KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Mantas Mitė

Automatinis testų generavimas, paremtas OCL apribojimais

Magistro darbas

Darbo vadovas:

prof. Eduardas Bareiša

Kaunas, 2011

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

PROGRAMŲ INŽINERIJOS KATEDRA

Mantas Mitė

Automatinis testų generavimas paremtas OCL apribojimais

Magistro darbas

Recenzentas:
prof. Lina Nemuraitė
2011 - 05 - 27

Vadovas:
prof. Eduardas Bareiša
2011-05-27

Atliko:
IFM – 9/2 gr. stud.
Mantas Mitė
2011-05-27

Kaunas, 2011

Automated test generation using OCL constraints

Mantas Mitė

SUMMARY

The development of modern software is a difficult process, there is a high possibility to leave uncorrected mistakes in the software, it would be more precisely to say that it is impossible to make software without bugs. Software testing is maybe the biggest part of development process. The unit testing is very powerful testing. It prevents from defects 20% and more [1]. The quality increase came at a cost of approximately 30% more development time. The main goal is create generator for unit testing. Automated test generator can reduce development time.

Unit test generation is based on the OCL (*Object Constraint Language*) and software static model. OCL eliminate a test oracle problem. Software static model can be UML (Unified Modeling Language) a class diagrams, but it very complicated approach. We use a reflection technology, because it is more precise and better today. Also OCL constrains are inserted in code.

Santrauka

Šiuolaikinės programinės įrangos kūrimas yra sudėtingas ir brangus procesas. Didžiausia kaštų dalį sudaro programinės įrangos testavimas, kuris gali viršyti net kuriamos programinės įrangos kūrimo kainą [2] [3]. Vienas iš galimų programinės įrangos kaštų mažinimo būdų yra testavimo automatizavimas.

Šio darbo tikslas buvo sukurti vienetų testų generatorių, kuris bent dalinai leistų automatizuoti programinės įrangos vienetų testavimą. Vienetų testų generavimas remiasi OCL (*angl Object Constraint Language*) apribojimais ir programiniu kodu, kaip programos statiniu modeliu. Buvo atsisakyta naudotis UML (*angl. Unified Modeling Language*) klasių diagramos modeliu. Testuotojų patogumui OCL apribojimai yra tiesiogiai įkomponuojami į programinį kodą.

Eksperimentinėje dalyje buvo įrodytas generatoriaus efektyvumas. Įrodymui buvo pasitelktas mutacinis testavimas ir iteracinis rezultatų gerinimas iki norimo tikslo. Buvo pasiektas virš 70% sugautų mutantų kiekis ir virš 90% kodo padengiamumas.

Turinys

1. ĮVADAS 9

2. PROJEKTAVIMO IR DIEGIMO ĮRANKIO ANALIZĖ 11

2.1. Programinės įrangos kokybė 11

2.2. Testavimo metodologijos 11

2.3. Testavimo tipai 12

2.4. Vienetų testai 13

2.5. Object Contsrain Language – OCL 13

2.5.1. OCL pavyzdys 14

2.6. Design by Contract – DbC 15

2.7. Orakulo šablonas 15

2.7.1. Orakulo šablonas, OCL ir vienetų testų karkasas 16

2.8. Darbo tikslai ir uždaviniai 17

2.9. Vienetų testavimo įrankiai, jų klasifikacija. 18

2.10. Panašių sprendimų apžvalga 19

2.10.1. Java Modeling Language - JML 19

2.10.1. TestGen4J 21

2.10.2. JTest 22

2.10.3. Code Contracts 22

3. AUTOMATINIO TESTŲ GENERATORIAUS PROJEKTINĖ DALIS 24

3.1. Automatinio testų generatoriaus paskirtis 24

3.2. Esminiai reikalavimai 24

3.3. Nefunkciniai reikalavimai 25

3.4. OCL kalbos įgyvendintos funkcijos 25

3.5. OCL apribojimai užrašomi programavimo kalboje 26

3.6. Architektūra, jos savybės 26

3.6.1. Sąsaja 26

3.6.2. OCL 26

3.6.3. Programinio kodo analizės 27

3.6.4. Šablonų paketas 27

3.6.1. Generatoriaus paketas 28

3.7. Atsitiktinių reikšmių generavimas 29

3.7.1. Pavyzdys apribojimų ir sugeneruoto testinio atvejo 30

3.7.2. Sistemos testavimas 31

4. AUTOMATINIO TESTO GENERATORISUS TYRIMAS 32

4.1. Įvadas 32

4.2. Kokybės analizė 32

4.2.1. Specifikacijos atitikimas 32

4.3. Iškilusios problemos 32

4.3.1. OCL susiejimas su programavimo kalba. 32

4.3.2. Objektų kūrimo problematika 32

4.4. Siūlymai tobulinti programą 33

4.4.1. Grafinis nustatymų valdymas 33

4.4.2. Suderinti su ANT 33

4.5. Išvados 33

5. AUTOMATINIŲ TESTŲ GENERATORIAUS EKSPERIMENTINIS TYRIMAS 34

5.1. Tikslas 34

5.2. Mutacinis testavimas 34

5.2.1. Mutantų sudarymo taisyklės 34

5.3. Testavime naudojamos metrikos 34

5.4. Tyrimo vykdymo scenarijus. 35

5.5. Eksperimentinių programų aprašymas. ... 36
5.5.1. Sinuso $\boldsymbol{sin\alpha}$ skaičiavimo programa. ... 36
5.5.2. Radianų į laipsnius vertimo programa. ... 37
5.5.3. Bessel funkcijos skaičiavimo programa. ... 37
5.5.4. Dešimtainio logaritmo funkcija. ... 38
5.5.5. Eksponentinė $\boldsymbol{ex}$ funkcija. ... 38
5.6. Eksperimento rezultatai ... 39
5.6.1. Rezultatai pirmos iteracijos ... 39
5.6.2. Rezultatai pokyčiai po antros iteracijos ... 41
5.7. Išvados ... 42
6. IŠVADOS ... 43
7. LITERATŪRA ... 44
8. TERMINŲ IR SANTRUMPŲ ŽODYNAS ... 47
9. PRIEDAI ... 48
9.1. Straipsnis „Programos elgsenos testavimo posistemė“. ... 48

Paveikslėlių turinys

1 pav. Aptiktų klaidų kiekis iš visų galimų klaidų programoje 9

2 pav. Kuriamos sistemos kontekstas 17

3 pav. Vienetų testų generavimo būdai 19

4 pav. Paketų diagrama 26

5 pav. OCL susiejimas su programos kodu 27

6 pav. Šablonų susiejimas su Junit ir OCL 27

7 pav. Vienetų generavimo veikimo charakteris 29

8 pav. Tyrimo – testavimo veiklos schema 36

9 pav. Sugautų mutantų kiekis 39

10 pav. Kodo padengimo tyrimo rezultatai 40

11 pav. Šakų padengimo tyrimo rezultatai 40

12 pav. Rezultatų pokyčiai 2-os iteracijos metu 41

Lentelių turinys

1 lentelė. JML kalbos trumpas aprašymas 20

2 lentelė. Code Contracts trumpas raktažodžių aprašas 22

3 lentelė. OCL kalbos suderinami elementai 25

4 lentelė Parametrų generavimo taisyklės 30

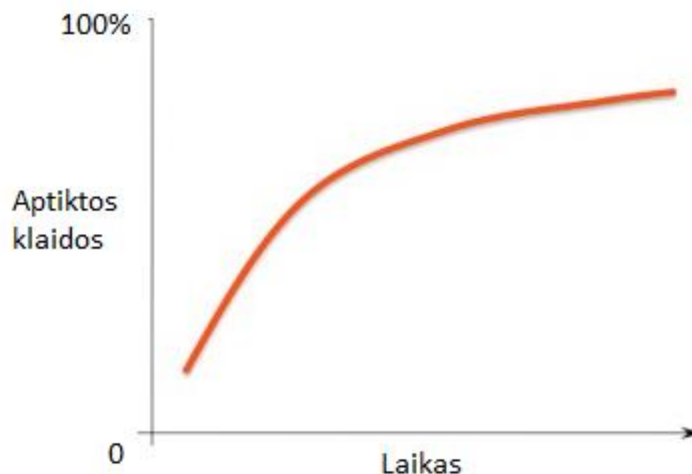
5 lentelė. Tiriamųjų programų charakteristikų apibendrinimas 38

1. ĮVADAS

Šiuolaikinės programinės įrangos kūrimas yra ganėtinai sudėtingas ir komplikuotas procesas. Programinės įrangos kūrimas susideda iš keleto esminių etapų:

- Reikalavimų surinkimo
- Projektavimo
- Plėtojimo
- Testavimo
- Palaikymo

Programinės įrangos kūrimas be testavimo šiais laikas yra beveik neįmanomas, be to yra vienas iš daugiausiai resursų reikalaujančių programinės įrangos kūrimo etapų. Programinei įrangai vis sudėtingėjant ir didėjant, darosi vis sunkiau sukurti ją be klaidų, dar tiksliau kalbant, beveik neįmanoma sukurti ją be klaidų, o surasti absoliučiai visas klaidas yra neįmanoma. Teoriškai, be klaidų programinės įrangos kūrimas yra įmanomas naudojant formalias specifikacijas, tačiau tokia programinė yra labai brangi ir kuriama tik kritinio veikimo sistemoms. Dažniausiai programinėje įrangoje stengiamasi ištaisyti tiek klaidų, kad būtų tenkinami vartotojų poreikiai, o ne visas klaidas [4]



1 pav. Aptiktų klaidų kiekis iš visų galimų klaidų programoje

Testavimo procesą galima gerinti jį optimizuojant. Vienas iš galimų gerinimo būdų yra vienetų testai. Vienetų testai padeda sumažinti klaidų kiekį nuo 20 % iki 90 %, tačiau tuo pačiu programų kūrimo išlaidos gali padidėti apie 30 % [1]. Vienas iš galimų būdų išlaikyti pasiektą vienetų testų efektyvumą ir palaikyti mažą kainą yra automatinis jų generavimas. Testų generavimas yra ganėtinai sudėtingas procesas, nes iškyla problemos dėl duomenų ir rezultatų

patikrinimo. Nežinant kokius tiksliai duomenis paduoti programai ir kokius rezultatus tikrinti, vienetų testų efektyvumas bus labai mažas. Būtina išspręsti šią problemą. Galimi tokie duomenų parinkimo ir/ar rezultatų tikrinimo variantai:

- Atsitiktinis duomenų generavimas.
- Duomenų generavimas paremtas programinio kodo analize.
- Duomenų generavimas ir rezultatų reikšmių nustatymas paremtas programos stebėjimu.
- Duomenų generavimas paremtus genetiniais algoritmais.
- Panaudojant vartotojų paruoštus duomenis ir rezultatus.
- Duomenų generavimas ir rezultatų patikra paremta OCL ar DbC apribojimais.
- Taip pat gali būti taikomi šių metodų deriniai.

Sprendimui spręsti pasirinktas variantas yra OCL apribojimai, nes tai yra formali kalba, turinti savitą specifikaciją, o DbC turi tiktai sprendimo idėją, tad vartotojas pats renkasi kaip jam apibrėžti programos apribojimus. OCL apribojimai išspręstų tiek duomenų parinkimo, tiek patikrinimo problemą.

2. PROJEKTAVIMO IR DIEGIMO ĮRANKIO ANALIZĖ

Šiame skyriuje apžvelgsime vienetų testų vaidmenį testavime, aptarsime automatinių vienetų testų generavimą remiantis OCL apribojimais ir su tai susijusias problemas. Taip pat bus apžvelgtos OCL vienetų testų generavimo bibliotekos problemos ir palyginti egzistuojantys sprendimai.

2.1. Programinės įrangos kokybė

Programinės įrangos kokybė yra vienas iš svarbiausių aspektų programinės įrangos kūrime. Nepasiekus tam tikro kokybės lygio, kuriamoje programinėje įrangoje programos tampa bevertės, nekonkurencingos su konkuruojančiais produktais, vartotojai nenori jos pirkti ir naudoti.

Programinės įrangos kokybę apibrėžia ISO 9126 standartas [5]. Standarte yra 6 rodikliai apibrėžiantys kokybę:

- Funkcionalumas
- Patikimumas
- Tinkamumas
- Naudingumas
- Palaikomumas
- Mobilumas.

Gamintojo požiūriu [6] kokybei įtaką daro, dar ir tokie veiksniai:

- Kodo skaitomumas.
- Palaikomumas.
- Testavimas.
- Kodo sudėtingumas ir t.t.

2.2. Testavimo metodologijos

Ekstremalus programų programavimas (*angl. Extreme Programing*), kurio pagrindinis tikslas yra išlaikyti aukštą kokybę ir greitai reaguoti į vartotojo reikalavimų pokyčius. Viena iš taisyklių tokiems tikslams išlaikyti yra būtinas vienetų testų kūrimas [7].

