KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ INŽINERIJOS STUDIJŲ PROGRAMA

JONAS PRAPUOLENIS

AUTOMATINIO PASKIRSTYTŲ SISTEMŲ TESTAVIMO METODO NAUDOJANČIO UML MODELIUS SUDARYMAS IR TYRIMAS

Magistro baigiamasis darbas

Darbo vadovas
doc. dr. T. Blažauskas

KAUNAS, 2013

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

PROGRAMŲ SISTEMŲ INŽINERIJOS STUDIJŲ PROGRAMA

JONAS PRAPUOLENIS

AUTOMATINIO PASKIRSTYTŲ SISTEMŲ TESTAVIMO METODO NAUDOJANČIO UML MODELIUS SUDARYMAS IR TYRIMAS

Magistro baigiamasis darbas

Darbo vadovas
doc. dr. T. Blažauskas

Recenzentas
doc. dr. Š. Packevičius

KAUNAS, 2013

AUTENTIŠKUMO PATVIRTINIMAS

Patvirtinu, kad įteikiamas baigiamasis darbas „Automatinio paskirstytų sistemų testavimo metodo naudojančio UML modelius sudarymas ir tyrimas“:

1. Autoriaus atliktas savarankiškai, jame nėra pateikta kitų autorių medžiagos kaip savos, nenurodant tikrojo šaltinio.
2. Nebuvo to paties autoriaus pristatytas ir gintas kitoje mokymo įstaigoje Lietuvoje ar užsienyje.
3. Nepateikia nuorodų į kitus darbus, jeigu jų medžiaga nėra naudota darbe.
4. Pateikia visą naudotos literatūros sąrašą.

____

(studento vardas, pavardė) (data) (parašas)

SANTRAUKA

Didžioji dalis programinės įrangos šiais laikais yra paskirstyta. Jos testavimas bei verifikavimas yra sudėtingas. Dažniausiai testavimas yra būtina, tačiau daug laiko bei išteklių reikalaujanti, programinės įrangos kūrimo proceso dalis. Automatiniai testavimo įrankiai palengvina paskirstytų sistemų testavimą.

Šiame darbe pateikiamas automatinis paskirstytų sistemų testavimo metodas naudojantis UML modelius, sukurtas įrankis šio metodo pagrindu bei atliktas eksperimentinis tyrimas.

Pirmoje dalyje aprašoma atlikta paskirstytų sistemų testavimo problemų analizė bei bėdos esant lygiagretumui bei lenktynių dėl resursų sąlygoms. Pateikiamos modelių tikrintuvo naudojimo galimybės atliekant paskirstytų sistemų testavimą bei UML modelių naudojimas testavimo tikslams. Antroje dalyje pateikiamas naudojamas metodas, o trečioje dalyje šį metodą realizuojančios programinės įrangos projektas. Programinė įranga sukurta Java kalba, naudojamas Java *PathFinder* modelių tikrintuvas dviejų klientų simuliavimui. Serveris veikia už modelio tikrintuvo ribų. UML klasių diagramos, su tam tikrų stereotipų pagalba, naudojamos serverio testavimo parametrams nurodyti.

Eksperimentinėje dalyje pateikiami atlikti bandymai su sukurta programine įranga. Testavimui panaudotos dvi paprastos paskirstytos sistemos. Panaudojant mutacinį testavimą sužinotas metodo aptinkamų mutantų kiekis, kuris yra palygintas su *JUnit* testų aptinkamu kiekiu. Atlikta aptiktų bei neaptiktų mutantų analizė parodė metodo privalumus bei trūkumus.

FORMATION AND RESEARCH OF AN AUTOMATED DISTRIBUTED SYSTEM TEST METHOD USING UML MODELS

SUMMARY

Most applications today communicate with other processes over a network. Testing and verification of such systems is complex. In many cases testing is an essential, but time and resource consuming activity in the software development process. Automated test methods are used to make testing of such distributed systems easier.

In this document an automated distributed system test method using UML models is presented. It uses a couple of clients communicating with server and is able to detect resource racing related failures.

The first section describes a research of distributed system testing issues, concurrency and resource racing error detection, and ways of using model checker for testing distributed systems as well as the use of UML models for test purposes. The second section presents a testing method used for the tool. Tool is described in the third section. It is based on Java and uses Java PathFinder model checker for simulating two clients. Server is run outside of model checker. UML class diagrams are used for defining server test parameters by applying certain stereotypes on methods being tested.

The investigation section describes a performed experiment with created tool. Two simple distributed systems are presented for mutation test purposes. Mutation testing is performed with defined mutation operators and results are analyzed. Additionally results are compared to basic JUnit tests.

TURINYS

Lentelių sąrašas **7**

Paveikslų sąrašas **8**

Terminų ir santrumpų sąrašas **9**

Įvadas **11**

1. Automatinio paskirstytų sistemų testavimo analizė **13**

1.1. Tyrimo sritis, objektas, problema ir darbo aktualumas *13*

1.2. Paskirstytų sistemų testavimo analizė *14*

1.2.1. Testavimas monitorių pagalba 14
1.2.2. Paskirstytų sistemų testavimas pasinaudojant modelių tikrintuvu 15
1.2.3. Modelių tikrintuve vykdomos programos sinchronizavimas su išorine aplinka 17
1.2.4. Problemos su kuriomis susiduriama atliekant paskirstytų sistemų testavimą 17

1.3. Testavimas naudojant UML modelius *19*

2. Automatinio paskirstytų sistemų testavimo metodo aprašymas **21**

2.1. Įvadas *21*

2.2. Problema *22*

2.3. Vykdymo medis *23*

2.4. UML klasės diagramų stereotipai *24*

2.5. Automatinio paskirstytų sistemų testavimo metodo sąlygos, įrankio veikimo principai bei problemos .. *26*

3. Automatinio paskirstytų sistemų testavimo įrankio projektas **29**

3.1. Įvadas *29*

3.2. Funkciniai reikalavimai *29*

3.3. Nefunkciniai reikalavimai *31*

3.4. Sistemos architektūra *32*

3.5. Sistemos naudojimas *37*

3.6. Sistemos kokybės užtikrinimas *37*

3.7. Sistemos ateities tobulinimo darbai *38*

4. Eksperimentai su sukurtu testavimo įrankiu **39**

4.1. Įvadas *39*

4.2. Eksperimento aprašyme naudojamos sąvokos *39*

4.3. Testinės sistemos programų mutantų generavimui *40*

4.4. Programų mutantų generavimas *40*

4.5. Rezultatai *42*

4.6. Atlikto eksperimento išvados *43*

5. Išvados **44**

6. Literatūra **45**

LENTELIŲ SĄRAŠAS

1 lentelė. **Stereotipų pseudokodas** **25**

2 lentelė. **XMI pateikimo panaudos atvejis** **30**

3 lentelė. **Kodo pateikimo panaudos atvejis** **30**

4 lentelė. **Serverio testų generavimo panaudos atvejis** **30**

5 lentelė. **Kliento simuliavimui pateikimo panaudos atvejis** **31**

6 lentelė. **Klientų simuliavimo panaudos atvejis** **31**

7 lentelė. **Naudojami operatoriai programų mutantų generavimui** **41**

8 lentelė. **Testinių sistemų parametrai** **41**

PAVEIKSLŲ SĄRAŠAS

1 pav. Kešavimo sluoksnio architektūra ... 16

2 pav. Metodo veikimo UML veiklos diagrama ... 21

3 pav. Kėdės rezervavimo UML sekų diagrama ... 22

4 pav. Blogai realizuotos sistemos su dviem klientais sekų diagrama ... 23

5 pav. Galimų vykdymo variantų medis ... 24

6 pav. UML klasių diagrama su stereotipais ... 24

7 pav. Testinių atvejų generavimo ir įterpimo veiklos diagrama ... 27

8 pav. Sistemos vartotojo panaudos atvejų diagrama ... 29

9 pav. Klasių diagrama vaizduojanti bendrą sistemos paketų bei funkcijų pasiskirstymą ... 32

10 pav. XMI failą analizuojančių klasių diagrama ... 33

11 pav. Serverio kodo ir XMI failo atitikimą tikrinančios klasės ... 33

12 pav. Testus generuojančių bei įterpiančių klasių diagrama ... 34

13 pav. Tinklo užklausų klausymą bei naujų šakų kūrimą atliekančios klasės ... 35

14 pav. Naudojamos JPF sisteminės klasės būsenų valdymui ... 36

15 pav. Vartotojo darbo veiklos diagrama ... 37

16 pav. Kėdžių rezervavimo sistemos eksperimento rezultatai kairėje, banko–bankomato dešinėje. ... 42

TERMINŲ IR SANTRUMPŲ SĄRAŠAS

Aklavietė	Situacija, kuomet du arba daugiau besivaržančių veiksmų laukia vienas kito pabaigos. Ši padėtis gali trukti neribotą laiką *(angl. Deadlock).*
Apache Tomcat	Java kalba parašytas daugiaplatformis HTTP serveris.
Būsenų kiekio sprogimas	Problema, su kuria susiduria modelių tikrintuvai. Didėjant programų apimčiai sparčiai didėja būsenų kiekis peržengdamas kompiuterio skaičiavimo galimybes. *(angl. State space explosion)*
Grįžimas tuo pačiu keliu	Modelių tikrintuve tikrinamos sistemos buvusios būsenos atstatymas. *(angl. Backtracking)*
JPF	Sistema skirta verifikuoti Java programoms. Veikdama kaip Java virtuali mašina vykdo programą bei gali saugoti bei atstatyti vykdymo būsenas. Dažniausiai naudojama kaip modelių tikrintuvas lygiagrečioms programoms. *(angl. Java PathFinder)*
JavaBeans	Perpanaudojamas Java PĮ komponentas savyje laikantis kitus objektus bei turintis metodų veikimo bei pavadinimų standartą.
JUnit	Atviro kodo sistema Java programų automatiniams testams kurti ir naudoti.
MDA	Modeliu paremta architektūra *(angl. Model Driven Architecture)*
Modelių tikrinimas	Metodas automatiškai verifikuojantis savybių korektiškumą baigtinių būsenų sistemoje. Modelių tikrinimo metu yra peržiūrimos visos galimos programos būsenos. Peržiūros metu patikrinamas atitikimas su specifikacija. *(angl. Model Checking)*
Monitorius	Programa ar jos dalis stebinti, prižiūrinti arba valdanti kitų programų veiklas. *(angl. Monitor)*
Mutacinis testavimas	Programinės įrangos testavimo metodas, kuris remiasi programos išeities kodo nedideliu modifikavimu. *(angl. Mutation Testing)*
OCL	Deklaratyvioji ribojimų kalba, skirta aprašyti taisykles taikomas UML modeliams. *(angl. Object Constraint Language)*
OMG	Tarptautinis, atviras, pelno nesiekiantis kompiuterių industrijos standartų konsortiumas. *(angl. Object Management Group)*

Orakulas	Mechanizmas naudojamas testavimo metu norint nustatyti ar testas praėjo sėkmingai ar ne. *(angl. Oracle)*
PĮ	Programinė įranga. *(angl. Software)*
Programa mutantas	Tam tikros originalios programos kopija turinti vieną pasikeitimą programos išeities tekste. *(angl. Mutant)*
SUT	Sistema, kurios veikimas yra testuojamas. *(angl. System under test)*
UML	Unifikuota modeliavimo kalba. *(angl. Unified Modeling Language)*
XMI	Formatas, skirtas keitimuisi UML modeliais tarp CASE įrankių, pagrįstų XML. *(angl. XML Metadata Interchange)*
XML	Bendros paskirties duomenų struktūrų bei jų turinio aprašomoji kalba *(angl. Extensible Markup Language).*

ĮVADAS

Temos aktualumas. Kompiuterių pagalba vykdomos beveik visos pasaulio bankų operacijos, valdomi lėktuvų skrydžiai, gamybos procesai bei gydymo aparatai. Kaip žinome iš kelių praeitų dešimtmečių, programinės įrangos klaidos šiose sistemose atneša didžiulius nuostolius ar netgi prarastas gyvybes [10].

Šiais laikais kuriama daug programinės įrangos, kuri yra paskirstyta. Tokios sistemos beveik visada yra sudėtingos. Jei paprastos, nepaskirstytos sistemos veikia viename kompiuteryje, tai paskirstytos sistemos turi kelis komunikuojančius komponentus, kurie gali veikti skirtinguose kompiuteriuose. Kai kurie paskirstytų sistemų komponentai gali būti kompiuteriuose, kurie yra tame pačiame kambaryje, gretimose patalpose, kitame mieste, arba net kitoje pasaulio pusėje. Taigi tokių sistemų testavimas ir verifikacija būna sudėtingi, kas sąlygoja didelę šių procesų kainą.

Norint palengvinti testavimą, ir, tuo pačiu sumažint kainą, nagrinėjami automatinio testavimo metodai, kuriami įrankiai realizuojantys šiuos metodus. Padedant šiems įrankiams testuotojai gali aptikti daugiau klaidų per mažesnį laiko tarpą. Tokiu būdu išleidžiama programinė įranga yra aukštesnės kokybės, o testavimo kaina yra sąlyginai sumažinama.

Paskirstytos sistemos turi daug testuojamų aspektų, kuriuos yra aprašęs ne vienas autorius [6, 7]. Išskiriamas vykdymo greitis (angl. *latency*), atsparumas klaidoms (angl. *partial failure*), vykdymo aplinkos bėdos, saugumas bei lygiagretumas (angl. *concurrency*) [6]. Pastaroji problema yra gerai žinoma ir išnagrinėta programinę įrangą vykdant per kelias gijas arba elektroninėse schemose. Šiame darbe nagrinėjama į gijų lygiagretumą panaši problema, kuomet klientai varžosi dėl resursų esančių serveryje.

Tyrimo objektas – paskirstytų sistemų automatinis testavimas.

