KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

PROGRAMŲ INŽINERIJOS KATEDRA

Nida Cibulskytė

Grafinė sistema lygiagrečioms programoms stebėti

Magistro darbas

Darbo vadovas

doc. R.Marcinkevičius

Kaunas, 2010

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

PROGRAMŲ INŽINERIJOS KATEDRA

Nida Cibulskytė

Grafinė sistema lygiagrečioms programoms stebėti

Magistro darbas

Kalbos konsultantė

doc. J. Mikelionienė

2010 05

Vadovas

doc. R. Marcinkevičius

2010 05

Atliko

IFM 4/2 grupės stud.

Nida Cibulskytė

2010 05 29

Recenzentas

doc. S. Maciulevičius

2010 05

Kaunas, 2010

TURINYS

LENTELIŲ SĄRAŠAS

PAVEIKSLŲ SĄRAŠAS

SANTRAUKA ANGLŲ KALBA

SUMMARY

This research states problems of parallel programming and tools specialized for such programs. The review of existing visualization and analysis tools is made. These tools are compared and evaluated with respect to introduced criteria. After summarizing the result of analysis, the new solution is proposed. The new tool provides the functionality of real time profiling, visualization of thread synchronization, representation of the program structure and performance metrics. Additionally, it gives the information of execution platform. The analysis of the results, provided by this tool, leads to faster identification and correction of synchronization problems, gives deeper knowledge of the program structure, which can be helpfull in preventing the problems. This paper describes the major architecture aspects and implementation techniques. The functionality of tool is described and demonstrated. Two clasical synchronization problems – dining philosopher and producer-consumer – are reviewed and simulating programs are constructed. These programs are tested and analysed using new tool. These tests show how to identify the problems of parallel programs.

1. ĮŽANGA

Lygiagrečių programų naudojimas apima vis daugiau gyvenimo sričių. Lygiagrečios programos sudėtingėja, auga jų apimtis. Dėl tokių programų sudėtingumo sunku kurti efektyvias programas, kurios pilnai išnaudotų lygiagretaus programavimo teikiamas galimybes bei pilnai panaudotų turimą techninę bazę. Todėl vis daugiau dėmesio skiriama problemų, kylančių kuriant tokias programas, sprendimui.

Lygiagrečios programos susideda iš keleto dažniausiai nepriklausomų sąveikaujančių dalių, gijų, veikiančių tuo pat metu [1]. Lygiagrečios programos vykdymas gali būti traktuojamas kaip individualių gijų vykdomų atominių veiksmų rinkinys. Tokių progamų teisingumo įvertinimas yra daug sudėtingesnis, padidėja matavimo charakteristikų kiekis. Taigi kuriant lygiagrečias programas susiduriama su visiškai kitokio pobūdžio problematika.

Programos derinimas prasideda kodo lygyje, kuris vėliau sukompiliuojamas ir vykdomas. Programa paleidžiama, užfiksuojami programos vykdymo seka, parametrai bei charakteristikos, kurie vėliau gali būti peržiūrimi ir analizuojami pasirinktame įrankyje.[4] Tačiau toks būdas reikalauja išankstinės surinktų duomenų peržiūros, jų surūšiavimo bei dominančių parametrų pasižymėjimo. Dėl pakankamai didelio sudėtingumo trasavimo bylų apimtis gali būti didėlė, todėl jų analizavimas yra sudėtingas bei perteklinis. Be to net ir suradus silpniausias programos vietas tenka vėl kartoti programos vykdymą ir iš naujo analizuoti gautus rezultatus. Todėl kartais net akivaizdžios klaidos ar neefektyvumo suradimas gali tapti varginančiu procesu.

Tradiciniai nuoseklių programų testavimo būdai yra sunkiai pritaikomi lygiagrečioms programoms, kadangi tokie testai padengia tik labai mažą dalį visų lygiagrečios programos vykdymo kelių. Taip yra dėl nedeterministinio lygiagrečių programų veikimo : programa su tokiais pačiais pradiniais duomenimis gali duoti vis kitokius rezultatus[9]. Taigi įėjimo duomenų keitimas nepakankamai kontroliuoja programos elgesį. Todėl net ir suradus potencialias klaidas bei pakartojus tuos pačius testus nebūtinai gaunamas toks pat programos veikimas, taigi egzistuojančios problemos nebūtinai bus išspręstos.

Nepaisant vis dažniau iškylančių problemų, kūrimo, trasavimo bei stebėjimo įrankiai specializuoti lygiagrečioms programoms vis dar nėra pakankamai išvystyti. Dauguma trasavimo ir stebėjimo įrankių remiasi statine programinio kodo analize arba vykdymo metu suformuotų rezultatų bylų analize. Įrankiai skirti programos veikimo stebėjimui realiu laiku paprastai yra menkai pritaikyti lygiagrečioms programoms, todėl galime gauti tik tekstinę informaciją apie

veikiančias gijas. Be to ši informacija yra perteklinė ir neduoda bendro programos struktūros suvokimo, nenurodo jos atliekamų veiksmų sekos ar sąveikos su kitomis gijomis.

Šio darbo tikslas – supažindinti su lygiagretaus programavimo principais, apibrėžti kylančias problemas, išanalizuoti egzistuojančius įrankius pagal įvestus kriterijus. Taip pat ištirti darbo metu buvo suprojektuotą ir realizuotą naują lygiagrečių programų stebėjimo įrankį.

Darbo struktūra:

1. Analizės dalyje aprašomas kontekstas, pateikiami bendri lygiagretaus programavimo principai, detalizuojamos lygiagretaus programavimo problemos, įvedami vertinimo kriterijai. Išanalizuojami bei pagal įsivestus kriterijus įvertinami egzistuojantys lygiagrečių programų stebėjimo produktai.
2. Projektinėje dalyje pateikiami bei pagrindžiami sukurto lygiagrečių programų stebėjimo įrankio esminiai projektavimo sprendimai.
3. Tyrimo dalyje aprašoma pietaujančių filosofų problema, apibrėžiama, sąsaja su lygiagretaus programavimo problemomis, sudaroma problemą simuliuojanti tipinė programa, panaudojant sinchronizacijos mechanizmus.
4. Eksperimentinėje dalyje pasinaudojus sukurtu įrankiu tiriama tyrimo dalyje sudaryta programa, aprašoma įrankio panaudojimo principai bei pademonstruojama , kaip aptikti lygiagretumo klaidas.
5. Darbo pabaigoje pateikiamos išvados, siūlomos tolimesnės sukurto įrankio tobulinimo kryptys.

2. ANALITINĖ DALIS

2.1. Tyrimo sritis ir problemos

Lygiagrečių programų bei skaičiavimų naudojimas nuolat auga. Tai įtakoja tiek techninių, tiek programinių priemonių vystymas. Sparčiai auga daugiabranduolinių procesorių naudojimas, todėl lygiagrečios programos tampa vis svarbesnės. Tačiau įrankiai, specializuoti tokių programų analizei ir testavimui nėra stipriai išvystyti.

Egzistuojančius įrankius galime suskirstyti į tris kategorijas:

- įrankiai specializuoti nuoseklioms programoms;
- įrankiai statiškai analizuojantys išeities kodus bei padedantys surasti potencialias klaidas;
- įrankiai formuojantys trasavimo bylas, kurios vėliau gali būti analizuojami atitinkamame įrankyje.

Daugumoje atvejų aprašyti įrankiai nurodo potencialias klaidas. Tačiau vien tik klaidos egzistavimo faktas, be priežastinių ryšių yra menkavertis. Jeigu įrankis neparodo lygiagrečių mechanizmų, veikiančių programoje, iš esmės tampa neįmanoma suvokti sinchronizacijos klaidų priežaščių bei galimų pasekmių ir neretai tokios klaidos būna ištaisomos neteisingai. Pavyzdžiui sinchronizacijos klaidos gali būti ištaisomos panaikinant lygiagretumą, bet tai iš esmės pakeičia programos principą.

Analizuojant trasavimo bylas, atliekamas didelis perteklinis darbas. Be to dažnai neįmanoma aiškiai nustatyti, ką konkreti gija darė konkrečiu metu. Taigi nustatyti gijų sinchronizaciją bei sinchronizacijos klaidas be papildomų priemonių praktiškai neįmanoma. Be to net ir nustačius klaidą jos pakartojimas gali būti sunkus proceas. Taip yra dėl to, kad lygiagrečių programų veikimas yra nederministinis. Dėl šios priežasties nuoseklioms programoms taikomas pakartojijmo metodas taip pat nėra tinkamas.

2.2. Tikslas ir uždaviniai

Tyrimo tikslas – ištirti tipinius egzistuojančius lygiagrečioms programoms specializuotus vizualizacijos ir analizės įrankius, atlikti palyginamąją šių įrankių analizę pagal įsivestus kriterijus bei pasiūlyti ir ištirti naują įrankį, skirtą lygiagrečių programų stebėjimui. Eksperimento būdu siekiama ištirti sukurto įrankio funcionalumą bei pademonstruoti įrankio pritaikymą sincrhronizacijos problemų sprendimui.

2.3. Lygiagretaus programavimo principai

Lygiagreti programa gali būti apibrėžiama kaip tuo pat metu veikiančių sąveikaujančių procesų rinkinys. Ši sąveika susidaro dėl dviejų priežasčių:

1. Procesai varžosi dėl bendrų resursų, tokių kaip fiziniai įrenginiai ar duomenys.
2. Procesai sąveikauja tam, kad apsikeistų duomenimis.

Abiem atvejais svarbu, kad abu procesai veiktų sinchronizuotai bei nekiltų konfliktinių situacijų, bandant pasinaudoti bendrais resursais ar duomenimis. Programos dalis, kurioje atliekami veiksmai su bendrai naudojamais resursais vadinama *kritine sekcija*. Teisingas lygiagrečios programos vykdymas pasiekiamas į kritinę sekciją įleidžiant tik vieną procesą vienu metu arba užlaikant procesų vykdymą, kol programos būsena tenkina tam tikrą sąlygą. Pirma sinchronizacijos forma vadinama *tarpusavio išskyrimu*, o antra *sąlygine sinchronizacija,* kuri užtikrina tarpusavio išskyrimo reikalavimą. [4]

Tarpusavio išskyrimas

Tarpusavio išskyrimas padeda užtikrinti naudojimąsi bendrais resursais. Programos dalis, kurioje bandoma pasiekti bendrai prieinamus resursus arba duomenis vadinama *kritine sekcija.* Tarpusavio išskyrimas nenurodo, kuri gija gali pasiekti kritinę sekciją pirma, jis tik užtikrina, kad kritinė sekcija būtų prieinama vienai gijai vienu metu. Tik viena gija gali būti savo kritinėje sekcijoje, visos kitos gijos, norinčios į ją patekti yra blokuojamos, kol darbą baigs kritinėje sekcijoje esanti gija.

Tarpusavio išskyrimo samprata turi tenkinti sekančias savybes:

1. Daugiausia viena gija vienu metu gali vykdyti savo kritinę sekciją.
2. Aklavietės vengimas. Jeigu du ar daugiau gijų bando įeiti į kritinę sekciją, bent vienam turi tai pavykti.
3. Nebūtino užlaikymo vengimas. Jei gija bando įeiti į savo kritinę sekciją, o kitos gijos vykdo ne kritines sekcijas arba yra nutraukti, tuomet gijai leidžiama įeiti į kritinę sekciją.
4. Įmanomas įėjimas. Gija bandanti įeiti į kritinę sekciją galiausiai į ją įeis. [8]

Sąlyginė sinchronizacija

Sinchronizacija - tai lygiagrečioje sistemoje veikiančių gijų tarpusavio bendravimo mechanizmas. Sąlyginė sinchronizacija tai sincronizacijos procesas, kuomet viena gija siunčia pranešimą kitai gijai, kai tenkinama tam tikra sąlyga. Pavyzdžiui, jei gija bando nuskaityti simbolį iš buferio, kai jisai yra tuščias, ji turėtų būti laikinai sustabdoma. Buferio gija, gavus simbolį turėtų informuoti laukiančią giją, kad buferis nebetuščias. [8]

2.4. Lygiagretaus programavimo mechanizmai Java programavimo kalboje

Java lygiagretaus darbo mechanizmas realizuotas gijomis, sinchronizacijos mechanizmu bei virtualios mašinos struktūra. Gija iš esmės yra panaši į nuoseklią programą, kadangi ji taip pat turi pradžią, vykdymo seką bei pabaigą, tačiau tai nėra programa, kadangi ji negali būti vykdoma savarankiškai. Java programavimo kalboje šį funkcionalumą padeda pasiekti *Thread* klasė. Šios klasės egzemploriai ir yra gijos. Tik vienas *Thread* klasės objektas gali būti vykdomas viename procesoriuje vienu metu. Tuo tarpu kitos gijos yra įvariose laukimo būsenose.

1 pav. Java gijų būsenos

Java gijos gali būto 6 būsenų:

- *Nnew* – naujai sukurta, nepriskirti jokie resursai.
- *Runnable* – gija paleista, t.y. įvykdytas metodas *start(),* priskirti resursai.
- *Waiting* – gija laukia neribotą laiką, kol kita gija atliks tam tikrą veiksmą.
- *Terminated* – gija, kurios darbas buvo nutrauktas.
- *Blocked* – gija buvo užbluokta, norėdama patekti į kritinę sekciją.
- *Timed_waiting.* – gija laukia nustatytą laiką, kol kita gija atliks tam tikrą veiksmą.

Java gijos naudoja bendrą atmintį[2]. Objektai, kurie yra naudojami dviejų ar daugiau gijų vadinami *sąlyginiais kintamaisiais* ir jų naudojimas turi būti sinchronizuotas. Tam tikslui pasiekti gali būti naudojami Java programavimo kalbos teikiami sinchronizacijos mechanizmai. Detaliau aptarsime keletą iš jų.

2.4.1. Monitoriai

Monitorius – tai struktūrizuotas lygiagretaus programavimo elementas, kuris apima duomenis ir operacijas, kurios veikia kaip vienas modulis. Jis užtikrina, jog tik vienai gijai leidžiama vykdyti monitoriaus kodą vienu metu. Kiekvienas monitorius turi užraktą. Jeigu gija bando naudoti monitorių ir jeigu užraktas yra užimtas tuomet gija turi laukti kol jis bus atlaisvintas arba baigia darbą.

Monitoriaus idėja Java programavimo kalboje gali būti įgyvendinta naudojantis *synchronized* kintamaisiais bei blokais. Java programavimo kalboje monitoriais gali būti bet kokie objektai. Jeigu monitoriaus saugomas kodas yra užimtas, tuomet java gija pereina į laukimo būseną, kol jis bus atlaisvintas. Yra du būdai objekto užrakinimui[4].

1. Kviečiant sinchronizuotą bloką.

   ```
   synchronized (anObject) { ... }
   ```

   Čia užrakinamas obejktas anObject. Užraktas atlaisvinamas tuomet, kai baigiamas kodo vykdymas sinchronizuotame bloke. Jeigu kita gija jau vykdo kodą sinchronizuotame bloke, tuomet gija yra blokuojama kol blokas atsilaisvins.

2. Metodo sinchronizavimas

   ```
   public synchronized void aMethod() { ...  }
   ```

   Metodo sinchronizavimas, tai tas pats kad objekto *this* užrakinimas sinchronizuotame bloke.

2.4.2. Užraktai

Kitas Java sinchronizacijos mechanizmas yra vadinamas užraktu ir yra pasiekiamas naudojantis java.util.concurrent pakete esančia klase *Lock*. Užraktas – tai objektas, saugantis įėjimą į kritinės sekcijos kodo dalį. Sukūrus tam tikrą užraktą jo įgijimas ir atidavimas valdomas metodais *lock()* ir *unlock()*. Kodo dalis tarp šių metodų ir yra kritinė sekcija. Kadangi užraktai yra realizuoti kaip Lock klasės objektai, naudojantis jais galima sukurti kritines sekcijas, kurios persidengia.

