**KAUNO TECHNOLOGIJOS UNIVERSITETAS**

**INFORMATIKOS FAKULTETAS**

**Vidmantas Lukšas**


# AUTOMATINIŲ TESTŲ KŪRIMO ĮRANKIO NAUDOJANČIO MODELIU PAREMTĄ TESTAVIMĄ KŪRIMAS IR TYRIMAS

Baigiamasis magistro projektas

**Vadovas**
dr. Šarūnas Packevičius

**KAUNAS, 2017**

**KAUNO TECHNOLOGIJOS UNIVERSITETAS**
**INFORMATIKOS FAKULTETAS**

# AUTOMATINIŲ TESTŲ KŪRIMO ĮRANKIO NAUDOJANČIO MODELIU PAREMTĄ TESTAVIMĄ KŪRIMAS IR TYRIMAS

Baigiamasis magistro projektas
**Programų sistemų inžinerija (621E16001)**

**Vadovas**
(parašas) dr. Šarūnas Packevičius
(data)

**Recenzentas**
(parašas) dr. Dominykas Barisas
(data)

**Projektą atliko**
(parašas) Vidmantas Lukšas
(data)

**KAUNAS, 2017**

# Turinys

1.ĮVADAS....2
1.1.Dokumento paskirtis....2
1.2.Problema....2
1.3.Tikslas ir uždaviniai....2
1.4.Santrauka....2
2.ANALITINĖ DALIS....3
2.1.Egzistuojantys testų generavimo sprendimai....3
2.1.1.Simboliniu vykdymu paremtas generavimas....4
2.1.2.Paieška paremtas testavimas....8
2.1.3.Kombinatorinis testavimas....9
2.1.4.Atsitiktinis generavimas ir prisitaikantis atsitiktinis generavimas....10
2.1.5.Modeliu pagrįstas testavimas....13
2.2.Testų generavimo įrankių savybių kiekybinis ir/ arba kokybinis palyginimas....15
2.2.1.Įrankių apžvalga....15
2.2.1.1.Randoop....15
2.2.1.2.AETG....16
2.2.1.3.EvoSuite....17
2.2.1.4.Spec Explorer....17
2.2.1.5.KLEE....17
2.2.1.6.Smartesting CertifyIt....17
2.2.1.7.T3i....18
2.2.1.8.Jalangi....18
2.2.1.9.RT-Tester....19
2.2.1.10.JTest....19
2.2.1.11.JTExpert....19
2.2.2.Palyginimo kriterijai....20
2.2.3.Palyginimas....20
3.PROJEKTINĖ DALIS....22
3.1.Architektūros pateikimas ....22
3.2.Architektūros tikslai ir apribojimai....23
3.3.Panaudojimo atvejų vaizdas....23
3.3.1.Sistemos sudėtis....23
3.3.1.1.Sistemos ribos (panaudojimo atvejų modelis)....23
3.3.1.2.Panaudojimo atvejai....25
3.4.Sistemos statinis vaizdas....28
3.4.1.Apžvalga....28
3.4.2.Paketų detalizavimas....28
3.4.2.1.Vartotojo sąsaja....29
3.4.2.2.Testų generatorius....29
3.4.2.3.SUT analizatorius....30
3.4.2.4.Testų vykdymas....30
3.5.Sistemos dinaminis vaizdas....31
3.5.1.Testų generavimas....32
3.5.2.OCL apribojimų įvedimas....39
3.5.3.OCL apribojimų modifikavimas....40
3.5.4.Sugeneruotų testų peržiūra....41
3.5.5.Sugeneruotų testų modifikavimas....41
3.5.6.Testų paleidimas....42
3.5.7.Kodo padengimo matavimas....43
3.5.8.Veiklos ir būsenų diagramos....44

3.6.Išdėstymo (deployment) vaizdas....................45
3.7.Kokybė....................46
4.TYRIMO DALIS....................46
4.1.Tyrimo aprašas....................46
4.1.1.Tiriami algoritmai....................46
4.1.2.Tiriamos charakteristikos....................47
4.1.3.Tyrimui naudojama programinė įranga....................47
4.1.4.Tyrimo vykdymo aplinka....................47
4.1.5.Tyrime testuojama programinė įranga....................48
4.1.6.Tyrimo eiga....................48
4.2. Tyrimo rezultatai....................50
4.2.1.Kodo padengimo tyrimas....................50
4.2.1.1.Basic Calculator....................50
4.2.1.2.LT Converter....................50
4.2.1.3.Geometry Calculator....................51
4.2.1.4.CL Tic Tac Toe....................52
4.2.2.Rezultatų apibendrinimas....................52
4.2.3.Siūlomi patobulinimai....................53
5.EKSPERIMENTINĖ DALIS....................54
5.1.Eksperimento aprašas....................54
5.1.1.Tiriami algoritmai....................54
5.1.2.Eksperimento eiga....................54
5.2.Eksperimento rezultatai....................55
5.2.1.Basic Calculator....................55
5.2.2.LT Converter....................56
5.2.3.Geometry Calculator....................58
5.2.4.CL Tic Tac Toe....................59
5.2.5.Eksperimento rezultatų analizė ir išvados....................60
6.IŠVADOS....................63
7.LITERATŪRA....................64
8.TERMINŲ IR SANTRUMPŲ ŽODYNAS....................68
9.PRIEDAI....................70
9.1.OCL apribojimai naudoti eksperimente....................70
9.1.1.Basic Calculator....................70
9.1.2.LT Converter....................70
9.1.3.Geometry Calculator....................71
9.1.4.CL Tic Tac Toe....................72

## Paveikslėlių sąrašas

1 pav. Pavyzdinė funkcija....................5
2 pav. Simbolinio vykdymo medis....................6
3 pav. Taškinis klaidą sukeliančių įvesčių tipas....................11
4 pav. Juostinis klaidą sukeliančių įvesčių tipas....................11
5 pav.Blokinis klaidą sukeliančių įvesčių tipas....................12
6 pav. Modeliu paremto testavimo procesas pagal M. Utting'o, A. Pretschner'io ir B. Legeard'o straipsnį„A taxonomy of model-based testing approaches“....................14
7 pav. Panaudojimo atvejų diagrama....................24
8 pav. Paketų diagrama....................28
9 pav. Paketas „Vartotojo sąsaja“: klasių diagrama....................29
10 pav. Paketas „Testų generatorius“: klasių diagrama....................30
11 pav. Paketas „SUT analizatorius“: klasių diagrama....................30
12 pav. Paketas „Testų vykdymas“: klasių diagrama....................31
13 pav. Sekų diagrama: Parinkčių lango atvėrimas....................32
14 pav. Sekų diagrama: Taisyklės pasirinkimas....................33
15 pav. Sekų diagrama: Testų generavimas – sąveika su UI....................34
16 pav. Sekų diagrama: Testų generavimas – SUT analizė....................35
17 pav. Sekų diagrama: Testų generavimas – generavimo procesas....................36
18 pav. Sekų diagrama: Testų generavimas metodui....................37
19 pav. Bendradarbiavimo diagrama: Testų generavimas metodui....................38
20 pav. Sekų diagrama: OCL apribojimų įvedimas....................39
21 pav. Sekų diagrama: OCL apribojimų modifikavimas....................40
22 pav. Sekų diagrama: Sugeneruotų testų peržiūra....................41
23 pav. Sekų diagrama: Sugeneruotų testų modifikavimas....................41
24 pav. Sekų diagrama: Sugeneruotų testų paleidimas....................42
25 pav. Sekų diagrama: Kodo padengimo matavimas....................43
26 pav. Veiklos diagrama....................44
27 pav. Būsenų diagrama....................45
28 pav. Sistemos išdėstymo diagrama....................46
29 pav. Vidutinis kodo padengimas....................61
30 pav. Algoritmų patobulinimo rezultatai....................62

## Lentelių sąrašas

1 lentelė. PC sprendiniai.....7
2 lentelė. Kombinatorinis testavimas (pavyzdys).....9
3 lentelė. Modeliu paremto testų generavimo metodų klasifikatoriai.....14
4 lentelė. Įrankių palyginimas (1) (pagal generavimo metodą).....20
5 lentelė. Įrankių palyginimas (2) įvesties duomenys.....21
6 lentelė. Įrankių palyginimas (3).....21
7 lentelė. Architektūriniai vaizdai ir juos sudarantys modeliavimo elementai.....22
8 lentelė. Panaudos atvejis: generuoti testus.....25
9 lentelė. Panaudos atvejis: įvesti OCL apribojimus.....26
10 lentelė. Panaudos atvejis: redaguoti OCL apribojimus.....26
11 lentelė. Panaudos atvejis: peržiūrėti sugeneruotus testus.....26
12 lentelė. Panaudos atvejis: redaguoti sugeneruotus testus.....27
13 lentelė. Panaudos atvejis: paleisti sugeneruotus testus.....27
14 lentelė. Panaudos atvejis: peržiūrėti kodo padengimą.....27
15 lentelė. Tiriamos charakteristikos.....47
16 lentelė. Testuojama programinė įranga.....48
17 lentelė. „Basic Calculator“ programos kodo padengimas.....50
18 lentelė. „LT Converter“ programos kodo padengimas.....51
19 lentelė. „Geometry Calculator“ programos kodo padengimas.....51
20 lentelė. „CL Tic Tac Toe“ programos kodo padengimas.....52
21 lentelė. Eksperimentas: „Basic Calculator“ padengimas (atsitiktinis generavimas ir OCL paremtas generavimas).....55
22 lentelė. Eksperimentas: „Basic Calculator“ padengimas („Hill Climbing“ ir „Simulated Annealing“).....55
23 lentelė. Eksperimentas: „LT Converter“ padengimas (atsitiktinis generavimas ir OCL paremtas generavimas).....56
24 lentelė. Eksperimentas: „LT Converter“ padengimas („Hill Climbing“ ir „Simulated Annealing“).....57
25 lentelė. Eksperimentas: „Geometry Calculator“ padengimas (atsitiktinis generavimas ir OCL paremtas generavimas).....58
26 lentelė. Eksperimentas: „Geometry Calculator“ padengimas („Hill Climbing“ ir „Simulated Annealing“).....58
27 lentelė. Eksperimentas: „CL Tic Tac Toe“ padengimas (atsitiktinis generavimas ir OCL paremtas generavimas).....59
28 lentelė. Eksperimentas: „CL Tic Tac Toe“ padengimas („Hill Climbing“ ir „Simulated Annealing“).....60

KAUNO TECHNOLOGIJOS UNIVERSITETAS

Informatikos fakultetas

(Fakultetas)

Vidmantas Lukšas

(Studento vardas, pavardė)

Programų sistemų inžinerija (621E16001)

(Studijų programos pavadinimas, kodas)

Baigiamojo projekto „Automatinių testų kūrimo įrankio naudojančio modeliu paremtą testavimą kūrimas ir tyrimas“

**AKADEMINIO SĄŽININGUMO DEKLARACIJA**

20 17 m. gegužės 25 d.

Kaunas

Patvirtinu, kad mano, **Vidmanto Lukšo**, baigiamasis projektas tema „Automatinių testų kūrimo įrankio naudojančio modeliu paremtą testavimą kūrimas ir tyrimas“ yra parašytas visiškai savarankiškai, o visi pateikti duomenys ar tyrimų rezultatai yra teisingi ir gauti sąžiningai. Šiame darbe nei viena dalis nėra plagijuota nuo jokių spausdintinių ar internetinių šaltinių, visos kitų šaltinių tiesioginės ir netiesioginės citatos nurodytos literatūros nuorodose. Įstatymų nenumatytų piniginių sumų už šį darbą niekam nesu mokėjęs.

Aš suprantu, kad išaiškėjus nesąžiningumo faktui, man bus taikomos nuobaudos, remiantis Kauno technologijos universitete galiojančia tvarka.

_______________________________ ________________

*(vardą ir pavardę įrašyti ranka)* *(parašas)*

Lukšas, Vidmantas. *Model Based Tests Generation Algoritmhs Reasearch and Experimetal Tool Development*: Master‘s thesis in software engineering / supervisor dr. Šarūnas Packevičius. The Faculty of Informatics, Kaunas University of Technology.

Research area and field: Technological sciences, software engineering

Key words: Model-based testing, software testing, automatic test generation

**Kaunas, 2017. 81 p.**


**SUMMARY**

The volume and speed of software production is increasing rapidly. So does the complexity of the systems being created. As the number of software systems and their complexity rises, an increase of work required in the field of quality assurance is also observed. One of the core QA activities is testing. Performing testing of complex systems manually is not cost-effective and requires extensive amounts of time. As a result, usage of automated tests becomes more and more widespread. Unfortunately, writing tests also requires a lot of time. Consequently, the need for automatic test generation rises.

Nowadays, there is a large amount of test generation algorithms to choose from: symbolic execution, search-based algorithms (e.g. Hill Climbing), genetic algorithms, random generation, model-based generation, etc.

The thesis focuses on the area of model-based generation. As a result of this research, an OCL constraints based test generation tool is designed. The tool is implemented as an "Eclipse" IDE plug-in, using Java programming language.

The tool is then examined to discover what code coverage can the tests it generates achieve on different kinds of software. A set of recommendations for improving the tool, based on the results of aforementioned testing, are provided. After implementing the recommended features, an experiment is carried out to investigate whether the recommendations allowed to improve the tool's performance compared to the initial version.

# 1. ĮVADAS

## 1.1. Dokumento paskirtis

Dokumentas yra magistrantūros studijų programos „Programų sistemų inžinerija“ baigiamasis darbas. Jis skirtas apibendrinti studijų metu atliktą tiriamąjį bei projektinį darbą.

Dokumente pateikiama programinės įrangos testų generavimo dalykinės srities analizė, aprašomas pasirinktas sprendimas, pateikiama esminė darbo metu sukurto įrankio projektavimo dokumentacija bei atlikto eksperimentinio tyrimo rezultatai.

Raktiniai žodžiai: Modeliu paremtas testavimas, programinės įrangos testavimas, automatinis testų generavimas

## 1.2. Problema

Darbe nagrinėjama automatinio programinės įrangos testų generavimo problematika, siekiama ištirti OCL modeliu pagrįsto generavimo efektyvumą, privalumus ir trūkumus lyginant su kitais metodais.

## 1.3. Tikslas ir uždaviniai

Darbo tikslas – sukurti OCL apribojimais pagrįstą automatinio testų generavimo įrankį bei ištirti juo naudojantis sukurtų testų kokybę. Siekiant įgyvendinti užsibrėžtą tikslą, iškelti tokie uždaviniai:

1. Atlikti testų generavimo srities analizę;
2. Atlikti rinkoje esančių spendimų analizę;
3. Suprojektuoti bei sukurti modeliu paremtą testų generavimo įrankį;
4. Ištirti sukurto įrankio generuojamų testų charakteristikas;
5. Pateikti tyrimo metu nustatytus galimus patobulinimus bei juos realizuoti;
6. Atlikti eksperimentą, siekiant nustatyti ar atlikti patobulinimai davė lauktą rezultatą;

## 1.4. Santrauka

Programinės įrangos kūrimo tempai ir apimtys nuolat auga. Taip pat didėja ir kuriamų sistemų sudėtingumas. Augant sistemų skaičiui ir sudėtingumui, didėja ir darbo apimtys reikalingos jų kokybės užtikrinimui. Viena pagrindinių kokybės užtikrinimo veiklų – testavimas. Testuoti sudėtingas sistemas rankiniu būdu yra neefektyvu tiek laiko, tiek kokybės atžvilgiu, todėl yra būtinas testavimo automatizavimas. Tačiau testų rašymas taip pat yra nemažai laiko reikalaujantis

procesas. Dėl šios priežasties atsiranda poreikis bent dalį testų sugeneruoti automatiškai.

Egzistuoja didelė testavimo algoritmų aibė: simbolinis vykdymas, paieška paremti algoritmai (pvz. „Hill Climbing“), genetiniai algoritmai, atsitiktinis generavimas, modeliu paremtas generavimas ir kt.

Darbo metu gilinamasi į modeliu paremto testavimo sritį bei suprojektuojamas įrankis, kuris remiasi OCL apribojimais testų generavimui. Šis įrankis realizuojamas Java kalba kaip „Eclipse“ programavimo aplinkos įskiepis.

Taip pat atliekamas tyrimas, kurio metu nustatomi projekto metu sukurto įrankio generuojamų testų kodo padengimo rodikliai bei pateikiamos gairės tolesniam įrankio vystymui. Šių pakeitimų efektyvumas įvertinamas atliekant eksperimentinius generuojamų testų pasiekiamo kodo padengimo matavimus ir lyginant jų metu gautus rezultatus su pradinės įrankio versijos rezultatais.

# 2. ANALITINĖ DALIS

Programinės įrangos kiekis, sudėtingumas ir svarba nuolat auga. Kadangi programinė įranga atlieka vis daugiau ir svarbesnių užduočių, didėja jos kokybės užtikrinimo poreikis. Testavimas – viena pagrindinių kokybės užtikrinimo priemonių [18]. Didėjančios šios veiklos apimtys kelia rimtą problemą, nes testavimas yra pakankamai brangi ir daug laiko reikalaujanti veikla (pastebėta, kad testavimo veikloms skiriama beveik 50 % visų programinės įrangos kūrimo resursų) [25]. Dėl šios priežasties, yra prasminga sumažinti testavimui skiriamų resursų kiekį automatizuojant procesą [2].

## 2.1. Egzistuojantys testų generavimo sprendimai

Testinių atvejų (duomenų) generavimas yra viena iš sudėtingiausių, bet ir svarbiausių testavimo veiklų, nes daro didelę įtaką viso testavimo proceso efektyvumui. Dėl šios priežasties, per kelis pastaruosius dešimtmečius buvo atliekamas didelės apimties mokslinis darbas testinių atvejų generavimo srityje. Šio darbo metu buvo sukurta aibė įvairių testų generavimo strategijų. Šiame skyrelyje aptarsime penkias labiausiai paplitusias technikas: [2]:

1. Simboliniu vykdymu paremtas generavimas;
2. Paieška paremtas testavimas;
3. Kombinatorinis testavimas;
4. Atsitiktinis generavimas ir prisitaikantis atsitiktinis generavimas;
5. Modeliu pagrįstas testavimas;

### 2.1.1. Simboliniu vykdymu paremtas generavimas

Simbolinis vykdymas yra apibrėžiamas kaip programos vykdymas, pakeičiant realias įvesties

reikšmes, simboliais. Pats vykdymo procesas nesiskiria nuo normalaus, išskyrus tai, kad vietoje reikšmių gauname formules iš įvesties simbolių [13]. Šios formulės – tai sąlygų, kurias turi tenkinti įvesties parametrai, kad būtų pasirinktas tam tikras konkretus vykdymo kelias, rinkinys. Jos yra vadinamos kelio apribojimais (angl. path constraint, žym. PC).[2]

Remiantis šiuo metodu yra sudaroma speciali duomenų struktūra – simbolinio vykdymo medis. Kiekviena šio medžio viršūnė atitinka vieną programos sakinį, o lankai (orientuotos briaunos) – perėjimus tarp šių sakinių. „If“ sakinį vaizduoja viršūnė, iš kurios išeina du lankai, vedantys į skirtingas viršūnes, priklausomai nuo ar sąlyga yra tenkinama („true“ šaka) ar ne („false“ šaka). Taip pat prie kiekvienos medžio viršūnės pažymima visa svarbi su vykdymo būsena susijusi informacija. Pavyzdžiui: sakinio numeris, kintamųjų reikšmės ir kelio sąlyga. Tokie simbolinio vykdymo pagrindu sudaryti medžiai pasižymi šiomis savybėmis:

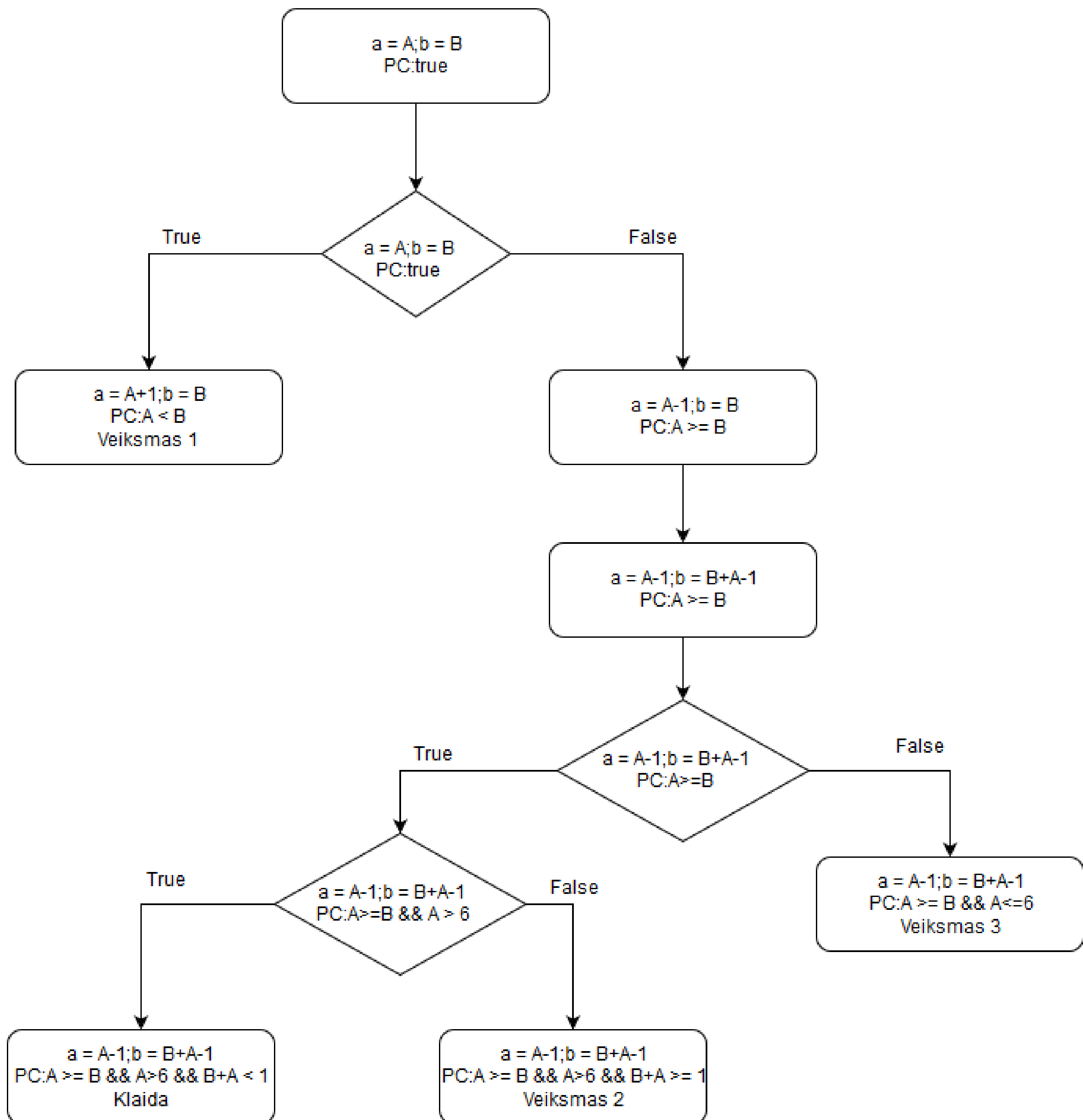
1. Kiekvienam medžio lapui (viršūnei, neturinčiai „vaikų“) egzistuoja tam tikras įvesties duomenų rinkinys, su kuriuo bus atliktas vykdymo kelias, kurį šis lapas žymi. Tai reiškia, kad PC niekada nėra lygus tiesiog „false“.
2. PC susiję su be kuriais dviem tokio medžio lapais yra skirtingi. Du vykdymo keliai, einantys iš bendros medžio šaknies (viršūnės neturinčios „tėvo“) ir vedantys į bet kurį iš lapų, turi unikalią šakojimosi viršūnę, kur vienam keliui sąlyga buvo tenkinama, o kitam ne (į vieno PC yra įrašoma x_sąlyga = true, į kito x_sąlyga = false).[13]

Programų testavimui simbolinis vykdymas yra naudojamas tam, kad būtų sugeneruojama testinių duomenų aibė, kuri padengia visus arba, jei tai neįmanoma (dėl resursų stygiaus), tam tikrą priimtiną dalį programos vykdymo kelių ir kiekvieną jų įvykdytų tik po kartą [6]. Simbolinio vykdymo taikymą testavime iliustruoja toliau pateiktas pavyzdys. Tarkime, turime programos fragmentą, pavaizduotą 1 pav. Uždavinys yra sugeneruoti testinių duomenų aibę taikant simbolinį vykdymą.

```
public void Pavyzdys(int a, int b)
    {
        if (a < b)
        {
            //Veiksmas 1
        }
        else {
            if (a > 5)
            {
                if (b < 0)
                {
                    //Klaida
                }
                else
                {
                    //Veiksmas 2
                }
            }
            else {
                //Veiksmas 3
            }
        }
    }
```

**1 pav.** Pavyzdinė funkcija

2 paveikslėlyje pateiktas simbolinio vykdymo medis sudarytas pasirinktam programos fragmentui:




**2 pav.** Simbolinio vykdymo medis

Kiekvienas šio medžio lapas atitinka vieną programos vykdymo kelią. Norint rasti įvesties duomenų rinkinį, kuris sukelia to kelio vykdymą reikia išspręsti jį atitinkančio lapo PC. Sprendimas gali būti vykdomas taip:

1. Pagal PC pasirenkame pirmojo kintamojo reikšmę, nekreipdami dėmesio į apribojimus, kurie sieja jį su kitais kintamaisiais (pvz., spręsdami kelio „Veiksmas2“ PC, pagal apribojimą A>6 pasirenkame reikšmę A = 10. Šiame žingsnyje nekreipiame dėmesio į apribojimus A>=B ir B+A >= 1, juos įvertinsime rinkdamiesi kintamojo b reikšmę).
2. Pagal ankstesniame žingsnyje pasirinktas kintamųjų reikšmes renkamės dar nenustatyto

kintamojo reikšmę taip, kad nebūtų pažeisti PC (pvz., spręsdami kelio „Veiksmas2“ PC ir prieš tai pasirinkę kintamojo A reikšmę 10, kintamajam B reikšmę renkame atsižvelgdami į šias sąlygas: A>=B ---> 10>=B, A > 6 – neaktualu, B+A >= 1 ---> B+10 >= 1. Taigi B reikšmių sritis yra: B <=10; B >= -9 ---> B∈[-9;10].)