Tyrimo tikslas – tobulinti automatinį paskirstytų sistemų testavimą, automatiškai aptikti klaidas atsirandančias dėl varžymosi dėl resursų sąlygų, naudoti sistemos UML modelį testuojamų vietų nustatymui.

Tyrimo uždaviniai:

Išnagrinėti paskirstytų sistemų testavimo problemas.

Ištirti esamus automatizuotus paskirstytų sistemų testavimo būdus.

Pateikti automatinio testavimo metodą

Realizuoti automatinio paskirstytų sistemų testavimo metodą.

Eksperimentiškai parodyti automatinio testavimo rezultatus.

Tyrimo problema – paskirstytų sistemų testavimas dažnai vykdomas su įrankiais nepritaikytais tokioms sistemoms. Kai kuriais atvejais tokių sistemų testavimas vykdomas rankiniu būdu. Egzistuojančių automatinių priemonių neužtenka kokybiškam paskirstytų sistemų testavimui.

Antrame darbo skyriuje pateiktos literatūroje aprašomos paskirstytų sistemų testavimo problemos, nagrinėjamas testavimas pasinaudojant monitoriais bei modelio tikrintuvo naudojimas paskirstytų sistemų testavimui. Apžvelgiamas UML modelių naudojimas testavime. Trečiame skyriuje pateiktas automatinio paskirstytų sistemų testavimo metodas, kuris remiasi R. Jasaičio atliktais darbais [1, 2]. Ketvirtame skyriuje pateiktas šį metodą realizuojantis automatinio paskirstytų sistemų testavimo naudojančio UML modelius projektas. Penktame skyriuje pateiktas eksperimentas atliktas su sukurtu įrankiu, jo sąlygos, rezultatai ir išvados. Šeštame skyriuje pateiktos bendros darbo išvados.

1. AUTOMATINIO PASKIRSTYTŲ SISTEMŲ TESTAVIMO ANALIZĖ

1.1. Tyrimo sritis, objektas, problema ir darbo aktualumas

Pagrindinis darbo tikslas – realizuoti programinę įrangą, gebančią automatiškai testuoti paskirstytas sistemas panaudojant UML modelį. Eksperimentiškai ištirti sukurtos programinės įrangos naudojamo testavimo metodo efektyvumą. Kuriama programinė įranga turėtų būti realizuota Java programavimo kalba. Taigi šio darbo tyrimo sritis apims paskirstytų sistemų testavimo automatizavimo būdų analizę.

Šiais laikais kuo toliau tuo dažniau kuriamos programos turi kelis komponentus, sudarytus iš kliento ir serverio. Taip vyksta dėl didėjančio interneto greičio ir jo padengiamumo. Dalis programinės įrangos funkcionalumo yra perkeliama į serverius, kurie gali sudėtingus skaičiavimus vykdyti greitai, kas yra labai svarbu atsižvelgiant į populiarėjančius išmaniuosius mobiliuosius įrenginius.

Kliento–serverio architektūra paremtos sistemos dažnai turi bėdų dėl kelių klientų, kurie beveik tuo pačiu metu kreipiasi į serverį [4]. Negalima garantuoti, kad klientai nepaveiks serverio nenumatytai, net jeigu serveris vykdomas ant vienos gijos [1]. Visgi didžioji dalis serverių atsako klientams naudojant paskirstytos kompiuterijos savybes, kaip, kelias gijas.

Yra daug literatūros kaip atliekamas paprastų sistemų testavimas. Tačiau pasiūlymų kaip testuoti kliento–serverio architektūra paremtas arba lygiagrečias sistemas nėra daug. O esami pasiūlymai dažnai nėra pakankamai detalūs bei pritaikomi [6]. Automatinių paskirstytų sistemų testavimo metodų, kurie spręstų resursų lenktynių problemą beveik nėra. Todėl šiame skyriuje bus daugiau dėmesio skiriama kliento–serverio architektūra paremtų sistemų bėdoms bei jų sprendimams, kurie yra lygiagretūs resursų lenktynių problemai.

1.2. Paskirstytų sistemų testavimo analizė

1.2.1. Testavimas monitorių pagalba

Brinch Hansen 1978 savo darbe [8] aprašo metodą, kurį naudojant testuojami monitorių moduliai. Pastarieji valdo sąveiką tarp procesų lygiagrečiose programose. Padedant monitoriams testavimas atliekamas vykdant programą, kurioje procesai yra sinchronizuoti pagal laikrodį, taip padarant sąveikavimo sekas atkartojamas. Aprašomo testavimo žingsniai susideda iš šių veiksmų:

1. Programuotojas nustato sąlygas kiekvienai operacijai su monitorium, kurios privers kiekvieną operacijos šaką būti įvykdytai bent kartą.
2. Programuotojas suformuluoja monitorių kreipinių seką, kuri išbandys kiekvieną operaciją pasinaudojant nustatytomis sąlygomis.
3. Programuotojas sudaro aibę testinių procesų, kurie sąveikaus tarpusavyje taip, kaip nustatyta praeituose žingsniuose.
4. Galiausiai testinė programa įvykdoma ir jos išvestis yra palyginama su numatytąja išvestimi.

Brad Long ir Paul Strooper 2001 metais straipsnyje [6] pateikia vienos paskirstytos sistemos, susidedančios iš klientų, serverio ir duomenų bazės, testavimą. Serverio komponentui aprašomi atlikti testavimai susidedantys iš vienetų testavimo, lygiagretumo testavimo, integracinio testavimo bei sisteminio testavimo. Jie, pratęsdami Brich Hansen darbą, lygiagretumo testavimo metu naudoja Java monitorius.

Testavimo programa sukuria testuojamo serverio objektą (kaip monitorių). Bendrai sakant, monitorius šiuo atveju yra klasė turinti du sinchronizuotus metodus: pridėti į eilę ir gauti. Testavimo programa, pasinaudodama laikrodžiu, nustato gamintojo ir vartotojo vykdymo seką. Gamintojas sukuria lizdų ryšius, kurie pridedami į eilę monitoriuje. Paprasta testavimo programėlė pastoviai klausosi gamintojo ryšio užklausų. Vartotojas kviečia monitoriaus gauti metodą ryšio gavimui iš eilės. Tokiu būdu monitorius gali valdyti kreipinių eiliškumą. Klaidos aptinkamos jei gija nepabunda išvis, arba pabunda jai nenumatytu laiku.

Jų naudotas paskirstytų sistemų testavimo būdas, jų teigimu, buvo sėkmingai išbandytas pasinaudojant nedidele programa. Tačiau išsamesnių rezultatų darbe nėra pateikta, taigi nėra aišku kiek realiai šis testavimo būdas gali būti naudingas testuojant paskirstytas sistemas.

1.2.2. Paskirstytų sistemų testavimas pasinaudojant modelių tikrintuvu

Komunikuojančios tinklu programos dažniausiai yra kuriamos kaip lygiagrečios, naudojančios kelias gijas, kurios valdo tinklo ryšius. Gijų vykdymo tvarka yra atliekama operacinės sistemos ir negali būti kontroliuojama programuotojo. Dėl to programinės įrangos testavimo metu gali būti praleistos kai kurios klaidos atsirandančios esant tam tikrai vykdymo tvarkai. Vienas iš geresnių verifikavimo būdų yra modelių tikrinimas (angl. *model checking*). Kai kurie modelių tikrintuvai, kaip Java *PathFinder* (JPF) realiai vykdo programos kodą. Verifikavimo metu modelių tikrintuvas testuojamą sistemą gražina į praėjusią būseną ir tikrina kelis galimus vykdymo kelius, tokius kaip skirtinga gijų vykdymo tvarka ar skirtinga programos įvestis.

JPF modelių tikrintuvas gali simuliuoti nedeterminizmą [14]. Būdama kaip virtuali mašina, kuri vykdo Java kodą, vykdymo metu aptinka tam tikrus nedeterminuotus veiksmus, kaip gijų vykdymo tvarką ar įvestį. Norint simuliuoti visus nedeterministiškus veiksmus yra reikalingas šis funkcionalumas[14]:

1) **Grįžimas tuo pačiu keliu** (angl *backtracking*). Turi būti galimybė grįžti į prieš tai buvusią vykdymo būseną. Pasiekus ankstesnę būseną, galima pasukti kitu vykdymo keliu, taip patikrinant kitą vykdymo tvarką. Nors šį funkcionalumą galima pasiekti vykdant programą iš naujo, tačiau daug efektyvesnis būdas yra turėti išsaugotą būseną, į kurią galima tiesiogiai sugrįžti.
2) **Būsenų lyginimas**. Vykdymo būsena susideda iš krūvos (angl. heap) bei išsaugotos momentinės gijų eilės. Lyginant būsenas galima nustatyti ar ketinamas vykdyti kelias jau buvo vykdomas, taip atsikratant bereikalingo darbo.

Teoriškai modelių tikrintuvas yra atšiaurus, griežtas būdas – jei yra defektas, jis bus surastas. Tačiau toks tikrinimas yra apribotas verifikuojamos programos dydžiu dėl būsenų kiekio sprogimo efekto. Verifikuojant didelę programą staigiai didėja būsenų skaičius ir paprasčiausiai nebeužtenka skaičiuojamosios galios visų būsenų patikrinimui.

Būsenų skaičiaus sprogimo bėdą JPF sprendžia keliais būdais:

1) Leidžiant vartotojui susikonfigūruoti medžio paieškos algoritmą. Dėl šios priežasties JPF turi sąsają, kurią realizavus galima pasirašyti savo paieškos algoritmą.
2) Mažinant būsenų skaičių;
3) Mažinant vietą, kurią užima kiekviena būsena.

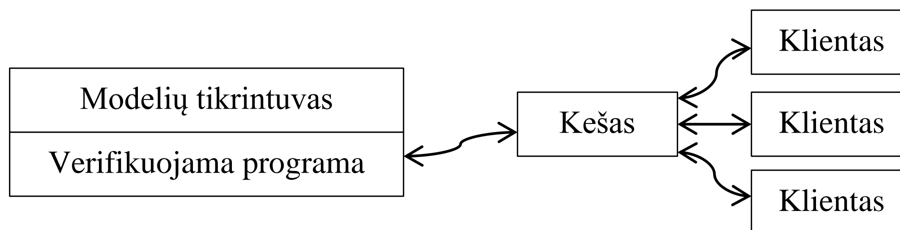
Verifikuoti paskirstytą sistemą su modelių tikrintuvu nėra visiškai paprasta. Paskirstyta sistema sudaryta iš kelių komponentų, kurie keičiasi informacija ir bendrauja vienas su kitu per tinklą. Modelių tikrintuvai dažniausiai apdoroja tik vieną procesą, o paskirstytų sistemų procesai yra keli ir gali būti išsimėtę po skirtingas aplinkas. Kai modelių tikrintuvas vykdo testuojamą procesą, kiti procesai vykdomi paprastai – neįeina į modelio tikrintuvo aplinką. Dėl šios priežasties pastarųjų būsena nėra gražinama kartu su verifikuojamu procesu. Tai gali sąlygoti išsiderinimą tarp šių procesų jei verifikuojama programa su jais komunikuoja [9]. Šis išsiderinimas turi dvi pasekmes[5]:

1) Verifikuojama programa iš naujo išsiųs informaciją po grįžimo tuo pačiu keliu. Tai sąlygos neteisingą išorinių, neverifikuojamų, programų funkcionalumą.
2) Po grįžimo į buvusią būseną, verifikuojama programa vėl tikėsis įvesties, tačiau jos negaus, nes išorinis procesas jos antra karta nesiunčia.

Keliuose darbuose [3, 5, 9, 13] yra aprašomi būdai kaip automatizuoti tokių sistemų testavimą.

Naudojant centralizavimo techniką [13] procesai yra automatiškai sujungiami į vieną, o tada vykdomas jų verifikavimas. Sujungimo metu kiekvienas procesas yra paverčiamas į giją, kurios paskiau paleidžiamos viename bendrame procese. Tokiu būdu modelių tikrinimo programos, kaip JPF, gali automatiškai verifikuoti visą sistemą, nes ji yra paversta į vieną procesą. Šiame viename procese tinklo komunikacijos yra paverčiamos į vidines programos komunikacijas įrankio pagalba. Ši technika susiduria su keliomis bėdomis. Viena problema yra, kad visi sistemos komponentai turi būti parašyti ta pačia programavimo kalba ir sukompiliuoti ant vienos platformos. Kita problema, kad bendravimas tarp gijų gaunamas labai sudėtingas, dėl ko patiriamas būsenų kiekio sprogimas ir verifikavimui reikia daug laiko ir atminties.

Kitas siūlomas būdas yra kešuoti įvesties ir išvesties operacijas verifikavimo vykdymo metu [4]. Čia daroma prielaida, kad visos įvesties ir išvesties operacijos yra deterministiškos.



1 pav. Kešavimo sluoksnio architektūra [3]

Šį metodą realizavus yra sukurtas ir naudojamas tinklo užklausų kešavimo įrankis [3] kurio principas yra valdyti ir kešuoti tinklo užklausas ateinančias iš modelio tikrintuvo į aplinką. Vienas procesas vykdomas modelio tikrintuve, o kiti procesai bendroje aplinkoje. Atsikartojančios įvesties bei išvesties operacijos ateinančios iš modelio tikrintuvo yra paslepiamos nuo kitų procesų. Kešavimo sluoksnis perima bet kokią tinklo komunikaciją ir nusprendžia ar tokį kreipinį daryti į išorę ar naudoti kešuotą atsakymą. 1 pav. atvaizduoja šio sluoksnio architektūrą.

1.2.3. Modelių tikrintuve vykdomos programos sinchronizavimas su išorine aplinka

Užklausų kešavimas yra tinkamas naudojimui kai serverio išvestis yra nedeterminuota, o kliento yra determinuota. Kitame darbe [9] yra pateiktas metodas, įvedantis klientų atskaitos taškų įrankį prie kešavimo. Juo pasinaudojant ir klientas, ir serveris gali komunikuoti nedeterminuotai. Būsenos yra išsaugomos kaip atskaitos taškai ir prie jų pereinama modelio tikrintuvui grįžus atgal buvusiu keliu.