1 lentelė Persidengiančios kritinės sekcijos, naudojant užraktus

```
method () {
      A.lock ();
      B.lock ();
      A.unlock ();
      B.unlock

}
```

2.5. Lygiagretaus programavimo problemos

Lygiagrečiame programavime egzistuoja naujo pobūdžio problematika lyginant su nuosekliomis programomis. Lygiagrečios sistemos veikimas yra nederministinis: tokie patys įėjimai gali duoti vis kitokius išėjimus. Taip yra dėl to, kad pati programa neįtakoje atskirų gijų veikimo. Jeigu programos dalis priklauso nuo kelių gijų veikimo sekos, tuomet tokia programa nėra saugi[5].

Egzistuoja kelios tipinės lygiagrečių programų problemos:

- Kova dėl resursų - kelios gijos bando gauti tą patį resursą. Bendrais resurasais gali būti tiek kintamieji, tiek techniniai įrenginiai.
- Aklavietė - gija laukia tokio įvykio iš kitos gijos, kuris niekuomet neįvyks. Tarkime viena gija laiko tam tikrą resursą, kurio reikia kitai gijai. Laikanti gija laukia signalo iš laukiančios, kurtis sunčiamas gavus resursą. Akivaizdu, kad taip niekuomet neįvyks, todėl susidaro aklavietė.

Thread 1

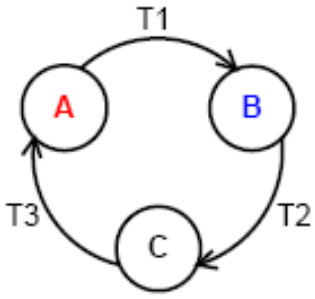
```
synchronized(A) {
synchronized(B) { }
}
```

Thread 2

```
synchronized(B) {
synchronized(C) { }
}
```

Thread 3

```
synchronized(C) {
synchronized(A) { }
}
```



2 pav. Mirties taško pavyzdys

- Užsiciklinimas – jeigu tarp gijų atsiranda nesibaigiantis ciklas. Dažniausia pasitaikantys du tipai: badavimas arba amžinas ciklas. Badavimas susidaro tuomet, kai kitos gijos visą laiką naudoja tam tikrus resursus neleisdamos to padaryti kitoms gijoms. Pavyzdžiui, jeigu turime tam tikri aukšto prioriteto darbai yra vykdomi nuolat ir nepertraukiamai, tai žemesnio pririoteto darbas gali niekada nebūti įvykdytas. Amžinas ciklas gali susidaryti tuomet, kai

atskirų gijų elgesys yra korektiškas, tačiau jų sąveikavimas priverčia programą sukti amžiną ciklą. [3]

Šių problemų aptikimas neturint tinkamų įrankių gali būti sudėtingas ir varginantis procesas. Be to norint aptikti sinchronizacijos klaidas būtina žinoti veiksmų eiliškumą.

2.6. Stebėjimo įrankių vertinimo kriterijai

Rinkoje egzistuoja keletas įrankių, specializuotų lygiagrečių programų stebėjimui. Juos galime suskirstyti į dvi kategorijas – skirti programos sugeneruotų trasavimo failų analizei bei atvaizdavimui ir skirti programos gijų stebėjimui realiu laiku. Tam, kad įvertinti bei palyginti egzistuojančius įrankius įvedami vertinimo kriterijai:

- Programos stebėjimas realiu laiku – šis kriterijus nurodo, ar įrankis leidžia stebėti paleistą programą realiu laiku. Kitaip tariant ar įrankis leidžia stebėti programą jos vykdymo metu.
- Panaudojimo paprastumas – nusako, kaip paprasta panaudoti įrankį egzistuojančiai vartotojo programai, kiek pakeitimų reikia atlikti norint ją stebėti, ar reikalingas išankstinis pasiruošimas, pvz. ar reikalinga iš anksto žinoti tam tikrų žymėjimų specifika.
- Galimybė stebėti atskiras gijas – kriterijus nurodo, ar yra galimybė vartotojui matyti kokius metodus vykdė atskiros gijos, kaip kito jų būsenos, kurios buvo kritinėse sekcijose konkrečiu metu.
- Kovos dėl resursų aptikimas – ar yra galimybė aptikti gijų kovą dėl resursų (apibrėžta 2.4 skyrelyje), ar reikia atlikti daug analizės darbų, norint tai pastebėti.
- Aklavietės aptikimas – kriterijus nurodo, ar yra galimybė aptikti aklavietes(apibrėžta 2.4 skyrelyje) ir ar daug vartotojo pastangų reikia įdėti norint tai padaryti.
- Užsiciklinimų aptikimas – kriterijus nurodo, ar yra galimybė aptikti užsiciklinimus (apibrėžta 2.4 skyrelyje) ir ar daug vartotojo pastangų reikia įdėti norint tai padaryti.
- Rezultatų pernešamumas – ar yra galimybė gautus rezultatus pernešti į kitą kompiuterį, ar nesudėtingas išsaugotų duomenų importavimas, ar rezultatų failai nėra per didelės apimties.
- Konfigūravimo paprastumas – ar yra galimybė konfigūruoti įrankio parametrus iš grafinės sąsajos, ar yra galimybė parinkti klases, kurių stebėti nereikia, ar yra galimybė prisitaikyti grafinę sąsają pagal savo norus, t.y. keisti spalvas, matavimo vienetus.
- Rezultatų išsamumas – šis kriterijus nusako, kaip išsamiai pateikiami rezultatai, ar iš jų matosi programos struktūra, kvietimų skaičius, vykdymo laikas.

- Nepriklausomumas nuo aplinkos – kriterijus nurodo, kaip stipriai įrankis priklauso nuo vykdymo aplinkos, t.y. ar norint panaudoti įrankį reikia įdėti daug paruošiamųjų administracinių darbų, t.y. ar būtina konkreti IDE, ar būtina konkreti virtualios mašinos versija ir pan.

2.7. Egzistuojantys įrankiai

Šiame skyrelyje aptarsime keturis egzistuojančius lygiagrečių programų įrankius skirtus dirbti su Java programavimo kalba parašytomis programomis. Kiekvienam įrankui suformuluotas bendras apibūdinimas, išskiriamos esminės įrankio funkcijos, aprašytos galimybės bei būdai problemoms spręsti. Akcentuojama, kaip kiekviekvienas įrankis sprendžia lygiagretaus programavimo problemas, trumpai aprašomas naudojamas algoritmas tokių problemų aptikimui ir sprendimui.

I. Java ConTest

JavaConTest – tai galingas testavimo įrankis skirtas Eclipse integracinei aplinkai. Jis leidžia testuoti sistemas, dirbančias keliuose procesoriuose arba sudarytas iš kelių gijų. Testavimas užfiksuojant duomenis, o baigus darbą juos analizuojant. Pasižymi tokiomis savybėmis[13]:

- Kodo aprėpties – nurodo, kokia kodo dalis buvo padengta.
- Lygiagretumo aprėpties – nurodo, kokia dalis lygiagretumo buvo padengta.
- Aklavietės vengimas – nurodo aklavietės galimybes, net jeigu jis ir neįvyko.
- Trasavimo rinkinys – jeigu programa neleistinai baigia darbą, nurodo potencialias klaidas.

Analizuojant aklavietės vietas galimi du ataskaitų tipai:

1. **Užraktų būsenų** – kuri gija, kuriuo metu laiko užraktą.
2. **Gijos paskutinio buvimo vieta** – kodo vieta, kurią vykdo kiekviena gija

JavaConTest turi papildomą Orange Box įrankį. Jis naudojamas tuomet, kai programa neleistinai baigia darbą. Čia standartiškai fiksuojamos gijų dvi paskutinės vykdymo vietos, tačiau šį dydį galima keisti.

JavaConTest įrankyje aklavietės aptinkamos naudojantis trasavimo rinkiniu. Jis gali fiksuoti realias aklavietės vietas, stebėti nurodytų kintamųjų reikšmių kitimą vykdymo metu. Aptinka aklavietes tik joms įvykus, t.y. kai programa neleistinai baigia darbą. Tuomet analizuojant paskutines gijos veikimo vietas bei kintamųjų reikšmes nustatoma ar programos neleistinas

užbaigimas buvo įtakotas aklavietės. Java ConTest rezultatai pateikiami tekstiniu formatu, todėl įmanoma juos pernešti. [13]

II. Visual Threads

Įrankis skirtas su lygiagretumo susijusių klaidų aptikimui, gijų būsenų atvaizdavimui. Skirtas Java programavimo kalba parašyto kodo analizei. Programa stebima realiu laiku, t.y. jai veikiant. Nepririštas prie jokios integracinės aplinkos. Automatinis klaidų aptikimas gali būti suskirstytas į tokias kategorijas: aklavietės, duomenų apsaugos ir kitokias programavimo klaidas. Klaidų aptikimo taisyklės gali būti keičiamos pagal vartotojo poreikius.

Kai tam tikra taisyklė yra įjungta, programa įvertina taisyklės sąlygą bei laukia įvykių, kuriuos sugeneruoja sistema. Kai taisyklė pažeidžiama, vartotojas informuojamas bei pateikiami visi paskutiniai sistemos veikimo aspektai: duomenys, kuri taisyklė kurioje kodo vietoje buvo pažeista taisyklė ir kt. [10]

Visual Threads savybės:

1. Gali būti pritaikoma jau parašytam kodui
2. Nereikalauja daug vartotojo įsikišimo
3. Nepritaikytas konkrečiai programavimo kalbai
4. Nemokamas mokslinėms reikmėms

VisualThreads aklaviečių aptikimui naudoja programos gijų priklausomybės grafą. Aklavietė apibrėžiama kaip ciklas šiame grafe. Jis aptinkamas panaudojus rekurentinę paiešką, pritaikius priklausomybės grafui, kurį sukonstruoja Visual Threads iš programinio kodo. Siekiant didesnio efektyvumo šis algoritmas taikomas tik tuomet, kai gija užblokuojama. [11]

Akivaizdu, kad aptikus aiškią aklavietę, programa nebegali tęsti darbo ir neleistinai baigia darbą. Taigi Visual Threads tikrina, ar sistema tenkina daugelį sąlygų. Viena iš tokių sąlygų aptinka nesuderinamą užraktų atidavimą gijoms. Visual Thread tai atlieka tikrindamas užraktų įgijimo seką ir tikrina visus galimus užraktų įgijimus. Jeigu jie įgyjami nesuderinamai, tuomet tai gali privesti prie aklavietės. Taip aptinkamos potencialios aklavietės.

Nesuderinamo užraktų įgyjimo tikrininimo algoritmas remiasi užraktų porų rinkiniu, kurie negali būti įgyjami vienu metu. Kai gija įgyja naują užraktą sugeneruojamos viso galimos poros su užraktais, kuriuos ta gija jau turi. Tuomet šios poros tikrinimamos su išanksto sugeneruotomis negalimomis poromis. Jeigu randamas atitikimas informuojama apie galimą aklavietę. [11]

Visual Threads aptinka bendrus duomenis, kuriuos dalinasi kelios gijos, netenkinančius tarpusavio išskyrimo sąlygos. Algoritmas daro prielaidą, jog bendri duomenys turi būti valdomi

užrakto **l**, arba gijų sinchronizacijos metodu. Bet koks neatitikimas šiai prielaidai traktuojamas kaip potenciali klaida. [11]

III. Jlint

Jlint – tai statinis kodo analizatorius aptinkantis lygiagretumo problemas. Neprištas prie jokios aplinkos, tačiau neturi galimybės pernešti rezultatus. Jlint aptinka aklavietes, gijų užraktų įgijimo klaidas. Jint sistema sudaryta iš dviejų programų: sintaksės verifikatoriaus ir semantikos verifikatoriaus. Analizuoja programinę įrangą jos nepaleidus. Pirmoji tikrina sintaksės klaidas. Antroji atlieka srautų analizę. [10]

Savybės:

1. Gali būti pritaikoma jau parašytam kodui
2. Aptinka tiek sintaksės, tiek sinchronizacijos klaidas
3. Pritaikytas Java, C, C++ kalboms

Algoritmas tikrina Java klasių failus priklausomybės grafe ieškodamas ciklų. Tokie ciklai gali privesti prie aklavietės. Šiame grafe vaizduojami tiek statiniai, tiek dinaminiai metodai. Kad tikrinimas veiktų efektyviau įrankis riboja galimų paieškų kiekį ta pačia viršūne. Dėl to efektyviau naudojami resursai, sumažėja sistemos apkrovimas.

Taip pat tikrina ir kitą aklavičių šaltinį – metodo *wait()* veikimą. Šis metodas atiduoda objekto užraktą ir laukia kol kita gija jam atsiųs pranešimą dėl galimo tolimesnio veikimo. Dėl to gali susidaryti situacijos, kuomet viena gija laukia, kitos gijos pranešimo, kuri taip pat laukia. Tuomet gali susidaryti aklavietė. Jlint sėkmingai aptinka tokias situacijas.

Jlint aptinka kovą dėl resursų, remiantis tuo, kad ji susidaro tuomet, kai gijos kovoja dėl bendrų resursų ir rezutatai priklauso tik nuo gijų veikimo greičio. Jlint sudaro lygiagrečių metodų sąrašą. Tuomet tikrina sekančias sąlygas konkrečiam kintamajam [12]:

1. Metodai, kurie naudoja pažymėtą lauką lygiagretūs.
2. Kintamasis nėra apibrėžtas kaip final
3. Kintamasis nepriklauso metodo this objektui
4. Kintamasis nėra naujai sukurto objekto, pasiekiamas tik per lokalius kintamuosius – kai obejektas sukuriamas, dažniausiai jį gali pasiekti tik jį sukūrusi gija, per savo lokalius kintamuosius. Taigi sinchronizacija nėra būtina.

5. Kintamasis gali būti pasiekiamas iš keleto metodų, esančių skirtingose klasėse – ne visi kintamieji, kurie yra naudojami keletos gijų turi būti sinchronizuojami, tarkime jie jau yra apsaugoti kritine sekcija.

IV. Rivet

Rivet – tai Java trasavimo ir testavimo įrankių rinkinys. Veikia kartu su java virtualia mašina., jo tikslas išnaudoti vidinę virtualios Java mašinos struktūrą. Taip išgaunamas deterministinis programos veikimas. Deterministinis atkartojimas leidžia virtualiai mašinai vykdyti veiksmus atgaline tvarka, gaunant lygiai tokius pat rezultatus. Todėl įmanoma sukurti galingą testavimo ir trasavimo įrankį. Tačiau esant didelėms programoms, testavimas tampa sudėtingas ir daug naudoja labai daug resursų. Dėl menko našumo šio įrankio vystymas buvo nutrauktas. [10]

Realias aklavietę aptinka deterministiškai pakartojant programos veikimą prieš jai baigiant darbą neleistinu būdu. Rivet įrankis vykdo kiekvieną giją paeiliui iki tol kol ji įgyja užraktą. Tuomet patikrinama, ar dėl to negali susidaryti potenciali aklavietė, t.y. vykdoma atbuline tvarka ir imama kita gija. Taip patikrinamos visos galimos situacijos.

Dėl determinisitinio programos atkartojimo, taip pat įmanoma aptikti ir užsiciklinimus, ir kovą dėl resursų. Programa vykdoma iki klaidos, tada veiksmai atliekami atbuline seka. Tokiu būdu gana paprasta nustatyti klaidos priežastis. [15]

Šis įrankis teisingiausiai bei tiksliausiai aptinka visas lygiagretumo klaida bei priežastis. Taip yra dėl to, kad tiksliai atkartojamas stebimos programinės įrangos veikimas. Tačiau yra senas ir nebenaudojamas dėl našumo problemų.