Kitas programinės įrangos kūrimo būdas, tai testais pagrįstas programinės įrangos kūrimas, kai jie sukuriami prieš programinės įrangos sukūrimą. Šis būdas vadinamas testais paremtu programavimu (*angl. Test-driven development*) [8].

Dar vienas būdas, tai formalieji metodai. Formaliais metodais aprašyta programinė įranga, suteikia galimybę ją patikrinti (*angl. validation, verification*) [9] iš formaliosios specifikacijos,o taip pat ir sugeneruoti testus . Tačiau toks metodas yra labai sudėtingas ir per brangus standartinės įrangos kūrimui, tad yra naudojamas tik išskirtiniuose projektuose.

Kokybės gerinimui yra naudojamas ir modeliais pagrįstos programinės (*angl. Model-driven engineering*) įrangos kūrimas [10]. Kadangi programinė įranga yra generuojama iš modelių, tai ypač sumažina klaidų tikimybę. Taip pat iš modelių galima butų sugeneruoti ir testus. Šis metodas praktikoje yra mažai išplėtotas metodas.

2.3. Testavimo tipai

Programinės įrangos testavimas yra bet kokia veikla, kuria siekiama įvertinti, ar atributas yra tinkamas programai ar sistemai, taipogi nustatyti, ar rezultatas yra toks, kokio mes laukiame [11]. Programinės įrangos testavimo sunkumai kyla dėl programinės įrangos sudėtingumo, jos dydžio, nes neįmanoma išbandyti visų galimų variantų. Pagrindinis testavimo tikslas gali būti kokybės užtikrinimo, tinkamumo ar patikimumo įvertinimas. Testavimo bandymai taip pat gali būti naudojami kaip bendras rodiklis kokybei užtikrinti. Testavimą galime apibūdinti kaip kompromisą tarp išlaidų, laiko ir kokybės.

Pagrindinės klaidų atsiradimo priežastys yra tokios [12]

- Vartotojas įvykdė neištestuotą kodo dalį.
- Vartotojas vykdė funkcijas, kitokia tvarka negu buvo testavime.
- Vartotojas įvykdė programą su neištestuotomis reikšmėmis.
- Vartotojas vykdė programa neištestuotoje aplinkoje.

Testavimas yra gana sudėtingas procesas kurį galime suskaidyti į du pagrindinius etapus. Pirmasis etapas sudarytas iš

- Vienetų testavimas (*angl.Unit testing*). Testuojama daugiausiai klasės ar kitokie struktūriniai vienetai.
- Integracinis (*angl.. Integration testing*). Yra apjungiami vienetai ir bandoma juos ištestuoti
- Sisteminis (*angl. System testing*). Testuojamas galutinis produktas.

Antrąjį etapą sudarys:

- Funkcinis (*angl. Functional testing*) arba juodos dėžės testavimas. Stengiama ištestuoti programos funkcijas neatsižvelgiant į kodą
- Struktūrinis (*angl. Structural testing*) arba baltos dėžės testavimas. Atsižvelgiama į testuojamos programos kodą ir pagal jį bandoma ištestuoti.

2.4. Vienetų testai

Vienetų testavimas – tai testavimas individualios arba susijusios grupės vienetų programinėje įrangoje [13]. Pagrindinis to tikslas yra izoliuoti kiekvieną programos dalelę ir parodyti, kad ji veikia teisingai [14]. Šitoks testavimo būdas leidžia aptikti klaidas dar programos kūrimo stadijoje. Atliekant tyrimus nustatyta, kad taip sumažina klaidų bent 20% ir daugiau procentų [1] [15] [16] [17] [18] [19]. Vienetų testai palengvina integracinį testavimą bei tinka regresiniam testavimui.

Pagrindiniai vienetų testų trūkumai:

- Negalima surasti visų klaidų.
- Sunku aptikti sistemines klaidas.
- Negalima atlikti ne funkcinių reikalavimų testavimą.

Vienetų testų rašymas yra brangus procesas, dažnai vienai kodo eilutei reikia nuo 3 iki 5 kodo eilučių testavimo [20]. Automatizavimas šioje srityje nėra labai toli pažengęs. Pačiu paprasčiausiu atveju vienetų testai generuojami su atsitiktinėmis reikšmėmis ir tikrinama ar programos veikla nesutrinka kritiškai. Toks testavimas dar vadinamas paviršutinišku (*angl. smoke testing*). Kitas panašus, tik sudėtingesnis, testavimo būdas -yra priimama, kad programa veikia teisingai ir laikoma etaloną. Tada tokiai sistemai yra sugeneruojami vienetų testai. Po to su kitomis programos versijomis sugeneruotais testais yra tikrinama ar gerai veikia programa. Toks testavimo būdas yra vadinas regresiniu (*angl. regresion testing*) [21]

2.5. Object Contsrain Language – OCL

OCL - formalioji modeliavimo kalba, kuria galima išreikšti santykius ir savybes modeliuojamiems elementams [22]. Ši kalba buvo sukurta IBM remiantis Syntropy[1] [23], jos pagrindinis tikslas specifikuoti programas, kurios neturi stipraus matematinio pagrindo ir kurių

[1] Syntropy – programų analizės ir projektavimo metodas, sukurtas 1990.

negalima specifikuoti kitomis formaliosiomis kalbomis, pavyzdžiui kaip Z[2]. Dabar OCL yra dalis UML standarto. Pagrindiniai atvejai, kuriais galima naudoti OCL yra [24]:

- Užklausų kalba.
- Apibrėžti invariantus klasėms ir tipams klasių modelyje.
- Apibrėžti prieš ir po sąlygas operatoriams ir metodams.
- Specifikuoti žinutes ir veiksmus.
- Specifikuoti apribojimus operatoriams.
- Specifikuoti išvestines taisykles UML modelių atributams.

OCL specifikuotas UML modelis suteikia galimybes iš jo automatiškai generuoti testų kūrimą, nes turime specifikuotus programos įėjimo duomenis ir rezultatus, o tai yra pakankama informacija testavimui. Eksperimentinių spendimų pavyzdžiai:

- *Unit Tests Generation Using Software Models And Imprecise Constraints* [25]. Vienetų testų generavimas, paremtas OCL apribojimais. Tyrime buvo naudojamas mutacinis testavimas
- *Automating Java Program Testing Using OCL and AspectJ* [26]. Testavimas grįstas, kai apribojimai tikrinami aspektiniu programavimu. Statinis programos modelis yra sužinomas iš klasių diagramos.
- *Quality of Automatically Generated Test Cases based on OCL Expressions* [27]. Testų generavimas remiantis OCL apribojimais. Pagrindinė tikslas buvo padengiamumo kriterijau tyrinėjimai..

2.5.1. OCL pavyzdys

Pateikiamas klasės pavyzdys, kuris yra specifikuotas OCL apribojimais. Klasė turi atributą `savybe` , kuri yra visada didesnė už 0. Taip pat užrašyti apribojimai sudėties metodas. Jo įėjimo parametrai yra visada didesni už nulį, o rezultatas lygus abiejų sumai.

```
context Klase
      inv: self.savybe >= 0 //Čia yra komentaras

context Klase::sudetis(parametras1 : number, parametras2 : number ): number
      pre: parametras1 > 0
      pre: parametras2 > 0
      post: result =  parametras1 + parametras2
```

[2]Z – formali specifikavimo kalba.

2.6. Design by Contract – DbC

Design by Contract arba ***Programming by Contract*** – yra būdas kaip projektuoti, kurti programinę įrangą [28]. Šis būdas nurodo, kaip programų kūrėjai turėtų formalizuoti programinės įrangos komponentus. Ši idėja pirmiausia atsirado Eiffel programavimo kalboje ir buvo suderinama programavimo kalbos lygmeniu. Viso to privalumai tai, kad :

- Kompiliatorius užtikrina sintaksės teisingumą ir teisingą panaudojimą.
- Kompiliavimo metu galima valdyti apribojimų tikrinimą.
- Visos sąlygos yra matomos programų kūrėjams.

Specifikuoti galima taip:

```
class                   -- Klasės pavadinimas
      feature           -- Metodas
            require     -- Sąlygos prieš metodo vykdymą
            local       -- Kintamųjų aprašai
            do          -- Metodo kūnas
            ensure      -- Po salygps
            rescuse     -- Išimtys (angl. Exception)
            end         -- Metodo pabaiga

      invariant         -- Klasės invariantai
end                     -- Klasės pabaiga
```

Dabar DbC būdas yra naudojamas ne vienoje programavimo kalboje: C#, VB.NET, Cobra, D ir kitose . Taip pat yra sukurtas ne vienas karkasas skirtas programavimo kalboms, kurios neturi suderinamumo su šiuo DbC būdu. Tai yra pagrindinis DbC privalumas prieš OCL.

DbC ir OCL turi tam tikrų bendrų bruožų. Pagrindinis panašumas tas , kad galime specifikuoti programos veikimą ir kad galime apibrėžti:

- Klasės invariantus
- Prieš sąlygas
- Po sąlygas.

Skirtumas tarp šių ideologijų toks, kad DbC aprašo patį specifikavimo būdą ir jo naudojimą, o OCL yra specifikavimo kalba, turinti savo sintaksę, platesnius panaudojimo atvejus.

2.7. Orakulo šablonas

Vienetų testų generavimo paremto OCL apribojimais veikimo idėja yra grįsta orakulo šablonu. Orakulu gali būti programos specifikacija, pavyzdžiai kitų programų ar programuotojo požiūris kaip programa turėtų veikti [29]. Šablono veikimo mechanizmas grįstas tuo, kad jisai

turi įvertinti, ar su testiniais duomenimis testai veikia teisingai. Įvertinimui reikalingi rezultatų generatorius, kuris generuoja tikėtinus rezultatus įėjimo parametrams, ir palygintojas, kuris lygina generatorius rezultatus su gautais. Šitoks veikimo principas gali būti rankinis, automatinis arba dalinai automatinis. Testavimo rezultatų teisingumas gali būti toks:

- Teisingai praeitas *(angl. valid pass*) – kai rezultatas sutampa su rezultatu, kurio buvo tikėtąsi.
- Teisingas nepraeitas (*angl. valid no pass*) – kai orakulas yra teisingas, tačiau gautas rezultatas toks nėra .
- Suklastotas praėjimas (*angl. spurious pass*) – kai orakulas yra klaidingas ir gautas rezultatas su sutampa orakulu.
- Suklastotas nepraėjimas (*angl. spurious no pass*) – kai orakulas yra klaidingas ir rezultatas nesutampa su orakulu.
- Atsitiktinai praeina (*angl. concidental pass*) – kai rezultatai atsitiktinai sutampa
- Atsitiktinai nepraeina (*angl. concidental no pass*) – kai rezultatai nesutampa. Klaidų turi ir programa ir orakulas.

Rezultatų palyginimui naudojami šie būdai:

- Teisimo (*angl. judging*) – kai testuotojas nusprendžia ar testas praėjo.
- Specifikuoti (*angl. prespecification*) – kai rezultatai žinomi iš anksto ir gali būti palyginti su gautaisiais..
- Aukso standartas (*angl. gold standart*) – kai turima analoginė sistema. Yra tikrinama ar analoginėje sistemoje su tokais pat iėjimo parametrais bus tokie pat rezultatai.

2.7.1. Orakulo šablonas, OCL ir vienetų testų karkasas

Įgyvendinimui orakulo šablono vienetų testų generavime reikia OCL apribojimų, nes jie išspręs orakulo problemą:

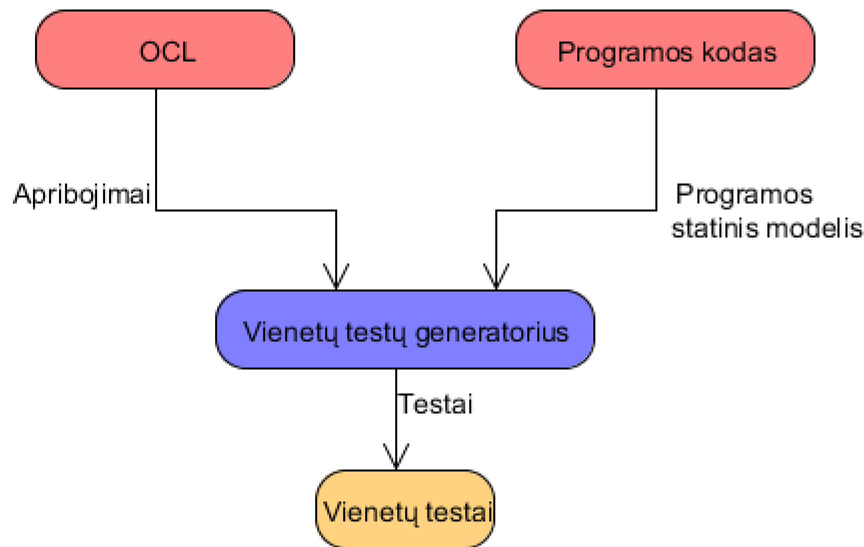
- Generuojamas vienetų testų įėjimo reikšmės atitinka OCL prieš sąlygas.
- Generuojamų testų rezultatų reikšmes atitinka po ir invariantines sąlygas.