Procesų būsenos išsaugojimui į atmintį bei atstatymui naudojami įvairūs virtualizacijos įrankiai. Šie įrankiai dažniausiai turi saugojimo bei atstatymo funkcijas. Kešavimo sluoksnis sujungiamas su virtualizacijos įrankiu[9]. Pateiktos trys procesų būsenų saugojimo strategijos, leidžiančios saugoti klientų būsenas pagal konkretų testavimo atvejį:

1) Visada saugoti;
2) Saugoti po kiekvieno tinklo komunikacijos veiksmo;
3) Saugoti po kiekvieno perėjimo aptikus nedeterminizmą.

Kešavimo sluoksnis gali nuspręsti, ar naudoti kešuotą atsakymą iš kliento, ar paimti vieną iš išsaugotų kliento būsenų ir ją tęsti, ar paleisti klientą per naują. O strategija parenkama pagal kliento tipą. Antrąją strategiją verta naudoti, kai klientas, vykdydamas užklausą, daro didelius skaičiavimus, nors atsakymas ir yra determinuotas. Jei skaičiavimą ilgiau užtrunka atlikti nei išsaugoti būseną, tuomet antrasis būdas paspartina testavimo eigą.

Klientų procesų būsenos išsaugojimo metodas įneša naują apribojimą – klientas turi būti paleistas specialioje, būseną galinčioje saugoti aplinkoje. Tai gali sąlygoti kitokį kliento veikimą nei įprastoje aplinkoje.

1.2.4. Problemos su kuriomis susiduriama atliekant paskirstytų sistemų testavimą

Sudipto Ghosh bei Adityja Mathur savo darbe [7] aprašo problemas, su kuriomis susiduriama vykdant paskirstytų komponentais paremtų sistemų testavimą. Šios problemos išvardintos kitame puslapyje.

- **Testų kriterijų adekvatumo dydžio keitimas.**
- **Persidengiantys testai komponentų integravimo metu.** Sistemos komponentai dažnai testuojami atskirai, o testuojant visą sistemą bendrai vėl iš dalies tikrinami šie komponentai. Čia atsiranda testavimo dubliavimas.
- **Išeities kodo prieinamumas.** Sistemos komponentai gali būti sukurti atskirai ir vienos ar kitos dalies išeities kodas nepasiekiamas testavimo metu.
- **Kalbos, platformos bei architektūros skirtumai.** Sistemos komponentai gali būti sukurti pasinaudojant skirtingomis programavimo kalbomis, tinkami skirtingoms operacinėm sistemom ar net pritaikyti kitokiai procesoriaus architektūrai.
- **Stebėjimas bei valdymo mechanizmas paskirstytų sistemų testavimo metu.** Testų vykdymas atliekamas keliais kompiuteriais tinklu. Tai atlikti yra daug sudėtingiau negu testuojant centralizuotą sistemą viename kompiuteryje. Reikia atsižvelgti į duomenų surinkimą bei saugojimą testavimo metu – jų yra daugiau nei įprasta dėl komponentų skaičiaus bei jų vykdymo trukmės.
- **Testavimo atkartojamumas**. Lygiagretūs skaičiavimai bei asinchroninė komunikacija yra dažnai pasitaikantys šiose sistemose. Taip pat nėra galimybės visiškai kontroliuoti šios dalykus. Tai sąlygoja testavimo atkartojamumo problemą, ir yra priežastis, kodėl sistemos komponentus reikia testuoti ne tik atskirai, bet ir kaip visumą.
- **Aklavietės bei lenktynių dėl resursų sąlygos**. Paskirstytos sistemos dažnai turi šias bėdas. Aklavietės gali atsirasti įvairiomis aplinkybėmis. Jei du komponentai tuo pačiu metu kreipiasi vienas į kitą, yra tikimybė, kad abu komponentai lauks atsakymo, o tik tada patys atsakys. Tai yra paprastas aklavietės pavyzdys. Šias problemas reiktų aptikti testavimo metu.
- **Sistemos greičio bei apimties testavimas**.
- **Tolerancijos klaidoms testavimas**. Kritinėms sistemoms kaip atominės elektrinės turi būti kuriama programinė įranga atspari klaidoms. Klaidas galima įterpti į sistemą norint tai ištestuoti. Susiduriama su dviem problemomis klaidų įterpimo metu – reikia pasiekti klaidos vietą, kad klaida būtų iššaukiama bei gali egzistuoti ilgas laiko tarpas, kartais ir kelios dienos, kol klaida padarys poveikį sistemai.

Darbe [7] šias testavimo problemas siūloma spręsti kuriant testus, apimančius tik būtiniausias sistemos dalis. Reikia apsibrėžti tikslią komponento sąsają ir komponentą ištestuoti būtent per ją. Šis testavimo metodas yra labai trumpai aprašytas ir nėra aiškiai pateiktas, tačiau jo esmė yra parašyti komponento sąsajos testus kurie padengtų ir metodus, ir klaidų valdymą metode.

1.3. Testavimas naudojant UML modelius

UML modeliai atspindi sistemą skirtingame detalumo lygyje. Vieni modeliai aprašo sistemą iš aukštesnio, abstraktesnio lygio, o kiti modeliai pateikia daug detalių. UML modeliai turi tam tikrus elementus, kaip aktoriai, panaudos atvejai, klasės, paketai bei diagramos. IBM pateikia šias modelių pritaikymų galimybes [16]:

- Vizualiai atvaizduoti norimą sukurti sistemą;
- Pateikti sistemos viziją klientams bei kolegoms;
- Sukurti ir patikrinti sistemos architektūrą;
- Naudoti UML diagramas kodo generavimui.

Tačiau UML modeliai naudojami ir sistemos testavimo metu. Specialios paskirties išplėtotinos notacijos, kaip stereotipai, leidžia intuityviau bei labiau tiesiogiai susieti sumodeliuotas esybes ir realizuotą sistemą [17]. Taigi atsiranda galimybė UML modelius naudoti kaip pagrindą atliekant automatizuotus testus.

Eddy Bernard straipsnyje [18] teigiama, kad modeliuojant sistemą ir norint modelį pritaikyti automatiniam testų generavimui reikia atsižvelgti ne tik į struktūrą, bet ir į sistemos elgseną. UML modeliai turi turėti aiškią prasmę tam, kad testų generatorius galėtų ją suprati. Tokiu būdu testų generatorius modelį gali naudoti kaip orakulą.

Straipsnyje [18] taip pat pateikiamas diagramų naudojimo variantas kuriuo buvo pasinaudota kuriant automatinio testavimo metodą:

- **Klasių diagramos** atspindi testuojamos sistemos duomenų modelį bei sistemos valdymo ir stebėjimo taškus, kurie pateikti kaip veiksmai klasėse.
- **Objektų diagramos** testų generavimui parodo pradinę sistemos būseną. Aprašomi objektai bei jų pradinės reikšmės, santykiai bei atributai.
- **Būsenų diagramos** papildo klasių diagramas parodydamos kokie yra numatyti kiekvienos klasės veiksmai. Tai yra atvaizduojama perėjimais tarp būsenų.
- **OCL** naudojamas klasių diagramose nurodyti laukiamą operacijų elgseną pasinaudojant sąlygomis prieš veiksmą bei sąlygomis po veiksmo. Taip pat naudojama būsenų diagramoje formalizuoti perėjimus tarp būsenų aprašant saugiklius.

Taigi klasių ir objektų diagramos aprašo sistemos veikimą statiškai, kai būsenų diagramos bei OCL pateikia dinaminį vaizdą.

UML modeliais paremtas testavimas dažnai būna sėkmingas dėl šių priežasčių [18]:

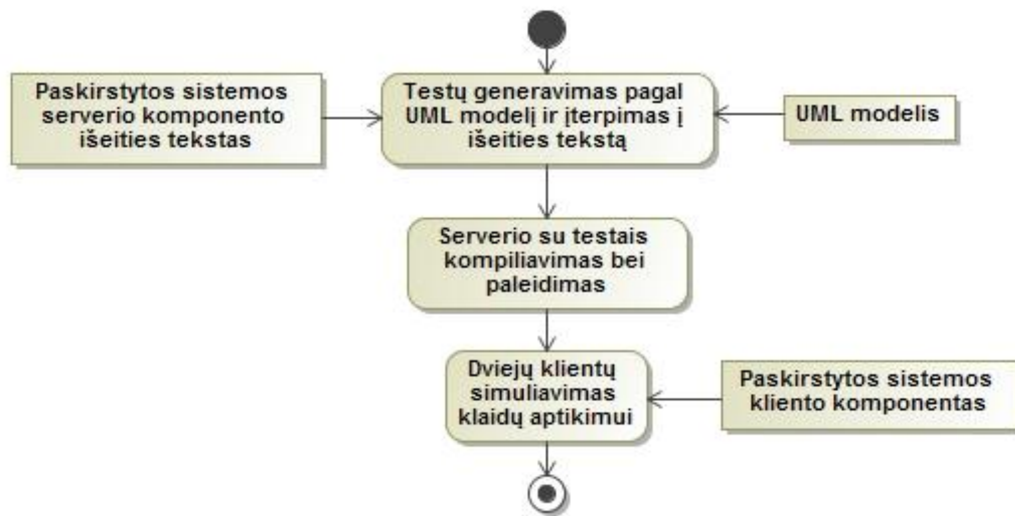
- Pakankamo kiekio modeliavimo notacijų, kurios sudaro galimybes kurti modelius orientuotus testams. UML turi įvairias OCL prieš ir po sąlygas, būsenų ir sekų diagramas, stereotipus bei kita, kas sudaro galimybę sukurti modelį žvelgiant iš įvairių kuriamos sistemos pusių ir aprašyti lauktinus sistemos veiksmus.
- Automatiškai sukurtų testinių atvejų tikslumo žvelgiant į testavimo tikslus.
- Galimybės sukurti testų generavimo bei vykdymo automatizavimą.

Minėti UML stereotipai yra vienas iš trijų modelio elementų, kurie suteikia UML kalbos papildymo galimybes. Šie elementai leidžia papildyti UML žodyną naujų modelių elementų sukūrimu iš esamų. Pridedamos papildomas savybės, kurios tinka tam tikrai probleminei sričiai spręsti. Šios savybės gali tiesiog palengvinti UML skaitomumą arba sukurti galimybę atlikti testavimo veiksmus. UML stereotipai yra žymimi tarp „<<“ ir „>>“ ženklų.

2. AUTOMATINIO PASKIRSTYTŲ SISTEMŲ TESTAVIMO METODO APRAŠYMAS

2.1. Įvadas

Darbe nagrinėjamas testavimo metodas orientuotas į automatinį paskirstytų sistemų varžymosi dėl resursų sąlygų aptikimą. Trumpai pateiksime metodo pagrindines dalis bei veikimą.



2 pav. Metodo veikimo UML veiklos diagrama

2 pav. trumpai pateiktas metodo veikimas bei jam reikalingi elementai. Iš jo matome, kad automatinio paskirstytų sistemų testavimo metodas sudarytas iš dviejų pagrindinių dalių:

1. Automatinio klientų simuliavimo
2. Klaidų serveryje aptikimo

UML modeliai naudojami nurodyti serverio metodų vykdymo apribojimus. Pagal šiuos apribojimus generuojami testiniai atvejai. Tokiu būdu išsprendžiama orakulo problema.

Automatinis klientų simuliavimas, pasinaudojant modelių tikrintuvu, vykdo tikrinimą bandant aptikti klaidas pagal sugeneruotus testinius atvejus. Šis klientų simuliavimas yra išskirtinis dėl užklausų į serverį vykdymo eiliškumo. Yra išbandomi visi galimi vykdymo variantai.

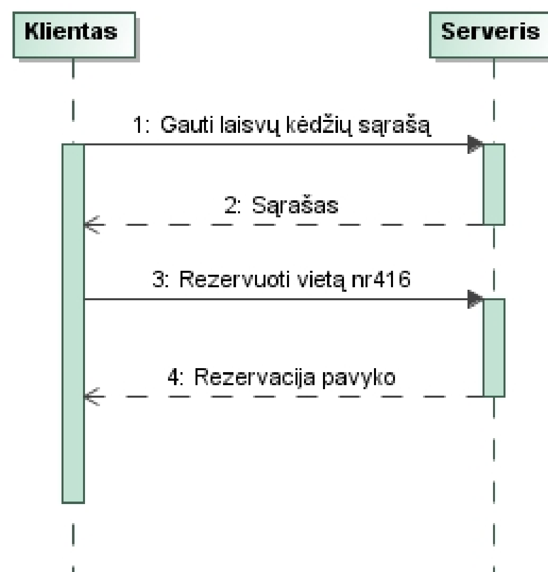
Šiame skyriuje bus detaliau apžvelgiamas nagrinėjamas klaidos tipas, pateikiamas klaidos pavyzdys, aprašomas klientų vykdymo simuliavimas. Detaliai pateikiamas UML modelių naudojimas bei pagal juos generuojami testiniai atvejai.