2.8. Egzistuojančių sprendimų palyginimas

Ankstesniame skyrelyje buvo apžvelgti tipiniai egzistuojantys Java programavimo kalba parašytų programų stebėjimo įrankiai: pateiktas bendras apibūdinimas, išskirtas kiekvieno įrankio esminės funkcijos, aprašytos galimybės bei realizavimo ypatumai. Rezultatai apibendrinami lentele. Jeigu įrankis neturi nurodytos savybės ar neįmanoma įvertinti tam tikro kriterijaus arba jis tenkinamas prastai žymėsime „-“, jeigu įrankis pilnai kriterijų tenkina žymėsime „+“. Kriterijai apibrėžti ir paaiškinti 2.6 skyrelyje.

2 lentelė Įrankių palygimo rezultatai

	Java ConTest	**Visual Threads**	**Jlint**	**Rivet**
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	+	+	+	-

2.9. Analizės išvados

Analizės metu atlikta egzistuojančių tipinių stebėjimo įrankių analizė. Pasirinkti įrankiai įvertinti pagal įvestus kokybės kriterijus. Analizė parodė, jog pagrindinė problema – rezultatų sudėtingumas bei stebėjimo realiu laiku galimybės nebuvimas. Taip pat akivaizdu, jog tirti įrankiai netenkina lygiagrečių sistemų kūrėjų poreikių. Pavyzdžiui taip ir nėra aiški atskirų gijų veikla, kitaip tariant nė vienas įrankis neleidžia be papildomų veiksmų matyti, kokius metodus vykdo gija. Be to nustatyta, kad nė vienas iš analizuotų įrankių netenkina visų reikalavimų. Taigi seka išvada, jog sistemų kūrėjas, norėdamas gauti išsamius bei aiškius rezultataus, turėtų naudotis keletu įrankių bei gautus rezultatus apjungti. Taigi atlikta analizė parodė, jog egzistuoja naujo įrankio apjungiantis visas aptartų sprendimų galimybes. Be to naujas įrankis turi turėti vertintą funcionalumą, kurio neturėjo nė vienas egzistuojantis sprendimas.

3. PROJEKTINĖ DALIS

3.1. Lygiagrečių programų stebėjimo įrankio paskirtis

Sukurtas įrankis skirtas Java programavimo kalba parašytų lygiagrečių programų veikimo stebėjimui bei analizei. Pagrindinė įrankio paskirtis – palengvinti su lygiagretumu susijusių klaidų aptikimą, padėti kūrėjui suvokti sistemos struktūrą bei elgseną, nustatyti stebimos programinės įrangos silpnąsias vietas. Programų stebėjimas atliekamas realiu laiku, atskirai matant kiekvienos gijos būsenas, vykdomus metodus, kritines sekcijas. Apibendrintai galime išskriti tokias sukurto įrankio pritaikymo sritis:

- Studentams, lygiagretaus programavimos mokymosi proceso metu.
- Sukurtų programų verifikacijai.
- Pažengusiems varotojams, kaip alternatyvą komerciniams programų stebėjimo įrankiams, su papildomomis funkcijomis lygiagrečių elementų aptikimui.

3.2. Lygiagrečių programų stebėjimo įrankio vartotojai

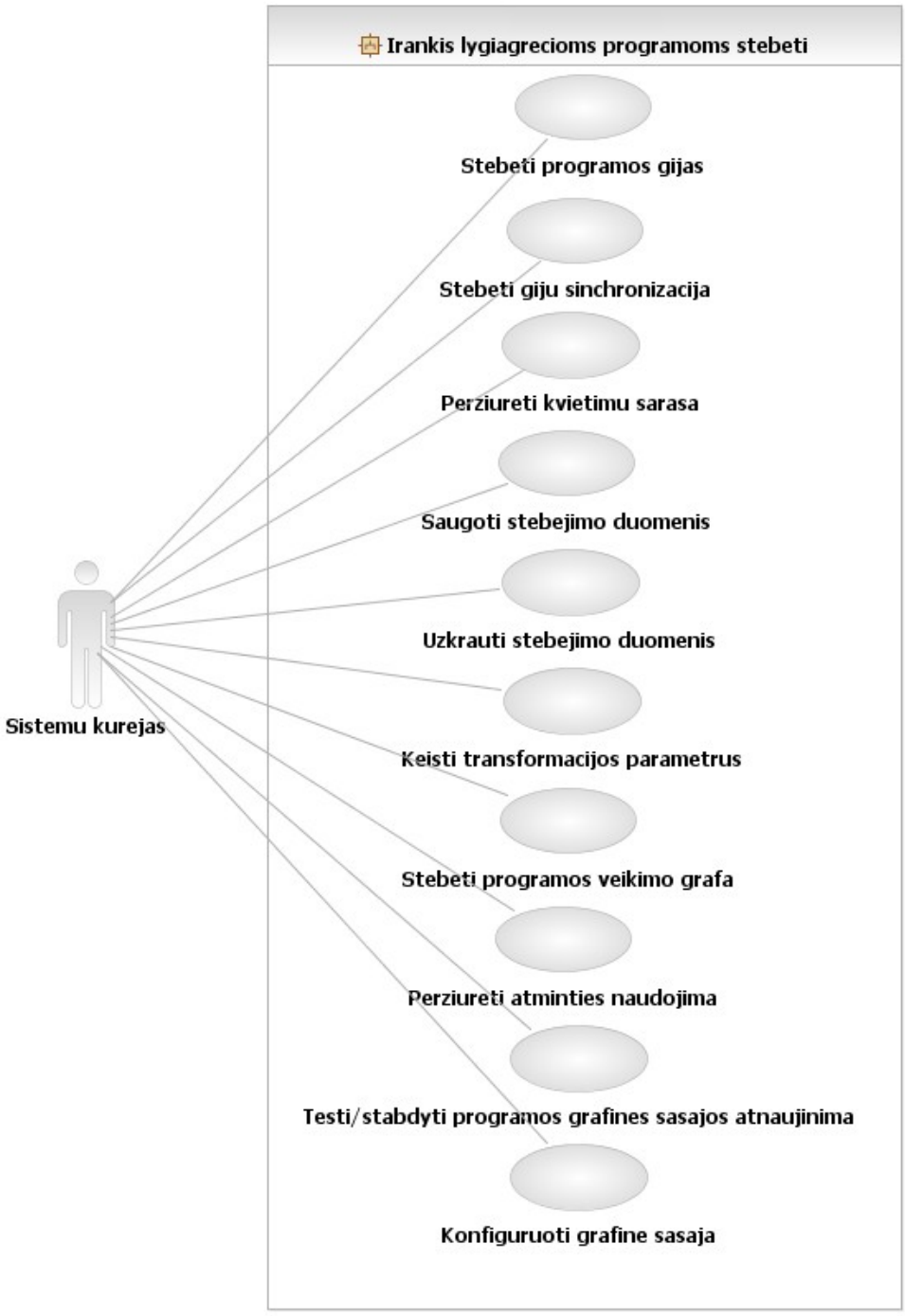
Sistemos pagrindinis vartotojas – programuotojas, norintis stebėti savo programinės įrangos veikimą, joje veikiančias gijas. Taip pat egzistuoja programos administratorius, kuris sutampa su visos sistemos administratoriumi. Jo funkcija – užtikrinti sėkmingą sistemos diegimą, administravimą. Programinėje įrangoje veikiančių gijų grafinis atvaizdavimas, padeda suvokti programos struktūrą, jos elgseną, ko pasekoje jis gali būti sėkmingai panaudotas vizualizuojant pagrindinius lygiagretaus programavimo aspektus bei taip palengvinti mokymosi procesą.

3 lentelė Tipinio vartotojo charakteristikos

Vartotojo kategorija:	Programinės įrangos programuotojas
Vartotojo sprendžiami uždaviniai:	Lygiagrečios programos vykdymas bei stebėjimas Lygiagrečios programos veikimo charakteristikų optimizavimas.
Patirtis dalykinėje srityje:	Naujokas/vidutinis/patyręs
Patirtis informacinėse technologijose:	Naujokas/vidutinis/patyręs
Papildomos vartotojo charakteristikos:	Mokėti parašyti lygiagrečią programą. Turėti minimalios patirties programavimo srityje. Suvokti pagrindines programos vykdymo charakteristikas.

3.3. Panaudos atvejai

Sistemos panaudojimo atvejų diagrama pateikta 4 paveikslėlyje. Joje pateikti pagrindiniai funkciniai reikalavimai sistemai, kurie buvo užfiksuoti reikalavimų ir analogiškų sistemų analizės metu.



3 pav. Sistemos panaudojimo atvejai

Visų panaudojimo atvejų aktorius – sistemų kūrejas, norintis stebėti savo kuriamas ar jau sukurtas programas bei taip aptikti potencialias klaidas ar pasiekti kitą tikslą. Toliau trumpai aprašomas kiekvieno panaudojimo atvejo tikslas:

Stebėti programos gijas – galimybė vartotojui matyti stebimos programos gijas, jų būsenas, vykdomus metodus. Laiko vienetai bei išvaizdos parametrai gali būti keičiami konfigūracijos lange.

Stebėti gijų sinchronizaciją – vartotojas gali matyti, kada tam tikra gija patenka į sinchronizuotą bloką bei kokį laiko tarpą užtrunka.

Peržiūrėti kvietimų sąrašą – peržiūrėti kiekvienos gijos kviečiamų metodų sąrašą. Papildomai pateikiama metodų vykdymo skaičius bei vidutinė trukmė. Be to matoma kokie metodai kokius užraktus įgijo.

Saugoti stebėjimo duomenis – galimybė išsaugoti stebėjimo rezultatus. Stebėjimo rezultatai išsaugomi vartotojo pasirinktoje vietoje, kaip specifinės struktūros xml byla.

Užkrauti stebėjimo duomenis – galimybė užkrauti išsaugotus stebėjimo rezultatus. Rezultatai užkraunami iš vartotojo nurodytos specifinės struktūros xml bylos.

Keisti tranformacijos parametrus – įrankis į kiekvieną stebimos programos klasę įterpia savo kodo segmentą. Sistemų kūrėjas gali nurodyti klases, jų sąrašus arba paketus, kurių stebėti nenori.

Stabdyti/Tęsti programos grafinės sąsajos atnaujinimą – sistemų kūrėjas gali sustabdyti, o vėliau tęsti grafinės sąsajos atnaujinimą. Rezultatai vistiek fiksuojami ir atvaizduojami grafinėje sąsajoje, tačiau sustabdžius grafinės sąsajos atnaujinimą „vaikščiojimą“ laiko juostoje reikės atlikti rankiniu būdu. Pratęsus grafinės sąsajos atnaujinimą, tai vėl atliekama automatiškai.

Konfigūruoti grafinę sąsają – keisti grafinės sąsajos išvaizdos parametrus: gijų būsenų spalvas, laikines charakteristikas, kuriomis norima matyti rezultatus.

Peržiūrėti atminties naudojimą – peržiūrėti atminties naudojimo grafikus. Pateikiama atminties sunaudojimo informacija gauta iš Java virtualios mašinos.

Stebėti programos veikimo grafą – stebėti programos kviečiamų metodų, įgijamų užraktų grafą. Grafe matosi kokie užraktai buvo įgyti, kokie metodai buvo vykdomi. Kiekviena grafo viršūnė vaizduojama atitinkama spalva, priklausomai nuo vykdymo trukmės.

3.4. Esminiai reikalavimai

Šiame skyrelyje aprašyti esminiai nefunkciniai reikalavimai keliami sukurtam lygiagrečių programų stebėjimo įrankui bei jo veikimui. Reikalavimai suskirstyti į 5 kategorijas: reikalavimai grafinei sąsajai, reikalavimai panaudojamumui, reikalavimai vykdymo charakteristikoms, reikalavimai veikimo sąlygoms bei reikalavimai sistemos priežiūrai.

Reikalavimai grafinei sąsajai:

- Grafinis sistemos vaizdas sinchronizuotas su sistemos veikimu. Sistemoje veikiantys procesai turi atsispindėti grafinėje aplinkoje.
- Sistemos naudotojo sąsaja turi būti angliška, tačiau architektūra sudaroma taip, kad būtų lengvai pritaikoma kitoms kalboms.
- Sistema turi būti malonios išvaizdos, nenaudoti ryškių spalvų.
- Grafinė sąsaja turi būti interaktyvi, konfigūruojama.

Reikalavimai panaudojamumui:

- Sistema turi būti pritaikoma jau sukurtai programinei įrangai be papildomo jos modifikavimo. Vienintelis reikalavimas stebimai programai – ji turi būti parašyta Java programavimo kalba.
- Turi būti paprasta išsaugoti ir pernešti stebėjimo rezultatus.

Reikalavimai vykdymo charakteristikoms:

- Sistema turi netrukdyti kitoms sistemoms, efektyviai naudoti resursus.
- Sistema turi būti sukurta taip, kad stebimos programinės įrangos veikimo neįtakotų arba įtakotų neženkliai.

Reikalavimai veikimo sąlygoms:

- Sistema turi veikti tiek Windows, tiek Linux operacinėse sistemose.
- Sistema turi veikti su Java 1.5 ir naujesnėmis versijomis.

Reikalavimai sistemos priežiūrai:

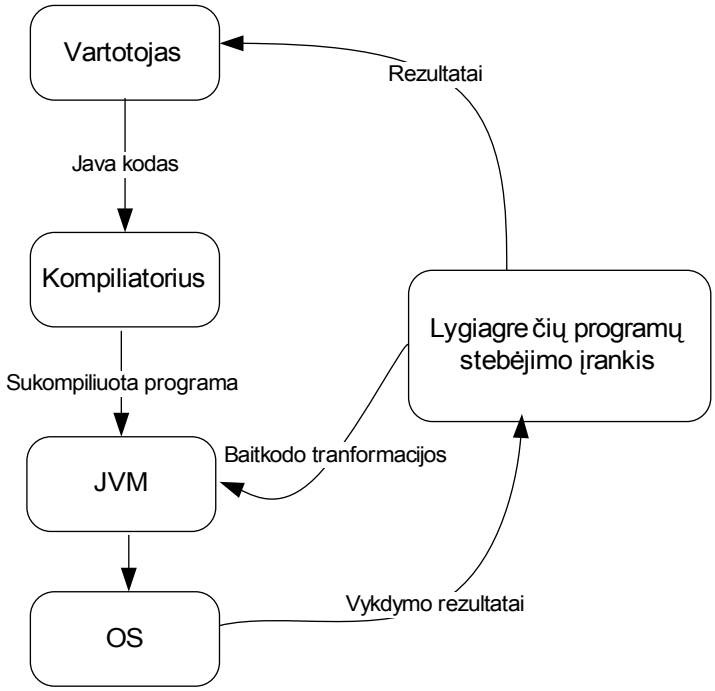
- Sistema turi būti suprojektuota taip, kad naujo funkcionalumo pridėjimas nereikalautų egzistuojančių komponentų radikalaus pakeitimo.
- Sistemos įdiegimas ir atnaujinimai atliekami aiškiai ir nesudėtingai.

3.5. Veikimo principas

Java programavimo kalboje programos kodas kompiliuojamas į specialų objektinį kodą, vadinamą baitkodu. Šį kodą toliau skaito ir vykdo Java abstrakti virtualioji mašina [14]:

`Kodas` → `Kompiliatorius` → **`baitkodas`** → `JVM`

Įrankis realizuotas kaip Java agentas – speciali biblioteka, kuri vykdoma užkraunant baitkodą į virtualią mašiną. Taigi naudojantis agentu tampa įmanoma keisti jau sukompiliuotas programas, kitaip tariant tiesiogiai redaguoti baitkodą. Įrankis baitkodo užkrovimo į JVM metu analizuoja sugeneruotą baitkodą ir reikiamose vietose kviečia savo vidinius metodus. Tokiu būdu stebima programa pakeičiama baitkodo lygyje jos užkrivimo metu, todėl papildomų veiksmų vartotojui imtis nereikia. Be to šie pakeitimai yra nematomi vartotojui bei neįtakoja programos darbo arba įtakoja neženkliai. Programos vykdymo metu įrankis fiksuoja gautą informaciją ir ją atvaizduoja vartotojui.