Palyginimą atlieka vienetų testų karkasas ir žmogus. Jeigu rezultatai tenkina testuotojus, tai užtenka ir karkaso palyginimo. Esant rezultatams, kurie netenkina testuotojo sprendimą gali priimti ir žmogus. Jis gali nuspręsti ką koreguoti - programą ar apribojimus.

Idealiu atveju OCL specifikuota programa turėtų atitikti idealųjį orakulą (*angl. perfect oracle*). Programa neturėtų turėti nei vienos klaidos. Tačiau realiai užtikrinti, kad su visais įėjimo parametrais, programa veiktų teisingai yra beveik neįmanoma [30]

2.8. Darbo tikslai ir uždaviniai

Pagrindinis darbo tikslas yra sukurti automatinį vienetų testų generatorių, kuris remtųsi OCL apribojimais ir programiniu kodu kaip programos modeliu, kuris atstotų UML klasių diagramą. Testavimo procesas remiasi orakulo šablonu. OCL apribojimai yra tiesiogiai į komponuojami į programinį kodą, nes vienetų testai yra skirti programuotojams, todėl jiems būtų lengviausia kode užrašinėti apribojimus.



2 pav. Kuriamos sistemos kontekstas

OCL apribojimai daugeliu atveju nenustatinėja tikslių įėjimo ir rezultato reikšmių [25], o tik išskiria tam tikrą aibę iš visų galimų atvejų. Testuojant programinę įrangą svarbiausia 3 dalykai [31]:

- Įėjimo reikšmių parinkimas.
- Sistemos paleidimas su įėjimo parametrais – SUT (*angl. System under test*),o detaliau UUT (*angl. Unit Under Test*) , nes bus naudojami tik vienetų testai
- Rezultatų patikrinimas.

Parametrų įėjimo reikšmių, generavimui bus naudojamas toks algoritmas:

```
Jeigu parametras neturi apribojimų, tai
      generuoti atsitiktinę reikšmę.
Jeigu parametras turi apribojimą(ų) , tai
      generuoti apribojimui ribines reikšmes ir atsitiktines reikšmes iš
galimos aibės
Generuoti vienetų testus, naudojant sugeneruotas įėjimo reikšmes, kol galima
sudarinėti testus su specialiomis reikšmėmis arba/ir iki vartotojo nustatyto
kiekio.
```

Pastabos:

- Maksimalus testų kiekis gali būti apribotas vartotojo, jeigu yra per didelis galimų variantų kiekis.
- Atsitiktinių skaičių generavimas vyksta pagal normalųjį pasiskirstymą.

Programos ištestavimo lygis priklauso nuo apribojimų kiekio reikalingo specifikuoti programą. Jeigu programa neturi apribojimų, tai bus atliekamas paviršutiniškas testavimas *(angl. smoke test*), o esant programai pilnai specifikuotai OCL apribojimais, jinai bus ištestuota teoriniu atveju pilnai ir atitiks specifikavimą.

2.9. Vienetų testavimo įrankiai, jų klasifikacija.

Vienetų testavimo automatizavimui yra sukurtas ne vienas įrankis. Šias programas pagal veikimo principą galime suskirstyti į tris pagrindines grupes [32]:

- Testų vykdytojai. Pvz.: Junit, CppUnit, Nunit ir t.t.
- Testų generatoriai. Pvz.: AgitatorOne, Parasoft Jtest
- Testų elgsenos stebėtojai: Pvz.: Parasoft Jtest.

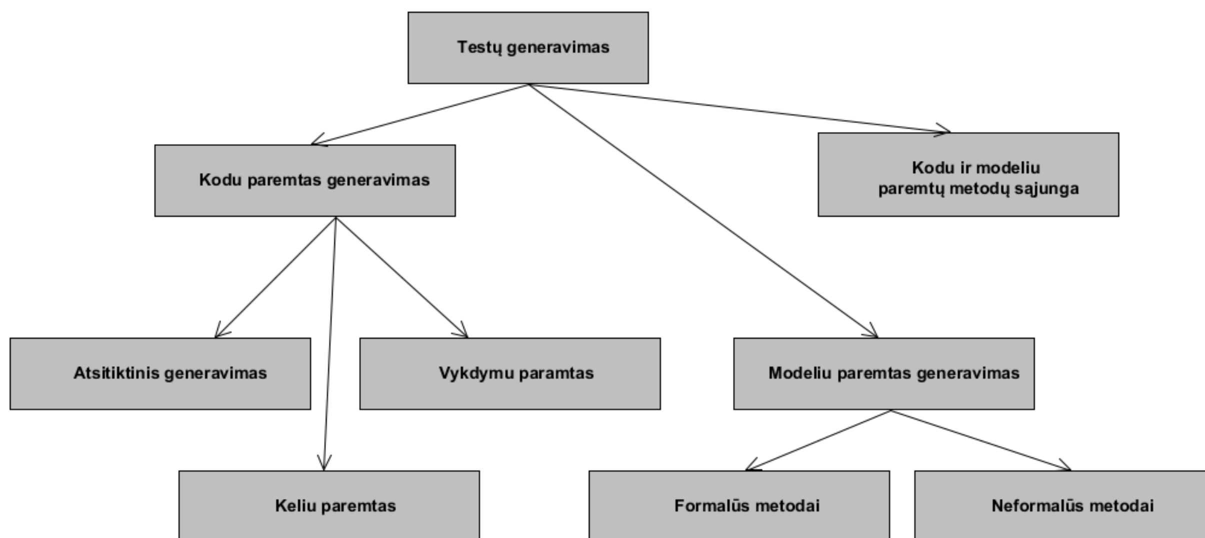
Automatinių testų generatoriai susiduria su pagrindinėmis trimis problemomis, kurios įtakoja testų kokybę.

1. Greitumas. Kiek greičiau įrankis gali sukurti testą programuotojui.
2. Tikslesni testų įėjimų parametrai. Kiek gali padėti generuoti tikslesnius įėjimus.
3. Tikslesni testavimo rezultatai (orakulai). Kiek gali padėti generuoti tikslesnius rezultatus.

Pirmajai problemai išspręsti sąsajos testavime naudojami, įrankiai, kurie gali įrašyti ir paleisti (*angl. capture-ant-replay*) sudaryta testą. Vienetų testavime automatiškai sugeneruoti metodų parametrus, netikrų (*angl. mock*) objektų generavime, šablonų panaudojimas ar įterpiamas iš anksto vartotojo parašytas kodas į testą.

Tikslesnių įėjimų generavimui gali būti panaudojamas OCL arba DbC [33]. Galimi dar atsitiktiniai, euristiškai parinkti metodų parametrai. Tikslesniems testų orakulams generuoti, galima panaudoti parametrizavimą ar teiginius (*angl . assertions*) .

Testų generatorių skirstomi taip [25]:



3 pav. Vienetų testų generavimo būdai

Galimi ir šių variantų kitokie deriniai, derinant jų savybes. Mano kuriamame sprendime yra savybių iš atsitiktinio generavimo ir formalių metodų.

2.10. Panašių sprendimų apžvalga

2.10.1. Java Modeling Language - JML

JML - yra specifikavimo kalba Java programoms. Specifikavimas remiasi DbC principu. Yra palaikomos prieš, po ir invariantinės sąlygos. Specifikacijos yra rašomos JAVA metodų, kintamųjų komentaruose tiesiogiai programos kode. Šitoks programinis kodas gali būti sukompiliuotas su bet kokiu JAVA kompiliatoriumi. Tačiau paleidimui reikalingas specialus JML įrankis. Įrankio pagrindinis tikslas realiu laiku tikrinti programos veikimą. Prieš vykdymą yra įterpiami JAVA kalbos tikrintojai (*angl. assertion*) į programą, Tačiau yra galimybė generuoti ir vienetų testus. Vienetų testų programos kodas vartotojui yra nematomas. Vartotojas turi tik aprašyti paduodamus programai duomenis

- Programos pliusai
  - Palaiko DbC apribojimų principą
  - Atvirojo kodo.

- Apribojimu užrašymo kalba yra išplėtota ir didelė.
    - Turi Eclipse papildinį.
    - Testų generavimo papildinį
    - Generuoja vienetų testus
- Programos minusai:
    - Testų generavimo paleidimas yra komplikuotas. Yra galimybė pasirinkti testavimo duomenis ir duomenų parinkimo strategijas. Testo kodo ,kurį galėtų vartotojas modifikuoti, nėra.
    - Reikalingi specialūs įrankiai paleidimui.
    - Programa yra labiau skirta tyrinėjimui negu profesionaliam darbui.

Trumpas specifikacijos aprašymas:

JML specifikacijos yra užrašomos programiniame kode pasinaudojant tokia sintakse:

```
//@ <JML specification>
```

arba

```
/*@ <JML specification> @*/
```

1 lentelė. JML kalbos trumpas aprašymas

Raktažodis	Aprašas
requires	Atitinka prieš sąlygas metodui
ensures	Atitinka po sąlygas metodui
invariant	Invariantas.
\result	Metodo gražinamas rezultatas

Programos specifikavimo pavyzdys:

```
class Plotas {
	private /*@ spec_public @*/ int plotas;
	//@ public invariant plotas >= 0;

	//@ requires krastineA > 0
	//@ requires krastineB > 0
	//@ ensures \result  == krastineA * krastineB
	public int staciakampioPlotas(int krastineA, int krastineB){
		plotas = krastineA * krastineB;
		return plotas;
	}
}
```

Vienetų testo duomenų parinkimo pavyzdys:

```
public abstract class Plotas_JML_TestData extends TestCase {

	public IntIterator vintIter(String methodName, int argNum){
	return vintStrategy.intIterator();
	}
	private IntStrategyType vintStrategy = new IntStrategy() {

	protected int[][] staciakampioPlotasData() {
		int data[1][2] = new int[1][1];
		int data[0][0] = 5;
		int data[0][1] = 1;
		return data;
	}
}
```

2.10.1. TestGen4J

Atvirojo kodo biblioteka skirta generuoti vienetų testus. Nėra palaikymo nei OCL, nei DbC. Testai yra generuojami Junit testavimo bibliotekai. Palaikomi tiktai nesudėtingi metodų parametrai. Tyrime „*Comparison of Unit-Level Automated Test Generation Tools*“ [42] buvo pasiektas 28% sugautų mutantų kiekis naudojantis TestGen4J. Kitų panašių įrankių rezultatai nebuvo geresnis ir siekė iki 42%.

Pagrindinis privalumas šio įrankio toks, kad galima aprašyti metodų parametrus XML (*angl. eXtensible Markup Language*) failuose.

Parametrų parinkimo taisyklės:

```
<Rules xsi:noNamespaceSchemaLocation="config/jtestcase.xsd">
	<DataType name="String">
		<Case value="NULL"></Case>
		<Case value="EMPTY"></Case>
		<Case value="SPACE"></Case>
	</DataType>

	<DataType name="double">
		<Case value="-1"></Case>
		<Case value="1"></Case>
		<Case value="0"></Case>
	</DataType>
</Rules>
```

Vienetų testo pavyzdys:

```
public void testPlotas (){
	Plotas.plotas = new Plotas();
	plotas.staciakampioPlotas(0 , 1);
	plotas.staciakampioPlotas(-1, 1);
	plotas.staciakampioPlotas(0 ,-1);
	plotas.staciakampioPlotas(1 , 0);

}
```

2.10.2. JTest

JTest yra universalus testavimo įrankis, vienas tobuliausių rinkoje esančių komercinės paskirties produktų. Palaiko statinę kodo analizę, realiu laiku vykdomą klaidų paiešką, vienetų testų generavimą. Generuoti vienetų testai yra skiriami regresiniam testavimui. Generavimo metu sistema laikoma orakulu ir kad sistema veikia teisingai.

- Programos pliusai
    - Palaiko DbC apribojimus
    - Generuoja karkasus ar netikrus (angl. *mock*) objektus.
    - Objektų bazės turėjimas.
    - Testų parametrizavimas.
    - Palaikomas funkcinis testavimas.
    - Turi integraciją su Eclipse ir kitais programinės įrangos kūrimo įrankiais.
    - Yra versijos skirtingoms kalboms.
- Programos minusai
    - Nėra palaikomas OCL
    - Labai brangus.

2.10.3. Code Contracts

Code Contracts yra .NET platformos plėtinys, kuris atsirado nuo 4 versijos. Šis plėtinys tiesiogiai leidžia tikrinti prieš, po ir invariantines sąlygas programavimo kalbos lygiu. Nereikalingos jokios papildomos bibliotekos. Šitoks tiesioginis programavimo kalbos praplėtimas leidžia išnaudoti visus programavimo kalbos privalumus. Programavimo kalbų įrankiai užtikrina sintaksės teisingumą. Tai sumažina sintaksinių klaidų kiekį apribojimuose.

Code Contracts veikimo idėja grįsta Eiffel DbC pagrindu.