3. Kartojame 2-ą žingsnį kol randame reikšmes visiems kintamiesiems arba susiduriame su situacija, kur tinkama reikšmė neegzistuoja (pvz., jei PC yra C > 6 && C == 3).

Sprendiniai pateikiami 1 lentelėje:

**1 lentelė.** PC sprendiniai

| Vykdymo kelias | PC | a | b | Pavyzdinis duomenų rinkinys |
|---|---|---|---|---|
| Veiksmas1 | A<B | <b | >a | a = 1;<br>b = 2; |
| Veiksmas2 | A>=B && A > 6<br>&& B+A >= 1 | >=b;<br>>6;<br>>=1-b; | <=a;<br>>=1-a; | a = 10;<br>b = 9; |
| Veiksmas3 | A>=B && A <= 6 | >=b;<br><=6; | <=a; | a = 1;<br>b = 0; |
| Klaida | A>=B && A > 6<br>&& B+A < 1 | >=b;<br>>6;<br><1-b; | <=a;<br><1-a; | a = 10;<br>b = -11; |

Nors simbolinio vykdymo technika pirmą kartą buvo pasiūlyta dar 8-ajame XX a. dešimtmetyje (1975m.), tačiau didesnio mokslininkų susidomėjimo ji sulaukė tik pastaruoju metu. To priežastys buvo:

1. Didelių programų PC yra labai sudėtingi. Tik per pastarąjį dešimtmetį buvo sukurti pakankamai galingi apribojimų sprendimo įrankiai, gebantys susidoroti su tokiais sudėtingais kelių apribojimais.
2. Simbolinis vykdymas reikalauja gerokai daugiau kompiuterio resursų nei kiti programų analizės būdai. Senesnės kartos kompiuteriai nebuvo pakankamai galingi, kad galėtų atlikti simbolinį didelių programų vykdymą.[2]

### 2.1.2. Paieška paremtas testavimas

Paieška paremtas testavimas – tai testinių atvejų (arba duomenų) generavimas, remiantis paieškos algoritmais, Jų veikimas priklauso nuo tikslo funkcijos (angl. fitness function [2],[1], objective function [15]), kuri nurodo testavimo tikslą [2]. Paieška paremto testavimo algoritmai siekia rasti „geriausią“ sprendimą, tačiau kiek geras yra sprendimas, jie nustato pagal tikslo funkciją, todėl jų tikslumas priklauso ne tik nuo paties algoritmo, bet ir nuo tikslo funkcijos [15]. Yra naudojami tokių tipų algoritmai:

- Hill Climbing – vietinės paieškos algoritmas. Atliekamas tokiais žingsniais:
    1. Pasirenkamas atsitiktinis paieškos erdvės taškas.
    2. Tikrinami aplinkiniai sprendimai.
    3. Priklausomai nuo algoritmo tipo pasirenkamas geriausias (pagal tikslo funkciją) kaimynas (greičiausio pakilimo strategija) arba pirmas pasitaikęs geresnis nei dabartinis (atsitiktinis arba pirmojo pakilimo strategija)
    4. Kartojamas 2 ir 3 žingsniai, kol neberandama geresnių sprendimų.

Šis algoritmas yra paprastas ir duoda greitus rezultatus, tačiau turi didelį trūkumą: pasitenkina lokaliu maksimumu ir todėl dažniausiai neranda optimaliausio sprendimo.

- Simulated Annealing – panašus į Hill Climbing algoritmą, tačiau kartais priima ir prastesnius sprendimus. Tai leidžia užtikrinti laisvesnį judėjimą po sprendimų erdvę. Tikimybė priimti prastesnį (pagal tikslo funkciją) sprendimą skaičiuojama pagal formulę

  $p=e^{\frac{-\delta}{t}}$ , čia δ – skirtumas tarp dabartinio ir svarstomo prastesnio sprendimo tikslo funkcijos reikšmių, t – temperatūra, kuri iš pradžių būna aukšta, kad užtikrintų laisvą judėjimą po paieškos erdvę, o vėliau mažėja. Svarbu užtikrinti, kad temperatūra nemažėtų per greitai, kitu atveju šis metodas taip pat kaip ir Hill Climbing gali užstrigti vietiniame maksimumo taške.
- Evoliuciniai algoritmai – naudoja evoliucijos modeliavimą kaip paieškos strategiją. Sprendimai evoliucionuoja taikant operatorius, įkvėptus genetikos ir natūraliosios atrankos. Geriausiai žinomas evoliucinių algoritmų tipas – genetiniai algoritmai. Jų esmė – sprendimai–kandidatai užkoduojami paprastų komponentų seka ir genetine chromosomos struktūra. Skirtingai nei anksčiau minėtos grupės, genetiniai algoritmai vienu metu dirba ne su vienu, bet keletu sprendimų. Tokia sprendimų grupė vadinama populiacija. Ši populiacija yra iteratyviai perkombinuojama ir keičiama išvedant populiacijas „palikuonis“ vadinamas kartomis.[15]

Remiantis S. Ali ir kt. straipsnyje „A systematic review of the application and empirical investigation of search-based test case generation“ ([1]) publikuotais duomenimis, populiariausia paieškos algoritmų klasė – genetiniai algoritmai (naudojami 73 % tirtų straipsnių), tuo tarpu antroje vietoje esantys „Simulated Annealing“ klasės sudaro tik 14 % visų naudojamų algoritmų. Tokį genetinių algoritmų populiarumą lemia tai, kad yra daug straipsnių apie genetinių algoritmų pritaikymą konkrečioms sritims. Taip pat yra didelis kiekis empirinių duomenų apie parametrus, reikalingus šiems algoritmams pritaikyti konkrečiai sričiai. Prie jų populiarumo prisideda ir tai, jog jie yra globalios paieškos algoritmai ir todėl jų rezultatai dažniausiai yra geresni nei lokalios paieškos algoritmų [1].

### 2.1.3. Kombinatorinis testavimas

Kombinatorinis testavimas – tai testavimo metodas, kuris testuoja SUT (sutr. system under test – testuojama sistema), naudodamas dengiančio masyvo testų rinkinį, kuris ištestuoja visas reikalingas parametrų reikšmių kombinacijas. Pagrindinis tokio testavimo tikslas – aptikti klaidas, kylančias dėl parametrų tarpusavio sąveikos [16].

Kombinatorinis testavimas, kurio stiprumo lygis (angl. strength) t (t >=2), reikalauja, kad kiekvienas t dydžio skirtingų sistemos įvesties parametrų reikšmių kortežas būtų padengtas nors vieno testinio atvejo[7]. Šią sąlyga galima patenkinti išbandant visus parametrų variantus, tačiau dažniausiai tai nėra įmanoma dėl milžiniško kombinacijų kiekio (pvz. yra rašoma, kad GCC – „GNU Compiler Collection“ – galimų konfigūracijų kombinacijų skaičius yra $10^{61}$ eilės [2]).

Pavyzdžiui, turime keturis parametrus $p_1$, $p_2$, $p_3$ ir $p_4$. Kiekvienas iš jų gali įgyti keturias reikšmes: 0, 1, 2 arba 3. Siekiame surasti testų rinkinį, kuris užtikrintų, kad kiekviena parametrų reikšmių pora būtų padengta (t. y. atlikti t = 2 stiprumo lygio kombinatorinį testavimą). Net ir šio paprasto pavyzdžio atveju yra $4^4$ = 256 skirtingų kombinacijų. Todėl tiesioginis perrinkimas nėra tinkamiausias variantas.

**2 lentelė.** Kombinatorinis testavimas (pavyzdys)

| $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 3 | 1 |
| 0 | 2 | 1 | 2 |
| 0 | 3 | 2 | 3 |
| 1 | 0 | 1 | 1 |

| | | | |
|---|---|---|---|
| 1 | 1 | 2 | 0 |
| 1 | 2 | 0 | 3 |
| 1 | 3 | 3 | 2 |
| 2 | 0 | 2 | 2 |
| 2 | 1 | 0 | 3 |
| 2 | 2 | 3 | 0 |
| 2 | 3 | 1 | 1 |
| 3 | 0 | 3 | 3 |
| 3 | 1 | 1 | 0 |
| 3 | 2 | 2 | 1 |
| 3 | 3 | 0 | 2 |
| 0 | 2 | 0 | 1 |
| 1 | 1 | 3 | 2 |
| 2 | 0 | 1 | 3 |
| 3 | 3 | 2 | 0 |

2 lentelėje pateikiamas 20 testinių duomenų rinkinį, kuris patikrintų visas galimas parametrų $p_1$, $p_2$, $p_3$ ir $p_4$ porų reikšmes. Iš šio pavyzdžio matome, kad tinkamai parinkus duomenis, galima žymiai sumažinti reikalingų testų skaičių (šiuo atveju reikėjo išbandyti tik kiek mažiau nei 8 % visų galimų parametrų kombinacijų, tam kad būtų padengtos visos galimos dviejų parametrų kombinacijos).

Matematiškai šis uždavinys (minimalaus testų rinkinio, padengiančio visas t eilės parametrų kombinacijas) yra ekvivalentus stiprumo t dengiančio masyvo radimui [7]. Dengiantis masyvas (angl. covering array, sutr. CA), CA(N; t ,k ,v) – tai N x k masyvas, toks, kad kiekviena N x t masyvo dalis (angl. sub-array) turėtų visus t dydžio kortežus iš v simbolių bent po vieną kartą. [2]

Dažniausiai naudojamas t=2 stiprumo kombinatorinis testavimas[7], tačiau tyrimai parodė, kad tai nėra pakankama. Pavyzdžiui, testai atlikti su NASA duomenų bazės taikomąja programa (angl. application) parodė, kad tokio stiprumo testavimas aptinka tik 93 % klaidų, tuo tarpu t=3 – 98 %. Todėl siekiant aukštesnio patikimumo rekomenduojama naudoti t = 4-6 stiprumo kombinatorinį testavimą [14].

### 2.1.4. Atsitiktinis generavimas ir prisitaikantis atsitiktinis generavimas

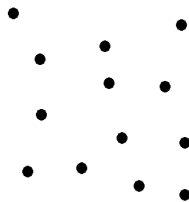
Yra du pagrindiniai testinių atvejų pasirinkimo variantai:

- Baltos dėžės testavimas
- Juodos dėžės testavimas

Viena iš paprasčiausių ir patogiausių juodos dėžės testavimo technikų yra atsitiktinis testų generavimas [8]. Jis yra lengvai suprantamas ir įgyvendinamas, todėl yra viena dažniausiai taikomų technikų tiek kaip savarankiškas testavimo metodas, tiek ir kaip kitų testavimo būdų komponentas. [2]

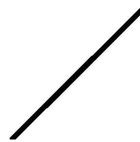
Yra skiriami trys pagrindiniai klaidą sukeliančių įvesčių tipai (angl. failure-causing input patterns) [22],[3]:

1. Taškinis (angl. point pattern) – klaidą sukeliančios įvestys išsimėčiusios po visą galimų reikšmių sritį ir/arba sudaro nedidelius (kartais tik vienos įvesties dydžio) regionus (žr. 3 pav. )



**3 pav.** Taškinis klaidą sukeliančių įvesčių tipas

2. Juostinis (angl. strip pattern) – klaidą sukeliančios įvestys išsidėsčiusios siaura juosta (žr. 4 pav. )



**4 pav.** Juostinis klaidą sukeliančių įvesčių tipas

3. Blokinis (angl. block pattern) – klaidą sukeliančios įvestys suformuoja vieną ar kelis didelius regionus (žr. 5 pav. )

**5 pav.**Blokinis klaidą sukeliančių įvesčių tipas

Įprasto atsitiktinio testavimo atveju šie tipai neturi įtakos tikimybei surasti klaidą, tačiau galima pastebėti, kad tiek juostiniam, tiek blokiniam klaidą sukeliančių įvesčių tipui galima ženkliai pagerinti aptikimo tikimybę šiek tiek modifikavus atsitiktinio generavimo algoritmą. Šios modifikacijos esmė yra tokia: kadangi klaidą sukeliančios įvestys dažnai suformuoja tam tikras sritis (juostas arba blokus), testinius atvejus reikėtų stengtis kiek įmanoma lygiau paskirstyti po visą galimų reikšmių sritį. Toks atsitiktinio testavimo variantas vadinamas prisitaikančiu atsitiktiniu generavimu[2],[22].

Yra daug būdų paskirstyti testinius duomenis po reikšmių sritį. To pasekmė – buvo sukurta daug prisitaikančio atsitiktinio testavimo algoritmų. Pagrindiniai jų tipai yra šie:

1. Geriausio kandidato pasirinkimas – atsitiktinai sugeneruojama testinių atvejų kandidatų aibė, iš kurios pagal tam tikrus kriterijus parenkamas geriausias;
2. Išskyrimas. Taikant šį metodą atliekami tokie žingsniai:
    1. Apibrėžiami išskyrimo regionai aplink kiekvieną jau įvykdytą testinį atvejį;
    2. Generuojamas atsitiktinis įvesties duomenų rinkinys;
    3. Jei nepatenka į nei vieną išskyrimo regioną, jis pasirenkamas kaip kitas testinis atvejis, kitu atveju kartojamas 2-asis žingsnis;
3. Dalinimas – naudodamasis informacija apie jau įvykdytų testinių atvejų vietas šis algoritmas dalina įvesties sritį į tam tikras dalis (angl. partitions) ir nustato vieną iš jų kaip regioną iš kurio bus generuojamas kitas testinis atvejis;
4. Testų profiliai – skirtingai nei kiti atsitiktinio testavimo būdai, šis nenaudoja vienodo testų profilio. Vietoje to, naudojamas specialus testų profilis, kuris leidžia tolygiai paskleisti testus po įvesties reikšmių sritį. Šiam algoritmui reikalinga galimybė dinamiškai keisti

testavimo profilį proceso metu;

5. Metrikomis grįstas – naudoja pasiskirstymo metrikas (pvz. dispersiją), kad pasirinktų tokius testinius atvejus, kad bendras testinių atvejų pasiskirstymas būtų kiek įmanoma lygesnis.

Reikia paminėti, kad be anksčiau išvardintų algoritmų grupių, egzistuoja ir kiti prisitaikančio atsitiktinio generavimo įgyvendinimo būdai. Be to, kiekvienai grupei priklauso daugiau nei vienas algoritmas. [2]

### 2.1.5. Modeliu pagrįstas testavimas

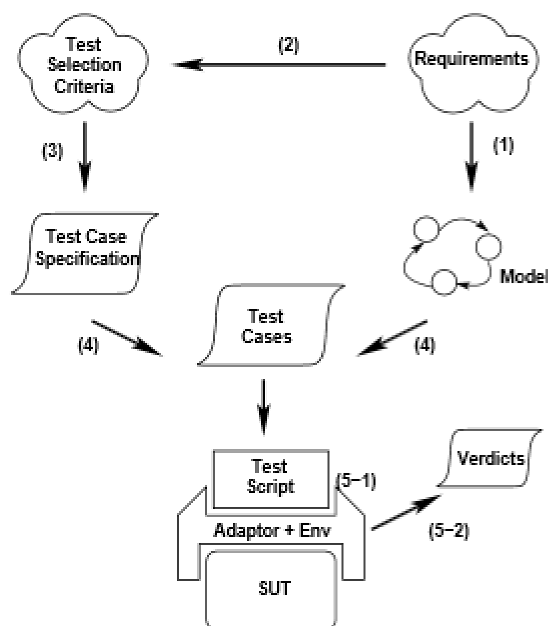
Modeliai yra naudojami daugelyje sričių, siekiant geriau suprasti, specifikuoti ir kurti sistemas. Modeliai taip pat yra svarbus programų inžinerijos proceso produktas, kuris aprašo numatomą sistemos elgseną, todėl gali būti pagrindu testų generavimui [3], [27].

Bendras modeliu pagrįsto testų generavimo procesas gali būti aprašomas taip:

1. Iš neformalių reikalavimų sudaromas SUT modelis. Jis dažnai vadinamas testo modeliu, nes yra tiesiogiai susijęs su testavimo tikslais. Šis modelis neturėtų būti tiesiogiai išvedamas iš jokių programos kūrimui naudojamų modelių, kad juose padarytos klaidos nepersiduotų į testus;
2. Parenkami testų pasirinkimo kriterijai (angl. test selection criteria), kurie funkcionuoja kaip gidas testų generatoriui, kurie testai yra „geri“, t. y. atitinkantys testavimo politiką nustatytą SUT;
3. Testų pasirinkimo kriterijai transformuojami į testinių atvejų specifikacijas (angl. test case specifications), kurios yra formalus antrajame žingsnyje apibrėžtų kriterijų pavidalas. Iš šiame žingsnyje sudarytų specifikacijų, automatinis testinių atvejų generavimo įrankis turi sugebėti generuoti testinius atvejus;
4. Pagal modelį ir testinių atvejų specifikacijas sugeneruojamas testinių atvejų rinkinys.
5. Sugeneruoti testai paleidžiami. Šį žingsnį gali atlikti žmogus arba testų paleidimas gali būti automatizuotas.[27], [28]

Šis procesas M. Utting'o, A. Pretschner'io ir B. Legeard'o straipsnyje „A taxonomy of model-based testing approaches“ ([28]) pavaizduotas taip (žr. 6 pav.):

MODEL-BASED TESTING TAXONOMY




**6 pav.** Modeliu paremto testavimo procesas pagal M. Utting'o, A. Pretschner'io ir B. Legeard'o straipsnį„A taxonomy of model-based testing approaches“

Šiuo metu egzistuoja daug ir įvairių modeliu pagrįstų įrankių: tiek komercinių, tiek ir tiriamųjų, bei mokslinių straipsnių šia tematika.[2] (pvz., Dias'as Neto A.C. ir kt. rado daugiau kaip 200 straipsnių susijusių su modeliu pagrįstu testavimu [11]). Dėl šios priežasties, modeliu paremti testavimo metodai skirstomi pagal įvairius aspektus, kurių keli pateikiami 3 lentelėje [2], [27], [28], [26].

**3 lentelė**. Modeliu paremto testų generavimo metodų klasifikatoriai

| Aspektas | Kategorijos | Komentarai |
|---|---|---|
| Modelio apimtis: Įvestis/išvestis | Įvestis | Modelis aprašo tik SUT įvestis |
| | Įvestis ir išvestis | Modelis aprašo ir SUT įvestis, ir išvestis. Privalumas – toks modelis gali veikti kaip testų orakulas (angl. test oracle) |
| Modelio notacija | Būsena (prieš- ir po-sąlygomis) paremtos notacijos | Pvz. Z, B, VDM, JML, OCL |
| | Perėjimais paremtos notacijos | Dažniausiai sistema vaizduojama kaip baigtinis automatas (FSM) |
| | Istorija paremtos notacijos | Modeliuoja sistemą |

| | | |
|---|---|---|
| | | apibrėždamos leidžiamą sistemos elgseną bėgant laikui |
| | Funkcinės notacijos | Apibrėžia sistemą matematinėmis funkcijomis |
| | Operacinės notacijos | Apibrėžia sistemą kaip grupę lygiagrečių procesų |
| | Stochastinės notacijos | Apibrėžia sistemą tikimybiniu modeliu |
| | Duomenų srautų notacijos | Modeliuoja sistemą koncentruojantis į duomenis, o ne programos valdymą |
| Testų parinkimo kriterijai | Struktūrinio modelio padengimas | Remiasi esamo modelio struktūra pvz., sąlygos sakiniais pre/post notacijoje |
| | Duomenų padengimas | Stengiamasi padengti maksimalią duomenų erdvę su minimaliu testinių atvejų skaičiumi (pvz., Kombinatorinis testavimas) |
| | Reikalavimais pagrįsti kriterijai | Stengiamasi padengti visus reikalavimus |
| | Ad-hoc testinių atvejų specifikacijos | Šalia modelio pateikiamos tiesiogiai išreikštos testinių atvejų specifikacijos |
| | Atsitiktinai ir stochastiniai kriterijai | Naudojami aplinkos modeliams |
| | Klaidomis pagrįsti kriterijai | Tinkami SUT modeliams, nes testavimo tikslas yra rasti klaidas SUT. |
| Testų vykdymas | Online | Testai generuojami programos (SUT) vykdymo metu, atsižvelgiant į jos grąžinamus rezultatus |
| | Offline | Testai generuojami prieš vykdant programą (SUT) |

## 2.2. Testų generavimo įrankių savybių kiekybinis ir/ arba kokybinis palyginimas

Egzistuoja daug ir labai įvairių testų generavimo įrankių. Šiame skyrelyje apžvelgsime ir palyginsime keletą iš jų.

### 2.2.1. Įrankių apžvalga

#### *2.2.1.1. Randoop*

„Randoop“ – tai vienetų testų generavimo įrankis, naudojantis grįžtamuoju ryšiu pagrįstą (angl. feedback-directed) testų generavimą. Generuodamas testus šis įrankis juos kartu ir vykdo. Pagal testų vykdymo metu gautus rezultatus, generuojami tolesni testai. Šio įrankio veikimo algoritmas yra toks:

1. Atsitiktiniu būdu sukuriamas testų rinkinys;
2. Sukurtas rinkinys įvykdomas ir patikrinimas pagal tam tikrų apribojimų (angl. contracts) rinkinį;
3. Rezultatai yra klasifikuojami į tris grupes:
   3.1. Apribojimus pažeidžiantys testai – pažeidžiami apribojimai, bet nesukelia išimčių (angl. exceptions). Pateikiami vartotojui ir naudojami tolesnių testų generavimui;
   3.2. „Teisingi“ testai – tokie, kurių metu nepažeidžiami apribojimai, ir jų vykdymas nesukelia išimčių. Pateikiami vartotojui (gali būti naudojami regresiniam testavimui) ir naudojami tolesnių testų generavimui;
   3.3. Išimtis sukeliantys testai – jų vykdymas sukelia išimtis (pvz., Java IllegalArgumentException). Jie yra pašalinami ir nenaudojami naujų testų generavimui;
4. Kartojami pirmi 3 žingsniai (1-ajame remiantis 3-čiojo metu sugeneruotais testais), kol sugeneruojamas reikiamas testų kiekis.

Įrankio vartotojai gali įtakoti Randoop veikimą aprašydami apribojimus (angl. contracts) arba naudodami anotacijas testuojamos sistemos kode (pvz. @Omit – negeneruoti testų taip anotuotam metodui).[17]

#### *2.2.1.2. AETG*

AETG (sutr. Automated Efficient Test Generator) – modeliu pagrįstas testinių duomenų generavimo įrankis kombinatoriniam testavimui. Šis įrankis testų generavimui naudoja:

- Testuojamos sistemos įvedimo duomenų modelį, aprašytą AETG įvedimo duomenų aprašymo kalba (angl. AETG Input Language);
- Tam tikrą kombinatorinio testavimo algoritmą (dažniausiai naudojamas porinis (angl. pair-wise) algoritmas, užtikrinantis, kad sugeneruoti testai padengs visas galimas kintamųjų porų reikšmių kombinacijas. Tačiau įrankis palaiko ir aukštesnės eilės kombinatorinio testavimo algoritmus);

Šis įrankis yra pasiekiamas kaip saityno paslauga (angl. web service).[28],[9]

#### 2.2.1.3. EvoSuite

„EvoSuite“ – tai testų generavimo įrankis naudojantis paieška pagrįsto testavimo metodologiją. Šis įrankis naudoja keletą žinomų technikų (pvz. hibridinė paieška ar dinaminis simbolinis vykdymas), tačiau jo išskirtinumą lemia šios dvi savybės:

- Viso testų rinkinio generavimas (angl. Whole test suite generation) – naudojamas evoliucinių grupės paieškos algoritmas, kuris tobulina testų rinkinį, atsižvelgdamas ne į atskirus padengimo tikslus (angl. coverage goals), o į visą padengimo kriterijų. Tai lemia, kad sugeneruoti testai nėra per daug įtakojami atskirų padengimo tikslų sudėtingumo ar neįmanomumo
- Mutaciniu testavimu (angl. mutation testing) pagrįstas prielaidų (angl. assertions) generavimas – padeda apriboti generuojamų prielaidų kiekį naudojant mutacinį testavimą.

[12]

#### 2.2.1.4. Spec Explorer

„Microsoft“ įrankis integruojamas į „Visual Studio“ programavimo aplinką. Leidžia kurti modelius ir iš jų generuoti testus, juos išsaugant arba iš karto įvykdant.

Modeliai yra aprašomi naudojant C# kalbą ir reguliariąsias išraiškas (angl. regular expressions). Yra aprašomas SUT įvesties/išvesties modelis, kuris pateikia orakulą kiekvienam generuojamam testiniam atvejui.[28]

#### 2.2.1.5. KLEE

KLEE – tai simbolinio vykdymo įrankis, gebantis automatiškai generuoti testus sudėtingoms programoms, kurie pasiekia aukštus kodo padengimo rodiklius. Pagrindinės šio įrankio savybės:

- įvairios apribojimų sprendimo optimizacijos;
- glaustas programos būsenų atvaizdavimas;
- euristinių paieškos metodų naudojimas, siekiant aukštų kodo padengimo rodiklių.

[5]

#### 2.2.1.6. Smartesting CertifyIt

Tai modeliu pagrįsto testų generavimo įrankis. „CertifyIt“ remiasi UML diagramomis ir OCL apribojimais. Šio įrankio naudojamą modelį sudaro trys UML diagramų tipai:

- Klasių diagrama. Aprašo statinį modelio vaizdą, abstrakčius sistemos objektus ir jų

priklausomybes. Leidžiami šie elementai:

  - Klasės;
  - Ryšiai;
  - Išvardinimai (angl. enumerations);
  - Klasių atributai;
  - Klasių operacijos;

- Objektų diagrama. Skirta pradinės būsenos ir SUT testinių duomenų modeliavimui;
- Būsenos diagrama (angl. state-machine diagram). Naudojama modeliuoti dinaminiam SUT elgesiui kaip FSM (sutr. angl. finite state machine – baigtinis automatas).