4 pav. Konteksto diagrama

Programos veikimo principas iliustruotas aukščiau esančioje konteksto diagramoje. Lygiagrečių programų stebėjimo įrankis – agentas – baitkodo užkrovimo metu prideda savo kodo segmentus į reikiamas vietas. Tokia pakoreguota programa vykdoma atitinkamoje platformoje. Programos vykdymo metu yra kviečiamos atitinkamos lygiagrečių programų stebėjimo įrankio komandos, o rezultatus vartotojas mato grafinėje sąsajoje.

3.6. Realizavimo priemonės

Realizacijai pasirinkta Java programavimo kalba. Ji gali būti vykdoma tiek Windows, tiek Linux aplinkoje.

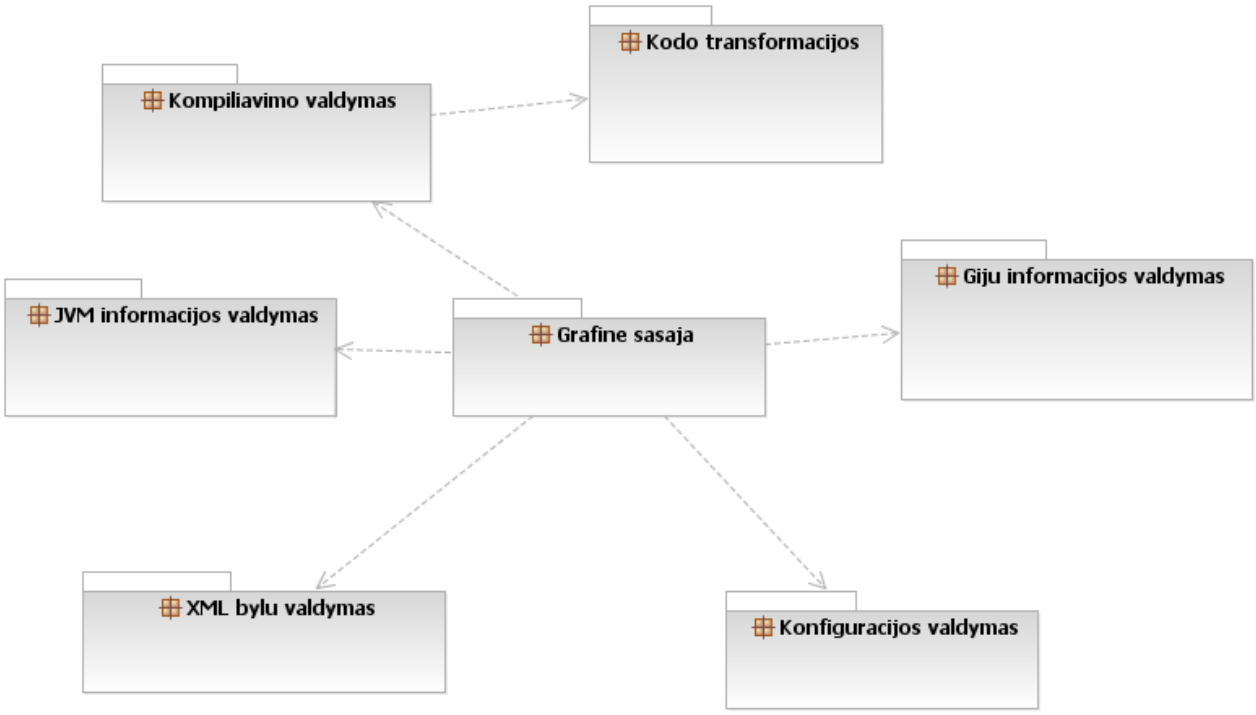
java.lang.instrument biblioteka – speciali biblioteka leidžia valdyti klasių užkrovimo į JVM procesą. Tai suteikia galimybę stebėti, kokios klasės yra vykdomos arba keisti užkrautų klasių baitkodą.

javaAssist – klasė skirta baitkodo transformacijoms atlikti. Leidžia paprastai įterpti tam tikrus kodo segmentus į metodų pradžią ar pabaigą, fiksuoti tam tikrų metodų kvietimus ir pan.

java.lang.management – biblioteka skirta JVM informacijai pasiekti, OS parametrams gauti.

3.7. Statinis vaizdas

Glaustai aprašomas statinis sukurto lygiagrečių programų stebėjimo įrankio statinis vaizdas. Sistema suskaidyta į septynis pagrindinius paketus, kurie pateikti antrame paveikslėlyje. Papildomai naudojamos bibliotekos paketų diagramoje nevaizduojamos.



5 pav. Paketų diagrama

Kodo transformacijos

Pakete saugomos klasės atsakingos už stebimos programinės įrangos sukompiliuoto baitkodo keitimą. Klasių užkrovimo metu įterpiami specialūs įrankio kodo blokai, kviečiantys metodus, organizuojančius duomenų struktūrų, saugomų gijų fiksavimo pakete, užpildymą. Tai atliekama baitkodo užkrovimo metu, todėl vartotojui šie pakeitimai nepastebimi. Pagrindinė paketo klasė – transformatorius, atliekantis pagrindinį darbą – baitkodo transformacijas. Žemiau pateikiamas baitkodo transformacijų pavyzdys. Pabrėžtina, jog dėl aiškumo pateikiamas ne baitkodas, o Java kodas.

4 lentelė Tranfornacijos pavyzdys

Kodas prieš transformaciją	Kodas po transformacijos

`public class TestThread extends Thread`
`{`
 	
`  public void run() {`
`  m1(); m2();`
`  }`
 	
`  public void m1(){`
`  Thread.sleep(3000);`
`  TestObject o = new TestObject();`
`  o.setName("nida");`
`  }`
 	
`  public void m2(){`
`  Thread.sleep(1000);`
`  }`
 	
`}`
	`public class TestThread extends Thread {`
	
	`  public void run() {`
	`  m1(); m2();`
	`  }`
	
	`  public void m1(){`
	`  **ThreadProfile.enterMethod(Thread.currentThread(), "m1");**`
	`  Thread.sleep(3000);`
	`  TestObject o = new TestObject();`
	`  o.setName("nida");`
	`  **ThreadProfile.leaveMethod(Thread.currentThread(), "m1");**`
	`  }`
	
	`  public void m2(){`
	`  **ThreadProfile.enterMethod(Thread.currentThread(), "m2");**`
	`  Thread.sleep(1000);`
	`  **ThreadProfile.leaveMethod(Thread.currentThread(), "m2");**`
	`  }`
	
	`}`

Grafinė sąsaja

Šiame pakete saugomos klasės atsakingos už informacijos perdavimą grafinei sąsajai bei visą grafinės sąsajos valdymą. Paketo klasės organizuotos grafinių langų principu, t.y. kiekviena panelė turi atskirą savo vaizdavimo klasę. Be to pakete saugomos klasės, reikalingos bendram visų sąsajų išvaizdos keitimui. Grafinei sąsajai naudojama `javax.swing` biblioteka.

Gijų informacijos valdymas

Pakete esančios klasės, atsakingos už stebimos programinės įrangos gijų fiksavimą bei informacijos apie jas kaupimą. Šis komponentas užfiksuotą informaciją organizuoja į sąrašą, kuris vėliau naudojamas skaičiavimuose bei vaizdavime. Užfiksuojama kokia gija einamu momentu vykdo kokį metodą. Ši informacija saugoma medžio tipo duomenų struktūroje. Taip pat komponentas turi metodus rezultatų pateikimui skirtingais matais.

Šiame pakete taip pat atliekmas užfiksuotos gijų informacijos apdorojimas. Skaičiuojamos metodų vykdymo trukmės, jų kvietimo skaičius, atminties sunaudojimas, procesoriaus apkrautumas. Komponentas taip pat turi klasę, organizuojančią programinės įrangos veikimo laiką: nustatoma kada programa pradedama vykdyti ir toliau laikas skaičiuojamas nuo to momento, pasirinktais matais. Taip pat klasė kontroliuoja su veikimo sustabdymu bei „vaikščiojimu“ laiko skalėje susijusius skaičiavimus.

JVM informacijos valdymas

Pakete saugomos klasės, atsakingos už bendros informacijos apie Java virtualią mašiną gavimą. Šiam tikslui naudojamos `java.lang.management` bibliotekos, leidžiančios gauti tokią informaciją. Naudojami trys šios bibliotekos valdymo interfeisai:

- `RuntimeMXBean` – JVM vykdymo aplinkos
- `MemoryMXBean` – JVM atminties
- `OperatingSystemMXBean` – operacinės sistemos, kurioje paleistas JVM.

Konfigūracijos valdymas

Panelė skirta stebimos programos gijų grafinio atvaizdavimo parametrų keitimui. Šios panelės pagalba galima pakeisti stulpelinių diagramų principu vaizduojamų gijų būsenų spalvas. Taip pat šioje panelėje galima atstatyti standartinius nustatymus bei importuoti naujus arba eksportuoti esamus gijų vaizdavimo parametrus.

Konfigūracijos valdyme taip pat saugomo klasės valdančios baitkodo transformacijų nustatymus. Vartotojui pageidaujant, galima nurodyti klases ar paketus, kurių stebėti vartotojas nenori. Tuomet šioms klasėms nebus taikomos jokios transformacijos, taigi ir informacija apie jas nebus fiksuojama.

XML bylų valdymas

Šiame pakete saugomų klasių pagalba atliekamas XML bylų apdorojimas bei naujų sukūrimas. Šio paketo pagalba galima nuskaityti XML failus į atitinkamą struktūrą arba suformuoti XML failą iš per parametrus perduodamos strukūros. XML formatu saugomi stebėjimo rezultatai, konfigūraciniai failai. Naudojama `jdom` biblioteka.

4. TYRIMO DALIS

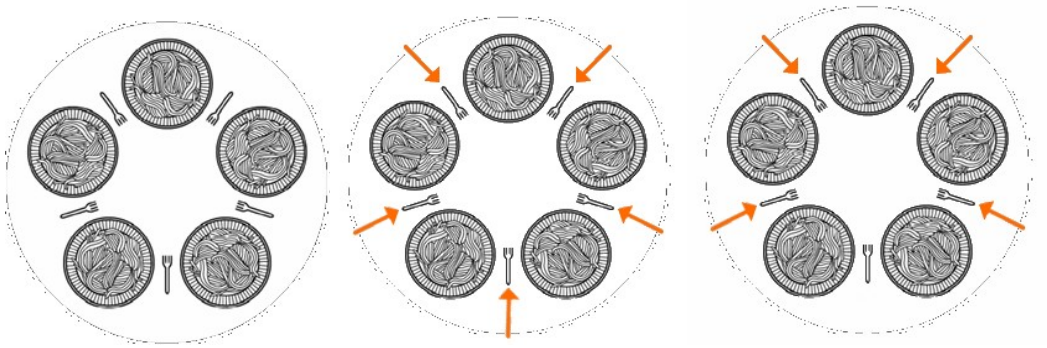
4.1. Tyrimo tikslas

Tyrimo tikslas – sukurti ekspermentui tinkančias programas, atspindinčias lygiagretaus programavimo principus. Šioje dalyje aptariama pietaujančių filosofų bei gamintojo-pirkėjo problemos, pateikiama kokiais Java teikiamais mechanizmais ši problema gali būti sprendžiama. Tyrimo rezultatas – simuliacinės programos.

4.2. Pietaujančių filosofų problema

Pietaujančių filosofų problema – tai klasikinė sinchronizacijos problema. Ji atspindinti pagrindines lygiagrečių sistemų problemas. Problema apibrėžiama taip:

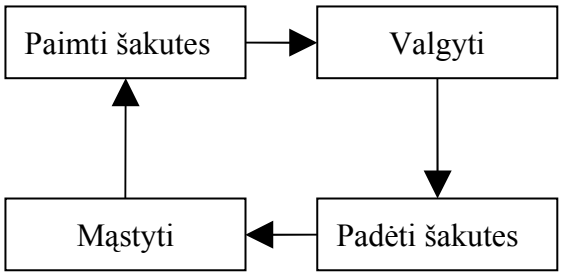
N filosofų sėdi prie apvalaus stalo. Tarp kiekvieno greta sėdinčio filosofo yra šakutė. Kitaip tariant yra N šakučių. Kiekvienas filosofas arba galvoja, arba valgo. Filosofas galvoja tol, kol išalksta. Tam, kad jis galėtų valgyti reikalingos dvi šakutės. Taigi išalkęs jis pasiima dvi šakutes ir pradeda valgyti. Kai filosofas pavalgo, jis padeda šakutes atgal ir ima vėl galvoti. [17]



6 pav. Pietaujantys filosofai: a) pietaujančių filosofų problema b) aklavietė c)badavimas

Kiekviena šakutė yra bendrai naudojamas resursas. Todėl gali susidaryti kova dėl resursų, t.y. jeigu filosofas bandys paimti šakutę vienu metu. Kadangi filosofai tarpusavyje nebendrauja, gali susidaryti aklavietė, pavyzdžiui jeigu visi filosofai išalks vienu metu pasiims po šakutę ir lauks antros. Taip pat gali susidaryti badavimas: jeigu visą laiką valgys vienas prieš kitą sėdintys filosofai, o kiti trys nespės paimti šakučių, tuomet jie niekuomet nevalgys.[16]

Filosofų atliekami vieksmai yra ciklikliški. Filosofas paima šakutes, pavalgo, padeda šakutes, masto ir vėl kartoja tuos pačius veiksmus.



7 pav. Pietaujančių filosofų problema - filosofo veiksmai

Esminis veiksmas yra „paimti šakutes“. Filosofai negali paimti tos pačios šakutės(resurso) tuo pat metu, kadangi susidarys kova dėl resursų. Be to filosofai negali laukti šakutės, kurios niekada negaus. Tam, kad išvengtume šių problemų tarkime, kad kiekviena šakutė yra bendrai naudojamas resursas, kuris turi būti apsaugomas užraktu. Tuomet filosofas gaus šakutę tik tuomet, jeigu ji bus laisvas, t.y. užraktas nebus užrakinti. Žemiau pateikiama veiklos diagrama.

8 pav. Pietaujantys filosofai veiklos diagrama

4.2.1. Simuliacinė programa

Testavimams sudaroma simuliacinė pietaujančių filosofų programa. Ji parašyta Java programavimo kalba, naudojantis standartinėmis gijų kūrimo procedūromis. Pilnas programos kodas pateikiamas 1 priede.

Paprastumo dėlei tarkime, jog prie stalo sėdi N=3 filosofų. Taigi egzistuoja 3 filosofai bei tiek pat šakučių . Kiekviena šakutė *fork* apsaugota užraktu, o ją vienu metu gali naudoti tik vienas filosofas *philosofer*. Filosofas – tai Java gija turinti penkis metodus bei du užraktais apsaugotus objektus – šakutes.

Klasės užraktai:

`ReentrantLock leftFork` – užraktas, saugantis pirmąją šakutę.

`ReentrantLock rightFork` – užraktas, saugantis antrąją šakutę.

Klasės metodai:

`Think()` – filosofas galvoja, gija laukia atsitiktinį laiką.

`TakeFirstFork()` – filosofas paima pirmąją šakutę, gija įgyja pirmąjį užraktą.

`TakeSecontdFork()` – filosofas paima antrąją šakutę, gija įgyja antrąjį užraktą.

`Eat()` – filosofas valgo, gija įgyja antrąjį užraktą.

`PutDownForks()` – filosofas padeda šakutes, gija paleidžia abu užraktus.

Be to filosofo klasė turi metodą `run()`, kuriame vykdoma tokia veiksmų seka: filosofas galvoja, paima pirmą šakutę, paima antrą šakutę, valgo, padeda šakutes. Šis seka vykdoma be perstojo tol, kol programos darbas nenutraukiamas.