Trumpas specifikacijos aprašymas:

2 lentelė. Code Contracts trumpas raktažodžių aprašas

Raktažodis	Aprašas
Contract.Requires	Atitinka pre sąlygas metodui
Contract.Ensures	Atitinka post sąlygas metodui
[ContractInvariantMethod]	Invariantas.
Contract.Result<>	Metodo gražinamas rezultatas

```
class Plotas {
      private int plotas;
      public int staciakampioPlotas(int krastineA, int krastineB)
      {
            Contract.Requires(krastineA > 0);
            Contract.Requires(krastineB > 0);
            Contract.Ensures(Contract.Result<int>() ==
                  Contract.OldValue(krastineA)
                        * Contract.OldValue(krastineB));
            plotas = krastineA * krastineB;
            return plotas
      }

      [ContractInvariantMethod]
      private void ObjectInvariant ()
      {
            Contract.Invariant(this.plotas > 0 );
      }
}
```

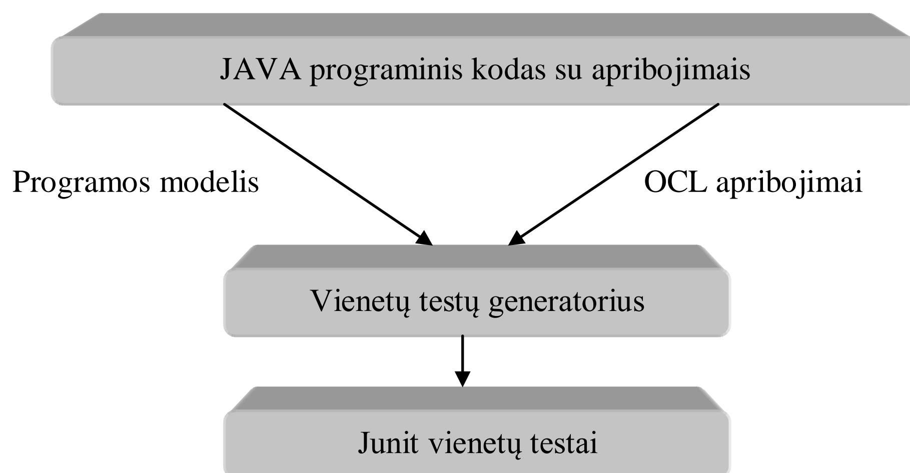
3. AUTOMATINIO TESTŲ GENERATORIAUS PROJEKTINĖ DALIS

3.1. Automatinio testų generatoriaus paskirtis

Sistemos tikslas yra sukurti vienetų testų generatorių. Vienetų testų generatorius generuotų testus iš OCL apribojimų ir programinio kodo. Panašaus tipo generatoriai naudoja UML modelius [27] [34], tačiau naudoja ne vien tik klasių diagramą, bet ir būsenų, sekų diagramas. Statinę informaciją, kurią galima sužinoti iš klasių diagramos, galima tiesiogiai sužinoti iš programinio kodo. Tokia tiesiogiai sužinoma informacija yra visada nauja, nes nereikia atnaujinti modelio. Taip pat yra pašalinami nesklandumai susiję su modelio sutapatinimu su tikra programa. UML modelio privalumas, kad nėra prisirišama prie tam tikrų įrankių ar programavimo kalbų.

Realizacijai pasirinkome:

- JAVA programavimo kalbą
- Junit 4 vienetų testų testavimo karkasą



3.2. Esminiai reikalavimai

- Generuoti vienetų testus:
    - Remiantis OCL apribojimais.
    - Remiantis programiniu kodu
- Generuojamų vienetų testų parametrų konfigūravimas:
    - Skaičių formatas

- Generuojamų testų kiekis
- Junit testo parametrų savybių nustatymas
- Atsitiktinių skaičių ypatingi atvejai:
  - -0,0 arba +0,0
  - ±∞
  - NaN (*angl. Not a Number*)

3.3. Nefunkciniai reikalavimai

- Programos veikimas turi nesutrikti, jeigu sintaksiškai buvo neteisingai užrašyti apribojimai arba buvo panaudotos dar neįgyvendintos OCL kalbos savybės.
- Naujų OCL kalbos savybių įdėjimas turi reikalauti kuo mažiau keitimų
- OCL versija 2.0
- OCL apribojimai užrašomi programiniame kode

3.4. OCL kalbos įgyvendintos funkcijos

Formalioji OCL kalba yra ganėtinai didelė: specifikacija užima virš 200 lapų. Visos jos įgyvendinimas yra komplikuotas ir labai sudėtingas uždavinys. Įgyvendintos buvo tik esminės ir reikalingiausios funkcijos.

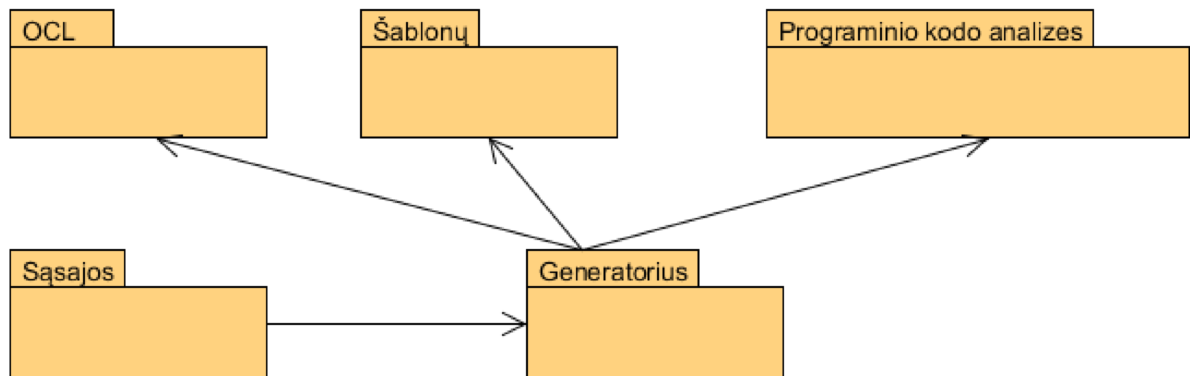
3 lentelė. OCL kalbos suderinami elementai

OCL kalbos funkcija (-os)	Aprašas	Pavyzdys
inv, pre, post	Sąlygos	pre: a < 5; post: b > 6; inv: a <> true
self	Elementas, išskiriantis klasės - konteksto lygio elementus	pre: self.a > 0;
result	metodo gražinamas rezultatas	post: result < 10;
--	Komentaras	// Čia yra komentaras
@pre	Prieš tai buvusi reikšmė	post: a = a@pre;
<, <=, >, >= , <>. +, -, /, *	Operatoriai	post: result <> 10;
selft.savybe	Klasės, konteksto savybė	pre: selft.savybė< 5;

3.5. OCL apribojimai užrašomi programavimo kalboje

OCL apribojimai užrašomi tiesiogiai programiniame kode. Pasirinktas šitas būdas, nes vienetų testų rašymas yra daugiausiai skirtas programuotojams [35]. Jiems apribojimus būtų patogiausia užrašyti tiesiogiai kode. Taip pat sumažėja tikimybė, kad pasikeitus kodui, nebus pakeisti apribojimai arba tiesiog pamiršta juos atnaujinti.

3.6. Architektūra, jos savybės



4 pav. Paketų diagrama

3.6.1. Sąsaja

Sąsajos paketas yra skirtas bendravimui tarp programos ir vartotojo. Sąsaja galima dviejų tipų:

- Komandinės eilutės
- Grafinė vartotojo sąsaja

Grafinė vartotojo sąsaja yra įgyvendinta Eclipse įrankyje. Norint turėti grafinę sąsają reikia įdiegti priedėlį. Vartotojo sąsaja yra minimalistinė. Iškvietimas testų generavimo ir konfigūravimo.

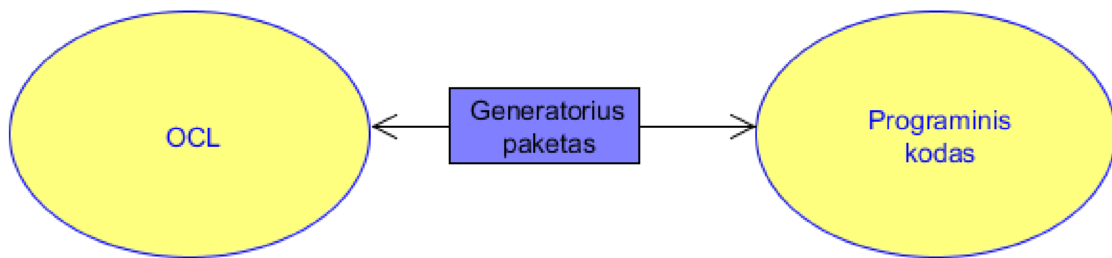
3.6.2. OCL

OCL paketo pagrindinė užduotis yra išanalizuoti apribojimus, paversti juos suprantamu formatu. Pakete OCL kalbos specifikacija yra aprašyta EBNF (Extended Backus–Naur Form) gramatika. Jam padavus tekstiniu pavidalu apribojimus, jisai paverčia juos programai vidiniu

suprantamu formatu. EBNF gramatikos analizei yra panaudotas JavaCC (Java Compiler Compiler) įrankis.

3.6.3. Programinio kodo analizės

Programinio kodo analizės paketas yra skirtas programos kodo analizei. Jisai išanalizuoja kokios yra klasės, kokie yra metodai, metodų parametrai. Visai šiai informacijos gavimui yra panaudota JAVA *Reflection* technologija[3]

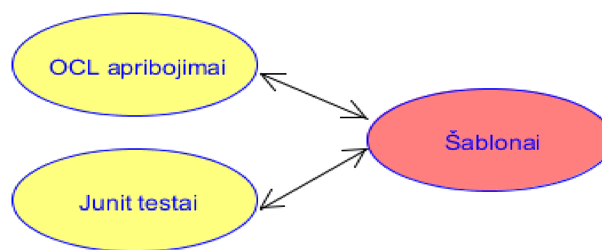


5 pav. OCL susiejimas su programos kodu

3.6.4. Šablonų paketas

Šablonų pakete yra saugojama vienetų testų ir OCL situacijų šablonai, tai

- Testų klasės
- Testo
- Palyginimo ir t.t.



6 pav. Šablonų susiejimas su Junit ir OCL

Junit vienetų testo šablono pavyzdys:

```
@Test
public void test[Pavadinimas testo]() {
    [Vieneto testo kūnas]
}
```

[3] Reflection – tai procesas, kai galima stebėti, modifikuoti programą jos veikimo metu.

OCL apribojimo šablono pavyzdys:

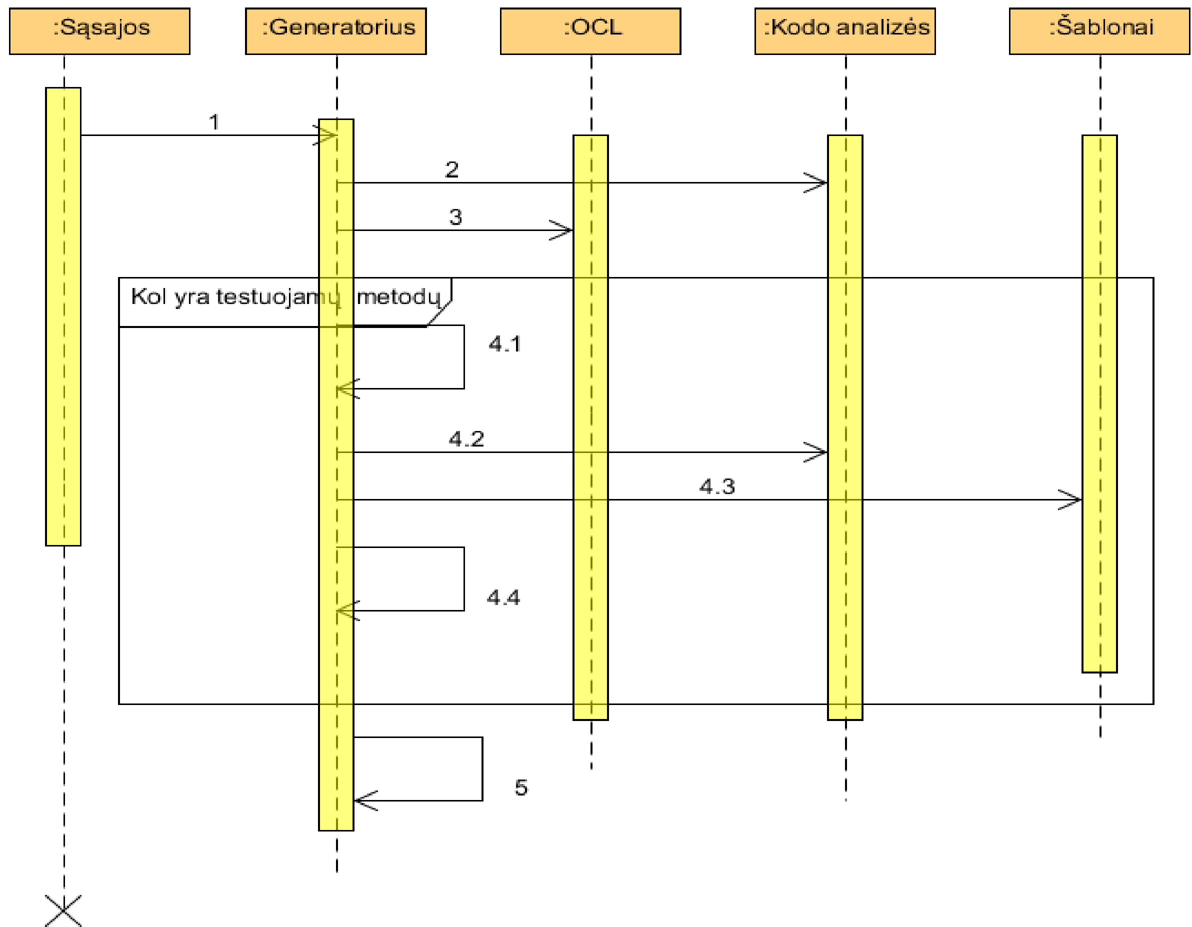
Loginė išraiška : `post: result > 5`

Šablonas: `assertTrue([Loginė išraiška]);`

3.6.1. Generatoriaus paketas

Generatoriaus paketas yra kaip valdiklis, kuris surenka informaciją iš kitų paketų ir sugeneruoja rezultatą – vienetų testą