OCL apribojimai naudojami formaliam SUT elgesio aprašymui. [4]

#### *2.2.1.7. T3i*

„T3i" – tai automatinio testų generavimo įrankis, kuris generuoja testinius atvejus kaip kreipinių į testuojamos klasės metodus seką. Išskirtinė šio įrankio savybė – jis laiko testų rinkinius pirmos klasės objektais, t. y. vartotojas gali su jais atlikti kombinavimo, filtravimo ir užklausų jiems siuntimo veiksmus. Naudodamasis anksčiau minėtas operacijas, vartotojas per kelias iteracijas gali sukurti jo poreikius tenkinantį testų rinkinį.

Šis įrankis naudoja atsitiktiniu generavimu pagrįstą algoritmą, kuris yra tam tikras „Randoop" įrankio naudojamo algoritmo variantas.

T3i taip pat leidžia atlikti šių formų sudėtingesnes užklausas:

- „Hoare triple" – naudojamos metodo priešsąlygių ir posąlygių aprašymui;
- LTL (angl. sutr. Linear Temporal Logic – tiesinės laiko logikos) formulės;
- Algebrinės užklausos (angl. algebraic queries);

[21]

#### *2.2.1.8. Jalangi*

„Jalangi" – tai karkasas, kuris atlieka įvairių tipų dinaminę analizę „JavaScript" kodui. Šis karkasas įgyvendina šias technikas:

1. Pasirinktinis įrašymas-atkartojimas (angl. Selective record-replay) – įrašo ir atkartoja tam tikros vartotojo pasirinktos dalies veikimą;

2. Šešėlinės reikšmės (angl. shadow values) – kiekviena reali reikšmė susiejama su tam tikra šešėline jos išraiška, kuri saugo naudingą informaciją apie realią reikšmę (pvz., simbolinę jos išraišką);

Karkasas gali atlikti aibę dinaminės analizės veiksmų:

- Simbolinį ir konkretų vykdymą apjungiantis metodas (angl. concolic testing);
- „null“ ir „undefined“ reikšmių kilmės aptikimas;
- dinaminė „dėmių“ (angl. taint) analizė;
- tipų nenuoseklumo (angl. inconsistency) aptikimas;
- atminties priskyrimo paprastiems objektams profiliavimas .

[24]. „Jalangi“ nėra orientuotas vien tik į testų generavimą, tačiau jis yra paminėtas šioje apžvalgoje, kadangi testinių duomenų generavimas, remiantis simboliniu ir konkrečiu programos vykdymu, yra dalis jo teikiamo funkcionalumo.

#### *2.2.1.9. RT-Tester*

„RT-Tester“ – tai modeliu pagrįsto testų generavimo įrankis. Šis įrankis modeliui aprašyti naudoja UML ir SysML modeliavimo kalbų poaibius. SUT struktūra išreiškiama UML struktūros diagramomis arba funkcinėmis schemomis (angl. block diagram), o dinaminiai aspektai modeliuojami baigtiniais automatais ir jų operacijomis. Įvairiais kitais įrankiais sudaryti modeliai gali būti eksportuojami XMI formatu ir pritaikomi naudojimui „RT-Tester“. Taip pat modelių sudarymui palaikoma ir „Matlab Simulink“ kalba. [19]

#### *2.2.1.10. JTest*

„Parasoft“ kompanijos produktas siūlantis tokį funkcionalumą:

- Statinė analizė;
- Vienetų testų generavimas;
- Kodo padengimo analizė;
- Kodo peržiūros;
- Klaidų aptikimas realiu laiku.

Šio įrankio generuojami vienetų testai gali būti regresinio testavimo pagrindu, kadangi kaip orakulą naudoja dabartinį SUT veikimą.

#### *2.2.1.11. JTExpert*

„JTExpert“ – tai paieška paremto testavimo įrankis, kuris generuoja JUnit testinius atvejus iš Java pirminio kodo (angl. source code) failų. Šiam įrankiui nereikalinga jokia papildoma informacija, išskyrus SUT programos kodą. [23]

### 2.2.2. Palyginimo kriterijai

Atliekant įrankių palyginimą bus remiamasi šiais kriterijais:

- Generavimo metodai, t. y. naudojama technika (atsitiktinis generavimas, simbolinis vykdymas, kombinatorinis testavimas, paieška paremtas generavimas, modeliu paremtas generavimas) ar jų kombinacija;
- Įvedamų duomenų formatai. Lyginama pagal tai, kokie įvesties duomenys naudojami testų generavimui;
- Platinimas. Nurodoma, ar įrankis yra komercinis, ar nemokamas;
- Vartotojo sąsaja. Pateikiami palaikomi vartotojo sąsajos tipai (komandinė eilutė, grafinė vartotojo sąsaja ar abu);
- Regresinis testavimas. Šis kriterijus parodo, ar galima įrankiu sugeneruotus testus naudoti kaip pagrindą regresiniam testavimui

### 2.2.3. Palyginimas

Toliau pateikiamas įvairių testų generavimo įrankių palyginimas ankstesniame skyrelyje aprašytais aspektais (žr. 4-6 lenteles):

**4 lentelė**. Įrankių palyginimas (1) (pagal generavimo metodą)

| Įrankis | Generavimo metodas | | | | |
|---|---|---|---|---|---|
| | Simbolinis vykdymas | Paieška pagrįstas testavimas | Kombinatorinis testavimas | Atsitiktinis generavimas | Modeliu pagrįstas testavimas |
| Randoop | - | - | - | + | - |
| AETG | - | - | + | - | + |
| EvoSuite | - | + | - | - | - |
| SpecExplorer | - | - | - | - | + |
| KLEE | + | - | - | - | - |
| Smartesting CertifyIt | - | - | - | - | + |
| T3i | - | - | - | + | - |
| Jalangi | + | - | - | - | - |
| RT-Tester | - | - | - | - | + |
| Jtest | ? | ? | ? | ? | - |

| JTExpert | - | + | - | - | - |
|---|---|---|---|---|---|

**5 lentelė**. Įrankių palyginimas (2) įvesties duomenys

| **Įrankis** | **Įvedamų duomenų formatai** | | | | |
|---|---|---|---|---|---|
| | **Programos kodas** | **Anotacijos** | **UML** | **OCL** | **Kiti** |
| Randoop | Java | + | - | - | - |
| AETG | - | - | - | - | AETG įvedimo duomenų aprašymo kalba |
| EvoSuite | Java | - | - | - | - |
| SpecExplorer | - | - | - | - | C# kalba ir reguliariosiomis išraiškomis aprašomas modelis |
| KLEE | C, LLVM bytecode | - | - | - | - |
| Smartesting CertifyIt | - | - | + | + | - |
| T3i | Java | - | - | - | - |
| Jalangi | Javascript | - | - | - | - |
| RT-Tester | - | - | + | - | SysML, MatLab Simulink modeliai |
| Jtest | Java | + | - | - | - |
| JTExpert | Java | - | - | - | - |

**6 lentelė.** Įrankių palyginimas (3)

| **Įrankis** | **Platinimas** | **Vartotojo sąsaja** | | **Regresinis testavimas** | **Orakulas** |
|---|---|---|---|---|---|
| | | **Komandinė eilutė** | **Grafinė vartotojo sąsaja** | | |
| Randoop | Nemokamas | + | Eclipse įskiepis | + | - |
| AETG | Komercinis | - | Tinklo sąsaja | - | - |
| EvoSuite | Nemokamas | + | Eclipse, Intellij | + | Iš dalies |

| | | | IDEA, įskiepiai | | aprašomas prielaidomis |
|---|---|---|---|---|---|
| SpecExplorer | Komercinis | - | Visual Studio įskiepis | - | Iš modelio |
| KLEE | Nemokamas | + | - | - | - |
| Smartesting CertifyIt | Komercinis | ? | ? | - | - |
| T3i | Nemokamas | + | Groovy | + | - |
| Jalangi | Nemokamas | + | - | - | - |
| RT-Tester | Komercinis | + | Eclipse įskiepis | - | Iš modelio |
| Jtest | Komercinis | + | Eclipse ir kt. | + | Dabartinė sistemos elgsena |
| JTExpert | Nemokamas | + | - | - | - |

# 3. PROJEKTINĖ DALIS

## 3.1. Architektūros pateikimas

OCL modeliu paremtų testų generavimo įrankio architektūros aprašą sudaro toliau pateikti architektūriniai vaizdai:

1. Panaudojimo atvejų vaizdas;
2. Sistemos statinis vaizdas;
3. Sistemos dinaminis vaizdas;
4. Sistemos išdėstymo vaizdas.

**7 lentelė.** Architektūriniai vaizdai ir juos sudarantys modeliavimo elementai

| Architektūrinis vaizdas | Modeliavimo elementai |
|---|---|
| Panaudojimo atvejų vaizdas | Panaudos atvejų diagrama |
| | Panaudojimo atvejų aprašas |
| Sistemos statinis vaizdas | Paketų diagrama |
| | Klasių diagramos |
| Sistemos dinaminis vaizdas | Būsenos diagramos |

| | |
|---|---|
| | Veiklos diagramos |
| | Sekų diagramos |
| | Bendradarbiavimo diagramos |
| Sistemos išdėstymo vaizdas | Išdėstymo diagrama |

## 3.2. Architektūros tikslai ir apribojimai

Toliau pateikiami reikalavimai ir sprendimai, turintys reikšmingą poveikį toliau šiame dokumente aprašomo įrankio architektūrai:

- Įrankis turi būti realizuotas kaip „Eclipse IDE“ programavimo aplinkos įskiepis;
- OCL apribojimų analizavimui (angl. parsing) gali būti naudojamos trečiųjų šalių COTS arba OSS priemonės;
- Kodo padengimo skaičiavimo (angl. code coverage) funkcionalumo realizavimui gali būti naudojamos trečiųjų šalių COTS arba OSS priemonės;

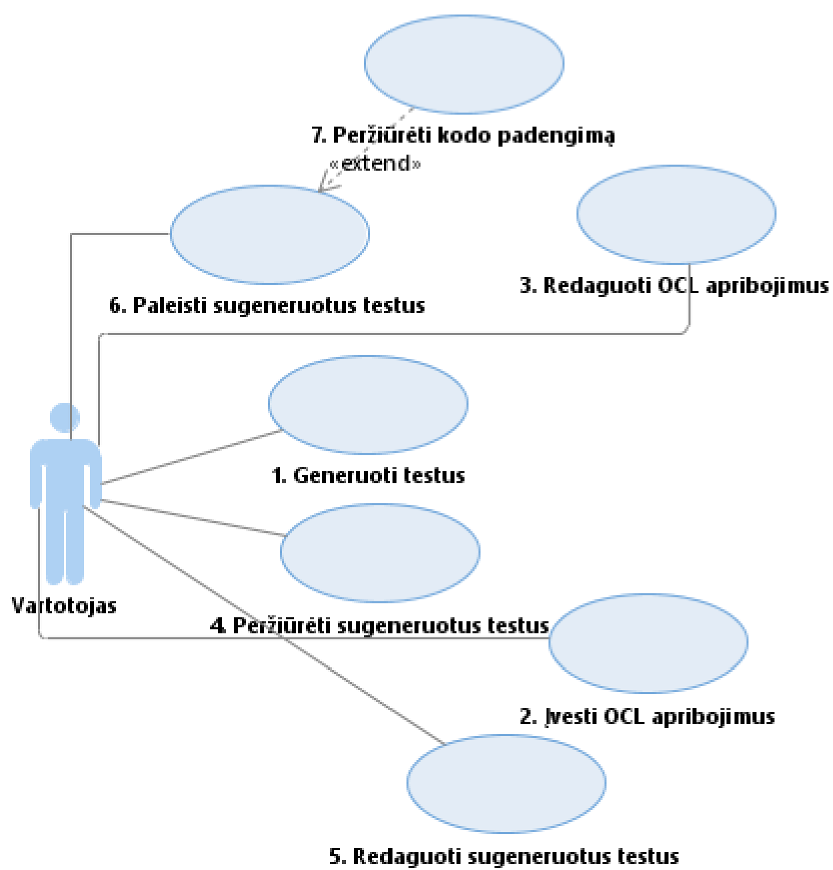
Projektavimui naudojami šie įrankiai:

- Magic Draw (UML diagramų braižymui);
- IBM Rational Software Architect (UML diagramų braižymui);
- Apache OpenOffice Writer (dokumentų rengimui);

## 3.3. Panaudojimo atvejų vaizdas

### 3.3.1. Sistemos sudėtis

#### *3.3.1.1. Sistemos ribos (panaudojimo atvejų modelis)*




**7 pav.** Panaudojimo atvejų diagrama

#### *3.3.1.2. Panaudojimo atvejai*

**8 lentelė.** Panaudos atvejis: generuoti testus

<table>
<tr><td>Numeris</td><td>1</td><td>Pavadinimas</td><td>Generuoti testus</td></tr>
<tr><td colspan="2">Tikslas</td><td colspan="2">Sugeneruoti testus pasirinktam elementui (klasei arba metodui)</td></tr>
<tr><td colspan="2">Aktoriai</td><td colspan="2">Vartotojas</td></tr>
<tr><td colspan="2">Prieš-sąlygos</td><td colspan="2">-</td></tr>
<tr><td colspan="2">Sužadinimo sąlyga</td><td colspan="2">Pasirenkamas elementas (klasė arba metodas);</td></tr>
<tr><td colspan="2">Po-sąlyga</td><td colspan="2">Yra sugeneruotas vienas arba daugiau testų pasirinktam elementui</td></tr>
<tr><td colspan="2">Pagrindinis scenarijus</td><td colspan="2">1. Pasirenkamas kontekstinio meniu punktas „Generuoti testus“<br>2. Įrankis nuskaito OCL apribojimus, turinčius įtaką testuojamam elementui<br>3. Sugeneruojamas testų rinkinys pasirinktam elementui</td></tr>
<tr><td colspan="2" rowspan="3">Alternatyvūs scenarijai</td><td>Pasirinkta klasė neturi metodų</td><td>1. Pasirenkama klasė;<br>2. Pasirenkamas kontekstinio meniu punktas „Generuoti testus“;<br>3. Sugeneruojamas tuščias testų rinkinys, parodomas įspėjimas, pranešantis, kad klasei nepavyko sudaryti testų, nes ji neturi testuojamų metodų.</td></tr>
<tr><td>OCL apribojimai neteisingi</td><td>1. Atliekamas pagrindinis scenarijus (ignoruojant neteisingus apribojimus);<br>2. Vartotojui pranešama, kad OCL apribojimai buvo neteisingi;<br>3. Pataisomi apribojimai (atliekamas PA 3. Redaguoti OCL apribojimus pagrindinis scenarijus);<br>4. Pakartojamas pagrindinis scenarijus.</td></tr>
<tr><td>OCL apribojimai neegzistuoja</td><td>1. Atliekamas pagrindinis scenarijus, testų generavime nenaudojant OCL apribojimų;<br>2. Vartotojui pranešama, kad OCL apribojimai – nerasti.</td></tr>
</table>

**9 lentelė.** Panaudos atvejis: įvesti OCL apribojimus

| **Numeris** | 2 | **Pavadinimas** | Įvesti OCL apribojimus |
|---|---|---|---|
| **Tikslas** | | Sukurti OCL apribojimų failą, papildyti jį naujais apribojimais | |
| **Aktoriai** | | Vartotojas | |
| **Prieš-sąlygos** | | - | |
| **Sužadinimo sąlyga** | | Vartotojas sukuria OCL apribojimų failą arba jį atveria | |
| **Po-sąlyga** | | Įvesti nauji OCL apribojimai | |
| **Pagrindinis scenarijus** | | 1. Sukuriamas OCL apribojimų failas;<br>2. Įvedamas naujas apribojimas (šis žingsnis kartojamas, kol gaunamas norimas apribojimų rinkinys). | |
| **Alternatyvūs scenarijai** | | **OCL apribojimų failas egzistuoja** | 1. Atveriamas OCL apribojimų failas;<br>2. Įvedamas naujas apribojimas (šis žingsnis kartojamas, kol gaunamas norimas apribojimų rinkinys). |

**10 lentelė.** Panaudos atvejis: redaguoti OCL apribojimus

| **Numeris** | 3 | **Pavadinimas** | Redaguoti OCL apribojimus |
|---|---|---|---|
| **Tikslas** | | Pakeisti OCL apribojimų turinį | |
| **Aktoriai** | | Vartotojas | |
| **Prieš-sąlygos** | | 1. Yra sukurtas OCL apribojimų failas;<br>2. Yra įvestas bent vienas apribojimas. | |
| **Sužadinimo sąlyga** | | Vartotojas atveria OCL apribojimų failą | |
| **Po-sąlyga** | | Įvesti nauji OCL apribojimai | |
| **Pagrindinis scenarijus** | | 1. Atveriamas OCL apribojimų failas;<br>2. Pataisomas vienas iš apribojimų (šis žingsnis kartojamas, kol atliekami visi norimi pataisymai). | |
| **Alternatyvūs scenarijai** | | **-** | |

**11 lentelė.** Panaudos atvejis: peržiūrėti sugeneruotus testus

| **Numeris** | 4 | **Pavadinimas** | Peržiūrėti sugeneruotus testus |
|---|---|---|---|
| **Tikslas** | | Pamatyti testų generavimo rezultatus | |
| **Aktoriai** | | Vartotojas | |
| **Prieš-sąlygos** | | Yra sugeneruotas bent vienas testas | |
| **Sužadinimo sąlyga** | | Vartotojas paspaudžia sugeneruoto testo pirminio teksto (angl. source code) failo piktogramą. | |
| **Po-sąlyga** | | Vartotojas mato kokie testai yra sugeneruoti atvertame faile. | |
| **Pagrindinis scenarijus** | | 1. Atveriamas pasirinktas failas | |

| **Alternatyvūs scenarijai** | - |
| --- | --- |

**12 lentelė.** Panaudos atvejis: redaguoti sugeneruotus testus

| **Numeris** | 5 | **Pavadinimas** | Redaguoti sugeneruotus testus |
| --- | --- | --- | --- |
| **Tikslas** | | Dažnai sugeneruotų testų prielaidos neturi prasmės (pvz. assert(true)), todėl juos reikia modifikuoti, pakeičiant prielaidas prasmingomis. | |
| **Aktoriai** | | Vartotojas | |
| **Prieš-sąlygos** | | Yra atvertas sugeneruotų testų failas, kuriame yra bent vienas testas | |
| **Sužadinimo sąlyga** | | Vartotojas atlieka pakeitimus sugeneruotų testų faile | |
| **Po-sąlyga** | | Sugeneruotų testų faile matomi vartotojo atlikti pakeitimai. | |
| **Pagrindinis scenarijus** | | 1. Keičiamas testų failo turinys; | |
| **Alternatyvūs scenarijai** | | - | |

**13 lentelė.** Panaudos atvejis: paleisti sugeneruotus testus

| **Numeris** | 6 | **Pavadinimas** | Paleisti sugeneruotus testus |
| --- | --- | --- | --- |
| **Tikslas** | | Patikrinti programos veikimą sugeneruotais testais | |
| **Aktoriai** | | Vartotojas | |
| **Prieš-sąlygos** | | Egzistuoja bent vienas testų rinkinys | |
| **Sužadinimo sąlyga** | | Vartotojas pasirenka testų rinkinį ir paspaudžia kontekstinio meniu punktą „Paleisti testus“ | |
| **Po-sąlyga** | | Matomi testų rezultatai | |
| **Pagrindinis scenarijus** | | 1. Paleidžiamas testas;<br>2. Patikrinama ar rezultatas atitinka prielaidą;<br>3. Parodomas testo rezultatas;<br>4. Jei rinkinyje dar yra testų, kartojami pirmi 3 veiksmai kitam testui. | |
| **Alternatyvūs scenarijai** | | **Testų rinkinys yra tuščias** | 1. Pranešama vartotojui, kad paleisti tuščio rinkinio neįmanoma. |

**14 lentelė.** Panaudos atvejis: peržiūrėti kodo padengimą

| **Numeris** | 7 | **Pavadinimas** | Peržiūrėti kodo padengimą |
| --- | --- | --- | --- |
| **Tikslas** | | Pamatyti kokia kodo dalis buvo ištestuota | |
| **Aktoriai** | | Vartotojas | |
| **Prieš-sąlygos** | | Egzistuoja bent vienas testų rinkinys | |
| **Sužadinimo sąlyga** | | Vartotojas, prieš vykdydamas testus (PA 7. Paleisti sugeneruotus testus), | |

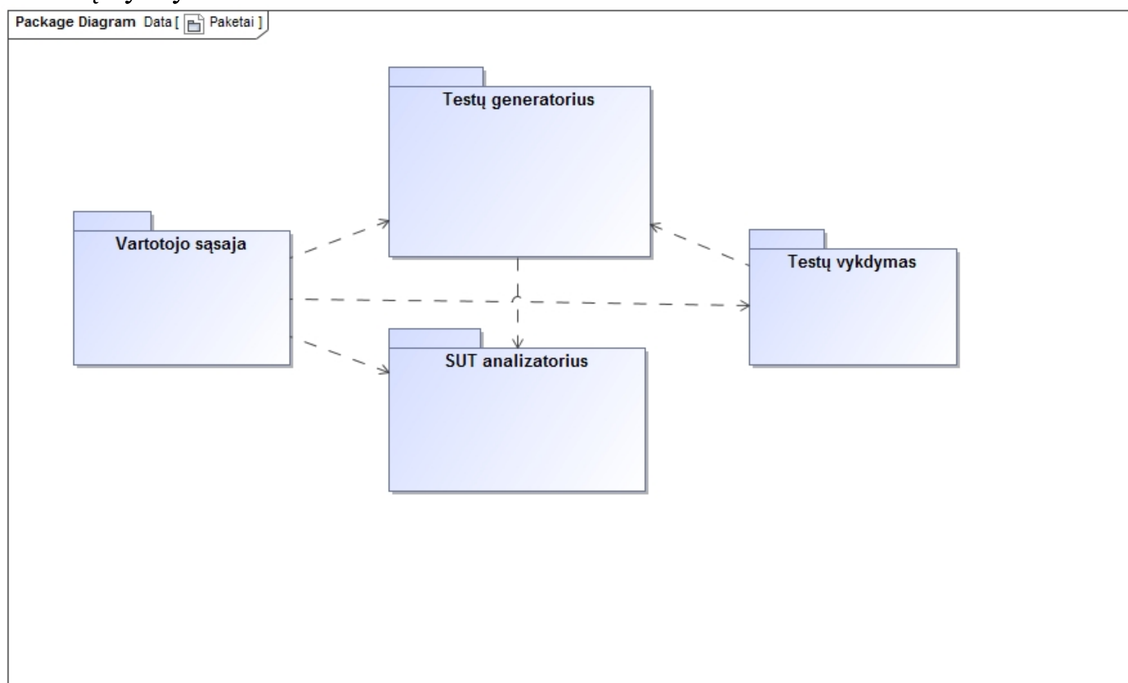
| Numeris | 7 | Pavadinimas | Peržiūrėti kodo padengimą |
|---|---|---|---|
| | | pasirenka matuoti kodo padengimą | |
| Po-sąlyga | | Matomi kodo padengimo rodikliai | |
| Pagrindinis scenarijus | | 1. Vartotojas paleidžia testų rinkinį<br>2. Atliekami PA 7. Paleisti sugeneruotus testus veiksmai, įvertinant kokias kodo dalis kiekvienas testas padengė | |
| Alternatyvūs scenarijai | | Testų rinkinys yra tuščias | 1. Pranešama vartotojui, kad paleisti tuščio rinkinio neįmanoma |

## 3.4. Sistemos statinis vaizdas

### 3.4.1. Apžvalga

Sistema skaidoma į paketus funkciniu pagrindu. Yra skiriami 4 paketai (kaip pavaizduota 8 pav.):

1. Vartotojo sąsaja;
2. Testų generavimas;
3. SUT analizavimui;
4. Testų vykdymas.




**8 pav.** Paketų diagrama

### 3.4.2. Paketų detalizavimas

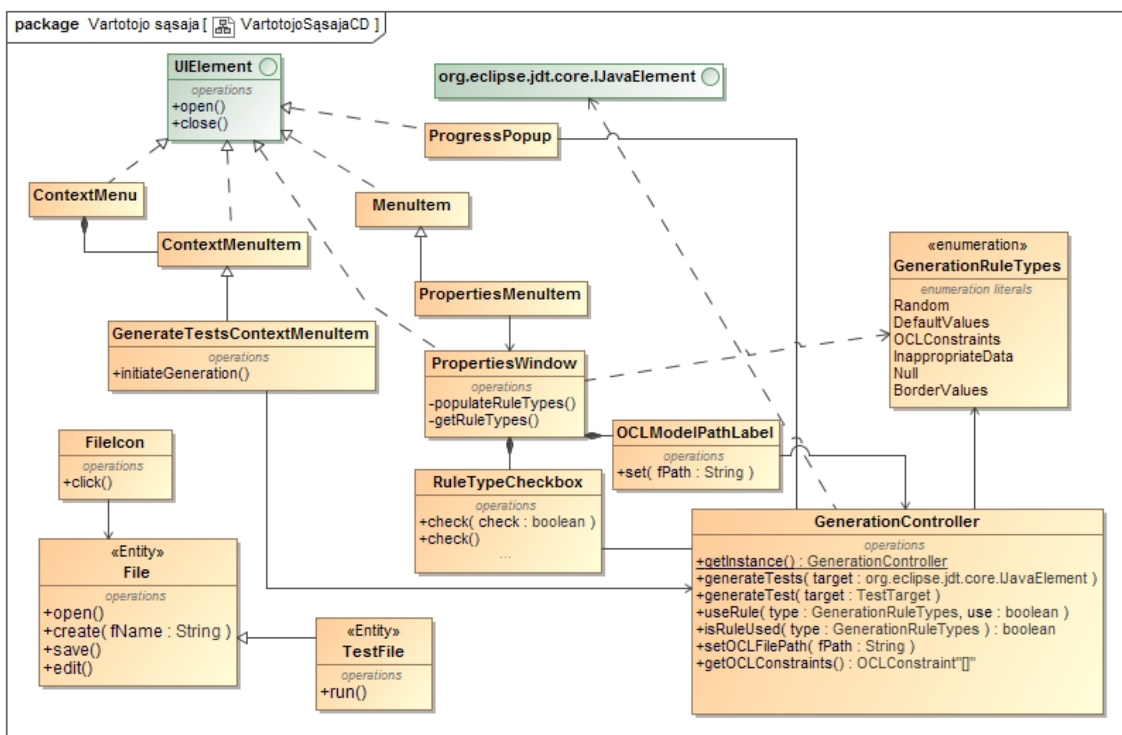
Toliau pateikiamos kiekvieno paketo klasių diagramos. Klasių vardai bei metodų pavadinimai realizuojant sistemą bus rašomi anglų kalba, todėl ši kalba vartojama ir modelyje.