2.2. Problema

Norint geriau suprasti nagrinėjamą problemą pateikiame pavyzdinę sistemą. Ši sistema bus naudojama kaip pavyzdys visame darbe. Kėdžių rezervavimo sistema yra sudaryta iš kliento ir serverio. Sistemos darbo eiga yra tokia:

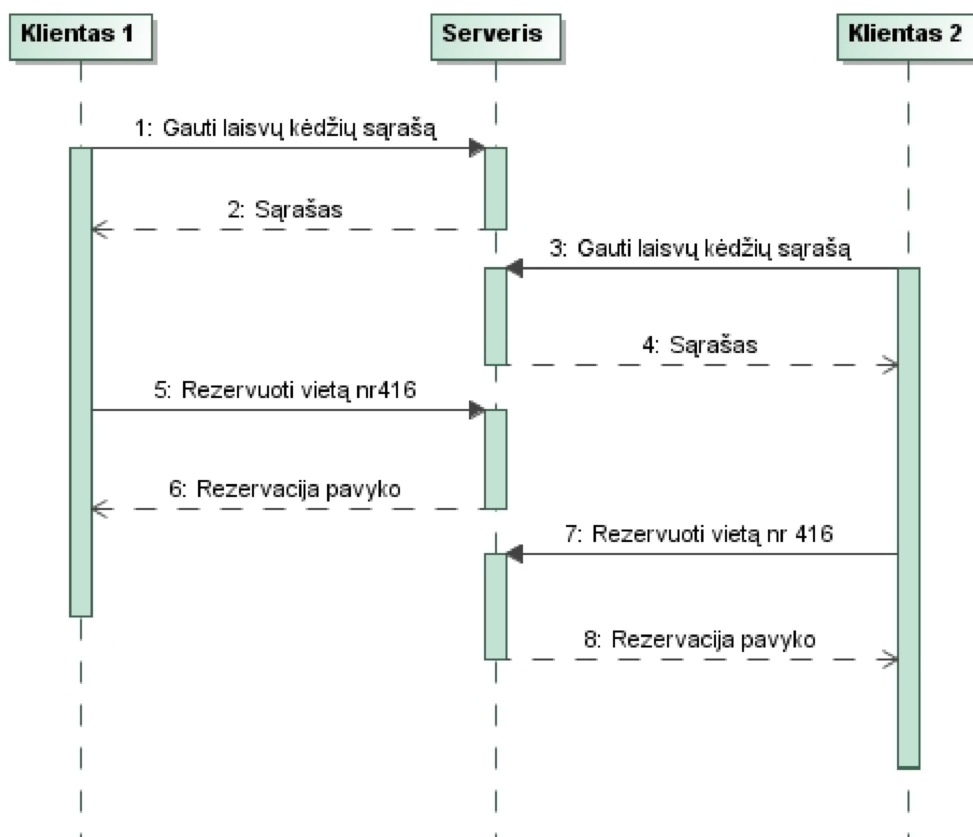
- Vartotojas paleidžia klientą;
- Vartotojas, kliento pagalba, pasitikrina kurios kėdės yra laisvos;
- Vartotojas, kliento pagalba, rezervuoja pasirinktą kėdę.

Ši darbo eiga tarp kliento ir serverio sistemoje sukurtų komunikaciją kuri, panaudojant UML sekų diagrama, pateikta 3 pav.



3 pav. Kėdės rezervavimo UML sekų diagrama

Paskirstytoje sistemoje, žinoma, gali būti keli klientai. Šie, jungdamiesi prie serverio, gali sudaryti įvairias jungimosi situacijas. Programuotojai ne visada numato šias situacijas – čia ir atsiranda lenktynės dėl resursų. 4 pav. pateiktas galimos klaidos serveryje pavyzdys. Nors serveris veikia teisingai esant vienam klientui, tačiau pateiktame pavyzdyje paaiškėja, kad serverio realizacija yra klaidinga prie tam tikros specifinės situacijos – kai sumaišomos klientų užklausos tarpusavyje. Čia abu klientai gauna atsakymą, kad jiems pavyko rezervuoti tą pačią kėdę, tačiau teisingoje realizacijoje antras klientas turėjo gauti neigiamą atsakymą.



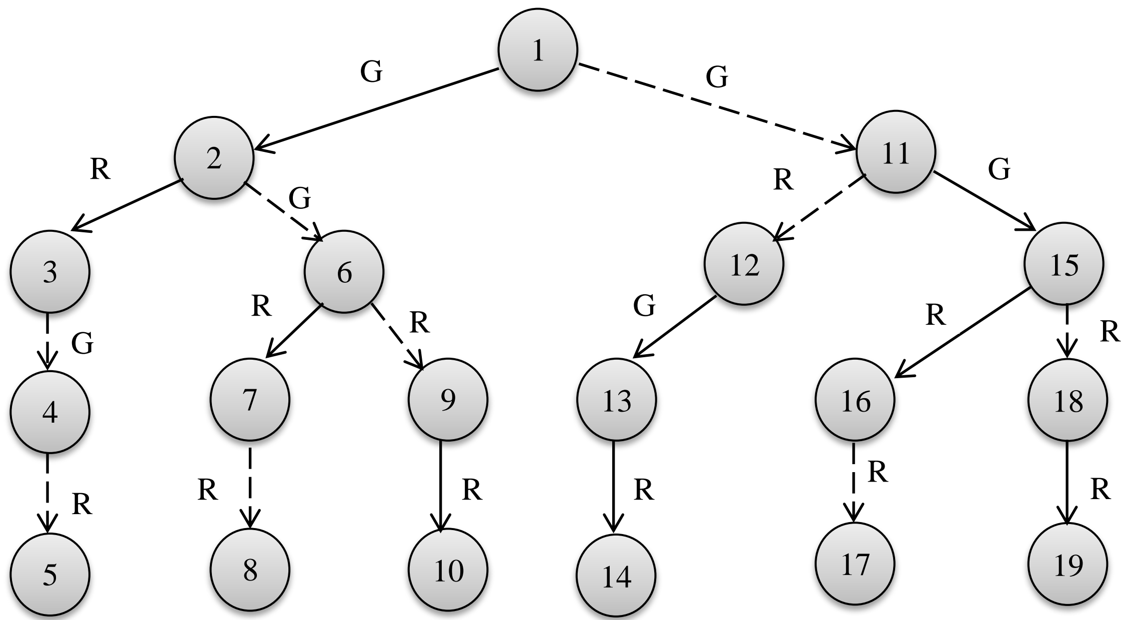
4 pav. Blogai realizuotos sistemos su dviem klientais sekų diagrama

Nagrinėti testavimo metodai nepateikia sprendimo kaip reiktų atlikti testavimą šiai klaidos kategorijai. Toliau detaliau panagrinėsime kelių klientų vykdymo kelius pagal minėtą pavyzdį bei pateiksime siūlomą metodą, kurį naudojant galima automatizuotai aptikti šią klaidą.

2.3. Vykdymo medis

Galimų šios sistemos dviejų klientų užklausų vykdymo tvarkų yra daug. Jas galima pavaizduoti medžiu pateiktu 5 pav. Šiame medyje linijomis yra pateiktos užklausos, o viršūnėse – išsišakojimo variantai. Sistema pradedama vykdyti nuo pirmosios viršūnės.

Reikia nepamiršti, kad vykdymo medis yra susijęs su pačių klientų realizacija ar norais. Jeigu abu klientai nori rezervuoti vieną ir tą pačią kėdę ir kitos jiems nėra išeitis, tuomet 1-2-3-4-5 vykdymo sekoje nebebus penktosios viršūnės. Šiuo atveju klientas nerezervuos jokios kėdės. Tačiau pavyzdyje nagrinėjame, kai kiekvienas klientas rezervuos pirmą laisvą kėdę iš gauto sąrašo



G – gauti laisvų kėdžių sąrašą — Klientas nr. 1 (ištisinė rodyklė)

R – rezervuoti kėdę — Klientas nr. 2 (brūkšninė rodyklė)

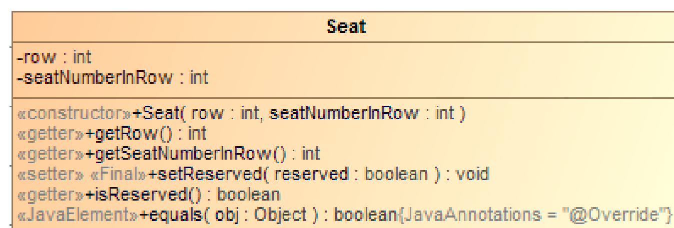
5 pav. Galimų vykdymo variantų medis

Iš 5 pav. matome, kad egzistuoja 6 vykdymo variantai, kai yra 2 klientai su 2 užklausomis. Didėjant užklausų skaičiui eksponentiškai didėja ir vykdymo variantų skaičius.

2.4. UML klasės diagramų stereotipai

Pateiktas vykdymo medžio principas 3.3 skyriuje bus naudojamas metodo vykdymo eigoje visų galimų situacijų simuliavimui. Tačiau jis tik simuliuoja klientų vykdymą – klaidų neaptinka. Reikalingas būdas kaip simuliavimo metu aptikti klaidas. Tam naudosime UML klasių diagramas.

UML turi keletą išplėtimo būdų. Vienas iš jų – stereotipai. Jie leidžia kūrėjams papildyti UML žodyną pridedant naujus modelių elementus ar savybes. Grafinis jų žymėjimas klasių diagramoje pateiktas 6 pav. Čia matome, kaip žymimas konstruktorius, kintamųjų gavimo metodai bei kiti stereotipai.



6 pav. UML klasių diagrama su stereotipais

Pasinaudodami stereotipais galime nurodyti apribojimus, į kuriuos testavimo įrankis atsižvelgtų automatinio testo metu, ir tokiu būdu nuspręstų ar vykdomas kodas veikia teisingai. Tai yra paprastas, didelių architektūros pakeitimų nereikalaujantis, ir UML 2.0 specifikaciją atitinkantis būdas. Siūlomi trys nauji stereotipai UML klasių diagramos metodams:

1) <<final>> - nurodo, kad metodas objekte turi būti iškviečiamas tik vieną kartą;
2) <<intermittent>> - nurodo, kad metodas turi būti iškviečiamas tik su kintančiais parametrais
3) <<unique>> - nurodo, kad metodas turi būti iškviečiamas tik su unikaliais parametrais;

Pasinaudodami <<final>> galime nurodyti, kad kėdė gali būti rezervuojama tik vieną kartą. Tam atlikti tereikia atitinkamam metodui diagramoje uždėti šį stereotipą. Tuomet testavimo įrankiui padavus šią UML diagramą XMI formatu jis automatiškai žinos kokių apribojimų pažeidimų ieškoti testavimo metu. Kad aptikti stereotipų pažeidimus kiekvienam minėtam stereotipui yra sugeneruojamas testo kodas, kuris, vėliau, yra įterpiamas prie atitinkamo metodo. Šių stereotipų pseudokodas pateiktas 1 lentelėje.

1 lentelė. Stereotipų pseudokodas

Stereotipas	**Pseudokodas**

<<final>>	**private static Vector<Object> *<metodoVardas>* = new Vector<Object>();**
	<metodo antraštė> {
	assertFalse(*<metodoVardas>*.contains(this));
	***<metodoVardas>*.add(this);**
	<metodo kodas>
	}

<<intermittent>>	<metodo antraštė> {
	assertTrue(!*this.equals(this.<getter>)*);
	<metodo kodas>
	}

<<unique>>	**private static Vector<Object> *<metodoVardas>Values* = new Vector<Object>();**
	<metodo antraštė> {
	assertFalse(*<metodoVardas>Values*.contains(<parametras>));
	***<metodoVardas>Values*.add(<parametras>);**
	<metodo kodas>
	}

1 lentelėje matome, kad stereotipų atitikimo tikrinimui naudojami Java *assert* metodai.

Kaip ankščiau minėjome, <<final>> nurodo, kad metodas iškviečiamas tik vieną kartą. Pavyzdinėje sistemoje, kur kėdės klasė yra kaip *JavaBean* objektas, pagal šį stereotipą yra papildomai įterpiamas masyvas. Šiame masyve yra laikomi kėdės objektai, kuriems jau buvo iškviestas rezervavimo metodas. Jeigu kėdei būtų iškviestas rezervavimo metodas antrą kartą, tuomet sąlyginis sakinys patikrinęs pamatytų, kad kėdė yra rezervuotų kėdžių sąraše, ir išmestų klaidą. Kėdės klasės modifikuota kodo dalis pagal <<final>> stereotipą galėtų būti tokia:

```
private static Vector<Object> reserved = new Vector<Object>();
public void setReserved(boolean reserved) {
    assertFalse(reserved.contains(this));
    reserved.add(this);
    if(reserved){
      DatabaseManager.getInstance().reserveSeat(this);
    }
}
```

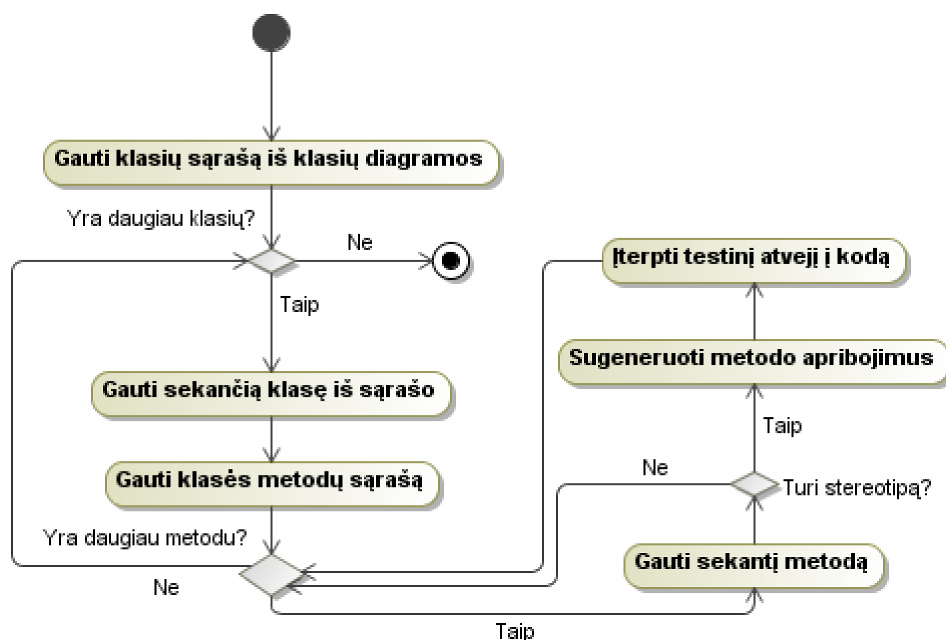
Taigi, norint automatiškai aptikti nagrinėjamą problemą reikia įterpti testinį kodą bei reikia simuliuoti kelių klientų vykdymą.