1. Gaunama užklausa testuojamai klasei
2. Iš testuojamos klases surenkama informacija apie OCL apribojimus
3. Apribojimai išanalizuojami
4. Generuojama tol, kol yra testuojamų metodų
   4.1. Analizuojamas apribojimas
   4.2. Surenkami informacija apie reikalinga apribojimą iš programinio kodo
   4.3. Atliekamos transformacijos pagal šablonus
   4.4. Generuojami testas
5. Išsaugojami testai



7 pav. Vienetų generavimo veikimo charakteris

3.7. Atsitiktinių reikšmių generavimas

Metodų parametrų reikšmės vienetų testavime yra vienas iš svarbiausių dalykų. Nepaisant to, kad reikšmės yra generuojamos atsitiktinai, vis tiek yra tam tikros generavimo taisyklės.

- Visi skaičiai generuojami pagal normalųjį pasiskirstymą.
- Realiųjų skaičių aibėje tikslumas yra nustatomas vartotojo, kiek skaičių po kablelio yra aktualu.
    - Pagal tikslumą yra atsižvelgiama į skaičių palyginimus. Esant tikslumui 2 skaičiams po kablelio, skaičiai 0,201 ir 0,202 bus lygus.
    - Tikslumas taip pat turi įtakos ir realiųjų atsitiktiniame generavime. Jei generuosime skaičių didesnį už 1, esant tikslumui 2 skaitmenų, tai bus sugeneruotas 1,01 skaičius.

4 lentelė. Parametrų generavimo taisyklės

Situacija	Taisyklė
> arba <	Sugeneruojamas skaičius vienu vienetu didesnis (mažesnis)
>= arba <=	Sugeneruojamas lygus skaičius, taip pat vienetu didesnis ir mažesnis
<> (nelygu)	Sugeneruojami dvi ribinės reikšmės, viena vienetu mažesnė, o kita vienetu didesnė
Jeigu į skaičių aibę patenka 0	Sugeneruojami +0,0 ir -0,0

Generuojant testus metodui yra:

1. Sugeneruojama visi galimi įėjimai pagal apribojimų taisykles skaičių generavime.
2. Generuojamas testų kiekis pagal sugeneruotus testinius duomenų rinkinius ir dar norimas, jeigu netenkinamas vartotojo nustatytų testų kiekis, su atsitiktinėmis reikšmėmis.

3.7.1. Pavyzdys apribojimų ir sugeneruoto testinio atvejo

Testinės programos pavyzdys:

```
@Invariant(ocl="inv:var1 > 0;")
public class Test1 {
        @Invariant(ocl="inv:var1 < 100;")
        private int var1;

        @Context(ocl="pre: p0 > 0 and p0 < 50;post:var1 > p0")
        public void test1(int p0)
        {
                var1 = p0 * 2;
        }
}
```

Testo pavyzdys:

```
public class Test{
    @Test
    public void testTest5() {
        test.mag.Test1 _test1 = new test.mag.Test1();// Sukuriamas objektas
        //Tikrinami invariantai
        assertTrue(_test1.getVar1 > 0);
        assertTrue(_test1.getVar1 < 100);
        //Nustatoma salyga pre: p0 < 0
        int p0 = 48;
        // Nustatoma salyga pre: slef.p0 > 0
        _test1.setVar1(6);
        //Metodo iškvietimas
        _test1.test5(p0);
        //Tikrinimas post salygu
        assertTrue(_test1.getVar1 > p0);
        //Tikrinami invariantai
        assertTrue(_test1.getVar1 > 0);
        assertTrue(_test1.getVar1 < 100);
    }
}
```

3.7.2. Sistemos testavimas

Sistemą, kuri testuos kitas sistemas, gerai ištestuoti yra būtina. Testų generatoriaus testavimui buvo pasirinkti du būdai:

- Pirmuoju būdu buvo rašomi vienetų testai sistemos funkcionalumui.
- Antruoju atveju buvo kuriama dirbtinė programa ir jai užrašomi apribojimai. Programa buvo pritaikyta prie tų apribojimų. Tokiai programai buvo sugeneruojami vienetų ir tikrinama ar vienetų testai duoda teigiamus rezultatus.

4. AUTOMATINIO TESTO GENERATORISUS TYRIMAS

4.1. Įvadas

Šioje dalyje pateikiama informacija, kaip projektas atitinka užsakovo keliamus reikalavimus. Aprašomi svarbiausi programos trūkumai, pateikiami trūkumų šalinimo būdai, siūlymų tobulinti programą būdai ir pateikiami siūlymų aprašymai.

4.2. Kokybės analizė

4.2.1. Specifikacijos atitikimas

Sukurtas automatinis testų generatorius įgyvendino visus specifikacijoje aprašytus reikalavimus. Tačiau ne visi reikalavimai atitinka užsibrėžtus kokybės tikslus.

4.3. Iškilusios problemos

4.3.1. OCL susiejimas su programavimo kalba.

OCL yra formali programavimo kalba, kurioje viskas yra detaliai apibrėžta - tipai, loginiai veikimo elementai. Tačiau joje nėra jokių išvestinių darinių. Pvz.: yra kintamųjų sąrašas pasižymintis tam tikromis savybėmis. Java programavimo kalba nėra formalioji programavimo kalba, joje apibrėžimo laipsnis yra mažesnis. Jeigu mes turime sąrašą, tai sąrašas gali būti ir masyvas, ir kolekcija ir dar kitokia struktūra. Atliekant transformaciją iš OCL į JAVA galėjo atsirasti tam tikrų nesklandumų. Nesklandumų buvo išvengta, tačiau ateityje ja naudojantis dėl to gali kilti nesuderinamumo problemų [36]

4.3.2. Objektų kūrimo problematika

JAVA programavimo kalba yra objektinė. Reikalingi objektų sukūrimai, daugeliu atveju reikia prieš iškviečiant klasės funkciją reikia sukurti objektą. Objekto sukūrimas pasidaro sudėtingas, jeigu sukūrimui reikia tam tikrų parametrų (konstruktorius yra su parametrais). Sukurta programa su sudėtingais parametrais negali inicializuoti objektų, todėl negalima sugeneruoti toje situacijoje vienetų testų. Buvo nuspręsta inicializuoti tik tas klases, kurios turi numatytąjį konstruktorių t.y. tokį, kuris neturi parametrų.

4.4. Siūlymai tobulinti programą

Atsižvelgiant į problemas, su kuriomis buvo susidurta tyrimo metu, buvo pasiūlyti patobulinimai.

4.4.1. Grafinis nustatymų valdymas

Dėl nepatogumo vartotojui parametrus įvedinėti naudojant komandinės eilutės sąsają parametrus, buvo nuspręsta padaryti grafinę programos sąsają, kuri leistų parametrų keitimą padaryti lengvesnį, greičiau perprantamą.

4.4.2. Suderinti su ANT[4]

Suderinti su automatiniais kompiliavimo įrankiais. Vartotojui tai leistų įtraukti testų generavimą ir vykdymą į automatinį programos kompiliavimo procesą.

4.5. Išvados

Programos silpnosios vietos yra žinomos. Žinoma kokios OCL kalbos savybės nėra įgyvendintos ir palaikomos. Visa tai leis iš anksto priimti sprendimus, prieš rašant apribojimus ir generuojant testus. Taip pat reikia testuotojams atkreipti dėmesį į OCL ir JAVA kalbos skirtumus.

[4] ANT – automatinis kompiliavimo įrankis, skirtas JAVA programoms.

5. AUTOMATINIŲ TESTŲ GENERATORIAUS EKSPERIMENTINIS TYRIMAS

5.1. Tikslas

Pagrindinis tikslas yra ištyrinėti vienetų testų generatoriaus veikimą, jo veikimo charakteristikas. Tyrimas bus atliktas remiantis defektais remtu testavimo (*angl. Fault-based testing*) principu [37], pritaikius mutacinį testavimą (*angl. mutation testing*) [38]. Taip bus patikrinta ar sugeneruoti testai sugeba atrasti klaidas.

5.2. Mutacinis testavimas

Mutacinis testavimas yra testavimo būdas kai yra nedaug modifikuojamas išeities kodas.[39]. Tokia modifikuota programa yra vadinama mutantu [40]. Pagrindinis tokio testavimo tikslas yra patikrinti testų efektyvumą:

- Ar testai sugeba rasti klaidingai veikiančius mutantus?
- Aptikti trūkumus, silpnąsias vietas testuojamuose duomenyse.
- Aptikti retai vykdomas arba niekada nevykdomas kodo vietas.

Jeigu testai surado, kad yra klaida mutante, toks mutantas yra vadinamas aptiktu mutantu.

5.2.1. Mutantų sudarymo taisyklės

Mutantai buvo sudaryti keičiant operatorius. Keitimai buvo atliekami aritmetinėse ir palyginimo operacijose.

- Aritmetiniai operatoriai: [+ , - , * , /]
- Palyginimo operatoriai: [<, <=, >, >=, !=, ==]

Mutantų generavimas vyko modifikuojant sukompiliuotą programos versiją. Peržiūrint programos *bytecode*[5], radus operatorių iš kurios nors grupės buvo sugeneruoja mutantai su likusiais grupės operatoriais. Kiekvienam operatoriui po mutantą.

5.3. Testavime naudojamos metrikos

Testų kokybei įvertinti bus naudojamos tokios metrikos [41]:

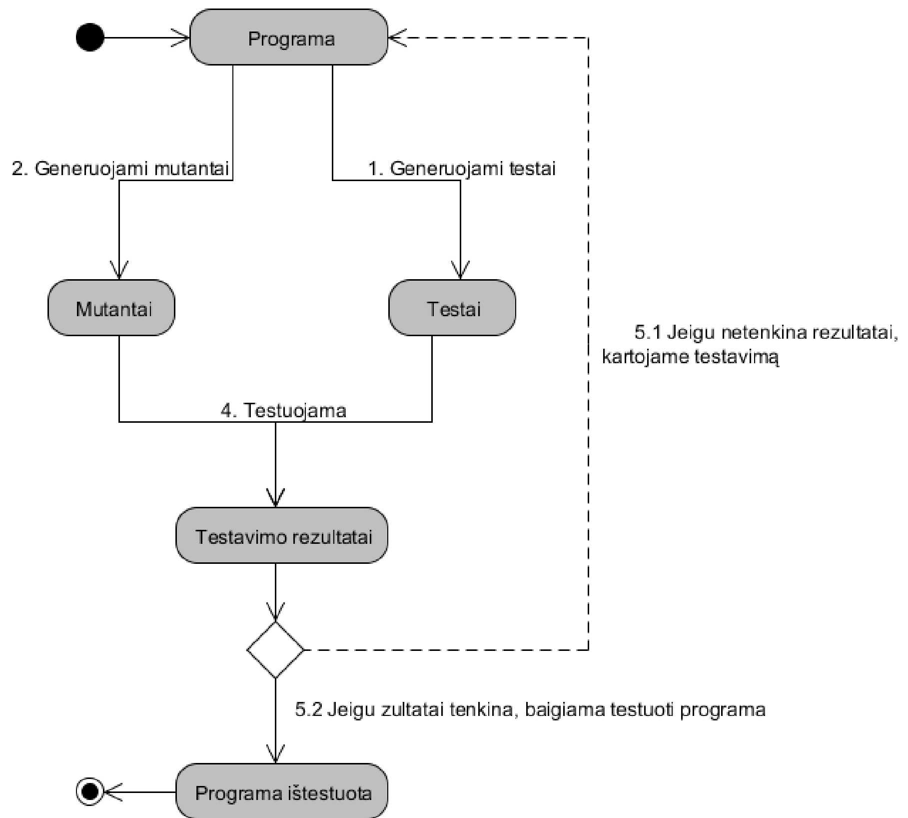
[5] Bytecode - instrukcijų rinkinys skirtas interpretatoriui, kuris vykdo programą.