### *3.4.2.1. Vartotojo sąsaja*

Šio paketo paskirtis – užtikrinti įrankio bendravimą su vartotoju. Svarbu pažymėti, kad dalis klasių diagramoje (9 pav.) pavaizduotų elementų kuriant įrankį nebus realizuojami, nes:

a) Jie nepriklauso įrankiui, tik yra jo naudojami (pvz. sąsaja org.eclipse.jdt.core.IJavaElement, kuri yra skirta Java modelio elemento aprašymui, bus reikalinga testų generatoriui, tačiau nėra jo dalis);

b) Jie neatspindi programinės klasės (pvz. File ar TestFile: šios klasės modeliuoja failų sistemos objektą „failą“ ir yra pateikiamos, nes dalis darbo su įrankiu vyksta tiesiogiai manipuliuojant failais, taigi jie yra laikomi vartotojo sąsajos dalimi).

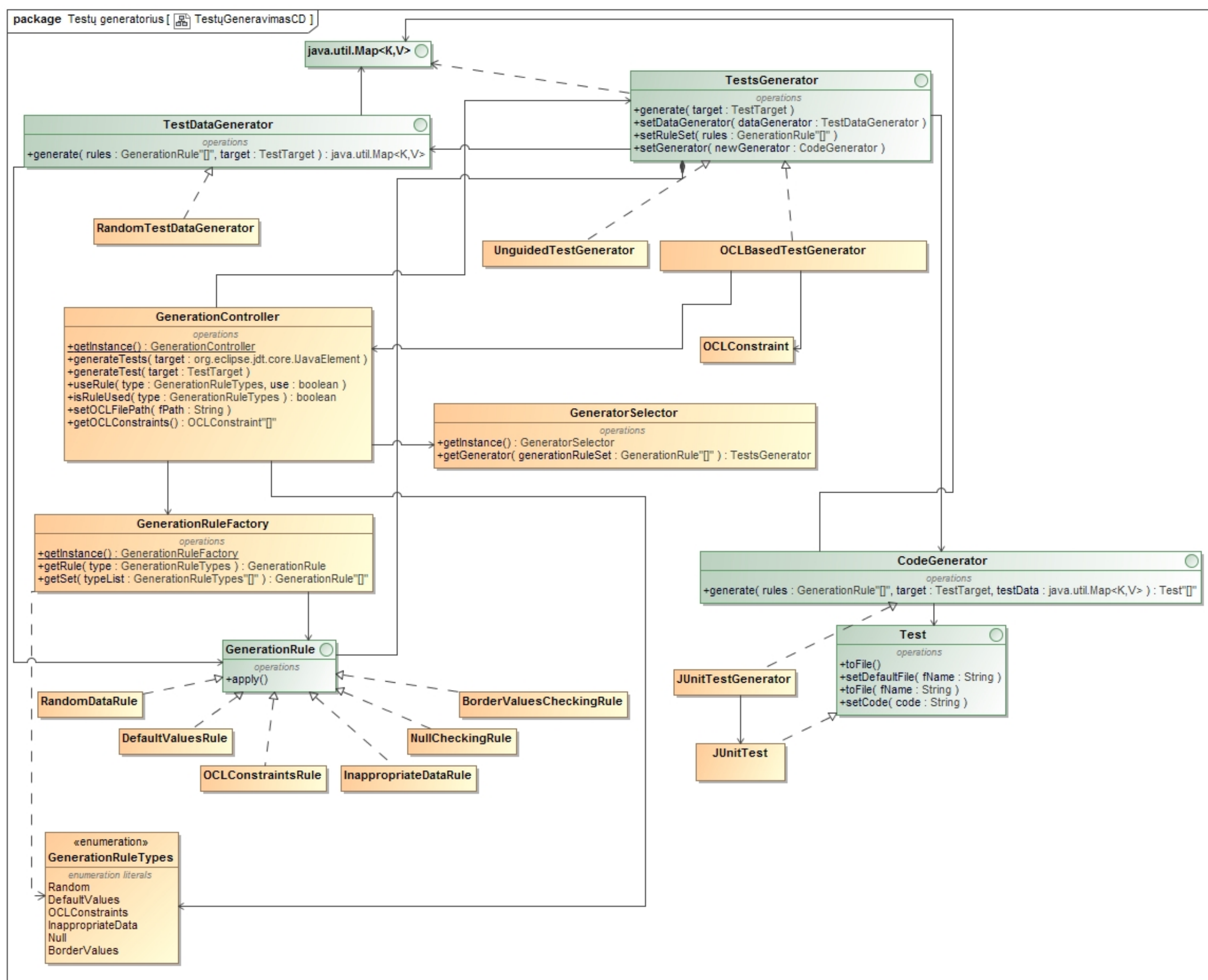



**9 pav.** Paketas „Vartotojo sąsaja“: klasių diagrama

### *3.4.2.2. Testų generatorius*

Šio paketo klasės atsakingos už testinių duomenų bei pačių testų generavimą. Testų generatoriaus paketo struktūra pateikiama 10 pav.

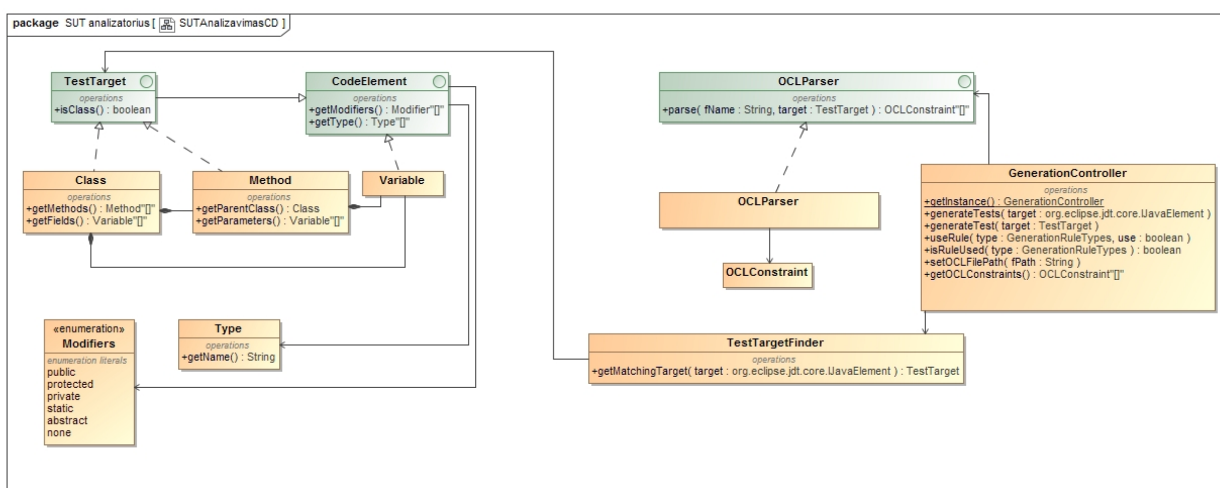
Pastaba: Klasė GenerationController yra atsakinga už testavimo proceso koordinavimą ir bendrauja su įvairių paketų klasėmis, todėl ji yra pateikiama ne tik vartotojo sąsajos ir testų generatoriaus, bet ir SUT analizatoriaus diagramose.




**10 pav.** Paketas „Testų generatorius“: klasių diagrama

### *3.4.2.3. SUT analizatorius*

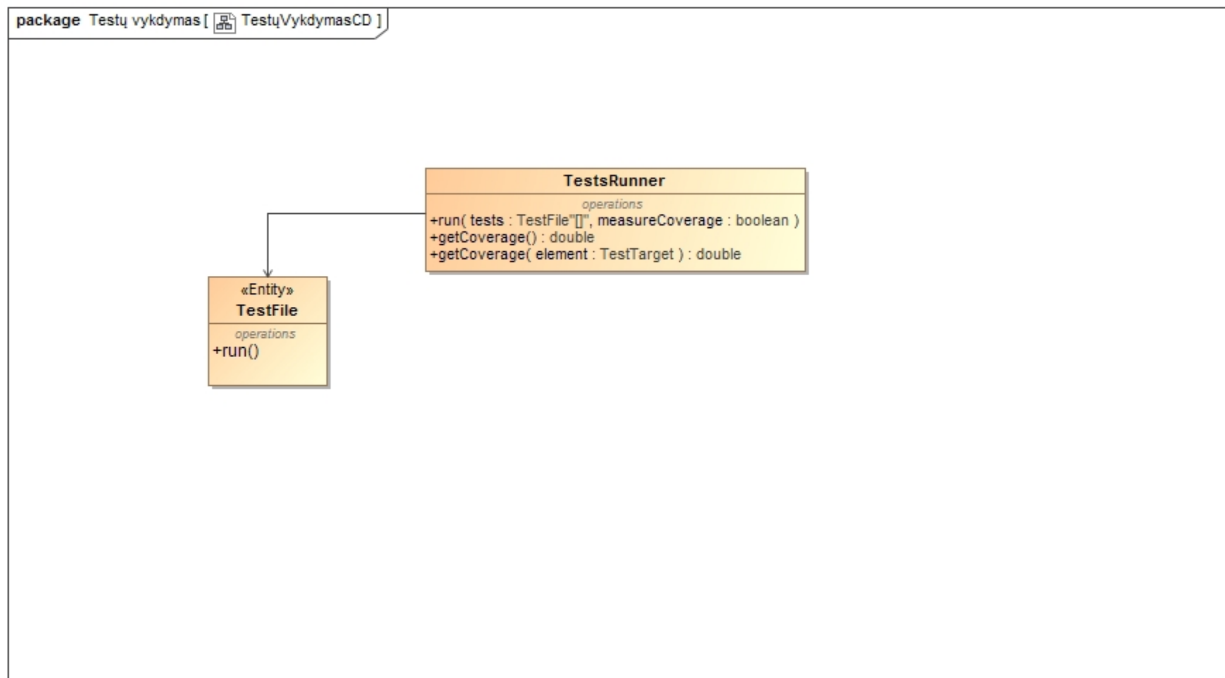
Paketas, kurio klasės atlieka testuojamos sistemos (sutr. SUT) ir OCL apribojimų analizę.




**11 pav.** Paketas „SUT analizatorius“: klasių diagrama

### *3.4.2.4. Testų vykdymas*

Kadangi testų vykdymo ir kodo padengimo skaičiavimo funkcionalumas bus realizuojamas trečiųjų šalių COTS ir OSS priemonėmis, šio paketo klasių diagrama nėra detalizuojama.



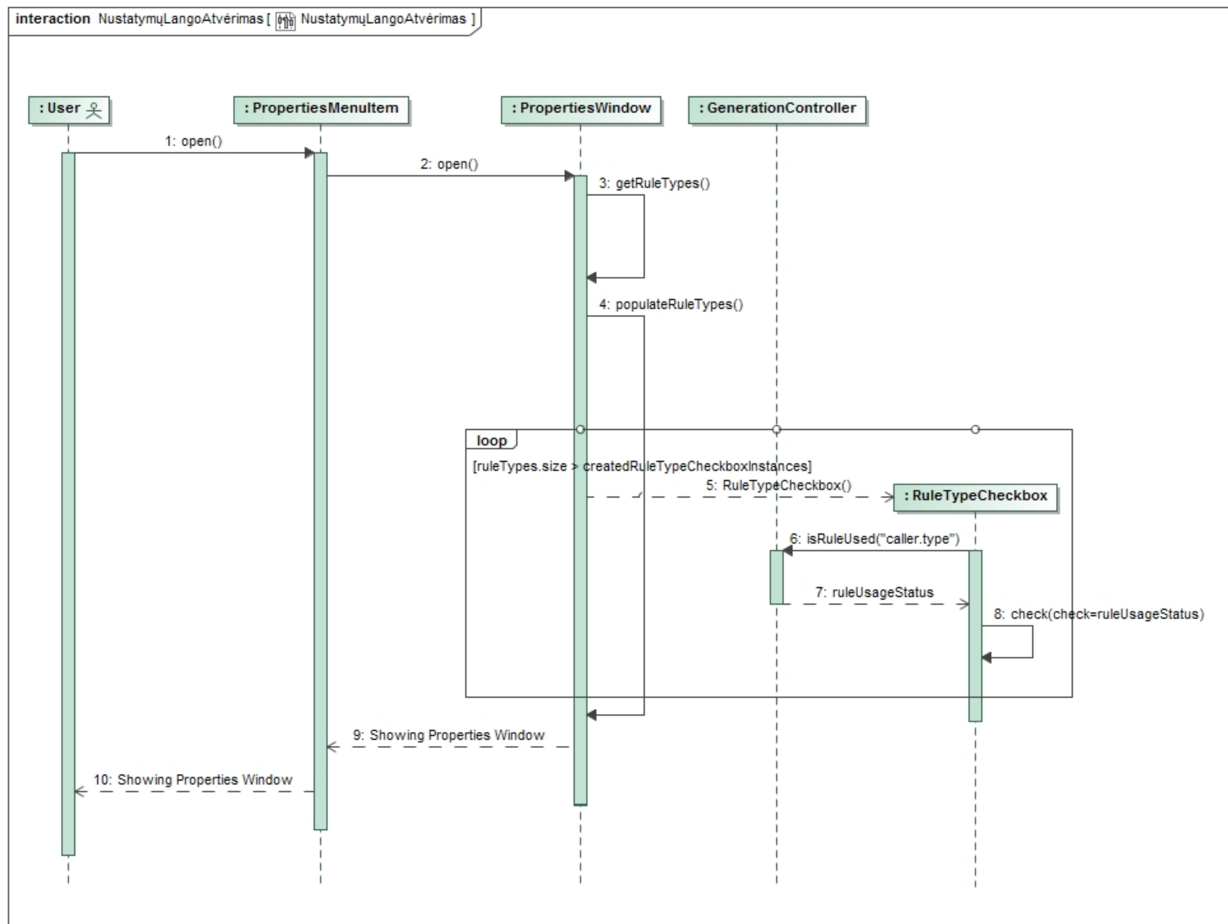

**12 pav.** Paketas „Testų vykdymas“: klasių diagrama

### 3.5. Sistemos dinaminis vaizdas

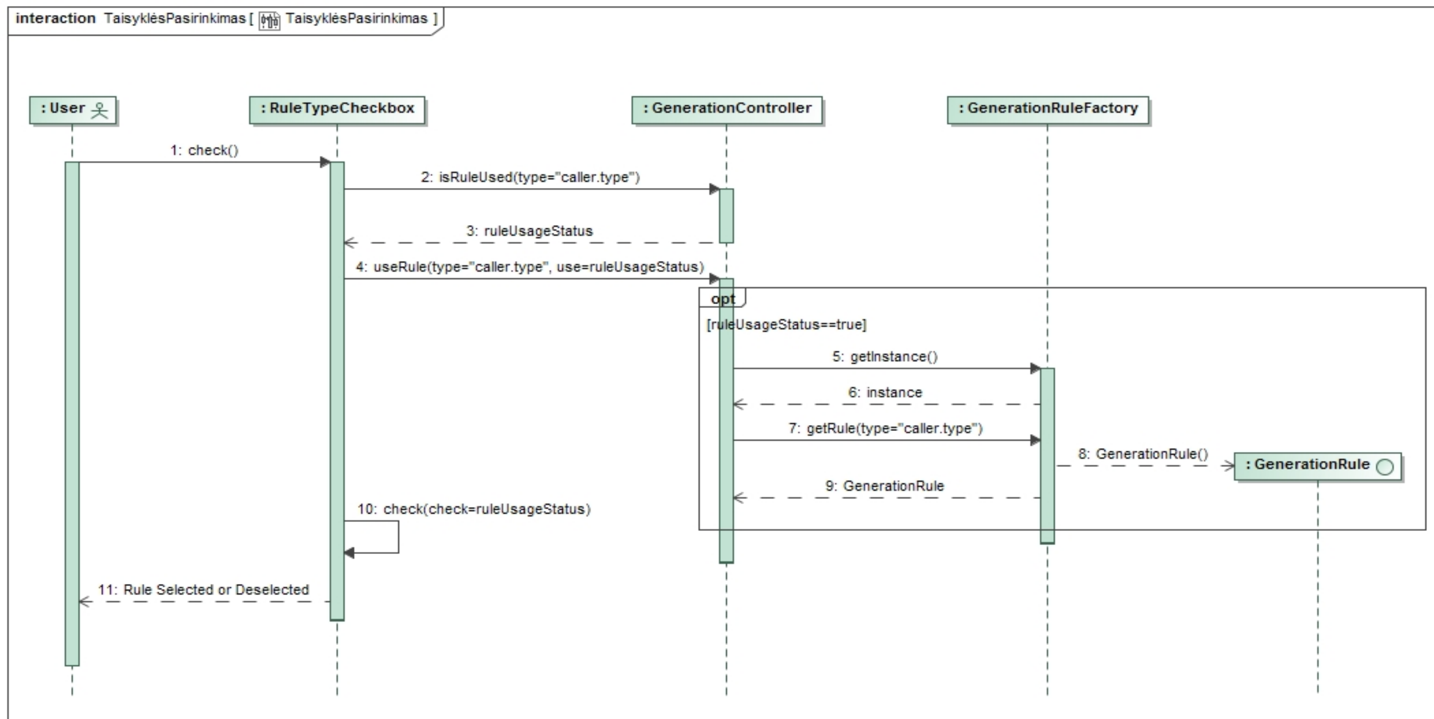
Šiame skyrelyje pateikiamas sistemos dinaminis vaizdas, išreikštas sekų diagramomis kiekvienam panaudos atvejui. Taip pat dinaminis architektūrinis vaizdas yra papildomas sistemos veiklos bei būsenų diagramomis.

## 3.5.1. Testų generavimas

Pirmasis testų generavimo žingsnis – naudojamų taisyklių rinkinio nustatymas. Šis procesas pavaizduotas 13, 14 pav. sekų diagramomis:

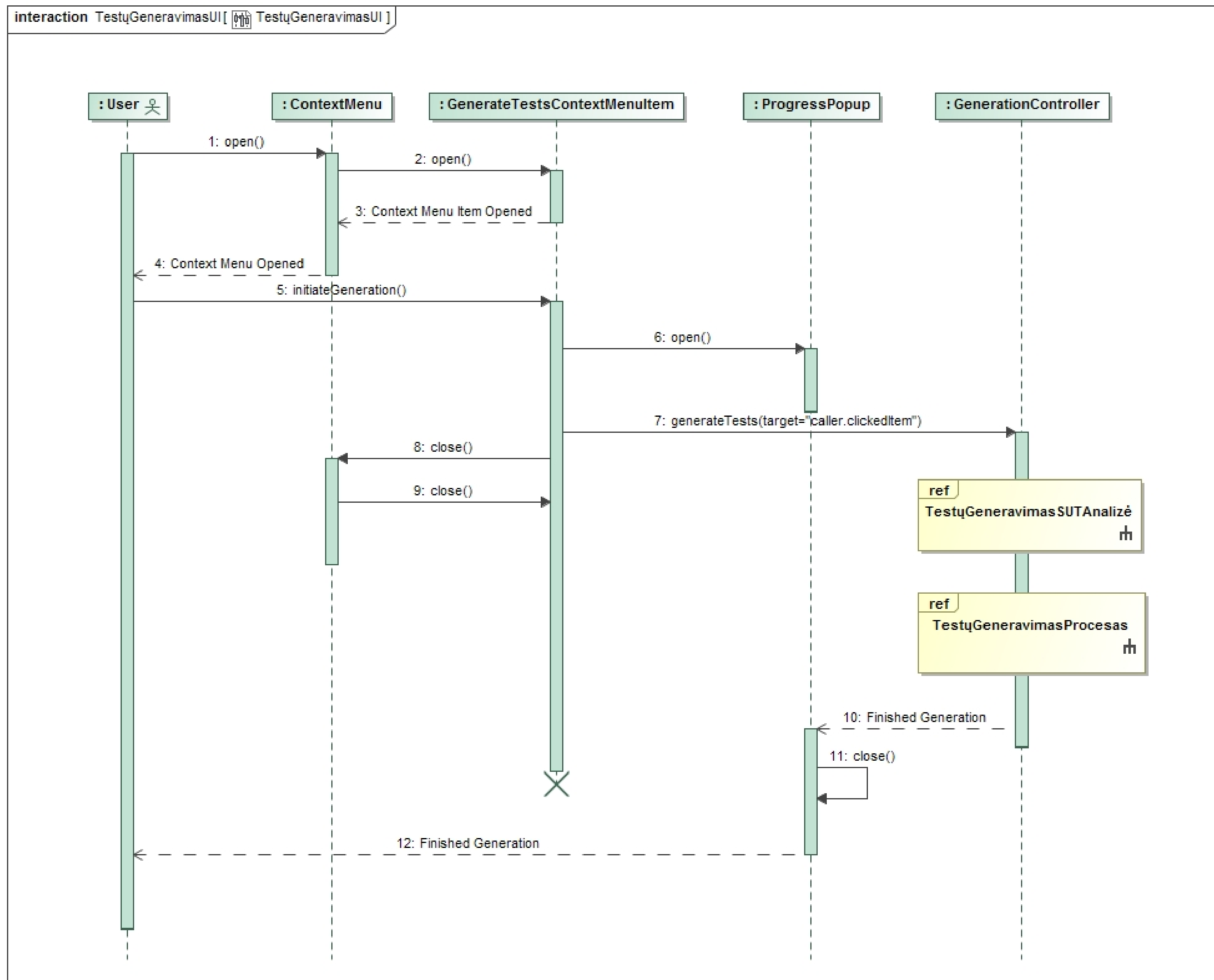



**13 pav.** Sekų diagrama: Parinkčių lango atvėrimas

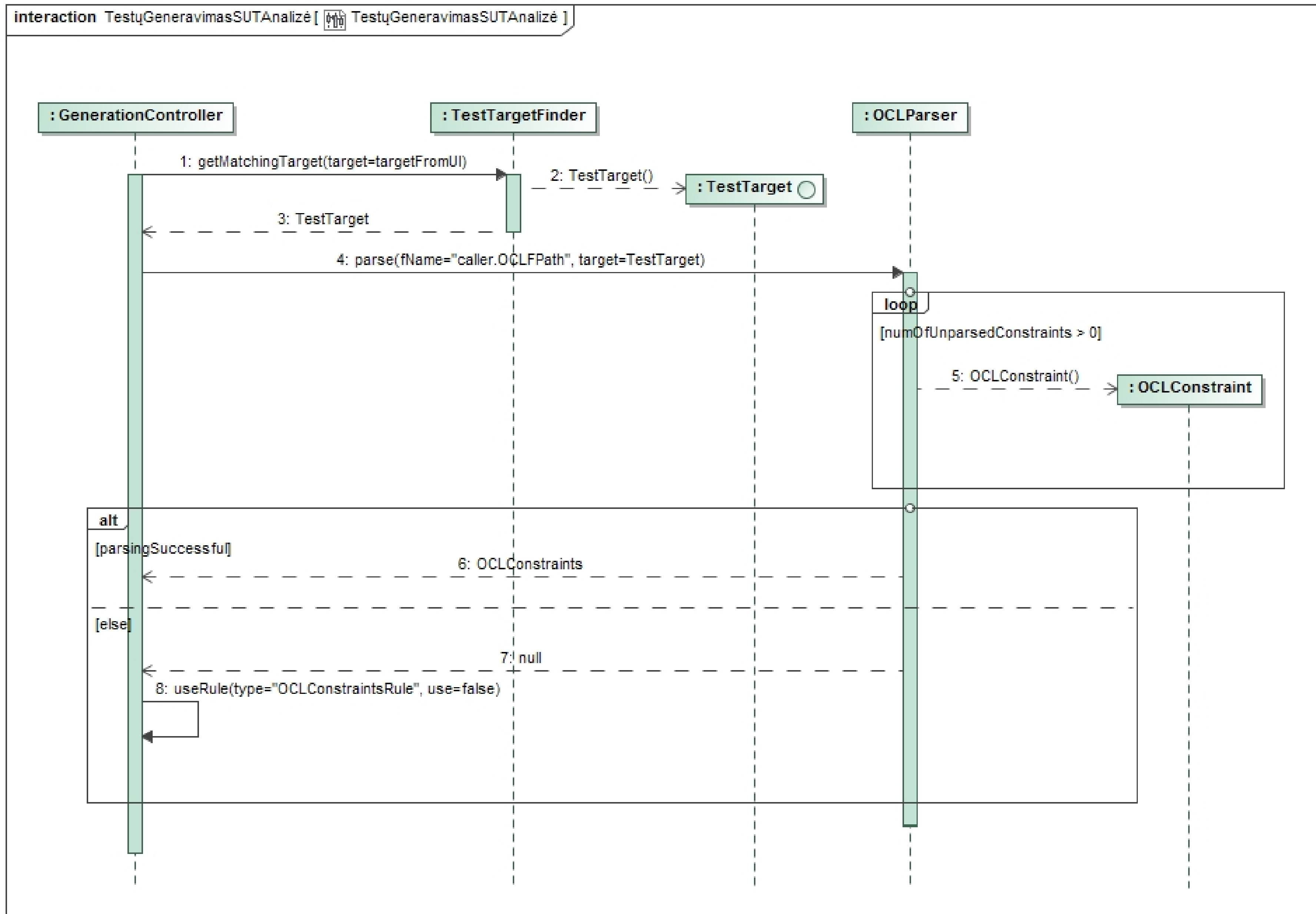



**14 pav.** Sekų diagrama: Taisyklės pasirinkimas

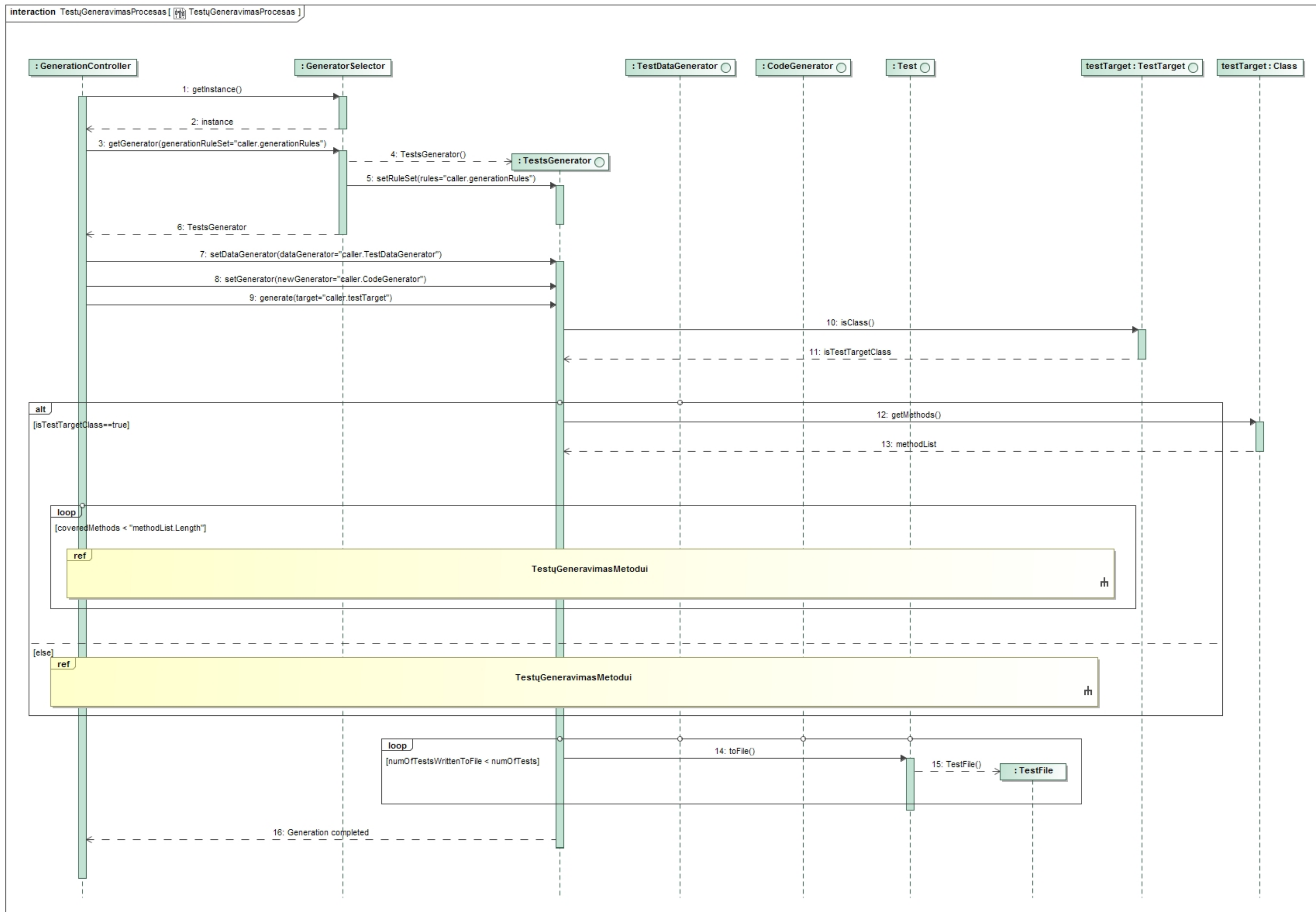
Toliau pateikiamos keletas sekų ir bendradarbiavimo diagramų aprašančių patį testų generavimo procesą (15–19 pav.):



