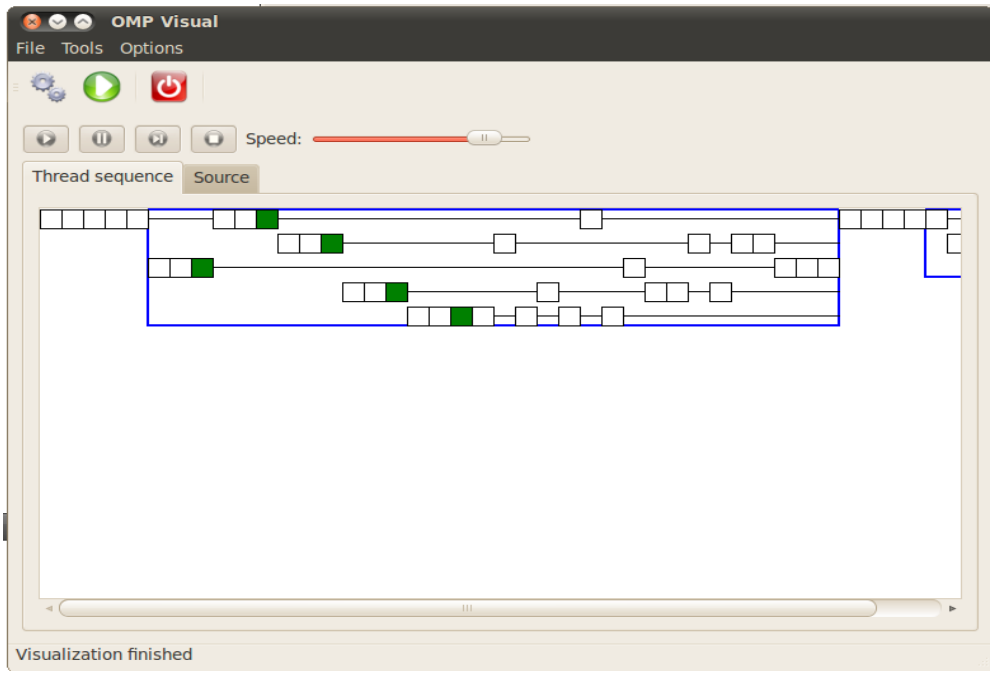


36 pav. Spalvų valdymas

Lygiagrečios programos vizualizacija

Suvedus programos kodą, bei pageidaujamus papildomus nustatymus (kintamuosius, spalvas), ją sukompiliavus ir paspaudus „Run“ galime stebėti kaip veikia mūsų parašyta programa (37 pav.). Vizualizacijos metu galime vykdyti interaktyvias sistemos funkcijas, kurios įtakoja vizualizavimo procesą:

- Vizualizacijos greitis
- Laikinas sustabdymas (pause)
- Pažingsninis veikimas (viena gija įvykdo tik vieną kodo eilutę)
- Vizualizacijos nutraukimas
- Vizualizacijos tęsimas



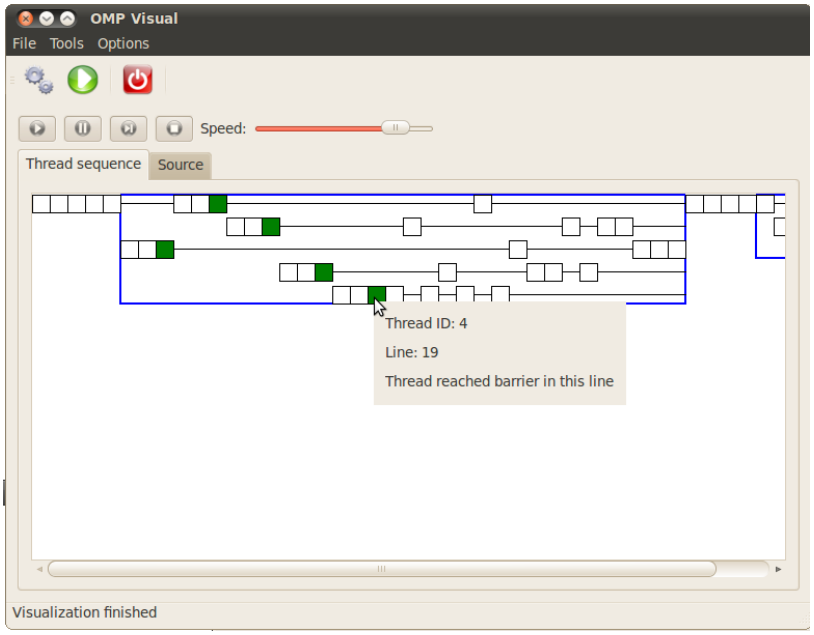
37 pav. Vizualizacija

Informacijos apie gijas stebėjimas

Vykstant vizualizacijos procesui ar jai baigus darbą, galime stebėti informaciją apie gijų veikimą, ją apima:

- Gijos numeris
- Kodo eilutės numeris
- Kintamųjų sąrašas
- Kritinių sekcijų ir kitų reikšmingų įvykių informaciją

Šis funkcionalumas pasiekiamas tiesiog užvedus pelės žymeklį ant reikiamos gijos veikimo vietos (kvadratėlio), tai yra atvaizduota paveikslėlyje nr. 38.