- Kodo eilučių padengiamumas (*angl. Line Coverage*) – parodo kiek kodo eilučių buvo programos įvykdyta.
- Šakų padengiamumas (*angl. Branch Coverage*) – parodo kiek kode sąlygų buvo įvykdyta.
- Aptiktų ir visų mutantų santykis.

5.4. Tyrimo vykdymo scenarijus.

1. Programai užrašomi OCL apribojimai
2. Sugeneruojamo testai
   2.1. Patikrinama ar originali programos versija su esamais testais veikia teisingai.
3. Sugeneruojami mutantai
4. Testuojami mutantai. Mutantams naudojami tokie patys testais su tokiais pat testavimo duomenimis.
5. Testavimo rezultatai lyginami su užsibrėžtais testuotojo tikslas. Ar jį tenkina pagautų mutantų kiekis, kodo ar šakų padengiamumas.
   5.1. Jeigu netenkinama grįžtama į 1 punktą ir tobulinami, tikslinami apribojimai, siekiant pagerinti charakteristikas. Jeigu nebegalima pasiekti tam tikrų šakų, ar kodo eilučių, gali būti, kad programa turi klaidų ir pasiekti tų vietų negalima.
   5.2. Jeigu tenkina, testavimas baigiamas

Tyrimo metu bus atliktos dvi iteracijos. Įvertinti kiek galima pagerinti tyrimo charakteristikas tobulinant, tikslinant apribojimus



8 pav. Tyrimo – testavimo veiklos schema

5.5. Eksperimentinių programų aprašymas.

Eksperimentui įgyvendinti buvo pasirinktos 5 programos, nuo nesudėtingų ir nedidelių iki labai sudėtingų ir didelių programų:

- Sinuso skaičiavimo programa
- Radianų į laipsnius vertimo
- Bessel funkcijos skaičiavimo
- Dešimtainio logaritmo skaičiavimo programa
- Eksponentinės e^x funkcijos skaičiavimo programa

5.5.1. Sinuso sin α skaičiavimo programa.

Primoji testavimui buvo pasirinkta trigonometrinė sin α. skaičiavimo programa.

- OCL apribojimai:
    - $pre\ \alpha \geq 0\ and\ \alpha \leq 3.14; post{:}\ result \geq 0\ and\ result \leq 1$
- Buvo sugeneruoti 66 mutantai.

- Ciklomatinis sudėtingumas [6]5
- Šakų skaičius 14
- Kodo eilučių kiekis 24

5.5.2. Radianų į laipsnius vertimo programa.

Antroji testavimui buvo pasirinkta funkcija kuri verčia radianus į laipsnius degree(α):

- OCL apribojimai:
  - $pre\ \alpha \geq 3\ and\ \alpha \leq 3.14; post{:}result \geq 170^0\ and\ result \leq 180^0$
- Buvo sugeneruoti 33 mutantai.
- Ciklomatinis sudėtingumas 4
- Šakų skaičius 8
- Kodo eilučių kiekis 14

5.5.3. Bessel funkcijos skaičiavimo programa.

Trečioji testavimui buvo pasirinka Bessel funkcija $x^2\frac{d^2y}{dx^2} + x\frac{dy}{dx} + x^2y = 0$, skaičiuojanti argumentą:

- OCL apribojimai:
  - $pre\ x \geq 0\ and\ x \leq 2.5; post{:}result \geq 0\ and\ result \leq 1$
  - $pre\ x \geq 6\ and\ x \leq 8.5; post{:}result \geq 0\ and\ result \leq 1$
  - $pre\ x \geq 2.5\ and\ x \leq 5.0; post{:}\ result \leq 0$
- Buvo sugeneruoti 145 mutantai.
- Ciklomatinis sudėtingumas 3
- Šakų skaičius 2
- Kodo eilučių kiekis 11

[6] Ciklomatinis sudėtingumas - nepriklausomų kelių programos kodo grafe skaičius

5.5.4. Dešimtainio logaritmo funkcija.

Ketvirtoji testavimui buvo pasirinkta $\lg(x)$

- OCL apribojimai:
  - $pre\ x > 0\ and\ x \leq 1; post{:}result < 0$
  - $pre\ x > 1\ and\ x \leq 2;\ post{:}result < 1$
  - $pre\ x > 2;\ post{:}result > 0.5$
  - $pre\ x = 1\ ; post{:}result = 0$
- Buvo sugeneruoti 421 mutantai.
- Ciklomatinis sudėtingumas 21
- Šakų skaičius 92
- Kodo eilučių kiekis 421

5.5.5. Eksponentinė e^x funkcija.

Penktoji testavimui buvo pasirinkta funkcija e^x

- OCL apribojimai:
  - $pre\ x > 0\ and\ x \leq 1; post{:}result > 1\ and\ result > 2.72$
  - $pre\ x < 0; post{:}result < 1$
- Buvo sugeneruoti 159 mutantai.
- Ciklomatinis sudėtingumas 21
- Šakų skaičius 72
- Kodo eilučių kiekis 345

5 lentelė. Tiriamųjų programų charakteristikų apibendrinimas

Programos pavadinimas	Kodo eilutės	Šakos	Mutantai
Sin	24	14	66
Bessel	11	2	145
Radianų į laipsnius	14	8	33
Log10	421	92	421
e^x	345	72	159

5.6. Eksperimento rezultatai

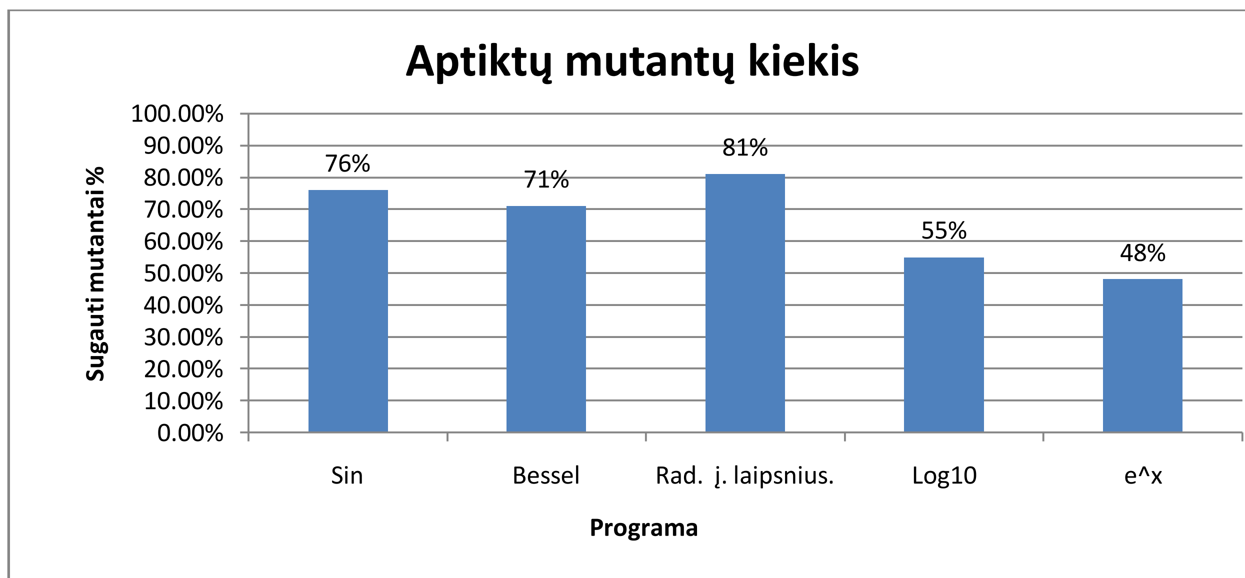
Tyrimo eiga rėmėsi iteraciniu metodu (8 pav.). Buvo atliktos 2 iteracijos.

I. Pirmos iteracija metu buvo ištirtos – ištestuotos bandomos programos ir gautas pradinis rezultatų įvertinimas

II. Antros iteracija metu buvo bandoma patikslinti arba pridėti naujus apribojimus ir ištestuoti, kaip pasikeitė skaičiuojamos tyrimo charakteristikos pakartojus tyrimą.

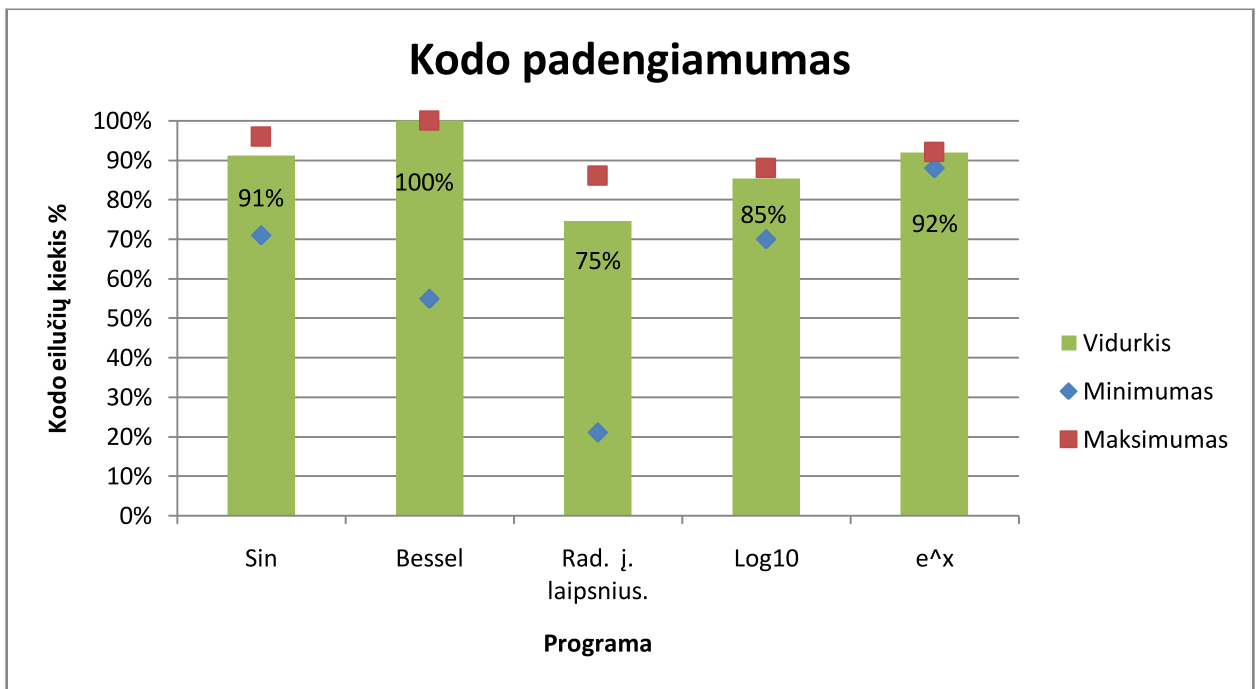
5.6.1. Rezultatai pirmos iteracijos

Vidutinis aptiktų mutantų kiekis yra lygus 66%. Dešimtainio logaritmo skaičiavimo ir konstantos e^x laipsniu kėlimo programa išsiskyrė mažesniu aptiktų mutantų kiekiu. Abi šios programos taip pat išsiskiria savo programinio kodo dydžiu. Kodo eilučių yra daugiau nei 10 kartų daugiau negu likusiose. Reikia toms funkcijoms tikslinti apribojimus.