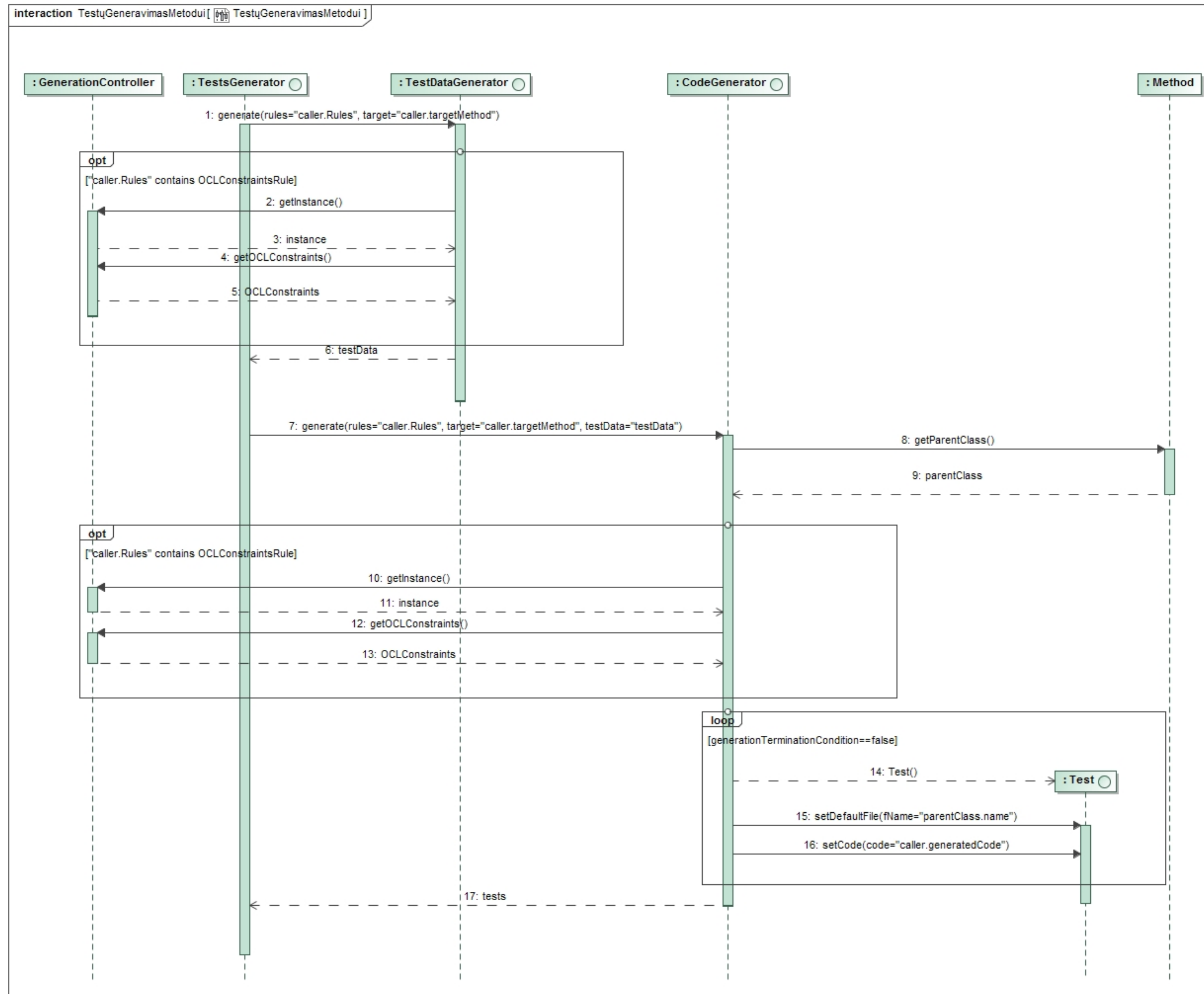

**15 pav.** Sekų diagrama: Testų generavimas – sąveika su UI



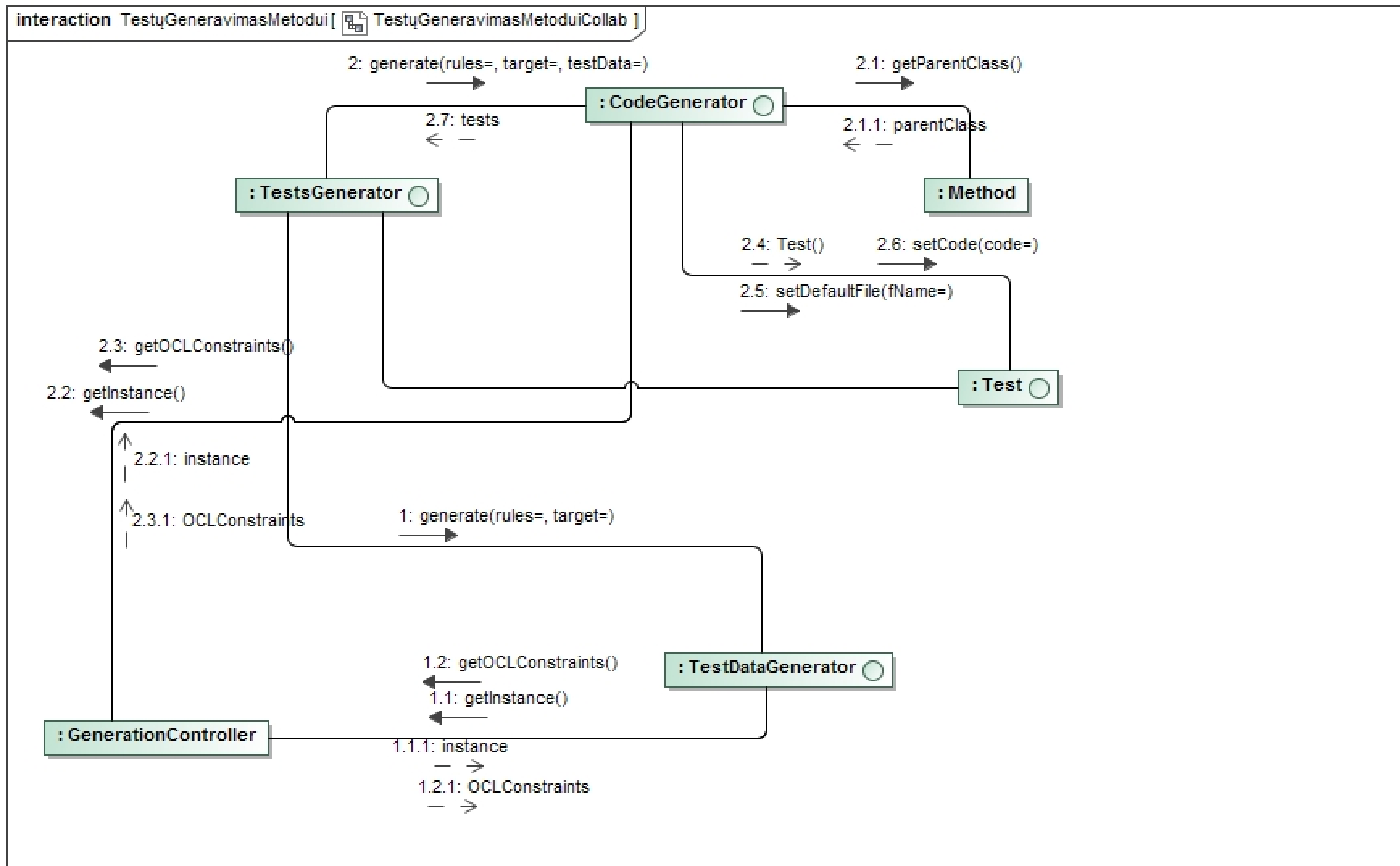

**16 pav.** Sekų diagrama: Testų generavimas – SUT analizė




**17 pav.** Sekų diagrama: Testų generavimas – generavimo procesas

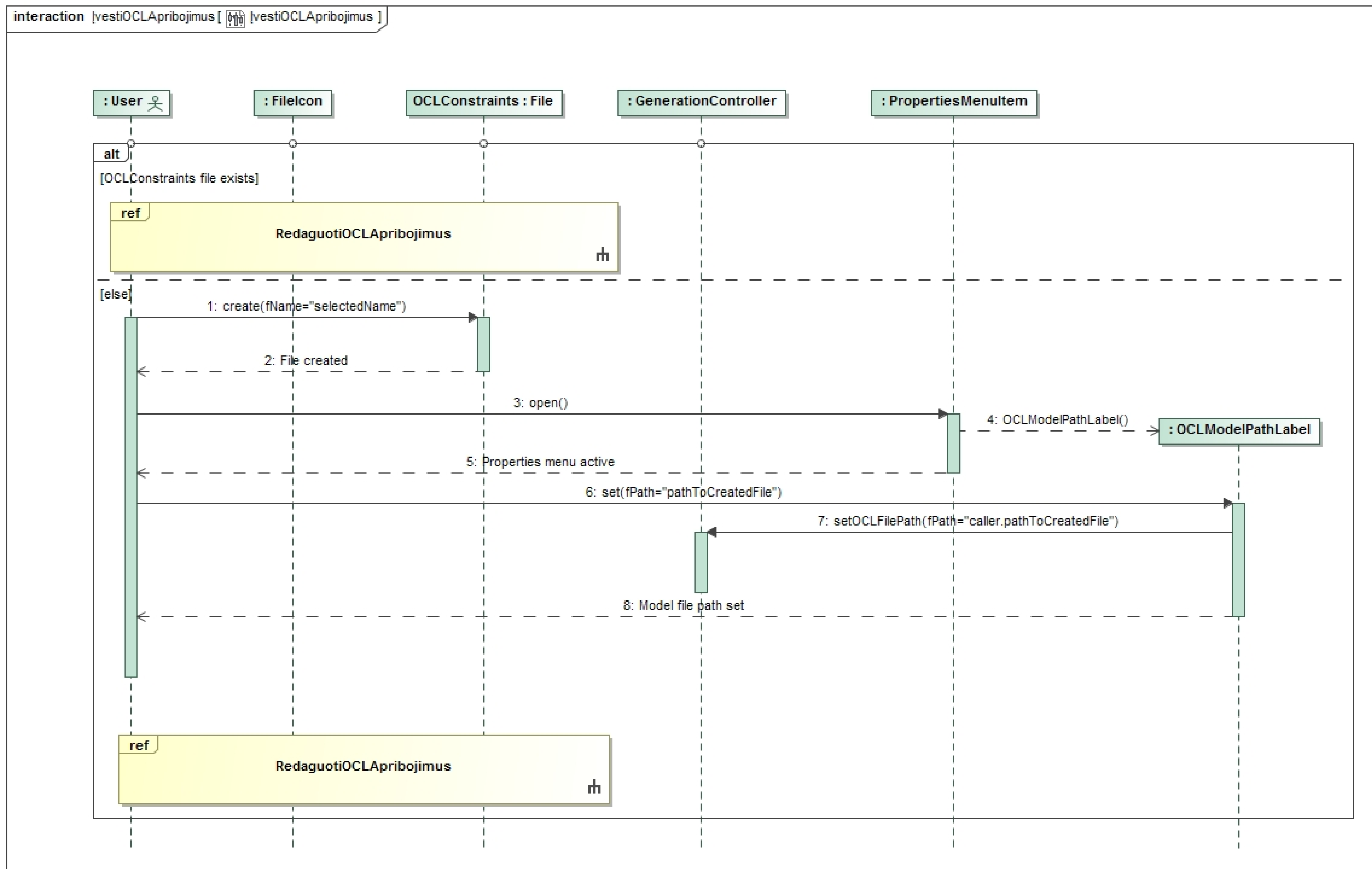



**18 pav.** Sekų diagrama: Testų generavimas metodui



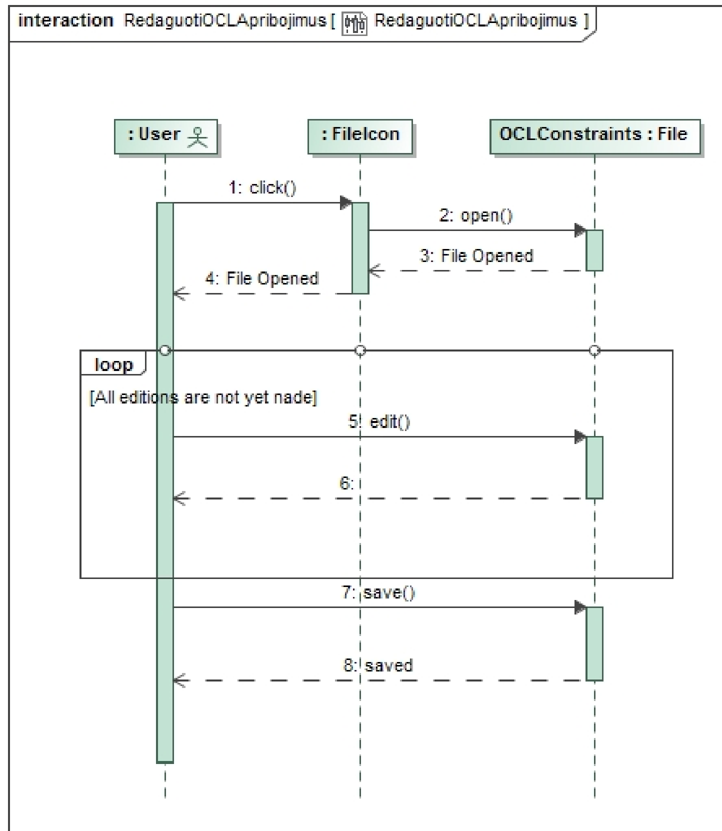

**19 pav.** Bendradarbiavimo diagrama: Testų generavimas metodui

### 3.5.2. OCL apribojimų įvedimas



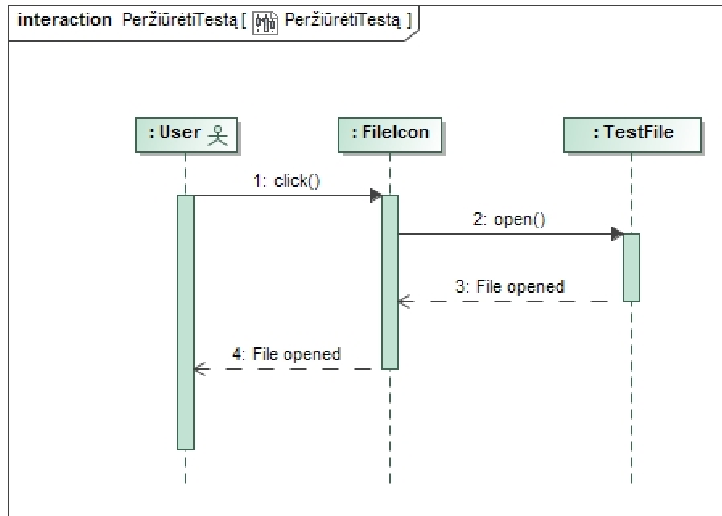

**20 pav.** Sekų diagrama: OCL apribojimų įvedimas

### 3.5.3. OCL apribojimų modifikavimas



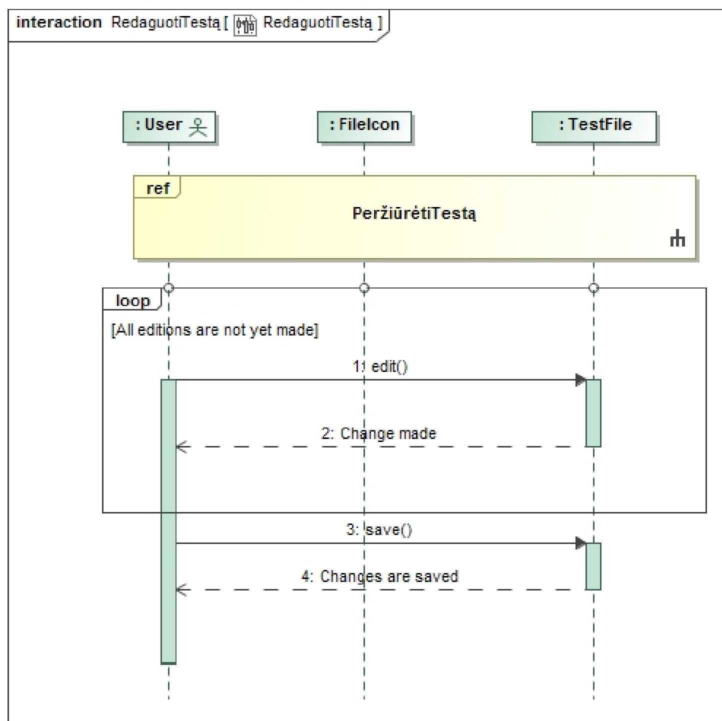

**21 pav.** Sekų diagrama: OCL apribojimų modifikavimas

### 3.5.4. Sugeneruotų testų peržiūra



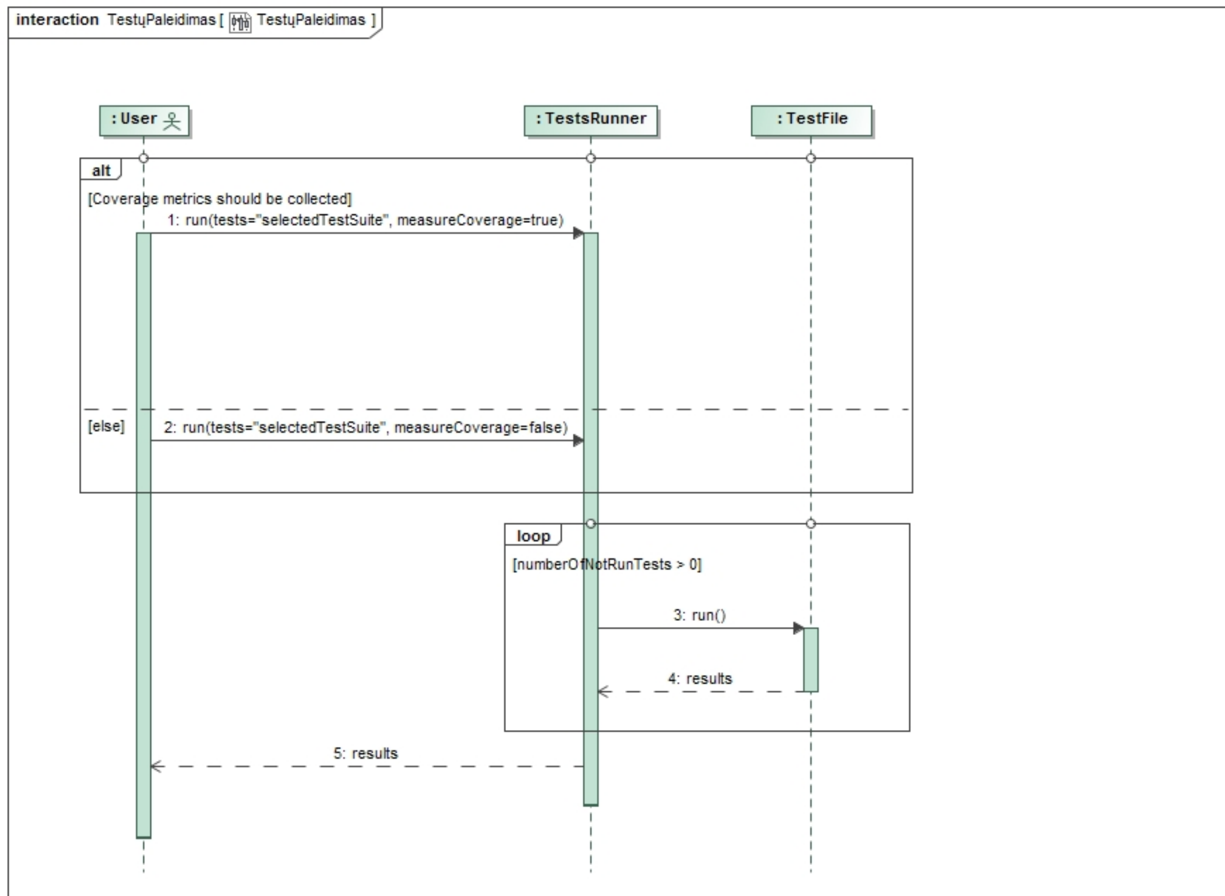

**22 pav.** Sekų diagrama: Sugeneruotų testų peržiūra

### 3.5.5. Sugeneruotų testų modifikavimas



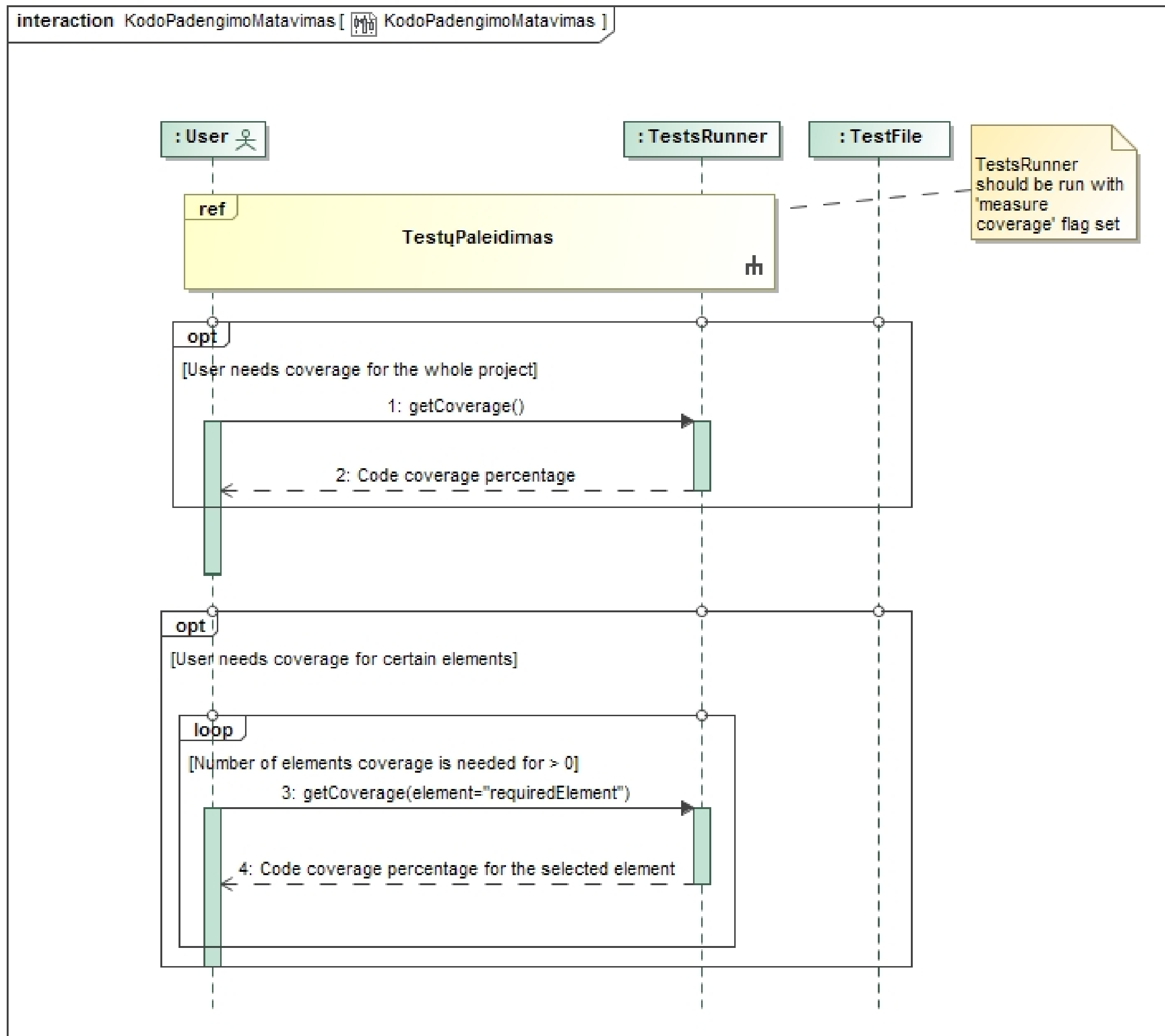

**23 pav.** Sekų diagrama: Sugeneruotų testų modifikavimas

### 3.5.6. Testų paleidimas




**24 pav.** Sekų diagrama: Sugeneruotų testų paleidimas

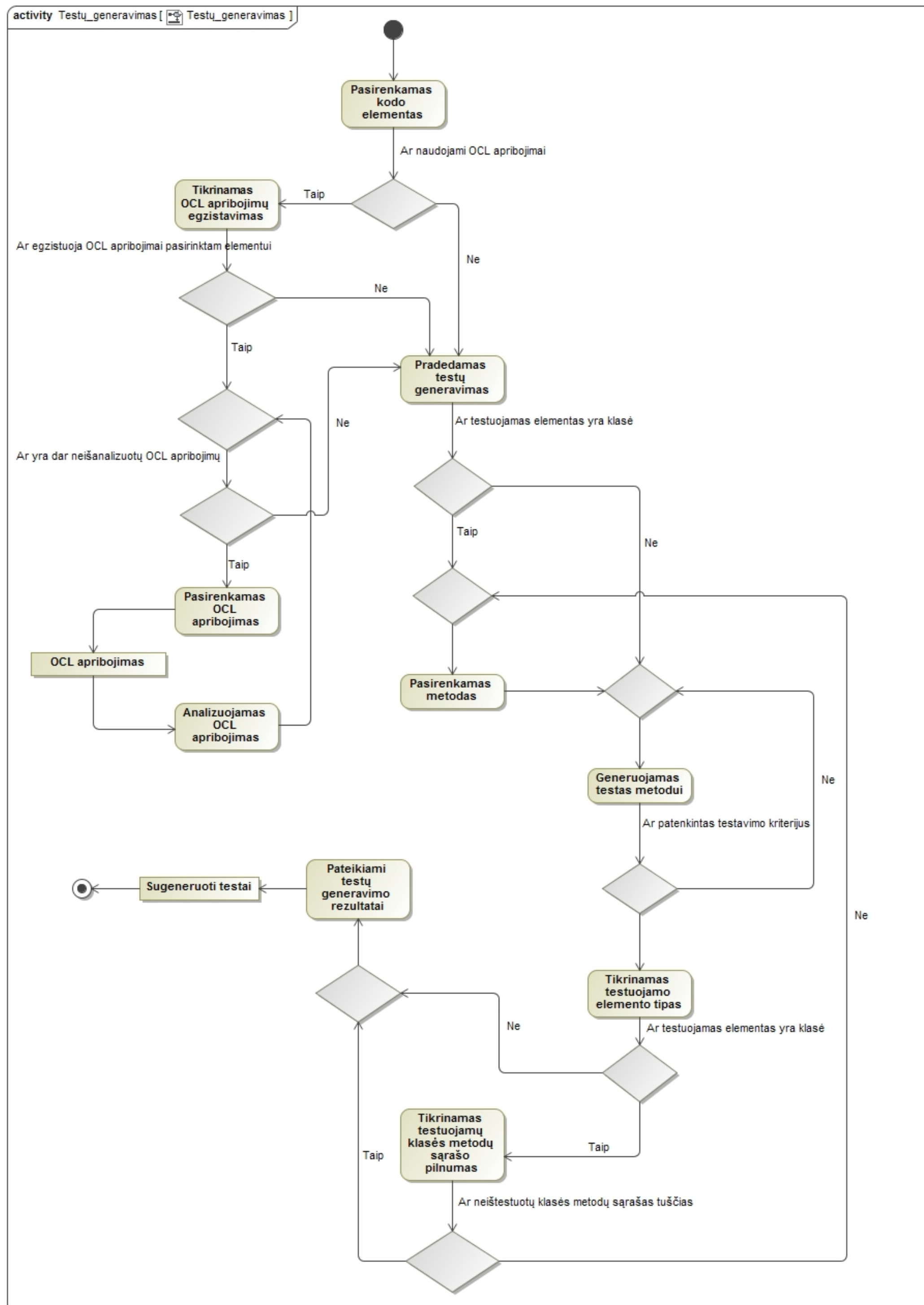
### 3.5.7. Kodo padengimo matavimas



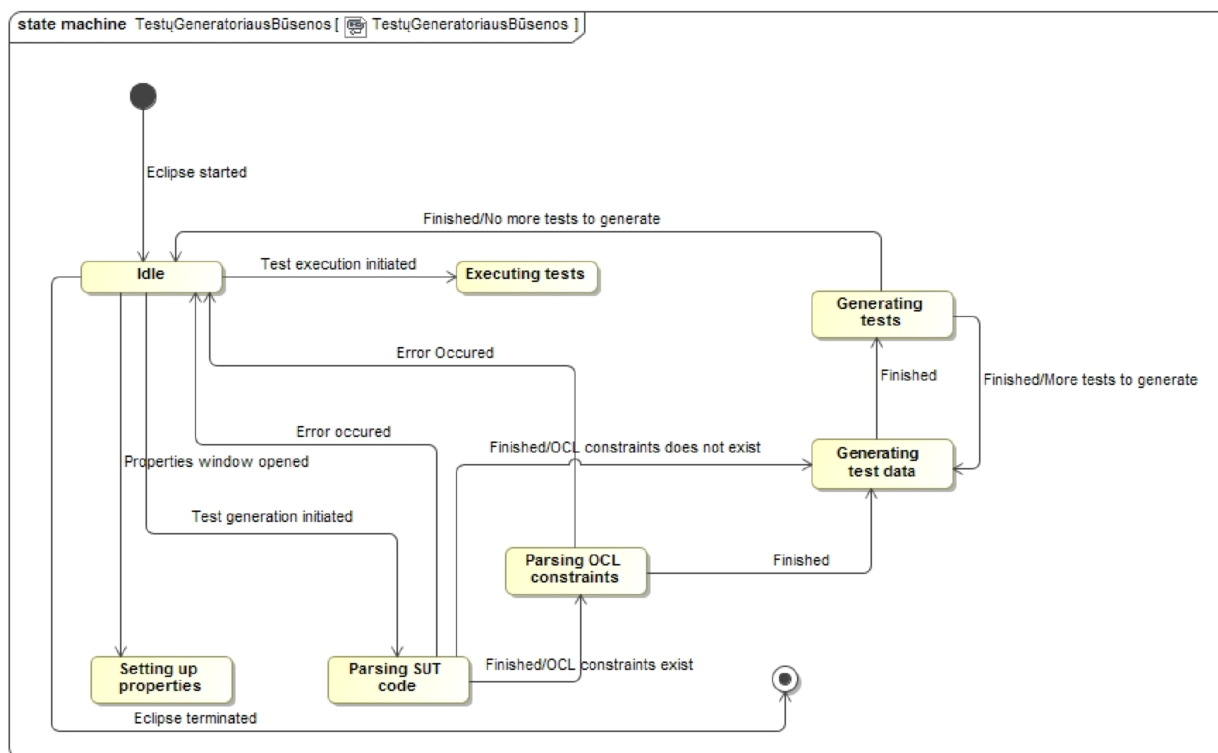

**25 pav.** Sekų diagrama: Kodo padengimo matavimas

### 3.5.8. Veiklos ir būsenų diagramos

Siekiant sudaryti pilnesnį sistemos dinaminį vaizdą, į architektūros specifikaciją yra įtrauktos bendrą sistemos veikimą iliustruojančios veiklos ir būsenų diagramos:




**26 pav.** Veiklos diagrama



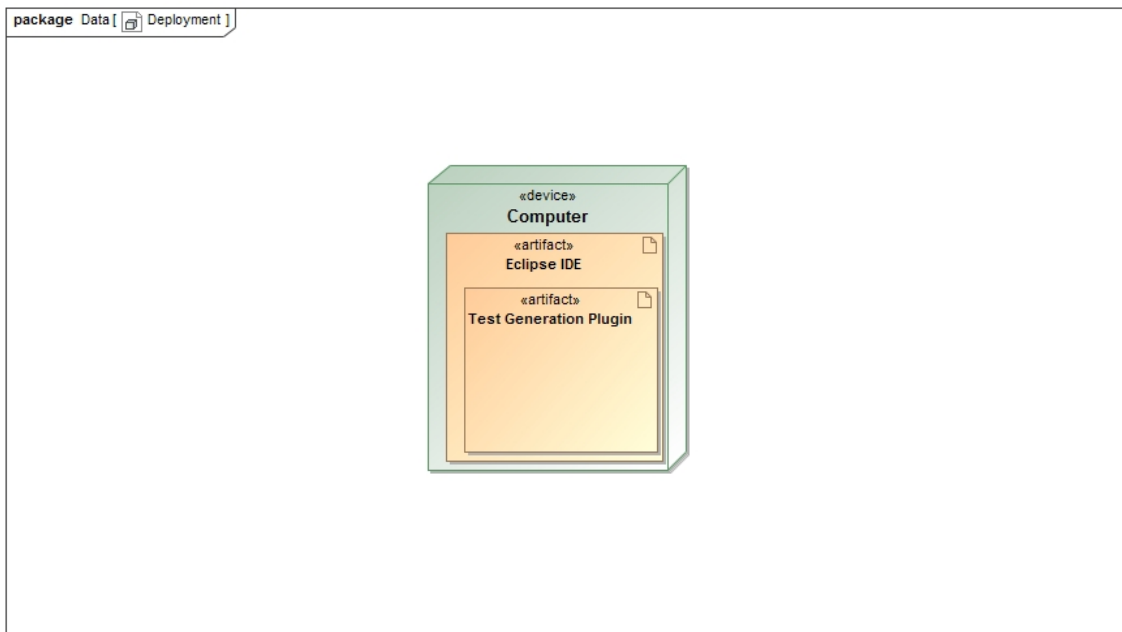

**27 pav.** Būsenų diagrama

### 3.6. Išdėstymo (deployment) vaizdas

Kadangi magistrinio projekto „Automatinių testų kūrimo įrankio naudojančio modeliu paremtą testavimą kūrimas ir tyrimas“ metu kuriamas produktas yra „Eclipse IDE“ įskiepis, reikalavimai techninei įrangai atitinka šios programavimo aplinkos keliamus reikalavimus:

- Rekomenduojama 2 GB RAM (tačiau gali veikti ir su 1GB);
- „Windows“, „Mac OS X“ arba „Linux OS“;
- Bent 512 MB laisvos vietos diske;

Kadangi įrankis veiks viename kompiuteryje, sistemos išdėstymo fiziniuose mazguose diagrama (žr. 28 pav.) – triviali:




**28 pav.** Sistemos išdėstymo diagrama

## 3.7. Kokybė

Pasirinkta architektūra užtikrina aukštą sistemos išplečiamumą bei leidžia lengvai atlikti pakeitimus ir papildymus, kadangi dauguma loginių ryšių tarp klasių yra realizuojami sąsajų (angl. interfaces) lygmenyje. Jei kurį nors komponentą reikia pakeisti, tereikia užtikrinti, kad naujasis elementas tenkintų esamą sąsają.

Testų generavimo algoritmas yra realizuotas strategijos projektavimo šablono principu, todėl papildyti įrankį naujomis generavimo taisyklėmis yra labai nesudėtinga.

Taip pat programos logika yra atskirta nuo vartotojo sąsajos. Testų generatoriaus rezultatai yra pateikiami per GenerationController klasės sąsają, vartotojo sąsajos komponentai su testų generatoriumi bendrauja tik per šią klasę (façade projektavimo šablonas). Šis architektūrinis sprendimas leidžia nesunkiai perkelti įrankį į kitas platformas, nekeičiant programos logikos.

# 4. TYRIMO DALIS

Šiame skyrelyje pateikiama OCL apribojimais pagrįsto testų generatoriaus veikimo analizė bei jos rezultatais paremti pasiūlymai tolesniam įrankio tobulinimui.

## 4.1. Tyrimo aprašas

### 4.1.1. Tiriami algoritmai

Projekto metu buvo realizuoti du testų generavimo algoritmai:

1. Atsitiktinis generavimas;
2. OCL apribojimais grįstas generavimas.

Tyrimo metu vertinami bei palyginami testai sugeneruoti tai pačiai programų aibei, naudojant kiekvieną iš šių algoritmų.

### 4.1.2. Tiriamos charakteristikos

Tyrimo tikslas – įvertinti įrankio generuojamų testų kokybę, remiantis testuojamos programos kodo padengimo laipsniu. 15 lentelėje pateikiamas šiame tyrime naudojamų kodo padengimo metrikų sąrašas bei kiekvienos iš jų apibūdinimas:

**15 lentelė.** Tiriamos charakteristikos

| Pavadinimas | Aprašas |
|---|---|
| Kodo eilučių padengimas | Parodo kokia programos kodo eilučių dalis buvo įvykdyta atliekant testus. Vertinant šią metriką, komentarai bei tuščios eilutės – neįskaičiuojamos |
| Šakų padengimas | Įvertina kokia dalis galimų programos vykdymo šakų įvykdoma. Šios metrikos 100 % rodiklis gaunamas, jei vykdant testus, visų sąlygos sakinių kiekviena baigtis yra bent kartą išbandoma (pvz. „if“ sakinio atveju, turėtų būti išbandoma ir „if“, ir „else“ dalis).[29] |
| Komandų padengimas | Nusako kokia dalis baitinio kodo (angl. bytecode) komandų yra įvykdoma. |
| Metodų padengimas | Nustato testų vykdymo metu iškviestų metodų skaičiaus santykį su bendru testuojamos sistemos metodų kiekiu. |
| Vykdymo kelių padengimas | Parodo kokia dalis galimų programos vykdymo kelių yra išbandomi testų rinkiniu |

### 4.1.3. Tyrimui naudojama programinė įranga

Tyrimui naudojami šie programinės įrangos paketai:

- „Eclipse“ programavimo aplinka (versija: 4.6.1, „Neon.1a Release“);
- „JaCoCo“ kodo padengimo matavimo biblioteka (integruota į tiriamą įrankį) [30];
- „Junit 4“ karkasas testų paleidimui;

### 4.1.4. Tyrimo vykdymo aplinka

Tyrimas atliktas naudojant kompiuterį turintį šias charakteristikas:

- Procesorius: „Intel Core i7-6700HQ CPU“, 2.6 GHz dažnis, 4 branduoliai, 8 loginiai

procesoriai;

- Atmintis: 8 GB RAM;
- OS: „Microsoft Windows 10 Home“;
- Vaizdo plokštė: „GeForce GTX 960“, 2 GB VRAM

### 4.1.5. Tyrime testuojama programinė įranga

Kadangi įrankio palaikomų OCL apribojimų aibė yra ribota, tyrime kaip įvedimo duomenys naudojama programinė įranga parinkta siekiant užtikrinti, kad didžiąją dalį jos kodo būtų galima padengti OCL apribojimais, palaikomais šio darbo metu sukurto testų generavimo įrankio. 16 lentelėje pateikiamas tyrimo metu testuojamos programinės įrangos aprašas:

**16 lentelė.** Testuojama programinė įranga

| Pavadinimas | Aprašas | Apimtis (kodo eilutėmis) |
|---|---|---|
| **Basic Calculator** | Skaičiuotuvas, skirtas paprastiems aritmetiniams veiksmams (sudėtis, atimtis, daugyba ir dalyba) atlikti. | 46 |
| **LT Converter** | Programinė įranga, pateikianti lietuvišką skaičių bei loginių kintamųjų išraišką (pvz. 66 – šešiasdešimt šeši, arba 11,35 – vienuolika ir trisdešimt penkios šimtosios). | 203 |
| **Geometry Calculator** | Skaičiuotuvas įvairių figūrų plotų, perimetrų, tūrių ar paviršiaus ploto skaičiavimui. | 133 |