2.5. Automatinio paskirstytų sistemų testavimo metodo sąlygos, įrankio veikimo principai bei problemos

Kelių klientų simuliavimui pagal 2.3 skyriuje nagrinėtą vykdymo medį yra patogu naudoti modelių tikrintuvą. Paskirstytų sistemų testavimas modelių tikrintuvo pagalba jau buvo nagrinėtas 1.2.2 skyriuje. Pasirinkus JPF modelių tikrintuvą prisirišama prie Java programavimo kalbos, tačiau jo pagalba galėsime atlikti kelių klientų simuliavimą. Naudojant kitus įrankius metodas gali būti taikomas ir kitomis kalboms, tačiau Java yra plačiai naudojama kuriant paskirstytas sistemas.

Klaidų aptikimui naudosime stereotipus aprašytus 2.4 skyriuje. Sistemos projektavimo metu šiuos stereotipus galima nesunkiai pažymėti UML diagramoje. Stereotipai pritaikyti objektų metodams pagal *JavaBean* specifikaciją, todėl projektuojant reiktų į tai atsižvelgti.

Pagal XMI failuose pateiktus stereotipus sugeneruoti apribojimai yra įterpiami į serverio kodą. Įterpimo schema pateikta 7 pav. Joje pavaizduota pagrindinė logika, kuri kiekvienam stereotipą turinčiam metodui įterpia po apribojimą.



7 pav. Testinių atvejų generavimo ir įterpimo veiklos diagrama

Atlikus apribojimų įterpimą serveris yra sukompiliuojamas ir paleidžiamas. Serverio kompiliavimas ir paleidimas nėra testavimo įrankio funkcijos, todėl jos čia nebus nagrinėjamos.

Klientas turi būti paruošiamas automatiniam vykdymui. JPF virtuali mašina simuliuoja du šiuos klientus visais įmanomais keliais bandant aptikti klaidas. Čia yra susiduriama su problema – įvykdžius du klientus vienu galimu keliu testavimas turi būti atliekamas kitu keliu, o serveris turi sugrįžti į pradinę būseną. Testavimo įrankis turi tai padaryti automatiškai. Serverio sugražinimo į pradinę būseną problema gali būti sprendžiama šiais būdais:

1) Serverio programą automatiškai perkrauti. Tačiau tai yra daug laiko užtrunkantis procesas, nėra aišku kada serveris baigė persikrovimą, ir, kai kuriais atvejais, sunkiai įgyvendinamas.
2) Serveris turi turėti perkrovimo adresą, kurį iškvietus kintamieji gražinami į pradinę būseną. Šis problemos sprendimo būdas pasunkins žmogaus reikalaujamo atlikti darbo kiekį – jam reiks rankomis rašyti išvalymo metodus.
3) Reikia naudoti procesų atskaitos taškus valdančią aplinką. Šitokių aplinkų naudojimas kartu su modelių tikrintuvu buvo nagrinėtas 1.2.3 skyriuje pagal Leungwattanakit pateikiamą sprendimą [9].

Reiktų pastebėti, kad kitaip negu darbuose nagrinėtuose 1 skyriuje, šis metodas modelių tikrintuve simuliuoja klientus, o serveris veikia už modelio tikrintuvo ribų. Dėl šios priežasties aptinkamos klaidos serveryje yra teikiamos į serverio aplinką. Pats testavimo įrankis gali atspausdinti tik klaidas atsirandančias simuliuojamame kliente. Metode nėra numatytas būdas kaip klaidas iš serverio perduoti į testavimo įrankį, tačiau tai galima būtų atlikti keliais būdais:

1) Vietoje Java kalbos *assert* metodų naudoti klaidų spausdinimą į konkretų failą, prieinamą testavimo įrankiui. Šis įrankis galėtų stebėti šio failo pokyčius
2) Vietoje Java kalbos *assert* metodų naudoti tinklo komunikacijos kreipinius. Čia testavimo įrankis galėtų atlikti papildomą serverio vaidmenį klaidoms gauti.
3) Konfigūruoti serverį klaidų failo saugojimui, o testavimo įrankį šio failo stebėjimui.

Sekančiame skyriuje pateikiamas įrankio projektas. Jis parengtas pagal šiame skyriuje nurodomus kai kuriuos įrankio veikimo principus bei aprašytą metodą.

3. AUTOMATINIO PASKIRSTYTŲ SISTEMŲ TESTAVIMO ĮRANKIO PROJEKTAS

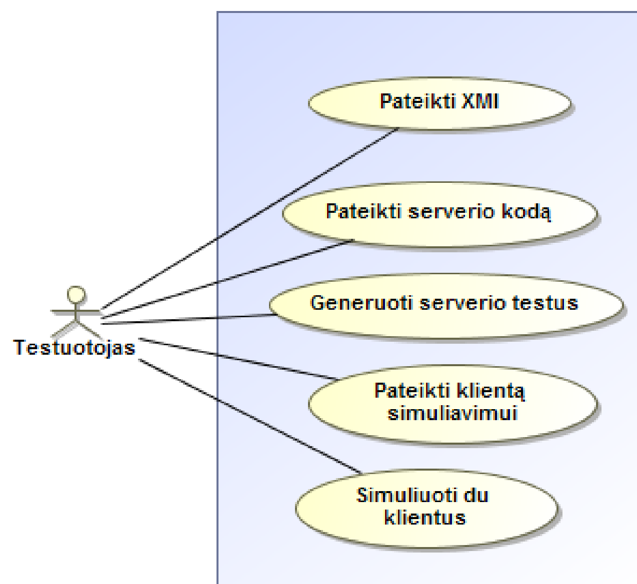
3.1. Įvadas

Projektavimas yra vienas sudėtingiausių sistemos kūrimo etapų. Nuo projektavimo kokybės priklauso tai, kaip lengva bus sistemą realizuoti, kaip sudėtinga bus sistemą pakeisti pasikeitus kliento reikalavimams bei kaip sudėtinga bus sistemą prižiūrėti.

Šiame skyriuje pateikiamas automatinio paskirstytų sistemų testavimo naudojančio UML modelius įrankio projektas. Apžvelgiami funkciniai bei nefunkciniai reikalavimai, architektūros sprendimai. Skyrius užbaigiamas pastebėjimais iš techninės pusės apie įrankį bei siūlymais kaip galima jį tobulinti.

3.2. Funkciniai reikalavimai

Kuriama sistema turi atlikti testų generavimą bei klientų simuliavimą. Testų generavimas atliekamas gavus XMI failą su stereotipų aprašais bei serverio kodą, kuriam šie testai įterpiami. Klientų simuliavimui reikalingas vykdomasis kliento failas, kurį turi pateikti vartotojas. Atlikus visus šiuos veiksmus galima paleisti automatinį testavimą, kurio metu, JPF pagalba, simuliuojami dviejų klientų kreipiniai į serverį visais galimais vykdymo variantais. Išskirti vartotojo panaudos atvejai pateikti UML panaudos atvejų diagramoje 8 pav.



8 pav. Sistemos vartotojo panaudos atvejų diagrama

Panaudos atvejai, pateikti 8 pav., yra toliau detalizuoti lentelėse kitame puslapyje

- Pateikti XMI - 2 lentelė
- Pateikti kliento ir serverio kodą - 3 lentelė
- Generuoti serverio testus - 4 lentelė
- Pateikti klientą simuliavimui - 5 lentelė
- Simuliuoti du klientus - 6 lentelė

2 lentelė. XMI pateikimo panaudos atvejis

Eil. nr.	1
Panaudos atvejis	Pateikti XMI
Vartotojas/Aktorius	Testuotojas
Aprašas	Testavimui yra reikalingas serverio klasių aprašas su stereotipais. Jį sistemai pateikia vartotojas.
Prieš sąlyga	Nėra
Sužadinimo sąlyga	Mygtuko, skirto XMI failo pasirinkimui, paspaudimas
Po-sąlyga	Gautas XMI aprašas

3 lentelė. Kodo pateikimo panaudos atvejis

Eil. nr.	2
Panaudos atvejis	Pateikti serverio kodą
Vartotojas/Aktorius	Testuotojas
Aprašas	Testavimui yra reikalingi serverio išeities tekstai. Juos sistemai pateikia vartotojas.
Prieš sąlyga	Nėra
Sužadinimo sąlyga	Mygtuko, skirto išeities tektų failų vietos pasirinkimui, paspaudimas
Po-sąlyga	Gautas serverio išeities kodas

4 lentelė. Serverio testų generavimo panaudos atvejis

Eil. nr.	3
Panaudos atvejis	Generuoti serverio testus
Vartotojas/Aktorius	Testuotojas
Aprašas	Į serverio išeities kodą yra įterpiamas testavimo kodas. Kad tai atlikti naudojamas serverio XMI klasių aprašas su numatytais stereotipais.
Prieš sąlyga	Pateiktas taisyklingas XMI serverio klasių aprašas su numatytais stereotipais bei jį atitinkantis serverio kodas.
Sužadinimo sąlyga	Mygtuko, skirto testų generavimui, paspaudimas
Po-sąlyga	Sugeneruoti testai serveriui bei įterpti į serverio kodą.

5 lentelė. Kliento simuliavimui pateikimo panaudos atvejis

Eil. nr.	4
Panaudos atvejis	Pateikti klientą simuliavimui
Vartotojas/Aktorius	Testuotojas
Aprašas	Klientų simuliavimui reikalingas vykdomasis failas.
Prieš sąlyga	Nėra
Sužadinimo sąlyga	Mygtuko paspaudimas
Po-sąlyga	Gautas vykdomasis kliento failas.

6 lentelė. Klientų simuliavimo panaudos atvejis

Eil. nr.	5
Panaudos atvejis	Simuliuoti du klientus
Vartotojas/Aktorius	Testuotojas
Aprašas	Simuliuojamos klientų užklausos į serverį analizuojant visus galimus kreipimosi variantus.
Prieš sąlyga	Paleistas serveris, gautas vykdomasis kliento failas
Sužadinimo sąlyga	Mygtuko paspaudimas
Po-sąlyga	Įvykdytas testavimas pasinaudojant JPF.

3.3. Nefunkciniai reikalavimai

Reikalavimai sistemos išvaizdai

Bendri reikalavimai vartotojo sąsajai:

- Turi būti paprastai naudojama ir greit suprantama;
- Lengvai skaitoma, sąveikaujanti sąsaja;
- Neįkyri sąsaja.

Reikalavimai panaudojamumui

- Vartotojui turėtų būti lengva naudotis sistema;
- Turi būti naudojama bendra formų struktūra;
- Turi būti naudojama anglų kalba.

Reikalavimai vykdymo charakteristikoms

Sistema turi efektyviai išnaudoti jai prieinamus resursus ir skaičiavimai turi būti įvykdomi per vartotojui priimtiną laiką.

Reikalavimai sistemos priežiūrai

Sistema yra ne kritinė, tačiau klaidų taisymas sistemoje turėtų būti vykdomas pakankamai greitai. Sistemos priežiūra turėtų būti paprasta.

Reikalavimai saugumui

Sistema dirbdama su serverio išeities tekstu bei klientu, turėtų dirbti kaip su svarbiais duomenimis. Šie duomenys neturėtų dėl sistemos klaidos patekti nenumatytiems asmenims. Duomenys neturi būti sugadinti dėl sistemos klaidos. Vartotojui turi būti aiškiai pranešami atliekami veiksmai, kad duomenų nesugadintų vartotojas, padaręs klaidą.

Reikalavimai standartams

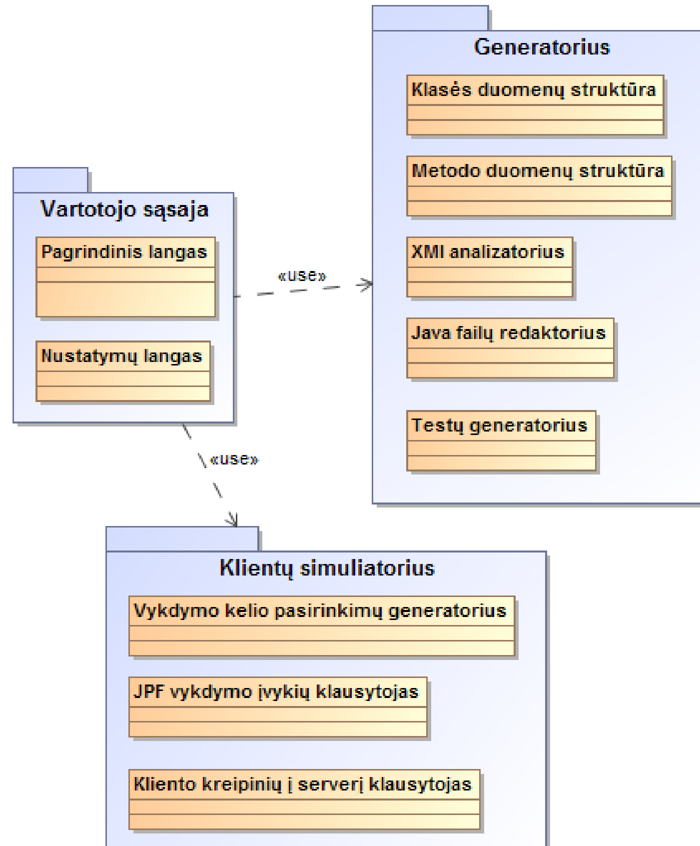
Sistema turi atitikti MDA standartus (UML, XMI)

Kultūriniai bei politiniai reikalavimai

Turi būti naudojama taisyklinga anglų kalba. Negalima naudoti įžeidžiančių frazių ar paveiksliukų.