38 pav. Informacijos apie giją stebėjimas

4 priedas.
Gamintotjo-vartotojo problemą modeliuojančios programos

Programa su „klaidomis“

```
#include<omp.h>
#include<stdio.h>
#include <semaphore.h>
const int bufferSize = 3;
int buffer[bufferSize];
sem_t fillCount;// = 0;
sem_t emptyCount;// = 10;
int main()
{
sem_init(&fillCount, 1, 0);
sem_init(&emptyCount, 1, bufferSize);
omp_set_num_threads(3);
#pragma omp parallel
{
if(omp_get_thread_num() == 0) // producer
{
for(int i = 0; i < 10; i++)
{
sem_wait(&emptyCount);
int val;
sem_getvalue(&fillCount, &val);
buffer[val] = 1;
sem_post(&fillCount);
}
}
```

```
else // consumer
{
for(int i = 0; i < 3; i++)
{
sem_wait(&fillCount);
int val;
sem_getvalue(&fillCount, &val);
buffer[val] = 0;
sem_post(&emptyCount);
}
}
#pragma omp barrier
}
return(0);
}
```

Tinkamai įgyvendinta programa

```
#include<omp.h>
#include<stdio.h>
#include <semaphore.h>
const int bufferSize = 1;
int buffer[bufferSize];
sem_t fillCount;// = 0;
sem_t emptyCount;// = 10;
int main()
{
sem_init(&fillCount, 1, 0);
sem_init(&emptyCount, 1, bufferSize);
omp_set_num_threads(2);
#pragma omp parallel
{
if(omp_get_thread_num() == 0) // producer
{
for(int i = 0; i < 3; i++)
{
sem_wait(&emptyCount);
int val;
sem_getvalue(&fillCount, &val);
buffer[val] = 1;
sem_post(&fillCount);
}
}
else // consumer
{
for(int i = 0; i < 3; i++)
{
sem_wait(&fillCount);
int val;
sem_getvalue(&fillCount, &val);
buffer[val] = 0;
sem_post(&emptyCount);
}
}
#pragma omp barrier
```

```
}
return(0);
}
```

5 priedas.
Rašytojo-skaitytojo problemą modeliuojančios programos

1. programa

```
#include<omp.h>
#include<stdio.h>
int sharedResource = 0;
omp_lock_t my_lock;
int main()
{
omp_init_lock(&my_lock);
omp_set_num_threads(3);
#pragma omp parallel
{
if(omp_get_thread_num() == 0) // reader
{
for(int i = 0; i < 3; i++)
{
omp_set_lock(&my_lock);
int i = sharedResource;
omp_unset_lock(&my_lock);
}
}
else // writer
{
for(int i = 0; i < 3; i++)
{
omp_set_lock(&my_lock);
sharedResource = omp_get_thread_num();
omp_unset_lock(&my_lock);
}
}
#pragma omp barrier
}
return(0);
}
```

2. programa

```
#include<omp.h>
#include<stdio.h>
int sharedResource = 0;
int main()
{
omp_set_num_threads(3);
#pragma omp parallel
{
if(omp_get_thread_num() == 0) // reader
{
for(int i = 0; i < 3; i++)
```

```
{
#pragma omp critical
{
int i = sharedResource;
}
}
}
else // writer
{
for(int i = 0; i < 3; i++)
{
#pragma omp critical
{
sharedResource = omp_get_thread_num();
}
}
}
#pragma omp barrier
}
return(0);
}
```

3. programa

```
#include<omp.h>
#include<stdio.h>
int sharedResource = 0;
int main()
{
omp_set_num_threads(3);
#pragma omp parallel
{
if(omp_get_thread_num() == 0) // reader
{
for(int i = 0; i < 3; i++)
{
#pragma omp critical (K1)
{
int i = sharedResource;
}
}
}
else // writer
{
for(int i = 0; i < 3; i++)
{
#pragma omp critical (K2)
{
sharedResource = omp_get_thread_num();
}
}
}
#pragma omp barrier
}
return(0);
}
```