| **CL Tic Tac Toe** | Žaidimo „Tic Tac Toe“ (taip pat žinomo „Kryžiukų – nuliukų“ pavadinimu) realizacija dviems žaidėjams, komandinėje eilutėje. | 217 |

Tyrime naudoti OCL apribojimai yra pateikiami prieduose (9.1. skyrelis, „OCL apribojimai naudoti eksperimente“).

### 4.1.6. Tyrimo eiga

Tyrimo vykdymo eigą galima apibrėžti tokiu algoritmu:

1. Pasirenkama testuojama programinė įranga (žr. 16 lentelę);
2. 1-ame žingsnyje pasirinktai PĮ sudaromas OCL apribojimų failas;
3. Pasirenkamas tiriamas algoritmas;
4. Paleidžiamas testų generavimo įrankis, 3-čiame žingsnyje pasirinktu algoritmu sugeneruojantis, 1-ame žingsnyje nustatytai PĮ, testinių atvejų rinkinį;
5. Pabaigus testų generavimą, įrankis automatiškai pateikia testuojamos programinės įrangos

kodo padengimo, 4-ame žingsnyje sugeneruotais testais, rodiklius (žr. 15 lentelę);

6. Kartojami 4-5 žingsniai, su įvairiais sugeneruotų testų kiekiais (1, 10, 100) ;
7. Rezultatai fiksuojami ir analizuojami;
8. Kartojami 3-7 žingsniai, kol ištiriami visi reikalingi algoritmai;
9. Kartojami 1-8 žingsniai, kol tyrimas atliekamas su visais testuojamos programinės įrangos paketais;

## 4.2. Tyrimo rezultatai

Šiame skyrelyje pateikiami anksčiau (žr. 4.1. Tyrimo aprašas) aprašyti tyrimo rezultatai bei jų analizė.

### 4.2.1. Kodo padengimo tyrimas

#### *4.2.1.1. Basic Calculator*

17 lentelėje pateikiami „Basic Calculator“ programos padengimo tyrimo rezultatai:

**17 lentelė.** „Basic Calculator“ programos kodo padengimas

| Algoritmas | Atsitiktinis generavimas | | | OCL apribojimais pagrįstas generavimas | | |
|---|---|---|---|---|---|---|
| **Metrika** | | | | | | |
| **Generavimo ciklų skaičius** | 1 | 10 | 100 | 1 | 10 | 100 |
| **Sugeneruotų testų skaičius** | 9 | 90 | 900 | 19 | 190 | 1900 |
| **Kodo eilučių padengimas** | 80 % | 80 % | 88 % | 88 % | 100 % | 100 % |
| **Šakų padengimas** | 50 % | 50 % | 75 % | 75 % | 100 % | 100 % |
| **Komandų padengimas** | 83,33 % | 83,33 % | 91,67 % | 91,67 % | 100 % | 100 % |
| **Metodų padengimas**[1] | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % |
| **Vykdymo kelių padengimas** | 81,8 % | 81,8 % | 90,9 % | 90,9 % | 100 % | 100 % |
| **Vidutinis kodo padengimo laipsnis** | 79,03 % | 79,03 % | 89,11 % | 89,11 % | 100 % | 100 % |

Iš lentelėje pateiktų duomenų matome, kad šiai nesudėtingai programai net ir atsitiktinio generavimo algoritmas leidžia pasiekti pakankamai aukštą (~80 %) kodo padengimo laipsnį. Didesnio testų kiekio naudojimas užtikrina iki 90 % kodo padengimo rodiklius. Tuo tarpu, taikant OCL apribojimais grįstą generavimą, su pakankamų generuojamų testų kiekiu, galima pasiekti ir 100 procentų kodo padengimą.

#### *4.2.1.2. LT Converter*

18 lentelėje matome kodo padengimo tyrimo rezultatus, naudojant „LT Converter“ programą kaip testuojamą sistemą:

1 100 % metodų padengimo metrikos rezultatus lemia naudojamas testų generavimo algoritmas, nustatantis visus išeities kodo metodus ir generuojantis testus kiekvienam iš jų atskirai.

**18 lentelė.** „LT Converter“ programos kodo padengimas

| Algoritmas | Atsitiktinis generavimas | | | OCL apribojimais pagrįstas generavimas | | |
|---|---|---|---|---|---|---|
| **Metrika** | | | | | | |
| **Generavimo ciklų skaičius** | 1 | 10 | 100 | 1 | 10 | 100 |
| **Sugeneruotų testų skaičius** | 4 | 40 | 400 | 6 | 60 | 600 |
| **Kodo eilučių padengimas** | 41,75 % | 64,07 % | 95,15 % | 40,77 % | 65,05 % | 93,2 % |
| **Šakų padengimas** | 24,44 % | 52,22 % | 90 % | 26,67 % | 52,22 % | 87,78 % |
| **Komandų padengimas** | 42,98 % | 63,91 % | 96,78 % | 45,75 % | 69,66 % | 95,86 % |
| **Metodų padengimas[1]** | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % |
| **Vykdymo kelių padengimas** | 20,48 % | 49,4 % | 90,36 % | 21,68 % | 50,6 % | 87,95 % |
| **Vidutinis kodo padengimo laipsnis** | 45,93 % | 65,92 % | 94,46 % | 46,98 % | 67,51 % | 92,96 % |

Matome, kad šiuo atveju pagrindinę įtaką padengimo rodikliams turėjo generuojamų testų kiekis. Nepastebimas esminis skirtumas tarp atsitiktinio generavimo ir OCL apribojimais grįsto generavimo rezultatų. To priežastis – „LT Converter“ programai nebuvo įmanoma sudaryti kokybiško, į kodo padengimą orientuoto OCL apribojimų rinkinio, naudojant įrankio palaikomą OCL apribojimų aibę. Todėl didžioji dalis OCL apribojimais pagrįsto algoritmo sugeneruotų testų buvo paremti ne OCL apribojimais, o alternatyviu algoritmu, taikomu nesant apribojimų, šiuo atveju: atsitiktiniu generavimu.

#### *4.2.1.3. Geometry Calculator*

19 lentelėje pateikiami tyrimo, naudojant „Geometry Calculator“ kaip SUT, rezultatai. Kaip ir „LT Converter“ atveju, pastebima ryški koreliacija tarp generuojamų testų kiekio ir kodo padengimo rodiklių. Šiuo atveju, matomas ir ženklus OCL apribojimais pagrįsto generavimo pranašumas.

**19 lentelė.** „Geometry Calculator“ programos kodo padengimas

| Algoritmas | Atsitiktinis generavimas | | | OCL apribojimais pagrįstas generavimas | | |
|---|---|---|---|---|---|---|
| **Metrika** | | | | | | |
| **Generavimo ciklų skaičius** | 1 | 10 | 100 | 1 | 10 | 100 |
| **Sugeneruotų testų skaičius** | 20 | 200 | 2000 | 116 | 1160 | 11600 |
| **Kodo eilučių padengimas** | 67,21 % | 91,8 % | 100 % | 93,44 % | 100 % | 100 % |
| **Šakų padengimas** | 31,58 % | 88,16 % | 100 % | 89,47 % | 100 % | 100 % |
| **Komandų padengimas** | 50,42 % | 87,74 % | 100 % | 94,98 % | 100 % | 100 % |

| Algoritmas | Atsitiktinis generavimas | | | OCL apribojimais pagrįstas generavimas | | |
|---|---|---|---|---|---|---|
| **Metrika** | | | | | | |
| **Metodų padengimas[1]** | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % |
| **Vykdymo kelių padengimas** | 35,59 % | 86,64 % | 100 % | 86,44 % | 100 % | 100 % |
| **Vidutinis kodo padengimo laipsnis** | 56,96 % | 90,83 % | 100 % | 92,87 % | 100 % | 100 % |

#### *4.2.1.4. CL Tic Tac Toe*

Atlikus testų generavimo „CL Tic Tac Toe“ programai tyrimą (rezultatai 20-oje lentelėje), pastebėta, kad šiai programai atsitiktinio generavimo metodas neefektyvus – kodo padengimas nesiekia 40 %. Kadangi šioje programoje yra privačių metodų, kurie kviečiami tik specifinėmis aplinkybėmis, atsitiktinio generavimo metodas negeba jų pasiekti (tą matome iš metodų padengimo rodiklio, kuris yra 60 %). Tuo tarpu aprašius šias sąlygas OCL apribojimais, pasiekiamas kur kas aukštesnis kodo padengimas.