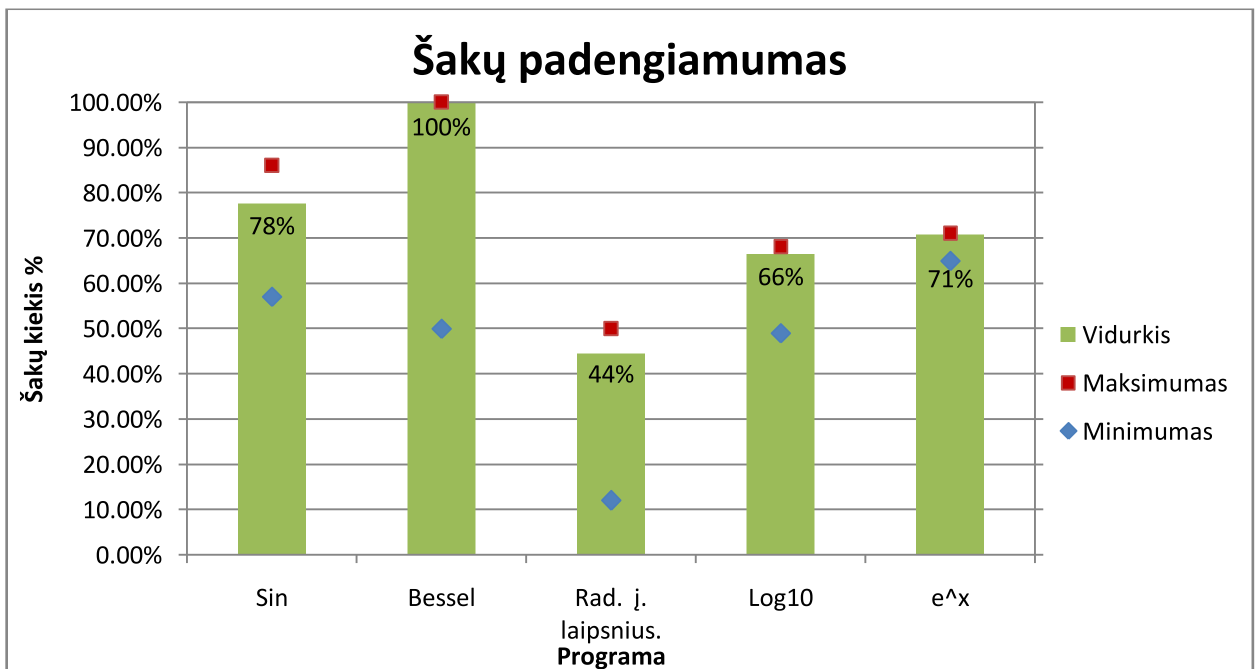
9 pav. Aptiktų mutantų kiekis

Kodo padengiamumo rezultatai yra žymiai geresni. Vidutinis kodo padengiamumas yra lygus 88%. Išsiskiria funkcija radianų į laipsnius vertimo programoje, tačiau kadangi joje yra didžiausias rezultatas sugautų mutantų ir mažas šakų skaičius - 8. Galima teigti, kad įėjimo duomenų aibė yra nepilna ir yra neįvykdoma viena ar kelios šakos, tačiau jose nėra skaičiuojami esminiai funkcijos rezultatai.



10 pav. Kodo padengimo tyrimo rezultatai

Šakų padengiamumas vidutiniškai buvo 72%. Tai yra 16% mažiau negu kodo padengiamumas. Šakos neįvykdomos, jeigu duomenų įėjime nėra atitinkamos reikšmės Radianų į laipsnius vertimo programa išsiskyrė nedideliu šakų padengiamumu. Tai dar labiau patvirtina, kad įėjimo duomenų aibė yra nepilna. Reiktų tobulinti OCL apribojimus.

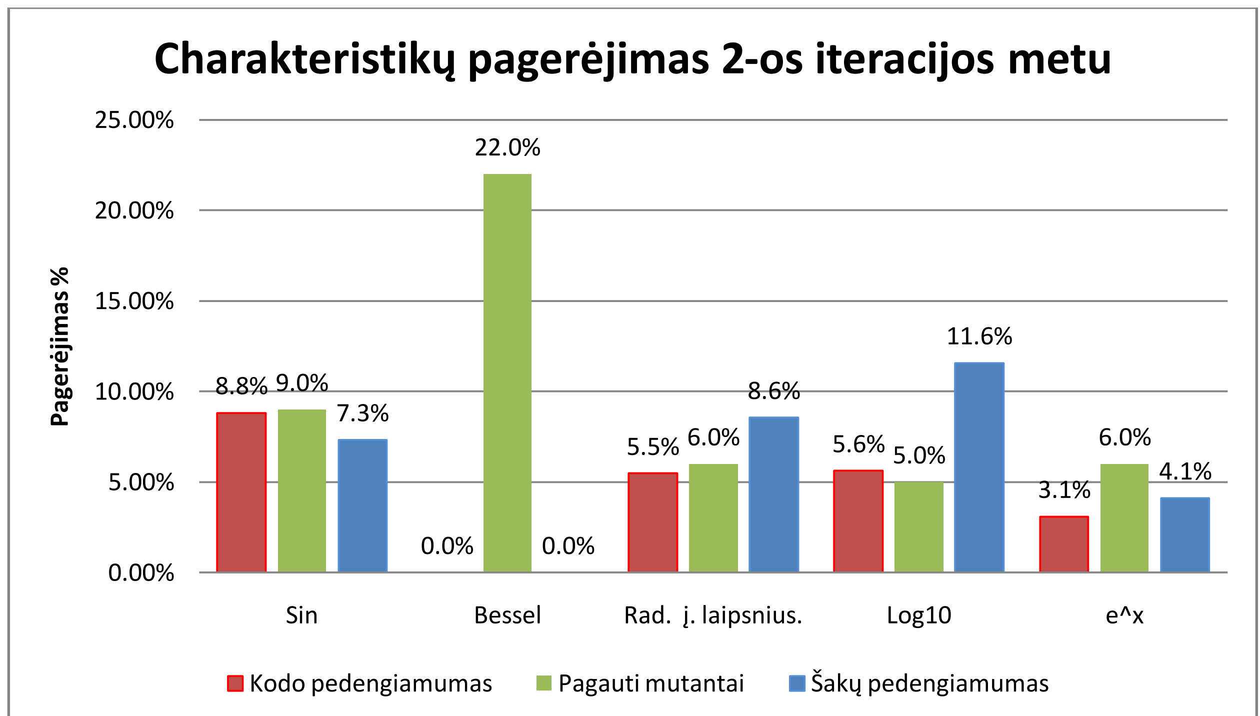


11 pav. Šakų padengimo tyrimo rezultatai

5.6.2. Rezultatai pokyčiai po antros iteracijos

Antros iteracijos metu pavyko pagerinti esamų programų tiriamąsias charakteristikas. Vidutiniškai buvo pagerinta po 7%. Esant dar nepakankamam pagerėjimui reikia bandyti atlikti dar vieną iteraciją.

- Geriausiai pavyko visus 3 parametrus sinuso skaičiavimo funkcijos. $pre\ x < 0\ and\ x > -1{,}57; post: result < 0$
- Bessel funkcijai buvo įdėtas apribojimas, praplečiantis įėjimų aibę skaičiais nuo 8,5 iki 11,5. Rezultatas gražinamas funkcijos turi būti neigiamas.
- Radianų į laipsnius papildyta apribojimu $pre\ \alpha = 0; post: result =\ 0^0$
- Logaritmo skaičiavimui buvo įdėtas tikrinimas $pre\ x > 1000;\ post: result > 3$
- Eksponentinė funkcija papildyta $pre\ x > 10; post: result > 22026$



12 pav. Rezultatų pokyčiai 2-os iteracijos metu

5.7. Išvados

1. Didėjant programos sudėtingumui, reikalingi sudėtingesni, tikslesni OCL apribojimai norint pasiekti aukštus kodo, šakų padengiamu ir aptiktų sugavimo rezultatus:
2. Reikalingas iteracinis gerinimas OCL apribojimų siekiant gerinti rezultatus, norint pasiekti geresnius rezultatus.
3. Nebuvo pasiekti rezultatai 100%, nes:
    a. ne visus apribojimus galima realizuoti esama generatoriaus versija.
    b. nebuvo patikrinti specialūs skaičių atvejai: begalybės, NaN[7] (*angl. Not a Number*)
    c. programa nebuvo idealiai ir pilnai specifikuota OCL apribojimais.
4. Palyginus rezultatus su atsitiktiniu generavimu [42], pagautų mutantų kiekis yra žymiai geresnis. Atsitiktinis generavimas pasiekė apie 40%, su apribojimais eksperimente pasiektas apie 73% (antroje iteracijoje). Tai yra bent 30% geresni rezultatai (žiūr. 2.10.1 skyrelį).

[7] NaN – slankiojo kabelio skaičiaus neapibrėžta reikšmė.

6. IŠVADOS

1. Šiame darbe buvo sukurtas vienetų testų generatorius, kuris generavo vienetų testus remdamasis OCL apribojimais. Generatoriaus efektyvumas buvo įrodytas eksperimentiškai, remiantis mutaciniu testavimu. Buvo pasiekti tokie tyrimo rezultatai pirmosios iteracijos metu:
   - Virš 65 % aptiktų mutantų.
   - Apie 85 % dodo padengiamumas.
   - Apie 72% šakų padengiamumas.
2. Vienetų testų generavimui buvo pritaikytas iteracinis testavimo metodas. Jo metu yra vis tikslinami esami apribojimai ir/ arba pridedami nauji apribojimai (8 pav.). Tokiu iteraciniu būdu galima:
   a. Gerinti apribojimus iki užsibrėžtų tikslų.
   b. Atrasti kode esančias klaidas
   c. Surasti vykdymui nepasiekiamas kodo dalis.

   Tyrimo metu antros iteracijos metu pasiektas vidutinis 7% pagerėjimas.
3. Eksperimente buvo panaudotos realios programos, tai įrodo kad galima įrankį panaudoti realiuose testavimo projektuose.

7. LITERATŪRA

[1] **LAURIE, W.; GUNNAS, K.; NACHIAPPAN N.** *On the Effectivennes of Unit Test Automation at Microsoft.* 2009 m., psl. 81 - 87.

[2] **ZHI, Q., Z.; ARNALDO, S.; LEI, Z.; WILLY S.; KAI-YUAN, C.** *Improving Software Testing Cost-Effectiveness Through Dynamic Partitioning.***.** 2009 m., psl. 249 - 258.

[3] **AVNER, E.; SHALOM, S.** *Measuring and optimizing systems' quality costs and project duration.* 2006. psl. 259–280.

[4] **GARRETT, T.** *Implementing Automated Software Testing - Continuously Track Progress and Adjust Accordingly.* 2009 m.

[5] *ISO9126 Information Technology Software Product Evaluation Quality* . Geneva : International Organisation for Standardization, 1992.

[6] **GARVIN, D.** *What Does "Product Quality" Really Mean? Sloan Management Review, Fall.* 1984. psl. 25-45.

[7] **BECK, K.** *Extreme programing.* [Tinkle] [Cituota: 2011 m. 05 22 d.] http://www.extremeprogramming.org.

[8] **ASTELS, D.** *Test Driven development: A Practical Guide.* 2003. ISBN:0131016490.

[9] **JEAN, F., M.; MICHAEL G., H.** *Understanding formal methods.* s.l. : 2003.

[10] **MILLER, J.; MUKERJI, J.** *Model Driven Architecture (MDA) A Draft with annotations of issues to resolve Architecture Board ORMSC* 2001

[11] **HETZEL, W., C.** *The Complete Guide to Software Testing, 2nd ed.* 1988. psl. 280.

[12] **WHITTAKER, J., A.** *What is software testing? And why is it so hard?* s.l. : IEEE, 2000. psl. 70-79.

[13] **IEEE.** IEEE Standart Glossary of Software ENgineering Terminology. 1990.

[14] **KOLAWA, A.; HUIZINGA, D.** *Automated Defect Prevention: Best Practices in Software Management.* 2007 : Wiley-IEEE Computer Society, T. ISBN 0470042125.

[15] **CHIH-WEI H.; MICHAEL, J.; JOHNSON, L., W.; MAXIMILIEN M.** *On Agile Performance Requirements Specification and Testing.* 2006 m., psl. 47-52.

[16] **WILLIAMS, E., M.; MAXIMILIEN L.** *Assessing Test-driven Development at IBM.* 2003 m., psl. 564-569.

[17] **NAGAPPAN, E., M.; MAXIMILIEN, T.; BHAT; WILLIAMS, L.** Realizing Quality Improvement Through Test Driven Development: Results and Experiences of Four Industrial Teams. *Empirical Software Engineering.* 2008 m., psl. 289-302.

[18] **SANCHEZ, J.; WILLIAMS, L.; MAXIMILIEN, M.** *A Longitudinal Study of the Test-driven Development Practice in Industry.* 2007. psl. 5-14.

[19] **WILLIAMS, E., L.; MAXIMILIEN, M.; VOUK., M.** *Test-Driven Development as a Defect-Reduction Practice.* 2003. psl. 34-45.

[20] **CRAMBLITT, B.** *Alberto Savoia sings the praises of software testing.* 2007.

[21] **GLENFORD, M.** *The Art of Software Testing.* 2004. ISBN 978-0471469124.

[22] **WARMER, J.; KEPPLE, A.** *The Object Constraint Language: precise modeling with UML.* 1998.

[23] **COOK, S.; DANIELS, J.** *Disigning object systems.* 1994.

[24] **DEPHNAT, N., C., ; GARIS, A.; RIESCO, D.; MONTEJANO, G.** *Defining OCL constraint for the Proxy Design Profile.* Winoma : IEEE, 2007 m.

[25] **PACKEVICIUS, Š.** *Unit Tests Generation Using Software Models And Imprecise Constraints.* Doktoranto disertacija. Kauno Technologijų Universitetas 2009.

[26] **YOOSIK, C.; AVILA, C.** *Automating Java Program Testing Using OCL and AspectJ.* 2010 m., psl. 1020-1025.

[27] **WEIBLEDER, S.** *Quality of Automatically Generated Test Cases based om OCL Expressions.***.** 2008

[28] **BERTRAND, M.** *Object oriented software construction.* 1997.

[29] **HOWDEN, W. E.; YUDONG H.; WIELAND, B.** *Proving properties of programs from program traces.* 1978 m.