Paprastumo dėlei gijoms priskiriami atitinkami pavadinimai: *Phil1*, *Phil2* ir *Phil3*.

Be teisingos programos sudarytos kelios klaidingos programos(mutantai) bei nurodyta, kokia klaida turėtų susidaryti:

5 lentelė Pietaujančių filosofų mutantai

Nr.	Originalus kodas	Pakeistas kodas	Klaida

1.	`if (number % 2 == 0)`
	`  leftFork.lock();`
	`else`
	`  rightFork.lock();`
		`leftFork.lock();`	Aklavietė

2.	`if (number % 2 == 0)`
	`  rightFork.lock();`
	`else`
	`  leftFork.lock();`
		`rightFork.lock();`	Aklavietė

3.	`public void putDownForks() {`
	`  leftFork.unlock();`
	`  rightFork.unlock();`
	`}`
		`public void putDownForks() {`
		`}`
			Badavimas

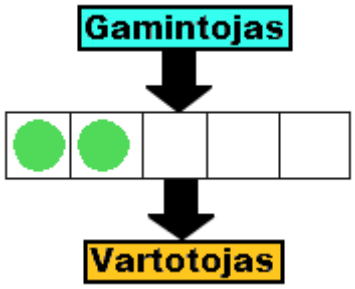
4.	`takeFirstFork();` `takeSecondFork();`	`takeSecondFork();` `takeSecondFork();`	Badavimas
5.	`takeFirstFork();` `takeSecondFork();`	`takeFirstFork();` `takeFirstFork();`	Badavimas
6.	`if (number % 2 == 0)`	`if (number % 2 == 2)`	Aklavietė

7.	`takeFirstFork();`
	`takeSecondFork();`
	`....`
	`if (leftFork.isLocked())`
	`  leftFork.unlock();`
		`takeSecondFork();`
		`takeSecondFork();`
		`...`
		`leftFork.unlock();`
			Gija nutraukiama

8.	`sleep(time);`	`while(true)` `  sleep(time);`	Aklavietė

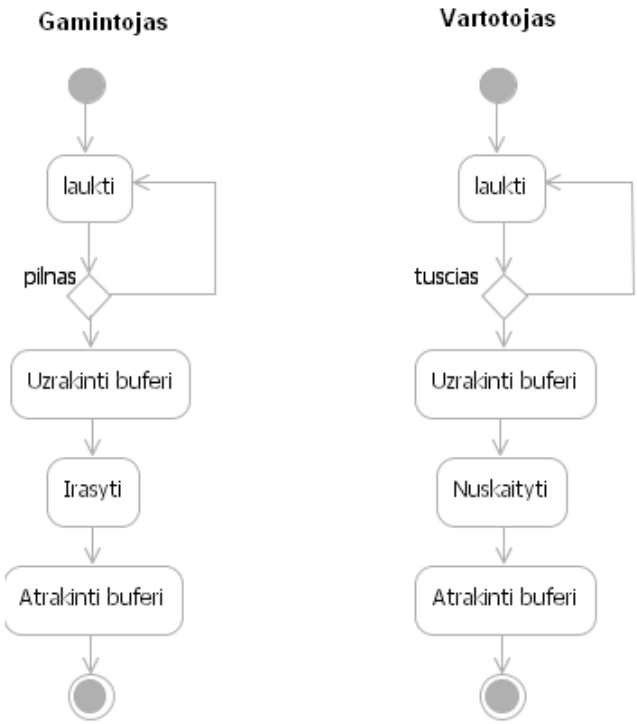
4.3. Gamintojo-vartotojo problema

Gamintojo-vartotojo problema – tai dar viena problema, parodanti, jog sistemose, kurios naudojasi bendrais resursais būtina gijų sinchronizacija. Tarkime turime bendrai naudojamą fiksuoto dydžio buferį. Viena gija – gamintojas – rašo informaciją į buferį, kita – vartotojas – skaito informaciją iš buferio. Abi gijos veikia lygiagrečiai. Problema susidaro tuomet, kai gija bando įrašyti į pilną buferį arba nuskaityti iš tuščio.



9 pav. Gamintojo-vartotojo problema

Taigi tam, kad nekiltų problemų naudojantis bendru buferiu, t.y. nesusidarytų kova dėl resursų, gijos turi būti teisingai sinchronizuotos. Gamintojas gali įrašinėti į buferį tik tuomet, kai jis nėra pilnas. Vartotojas gali skaityti iš buferio tada, kai jis nėra tuščias. Taigi tiek gamintojas tiek vartotojas gali laukti, kol šios sąlygos bus tenkinamos.



10 pav. Gamintojo-vartotojo problema veiklos diagramos

4.3.1. Simuliacinė programa

Testavimams sudaroma simuliacinė gamintojo-vartotojo programa. Ji parašyta Java programavimo kalba, naudojantis standartinėmis gijų kūrimo procedūromis. Pilnas programos kodas pateikiamas 2 priede.

Simuliacinė programoje turime 2 gamintojus ir 2 vartotojos, o buferio ilgis 3. Programa turi dvi vidines klases `Producer` ir `Consumer`. Šios klasės realizuotos kaip Java gijos. Be to programa turi sinchronizuotą buferį, t.y. rašyti arba skaityti gali tik viena gija vienu metu. Vartotojo gija, radusi tuščią buferį laukia (metodas *wait()*, o pasiėmusi gaminį signaliznalizuoja laukiančioms gamintojo gijoms (metodas *notifyAll()*. Analogiškai gamintojo gija radusi pilną buferį laukia, o įdėjusi gaminį signalizuoja laukiančioms vartotojo gijoms.

`Producer` klasė turi du metodus:

`put` – įdeda gaminį į buferį;

`produce` – gamina gaminį.

`Consumer` klasė taip pat turi du metodus:

`take` – paima gaminį iš buferio;

`consume` – naudoja gaminį.

Be teisingos programos sudarytos kelios klaidingos programos(mutantai) bei nurodyta, kokia klaida turėtų susidaryti:

6 lentelė Gamintojo-vartotojo mutantai

Nr.	Originalus kodas	Pakeistas kodas	Tikėtina klaida
1.	`synchronized (list)`	`list`	Kova dėl resursų
2.	`list.notifyAll();`	`list.notify();`	Badavimas
3.	`list.wait();`		Kova dėl resurų
4.	`int len = list.size();`	`int len = 7;`	Kova dėl resurų
5.	`list.addFirst(justProduced)`		Gija nutraukiama

5. EKSPERIMENTINĖ DALIS

5.1. Eksperimento tikslas

Šios dalies tikslas įvertinti sukurtą programinę sistemą pagal analizės dalyje įsivestus kriterijus. Tyrimo metu naudojamos sudarytos pietaujančių filosofų bei gamintojo-vartotojo simuliacinės programos, aprašytos ankstesniame skyrelyje. Skyriaus pradžioje aptariamas bendras įrankio funkcionalumas. Vėliau pademonstruojamas pritaikymas konkrečiai ankščiau sudarytai simuliacinei programai. Skyriaus pabaigoje pateikiami ekspermento rezultatai.

5.2. Įrankio diegimas ir paleidimas

Norint sėkmingai naudotis produktu, būtini tam tikri paruošiamieji darbai. Būtina įsidiegti programinę įrangą java programa parašytų kalbų kompiliavimui bei vykdymui.

Sistema gali būti naudojama su bet kokia integracine aplinka ar tiesiog paprastu kompiatoriumi. Vienintelis reikalavimas – turėti įrankį stebimos programos kompiliavimui bei vykdymui bei mokėti juo tinkamai naudotis.

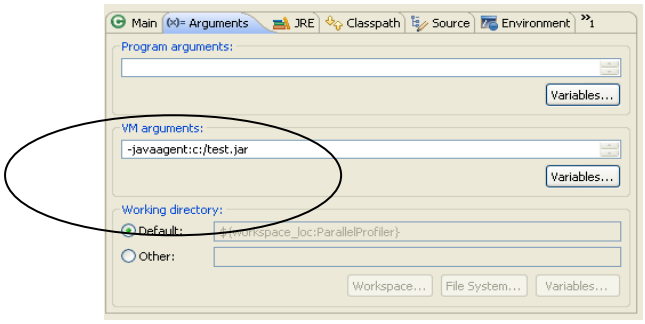
Naudojantis komandine eilute, programa, kurią norime stebėti, turi būti kompiliuojama ir paleidžiama įprastu būdu, pridedant papildomą įrankio argumentą. Standartinis paleidimas iš komandinės eilutės parodytas pavyzdyje:

`java ExampleProgram`	`-javaagent:home/ConcGraph.jar`
Standartinė komanda	`Papildomas argumentas`

Čia *home* – ConcGraph namų direktorija.

Standartinė komanda – įprasta stebimos programinės įrangos paleidimo komanda.

Naudojantis integracine aplinka papildomą vykdymo parametrą galima nurodyti vykdymo parinktyse. Pavyzdžiui, Eclipse integracinėje aplinkoje tai atliekama paleidimo nustatymuose keičiant „VM Arguments“ laukelį kaip parodyta paveikslėlyje:



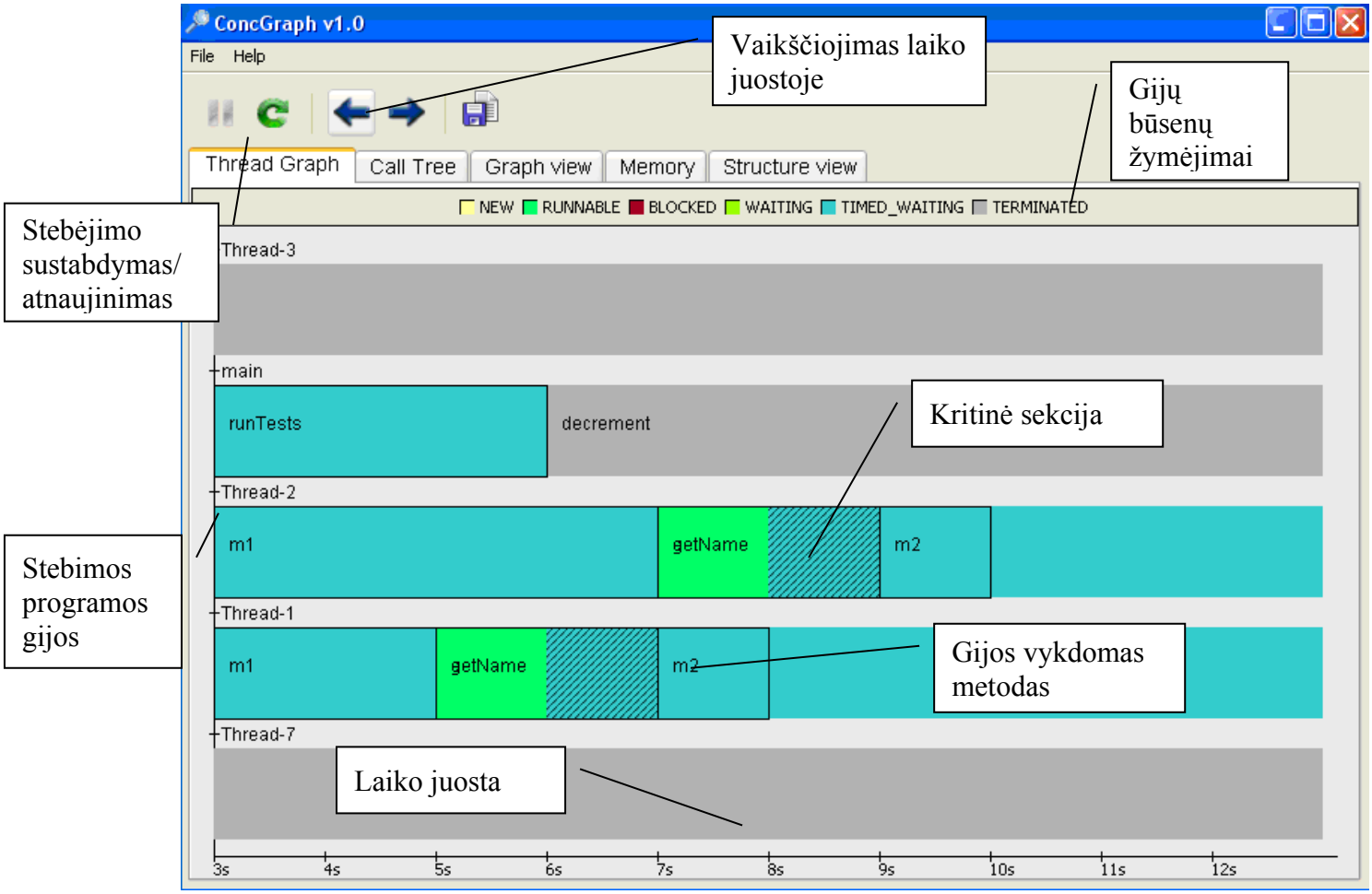
11 pav. Įrankio paleidimas su „Eclipse“ IDE

5.3. Įrankio pagrindinių funkcijų paaiškinimai

Šiame skyriuje trumpai aprašomas kiekvienas sukurto įrankio langas bei jo funkcionalumas. Paaiškinama pagrindiniai valdymo mygtukai, juostos bei jų paskirtis.

5.3.1. Gijų stebėjimas

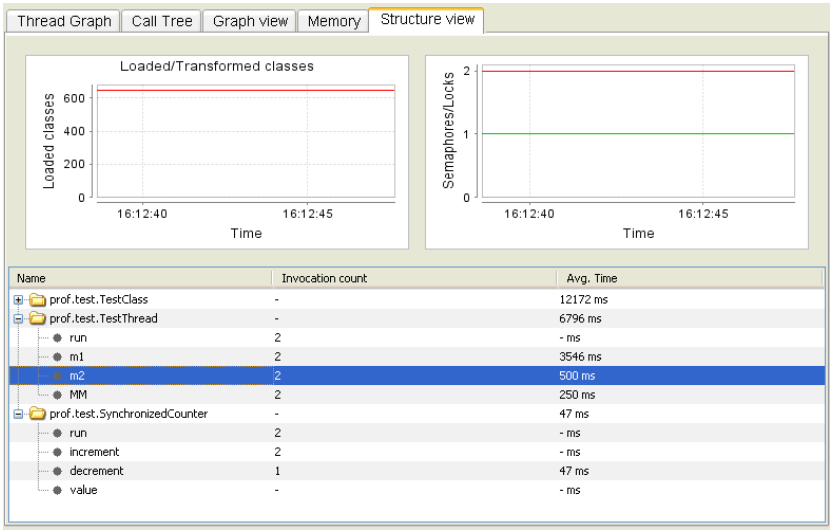
Pagrindinis gijų stebėjimo langas matomas vos tik paleidus programą. Čia pateikiamas gijų sąrašas laiko juostoje, kur atskaitos taškas – programos paleidimo momentas. Kiekviena gija vaizduojama savo būsenos spalva. Atitinkamų gijų būsenų spalvos pavaizduotos gijų būsenų legendoje (jas galima keisti konfigūracijos lange). Kiekviena kritinė sekcija vaizduojama atskirai išryškinant, kaip pavaizduota paveikslėlyje žemiau. Taip pat yra galimybė „vaikščioti“ laiko juostoje bei sustabdyti ir atnaujinti programos veikimą. Kiekvienoje gijoje demonstruojamas vykdomų metodų sąrašas. Norint matyti pilną vykdomų metodų vaizdą perjunkite metodų kvietimo sąrašo vaizdą.