**20 lentelė.** „CL Tic Tac Toe“ programos kodo padengimas

| Algoritmas | Atsitiktinis generavimas | | | OCL apribojimais pagrįstas generavimas | | |
|---|---|---|---|---|---|---|
| **Metrika** | | | | | | |
| **Generavimo ciklų skaičius** | 1 | 10 | 100 | 1 | 10 | 100 |
| **Sugeneruotų testų skaičius** | 14 | 140 | 1400 | 33 | 330 | 3300 |
| **Kodo eilučių padengimas** | 33,33 % | 34,88 % | 37,21 % | 38,76 % | 68,99 % | 73,64 % |
| **Šakų padengimas** | 22,45 % | 26,53 % | 28,57 % | 29,59 % | 55,1 % | 61,22 % |
| **Komandų padengimas** | 33,59 % | 34,75 % | 37,86 % | 39,81 % | 70,87 % | 76,7 % |
| **Metodų padengimas** | 60 % | 60 % | 60 % | 65 % | 100 % | 100 % |
| **Vykdymo kelių padengimas** | 28,57 % | 32,86 % | 34,28 % | 35,71 % | 52,86 % | 55,7 % |
| **Vidutinis kodo padengimo laipsnis** | 35,59 % | 37,81 % | 39,59 % | 41,77 % | 69,57 % | 73,46 % |

### 4.2.2. Rezultatų apibendrinimas

Atliekant darbo metu sukurto OCL apribojimais pagrįsto automatinio testų generavimo įrankio tyrimą, pastebėti tokie dėsningumai:

- Nesudėtingoms programoms atsitiktinio testų generavimo algoritmas, su pakankamu generuojamų testų kiekiu, gali pasiekti gerus (kai kuriais atvejais iki 100 %) kodo padengimo rodiklius;
- OCL apribojimais pagrįstas algoritmas dažniausiai leidžia gauti geresnius rezultatus nei

atsitiktinis generavimas, išskyrus atvejus, kai OCL apribojimai nepakankami;

- OCL apribojimais pagrįsto algoritmo efektyvumas priklauso ne tik nuo testuojamos programos, bet ir jai sudaryto OCL modelio;
- Testų skaičiaus įtaka testuojamos programos kodo padengimui priklauso nuo testuojamos programos struktūros (pvz., atsitiktinio generavimo „CL Tic Tac Toe“ programai atveju, padidinus testų generavimo iteracijų skaičių 100 kartų, kodo padengimo rodiklis išaugo tik ~1,11 karto, tuo tarpu toks pats testavimo iteracijų kiekio augimas „LT Converter“ sistemai, kodo padengimą padidino ~2,06 karto);

Analizuojant tyrimo rezultatus taipogi reikėtų atsižvelgti į tai, jog OCL apribojimais pagrįstas algoritmas reikalauja papildomų pastangų (reikia testuojamai programai sudaryti OCL modelį), tačiau jo sugeneruoti testai apribojimus gali naudoti kaip testų orakulą. Tuo tarpu atsitiktinio generavimo algoritmas orakulo problemos nesprendžia (norint tokiu būdu sugeneruotus testus naudoti, „assert“ sakinius tektų parašyti pačiam testuotojui). Todėl būtina pažymėti, kad šių algoritmų palyginimas vien kodo padengimo atžvilgiu nėra laikytinas pakankamu, norint teigti, jog vienas yra objektyviai pranašesnis už kitą.

#### 4.2.3. Siūlomi patobulinimai

Atsižvelgiant į tyrimo metu surinktą informaciją, nuspręsta atlikti tokius pakeitimus bei patobulinimus:

1. Remiantis pastebėjimu, kad daugeliu atvejų iteracijų skaičiaus pakėlimas nuo 1 iki 10 turi didesnę įtaką kodo padengimui, nei didinimas nuo 10 iki 100, tačiau atsižvelgiant ir į faktą, jog „LT Converter“ atveju testų skaičiaus pakėlimas intervale 10-100 buvo reikšmingas, nuspręsta padidinti įrankio naudojamą numatytąjį iteracijų skaičių nuo 10 iki 50.
2. Pastebėta, kad neretai negalima pasiekti aukšto kodo padengimo, remiantis vien OCL apribojimais pagrįstu generavimu (pvz., „LT Converter“ atveju), dėl ribotos palaikomų apribojimų aibės ar kodo struktūros. Remiantis šiuo pastebėjimu prieita išvados, kad algoritmas naudojamas generavimui, kai OCL apribojimai nėra pakankami, turi reikšmingos įtakos įrankio veikimui. Dėl šios priežasties, nuspręsta papildyti įrankio palaikomą algoritmų aibę „Hill climbing“ ir „Simulated annealing“ algoritmais;

# 5. EKSPERIMENTINĖ DALIS

Ankstesniame skyriuje aprašyto tyrimo metu nustatyta galima projekte sukurto įrankio automatiniam testų generavimui apribojimais tobulinimo kryptis, praplečiant jo palaikomų testų generavimo algoritmų aibę bei didinant generuojamų testų skaičių. Šiame skyriuje aprašomi atlikti eksperimentai, siekiant nustatyti ar pasiūlyti patobulinimai leido pagerinti įrankio rodomus kodo padengimo rezultatus.

## 5.1. Eksperimento aprašas

Eksperimentinėje dalyje aprašomo tyrimo eiga yra analogiška tyrimo dalyje aprašytajai (plačiau žr. 4.1. Tyrimo aprašas), todėl čia aprašomi tik skirtumai lyginant su tyrimo dalimi.

### 5.1.1. Tiriami algoritmai

Tiriamų algoritmų aibė papildyta dviem 4.2.3. skyrelyje paminėtais algoritmais:

- „Hill Climbing“;
- „Simulated annealing“.

### 5.1.2. Eksperimento eiga

Eksperimentas vykdomas pagal 4.1.6. „Tyrimo eiga“ skyrelyje aprašytą algoritmą. Vienintelis skirtumas – 6-asis žingsnis taip pat atliekamas su 50 generavimo iteracijų, siekiant nustatyti, ar pasiteisino tyrimo metu priimtas sprendimas padidinti numatytąjį iteracijų skaičių nuo 10 iki 50.

### 5.2. Eksperimento rezultatai

#### 5.2.1. Basic Calculator

21 ir 22 lentelėse pateikiami kodo padengimo eksperimento, generuojant „Basic Calculator“ klasei, rezultatai:

**21 lentelė.** Eksperimentas: „Basic Calculator“ padengimas (atsitiktinis generavimas ir OCL paremtas generavimas)

| Algoritmas | Atsitiktinis generavimas | | | | OCL apribojimais pagrįstas generavimas | | | |
|---|---|---|---|---|---|---|---|---|
| **Metrika** | | | | | | | | |
| **Generavimo ciklų skaičius** | 1 | 10 | 50 | 100 | 1 | 10 | 50 | 100 |
| **Sugeneruotų testų skaičius** | 9 | 90 | 450 | 900 | 28 | 280 | 1400 | 2800 |
| **Kodo eilučių padengimas** | 80 % | 80 % | 88 % | 88 % | 88 % | 100 % | 100 % | 100 % |
| **Šakų padengimas** | 50 % | 50 % | 75 % | 75 % | 75 % | 100 % | 100 % | 100 % |
| **Komandų padengimas** | 83,33 % | 83,33 % | 91,67 % | 91,67 % | 91,67 % | 100 % | 100 % | 100 % |
| **Metodų padengimas**[1] | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % |
| **Vykdymo kelių padengimas** | 81,8 % | 81,8 % | 90,9 % | 90,9 % | 90,9 % | 100 % | 100 % | 100 % |
| **Vidutinis kodo padengimo laipsnis** | 79,03 % | 79,03 % | 89,11 % | 89,11 % | 89,11 % | 100 % | 100 % | 100 % |

**22 lentelė.** Eksperimentas: „Basic Calculator“ padengimas („Hill Climbing“ ir „Simulated Annealing“)

| Algoritmas | Hill Climbing | | | | Simulated Annealing | | | |
|---|---|---|---|---|---|---|---|---|
| **Metrika** | | | | | | | | |
| **Generavimo ciklų skaičius** | 1 | 10 | 50 | 100 | 1 | 10 | 50 | 100 |
| **Sugeneruotų testų skaičius** | 9 | 90 | 450 | 900 | 9 | 90 | 450 | 900 |
| **Kodo eilučių padengimas** | 88 % | 88 % | 100 % | 100 % | 88 % | 88 % | 100 % | 100 % |
| **Šakų padengimas** | 75 % | 75 % | 100 % | 100 % | 75 % | 75 % | 100 % | 100 % |
| **Komandų padengimas** | 91,67 % | 91,67 % | 100 % | 100 % | 91,67 % | 91,67 % | 100 % | 100 % |

| Algoritmas | Hill Climbing | | | | Simulated Annealing | | | |
|---|---|---|---|---|---|---|---|---|
| **Metrika** | | | | | | | | |
| **Metodų padengimas**[1] | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % |
| **Vykdymo kelių padengimas** | 90,9 % | 90,9 % | 100 % | 100 % | 90,9 % | 90,9 % | 100 % | 100 % |
| **Vidutinis kodo padengimo laipsnis** | 89,11 % | 89,11 % | 100 % | 100 % | 89,11 % | 89,11 % | 100 % | 100 % |

Matome, kad „Hill Climbing“ ir „Simulated Annealing“ algoritmai, skirtingai nei atsitiktinis generavimas, leidžia pasiekti 100 % kodo padengimą šiai klasei, naudojant 50 ir daugiau testų generavimo iteracijų. OCL apribojimais pagrįstas generavimas vis dar yra efektyviausias, tačiau vienos iteracijos 100 % kodo padengimui pasiekti nepakanka.

Todėl galime teigti, jog atlikti patobulinimai tiesiogiai kodo padengimo rodiklių šiai programai nepagerino. Tačiau remiantis faktu, kad „Hill Climbing“ bei „Simulated Annealing“ algoritmai leido pasiekti geresnius kodo padengimo rodiklius, lyginant su atsitiktiniu generavimu, naudojant tą patį testų generavimo iteracijų skaičių, galima daryti prielaidą, jog bendro sugeneruotų testų paketo kokybė (kodo padengimo atžvilgiu) yra aukštesnė.

### 5.2.2. LT Converter

23 ir 24 lentelėse pateikti eksperimentinio testų generavimo „LT Converter“ programai rezultatai. Matome, kad šios programos kodo padengimo rezultatų priklausomybės nuo testų skaičiaus tendencija išlieka ir naudojant „Hill Climbing“ bei „Simulated Annealing“ algoritmus. Pastebėta, kad pastarojo padengimo rodikliai auga greičiau, nei atsitiktinio generavimo ar „Hill Climbing“ atveju, tačiau pasiekus 90 %, toliau beveik nebekinta. Taip pat rezultatai rodo, kad kodo padengimo augimo skirtumas tarp generavimo naudojant 50 ir 100 iteracijų yra nežymus (neviršija 8 %) visų algoritmų atveju, todėl galima teigti, kad šiai programai, 50-ties iteracijų generavimas yra optimalus sprendimas.

**23 lentelė.** Eksperimentas: „LT Converter“ padengimas (atsitiktinis generavimas ir OCL paremtas generavimas)

| Algoritmas | Atsitiktinis generavimas | | | | OCL apribojimais pagrįstas generavimas | | | |
|---|---|---|---|---|---|---|---|---|
| **Metrika** | | | | | | | | |
| **Generavimo ciklų skaičius** | 1 | 10 | 50 | 100 | 1 | 10 | 50 | 100 |

| Algoritmas | Atsitiktinis generavimas | | | | OCL apribojimais pagrįstas generavimas | | | |
|---|---|---|---|---|---|---|---|---|
| **Metrika** | | | | | | | | |
| **Sugeneruotų testų skaičius** | 4 | 40 | 200 | 400 | 14 | 140 | 700 | 1400 |
| **Kodo eilučių padengimas** | 41,75 % | 64,07 % | 87,38 % | 95,15 % | 57,28 % | 86,41 % | 94,17 % | 95,15 % |
| **Šakų padengimas** | 24,44 % | 52,22 % | 80 % | 90 % | 43,33 % | 78,89 % | 88,89 % | 90 % |
| **Komandų padengimas** | 42,98 % | 63,91 % | 88,5 % | 96,78 % | 60,23 % | 90,57 % | 94,25 % | 96,78 % |
| **Metodų padengimas**[1] | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % |
| **Vykdymo kelių padengimas** | 20,48 % | 49,4 % | 79,51 % | 90,36 % | 40,96 % | 78,31 % | 89,16 % | 90,36 % |
| **Vidutinis kodo padengimo laipsnis** | 45,93 % | 65,92 % | 87,08 % | 94,46 % | 60,36 % | 86,83 % | 93,3 % | 94,46 % |

**24 lentelė.** Eksperimentas: „LT Converter“ padengimas („Hill Climbing“ ir „Simulated Annealing“)

| Algoritmas | Hill Climbing | | | | Simulated Annealing | | | |
|---|---|---|---|---|---|---|---|---|
| **Metrika** | | | | | | | | |
| **Generavimo ciklų skaičius** | 1 | 10 | 50 | 100 | 1 | 10 | 50 | 100 |
| **Sugeneruotų testų skaičius** | 4 | 40 | 200 | 400 | 4 | 40 | 200 | 400 |
| **Kodo eilučių padengimas** | 34,95 % | 65,05 % | 88,35 % | 92,23 % | 43,69 % | 82,52 % | 91,26 % | 91,26 % |
| **Šakų padengimas** | 17,78 % | 52,23 % | 81,11 % | 86,67 % | 26,67 % | 72,22 % | 84,44 % | 85,56 % |
| **Komandų padengimas** | 35,63 % | 65,78 % | 93,56 % | 95,40 % | 45,97 % | 85,98 % | 94,94 % | 92,87 % |
| **Metodų padengimas** | 90 % | 100 % | 100 % | 100 % | 90 % | 100 % | 100 % | 100 % |
| **Vykdymo kelių padengimas** | 14,46 % | 50,6 % | 80,72 % | 86,75 % | 21,89 % | 71,08 % | 84,34 % | 85,54 % |
| **Vidutinis kodo padengimo laipsnis** | 38,56 % | 66,72 % | 88,75 % | 92,21 % | 45,6 % | 82,36 % | 91 % | 91,04 % |

### 5.2.3. Geometry Calculator

„Geometry Calculator“ klasei „Hill Climbing“ ir „Simulated Annealing“ algoritmų naudojimas leido pasiekti 100 % kodo padengimą, naudojant 50 generavimo iteracijų, tuo tarpu atsitiktinio generavimo atveju, prireikė – 100. Tiesa, kaip rodo 25, 26 lentelėse pateikti duomenys, skirtumas tarp atsitiktinio generavimo ir vietinės paieškos algoritmų yra labai nežymus.

**25 lentelė.** Eksperimentas: „Geometry Calculator“ padengimas (atsitiktinis generavimas ir OCL paremtas generavimas)

| Algoritmas | Atsitiktinis generavimas | | | | OCL apribojimais pagrįstas generavimas | | | |
|---|---|---|---|---|---|---|---|---|
| **Metrika** | | | | | | | | |
| **Generavimo ciklų skaičius** | 1 | 10 | 50 | 100 | 1 | 10 | 50 | 100 |
| **Sugeneruotų testų skaičius** | 20 | 200 | 1000 | 2000 | 116 | 1160 | 5800 | 11600 |
| **Kodo eilučių padengimas** | 67,21 % | 91,8 % | 98,36 % | 100 % | 93,44 % | 100 % | 100 % | 100 % |
| **Šakų padengimas** | 31,58 % | 88,16 % | 98,68 % | 100 % | 89,47 % | 100 % | 100 % | 100 % |
| **Komandų padengimas** | 50,42 % | 87,74 % | 94,99 % | 100 % | 94,98 % | 100 % | 100 % | 100 % |
| **Metodų padengimas[1]** | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % |
| **Vykdymo kelių padengimas** | 35,59 % | 86,64 % | 98,31 % | 100 % | 86,44 % | 100 % | 100 % | 100 % |
| **Vidutinis kodo padengimo laipsnis** | 56,96 % | 90,83 % | 98,07 % | 100 % | 92,87 % | 100 % | 100 % | 100 % |

**26 lentelė.** Eksperimentas: „Geometry Calculator“ padengimas („Hill Climbing“ ir „Simulated Annealing“)

| Algoritmas | Hill Climbing | | | | Simulated Annealing | | | |
|---|---|---|---|---|---|---|---|---|
| **Metrika** | | | | | | | | |
| **Generavimo ciklų skaičius** | 1 | 10 | 50 | 100 | 1 | 10 | 50 | 100 |
| **Sugeneruotų testų skaičius** | 20 | 200 | 1000 | 2000 | 20 | 200 | 1000 | 2000 |
| **Kodo eilučių padengimas** | 67,21 % | 91,8 % | 100 % | 100 % | 67,21 % | 90,16 % | 100 % | 100 % |
| **Šakų padengimas** | 30,26 % | 89,47 % | 100 % | 100 % | 32,89 % | 84,21 % | 100 % | 100 % |

| Algoritmas | Hill Climbing | | | | Simulated Annealing | | | |
|---|---|---|---|---|---|---|---|---|
| **Metrika** | | | | | | | | |
| **Komandų padengimas** | 50,97 % | 87,19 % | 100 % | 100 % | 52,09 % | 83,84 % | 100 % | 100 % |
| **Metodų padengimas**[1] | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % | 100 % |
| **Vykdymo kelių padengimas** | 35,59 % | 88,13 % | 100 % | 100 % | 35,59 % | 83,05 % | 100 % | 100 % |
| **Vidutinis kodo padengimo laipsnis** | 56,81 % | 91,32 % | 100 % | 100 % | 57,56 % | 88,25 % | 100 % | 100 % |

### 5.2.4. CL Tic Tac Toe

27 ir 28 lentelės apibendrina testų generavimo „CL Tic Tac Toe“ programai eksperimento rezultatus:

**27 lentelė.** Eksperimentas: „CL Tic Tac Toe“ padengimas (atsitiktinis generavimas ir OCL paremtas generavimas)

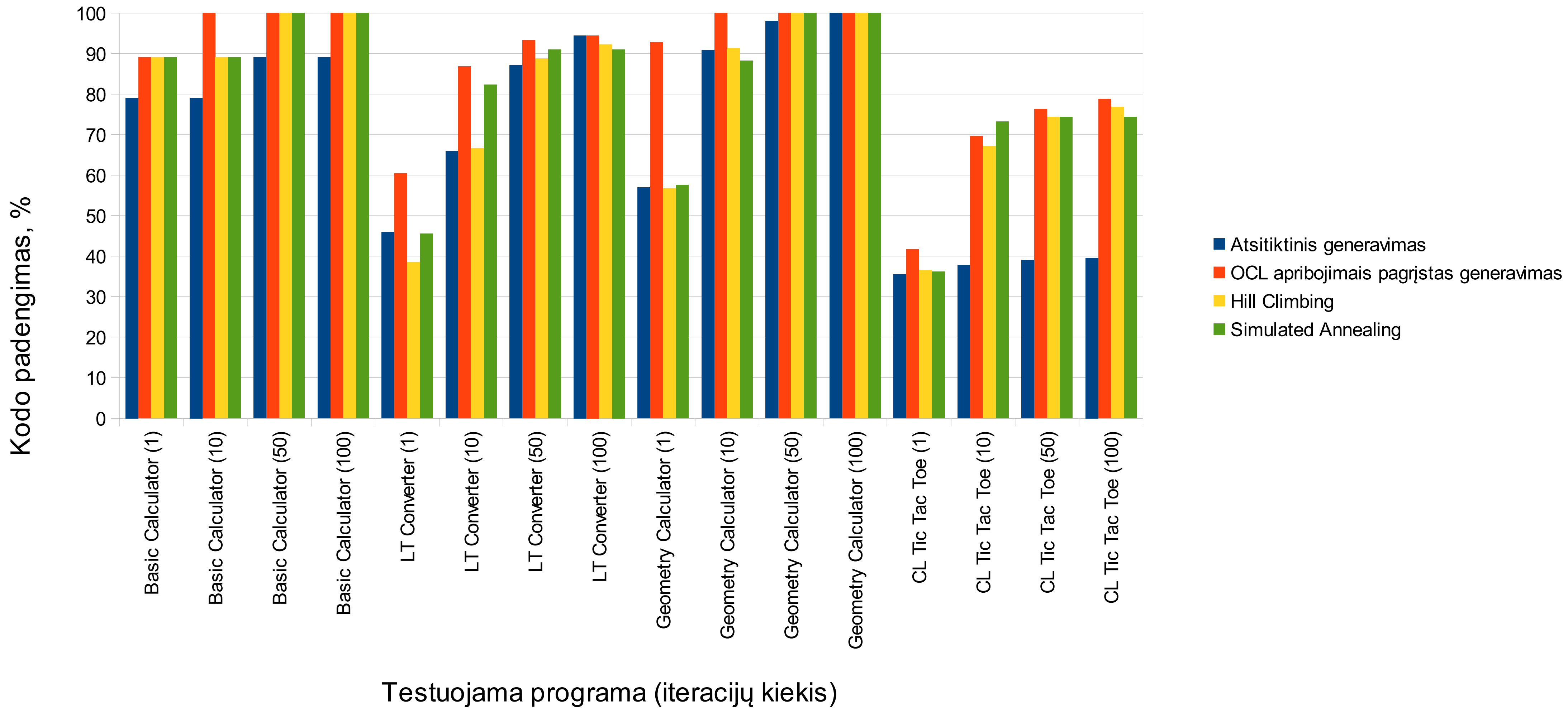
| Algoritmas | Atsitiktinis generavimas | | | | OCL apribojimais pagrįstas generavimas | | | |
|---|---|---|---|---|---|---|---|---|
| **Metrika** | | | | | | | | |
| **Generavimo ciklų skaičius** | 1 | 10 | 50 | 100 | 1 | 10 | 50 | 100 |
| **Sugeneruotų testų skaičius** | 14 | 140 | 700 | 1400 | 51 | 510 | 2550 | 5100 |
| **Kodo eilučių padengimas** | 33,33 % | 34,88 % | 37,21 % | 37,21 % | 38,76 % | 68,99 % | 75,97 % | 78,29 % |
| **Šakų padengimas** | 22,45 % | 26,53 % | 27,55 % | 28,57 % | 29,59 % | 55,1 % | 66,33 % | 70,4 % |
| **Komandų padengimas** | 33,59 % | 34,75 % | 37,86 % | 37,86 % | 39,81 % | 70,87 % | 79,42 % | 82,33 % |
| **Metodų padengimas**[1] | 60 % | 60 % | 60 % | 60 % | 65 % | 100 % | 100 % | 100 % |
| **Vykdymo kelių padengimas** | 28,57 % | 32,86 % | 32,86 % | 34,28 % | 35,71 % | 52,86 % | 60 % | 62,85 % |
| **Vidutinis kodo padengimo laipsnis** | 35,59 % | 37,81 % | 39,01 % | 39,59 % | 41,77 % | 69,57 % | 76,34 % | 78,78 % |

**28 lentelė.** Eksperimentas: „CL Tic Tac Toe“ padengimas („Hill Climbing“ ir „Simulated Annealing“)

| Algoritmas | Hill Climbing | | | | Simulated Annealing | | | |
|---|---|---|---|---|---|---|---|---|
| **Metrika** | | | | | | | | |
| **Generavimo ciklų skaičius** | 1 | 10 | 50 | 100 | 1 | 10 | 50 | 100 |
| **Sugeneruotų testų skaičius** | 9 | 90 | 450 | 900 | 9 | 90 | 450 | 900 |
| **Kodo eilučių padengimas** | 34,88 % | 67,44 % | 74,42 % | 76,74 % | 34,88 % | 73,64 % | 74,42 % | 74,42 % |
| **Šakų padengimas** | 23,47 % | 52,04 % | 64,29 % | 68,37 % | 22,45 % | 62,24 % | 64,28 % | 64,28 % |
| **Komandų padengimas** | 34,37 % | 67,57 % | 76,12 % | 79,03 % | 33,78 % | 75,92 % | 76,12 % | 76,12 % |
| **Metodų padengimas** | 60 % | 100 % | 100 % | 100 % | 60 % | 100 % | 100 % | 100 % |
| **Vykdymo kelių padengimas** | 30% | 48,57 % | 57,14 % | 60 % | 30 % | 54,29 % | 57,14 % | 57,14 % |
| **Vidutinis kodo padengimo laipsnis** | 36,54 % | 67,13 % | 74,39 % | 76,83 % | 36,22 % | 73,22 % | 74,39 % | 74,39 % |

Šio eksperimento metu pastebėtas aiškus vietinės paieškos algoritmų („Hill Climbing“ ir „Simulated Annealing“) pranašumas prieš atsitiktinio generavimo algoritmą: atliekant 10 ir daugiau iteracijų, matomas 25-30 % geresnis kodo padengimas.