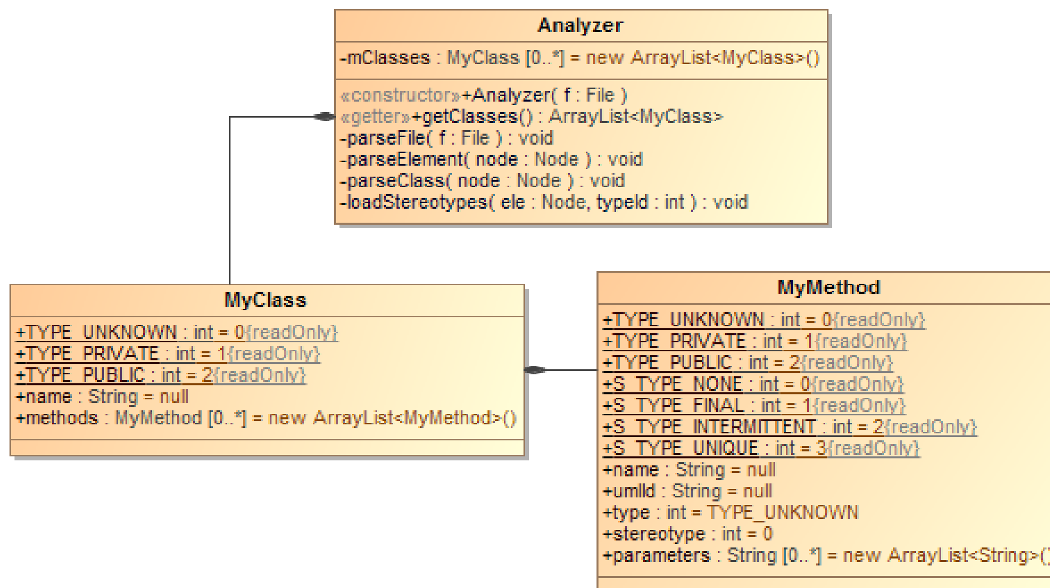
3.4. Sistemos architektūra

Įrankio architektūrą galima išskaidyti į kelias pagrindines dalis: vartotojo grafinę sąsają, testų generatorių bei įterpėją ir klientų simuliatorių naudojantį JPF sąsają. Šios pagrindinės dalys su atitinkamomis funkcijomis bendrai pavaizduotos 9 pav. panaudojant klasių diagramą. Toliau detaliau panagrinėsime konkrečias generatoriaus bei klientų simuliatoriaus klases.



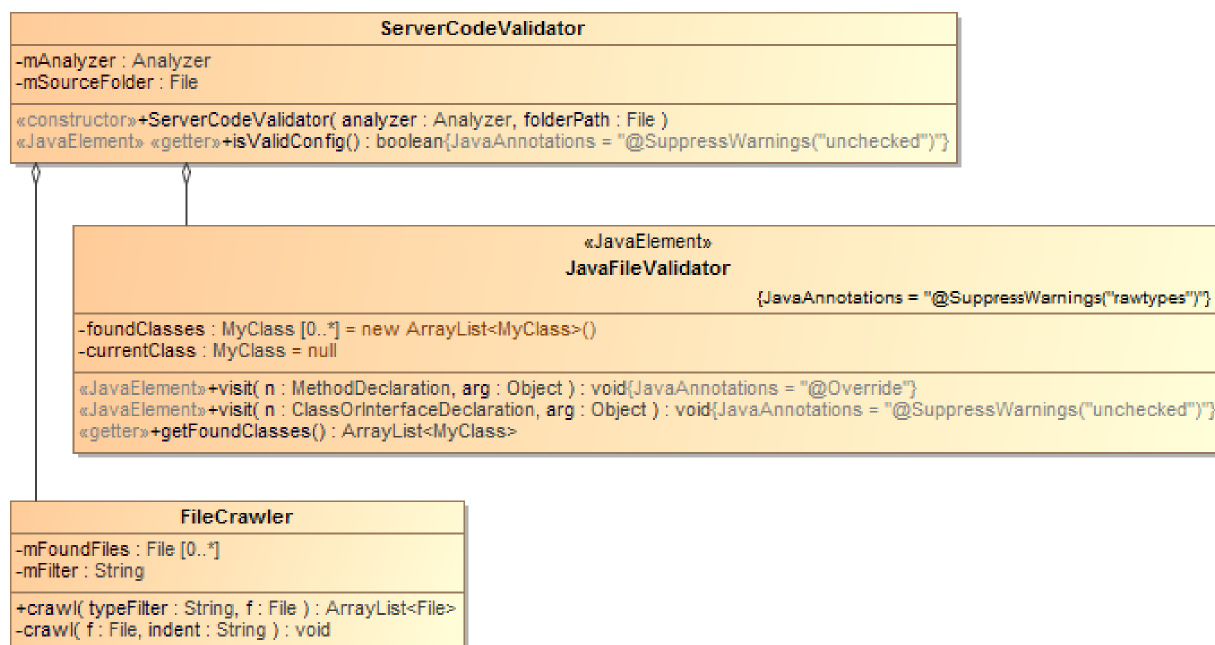
9 pav. Klasių diagrama vaizduojanti bendrą sistemos paketų bei funkcijų pasiskirstymą

Nagrinėsime klases atitinkamai pagal vartotojo darbo eiliškumą. Vartotojo pateikiamas XMI failas yra analizuojamas *Analyzer* klasės pateiktos 10 pav. Ši klasė, pagal XMI failą, sukuria metodų bei klasių struktūrą, kuri toliau naudojama testų generavimo metu.

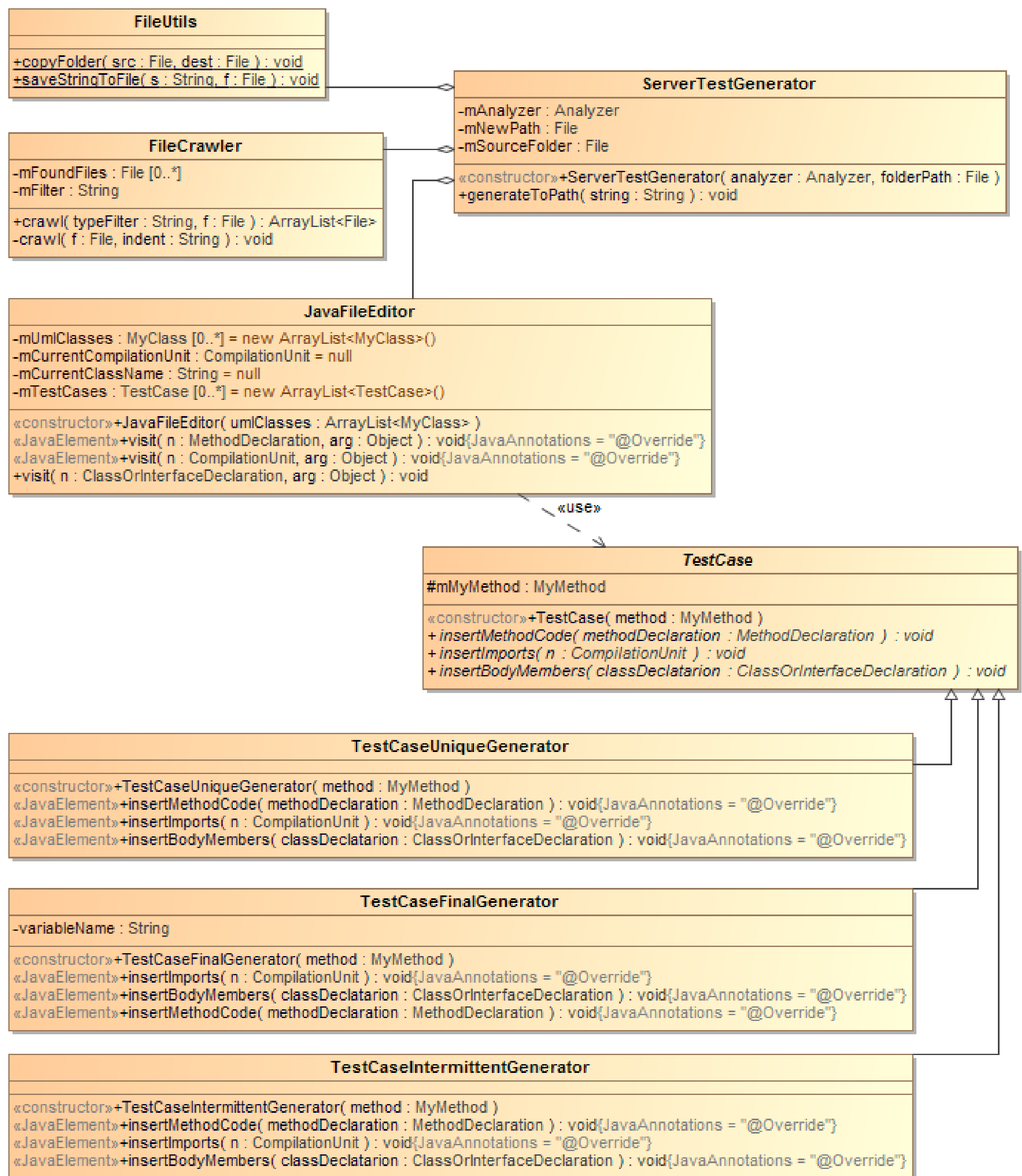


10 pav. XMI failą analizuojančių klasių diagrama

Vartotojui pateikus ir XMI failą ir serverio kodą yra patikrinamas jų atitikimas – ieškoma, kad raktiniai metodai su stereotipais egzistuotų atitinkamose klasėse. Šiam tikslui yra perskaitomi Java failai ir tikrinamas jų atitikimas. Klasės atliekančios šią veiklą pateiktos 11 pav.



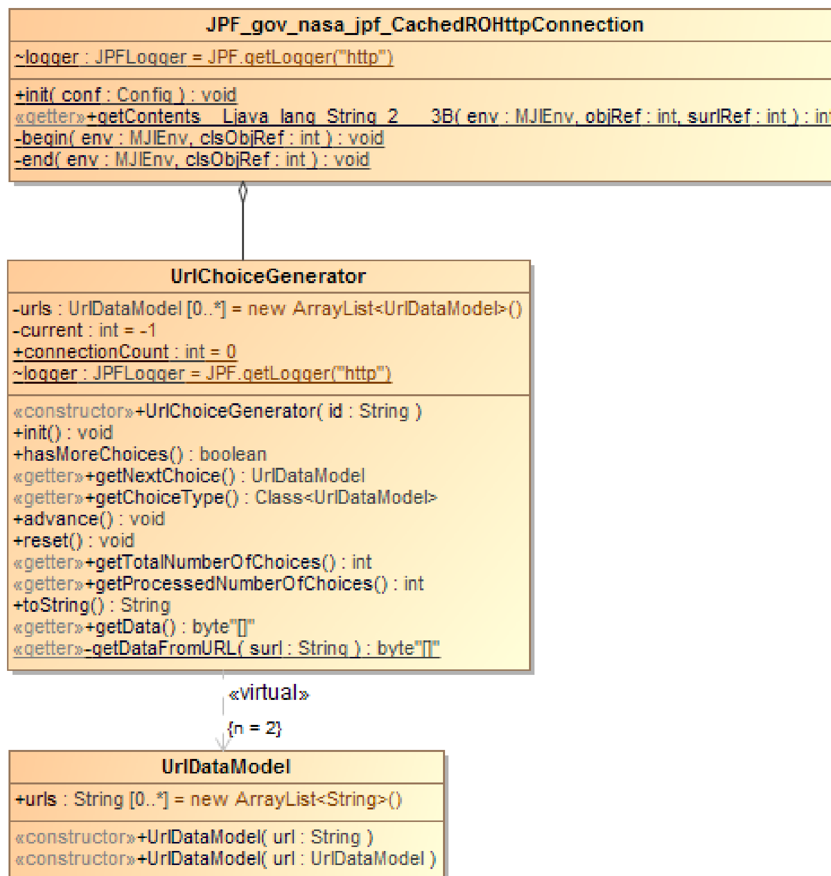
11 pav. Serverio kodo ir XMI failo atitikimą tikrinančios klasės



12 pav. Testus generuojančių bei įterpiančių klasių diagrama

12 pav. pateiktos klasės atsakingos už testų generavimą bei įterpimą į išeities kodą. *TestCase* klasė yra abstrakti. Ją paveldinčios klasės yra atsakingos už atitinkamo testo generavimą pagal stereotipą. Pavyzdžiui *TestCaseFinalGenerator* generuoja testo kodą pagal <<final>> stereotipą, kuris buvo aprašytas 2.4 skyriuje. *ServerTestGenerator* klasė, pasinaudodama *FileUtils* klase, kopijuoja serverio kodą į reikiamą direktoriją. Tuomet pasinaudodama *FileCrawler* klase eina per visus Java failus, ir atitinkamai pagal *Analyzer* klasės teikiamus duomenis, įterpia testą pasinaudojant *JavaFileEditor* klase.

13 pav. pateiktos pagrindinės klasės atliekančios simuliuojamo kliento tinklo užklausų aptikimą bei naujų vykdymo kelių sukūrimą. Kliento vykdymo metu JPF leidžia įsikišti į užklausas vykdomas per HTTP sąsają. Šioje vietoje sužinome, kad reikės atlikti pakartotinus kreipinius ir tam sukuriamas pasirinkimų generatorius.



13 pav. Tinklo užklausų klausymą bei naujų šakų kūrimą atliekančios klasės

Būsenų suvaldymui bei klientų simuliavimui yra paveldėtos JPF sisteminės klasės, duodančios pranešimus apie būsenų pasikeitimus. Šių klasių diagrama yra pateikta 14 pav. Naudojant tinklo kreipinių pagavimo klases galime išsaugoti kliento kreipimosi būsenas. O pasinaudodami perrašytu medžio paieškos bei JPF būsenų kitimų metodais valdome kliento būsenas, taip simuliuodami kelių klientų kreipimąsi į serverį.