12 pav. Pagridinis langas su paaiškinimais

5.3.2. Programos stuktūros vaizdas

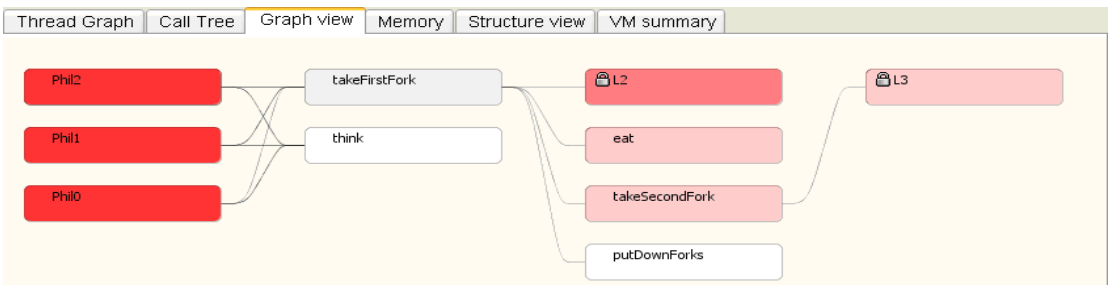
Programos struktūros vaizde pateikiama informacija apie stebimos programos struktūrą bei vydymo charakteristikas. Pateikiamas užkrautų klasių sąrašas grupuojant pagal paketus bei klases. Čia nerodomos klasės ar paketai, kurias jūs nurodėte atmesti konfigūracijos lange. Tokios klasės ar paketai įskaičiuojami į užkrautų klasių skaičių, tačiau neįtraukiamos į transformuotų klasių skaičių. Kiekvienam elementui nurodomas vidutinis vykdymo laikas bei kvietimų skaičius. Prie nekviestų metodų, klasių ar paketų rodomas „-„. Yra galimybė rikiuoti sąrašus spaudžiant ant atitinkamų lentelės pavadinimų. Taip pat pateikiami grafikai, kiek klasių buvo užkrauta/transformuota bei užfiksuotų užraktų, semaforų skaičius.



13 pav. Programos struktūros langas

5.3.3. Programos grafas

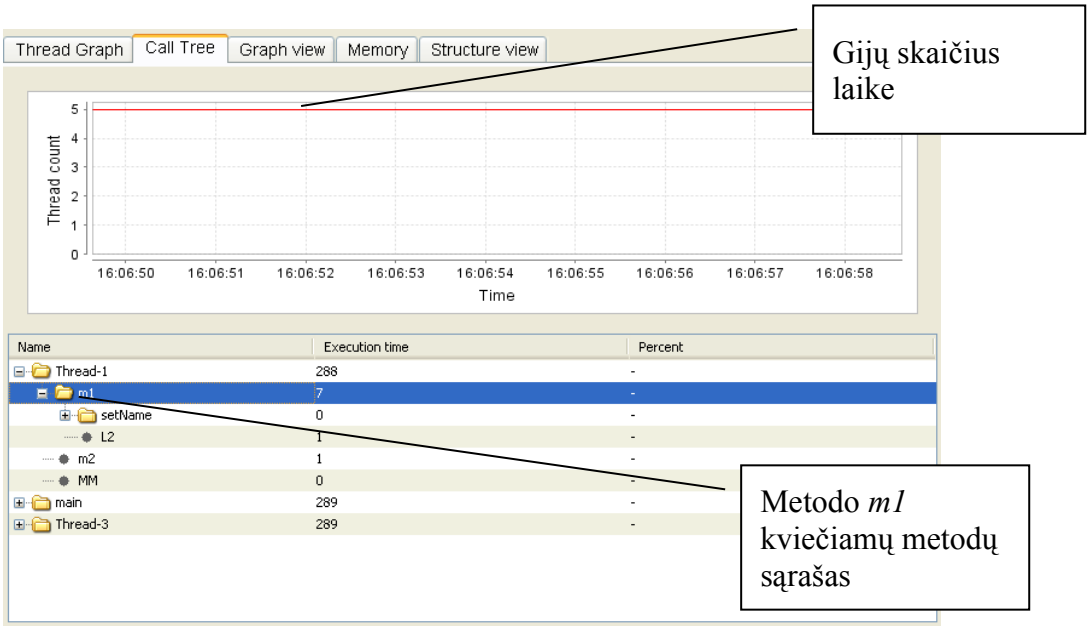
Šioje panelėje vaizduojamas grafinis visos programos vaizdas. Čia gijos vaizduojamos rutuliukais su gijos pavadinimu bei tipu jo viduje. Įgyjami užraktai taip pat atvaizduojami rutuliuku, su viduje pateiktu trumpu užrakto vardu



14 pav. Programos grafas

5.3.4. Metodų kvietimų medis

Metodų kvietimų medyje pateikiamas metodų sąrašas grupuojant pagal stambesnius elementus bei smulkinant, t.y. pirmiausiai metodai grupuojami pagal gijas, toliau grupuojama pagal tai kokie metodai kokius kviečia. Prie kiekvieno metodo pateikiamas jo vykdymo laikas bei procentinė išraiška kiek tai sudaro viso programos vykdymo laiko. Jeigu procentinė išraiška nėra vaizduojama, tai programa ar metodas dar nebaigė darbo. Minėta lentelė gali būti rikiuojama pagal vykdymo laiką paspaudžiant lentelės antraštes. Viršuje pateikiamas gijų skaičiaus grafikas laike.



15 pav. Metodų kvietimo medis

5.3.5. Informacijos saugojimas/importavimas

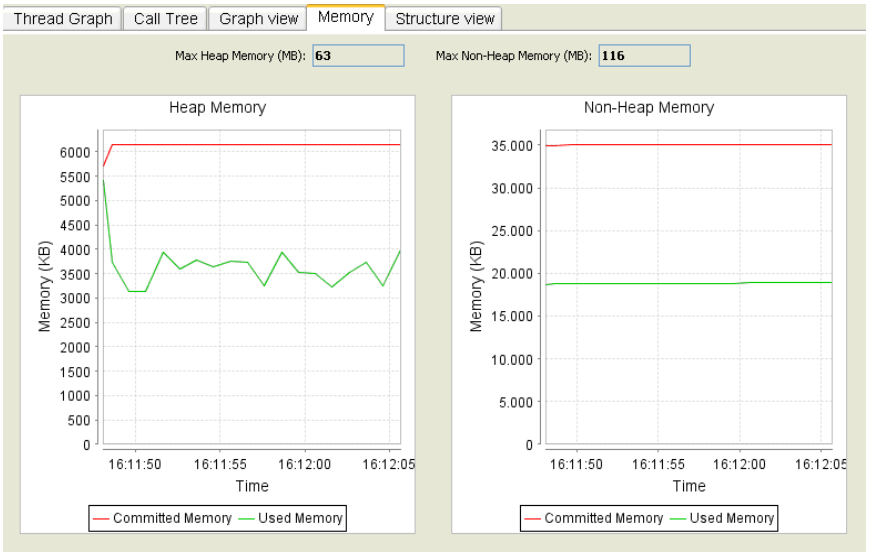
Stebima informacija gali būti išsaugoma bei atidaroma, kad peržiūrėti stebėjimo rezultatus vėliau. Saugojimas atliekamas įrankių juostoje norimu momentu spaudžiant saugojimo mygtuką.

16 pav. Informacijos saugojimas

Norint užkrauti išsaugotą informaciją pasirenkamas meniu punktas File->open. Atsidariusiame dialoge nurodomas reikiamas failas. Informacija atidaroma paspaudus mygtuką „Open“.

5.3.6. Atminties naudojimas

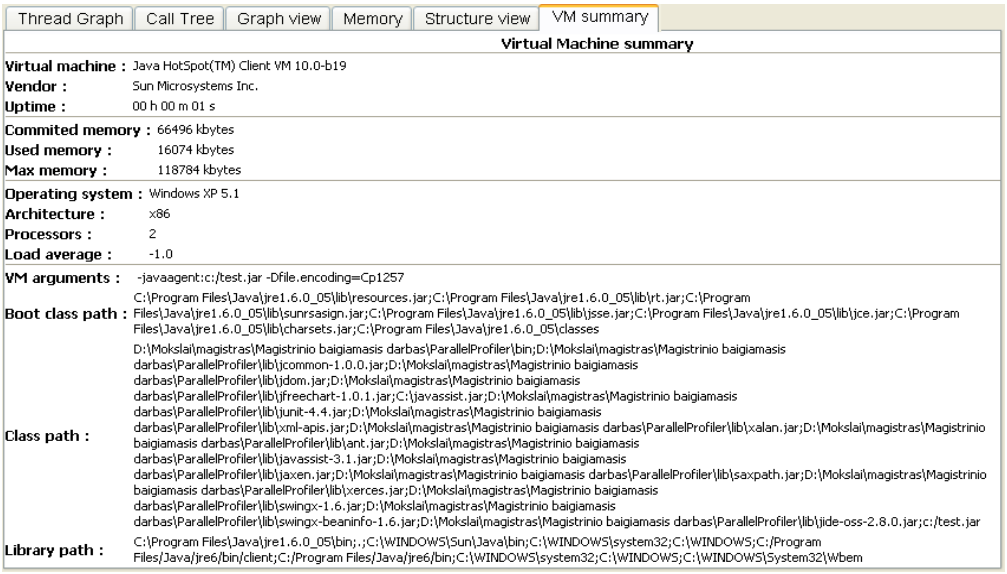
Atminties naudojimo grafe pateikiama informacija apie heap ir non-heap atmintį. Pateikiami abiejų atminčių naudojimo grafikai. Atskiromis grafikų linijomis pateikiama informacija kiek atminties išskirta ir kiek realiai naudojama.



17 pav. Atminties naudojimo grafikai

5.3.7. JVM informacija

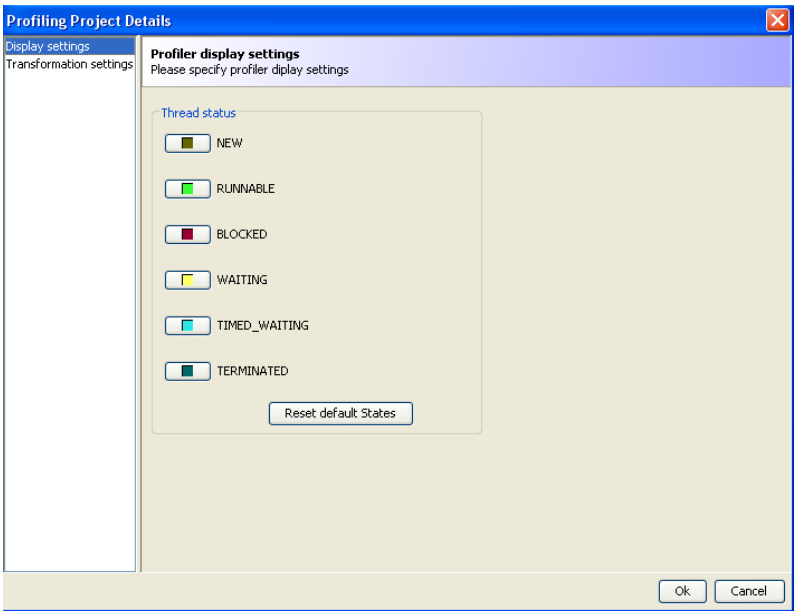
Lange JVM informacija pateikiama informacija apie virtualios mašinos veikimo aplinką, argumentus, naudojamą atmintį. Pateikiama JVM versija, veikimo laikas, taip pat operacinė sistema. Taip pat pateikiami klasių bei bibliotekų keliai.



18 pav. JVM informacija

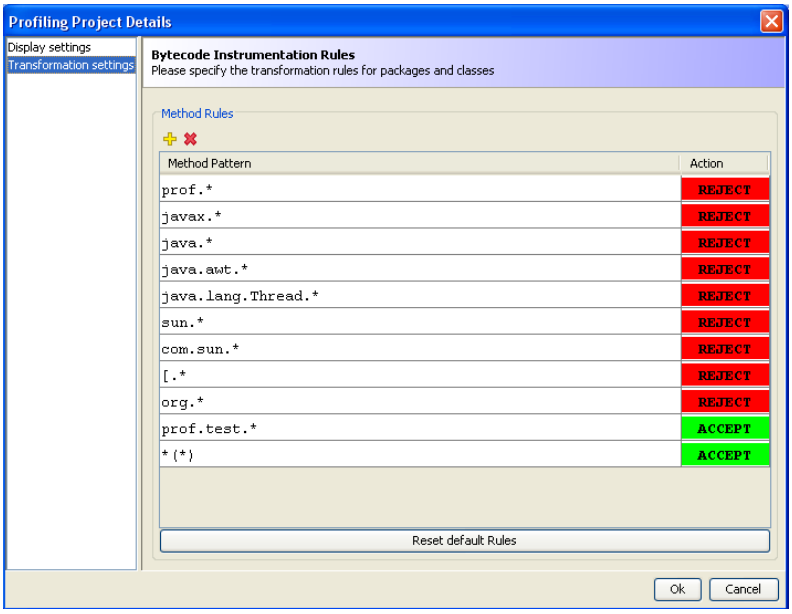
5.3.8. Konfigūracija

Konfigūracijos lange, galime keisti gijų būsenų spalvas. Kiekvienai gijai vaizduoti parenkama skirtinga spalva. Išsaugojus parinktis pagrindiniame stebėjimo lange matomų gijų išvaizda iškart pasikeičia. Taip pat galima atstatyti numatytąsias būsenų spalvas.



19 pav. Konfigūracijos langas

Sistemos nustatymuose taip pat konfigūruojamos baitkodo transformacijų taisyklės. Nurodoma, kuriems paketams ar klasėms vykdyti baitkodo transformacija, o kurioms ne. Klasės, kurios nurodytos šiame sąraše netransformuojamos todėl, nėra vaizduojamos įrankyje.

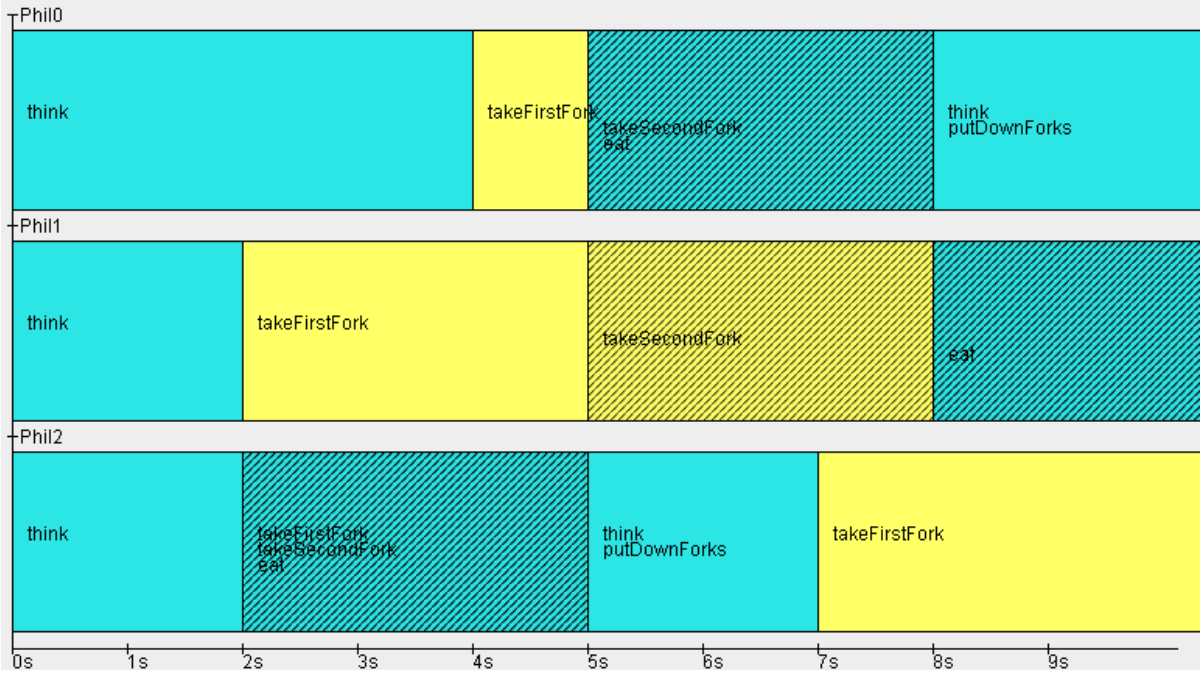


20 pav. Tranformacijų taisyklių konfigūravimas

5.4. Pietaujačių filosofų simuliacinės programos tyrimas

Tyrimo dalyje aptarta pietaujančių filosofų problema bei sudaryta ją simuliuojanti programa (išeities kodai pateikti 1 priede). Atliekama paruoštos programos vykdymas bei analizė realizuotu lygiagrečių programų stebėjimo įrankiu.