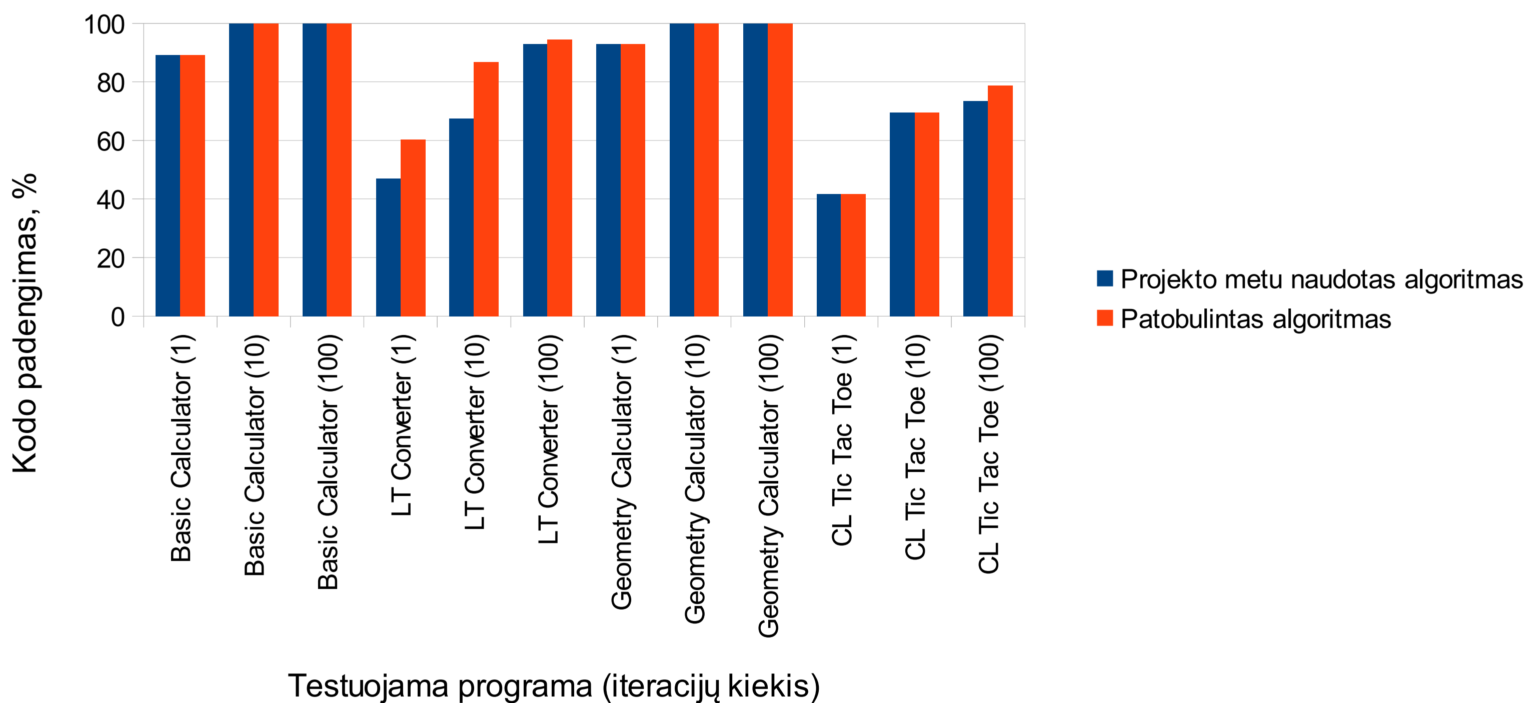
### 5.2.5. Eksperimento rezultatų analizė ir išvados




**29 pav.** Vidutinis kodo padengimas

29 pav. pavaizduotos eksperimento metu nustatytos vidutinės kodo padengimo reikšmės kiekvienai algoritmo, testuojamos programos bei testų iteracijų skaičiaus kombinacijai. Matome, kad atliekant testų generavimą „LT Converter“ bei „Geometry Calculator“ programoms, vietinės paieškos algoritmai neturi aiškaus pranašumo prieš atsitiktinį generavimą. Geriausiai „Hill Climbing“ ir „Simulated Annealing“ algoritmų privalumai prieš atsitiktinį generavimą atsiskleidžia generuojant testus „CL Tic Tac Toe“ programai, naudojant 10 ir daugiau iteracijų, kur jų rezultatai prilygsta patobulinto (naudojančio ne tik OCL apribojimais grįstą generavimą, bet ir atsitiktinį generavimą bei vietinės paieškos algoritmus), OCL apribojimais pagrįsto algoritmo rezultatams.

Vertinant optimalų iteracijų skaičių, pastebima tendencija, kad skirtumas tarp 50 ir 100 iteracijų, visais eksperimento metu tirtais atvejais yra minimalus (žr. 29 pav.).




**30 pav.** Algoritmų patobulinimo rezultatai

30 pav. grafiškai pavaizduoti OCL apribojimais pagrįsto algoritmo kodo padengimo rodikliai prieš ir po tyrimo metu pasiūlytų patobulinimų. Matome ryškų (10-20 %) pagerėjimą „LT Converter“ programos kodo padengime, tačiau kitų testuojamų programų atveju, pagerėjimas yra minimalus arba jo visai nėra.

Apibendrinant eksperimento rezultatus, galime prieiti tokių išvadų:

1. Algoritmų efektyvumas priklauso nuo programos, kuriai generuojami testai: pavyzdžiui, atsitiktinis generavimas buvo efektyvesnis „LT Converter“ atveju, taikant 1-ą testų iteraciją, tačiau „Hill Climbing“ ryškiai pranoko jį generuojant „CL Tic Tac Toe“ žaidimui, su 10 ir daugiau testų iteracijų;
2. Vietinės paieškos algoritmų naudojimas nesuteikė didelio kodo padengimo pranašumo, lyginant su atsitiktiniu generavimu. Atsižvelgus dar ir į faktą, kad šių algoritmų veikimas lėtesnis, galime daryti išvadą, kad šis patobulinimas pasiteisino tik iš dalies;
3. Patvirtinta tyrimo metu iškelta prielaida, kad siekiant balanso tarp generavimo laiko ir kodo padengimo, užtenka padidinti iteracijų skaičių nuo 10 iki 50, o ne 100: iteracijų skaičiaus didinimas nuo 50 iki 100 nei vieno algoritmo atveju nepadidino kodo padengimo daugiau kaip 8 %, o vidutinis iteracijų skaičiaus dvigubinimo šiame intervale efektas tesiekia 1,21 %;

# 6. IŠVADOS

1. Testų generavimas – viena efektyviausių testavimo automatizavimo priemonių. Testų generavimo srities analizės metu nustatyta, kad egzistuoja daugybė testų generavimo algoritmų, turinčių savų privalumų ir trūkumų. Didžiausias modeliu paremtų algoritmų privalumas – galimybė naudoti modelį kaip testų orakulą;
2. Rinkos analizės metu nustatyta, jog įrankis gali būti aktualus, nes, nepaisant didelės testų generavimo įrankių pasiūlos, nėra OCL apribojimais pagrįstų, nekomercinių produktų;
3. Projekto metu suprojektuoto bei sukurto įrankio architektūra užtikrina aukštą jo išplečiamumą. Tuo buvo įsitikinta atliekant patobulinimus prieš eksperimentinį tyrimą: norint papildyti įrankį nauju testų generavimo algoritmu, užteko pridėti papildomą, jį realizuojančią klasę. Jokie kiti įrankio kodo pakeitimai nebuvo reikalingi;
4. Atlikus sukurto įrankio tyrimą, nustatyta, jog nesudėtingoms, gerai OCL apribojimais aprašytoms programoms, jis gali pasiekti iki 100 % kodo padengimo rodiklį, tačiau didesnėms, sudėtingesnėms programoms OCL apribojimų naudojimas neužtikrina 100 % kodo padengimo ir, tam tikrais atvejais, nedaug skiriasi nuo atsitiktiniu generavimu gautų rezultatų. To priežastys – nepakankamai išsamūs OCL apribojimai arba nepakankamas testų kiekis visam kodui padengti;
5. Atsižvelgiant į tyrimo metu gautus rezultatus, nustatyti ir realizuoti tokie patobulinimai:

5.1. Siekiant pagerinti įrankio rezultatus esant nepakankamiems OCL apribojimams, nuspręsta papildyti įrankio palaikomų algoritmų aibę „Hill Climbing“ ir „Simulated Annealing“ algoritmais;

5.2. Siekiant spręsti nepakankamo testų kiekio problemą, nuspręsta padidinti numatytąjį testų generavimo iteracijų skaičių nuo 10 iki 50;

6. Atlikus eksperimentinį įrankio patobulinimų tyrimą prieita tokių išvadų:

6.1. „Hill Climbing“ ir „Simulated Annealing“ algoritmų įtraukimas į programą nedavė laukiamų rezultatų: kodo padengimo rodikliai ženkliai pagerėjo tik vienos iš keturių eksperimente naudotų programų atveju;

6.2. Numatytojo testų generavimo iteracijų skaičiaus padidinimas iki 50 yra naudingas: vidutinis kodo padengimo didėjimas keliant iteracijų skaičių nuo 10 iki 50 – 8,39 %. Tolesnis iteracijų skaičiaus didinimas nebūtų efektyvus. Eksperimento metu nustatyta, jog didinant iteracijų skaičių nuo 50 iki 100, kodo padengimas išauga tik 1,21 %.

# 7. LITERATŪRA


[1] ALI S., BRIAND L. C., HEMMATI H., PANESAR-WALAWEGE R. K. 2010. A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on*, *36*(6), 742-762. [Žiūrėta 2017-05-03]. Prieiga per: http://ai2-s2-pdfs.s3.amazonaws.com/69c1/46c3ea8af1bf8b1d95ec17bf32d9ca27cca8.pdf

[2] ANAND S., ir kt. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, *86*(8), 1978-2001.[Žiūrėta 2017-05-03]. Prieiga per: http://romisatriawahono.net/lecture/rm/survey/software%20engineering/Software%20Testing/Anand%20-%20Automated%20Software%20Test%20Case%20generation%20-%202013.pdf

[3] APFELBAUM L., DOYLE J. Model based testing. *Software Quality Week Conference*. 1997. [Žiūrėta 2017-04-28]. Prieiga per: http://www.testoptimal.com/ref/Model%20Based%20Testing.pdf

[4] BOUQUET F., GRANDPIERRE C., LEGEARD B., PEUREUX F. 2008. A test generation solution to automate software testing. *Proceedings of the 3rd international workshop on Automation of software test* (pp. 45-48). ACM. [Žiūrėta 2017-05-07]. Prieiga per:

http://s3.amazonaws.com/academia.edu.documents/40206632/A_Test_Generation_Solution_to_Automate_S20151120-5128-litaod.pdf?AWSAccessKeyId=AKIAIWOWYYGZ2Y53UL3A&Expires=1495125237&Signature=BwmuBbcplEzTsGTob7VePiKBiCI%3D&response-content-disposition=inline%3B%20filename%3DA_test_generation_solution_to_automate_s.pdf

[5] CADAR C., DUNBAR D., ENGLER D. R. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *OSDI* (Vol. 8, pp. 209-224).[Žiūrėta 2017-05-07]. Prieiga per: http://static.usenix.org/legacy/events/osdi08/tech/full_papers/cadar/cadar_html/paper.html

[6] CADAR C., SEN K. 2013. Symbolic execution for software testing: three decades later. *Communications of the ACM*, *56*(2), 82-90. [Žiūrėta 2017-05-03]. Prieiga per: https://pdfs.semanticscholar.org/1144/3efe465ad544f478524da6c66c085b16e28b.pdf

[7] CALVAGNA, A., GARGANTINI, A. 2010. A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, *45*(4), 331-358.[Žiūrėta 2017-05-03]. Prieiga per: http://ai2-s2-pdfs.s3.amazonaws.com/6392/b6cd0d1cf4273fb3c2390e5caa4eec132212.pdf

[8] CHEN T. Y., LEUNG H., MAK I. K. 2005. Adaptive random testing. *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making* (pp. 320-329). Springer Berlin Heidelberg. [Žiūrėta 2017-05-03]. Prieiga per: http://www.utdallas.edu/~lxz144130/cs6301-readings/art.pdf

[9] COHEN D. M., DALAL S. R., FREDMAN M. L., PATTON G. C. 1997. The AETG system: An approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on*, *23*(7), 437-444. [Žiūrėta 2017-05-07]. Prieiga per: http://courses.cs.washington.edu/courses/cse590n/11au/AETG.Oct3.2011.590N.pdf

[10] DAGIENĖ V., GRIGAS G., JEVSIKOVA T. 2013. Enciklopedinis kompiuterijos žodynas, [Žiūrėta Žiūrėta 2017-05-07]. Prieiga per: http://ims.mii.lt/EK%C5%BD/

[11] DIAS NETO A. C., SUBRAMANYAN R., VIEIRA M., TRAVASSOS G. H. 2007. A survey on model-based testing approaches: a systematic review. *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and*

*technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007* (pp. 31-36). ACM.[Žiūrėta 2017-05-03]. Prieiga per: https://www.researchgate.net/profile/Guilherme_Travassos/publication/234125573_A_survey_on_model-based_testing_approaches_a_systematic_review/links/00b7d52f408e960e53000000.pdf

[12] FRASER G., ARCURI A. 2011. Evosuite: automatic test suite generation for object-oriented software. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (pp. 416-419). ACM.[Žiūrėta 2017-05-07]. Prieiga per: https://pdfs.semanticscholar.org/216b/98bb3d9221d5f5d261864975612e4d0faaa6.pdf

[13] KING J. C. 1976. Symbolic execution and program testing. *Communications of the ACM*, *19*(7), 385-394. [Žiūrėta 2017-05-03]. Prieiga per: http://www.cs.umd.edu/class/fall2014/cmsc631/papers/king-symbolic-execution.pdf

[14] KUHN R., LEI Y., KACKER R. 2008. Practical combinatorial testing: Beyond pairwise. *IT Professional*, *10*(3), 19-23.[Žiūrėta 2017-05-03]. Prieiga per: https://pdfs.semanticscholar.org/745f/b0519302c6caa4a25536749e78a2700d9969.pdf

[15] MCMINN, P. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, *14*(2), 105-156. [Žiūrėta 2017-05-03]. Prieiga per: http://mcminn.io/publications/j1.pdf

[16] NIE C., LEUNG H. 2011. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, *43*(2), 11.[Žiūrėta 2017-05-03]. Prieiga per: http://romisatriawahono.net/lecture/rm/survey/software%20engineering/Software%20Testing/Nie%20-%20Combinatorial%20Testing%20-%202012.pdf

[17] PACHECO C., ERNST M. D. 2007. Randoop: feedback-directed random testing for Java. *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion* (pp. 815-816). ACM. [Žiūrėta 2017-05-07]. Prieiga per: http://homes.cs.washington.edu/~mernst/pubs/pacheco-randoop-oopsla2007.pdf

[18] PACKEVIČIUS Š., KRIVICKAITĖ G., BARISAS D., JASAITIS R., BLAŽAUSKAS T., Guogis E. 2013. „Test Data Generation for Complex Data Types Using Imprecise Model Constraints and Constraint Solving Techniques“. *Information Technology And Control*, *42*(2), 131-149. [Žiūrėta 2015-11-03]

[19] PELESKA J. 2013. Industrial-Strength Model-Based Testing - State of the Art and Current Challenges. Proc. Eighth Workshop on Model-Based Testing, pp. 3-28 [Žiūrėta 2017-05-07]. Prieiga per: https://arxiv.org/pdf/1303.1006.pdf

[20] PNUELI A. 1977 . The temporal logic of programs. *Foundations of Computer Science, 1977., 18th Annual Symposium on* (pp. 46-57). IEEE.[Žiūrėta 2017-05-07]. Prieiga per: http://fi.ort.edu.uy/innovaportal/file/20124/1/49-pnueli_temporal_logic_of_programs.pdf

[21] PRASETYA I. W. B. 2015. T3i: a tool for generating and querying test suites for Java. *ESEC/FSE 2015, Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (pp. 950-953). ACM.[Žiūrėta 2017-05-07]. Prieiga per: dspace.library.uu.nl/bitstream/handle/1874/321619/950.pdf?sequence=1

[22] SABOR K. K, MOHSENZADEH M. 2013. Adaptive Random Testing Through Dynamic Partitioning by Localization with Restriction and Enlarged Input Domain. Software Engineering & Digital Media Technology, 147-155 .[Žiūrėta 2017-05-07]. Prieiga per: http://link.springer.com/chapter/10.1007/978-3-642-34531-9_16

[23] SAKTI A., PESANT G., GUÉHÉNEUC Y. G. 2015. Instance Generator and Problem Representation to Improve Object Oriented Code Coverage. *Software Engineering, IEEE Transactions on*, *41*(3), 294-313.[Žiūrėta 2017-05-07]. Prieiga per: http://www.ptidej.net/publications/documents/TSE14b.doc.pdf

[24] SEN K., KALASAPUR S., BRUTCH T., GIBBS S. 2013. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (pp. 488-498). ACM. [Žiūrėta 2017-05-07]. Prieiga per: http://people.eecs.berkeley.edu/~ksen/papers/jalangi.pdf

[25] SHARMA, C., SABHARWAL, S., & SIBAL, R. 2014. A survey on software testing techniques using genetic algorithm. *arXiv preprint arXiv:1411.1154*. [Žiūrėta 2017-05-07].

Prieiga per: https://arxiv.org/ftp/arxiv/papers/1411/1411.1154.pdf

[26] UTTING M. The Role of Model-Based Testing. *Verified Software: Theories, Tools, Experiments Volume 4171 of the series Lecture Notes in Computer Science,* 510-517 [Žiūrėta 2017-05-03]. Prieiga per: http://dl.ifip.org/db/conf/vstte/vstte2005/Utting05.pdf

[27] UTTING M., PRETSCHNER A., LEGEARD B. 2006. A taxonomy of model-based testing. [Žiūrėta 2017-05-03]. Prieiga per: http://researchcommons.waikato.ac.nz/bitstream/handle/10289/81/content.pdf%3Bjsessionid%3DF408313CE2F0172865A764A782540858?sequence%3D1

[28] UTTING M., PRETSCHNER A., LEGEARD B. 2012. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, *22*(5), 297-312. [Žiūrėta 2017-05-03]. Prieiga per: http://eprints.qut.edu.au/57853/1/master_pdflatex.pdf

[29] MYERS G. J. „The art of software testing“ Second edition. ISBN 0-471-46912-2 [Žiūrėta 2017-05-07]. Prieiga per: http://ufrsciencestech.u-bourgogne.fr/master1/Qualite-Innovation-PI_archives/QUALITE%20INNOVATION%20M1%20STIC%20V2015/1_Qualit%C3%A9/TD%20AQL/The%20Art%20of%20Software%20Testing%20-%20Second%20Edition.pdf

[30] HOFFMANN, M., JANICZAK, B., MANDRIKOV, E., & FRIEDENHAGEN, M. (2016). Jacoco code coverage tool. Online, 2009 [Žiūrėta 2017-05-07]

## 8. TERMINŲ IR SANTRUMPŲ ŽODYNAS

- SUT – Software under test – testuojama programinė įranga
- Programos vykdymo kelias (angl. execution path) – su vienu įvesties duomenų rinkiniu įvykdytų programos sakinių seka.
- PC – Kelio apribojimas (angl. path constraint) – apribojimų, kuriuos turi tenkinti įvesties duomenys, kad būtų pasirinktas tam tikras konkretus vykdymo kelias, rinkinys
- Simbolinio vykdymo medis (angl. symbolic execution tree) – medžio tipo struktūra atvaizduojant i visus galimus programos vykdymo kelius.
- SBST Search-based software testing – paieška paremtas testavimas
- Kombinatorinis testavimas (angl. Combinatorial testing, sutr. CT) – testavimo metodas, kuris testuoja SUT naudodamas dengiančio masyvo testų rinkinį, kuris ištestuoja visas

reikalingas parametrų reikšmių kombinacijas

- Kortežas (angl. tuple) – baigtinio dydžio surikiuotų elementų seka.
- CA – covering array – dengiantis masyvas
- Baltos dėžės testavimas – testavimas, kai remiamasi išeities kodu
- Juodos dėžės testavimas – testavimo būdas, kai laikoma, kad programa yra juoda dėžė t. y. nežinoma nieko apie vidinį programos įgyvendinimą. Testuojant žinomos tik įvestys ir rezultatai.
- MBT – model based testing – modeliu paremtas testavimas
- Testų orakulas (angl. test oracle) – mechanizmas nusakantis ar testas pavyko (angl. pass) ar nepavyko (angl. fail).
- FSM – finite state machine – baigtinis automatas
- Regresinis testavimas (angl. regression testing) – pakartotinis testavimas siekiant patikrinti, ar pasikeitus daliai kodo senas funkcionalumas išlieka teisingas.
- AETG įvedimo duomenų aprašymo kalba (angl. AETG Input Language) – AETG įrankio naudojama kalba, kuria sudarytą modelį įrankis naudoja kombinatoriniam testavimui
- Saityno paslauga (angl. web service) – Programinės įrangos sistema, suprojektuota įvairių kompiuterių sąveikai tinkle užtikrinti. [10]
- Mutacinis testavimas (angl. mutation testing) – testavimo būdas, kai programoje atliekami nedideli pakeitimai. Pakeista programa vadinama mutantu. Mutacinio testavimo tikslas – patikrinti ar turimi testai gali aptikti mutantus ir jei reikia, papildyti turimą testų rinkinį naujais, galinčiais aptikti tam tikrus mutantus.
- Prieš-sąlyga, priešsąlygis (angl. precondition) – loginis reiškinys, kuris turi būti tiesa prieš tam tikro kodo fragmento vykdymą
- Po-sąlyga, posąlygis (angl. postcondition) – loginis reiškinys, kuris turi būti tiesa po tam tikro kodo fragmento vykdymo
- Hoare triple – trijų narių kortežas aprašantis kaip kodo fragmento vykdymas pakeičia programos būseną. Hoare triple yra tokios formos: {P} C {Q}, kur P – prieš-sąlyga, Q – po-sąlyga, o C – vykdoma komanda.
- LTL – Linear temporal logic – modalinės laiko logikos tipas, kuriuo galima užrašyti formules apie kelių ateitį pvz. sąlygą, kuri galiausiai bus tiesa arba sąlyga, kuri įgis reikšmę „tiesa“, kai kitas faktas taps tiesa. LTL pirmą kartą buvo pasiūlytas formaliam programų verifikavimui A.Pnueli 1977m., straipsnyje „The temporal logic of programs“ [20].

- Concolic testing – programinės įrangos verifikavimo technika, kurios metu atliekamas ir simbolinis (įvesties reikšmes pakeičiant simboliais), ir konkretus (naudojant konkrečias įvesties reikšmes) programos vykdymas.
- UML – Unified Modeling Language – bendros paskirties modeliavimo kalba naudojama programų inžinerijos srityje.
- SysML – Systems Modeling Language – bendros paskirties modeliavimo kalba naudojama sistemų inžinerijoje.
- Simulink – grafinė programavimo aplinka dinaminių sistemų modeliavimui ir analizei.

# 9. PRIEDAI

## 9.1. OCL apribojimai naudoti eksperimente

### 9.1.1. Basic Calculator

```
 package org::ktu::examples

 context BasicCalculator
 inv: self.magicNumber = 42

 context BasicCalculator::add(a:Integer, b:Integer):Integer
 post: result = a+b

 context BasicCalculator::addOne(a:Integer):Integer
 post: result = a+1

 context BasicCalculator::subtract(a:Integer, b:Integer):Integer
 post: result = a-b

 context BasicCalculator::subtractOne(a:Integer):Integer
 post: result = a-1
--
 context BasicCalculator::divide(a:Integer, b:Integer):Real
 post: result = a/b
 pre: b <> 0

 context BasicCalculator::multiply(a:Integer, b:Integer):Integer
 post: result = a*b

context BasicCalculator::setMagicNumber(a:Integer)
pre: self.magicNumber = 42


endpackage
```

### 9.1.2. LT Converter

```
 package org::ktu::examples

 context ConstructLTConverter::getBoolName(x:Boolean):String
```

```
post: result = 'tiesa' or result = 'melas'

endpackage
```

### 9.1.3. Geometry Calculator

```
package org::ktu::examples

context GeometryCalculator::areaOfCircle(radius:Real):Real
pre: radius > 0
post: result = 3.1415926*radius*radius

context GeometryCalculator::circumferenceOfCircle(radius:Real):Real
pre: radius > 0
post: result = 3.1415926*radius*radius

context GeometryCalculator::perimeterOfRectangle(length:Real, width:Real):Real
pre: length > 0
pre: width > 0
post: result = 2*length+2*width

context GeometryCalculator::areaOfRectangle(length:Real, width:Real):Real
pre: length > 0
pre: width > 0
post: result = length*width

context GeometryCalculator::perimeterOfSquare(a:Real):Real
pre: a > 0
post: result = 4*a

context GeometryCalculator::areaOfSquare(a:Real):Real
pre: a > 0
post: result = a*a

context GeometryCalculator::areaOfTriangle(base:Real, height:Real):Real
pre: base > 0
pre: height > 0
post: result = base*height*0.5

context GeometryCalculator::areaOfTrapezoid(base1:Real, base2:Real, height:Real):Real
pre: base1 > 0
pre: base2 > 0
pre: height > 0
post: result = 0.5*(base1+base2)*height

context GeometryCalculator::areaOfParallelogram(base:Real, height:Real):Real
pre: base > 0
pre: height > 0
post: result = base*height

context GeometryCalculator::volumeOfRectangularPrism(length:Real, height:Real,
width:Real):Real
pre: length > 0
pre: height > 0
pre: width > 0
post: result = length*height*width

context GeometryCalculator::surfaceAreaOfRectangularPrism(length:Real, height:Real,
width:Real):Real
```

```
pre: length > 0
pre: height > 0
pre: width > 0
post: result = 2*length*height+2*length*width+2*height*width

context GeometryCalculator::volumeOfGeneralPrism(areaOfBase:Real, height:Real):Real
pre: height > 0
pre: areaOfBase > 0
post: result = areaOfBase*height

context GeometryCalculator::volumeOfCylinder(areaOfBase:Real, height:Real):Real
pre: height > 0
pre: areaOfBase > 0
post: result = areaOfBase*height

context GeometryCalculator::surfaceAreaOfGeneralPrism(areaOfBase:Real, height:Real,
circumference:Real):Real
pre: height > 0
pre: areaOfBase > 0
pre: circumference > 0
post: result = 2*areaOfBase+circumference*height

context GeometryCalculator::volumeOfSquarePyramid(areaOfBase:Real, height:Real):Real
pre: height > 0
pre: areaOfBase > 0
post: result = 0.333333*areaOfBase*height

context GeometryCalculator::surfaceAreaOfSquarePyramid(perimeterOfBase:Real,
slantHeight:Real):Real
pre: perimeterOfBase > 0
pre: slantHeight > 0
post: result = 0.5*perimeterOfBase*slantHeight

context GeometryCalculator::volumeOfRightCircularCone(areaOfBase:Real,
height:Real):Real
pre: height > 0
pre: areaOfBase > 0
post: result = 0.333333*areaOfBase*height

context GeometryCalculator::surfaceAreaOfRightCircularCone(circumference:Real,
slantHeight:Real):Real
pre: circumference > 0
pre: slantHeight > 0
post: result = 0.5*circumference*slantHeight

context GeometryCalculator::volumeOfSphere(radius:Real):Real
pre: radius > 0
post: result = 4.188790204*radius*radius*radius

context GeometryCalculator::surfaceAreaOfSphere(radius:Real):Real
pre: radius > 0
post: result = 12.56637061*radius*radius

endpackage
```

### 9.1.4. CL Tic Tac Toe

```
package org::ktu::examples
```

```
context BadTicTacToe
inv: self.currentPlayer > 0 and self.currentPlayer < 3
inv: self.winner > 0 and self.winner < 3

context BadTicTacToe::reset()
post: self.over = false
post: self.currentPlayer = 1
post: self.winner = 0

context BadTicTacToe::mark(i:Integer, j:Integer, mark:Integer):Boolean
pre: mark <> 0
pre: i < 3 and i >= 0
pre: j < 3 and j >= 0

context BadTicTacToe::makeTurn(i:Integer, j:Integer):Boolean
pre: self.winner = 0
pre: self.over = false

endpackage
```