[30] **MARTIN, R.; RIEHLE, D.; BUSCHMANN, F.** *Patter languages of program disign* 1998.

[31] **YONG, L.; ANDREWS, J., H.** *Minimization of Randomized Unit Test Cases.* Ontario : 2005 m.

[32] **XIE, T.** *Improving Automation in Developer Testing: State of the Practice.*2009.

[33] **BERTRAND, M.** *Applaying "Design by Contract".* 1992.

[34] **MITĖ, M.** *Programos elgsenos testavimo posistemė.* Tarpuniversitetinės magistrų ir doktorantų mokslinės konferencijos „Informacinės Technologijos“, 2009-05-08.

[35] **RUNENSON, P.** *A Survey of Unit Testing Practice.* 2006 m.

[36] **HAMIE, A.** *On the Relationship between The Object Constraint Language (OCL) and The Java Modeling Language (JML) .* 2006.

[37] **MORELL, L., J.** *A Theory of Fault-Based Testing.* 1990 m. psl. 844-857.

[38] **AICHENIG, B., K.** *Test Case Generation by OCL and Constraint Solving.* 2005 ISBN: 0-7695-2472-9

[39] **OFFUT, A., J.** *A Practical System for Mutation Testing: Help for the Common Programmer.* 1994 ISBN:0-7803-2103-0

[40] **UNTCH, A., J.; OFFUTT, R.** *Mutation Testing in the Twentieth and the Twenty First.* 2000 m., psl. 45 -55.

[41] **HONG, Z.; PATRICK, A., V.; .** *Software Unit Test Coverage and Adequacy.* 1997 m., psl. 366-427. ISSN 0360-0300.

[42] **SHUANG, W.; JAFF, O.** *Comparison of Unit-Level Automated Test Generation Tools.* 2009 m., psl. 210-217.

8. TERMINŲ IR SANTRUMPŲ ŽODYNAS

UML (*angl. Unified Modeling Language*) –Vieninga modeliavimo kalba. Tai modeliavimo ir specifikacijų kūrimo kalba, skirta specifikuoti, atvaizduoti ir konstruoti objektiškai orientuotų program

- OCL (*angl. Object Constraint Language*) – formalioji apribojimų kalba, skirta išplėsti UML modelį, jam užrašant apribojimus.
- DbC (*angl. Design by Contract*) – tai programavimo būdas, kai programa yra specifikuojama formaliai: yra aprašomi invariantai, prieš ir po sąlygos
- Unit Test – vienetų testavimas
- Junit – vienetų testų karkasas, skirtas JAVA vienetų testams.
- JAVA – objektiškai orientuota programavimo kalba

Pav. – paveikslėlis;

- JAVACC (*angl. Java Compiler Compiler*) – EBNF gramatikos skaitytuvas
- EBNF (Extended Backus–Naur Form) - konteksto gramatika, skirta specializavimui formalioms kalboms.

9. PRIEDAI

9.1. Straipsnis „Programos elgsenos testavimo posistemė“.

Straipsnis yra publikuotas 14-osios tarp universitetinės magistrų ir doktorantų mokslinės konferencijos „Informacinės Technologijos“, 2009-05-08.

Programos elgsenos testavimo posisteme

Mantas Mitė

Kauno technologijos universitetas, Programavimo inžinerijos katedra.

Straipsnyje nagrinėsime UML (Unified Model Language) būsenų (state), bendravimo (collaboration) diagramas, sekų (sequence) t.y. jų išanalizavimą ir pateikimą atskirai sistemai (konkrečiu mano atveju tai testavimo sistemai) lengvai suprantamu būdu.

1. Įvadas

UML meta modeliu galima aprašyti ir specifikuoti programas. Programos dinamiką galima aprašyti būsenų ir bendravimo diagramomis ir jas galima panaudoti ne vien tik dėlto, kad būtų galima sužinoti kaip veikia programos dinamika, bet ir kitokiems tikslams testavimui. UML modelis neneša visos informacijos apie programą, tai modelis yra dar papildytas apribojimais, kurie yra aprašyti OCL (Object Constrain Language). Diagramos yra nupiešiamos specialias UML modeliavimo įrankiais. Taigi iškyla problema, kaip mums tą metamodelį suprasti, kaip iš jo paimti mus dominančia informaciją, ją išanalizuoti ir perteikti naudojimui rezultatus nebe nebekomplikuotu paprastu formatu. Bendravimo ir būsenų diagramas bendru atveju galime suprasti kaip grafą. Taigi bandysime sudaryti kelius ar jų rinkinius, kaip būtų galima pereiti grafą. Programos kodas yra visiškai nereikalingas ir todėl galima pvz. Sugeneruoti testus dar nesamai programinei įrangai.

2. Problemos

Būsenų ir bendradarbiavimo, sekų diagramos savyje turi informacija apie dinamiškumą programos ir tas nėra pakankama informacija analizei. Sekų ir bendradarbiavimo diagramos savyje kaupia beveik dalinai identiška informaciją ir jų išskyrimas esminio skirtumo neturi. Reikalinga ir statinė informacija apie metamodelį, ta informacija yra pasiimama iš klasių diagramos, kuri turi visą statinę informaciją. Pats UML metamodelis yra sukuriamas specializuota braižymo programa, kuri turi bent dalinį palaikymą OCL apribojimams ir

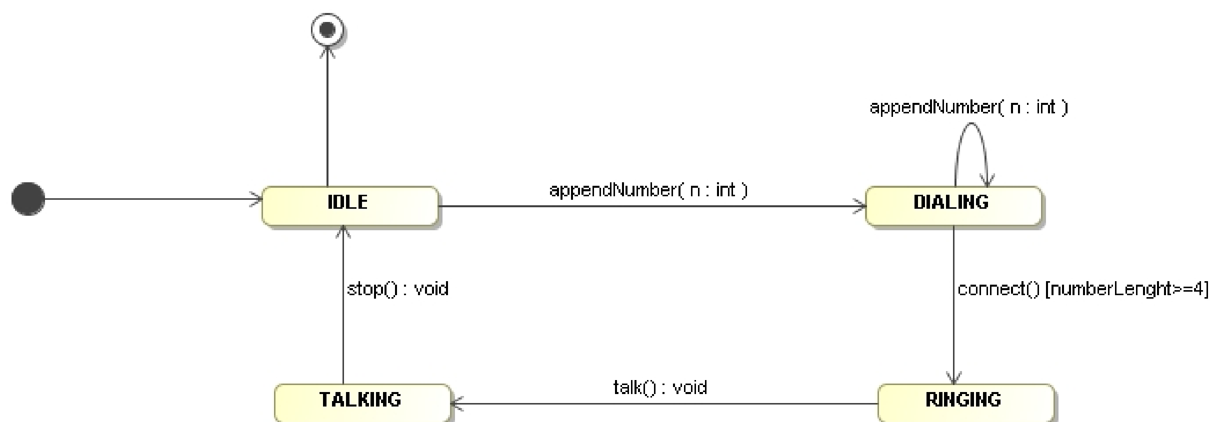
išsaugoma XMI (XML Metadata Interchange) formatu XML faile, kuris yra OMG standartas, tačiau skirtingi įrankiai informaciją saugo nevisai neidentiškai, tad reikia kiekvienam norimam naudoti įrankiui sukurti adapterį, kuris tuos skirtumus žinotų ir apeitų. Arba prisirišti prie konkretaus įrankio. Perėjimai taip grafo viršūnių nėra laisvi, jie yra apriboti OCL, tad perėjimui reikia parinkti konkrečius parametrus ar kita informaciją.

3. Analizė

Pradžioje turime sukurtą kažkieno XMI failą

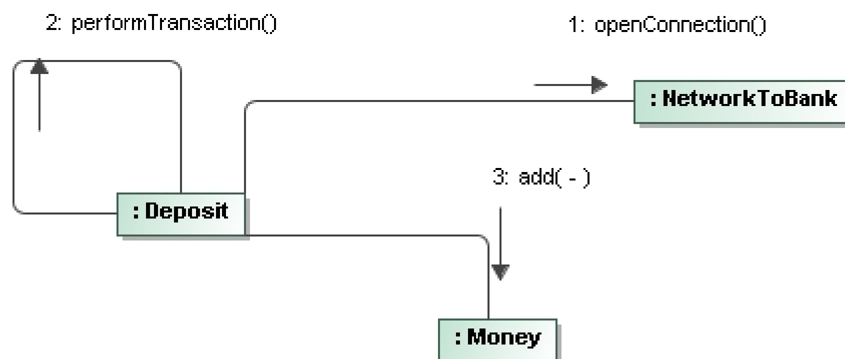
1) Nuskaitome ir išsisaugojama meta modelio informacija, vidiniame formate
2) Informacija yra išanalizuojame ir paruošiama naudojimui.
3) Palei pateiktus reikalavimus yra sugeneruojami ir po to gražinami keliai.

Būsenų diagramoje yra nurodytos kokiose būsenose gali būti programa ir kaip į tas būsenas pereiti. Prima reikia išspręsti nuo kur pradėti pereiti diagrama, jeigu vartotojas nenurodo kitaip, tai pradedama nuo pradinės standartinės būsenos. Užbaigimas ir perėjimo irgi gana komplikuotas, nes iškyla klausimai: ar reikia pereiti iki visas būsenas, kiek kartų pabuvoti būsenoje , kada sustoti. Duotame trivialiame pavyzdėlyje iškyla vienos iš esminių problemų. Kaip neužsiciklinti grafike (pvz. DIALING būsenos galima pereiti į save nors ir amžina), kita problema ne į visas būsenas galiam pereiti, yra tam tikri apribojimai (guard UML modeliuose), kad ją būtų galima suprasti jie turi būti užrašyti OCL.



1 paveikslas. Busenų diagram.

Bendradarbiavimo diagramose nėra būsenų, tik nusakytos objektų klasės ir pranešimai. Tai esminiai principai nuo bendradarbiavimo diagramos nėra ir problemos, ir sprendimai išlieka tokie patys.



2 paveikslas. Bendradarbiavimo diagrama.

4. Sprendimas

Siūlomas sprendimas yra vartotojui leisti pasirinkti, kokio tipo kelią sugeneruoti ir vartotojui visa informacija butų pateikiama, klasėmis, metodais, kintamaisiais ir t.t. ,nes meta modelyje tiesiogiai ta informacija nėra pasiekiama, reikalingi papildomi veiksmai, taip pat butų nurodomas apribojimai, su kokias galima eiti atitinkamus parėjimus ar kelius

Kelių tipai gali būti parenkami tokie, jei

1) Pereiti tiktai vieną kartą visas būsenas arba objektą (priklauso nuo diagramos).
   Būtų stengiamasi apeiti visas būsenas (objektus) tik tai vieną kartą.
2) Pereiti visus perėjimus.
   Būtų stengiamasi į kelią įtraukti visus perėjimas.
3) Atsitiktinis perėjimas.
   Būtų sudaromas atsitiktinis kelias.
4) Fiksuotas ciklo perėjimo kartų.
   Ciklai butų sukami tik fiksuota kartų skaičių, o kelias sudaromas palei vieną iš pirmų trijų būdų
5) Fiksuotas programos parametras(ai).
   Programoje būtų užfiksuojamas parametras(ai) ir būtų sudarinėjamas kelias, kadangi kai kurie parametrai taptų fiksuoti, tai perėjimai taptų negalimi, o kelius būtų galima formuoti pelei vieną iš pirmų trijų būdų.

5. Rezultatų pavyzdžiai

Pateiksime rezultatus pirmojo paveikslėlio, jeigu juos paduotume į programą palei visus penkis variantus rezultatų. Diagrama yra Phone klasėje, yra Initial būsena, kai sukuriamas objektas, ir Final būsena, kai sunaikinamas objektas, pradedama iš Initial būsenos.

1) appenNumber(int n) -> connect() [numberLenght>4]-> talk() –> stop()
2) appenNumber(int n) -> appenNumber(int n) -> connect() [numberLenght>4]-> talk() –> stop()
3) appenNumber(int n) -> connect() [numberLenght>4]-> talk() –> stop()
4) Jeigu ciklus pereisime du kartus. appenNumber(int n) -> appenNumber(int n) -> appenNumber(int n) -> connect() [numberLenght>4]-> talk() –> stop().
5) Jeigu n=2. appenNumber(int n)-> appenNumber(int n).

6. Išvados

Straipsnyje siūlomas būdas, kaip būtų galima išnaudoti UML modelyje esančių būsenų, bendradarbiavimo (sekų) diagramų informaciją: programos veikimą modeliuoti tam tikru vienu ar keliais įskaitant visus variantu, jei tokie galimi, programos veikimo būdais.

9.2. Literatūros sąrašas

[1]Shaukat Ali, Lionel C. Briand, Muhammad Jaffar-ur Rehman, Hajra Asghar,Muhammad Zohaib Z. Iqbal, Aamer Nadeem “A State-based Approach to Integration Testing based on UML Models” 2006

[2]Jorgenson Paul, Software testing: a craftsman’s approach 3rd ed. 2008