Programa paleidžiama kaip aprašyta 4.2 skyrelyje. Atidaromas pagrindinis programos langas, kuriame matomos realiu laiku veikiančios programos gijos, rodomi atskirų gijų vykdomi metodai, jų būsenos bei sinchronizuotos sekcijos.

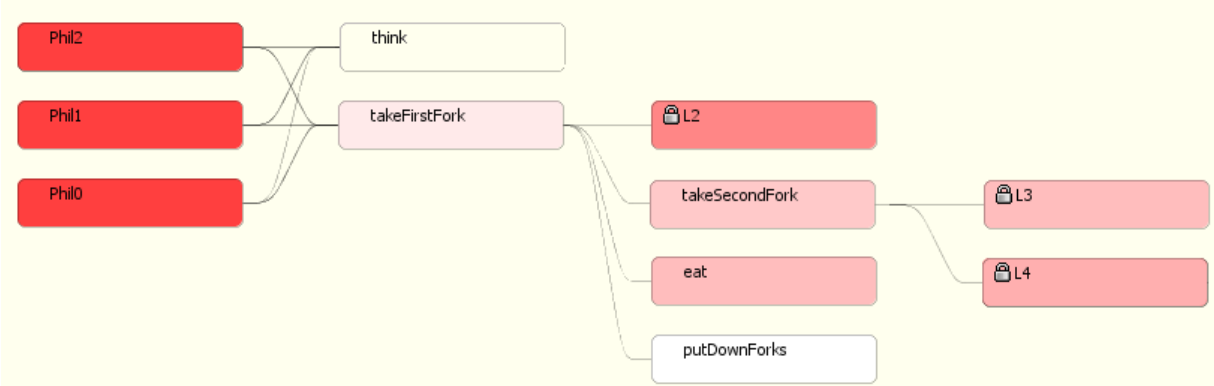


21 pav. Teisingos pietaujančių filosofų programa gijos

Gijų grafike matome, kad programoje veikia trys gijos, kurių pavadinimai: Phil0, Phil1 ir Phil2. Gijos darbą pradeda vienu metu atlikdamos tą patį galvojimo veiksmą (metodas *think),* t.y. visos gijos yra laukimo (*timed_waiting*) būsenoje. Taigi pirmasis filosofas galvoja 4 sekundes, antrasis bei trečiasis 2 sekundes. Toliau trečioji gija patenka į kritinę sekciją – sritį apsaugotą užraktais. Kitaip tariant filosofas *Phil2* paima abi šakutes ir pradeda valgyti. Antroji gija mėgina patekti į savo kritinę sekciją, tačiau dėl užrakinto užrakto pereina į laukimo būseną (*waiting*) - filosofas *Phil1* negali pradėti valgyti, nes užimta jo antroji šakutė. Tuo tarpu filosofas *Phil0* laukia pirmosios savo šakutės. Ją gavęs pradeda valgyti, todėl filosofas *Phil1* vis dar laukia antrosios šakutės. Kai *Phil0* padeda abi šakutes tik

tada *Phil1* pradeda valgyti. Taigi akivaizdu, jog programos darbas sinchronizuotas ir kad priėjimas prie kritinės sekcijos yra teisingas.

Tai galima patvirtinti išanalizavus programos grafo langą. Programos grafe pateikiamos pačios gijos, jų vykdomi metodai bei tvarka. Matome, jog programoje veikė trys gijos: *Phil0, Phil1* ir *Phil2*. Matome, kad kiekvienas filosofas iš pradžių galvojo. Vėliau paėmėmė šakutę apsaugotą užraktu L2, tuomet paėmė šakutę apsaugotą užraktų L3 arba L4, tuomet valgė ir padėjo šakutes į vietą.

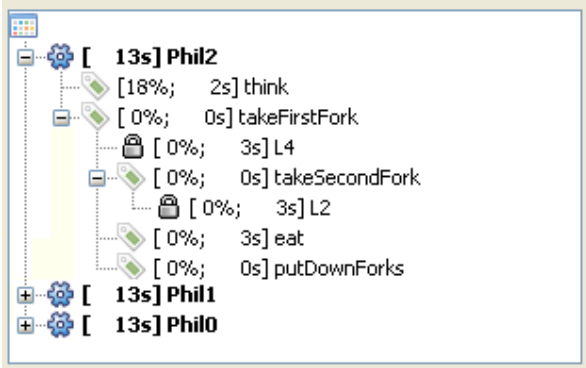


22 pav. Pietaujančių filosofų programos grafas

Programos struktūros lange matome, jog sistemoje užfiksuoti trys užraktai. Programos struktūroje pateikiama programos klasių struktūrą. Matome, jog stebima programa sudaryta iš dviejų klasių. Klasė *Philosofer* susideda iš 6 metodų. Parodomas kiekvieno iš metodų iškvietimų skaičius bei vidutinis vykdymo laikas. Jeigu šių skaičių vietoje rodomas brūkšniukas, metodo vykdymas dar nebaigtas.

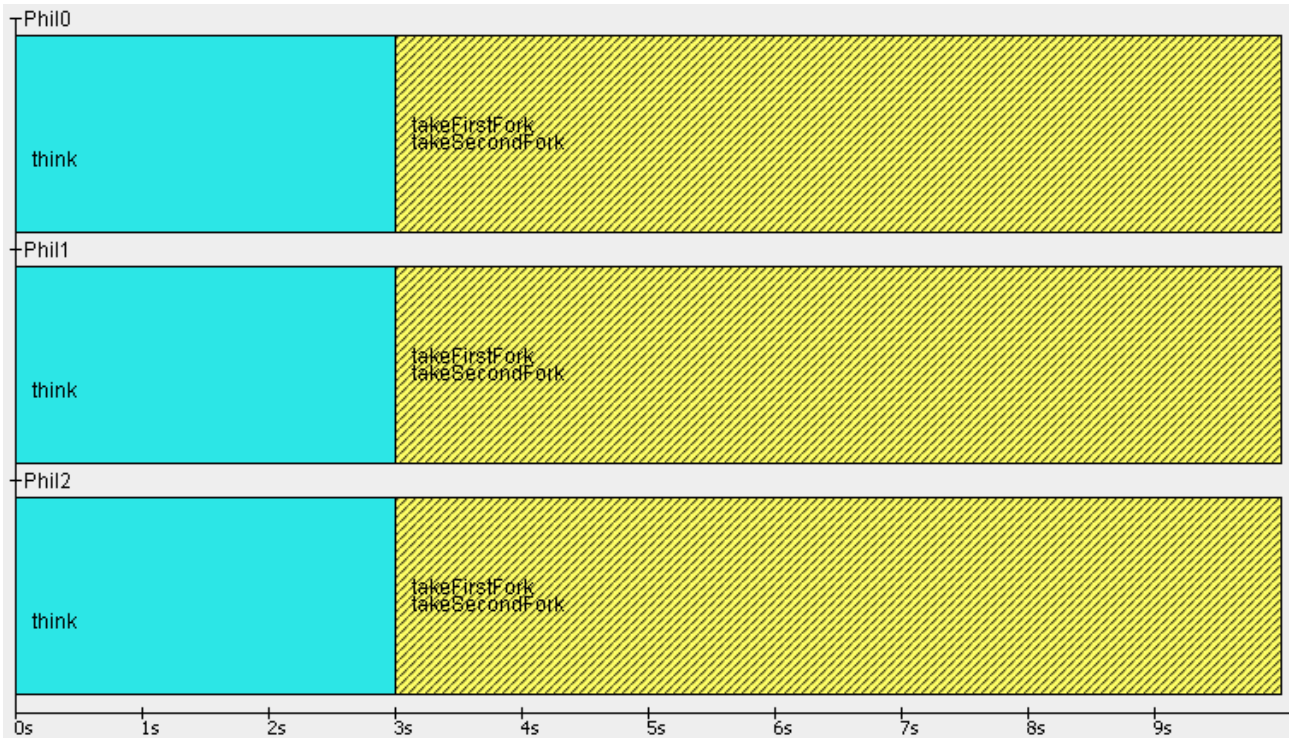
23 pav. Pietaujančių filosofų programos struktūra

Metodų kvietimo lange, galime matyti kokie metodai buvo kviesti, kiek laiko užtruko jų vykdymas bei kokie užraktai buvo įgyti vykdymo metu. Matome, jog buvo vykdytos trys gijos. Filosofas *Phil*1 galvojo, tuomet paėmė pirmąją šakutę ir užrakino užraktą L4. Tuomet paėmė antrą šakutę ir užrakino užraktą *L2.* Tuomet valgė bei padėjo abi šakutes.



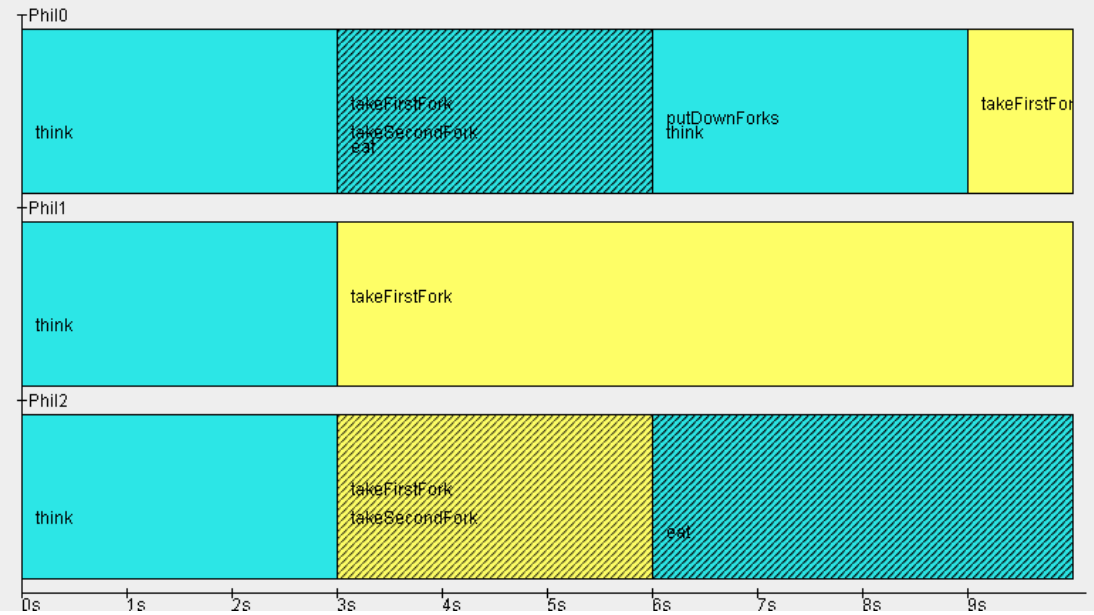
24 pav. Pietaujančių filosofų metodų kvietimas

Jeigu visi filosofai paima savo pirmąją šakutę vienu metu susidaro aklavietė. Ši problema akivaizdžiai pastebima įrankio pagrindiniame lange: matome, jog visi metodai pasiėmė pirmąją šakutę ir laukia antrosios, kurios yra užimtos.



25 pav. Aklavietė pietaujantys filosofai

Kita, galinti susidaryti problema – badavimas. Taip įvyksta tuomet, jeigu du filosofai be perstojo valgo, o vienas taip ir negauna šakutės. Ši problema pavaizduota žemiau pateiktame paveikslėlyje. Filosofas *Phil0* paima šakutes ir valgo. Tuo tarpu filosofas *Phil2* paima laisvą šakutę ir laukia kitos. Filosofas *Phil1* bando paimti pirmąją šakutę, tačiau ji užimta, todėl laukia. Filosofui *Phil0* atlaisvinus šakutes vieną iš jų paima *Phil2*, todėl *Phil1* toliau laukia. Šie veiksmai cikliški, todėl filosofas *Phil1* badauja.



26 pav. Pietaujantys filosofai badavimas

Tyrimo metu sudarytos 8 testinės programos. Kiekviena iš jų leista po penkis kartus. Testavimo rezultatai apibendrinami lentele:

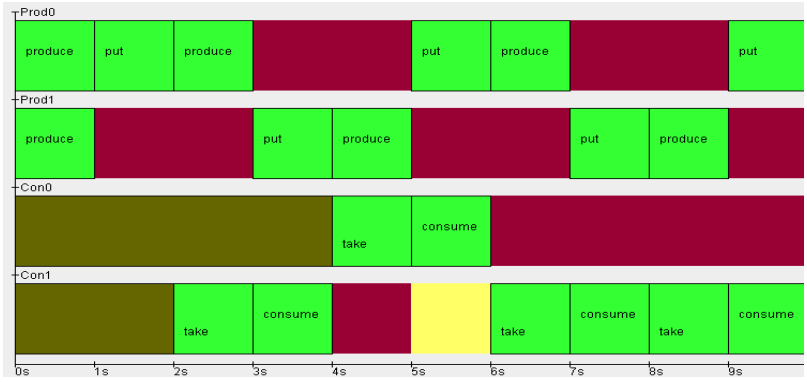
7 lentelė Pietaujančių filosofų tyrimo rezultatai

Programos nr.	Klaidos pobūdis	Klaida aptikta
1	Aklavietė	+
2	Aklavietė	+
3	Badavimas	+
4	Badavimas	+
5	Badavimas	+
6	Aklavietė	+
7	Gija nutraukiama	+
8	Aklavietė	+

Taigi rezultatai parodo, jog naudojantis sukurtu įrankiu galime aptikti lygiagretumo problemas bei atsekti jų priežastis.

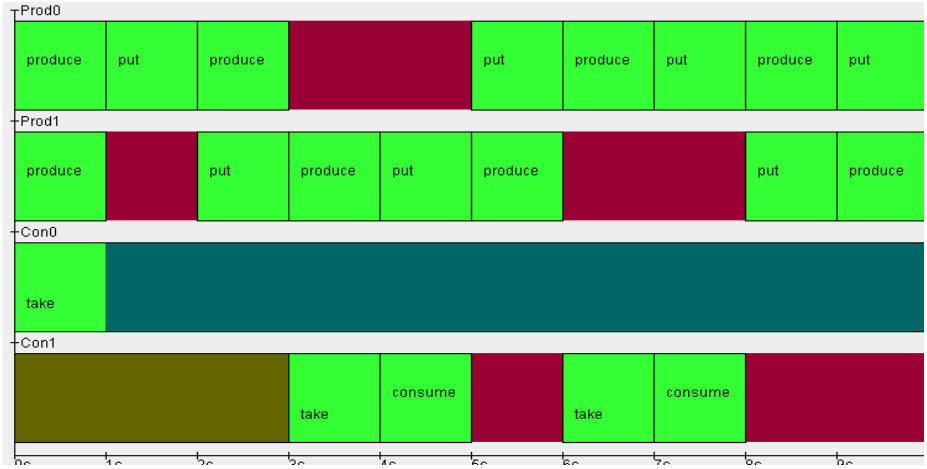
5.5. Gamintojo-vartotojo simuliacinės programos tyrimas

Tyrimo dalyje aptarta gamintojo-vartotojo problema bei sudaryta ją simuliuojanti programa (išeities kodai pateikti 2 priede). Atliekama paruoštos programos vykdymas bei analizė realizuotu lygiagrečių programų stebėjimo įrankiu. Pateikiamas tiek korektiškos, tiek nekorektiškos sinchronizacijos atvaizdavimas sukurtame įrankyje.