DFSearch klasė valdo ėjimą per paieškos medį. *MyScheduler* klasė leidžia žinoti medžio išsišakojimų pagal atitinkamus įvykius, sukūrimo momentus ir duoda perrašyti koks išsišakojimų generatorius yra sukuriamas. *NamcListener* praneša apie perėjimus tarp būsenų bei kitus sisteminius įvykius. Šioje klasėje sisteminių būsenų pasikeitimo metu yra papildomai suvaldomos kliento būsenos atlikti simuliavimui.

DFSearch

«constructor»+DFSearch(config : Config, vm : JVM)
+requestBacktrack() : boolean
+search() : void
+supportsBacktrack() : boolean

MyScheduler

«constructor»+MyScheduler(config : Config, vm : JVM, ss : SystemState)
«JavaElement»+createBeginAtomicCG(arg0 : ThreadInfo) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createEndAtomicCG(arg0 : ThreadInfo) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createInterruptCG(arg0 : ThreadInfo) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createMonitorEnterCG(arg0 : ElementInfo, arg1 : ThreadInfo) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createMonitorExitCG(arg0 : ElementInfo, arg1 : ThreadInfo) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createNotifyAllCG(arg0 : ElementInfo, arg1 : ThreadInfo) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createNotifyCG(arg0 : ElementInfo, arg1 : ThreadInfo) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createParkCG(arg0 : ElementInfo, arg1 : ThreadInfo, arg2 : boolean, arg3 : long) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createSharedArrayAccessCG(arg0 : ElementInfo, arg1 : ThreadInfo) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createSharedFieldAccessCG(arg0 : ElementInfo, arg1 : ThreadInfo) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createSyncMethodEnterCG(arg0 : ElementInfo, arg1 : ThreadInfo) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createSyncMethodExitCG(arg0 : ElementInfo, arg1 : ThreadInfo) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createThreadResumeCG() : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createThreadSleepCG(arg0 : ThreadInfo, arg1 : long, arg2 : int) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createThreadStartCG(arg0 : ThreadInfo) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createThreadStopCG() : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createThreadSuspendCG() : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createThreadTerminateCG(arg0 : ThreadInfo) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createThreadYieldCG(arg0 : ThreadInfo) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createUnparkCG(arg0 : ThreadInfo) : ChoiceGenerator{JavaAnnotations = "@Override"}
«JavaElement»+createWaitCG(arg0 : ElementInfo, arg1 : ThreadInfo, arg2 : long) : ChoiceGenerator{JavaAnnotations = "@Override"}

NamcListener

«constructor»+NamcListener()
«JavaElement»+instructionExecuted(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+executeInstruction(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+threadStarted(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+threadWaiting(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+threadNotified(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+threadInterrupted(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+threadScheduled(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+threadBlocked(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+threadTerminated(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+classLoaded(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+objectCreated(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+objectReleased(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+objectLocked(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+objectUnlocked(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+objectWait(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+objectNotify(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+objectNotifyAll(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+gcBegin(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+gcEnd(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+exceptionThrown(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+exceptionBailout(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+exceptionHandled(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+choiceGeneratorRegistered(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+choiceGeneratorSet(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+choiceGeneratorAdvanced(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+choiceGeneratorProcessed(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+methodEntered(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+methodExited(vm : JVM) : void{JavaAnnotations = "@Override"}
«JavaElement»+stateAdvanced(search : Search) : void{JavaAnnotations = "@Override"}
«JavaElement»+stateProcessed(search : Search) : void{JavaAnnotations = "@Override"}
«JavaElement»+stateBacktracked(search : Search) : void{JavaAnnotations = "@Override"}
«JavaElement»+statePurged(search : Search) : void{JavaAnnotations = "@Override"}
«JavaElement»+stateStored(search : Search) : void{JavaAnnotations = "@Override"}
«JavaElement»+stateRestored(search : Search) : void{JavaAnnotations = "@Override"}
«JavaElement»+propertyViolated(search : Search) : void{JavaAnnotations = "@Override"}
«JavaElement»+searchStarted(search : Search) : void{JavaAnnotations = "@Override"}
«JavaElement»+searchConstraintHit(search : Search) : void{JavaAnnotations = "@Override"}
«JavaElement»+searchFinished(search : Search) : void{JavaAnnotations = "@Override"}
«JavaElement»+publishStart(publisher : Publisher) : void{JavaAnnotations = "@Override"}
«JavaElement»+publishTransition(publisher : Publisher) : void{JavaAnnotations = "@Override"}
«JavaElement»+publishPropertyViolation(publisher : Publisher) : void{JavaAnnotations = "@Override"}
«JavaElement»+publishConstraintHit(publisher : Publisher) : void{JavaAnnotations = "@Override"}
«JavaElement»+publishFinished(publisher : Publisher) : void{JavaAnnotations = "@Override"}

14 pav. Naudojamos JPF sisteminės klasės būsenų valdymui

3.5. Sistemos naudojimas

Naudojant sukurtą automatinio paskirstytų sistemų testavimo įrankį vartotojas turi atlikti keletą veiksmų. Veiksmų eiliškumo atskyrimui bei kas kokius veiksmus turi atlikti 15 pav. pateikiama vartotojo darbo veiklos diagrama. Iš šios diagramos matome, kad po sėkmingo testų generavimo ir įterpimo į serverio kodą vartotojas turi pats sukompiliuoti serverį ir jį paleisti. Tuomet galima atlikti klientų simuliavimą.

15 pav. Vartotojo darbo veiklos diagrama

3.6. Sistemos kokybės užtikrinimas

Kokybės užtikrinimo tikslas yra įsitikinti, kad sistema veikia korektiškai, t.y. sistemos veikimas atitinka jai keliamus funkcinius ir nefunkcinius reikalavimus. Kad tai atlikti programuotojas ir testuotojai privalo naudoti kuo daugiau priemonių sistemos kokybės užtikrinimui. Norint užtikrinti automatinio paskirstytų sistemų naudojančio UML modelius įrankio kokybę proceso metu atliekami žemiau pateikti veiksmai:

- statinė kodo analizė atliekama programavimo stadijoje;
- komponentų (vienetų) testavimas atliekamas testavimo stadijoje;
- sistemos integracijos testavimas atliekamas testavimo stadijoje.

Statinės kodo analizės metu aptinkamos potencialios sistemos klaidos jau programavimo fazės metu. Paprastai statinė kodo analizė aptinka potencialias nenumatytų atvejų klaidas, amžinų ciklų klaidas ir panašiai. Tokio tipo klaidos gali ir būti neaptiktos normalaus sistemos vartojimo atveju, tačiau tokių klaidų atsiradimas dažnai formuoja sistemos funkcionalumo pabaigą, o tai gali iššaukti vykdomo darbo nutraukimą be galimybės sugrąžinti į prieš tai vykdytą žingsnį.

Programuojant komponentus siekiama išanalizuoti kiekvieno komponento metodo silpnąsias savybes, kraštutinumus ir pan. T.y. analizuojama kiekvieno komponento struktūra. Toks testavimas literatūroje yra vadinamas „baltos dėžės“ testavimu. „Baltos dėžės“ testavimas apibrėžiamas: „Testiniai atvejai gaunami iš programos struktūros. Žinios apie programą naudojamos nustatyti papildomus testinius atvejus. Tikslas yra išbandyti visus programos operatorius ir galimus skaičiavimo kelius.“ [12].

Sistemos integracijos testavimas atliekamas integruojant atskirus komponentus į sistemą. Čia jau testuojami ne atskiri komponentai, o jų sąveika tarpusavyje ir bendras visos sistemos ar jos dalies veikimas. Sistemos integracijos testavimas taip pat atliekamas „baltos dėžės“ principu.

3.7. Sistemos ateities tobulinimo darbai

Įrankio kūrimo metu pastebėta kaip šį įrankį galima būtų patobulinti ateityje.

- **Komandinė eilutė.** Įrankio vykdymo pilnam automatizavimui reiktų sukurti jam sąsają per komandinę eilutę;
- **Serverio kompiliavimas bei paleidimas.** Vartotojui būtų patogu galėti kompiliuoti bei perkrauti serverį per šį įrankį. Tam galima panaudoti vartotojo scenarijų vykdymą;
- **Aptiktų klaidų surinkimas į vieną vietą, ataskaitos generavimas.** Reikalingas aiškus atsakymas iš įrankio ar testai praėjo be klaidų.

Testavimo įrankį, kuris gali automatiškai vykdyti visą testavimo eigą, galima naudoti tęstinos integracijos sistemose. Šie siūlomi patobulinimai leistų testavimą vykdyti tokiose sistemose.

4. EKSPERIMENTAI SU SUKURTU TESTAVIMO ĮRANKIU

4.1. Įvadas

Šiame skyriuje pateikti eksperimentiniai bandymai, kurie buvo atlikti norint nustatyti sukurto įrankio ir metodo efektyvumą. Apžvelgus kai kurias eksperimento metu naudojamas sąvokas bei testuojamų sistemų funkcionalumą, pateikiami eksperimento rezultatai su išvadomis.

Eksperimentas atliktas panaudojant mutacinį testavimą. Naudojamos dvi testinės sistemos, kurioms buvo sugeneruoti mutantai. Ištirta kiek mutantų įrankis aptiko bei atkreiptas dėmesys į neaptiktus mutantus bei neaptikimo priežastis. Taip pat palyginimui patikrinta kiek mutantų aptinka *JUnit* testiniai atvejai.

4.2. Eksperimento aprašyme naudojamos sąvokos

Programa mutantas. Programa mutantas yra tam tikros originalios programos kopija turinti vieną pasikeitimą programos išeities tekste [11]. Kitaip tariant tai programa, kuri skiriasi nuo savo originalios programos bent viena klaida (vienas pakeitimas programos išeities tekste turi įtakoti bent vieną klaidą). Paprastai tai būna kiek įmanoma smulkesnis pasikeitimas. Programų mutantų originali programa turi apribojimą, kad visos išeities kodo eilutės turi būti reikalingos ir neturėti perteklinio, niekur nenaudojamo kodo, ar kodo, nuo kurio pasikeitimo nepriklauso programos elgsena. Pakeitimu paprastai yra laikomas bet kokio kintamojo priskyrimo pakeitimas, lyginimo operatoriaus pakeitimas kitu ar pan.

Mutacinis testavimas. Programinės įrangos testavimo srityje egzistuoja testavimo metodas, vadinamas mutaciniu testavimu [15]. Mutacinio testavimo tikslas yra patikrinti testavimo metodo kokybę. Mutacinio testavimo esmė yra sukurti aibę programų mutantų ir juos naudoti kaip testuojamas sistemas.

Programų mutantų generavimo operatorius. Tai atributas naudojamas programų mutantų generavimo procese. Programų mutantų generavimo operatorius susideda iš išeities teksto dalies, kurią tikimasi aptikti analizuojant originalios programos išeities tekstą ir išeities teksto dalies, kurią reikia naudoti vietoj ieškomos išeities teksto dalies ją aptikus analizuojant originalios programos išeities tekstą generuojamoje programoje mutante.

4.3. Testinės sistemos programų mutantų generavimui

Eksperimentinio tyrimo metu buvo sukurtos dvi pavyzdinės sistemos. Abi sistemos buvo realizuotos Java programavimo kalba. Sistemos sudarytos iš kliento ir serverio. Serverio dalis paleidžiama pasinaudojant *Apache Tomcat*.

Pirmoji sistema, atliekanti kėdžių rezervavimo funkciją, jau buvo aprašyta nagrinėjant problemą 2.2 skyriuje. Ja pasinaudojęs klientas gali gauti laisvų kėdžių sąrašą bei rezervuoti laisvą kėdę. Šios sistemos serverio dalį sudaro 219 kodo eilučių ir 5 klasės – tai iš ties paprasta ir maža sistema.

Antroji bandymui naudojama sistema simuliuoja banko ir bankomato komunikaciją. Ji turi kiek daugiau funkcijų nei pirmoji – pinigų nuėmimas nuo sąskaitos, PIN kodo pakeitimas, autentifikacija pasinaudojant PIN kodu ir sąskaitos likučio gavimas. Šios sistemos serverio dalį sudaro 318 kodo eilučių ir 6 klasės.

Šių sistemų kliento dalies daug nenagrinėsime, nes jos testavimo neatliekame. Pakanka žinoti, kad klientų realizacijoje eilės tvarka vykdomos serverio užklausos, kliento programų grafinė vartotojo sąsaja nebuvo kuriama.

Sistemoms buvo sukurtos klasių diagramos, ties metodais pažymėti stereotipai parodantys testavimo įrankiui apribojimus. Kėdžių rezervavimo sistemoje panaudotas <<final>> stereotipas, o banko ir bankomato sistemoje panaudoti <<intermittent>> ir <<unique>> stereotipai. Šių stereotipų reikšmės bei pagal juos kuriami testiniai atvejai buvo nagrinėti 2.4 skyriuje. Įsitikinta, kad abi sistemos įvykdo testus su automatiniu paskirstytų sistemų testavimo įrankiu be klaidų.

Taip pat buvo sukurti *JUnit* testai abiejų sistemų serverio komponentams. Šie *JUnit* testai patikrina serverio komponentų veikimą atliekant vienetų testavimą. Jais pasinaudojant dar kartą įsitikinta, kad sistemos veikia taip, kaip yra tikimasi. *JUnit* testai toliau lygiagrečiai naudoti lygiagrečiai kartu su sukurtu testavimo įrankiu, taip žiūrint į įrankio bei *JUnit* testavimo funkcionalumą mutacinio testavimo metu.