6 priedas.
Rūkoriaus problemą modeliuojančios programos

1. programa

```
#include<omp.h>
#include<stdio.h>
#include <semaphore.h>
#include <stdlib.h>
#include <iostream>
sem_t tobacco;
sem_t paper;
sem_t matches;
bool isEnd = false;
sem_t* ingredients[3];
int main()
{
sem_init(&tobacco, 1, 0);
sem_init(&paper, 1, 0);
sem_init(&matches, 1, 0);
ingredients[0] = &tobacco; ingredients[1] = &paper; ingredients[2] =
&matches;
omp_set_num_threads(4);
#pragma omp parallel
{
switch(omp_get_thread_num())
{
case 0 : // agent
{
for(int i = 0; i < 1; i++)
{
while(true){
int val1, val2;
int i1 = random() % 3; sem_getvalue(ingredients[i1], &val1);
int i2 = (i1 + 1) % 3; sem_getvalue(ingredients[i2], &val2);
if(val1 == 0 && val2 == 0)
{
sem_post(ingredients[i1]);sem_post(ingredients[i2]); break;
}}
}
isEnd = true;
}; break;
case 1 : // paper holder
{
while(!isEnd)
{
int val1, val2;
sem_getvalue(&tobacco, &val1); sem_getvalue(&matches, &val2);
if(val1 && val2){
sem_wait(&tobacco);
sem_wait(&matches); usleep(1000);}
}
}; break;
case 2 : // tobacco holder
```

```
{
while(!isEnd)
{
int val1, val2;
sem_getvalue(&paper, &val1); sem_getvalue(&matches, &val2);
if(val1 && val2){
sem_wait(&paper);
sem_wait(&matches); usleep(1000);}
}
}; break;
case 3 : // match holder
{
while(!isEnd)
{
int val1, val2;
sem_getvalue(&tobacco, &val1); sem_getvalue(&paper, &val2);
if(val1 && val2){
sem_wait(&tobacco);
sem_wait(&paper); usleep(1000);}
}
}; break;
};
#pragma omp barrier
}
return(0);
```

2. programa

```
#include<omp.h>
#include<stdio.h>
#include <semaphore.h>
#include <stdlib.h>
#include <iostream>
sem_t tobaccopaper;
sem_t papermatcher;
sem_t matchestobacco;
bool isEnd = false;
sem_t* ingredients[3];
int main()
{
sem_init(&tobaccopaper, 1, 0);
sem_init(&papermatcher, 1, 0);
sem_init(&matchestobacco, 1, 0);
ingredients[0] = &tobaccopaper; ingredients[1] = &papermatcher;
ingredients[2] = &matchestobacco;
omp_set_num_threads(4);
#pragma omp parallel
{
switch(omp_get_thread_num())
{
case 0 : // agent
{
for(int i = 0; i < 3; i++)
{
while(true){
int val;
```

```
int i1 = random() % 3; sem_getvalue(ingredients[i1], &val);
if(val == 0)
{
sem_post(ingredients[i1]); break;
}}
}
isEnd = true;
}; break;
case 1 : // paper holder
{
while(!isEnd)
{
int val; sem_getvalue(&matchestobacco, &val);
if(val){
sem_wait(&matchestobacco); usleep(1000);}
}
}; break;
case 2 : // tobacco holder
{
while(!isEnd)
{
int val; sem_getvalue(&papermatcher, &val);
if(val){
sem_wait(&papermatcher); usleep(1000);}
}
}; break;
case 3 : // match holder
{
while(!isEnd)
{
int val; sem_getvalue(&tobaccopaper, &val);
if(val){
sem_wait(&tobaccopaper); usleep(1000);}
}
}; break;
};
#pragma omp barrier
}
return(0);
```

7 priedas
Pietaujančių filosofų problemą modeliuojančios programos

1. programa

```
#include<omp.h>
#include<stdio.h>
#include <semaphore.h>
#include <stdlib.h>
#include <iostream>
sem_t forks[5];
int main()
{
for(int i= 0; i < 5; i++) sem_init(&forks[i], 1, 1);
omp_set_num_threads(5);
```

```
#pragma omp parallel
{
int threadId = omp_get_thread_num();
for(int i = 0; i < 10; i++){
sem_wait(&forks[threadId%5]);
sem_wait(&forks[(threadId + 1)%5]);
usleep(1000); // eat
sem_post(&forks[threadId%5]);
sem_post(&forks[(threadId + 1)%5]);
usleep(1000); // think
}
#pragma omp barrier
}
return(0);
}
```

2. programa

```
#include<omp.h>
#include<stdio.h>
#include <semaphore.h>
#include <stdlib.h>
#include <iostream>
sem_t forks[5];
int main()
{
for(int i= 0; i < 5; i++) sem_init(&forks[i], 1, 1);
omp_set_num_threads(5);
#pragma omp parallel
{
int threadId = omp_get_thread_num();
for(int i = 0; i < 3; i++){
#pragma omp critical
{
sem_wait(&forks[threadId%5]);
sem_wait(&forks[(threadId + 1)%5]);
usleep(1000); // eat
sem_post(&forks[threadId%5]);
sem_post(&forks[(threadId + 1)%5]);
}
usleep(1000); // think
}
#pragma omp barrier
}
return(0);
}
```

3. programa

```
#include<omp.h>
#include<stdio.h>
#include <semaphore.h>
#include <stdlib.h>
#include <iostream>
sem_t forks[5];
sem_t table;
int main()
```

```
{
for(int i= 0; i < 5; i++) sem_init(&forks[i], 1, 1);
sem_init(&table, 1, 3);
omp_set_num_threads(5);
#pragma omp parallel
{
int threadId = omp_get_thread_num();
for(int i = 0; i < 3; i++){
sem_wait(&table);
{
sem_wait(&forks[threadId%5]);
sem_wait(&forks[(threadId + 1)%5]);
usleep(1000); // eat
sem_post(&forks[threadId%5]);
sem_post(&forks[(threadId + 1)%5]);
}
sem_post(&table);
usleep(1000); // think
}
#pragma omp barrier
}
return(0);
}
```