27 pav. Gamintojo-vartotojo simuliacinė programa teisinga sinchronizacija

Matome, kad pirmąją sekundę *Prod0* pagamina gaminį ir jį įrašo į buferį. Tuo tarpu *Prod1* taip pat pagamina gaminį ir yra užblokuojamas, kadangi *Prod0* įrašinėja į buferį. Tuo tarpu vartotojai yra nauji. Antrąją sekundę vartotojas *Con1* paima gaminį ir jį vartoja. Tik tuomet *Prod1* gamintojas trečiąją sekundę gali įrašyti gaminį į buferį. Vėl pagaminęs gaminį gamintojas *Prod1* yra blokuojamas, kol atsilaisvins buferis. Vartotojas *Con0* paima gaminį iš buferio ir šis tampa tuščias. Tuomet vartotojas *Con1* bando paimti gaminį, bet kadangi buferis tuščias jis laukia. Gavęs pranešimą iš *Prod0* jis toliau tęsia darbą. Taigi gijos sinchronizuotos teisingai.



28 pav. Gamintojo-vartotojo simuliacinė programa neteisinga sinchronizacija

Aukščiau pateiktame paveikslėlyje pavaizduota neteisinga gijų sinchronizacija. Pradžioje gija *Con0* bando iš buferio paimti gaminį. Tačiau nė vienas gamintojas į jį duomenų dar neįrašė. Dėl to gijos darbas iškart nutraukiamas.

Tyrimo metu sudarytos 5 klaidingos testinės programos. Kiekviena iš jų leista po penkis kartus. Testavimo rezultatai apibendrinami lentele:

8 lentelė Gamintojo-vartotojo tyrimo rezultatai

Programos nr.	Klaidos pobūdis	Klaida aptikta
1	Kova dėl resursų	+
2	Badavimas	+
3	Kova dėl resursų	+
4	Kova dėl resursų	+
5	Gija nutraukiama	+

Kaip matome rezultatų lentelėje įrankis sėkmingai leidžia aptikti kovos dėl resursų problemas. Tiesa, ši problema pasireiškia ne kiekvienu programos vykdymo metu, todėl negalime teigti, jog aptinkamumas 100%.

5.6. Eksperimento rezultatai

Atliktas eksperimentas parodė, kaip sukurtą įrankį galima pritaikyti lygiagrečių problemų sprendimui bei aptikimui. Aptartos bei pademonstruotos dvi klasikinės lygiagrečių sistemų problemos, ištirti keli sudaryti simuliacinių programų mutantai. Testavimo rezultatai parodo, kad įrankio pagalba aptinkama dauguma gijų sinchronizacijos klaidų.

Sukurtas įrankis įvertinamas pagal analizės dalyje apibrėžtus kriterijus. Rezultatai apibendrinami lentele, pateikiant ir analizės metu gautus rezultatus:

	Java ConTest	**Visual Threads**	**Jlint**	**Rivet**	**Sukurtas įrankis**
Programos stebėjimas realiu laiku	-	+	-	+	Paleidus įrankį pagrindiniame lange iškart matome stebimoje programoje veikiančias gijas, jų būsenas.
Panaudojimo paprastumas	+	+	+	-	Panaudoti gali ir nepatyręs programuotojas, nenaudojama jokia papildoma semantika.
Galimybė stebėti atskiras gijas	-	-	-	-	Įrankyje matomi gijos vykdomi metodai bei būsenos.
Kovos dėl resursų aptikimas	-	-	+	+	Kova dėl resursų pastebima dėl to, kad rodomos gijų būsenos.
Aklavietės aptikimas	+	+	+	+	Aklavietė lengvai pastebima, nes rodomos sinchronizuotos programos sekcijos.
Užsiciklinimų aptikimas	-	+	-	+	Užsiciklinimai lengvai pastebimi, nes realiu laiku rodomos gijų būsenos bei gijų sinchronizacija.
Rezultatų pernešamumas	-	-	-	-	Rezultatus galime eksportuoti ir importuoti kaip xml failą.
Konfigūravimo paprastumas	+	+	+	-	Konfigūracija atliekama iš grafinės sąsajos.
Rezultatų išsamumas	-	-	-	-	Pateikiami trijų grafinių formų rezultatai (programos gijos, kviečiami metodai ir programos grafas) bei papildoma informacija.
Nepriklausomumas nuo aplinkos	+	+	+	-	Įrankis nepririštas prie jokios integracinės aplinkos.

6. IŠVADOS

1. Atlikta analizė parodo, jog egzistuojantys įrankiai gali būti suskirstyti į tris kategorijas: įrankiai specializuoti nuoseklioms programos, įrankiai statiškai analizuojantys išeities kodus bei aptinkantys potencialias klaidas bei įrankiai formuojantys trasavimo bylas. Analizės metu nustatyta, jog įrankiai maksimaliai naudingi juos naudojant kartu, tačiau tai nėra patogu.
2. Analizės metu nustatyta, jog reikalingas įrankis, leidžiantis stebėti programas realiu laiku bei padedantis aptikti lygiagrečių programų sinchronizacijos klaidas.
3. Įrankis suprojektuotas kaip Java agentas, kadangi tokiu būdu nereikalingas vartotojo įsikišimas bei stebimos programos veikimas nėra iškraipomas.
4. Tyrimo metu aptartos dvi klasikinės lygiagretaus programavimo problemos, sukurtos jų simuliacinės programos bei sudaryti šių programų mutantai. Apibrėžta, kokias lygiagretaus programavimo klaidas jos padengia.
5. Eksperimento metu, testines programas leidžiant po kelis kartus, nustatyta, jog sukurtas įrankis efektyviai padeda aptikti gijų sinchronizacijos sukeltas klaidas.
6. Sukurtas įrankis vaizduoja gijų vykdomus veiksmus bei būsenas laiko juostoje, todėl jų palyginimas nereikalauja sudėtingų analizės darbų.
7. Sukurtas įrankis įvertintas pagal įsivestus kriterijus bei nustatyta, jog yra pranašesnis už kitus analogiškus sprendimus.

7. LITERATŪRA

[1] H.M. Jarvinen, M. Tiusanen, A.T. Virtanene *Convit, a Tool for Learning Concurrent Programming* [Suomija], 2003 [žiūrėta: 2008-11-04] Prieiga per internetą: <http://www.cs.tut.fi/~edge/convit_paperi.pdf>

[2] J. Zhao *Slicing Concurrent Java Programs* [Japan], 2005 [žiūrėta: 2008-11-04] Prieiga per internetą: <http://stap.sjtu.edu.cn/pdf/1999/Slicing%20Concurrent%20Java%20Programs.pdf>

[3] A. Ho, S. Smith, S.Hand *On deadlocks, livelocks and forward progress,* techninė ataskaita[Anglija], Kembridžo universitetas, 2005 p. 4-5 [žiūrėta: 2008-11-04] Preiga per internetą: < http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-633.pdf>

[4] B. Long *Testing Concurrent Java Components* [Kvinslendas] : disertacija. Kvinslendo universitetas, 2005. p. 5-7

[5] A. Cyrille *Finding faults in multi-threaded programs* : magistro darbas. Šveicarijos federalinis technologijų institutas, 2001. p. 2-3., 15-18, 72-81

[6] A. Bordelon *Developing a scalable, extensible parallel performance analysis toolkit* [Teksasas] : magistro darbas. Rice universitetas, 2007. p. 5-16

[7] K. Zhang, T. Hintzt, X. Ma *The Role of Graphics in Parallel Program Development*. Journal of Visual Languages and Computing (1999), Nr.10, p. 217- 218

[8] BUSTARD, D. W. *Concepts of Concurrent Programming,* 1990. 48 p.

[9] D.L Bruening *Systematic testing of multithreaded java programs:* magistro darbas. Masačusetso technologijų institutas, [Masačusetsas] 1999. p. 67-71

[10] D.Y.Cheng *A Survey of parallel programming tools* [Kalifornija]: NASA Ames tyrimų centro tyrimo medžiaga, 1999 [žiūrėta: 2008-11-04]. P Prieiga per internetą: <http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19910018560_1991018560.pdf>.

[11] J. J. Harrow, *Runtime Checking of Multithreaded Applications with Visual Threads*: Compaq Computer Corporation dokumentacija, [Nashua], 1999. p. 6-11

[12] Jlint įrankio dokumentacija. [Bostonas] : 2005 [žiūrėta: 2008-10-18]. Prieiga per internetą: <http://artho.com/jlint/manual.html>.

[13] Java ConTest įrankio dokumentacija. 2005 [žiūrėta: 2008-10-18]. Prieiga per internetą <http://www.alphaworks.ibm.com/tech/contest>

[14] A. Riškus *Programavimas Java* [Kaunas] : 2007 [žiūrėta: 2010-05-04]. Prieiga per internetą: <*www.lksoft.lt/javosivadas/Programavimas%20Java/1%20skyrius.doc*>.

[15] P. Eugster, *Java Virtual Machine with Rollback Procedure allowing Systematic and Exhaustive Testing of Multi-threaded Java Programs*: magistrinis darbas, [Ciurichas], 2003 p.26-27

[16] J. Plank, R. Wolski paskaitų užrašai, Teneso universitetas [žiūrėta: 2008-11-04] Preiga per internetą: <http://www.cs.utk.edu/~plank/plank/classes/cs560/560/notes/Dphil/lecture.html>

[17] C.-K. Shene *Multithreaded Programming with ThreadMentor:* elektroninė knyga, Masačusetso technologijų institutas [Masačusetsas], 2007 Prieiga per internetą: <http://www.cs.mtu.edu/~shene/NSF-3/e-Book/MUTEX/TM-example-philos-1.html>

TERMINŲ IR SANTRUMPŲ ŽODYNAS

Procesorius – tai techninis įtaisas, vykdantis programos kodą.

RAM – laikinoji kompiuterio atmintis.

Gija,procesas – tai programos kodo gabalas, nuosekliai vykdomas viename procesoriuje.

Kritinė sekcija - tai programos dalis, kurioje atliekami veiksmai su bendrai naudojamais duomenimis.

Aklavietė – tai sinchronizacijos problema, kurios metu gija laukia įvykio iš kitos gijos, kuris niekuomet neįvyks.

Užraktas - objektas, saugantis įėjimą į kodo sekcijas, kurių vykdymas vienu metu yra negalimas arba apribotas. Užraktas turi bent vieną užrakinimo ir bent vieną atrakinimo metodą.

IDE (*Integrated Development Environment*) – integruota kūrimo aplinka. Tai programos, turinčios daug priemonių, palengvinančių ir pagreitinančių programinės įrangos kūrimo procesą.

Java – objektiškai orientuota programavimo kalba, turinti klases lygiagretumui palaikyti.

UML (Unified Modelling Language) – modeliavimo kalba, naudojama objektiškai orientuotame projektavime.

HTTP – pagrindinis žiniatinklio užklausos - atsakymo protokolas, jungiantis klientą ir serverį.

Baitkodas – sukompiliuota programa arba instrukcijų rinkinys vykdomas virtualioje mašinoje.

Agentas – speciali biblioteka, leidžianti keisti klasių baitkodą jų užkrovino į virtualią mašiną metu.

XML – bendros paskirties duomenų struktūrų bei jų turinio aprašomoji kalba. Pagrindinė XML kalbos paskirtis yra užtikrinti lengvesnį duomenų keitimąsi tarp skirtingo tipo sistemų,

JVM(Java virtuali mašina) – tai speciali *Sun Microsystems* firmos programa **–** baitkodo interpretatorius, **–** parašyta kiekvienai operacinei sistemai atskirai.

Eclipse – integruota kūrimo aplinka, skirta programų Java programavimo kalba kūrimui ir vystymui. Plačiai paplitusi dėl įskiepių gausos ir paprasto diegimas.

8. PRIEDAI

1 PRIEDAS. PIETAUJANČIŲ FILOSOFŲ PROGRAMOS IŠEITIES KODAI

```
public class Test {
    private static final int count = 3;
    public static void main(String[] args) {
      ReentrantLock[] forks = new ReentrantLock[count];
      for ( int i = 0; i < count; i++ )
          forks[i] = new ReentrantLock();
          Philosopher[] philosophers = new Philosopher[count];
          for ( int i = 0; i < count; i++ ) {
              ReentrantLock leftFork = forks[i];
              ReentrantLock rightFork = forks[(i+1)%count];
              philosophers[i] = new Philosopher( i, leftFork, rightFork );
              philosophers[i].setName("Phil" + i);
              philosophers[i].start();
           }
      }
}

public class Philosopher extends Thread{
    private int number;
    private ReentrantLock leftFork;
    private ReentrantLock rightFork;
    public Philosopher( int number, ReentrantLock leftFork, ReentrantLock
    rightFork ){
        this.number = number;
        this.leftFork = leftFork;
        this.rightFork = rightFork;
    }
    public void run(){
        while ( true ) {
            think();
            takeFirstFork();
            takeSecondFork();
            eat();
            putDownForks();
        }
    }
    public void think(){
        int time = (int)(3000*(0.5 + Math.random()));
        try { sleep( time ); }
        catch ( InterruptedException e ) {}
    }
    public void takeFirstFork(){
        if ( number % 2 == 0 )
            leftFork.lock();
        else
            rightFork.lock();
    }
    public void takeSecondFork(){
        if ( number % 2 == 0 )
            rightFork.lock();
        else
            leftFork.lock();
```

```
    }


    public void eat(){
        int time = (int)(3000*(0.5 + Math.random()));
        try { sleep( time ); }
        catch ( InterruptedException e ) {}
    }
    public void putDownForks() {
        if ( leftFork.isLocked())
           leftFork.unlock();
        if ( rightFork.isLocked())
           rightFork.unlock();
    }
}
```

2 PRIEDAS. GAMINTOJO-VARTOTOJO PROGRAMOS IŠEITIES KODAI

```
public class ProdCons2 {

    protected LinkedList<Object> list = new LinkedList<Object>();
    protected int MAX = 1;
    protected boolean done = false;

    public class Producer extends Thread {
        public Producer(String name){
            this.setName(name);
        }
        public void run() {
            while (true) {
                Object justProduced = produce();
                synchronized (list) {
                    while (list.size() == MAX) {
                    try {
                        System.out.println("Producer WAITING");
                        list.wait();
                    } catch (InterruptedException ex) {}
                }

                list.addFirst(justProduced);
                list.notifyAll();
                if (done)
                    break;
                }
            }
        }

        private Object produce() {
            return (new Object());
        }
    }

    public class Consumer extends Thread {

        public Consumer(String name){
            this.setName(name);
        }
```

```
        public void run() {
            while (true) {
                Object obj = null;
                synchronized (list) {
                    while (list.size() == 0) {
                        try {
                            System.out.println("CONSUMER WAITING");
                            list.wait();
                        } catch (InterruptedException ex) {}
                    }
                    obj = list.removeLast();
                    list.notifyAll();
                    int len = list.size();
                    if (done)
                        break;
                    }
                    consume(obj);
                }
            }

            void consume(Object obj) {
                System.out.println("Consuming object " + obj);
            }
    }

    ProdCons2(int nP, int nC) {
        for (int i = 0; i < nP; i++){
            new Producer("Prod" + i).start();
        }
        for (int i = 0; i < nC; i++){
            new Consumer("Con" + i).start();
        }
    }

    public static void main(String[] args) throws {
        int numProducers = 2;
        int numConsumers = 2;
        ProdCons2 pc = new ProdCons2(numProducers, numConsumers);
        Thread.sleep(1 * 10000);
        synchronized (pc.list) {
            pc.done = true;
            pc.list.notifyAll();
        }

    }
}
```