4.4. Programų mutantų generavimas

Siekiant ištirti sukurto automatinio paskirstytų sistemų naudojančio UML modelius įrankio bei metodo testavimo kokybę yra naudojamas mutacinis testavimas. Programų mutantų generavimas buvo atliktas automatinio programų mutantų generatoriaus pagalba. Programų mutantų generavimo operatorių sąvoka aprašyta 4.2 skyriuje.

Generavimo metu buvo naudojami šie mutantų generavimo operatoriai:

- aritmetinių operacijų pakeitimai;
- loginių operacijų pakeitimai;
- priskyrimo operacijų ištrynimas.

Operatoriai naudoti programų mutantų generavimui pateikti 7 lentelėje.

7 lentelė. Naudojami operatoriai programų mutantų generavimui

<table>
<tr><th>Išeities teksto dalis</th><th>Pakeitimas</th></tr>
<tr><td>„=“</td><td>Kodo eilutės ištrynimas</td></tr>
<tr><td>“!=”</td><td>“==”</td></tr>
<tr><td>“==”</td><td>“!=”</td></tr>
<tr><td>“<=”</td><td>“>=”</td></tr>
<tr><td>“>=”</td><td>“<=”</td></tr>
<tr><td>“>”</td><td>“<“</td></tr>
<tr><td>“<“</td><td>“>”</td></tr>
<tr><td>“++”</td><td>“--”</td></tr>
<tr><td>“--”</td><td>“++”</td></tr>
<tr><td>“||”</td><td>“&&”</td></tr>
<tr><td>“&&”</td><td>“||”</td></tr>
<tr><td>“-1”</td><td>“-2”</td></tr>
<tr><td>“0”</td><td>“1”</td></tr>
</table>

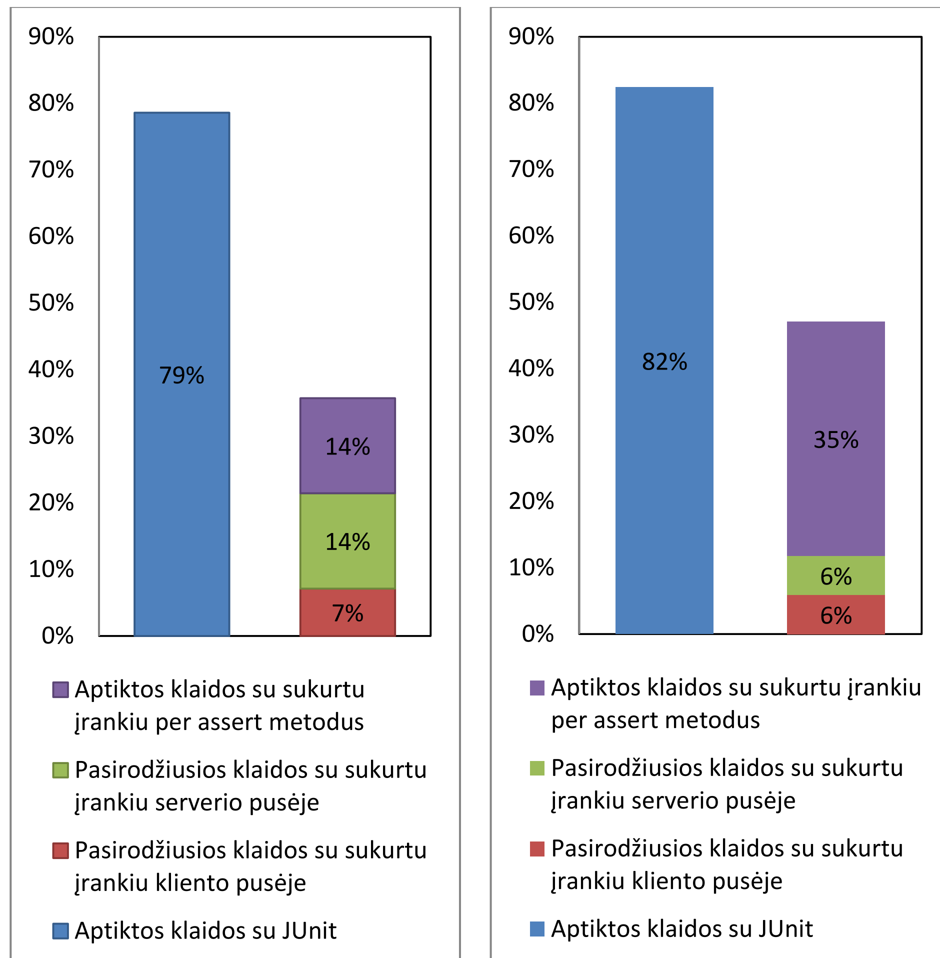
Paruošus testinių sistemų klasių diagramas su stereotipais bei sugeneravus aibę mutantų iš testinių programų buvo pradėtas vykdyti eksperimentas. Eksperimento metu naudotų sistemų parametrai bei mutantų skaičiai pateikti 8 lentelėje.

8 lentelė. Testinių sistemų parametrai

Testinė sistema	***Kėdžių rezervavimo sistema***	***Banko-bankomato sistema***
Kodo eilučių skaičius	219	318
Klasių skaičius	5	6
Klasės padengiamos testų	2	3
Klasių padengiamų testais kodo eilučių skaičius	98	149
Sugeneruotų mutantų skaičius	14	17
Pritaikyti stereotipai	<<final>> vieną kartą	<<intermittent>> du kartus; <<unique>> vieną kartą

4.5. Rezultatai

Sugeneruoti mutantai buvo testuojami naudojant sukurtą automatinį paskirstytų sistemų testavimo įrankį. Taip pat buvo atlikti *JUnit* testai kiekvienam sugeneruotam mutantui. Šio eksperimento rezultatai pateikti 16 pav.



16 pav. Kėdžių rezervavimo sistemos eksperimento rezultatai kairėje, banko–bankomato sistemos dešinėje.

JUnit testai aptiko apie 80% mutantų abiejose sistemose. Žinoma, įdėjus šiek tiek daugiau darbo į šiuos testus bei apgalvojus visas situacijas galima pasiekti ir geresnį mutantų aptikimą. Sukurtas automatinis paskirstytų sistemų testavimo įrankis, lyginant su *JUnit* rezultatais, gerokai nusileidžia pasiekdamas 35–47% mutantų aptikimą. Panašių eksperimento rezultatų ir buvo tikimąsi sulaukti.

Analizuojant eksperimento rezultatus įrankio vykdymo metu aptiktos klaidos buvo išskaidytos pagal jų pasirodymo vietą. Kai kurių mutantų testavimo metu klaidos aptiktos ne dėl naudotų stereotipų. Klaidos, kaip *NullPointerException* bei *NegativeArraySizeException*, atsirado vykdant patį klientą ar serverį. 16 pav. jos buvo išskirtos nuo klaidų aptiktų panaudojant stereotipus. Klaidos kliente buvo aptiktos, kai serveris grąžino nenumatytą, nelaukiamą rezultatą ir jos pasirodė pačiame testavimo įrankyje simuliuojančiame klientų vykdymą.

Taip pat buvo pastebėta, kad sukurtas įrankis dažniausiai neaptinka klaidų kuomet naudojami aritmetiniai mutantų operatoriai. Pavyzdžiui <<intermittent>> stereotipas neaptiko klaidos kai banko sąskaitoje turimų pinigų dydis padidėjo vartotojui nuėmus pinigus nuo sąskaitos. Tačiau tai yra suprantama, nes testavimo metodas orientuotas aptikti klaidas atsirandančias dėl varžymosi dėl resursų sąlygų.

4.6. Atlikto eksperimento išvados

Nepaisant mažo aptiktų mutantų skaičiaus įrankis sėkmingai aptiko mutantus, kuriuose egzistavo varžymosi dėl resursų sąlygos klaidos. Šios klaidos pavyzdys buvo aprašytas 2.2 skyriuje bei klaidos sekų diagrama buvo pateikta 4 pav.

Atliktas testavimas parodė mažą mutantų aptikimo kiekį, tačiau aptiko klaidą dėl varžymosi dėl resursų. Todėl šį įrankį reiktų naudoti kartu su kitais testavimo įrankiais norint pagerinti aptinkamų klaidų kiekį integracinio testavimo metu.

5. IŠVADOS

1. Išanalizavus literatūroje pateiktas paskirstytų sistemų testavimo problemas bei sprendimus buvo nustatyta, kad paskirstytų sistemų testavimas vis dar yra brangus ir lėtas procesas, o egzistuojančios paskirstytų sistemų testavimo automatizavimo priemonės yra retai taikomos praktikoje, todėl paskirstytų sistemų testavimo automatizavimo metodų ir priemonių kūrimas bei tyrimas šiuo metu yra aktualus.

2. Buvo pasiūlytas metodas automatiniam paskirstytų sistemų testavimui atlikti. Metodo tikslas yra automatiškai aptikti paskirstytų sistemų klaidas pasireiškiančias esant lenktynių dėl resursų sąlygoms. Šios klaidos yra sunkiai aptinkamos testuotojams rašant testinius atvejus, nes įvairių užklausų vykdymo variantų yra daug. Pasiūlytas metodas visus užklausų vykdymo variantus išbando automatiškai.

3. Metodo įgyvendinimui buvo pasiūlyti stereotipai, skirti metodų apribojimams aprašyti. Šie stereotipai gali būti naudojami su bet kuriuo UML 2.0 specifikaciją realizuojančiu įrankiu. Atlikus eksperimentinius tyrimus paaiškėjo, kad pasiūlyti stereotipai yra pakankami apribojimams aprašyti.

4. Darbo metu buvo sukurtas paskirstytų sistemų testavimo įrankis naudojantis JPF. Jis geba automatiškai generuoti ir įterpti testinius atvejus iš duotų UML klasių diagramų ir geba automatiškai vykdyti kelis klientus, simuliuodamas įvairius jų užklausų į serverį variantus.

5. Tiriant pasiūlytą metodą buvo atlikti eksperimentiniai tyrimai siekiant nustatyti surandamų klaidų kiekį bei pobūdį. Atlikus eksperimentus paaiškėjo, kad nagrinėtais atvejais aptinkama 41% mutantų. Atlikus analizę nustatyta, kad geresnių rezultatų nebuvo pasiekta dėl mažo testų padengiamumo, nes apibrėžtais stereotipais negalima aptikti klaidų atsirandančių kuomet klasių metodai gražina blogas reikšmes. Siekiant pagerinti rezultatus siūloma naudoti šį metodą kartu su kitais testavimo metodais, kurie aptinka kitokius klaidų tipus.

6. Sukurtą įrankį galima naudoti tolimesniuose eksperimentuose, metodo tobulinimo darbuose bei paskirstytų sistemų integracinio testavimo metu. Įrankio kūrimo metu pastebėta, kad į ateities sistemos tobulinimo darbus vertėtų įtraukti sąsajos per komandinę eilutę bei ataskaitų generavimo funkcijas.

6. LITERATŪRA

[1] JASAITIS, Robertas; BAREIŠA, Eduardas. Distributed system model checking design. *Information Technologies*, 2011, 27-29.

[2] JASAITIS, Robertas; BAREISA, Eduardas. Distributed System Automated Testing Design. Iš: *Information and Software Technologies*. Springer Berlin Heidelberg, 2012. p. 255-266.

[3] ARTHO, Cyrille, et al. Cache-based model checking of networked applications: From linear to branching time. Iš: *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*. IEEE, 2009. p. 447-458.

[4] ARTHO, Cyrille, et al. Efficient model checking of applications with input/output. Iš: *Computer Aided Systems Theory–EUROCAST 2007*. Springer Berlin Heidelberg, 2007. p. 515-522.

[5] ARTHO, Cyrille, et al. Efficient model checking of networked applications. Iš:*Objects, Components, Models and Patterns*. Springer Berlin Heidelberg, 2008. p. 22-40.

[6] LONG, Brad; STROOPER, Paul. A case study in testing distributed systems. Iš: *Distributed Objects and Applications, 2001. DOA'01. Proceedings. 3rd International Symposium on*. IEEE, 2001. p. 20-29.

[7] GHOSH, Sudipto; MATHUR, Aditya P. Issues in testing distributed component-based systems. Iš: *First ICSE workshop on testing distributed component-based systems*. 1999.

[8] HANSEN, Per Erinch. Reproducible testing of monitors. *Software: Practice and Experience*, 1978, 8.6: 721-729.

[9] LEUNGWATTANAKIT, Watcharin, et al. Model checking distributed systems by combining caching and process checkpointing. Iš: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011. p. 103-112.

[10] Sommerville, I., Software engineering. 7th ed. 2004, Boston: Addison-Wesley. 759.

[11] ALEXANDER, Roger T., et al. Mutation of Java objects. Iš: *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on.* IEEE, 2002. p. 341-351.

[12] British Computer Society Specialist Interest Group in Software Testing. Standard for Software Component Testing. British Computer Society, SIGIST, 2001

[13] ARTHO, Cyrille; GAROCHE, P.-L. Accurate centralization for applying model checking on networked applications. Iš: *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 2006. p. 177-188.

[14] Java PathFinder. [Žiūrėta 2013 04 11] , prieiga internete <http://javapathfinder.sourceforge.net>

[15] BYBRO, Mattias; ARNBORG, S. A mutation testing tool for java programs. *Master's thesis, Stockholm University, Stockholm, Sweden*, 2003.

[16] Help - Rational Software Architect RealTime Edition. [Žiūrėta 2013 05 06], prieiga internete <http://pic.dhe.ibm.com/infocenter/rsarthlp/v8>

[17] RICCA, Filippo, et al. The role of experience and ability in comprehension tasks supported by UML stereotypes. Iš: *International Conference on Software Engineering: Proceedings of the 29 th International Conference on Software Engineering*. 2007. p. 375-384.

[18] BERNARD, Eddy, et al. Model-based testing from UML models. Iš: *Procs. of the Int. Workshop on Model-based Testing (MBT'2006)*. 2006. p. 223-230.